

FIT2099 Assignment 2

Applied Session 4 Group 8

Updated Design Rationale

Group Members:

Wee Jun Lin (32620861)

Kok Tim Ming (32619138)

Lau Ka-Kiat (32967136)

Overview	4
Design goals	4
UML Notes	4
Requirement 1	5
Brief	6
Environments	6
Enemies	6
Enemy	6
HeavySkeletalSwordsman, SkeletalEnemy, PileOfBones, Revivable	6
Weapons	7
Behaviours	7
AttackBehaviour	7
Actions	8
RemoveActorAction	8
TransformAction	8
ReviveAction	8
SpinAttackAction, AreaAttackAction	8
Requirement 2	9
Brief	10
Economy	10
RuneManager & Rune	10
DropperManager	10
CollectorManager	10
Sellable & Purchasable	11
SellableWeapon	11
PurchasableWeapon	11
Friendlies (Trading)	11
MerchantKale	11
Actions	11
SellAction	12
PurchaseAction	12
DeathAction	12
Req 3	13
Brief	14
FlaskOfCrimsonTears	14
Consumable	14
Charge	14
ResetManager	14
Resettable	15
Actions	15
ConsumeAction	15
ResetAction	15
RestAction	15

SiteOfLostGrace & Game Reset	15
Rune handling	15
DroppedRuneManager	16
Rune	16
DropRuneAction	16
PickUpRuneAction	16
Req 4	17
Brief	18
Archetypes	18
ArchetypeFactory	18
Weapons	19
Quickstep, Unsheathe	19
WeaponModifier	19
Req 5	20
Brief	21
Environments	21
EnemyFactory, WestMapEnemyFactory, EastMapEnemyFactory	21
EnemyEnvironment	21
Enemies	21
Weapons	22
Actions	22
Contribution Log link	23

Overview

Design goals


Our 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.

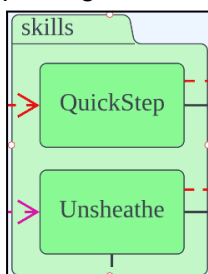
We strive to:



- 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.

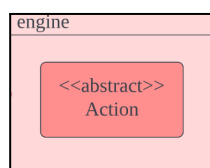
Notes

All full sequence/UML diagrams are attached at the end of this document.

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.



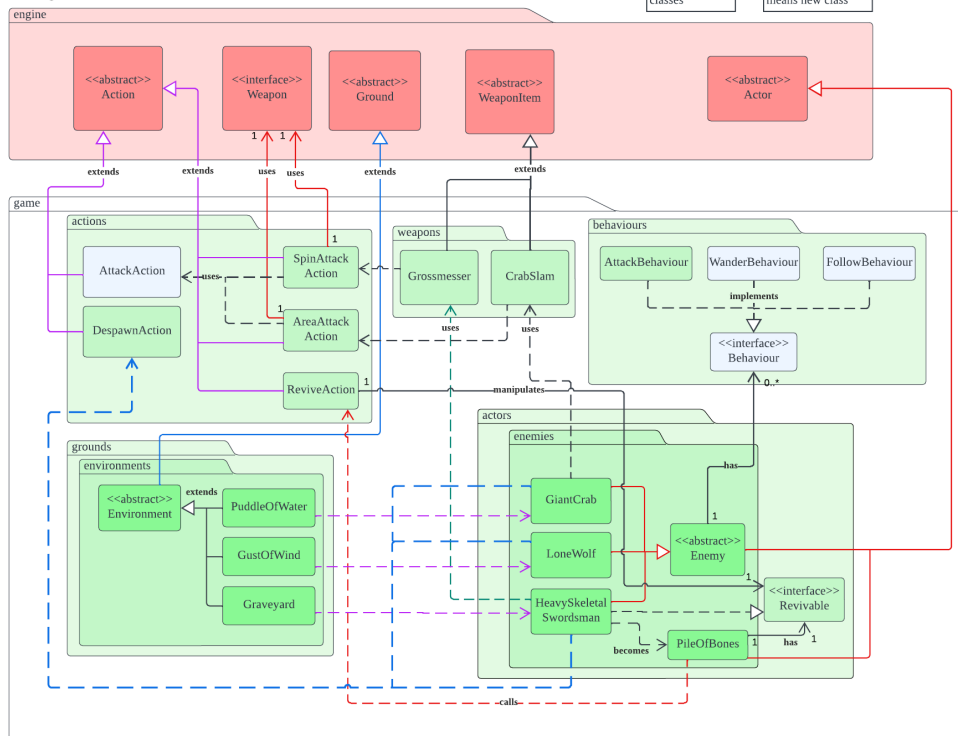
Classes coloured in **light blue**  are classes that will not be extended/modified from the previous requirement, or classes that have been provided. Classes coloured in **red**  are engine classes.



