Design Rationale

FIT 2099 - Assignment 3

Lab1Team3

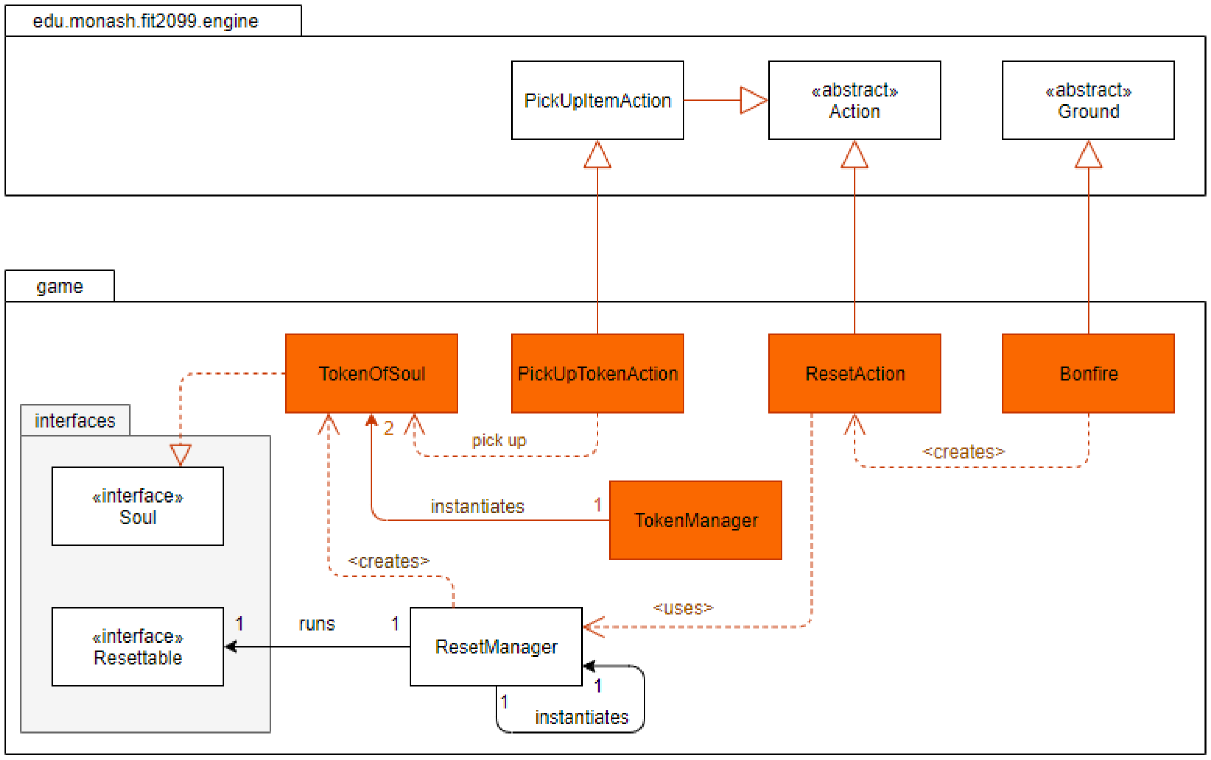
17th October, 2021

** Coloured blocks in UMLs indicates a newly created class**

Table of Content

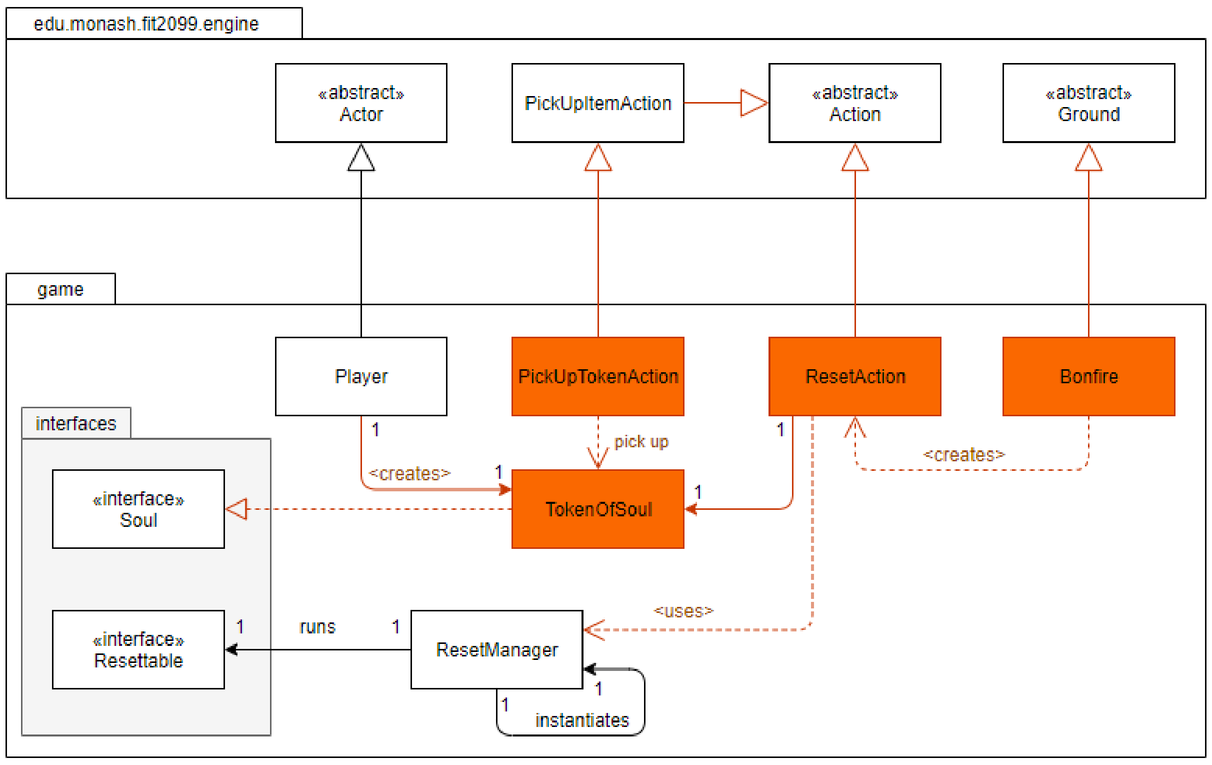
Token of Souls

Our team has made some changes on top of our previous submission regarding the implementation of Token of Souls.



