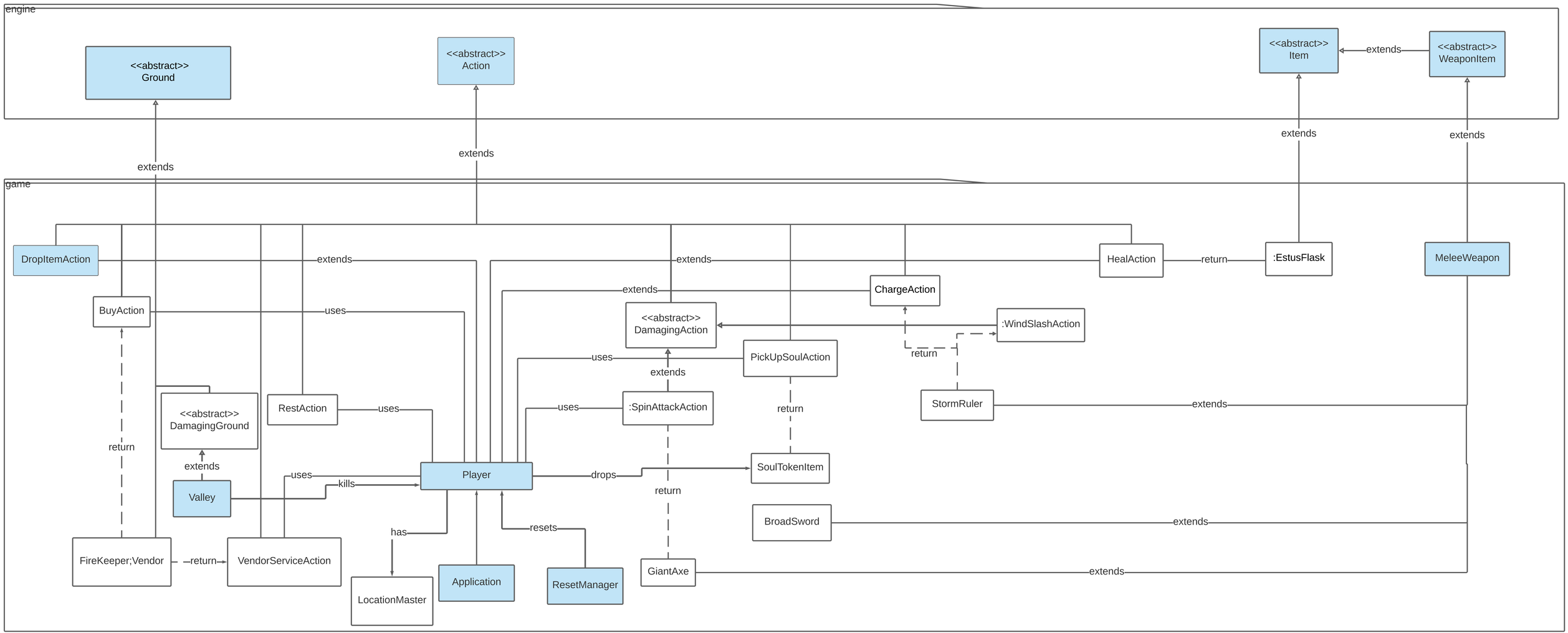
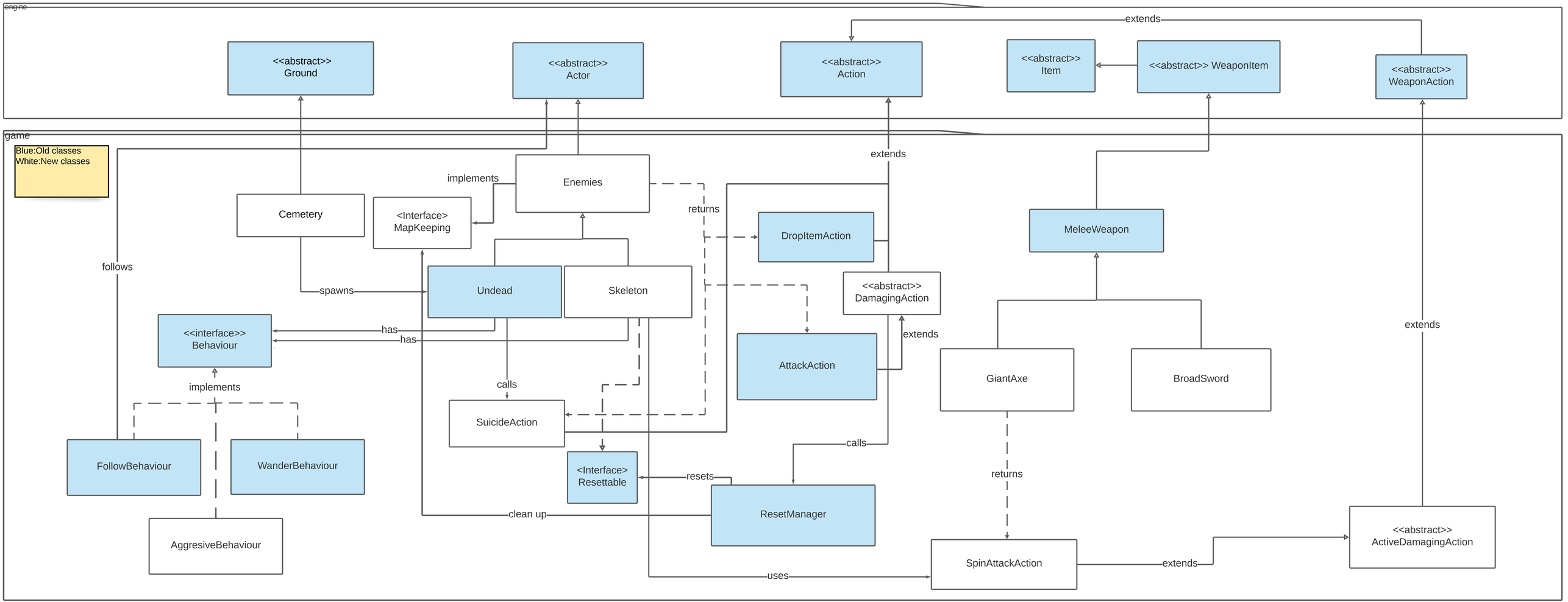
Blue:Old classes
White:New classes