Requirement 1

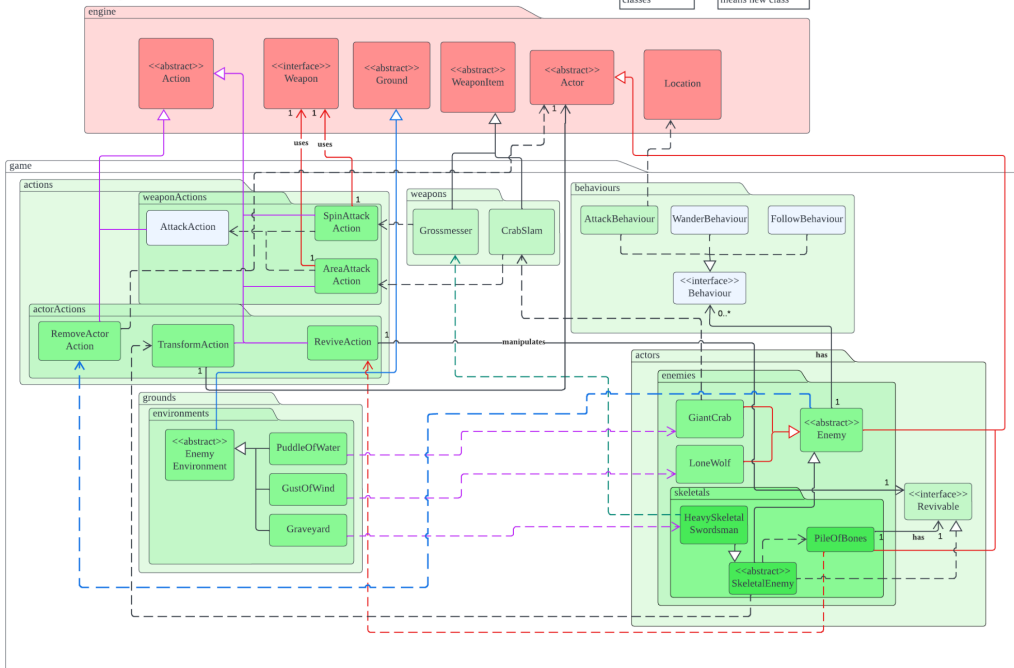
Old Design:

REQ1: Environments & Enemies



New Design:

REQ1: Environments & Enemies



Brief

The diagram represents an updated object-oriented system for requirement 1. It now has 15 new concrete classes, 2 abstract classes and 1 new interface, with all related classes in the game package categorised into respective packages.

Environments

All types of environments extend the *EnemyEnvironment* abstract as they share common methods (DRY).

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

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

Also in the future if there arises the need for different spawning mechanisms (i.e., distinguish by cardinal directions), the Environment make will make use of composition over inheritance by adapting (associate) an abstraction of an *EnemyFactory* class. This approach follows the Abstract Factory method pattern, removing tight coupling between the concrete environments and respective *Enemy* classes, thus increasing maintainability (OCP & SRP).

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, SkeletalEnemy, PileOfBones, Revivable

The *SkeletalEnemy* abstract class is a new abstract class that extends the *Enemy* abstract class, containing the mechanisms for cheating death (don't call *DeathAction*), *transforming* into *PileOfBones* and other similar behaviors. This way, behaviours exclusive to *Skeletal* enemies do not have to be repetitively written (DRY), and also ensures consistency within the code. This implementation adheres to the Liskov Substitution Principle.

It would be ideal to implement *Skeletal* as an interface, but for now, while this could be seen as an issue of over abstraction (not flexible), we find it is justifiable as any basic *Skeletal* enemy (be it a *Flyable* or *Swimmable* creature) to be added in the future would share similar behaviors (DRY). Also in the future if there are needs for implementing stronger *PileOfBones*, the respective *Bone* classes could be passed into the constructor of

SkeletalEnemy (DIP) and *SkeletalEnemy* only needs to be modified once to extend the functionality easily (OCP).

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

PileOfBones will store the *Revivable Actor* through dependency injection (DI) to retain the class instance (association), then call *ReviveAction* the *Revivable Actors* (DIP). This allows future *Revivable Actors* to be revived from *PileOfBones*. Furthermore, *PileOfBones* could also be classified and dissected into a *RespawnPoint* type in the future, improving extensibility. It is also important to note that if in the future *Skeletal* interface is ever considered, the *PileOfBones* could be made to associate with *Skeletal* Actors instead as it is only exclusive to skeletons.

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.

Behavior was made to implement an interface as there is no reason for it to be an abstract class (i.e., no shared attributes, code redundancy, behaviors should only return an Action through *getAction()*). Also, this provides the flexibility for future abstractions, as mentioned in AttackBehavior below.

AttackBehaviour

Attempts to get a random valid target (filtering restricted enums) in surroundings and performs 50/50 attack/special attack. The current implementation of AttackBehavior is very simple, but in the future better algorithms (behavior concrete classes) could be easily implemented based of an AttackBehavior abstract class (LSP, OCP, DRY).

Also, by utilising enums, the *AttackBehavior* could be made to either attack the included enums, or attack the excluded enums with an overload constructor (OCP), proving that it is easily extensible.

Actions

The updated UML diagram now contains 3 new Action classes: *Revive Action*, *RemoveActorAction* and *TransformAction*. These classes extend the abstract *Action* class to enforce common methods, allowing for polymorphism through abstractions (DIP).

RemoveActorAction

DespawnAction was renamed to *RemoveActorAction* because *DespawnAction* only removes the actor from the map. *RemoveActorAction* has a dependency on the Actor classes, where the execution of this action would remove the actor from the map, and looking into the future (req 2 and req 3), unregister the actor from the *RuneManager* and *ResetManager* classes properly to prevent memory leaks. *RemoveActorAction* was chosen to enforce the SRP, separating responsibilities (Actor removal and Unregistration) from Actor, *DeathAction*, and future implementations, improving extensibility and adhering to OCP.

TransformAction

TransformAction replaces Actor A with Actor B. Although it is not desirable to have two concrete classes under the same superclass depending on the other, *TransformAction* was made to handle the Connascence of Execution whenever an Actor needs to replace itself with another (ISP). While it does increase maintainability in that sense, it introduces some degree of coupling between the class and *RemoveActorAction*.

ReviveAction

ReviveAction will have an association with *Revivable* Interface. This allows *ReviveAction* to invoke the *Revivable* interface's method when executed, adhering to SRP. *ReviveAction* can also be used by other *Revivable* actors in the future, allowing for flexible extensions of custom revival features alongside other extensions (ISP).