Previous Design

The Token of Souls were previously created and managed by **ResetAction** class and **TokenManager** Class. The **TokenManager** class was implemented as a **singleton** class, which is not really a good practice in terms of **transparency**, more details regarding singleton class are described in Reset Manager. Additionally, creating a class to manage 2 TokenOfSouls instances is inefficient, hence, our team decided to **redesign** it to improve the code performance.



New Design

The **TokenManager** class is **removed** and its functionality is substituted by **Dependency Injection**. A new attribute of type TokenOfSouls is introduced in the Player class, representing the last token that the player has dropped. As such, the tokens are managed within the player class, indicating that the Player class is able to **control** the tokens without rendering TokenManager class.

Dependency Injection is also applied to the ResetAction class. The ResetAction class now reads an extra parameter, representing the token to be dropped. This technique is also known as **Constructor Injection**, where the required dependency is passed into the class by specifying them as a parameter in the constructor.

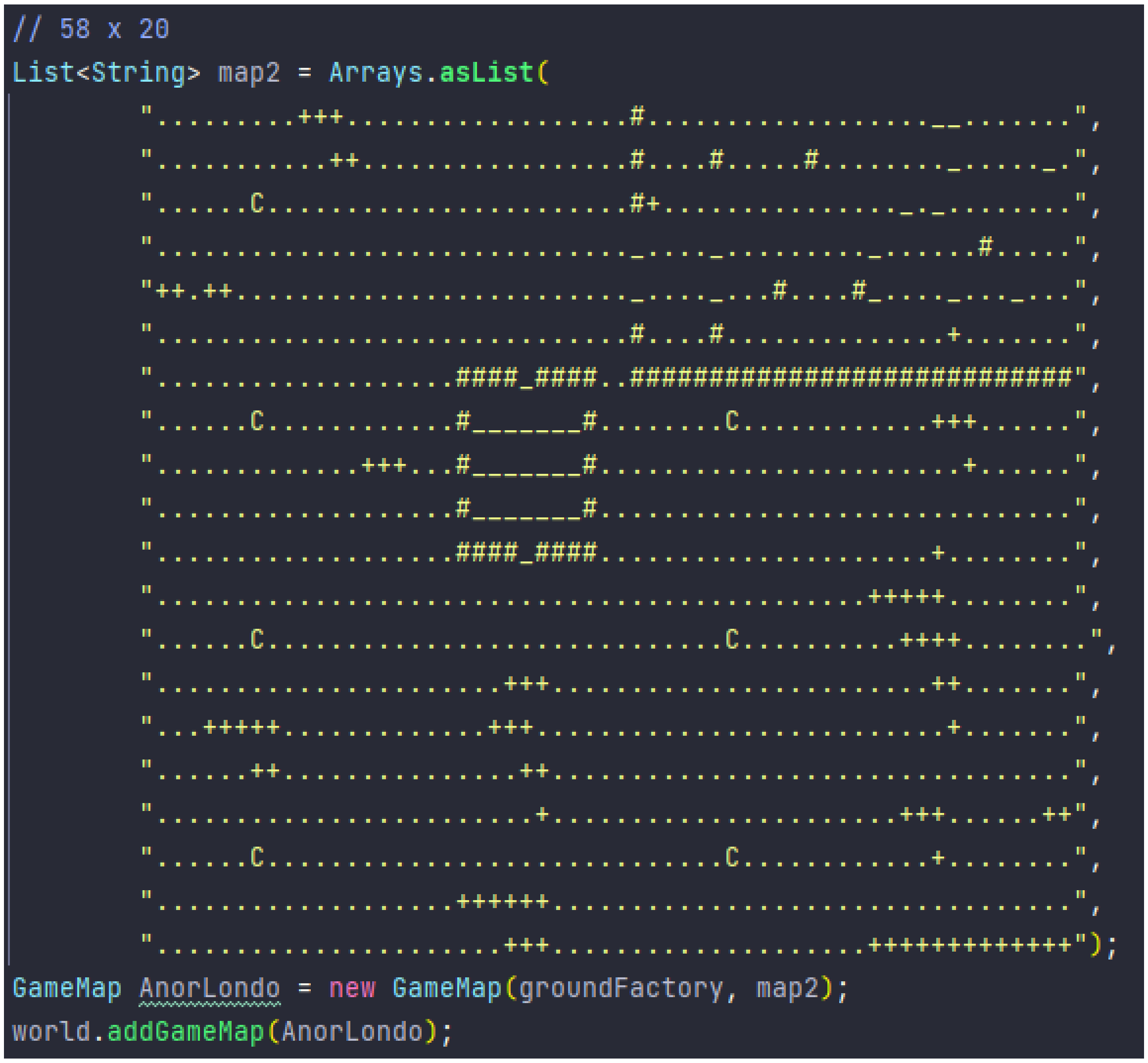
In summary, **Player ---<<create>>---> TokenOfSouls** and **ResetAction ---<<read>>---> TokenOfSouls**.