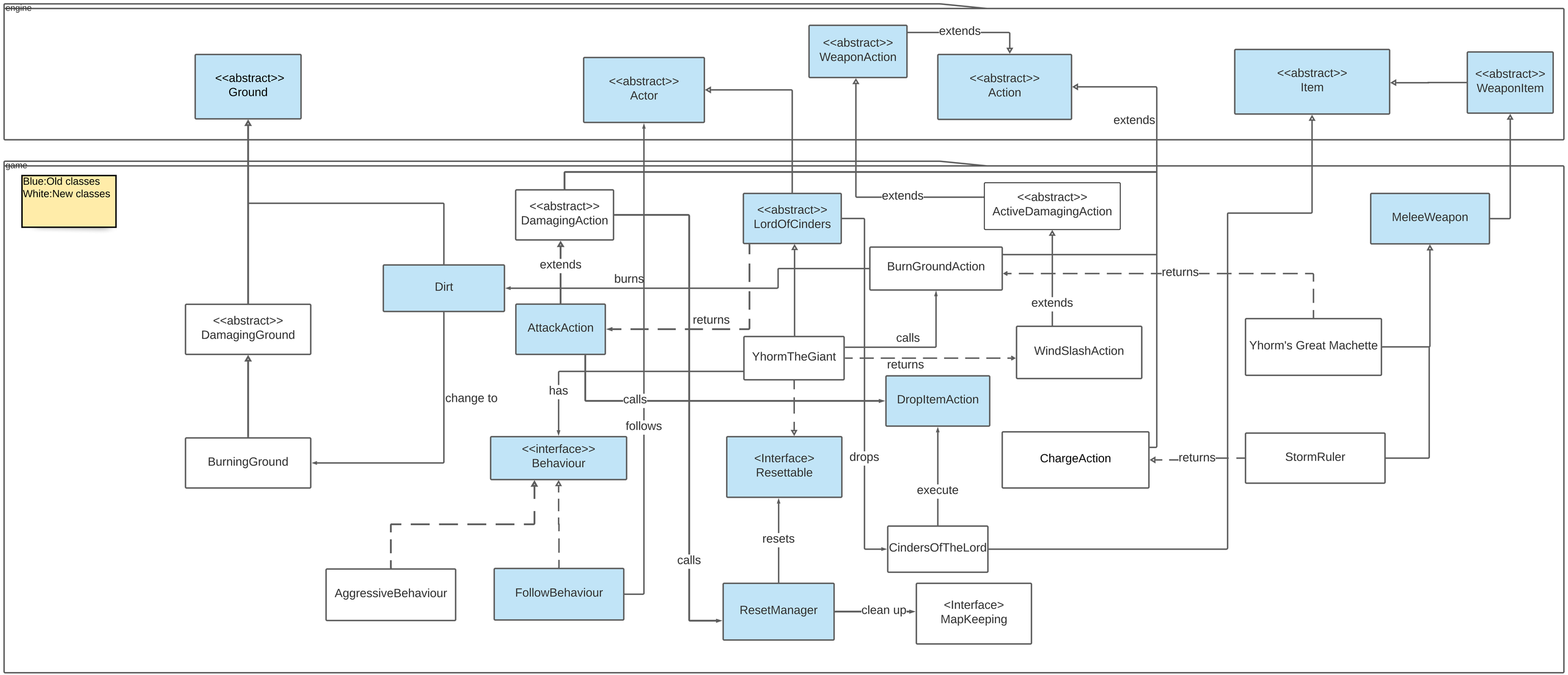
Class Diagram: Player



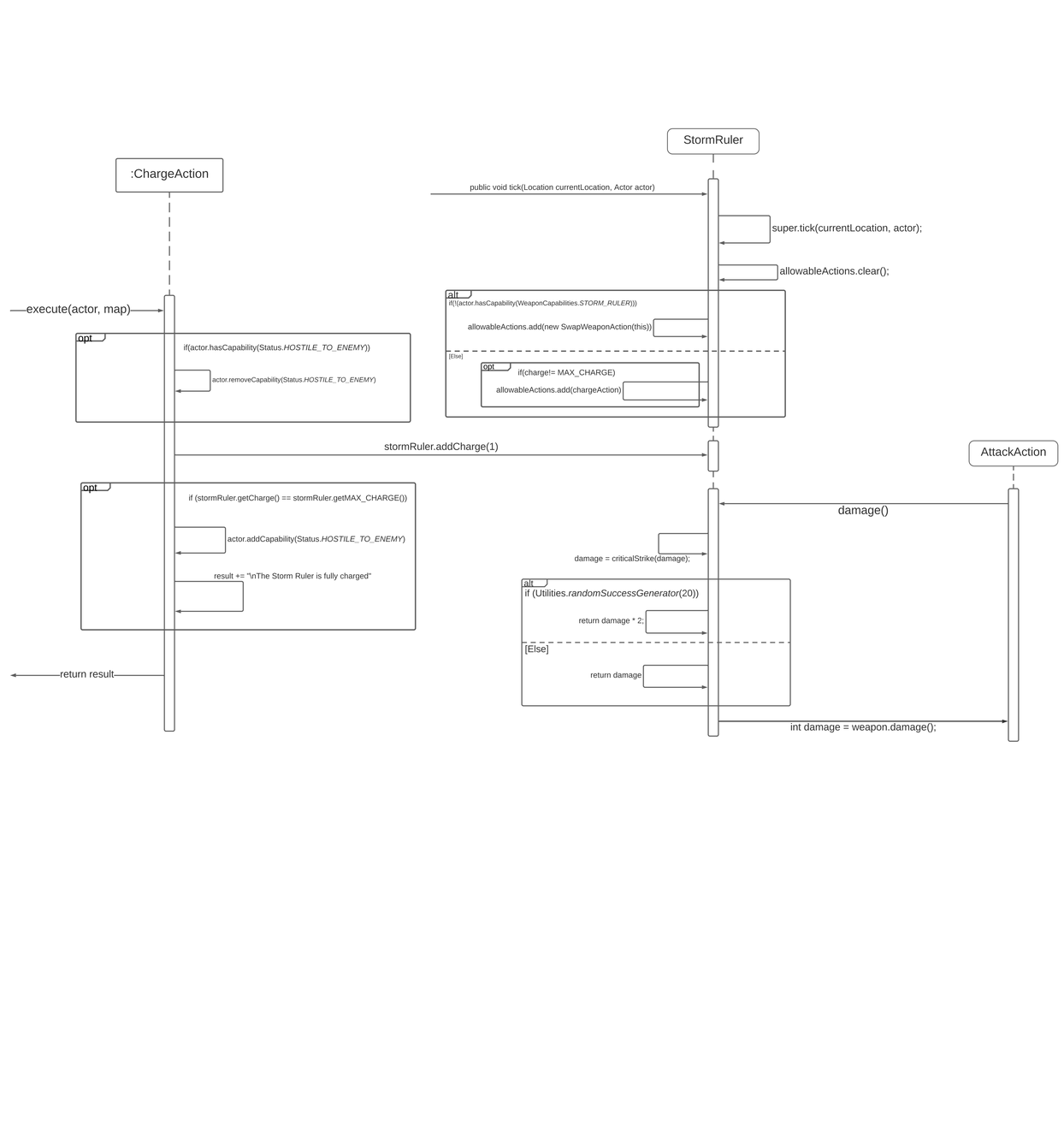
Class Diagram: Enemies

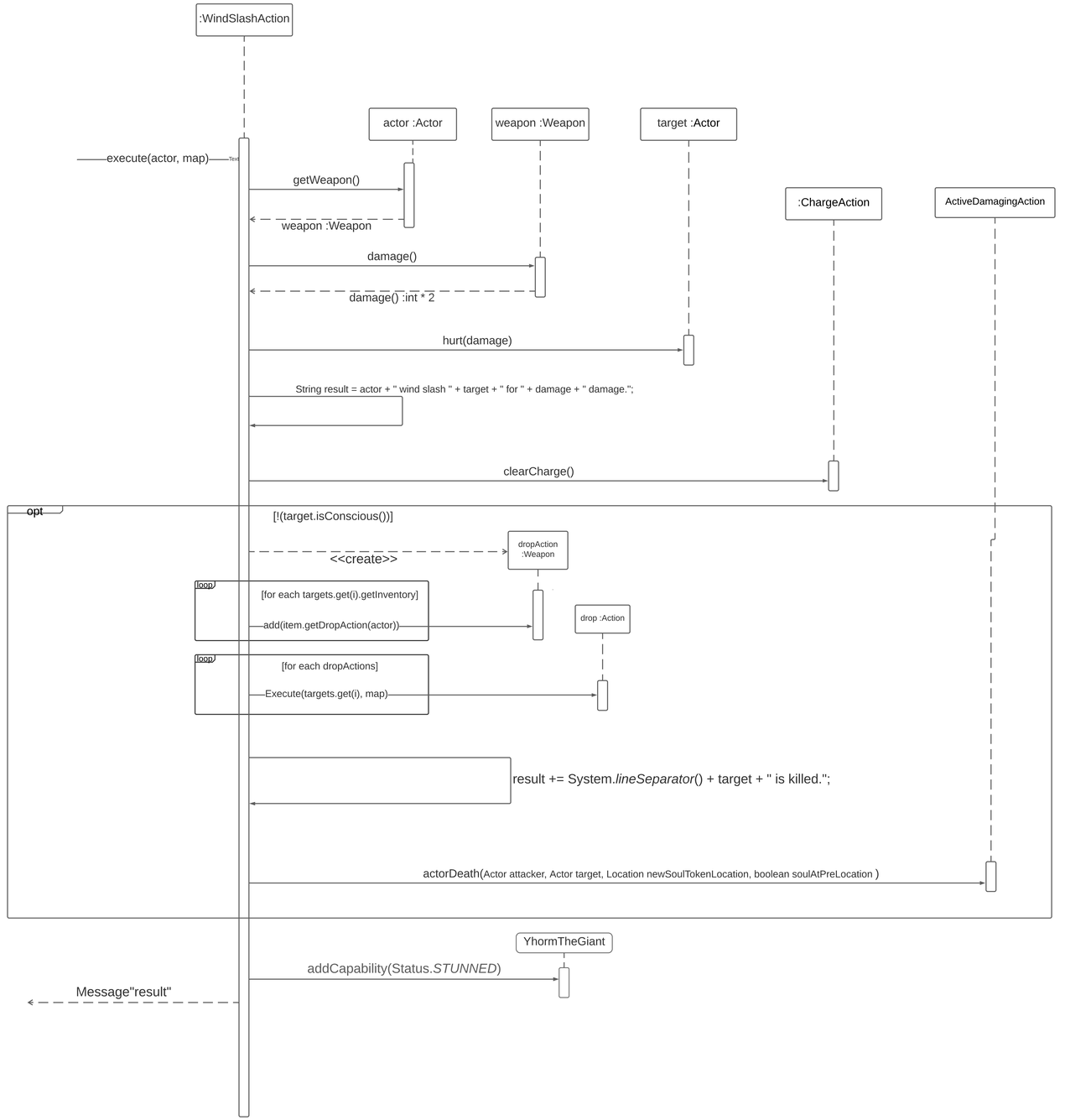


Class Diagram: LordOfCinders

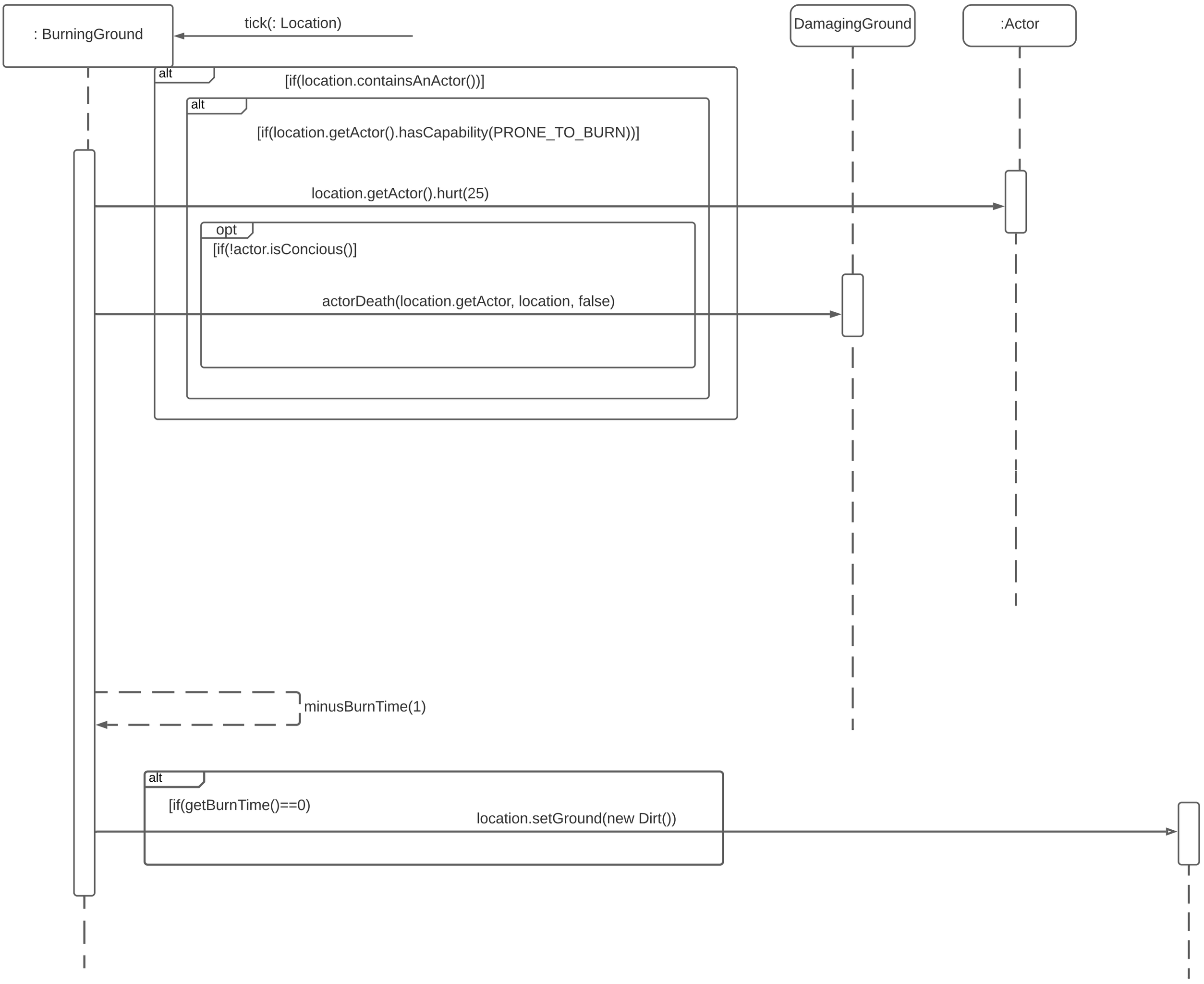


Interaction Diagram - Storm Ruler

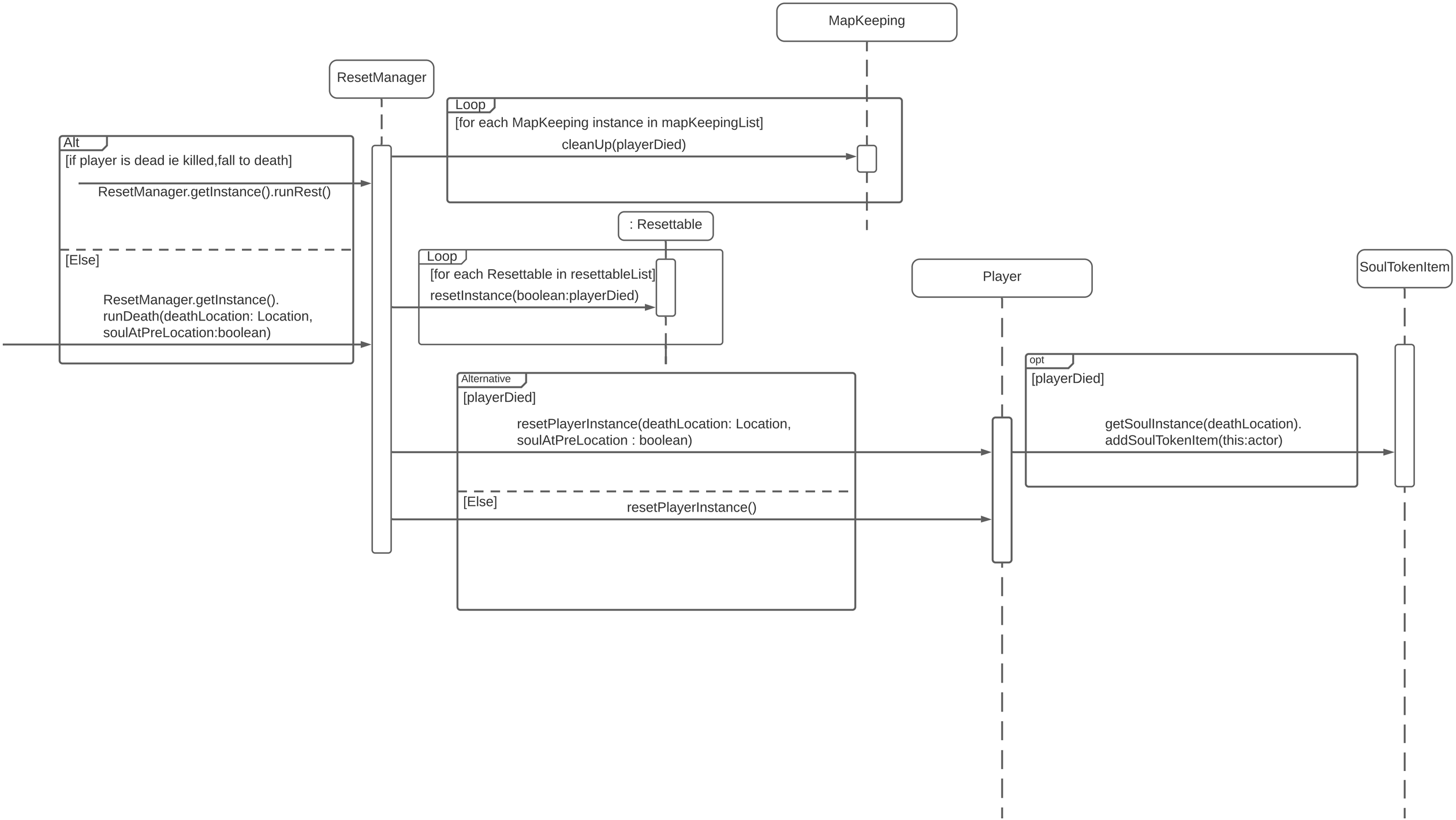




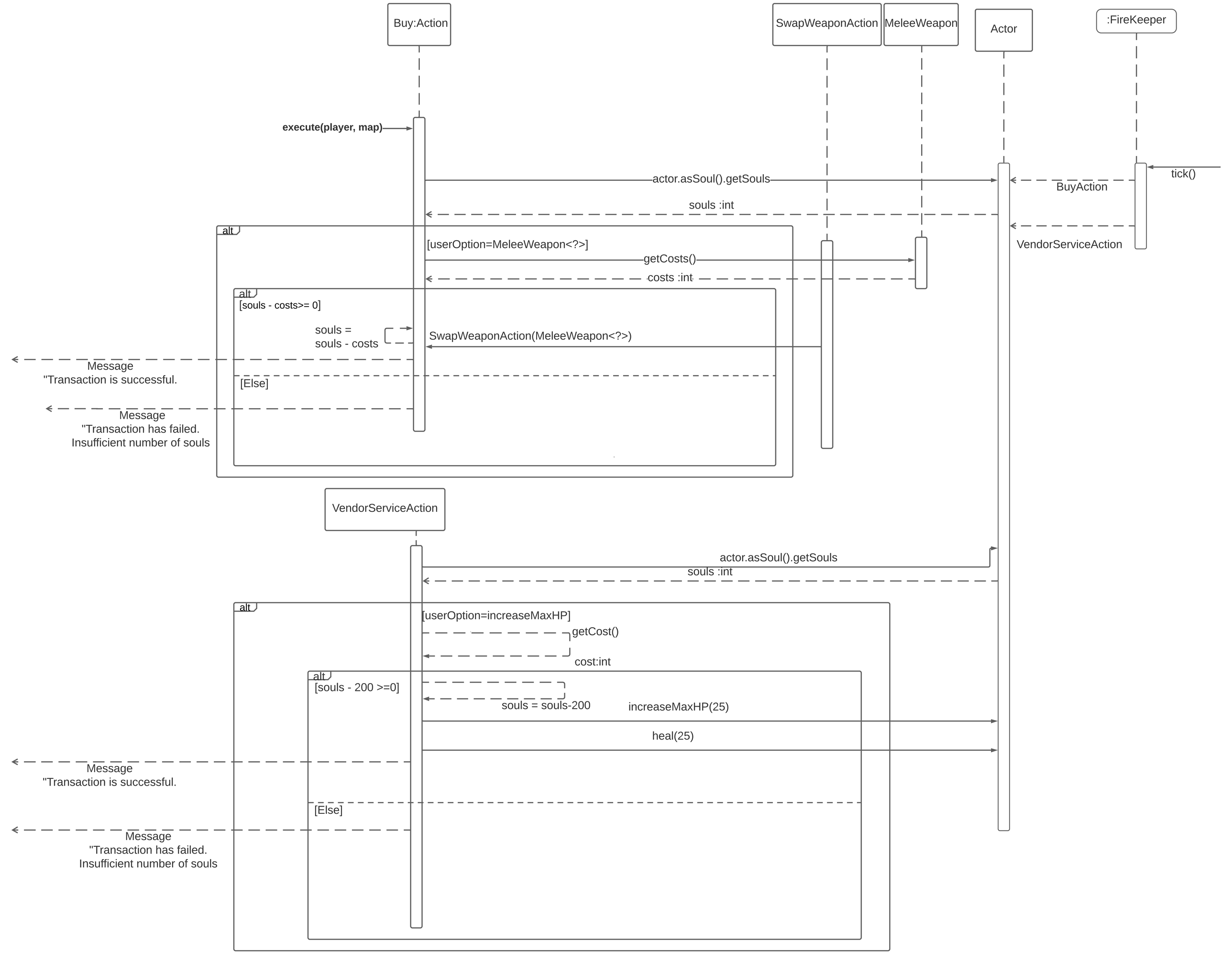