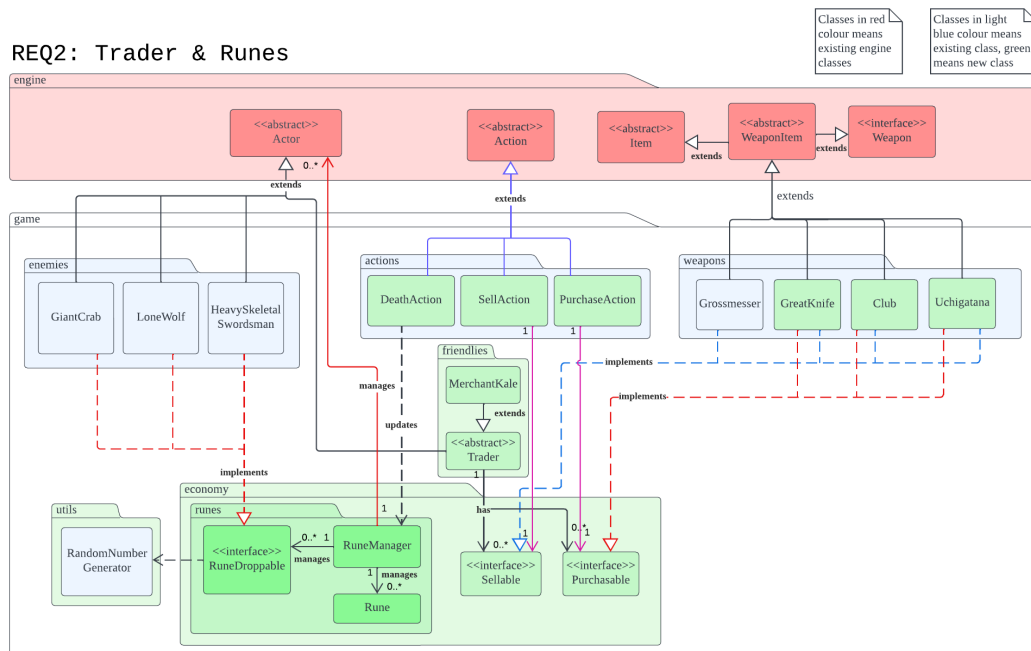
SpinAttackAction, AreaAttackAction

Basic *SpinAttackAction* and *AreaAttackAction* will depend on *AttackAction* as they are assumed to deal the same damage with the same accuracy (DRY). While it is undesirable to have two concrete classes under the same superclass depend on the other, they were not made to inherit *AttackAction* as 1) they do not have identical behaviors, 2) to avoid over abstraction. While over-abstracting could also give rise to maintainability issues, it is important to strike a balance between abstraction and maintainability; Under foreseeable circumstances, *SpinAttackAction* and *AreaAttackAction* could be made to inherit *AttackAction* to achieve optimal flexibility and extensibility.

Requirement 2

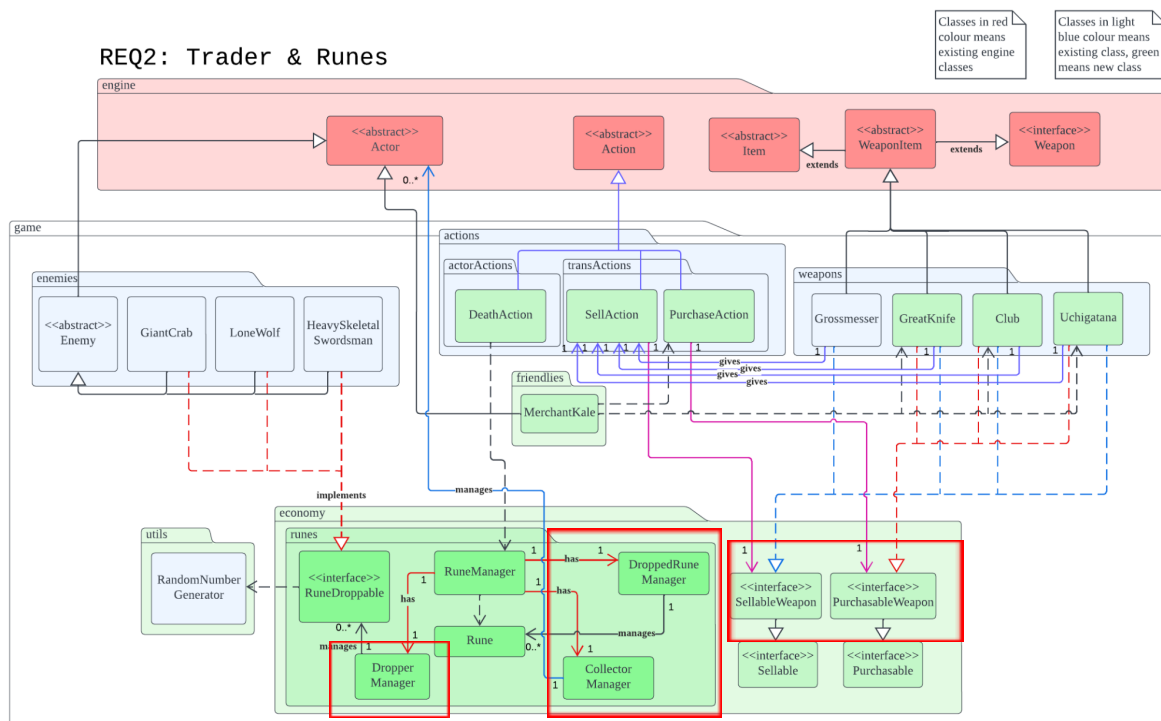
Old Design:

REQ2: Trader & Runes



New Design:

REQ2: Trader & Runes



Brief

The new diagram represents an updated UML class diagram for requirement 2. It has 3 new concrete classes, 2 new interface classes. as well as the removal of the abstract trader class.

Economy

RuneManager & Rune

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

The *RuneManager* class now has an association with 2 other manager classes through composition (OCP). It originally comprised all of the Rune-handling responsibilities, but was segregated into 2 parts, *DropperManager* and *CollectorManager* to avoid it becoming a “god class”. The segregation breaks down the responsibilities tied to the *RuneManager* class in relation to operations that we have to perform on the *Runes* class, as the implementation of these other 2 managers allows for us to reduce the amount of responsibilities that the *RuneManager* has, enforcing the Single Responsibility Principle.