As a result, the code is now **loosely coupled** and has **high reusability** compared to the previous design.

Requirement 1 : New Map & Fog Door

New Map

A new map of size 58*20 is created in the **Application** class and is given the name "**AnorLondo**".



```java
// 58 x 20
List<String> map2 = Arrays.asList(
        ".........+++..................#..................__.......",
        "...........++.................#....#.....#........_....._.",
        "......C.......................#+..............._._........",
        ".............................._...._........._......#.....",
        "++.++........................._...._...#....#_...._..._...",
        "..............................#....#..............+.......",
        "...................####_####..############################",
        "......C............#_______#........C............+++......",
        ".............+++...#_______#.......................+......",
        "...................#_______#..............................",
        "...................####_####.....................+........",
        ".............................................+++++........",
        "......C.............................C..........++++........",
        "......................+++........................++.......",
        "...+++++.............+++..........................+.......",
        "......++...............++.................................",
        "........................+......................+++......++",
        "......C.............................C............+........",
        "...................++++++.................................",
        "......................+++....................+++++++++++++");
GameMap AnorLondo = new GameMap(groundFactory, map2);
world.addGameMap(AnorLondo);
```

Fog Door

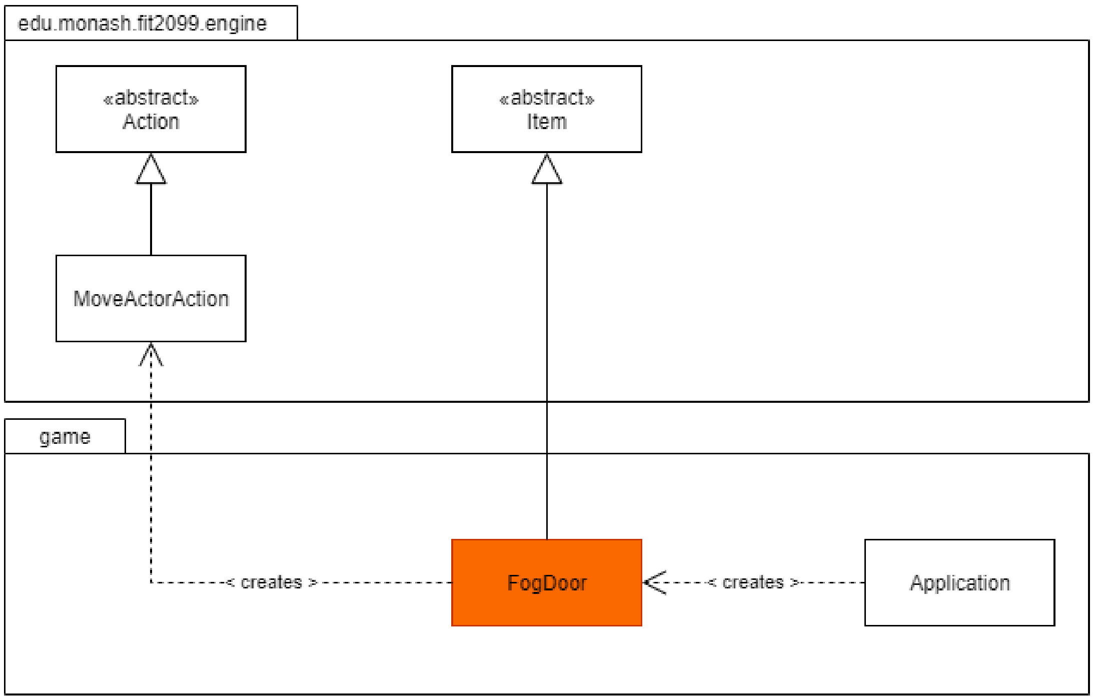
A new class **FogDoor** is created to represent the fog door in the game. **FogDoor extends** the **Item** class, which allows it to **inherit** the behaviour of Item class.

The FogDoor utilises the **MoveActorAction** class to move the player from one map to another map. The exact location of the destinations are defined in the Application class, which will be the location of the other fog door.

In summary, **Application---<<create>>---> FogDoor** and **FogDoor ---<<create>>---> MoveActorAction**.

Why Item but not Ground?

The main reason behind implementing fog door as a **subclass** of **Item** instead of Ground is that the Ground class allows the player to interact with the ground without directly standing on it. Contraversaly, the Item class only allows the player to interact with it while they both are on the same tile.