Interaction Diagram - BurningGround



Interaction Diagram - Reset Manager



Interaction Diagram - Vendor



Assignment 2 Design Rationale
Designosouls

Note: Changes made to the design in Assignment 2 have been highlighted with green color.

Player and Estus Flask:

1) **HealAction** extends **Action** and is used to increase the player's health
    - HealAction will handle all the logic of healing Player.
    - Player will need to override its heal method to make sure that it accepts multipliers.
    - This is to reduce unwanted downcasting
2) **Estus Flask** will check whether to return a HealAction to the holder, by doing this the Player class doesn't have to check whether estus flask still has charges. This is to remove dependency on the Player class.
3) Player can use **HealAction** to heal 40 percent of max hitpoint

Bonfire:

1) Returns **RestAction**
2) **RestAction** extends **Action**
3) Player can use **RestAction** to heal to max Hitpoint, fill the **Estus Flask** to full charge
4) At the same time, will calls the **ResetManager** to reset the world
5) Bonfire is a ground because it's easier for inserting them into the map
6) RestAction is an action so that it will take care of the logic of rest (rest reset) to reduce dependency of the Bonfire. (Don't want the Bonfire to be implementing more than what it should)

Souls:

1) Souls are like currency in the game, **ALL CURRENT ACTORS** will have instance variable soul, FOR NOW.
2) Player can use soul to buy item from **FireKeeper**
3) **Player** can get soul by killing monster
4) Every actor in the game (for now) will implement the Soul interface, this forces a contract of them implementing all the soul related logic, then when we need to do any transaction of soul with them we can upcast them to Soul instance and do the transaction easily.
5) This is done so that actors that don't need this implementation don't need to implement this logic.