The *RuneManager* class follows a Registry & Singleton pattern, hence will store and handle all *Rune* related logic for registered classes. This approach provides a centralised control among *Runes*, reduced coupling between other classes, allows for code reusability and simplifies object retrieval.

While the Registry & Singleton pattern makes things relatively easy to implement, the approach could come across as controversial as it causes tight coupling along with encapsulation problems between *RuneManager* and other dependent objects. The approach could promote the violation of SRP as the application extends (god class) if good coding practices are not exercised in the future.

It is important to note that the approach takes into account that any actors, not just the *Player*, can collect/drop runes. Given the current game engine provided along with the note, the Registry & Singleton approach was the only way to implement the economy functionality without severe performance/downcasting compromises.

DropperManager

A new *DropperManager* class is created to catalogue and handle the removal (or dropping) of the classes that inherit the *RuneDroppable* interface. When a *RuneDroppable* class dies, the rune value that they were holding would be given to the *Player*, and the class would be unregistered (removed) from the *DropperManager*'s list. This enforces SRP as the *Rune*-giving functionalities are diverted to the *DropperManager* instead of the *RuneManager*.

CollectorManager

A new *CollectorManager* class is also created to handle the collection of runes by Actors that can get runes from kills. This enforces SRP as the *RuneManager* will not need to do these responsibilities and just create an instance of *CollectorManager*.

Sellable & Purchasable

The 2 new interface classes: `SellableWeapon` and `PurchasableWeapon` extend from the `Sellable` and `Purchasable` interface respectively. This is done in order to enforce the Single Responsibility Principle (SRP) as well as the Interface Segregation Principle (ISP), as in the future we may have different item types (e.g. `Armour`), and therefore the return and parameter values required when performing a `PurchaseAction` or `SellAction` may differ, and we want to ensure that every module has a responsibility to only a single part of the program's functionality, and that the interface is not solely responsible for all possible item types. Having these be separate interfaces also ensures that we do not violate the Liskov Substitution Principle (LSP), as this will prevent us from having empty implementations of methods in subclasses (for example a different item type, say `Armour`, would have no use for the `getWeapon` method.)

In the future, new sellables / purchasables can be created and implement the `Sellable` / `Purchasable` interface easily (OCP, LSP). These interfaces can either be used as a contract, or to serve common default methods such as (*`removeItemFromInventory()`*), adhering to the DRY principle.

SellableWeapon

The `SellableWeapon` interface has all the functionality of the `Sellable` interface (implements), but with 1 overridden default method *`sell()`*. This is done in order to ensure consistency across all the `WeaponItem` classes that inherit the `SellableWeapon` interface (ISP, LSP). In the future, new sellable weapons can be added easily by implementing the `SellableWeapon` interface (OCP).

PurchasableWeapon

The `Purchasable` interface has all the functionality of the `Purchasable` interface, but with 2 overridden default methods *`purchase()`*. This is done in order to ensure consistency across all the `WeaponItem` classes that inherit the `PurchasableWeapon` interface (ISP, LSP). In the future, new purchasable weapons can be added easily by implementing the `SellableWeapon` interface (OCP).

Friendlies (Trading)

MerchantKale

MerchantKale will depend on *Purchasable WeaponItems*, with *PurchaseAction* to give the purchasing functionality in *allowableActions()*.

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

Stores the *Sellable* and calls *sell()* when executed, adhering to SRP and ISP in implementing classes. Any future *Sellable* instance (e.g., *SellableArmour*) can be also called by *SellAction* (OCP).

Our implementation involves the *Sellable* instance itself returning/removing the *SellAction* if (or not) there is a trader nearby. Although this violates the DRY principle, it avoids the need for downcasting and over-abstraction. Since *WeaponItem* (the *Sellable* implementer) already has several different abstractions, it was considered not ideal to wrap it in another abstract class just for the sole purpose of the selling feature and reducing code redundancy. Thus, the violation of the DRY principle in this case, was a necessary drawback for code maintainability and extensibility.

PurchaseAction

Stores the *Purchasable* and calls *purchase()* when executed, adhering to SRP and ISP in implementing-classes. Any future *Purchasable* instance (e.g., *PurchasableArmour*) can be also called by *PurchaseAction* (OCP).

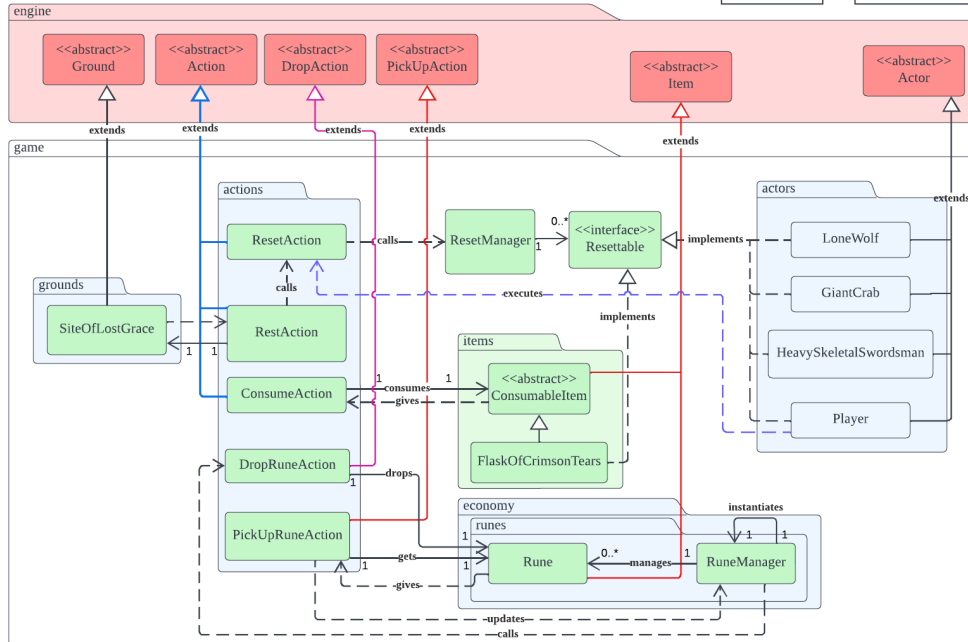
DeathAction

DeathAction depends on the *RuneManager* class, transferring *Runes* to killer from a registered *RuneDroppable* in *DropperManager*.