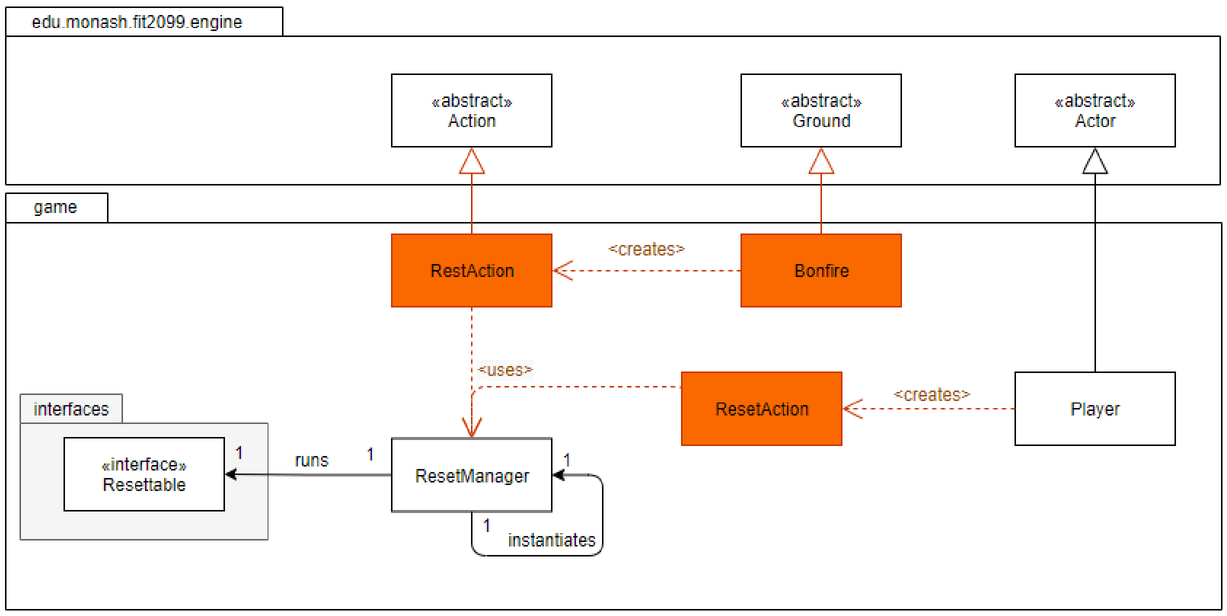
Requirement 2 : New Bonfire

Rest & Reset

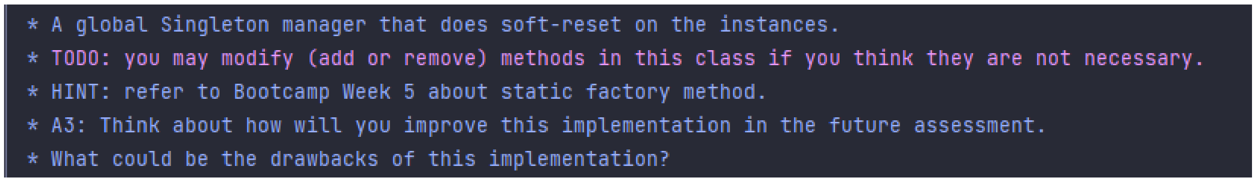
According to the previous design in assignment 2, the “**Rest**” and “**Reset**” features both utilise the **ResetAction** class to perform the operations. In assignment 3, we are to separate them into 2 different classes, namely **RestAction** and **ResetAction** in order to follow the **Single Responsibility Principle** (SRP).

By such, we ensure that both classes serve **one and only one responsibility**, which is either rest or reset. Furthermore, **low coupling and high cohesion** are also achieved.

In summary, **Bonfire ---<<create>>---> RestAction** and **Player ---<<create>>---> ResetAction**.



Reset Manager



```
* A global Singleton manager that does soft-reset on the instances.
* TODO: you may modify (add or remove) methods in this class if you think they are not necessary.
* HINT: refer to Bootcamp Week 5 about static factory method.
* A3: Think about how will you improve this implementation in the future assessment.
* What could be the drawbacks of this implementation?
```

Answering the question mentioned in the class header of the ResetManager class, from our perspective, singleton is the best choice to represent the ResetManager at the moment.

Singleton allows us to conveniently control object creation while hiding the dependencies. It is easy to maintain as it provides a single point access to the particular instance. Anyhow, there are few important disadvantages of such a design pattern.

The drawbacks of singleton pattern are

- Makes unit test harder
- The classes are tightly coupled
- Sacrifices transparency [1]

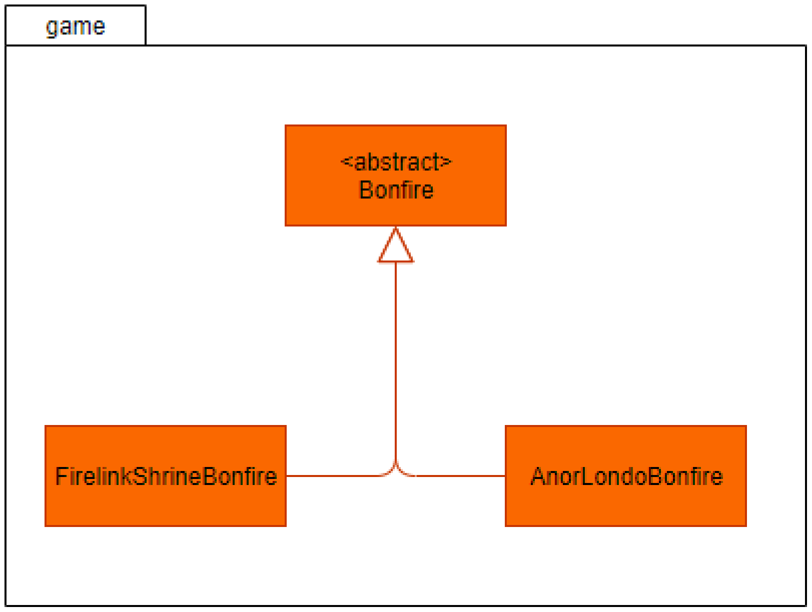
A way to improve this implementation is to **replace Singleton** with **Dependency Injection.** As ResetManager could have many Resettable instances at the same time, a singleton would be more suitable in such a case for the seek of simplicity.

New Bonfire

The original **Bonfire** class is now converted into an **abstract class**, it has 2 **subclasses**, which are **FirelinkShrineBonfire** and **AnorLondoBonfire**. Each of the subclasses represents a different bonfire in the game.

Such design adheres to the **Liskov Substitution Principle** (LSP) and **Dependency Inversion Principle** (DIP).

- The program are always able to use an instance of subclass when the code is expecting an instance of the parent class
- High level component are not depending on low level component, meanwhile they are both depending on abstractions
- Abstraction layer limits the amount of effort to modify the subclasses



Activating a bonfire

A bonfire can be activated through a new class named **ActivateBonfireAction**. This class **extends** the **Action** class.