Requirement 3

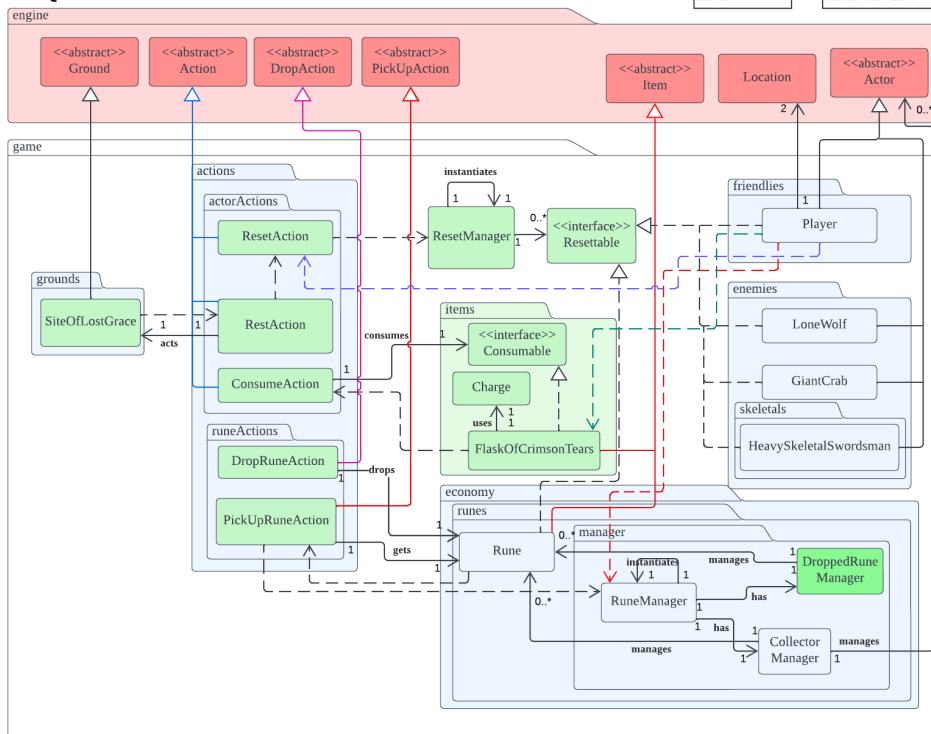
Old Design

REQ3: Grace & Game Reset



New Design

REQ3: Grace & Game Reset



Brief

The new design diagram represents a UML class diagram for requirement 3, updated to suit our current implementation. It has 2 new concrete classes, and all classes shown are in the game package, and organised into their appropriate subpackages. The rune class also now implements *Resettable* as based on the feedback.

FlaskOfCrimsonTears

A new class that inherits from the abstract *Item* class and the *Consumable* interface. The revamped implementation no longer relies on keeping track of the number of uses the item has using attributes, but rather the *Charge* class. This allows for us to adhere to the Single Responsibility Principle (SRP) through composition, as the item no longer has to handle charges, diverting them to the *Charge* class (OCP).

Consumable

An interface class implemented by classes that can be consumed, forming a contract for common methods like *consume()*. The use of polymorphism also ensures that the Open-Closed Principle is not violated as no other classes are needed to be modified.

In the future, there might be any game objects that can consume one another and can also have unique consumption effects, which why *Consumable* was implemented as an interface to open up the doors for future extensions (OCP, LSP, ISP). One downside of implementing *Consumable* as an interface is that the implementing-classes need to define the consume constraints by themselves (charges, removal, conditions etc). However, this provides flexibility for defining all sorts of consumption constraints/mechanisms (OCP).

Charge

In the previous design, all methods and attributes related to the item's number of charges were within the *ConsumableItem* class. However, this design is very flawed and makes it harder to extend (violates OCP) in the future if new types of consumable items are introduced (i.e., consumable items that don't use charge). Therefore, a new concrete class *Charge* was made.

The *Charge* class will handle methods like incrementing/decrementing and checking for the *FlaskOfCrimsonTears*. If more items are added in the future, they can also use the *Charge* class through composition. This new pluggable class encourages OCP, DRY and ISP in attached classes.

ResetManager

Similar to *RuneManager*, *ResetManager* follows the Registry & Singleton pattern, meaning that it will store and handle all Reset related logic for registered *Resettable* classes. This approach provides a centralised control among *Resettables* (OCP).

While the Registry & Singleton pattern makes things relatively easy to implement, the approach could come across as controversial as it causes tight coupling between *ResetManager* and other dependent objects. The approach could promote the violation of

SRP as the application extends (god class) if good coding practices are not exercised in the future.

Though in the case for *ResetManager*, it is justifiable as it provides a centralised interface to execute timely resets (only in this game's turn based context) on all registered *Resettables* when needed, outweighing the potential drawbacks of tight coupling and encapsulation issues. This approach also allows the game to remain consistent and predictable even if the number of *Resettable* object grows as the game progresses.

Resettable

An interface that indicates any game object that can be resetted (ISP). Enforces the *reset()* method to be implemented, which will be called by *ResetAction()* (DIP & LSP). This approach allows implementing-classes to define custom reset behaviors, improving flexibility.

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 *Consumable's consume()* method, utilising polymorphism (DIP) for this feature extension.

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

ResetAction

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

RestAction

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

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.

In the future (A3), *SiteOfLostGrace* could be made to provide *TravelAction* in the future to handle travelling between sites across different maps (OCP).

Rune handling

As a side note, our implementation takes into account that any game object, not just the player, can drop a Rune on the ground.

DroppedRuneManager

A new `DroppedRuneManager` class is created to handle the dropping (Item) of any collector's Runes upon their death. This enforces SRP as the responsibilities are diverted from `RuneManager` toward the `DroppedRuneManager`. The `DroppedRuneManager` class keeps track of any Rune (Item) that are dropped on the ground, allowing for future extensions such as dropped Rune from chests, Rune-holding enemies, random Rune drops etc.

Rune

While it is counter-intuitive to make the `Rune` class implement the `Resettable` interface (ISP) given the `DroppedRuneManager`, it is justifiable to do so as they should despawn when required to (through resets). Also, the `Resettable` interface opens up opportunities for future behaviors when Runes reset.

DropRuneAction

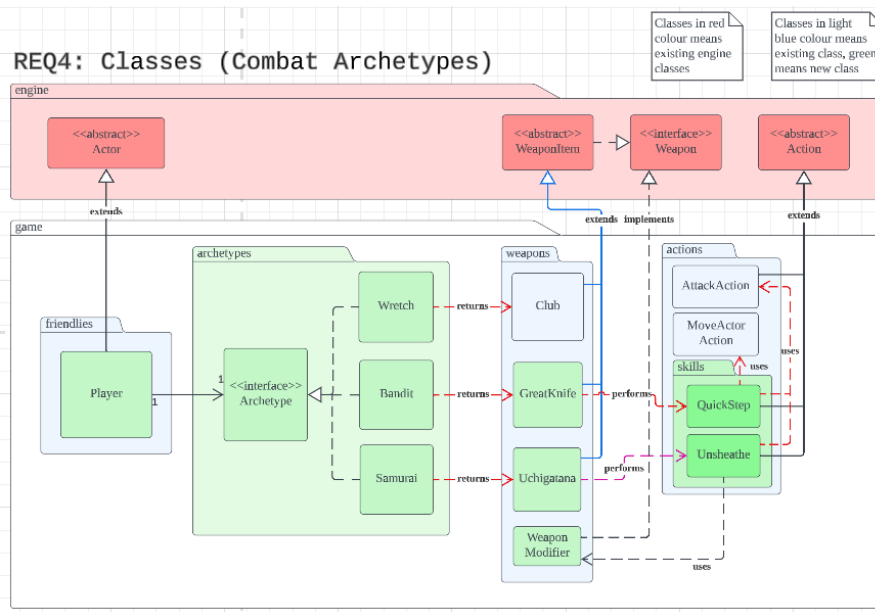
DropRuneAction extends *DropAction* as they share similar behaviors (DRY & LSP). *DropRuneAction* obtains the actor's current location and adds a *Rune* item to the location. In the case for Actor dropping Runes, *DropRuneAction* is invoked by the *dropRuneItem()* method in the *RuneManager* class when the Actor is killed. This separation of responsibility adheres to the Single Responsibility Principle (SRP) and Open-Closed Principle (OCP), as only the *DropRuneAction* would be responsible for adding the *Rune* item to the gamemap, and that *DropRuneAction* allows for any future extensions regarding Rune dropping.

PickUpRuneAction

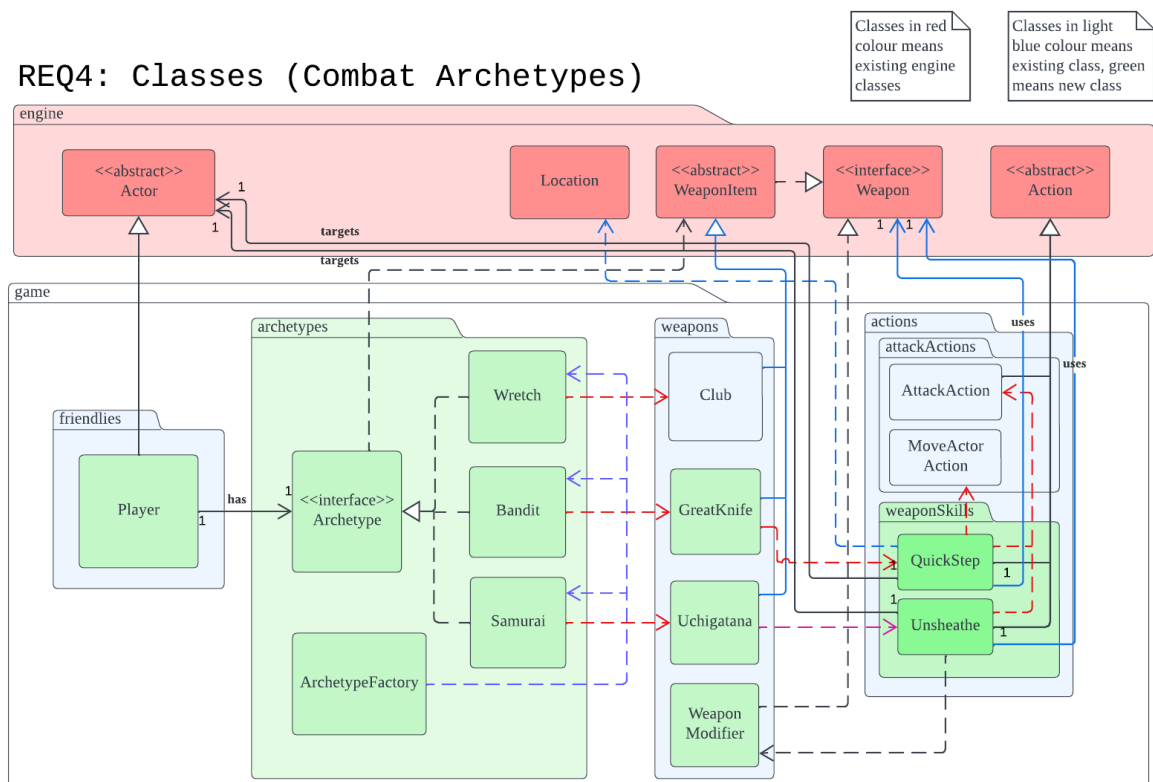
PickUpRuneAction extends *PickUpAction* as they share similar behaviors (DRY & LSP). This abstraction also allows for polymorphism in the *Rune* class. The *PickUpRuneAction* when executed allows for an *Actor*, one that is registered in the *CollectorManager*, to pick up a *Rune* at any given location. The *PickUpRuneAction* invokes the *pickUpRuneItem()* method in the *RuneManager* class, allowing them to pick up the runes that are on the ground. This separation of responsibility adheres to the Single Responsibility Principle (SRP), as only the *PickUpRuneAction* would be responsible for updating the *Actor's Rune* value when picking up a *Rune* item on the gamemap. In the future, *PickUpRuneAction* could also be used for picking up any sort of Runes such as Rune from chests, Rune-holding enemies, random Rune drops (OCP).