A “NOT_ACTIVATED” status is bound to a bonfire if it has not been activated yet, the player has only one available action at this time which is “lights up the bonfire”. An ActivateBonfireAction instance is created whenever the player selects to perform this action. The “NOT_ACTIVATED” status will then be removed upon the execution of this action.

This implementation follows the **Single Responsibility Principle**, where each class should have one and only one responsibility.

In summary, **Bonfire ---<<create>>---> ActivateBonfireAction**.

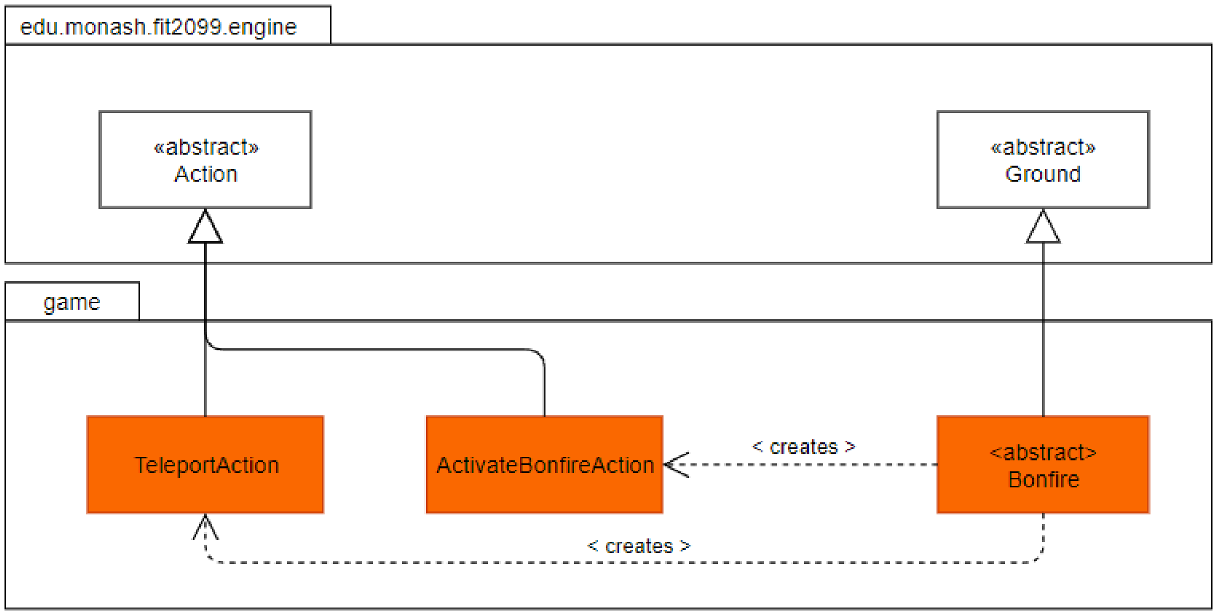
Teleport

A player can perform teleportation through a new class named **TeleportAction**. This class **extends** the **Action** class.

A **static attribute** is initiated in the Bonfire abstract class to **store** all the available **bonfires** in the game. By such, each bonfire is able to perform teleportation from its current position to other bonfires. Such action is performed by TeleportAction. Notice that the constructor of TeleportAction reads only 2 parameters, which are the location of destination and the string to be displayed upon teleporting. This design **lowers the coupling** between Bonfire class and TeleportAction class, while also enhancing the **encapsulation** within Bonfire class.

This implementation follows the **Single Responsibility Principle**, where each class should have one and only one responsibility.

In summary, **Bonfire ---<<create>>---> TeleportAction**.



Keep Track of the last Bonfire

In order to keep track of the last bonfire that the player has interacted with, a new **Enum** is introduced. The Enum named **LastBonfire** stores the status of each bonfire. The player is expected to carry one status (Capability) from LastBonfire at a time, representing the last bonfire it has interacted with. The status is updated through **RestAction** and **ActivateBonfireAction**. While the player dies, the status will return a location instance to the playTurn method in the player class, indicating the expected respawn location of the player. This location instance is then passed to the **ResetAction**, allowing the ResetAction to respawn the player on the given position.

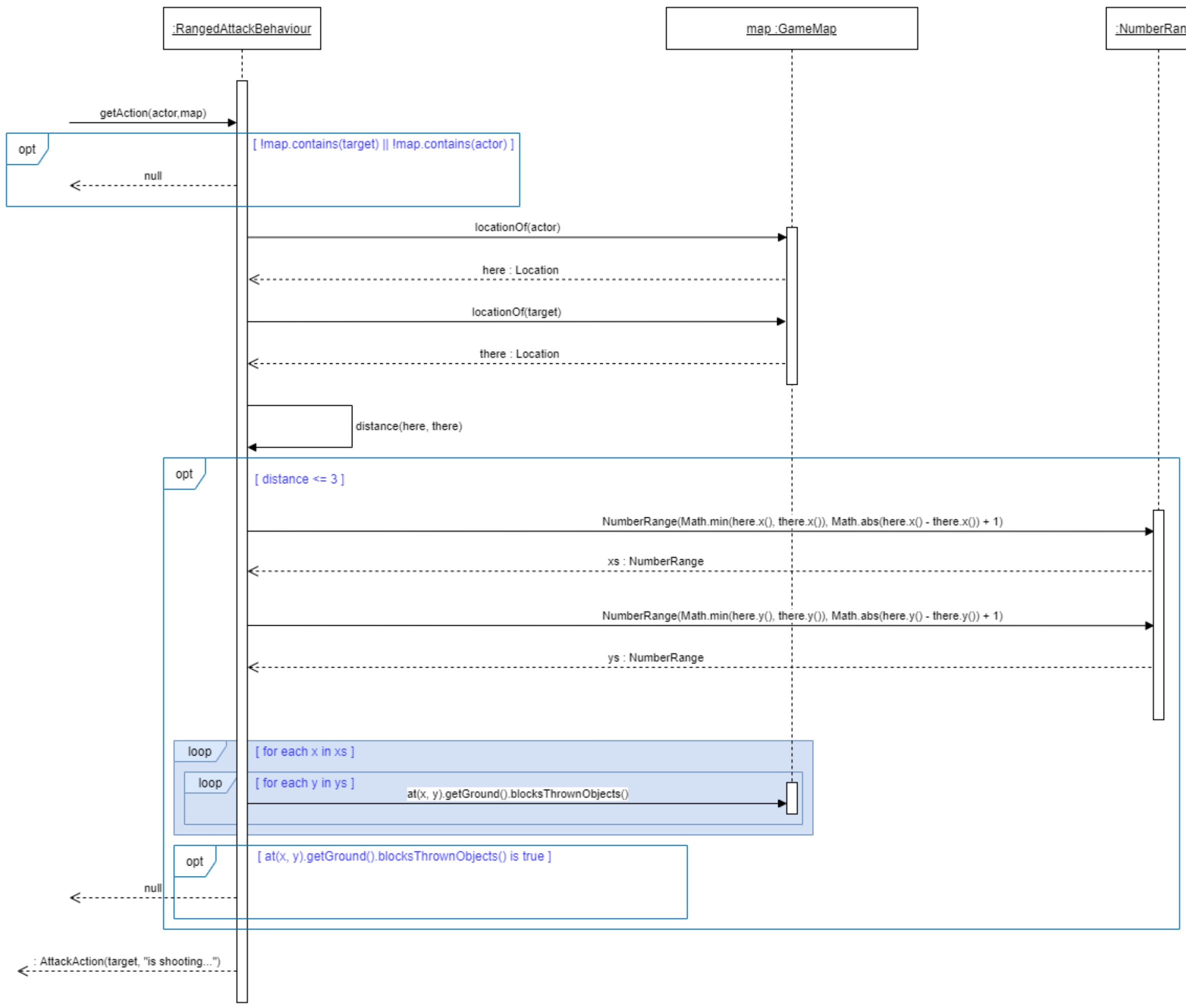
Such design **enhances** the **encapsulation** and **high cohesion** of the Bonfire abstract class and its subclasses, where the respawn location is not available outside of the class, it could only be accessed through a method.

Requirement 3 : New Lord of Cinder & Weapon

Aldrich The Devourer (New Lord of Cinder)

A new class which extends from the **LordOfCinder** class will be created for **Aldrich The Devourer**. This Lord of Cinder will be located at the new map, **Anor Londo**, as a new Lord of Cinder.

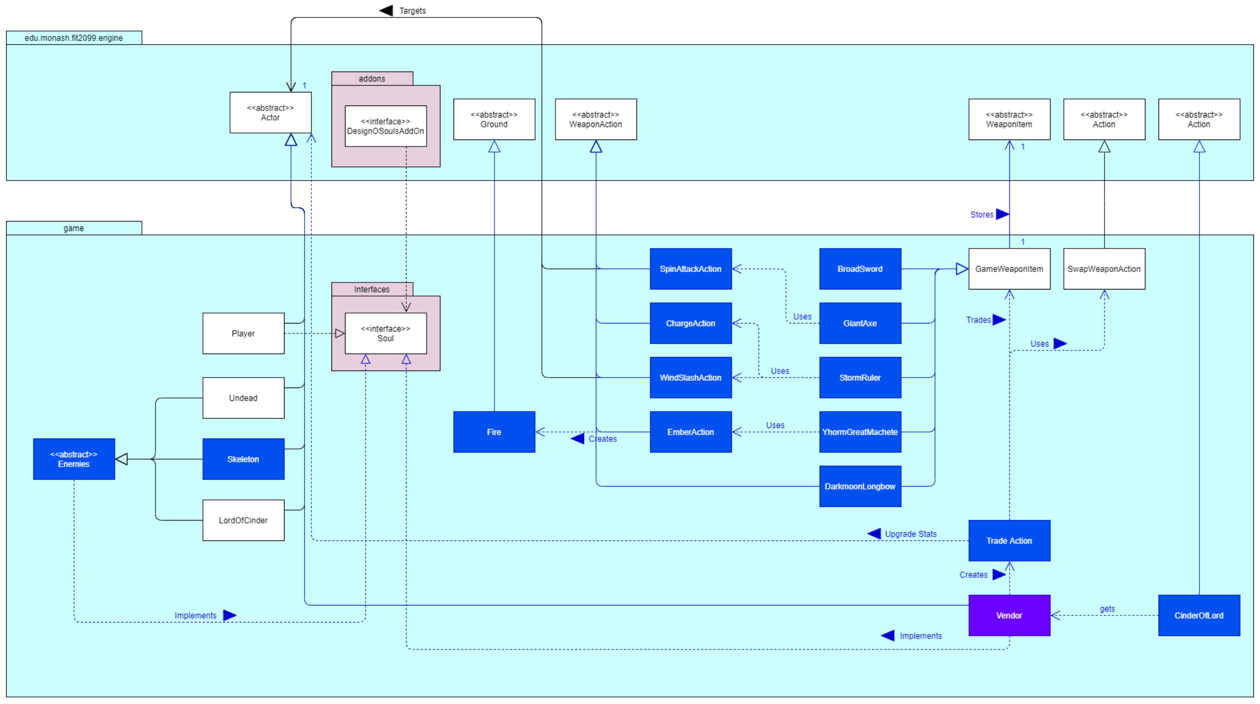
During initialization, a weapon, **Darkmoon Longbow** and an item, **Cinder of Lord** will be added to its inventory. Besides that, at the beginning of each playturn of this Lord of Cinder, it will check if the actor dies by returning a new **DieAction** if this actor is not conscious. Then, it will add a **RangedAttackBehaviour** and **FollowBehaviour** if it has ranged passive.