Requirement 4

Old Design:



New Design:



Brief

The bottom diagram represents the updated UML class diagram for requirement 4. The updated design includes a new concrete factory class, *ArchetypeFactory*. The *weaponSkills*, *QuickStep* and *Unsheathe* have updated relationships with the engine classes based on the feedback.

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 it will retrieve corresponding attributes and weaponItem from the Archetype through composition, promoting adherence to the SOLID principles, especially DIP.

In the future, new archetypes can be added easily through the Archetype interface. Archetype abilities could also be extended and implemented easily through polymorphism (OCP).

While it is modular to include Archetype as an interface for now, it can only define a contract for implementations. If the need for complex features that require common behaviour arises in the future (e.g., complex talent trees), Archetype should be converted into an abstract class to provide common implementation details, and the subclasses should implement other contract-bounded features through one or more interfaces.

ArchetypeFactory

In our previous design the creation of the *Player's Archetypes* were done within the game's *Application* class. However, this is not feasible as the *Application* class will quickly become bloated and hard to maintain (violation of OCP).

The *ArchetypeFactory* class has dependencies on the individual archetypes. This will improve maintainability by centralising the creation archetype abstractions, adhering to SRP by removing the archetype creation responsibility (coupling) from the *Application* class. Though it is worth noting that the design gets slightly more complex due to the introduction of the Factory method pattern, having the *ArchetypeFactory* class also promotes OCP, acting as an interface to easily add in more archetypes in the future without modifying the existing code.

Weapons

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. While both can be made to extend *AttackAction*, dependency was considered over inheritance to avoid over abstraction. While avoiding over-abstraction can promote maintainability, it is important to strike a balance between abstraction and maintainability; In probable future circumstances, the *weaponSkills* could be made to inherit *AttackAction* to ensure optimal flexibility and extensibility.

For *Unsheathe*, *WeaponModifier* is used as a substitute to adhere to OCP.

For *Quickstep*, *MoveActorAction* is required for moving users (hence the dependency).

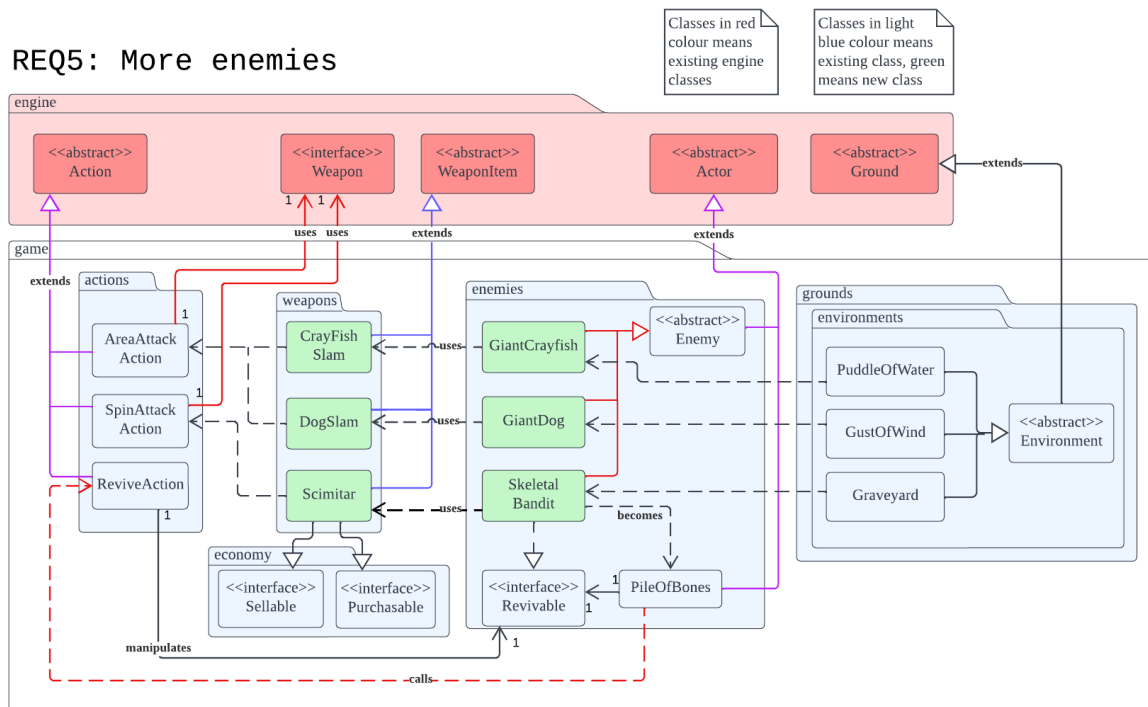
WeaponModifier

This concrete class implements *Weapon* interface to act as a substitute for *Weapons* (any in the future) that temporarily require damage/accuracy modification, such as *Uchigatana's Unsheathe*. Any damage/accuracy modification responsibilities will be diverted to *WeaponModifier* than implementing into the weapon/skills itself, hence increasing maintainability (OCP & SRP).