:RangedAttackBehaviour
map :GameMap
:NumberRange
getAction(actor,map)
opt
[!map.contains(target) || !map.contains(actor)]
null
locationOf(actor)
here : Location
locationOf(target)
there : Location
distance(here, there)
opt
[distance <= 3]
NumberRange(Math.min(here.x(), there.x()), Math.abs(here.x() - there.x()) + 1)
xs : NumberRange
NumberRange(Math.min(here.y(), there.y()), Math.abs(here.y() - there.y()) + 1)
ys : NumberRange
loop
[for each x in xs]
loop
[for each y in ys]
at(x, y).getGround().blocksThrownObjects()
opt
[at(x, y).getGround().blocksThrownObjects() is true]
null
: AttackAction(target, "is shooting...")

Darkmoon Longbow (New Weapon)

Darkmoon Longbow is a weapon for the new Lord of Cinder, **Aldrich The Devourer**. Therefore, a new class which extends the **GameWeaponItem** is created for this weapon.

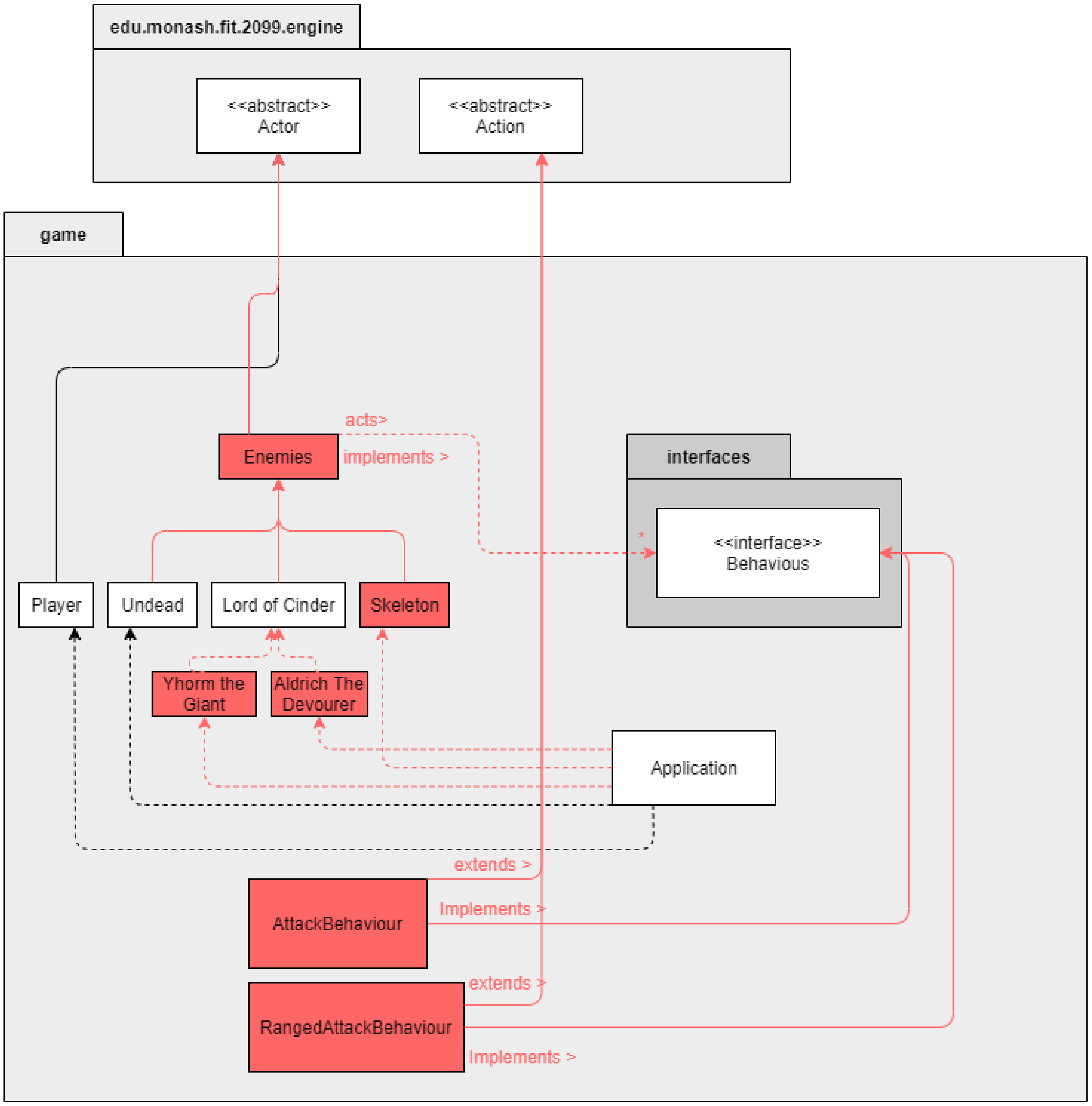
This weapon has a passive which can deal damage to an actor when the actor is 3 squares away from the holder of the weapon. This passive is done by overriding the **tick** method in the Item class and checking for actors in range. Hence this means that it will check if any actor is in range of the holder every playturn. If the actor is a player then the holder will have a **RangedAttackBehaviour** on the player.



RangedAttackBehaviour

RangeAttackBehaviour is a new behaviour class for the holder of the range weapon, so that the holder will perform an **AttackAction** when the target is in range of the holder.