Requirement 5

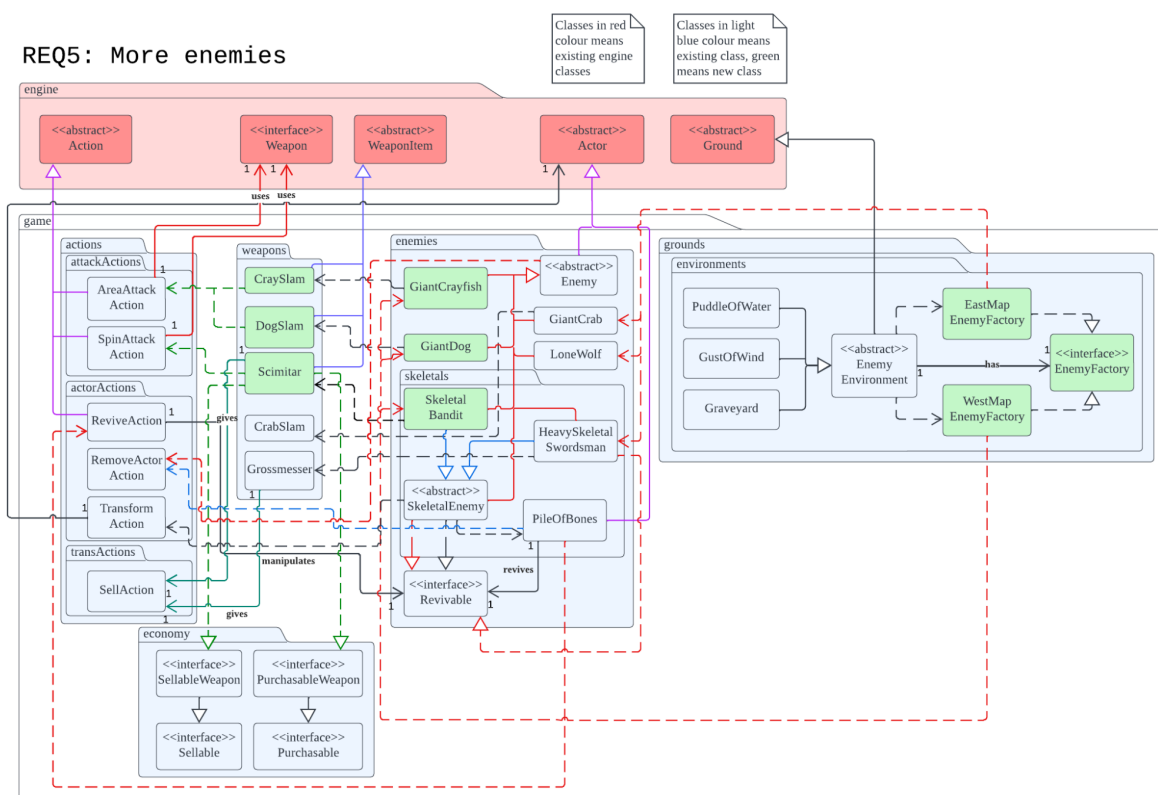
Old Design

REQ5: More enemies



New Design

REQ5: More enemies



Brief

The updated UML class diagram for requirement 5 extends requirement 1: it includes a new interface *EnemyFactory*, which is implemented by the concrete classes *WestMapEnemyFactory* and *EastMapEnemyFactory*. The updated class diagram also includes all the enemies and weapons from REQ1 and shows the relationships between them.

Environments

EnemyFactory, WestMapEnemyFactory, EastMapEnemyFactory

Previously, *Enemies* were created and spawned within the *EnemyEnvironment* subclasses itself. However in requirement 5, it was found that the design is flawed in several ways as these classes became bloated (not adhering to SRP) and not are easily extensible (violation of OCP).

The updated design includes two new factory classes (*EastMapEnemyFactory*, *WestMapEnemyFactory*) and one abstract factory class (*EnemyFactory*) that handles the creation of *Enemy* instances in either east or west parts of the map. This is inline with SRP and ISP as it abstracts/removes the spawning responsibility from the *EnemyEnvironment*.

By implementing the spawning of enemies with Factory classes, the design will adhere to the Open-Close Principle (OCP) as we can add more spawnable enemies in the future easily without modifying the existing code. While Abstract Factory and Factory method pattern will make the codebase more extensible and easier to maintain in the future, there will be inevitable design complexity tradeoffs.

EnemyEnvironment

In this requirement, the *EnemyEnvironment* is dependent on *East/WestMapEnemyFactory* as it uses its methods to spawn the enemies based on the location. The *EnemyEnvironment* is also associated with the *EnemyFactory* class as it creates an instance of its abstraction through polymorphism (DIP) in order to singleton instances of it to spawn enemies independently of other instances.

See also: [REQ1 Environments](#).

Enemies

New enemies, *GiantCrayfish*, *GiantDog*, *SkeletalBandit* are added as part of this requirement. These enemies extend from the abstract *Enemy* class, while *SkeletalBandit* extends from the abstract *SkeletalEnemy* class. No modifications to the existing code were made to add these new enemies (OCP, DRY, LSP).

See also: [REQ1 Enemies](#).

Weapons

DogSlam and *CraySlam* extends *WeaponItem* to have *AreaAttackAction* as a skill (OCP). This allows wielders to have access to the skill only 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 *WeaponItems* could be made undroppable. Scimitar on the other hand, is similar to Grossmesser, and could be added without modifying the source code (OCP).


See also: [REQ1 Weapons](#).

Actions

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

See also: [REQ1 Actions](#).

Contribution Log link

 FIT2099 MY (THU 10 AM) Assignments' Contribution Logs

https://docs.google.com/spreadsheets/d/1c-rrjRwR8Huk-2wwv6Zpmb_kAb5kch_zlCful5oIHS/edit#gid=0, under the Assignment 2 tab.