This class is similar to the **AttackBehaviour** class. However in the **getAction** method of this class, it will change the hit rate of the holder's weapon to 0 in **AttackAction** so that it will miss the target if there is a wall between the holder and the target.



Compared to the previous class diagram, this new class diagram added the new Lord of Cinder, Aldrich The Devourer, which extends the Lord of Cinder class. Besides that, a new behaviour named RangeAttackBehaviour is also added into the class diagram.

The implementation of RangedAttackBehaviour can actually be done in AttackBehaviour by checking for the enums of the actor. However, a new class is created for it as we have decided to follow the **Single Responsibility Principle.**

Requirement 4 : Mimic / Chest

Introduction

The mimic is an enemy that resembles a chest, tricking the player that it's loot before attacking the player, because why would there be loot with no consequences in a souls game? Anyway, in this version the chest will remain as a chest until the player interacts with it, which then they will have a 50% chance of either being an actual chest or a mimic.

Chest

The chest as an object, doesn't actually exist. The chest will actually be a **Mimic**, but with their name and display character changed to what a chest would be. When the mimic is in this state, it will do absolutely nothing except, letting the player open it. If the player is lucky, the chest will remove itself and drop the tokens. One weird detail, the way to remove the chest is to 'kill' the chest with something like a **DieAction**.

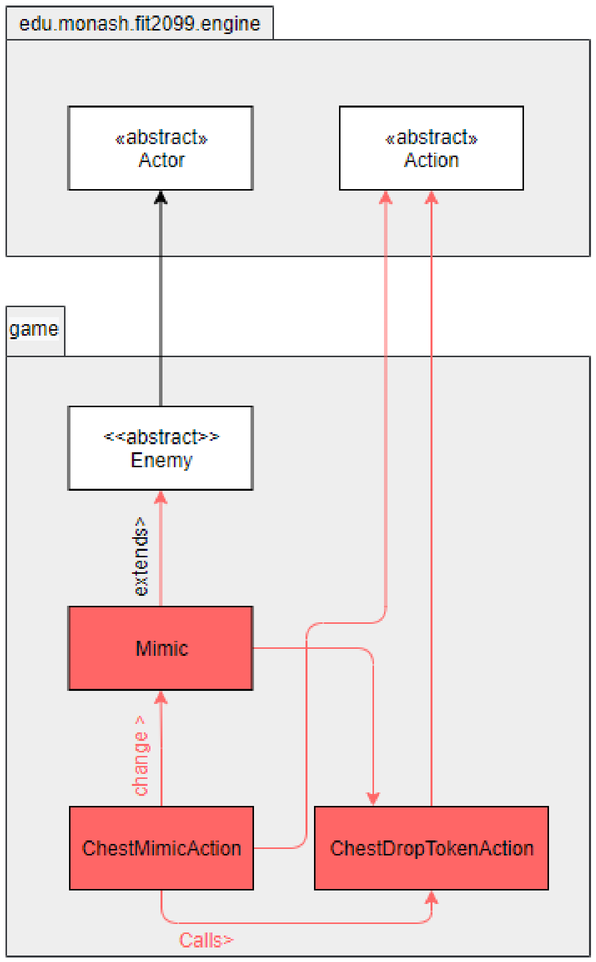
Mimic

If the player is unlucky, then the chest will change its name and display character to a Mimic, and will function mostly the same as any enemy. One difference it has with other enemies is when it dies, it too will drop tokens the same way it does if it was a chest.

Resetting

Resetting a wandering Mimic will include turning their name and display character back to a chest, removing any status and abilities they had that helped them to become a mimic, and of course, putting them back to their original position.

UML



The UML for this requirement will look something like this. All chest and mimic action will be done in a Mimic class. The action of determining if the chest should give tokens or become a mimic when the player interacts with it is done by the 'chestMimicAction'. If the player is unlucky, the action will tell the Mimic class to turn into a mimic, this will be done by giving the Mimic a mimic ability. If the player is lucky, then the Action will call another action which is the **'ChestDropToken'** action then tell the Mimic to remove itself. The **ChestDropToken** action is responsible for randomizing how many tokens it should drop on the map and dropping them, this action is also called when the mimic gets defeated in their mimic form.

Separating these actions will make sure no one class is overwhelmed with too many tasks, following the **SRP (Single Responsibility Principle)**.

Requirement 5 : Trade Cinder of Lord

Cinder Of Lord

Since, Cinder of Lord will be an item to be traded for the corresponding boss weapon, a class named CinderOfLord which **extends Item**, will be created.

In this class, **Dependency Injection** is applied. The constructor accepts an enum of Abilities is added as a parameter everytime the object is created. This is because it is needed for differentiating who's Cinder of Lord it represents so that we can add a TradeAction for the corresponding weapon when the actor has the Cinder of Lord of any bosses.

Vendor

Some minor changes have been made for the Vendor class. After adding two TradeAction for the BroadSword and also the GiantAxe, it will loop through the actor's inventory and add a new TradeAction for the corresponding weapon based on the Cinder of Lord that the actor has.

Trade Action

In the execute method in TradeAction class, some conditions statements are added because the Cinder of Lord needs to be removed when the player trades for the boss weapon. It will also loop through the actor's inventory and check if the actor has the Cinder of Lord for the corresponding boss, then remove the Cinder of Lord from the inventory and swap the corresponding boss weapon to the actor.

In the menuDescription method, the return String also shows the name of the weapon and the cost of the weapons for the newly added weapons for trading so that it shows user friendliness to the players.

References

[1] *Are Singletons Bad.* (n.d.). Cocoacasts. Retrieved October 4, 2021, from https://cocoacasts.com/are-singletons-bad