Enemies:

1) All enemies will be hostile to Player, they have Capabilities (Enum Status.HOSTILE_TO_UNKINDLED)
    - So that its more general as we might have an item in the future to not aggro the enemies
2) All enemies will have their own behaviour, and will apply one behaviour at a time only

3) If **Player** is not in aggro range, enemies will either have Wandering Behaviour or they does nothing (DoNothingAction)
4) All enemies will follow the **Player** using **FollowBehaviour and AggresiveBehaviour**
5) **This is done so that all enemies can just loop through all the behaviours and return appropriate action instead of doing all the implementation within the playTurn method, which will be repetition of code.**

- Skeleton
    1) **Skeleton** will hold either **Broadsword** or **Giant Axe**
    2) When hitpoint reaches < 0, **Skeleton** has a 50 percent chance to restore to full hitpoint. This logic is done within the hurt method(override), so we don't need to purposely create a new method to deal with this logic, also we avoided downcasting with this implementation.
- Undead
    1) **Undead** will spawn within the vicinity of **Cemetery**
    2) While in **Wandering Behaviour**, will call **SuicideAction** randomly by chance at a 10 percent rate. (**SuicideAction** is an extension of **Action**).

Terrains:

1) All **Terrains** will be extending **Ground, this is so that they inherit the tick method of the ground, we do not want to extend from item because they are not supposed to be portable, even though we can implement most of the logic in the tick method inherited from the Item.**
2) They will return a suitable action if possible

- **Valley**
- **Extending from DamagingGround since it will damage any actor standing on it(kills will be a better word), this will allow them to reuse the actorDeath method to reduce the repetition of code when dealing with actor's death**
- **Also we try to not make this class handle too many implementations. So anything related to actor's death can just call the inherited method.**
    1) Only the **Player** can enter currently
    2) Player with Capabilities(Status.PRONE_TO_FALL)
        - So that it allows Player to gain resistance to fall damage, if game allows, (maybe allow Player to float?)
    3) Will kill the **Player** and call the **ResetManager via the actorDeath method**
    4) The **SoulTokenItem** will be placed at the tile where the **Player** was before stepping into Valley, this logic will also be handled by actorDeath method.

- **Cemetery**
    1) Will spawn one **Skeleton** at a 25 percent rate every turn

- **BurningGround**
- Extend DamagingGround, see Valley
    1) **YhormTheGiant** with his **Great Machete** will burn the **Dirt** tiles in a small area, causing **Dirt** to change into **BurningGround.**
    2) **BurningGround** will last for 3 turns before reverting back into **Dirt,** done within the tick method since each burning ground knows which location they

are in. Since they know their own location, they can call the setGround method to replace itself with new Dirt easily.

3) **BurningGround** will burn any **Actor** with the Capabilities(Status.PRONE_TO_BURN), causing them to lose 25 hitpoints every turn they are standing on the BurningGround
    - If the actor is immune to burn damage, then it will not take damage.
    - For example YhormTheGiant will not have (Status.PRONE_TO_BURN)

Soft Reset:

1) For every class that needs to be reset must implement **Resettable**. They will implement their **resetInstance()** method that is unique to them

    - So that ResetManager will not need to care how they reset, they just need to know they already reset.

2) The instances of classes that implement **Resettable** must not be removed from the map and respawn by creating a new instance again

    - This is so that we do not lose progression of any Actor that implements Resettable

3) Those that need to be removed and respawn from the map will be done using the **cleanUp()** and **resetInstance()** method.

    - cleanUp() will be in charge of removing actors that need to be removed from the map
    - resetInstance() will take care of respawning too. Since each enemy that needs to reset knows their initial position.

4) Those classes that already remove from the map permanently will not be in the list of reset by returning false in **isExist()**

    - For example, after killing a boss aka LordOfCinders, then it will not be spawn again and thus it will not be handled by ResetManager in the future

5) If player died, we will check whether **SoulTokenItem** exist in the map, if True, remove from the map and destroy the instance, the insertion of new **SoulTokenItem** will be implemented by classes that kills the **Player**

    - SoulTokenItem will be a singleton class, only one SoulTokenItem can exist at a time
    - This is so that we make sure we don't have more than one instance, as everytime the Player dies, the existing SoulTokenItem will be destroyed.

6) We added a new MapKeeping interface to separate the task of cleaning up the map and reset.

    - This is done to make sure that some enemies are not implementing Resettable, as they don't need to reset themselves.

- Those classes that implemented MapKeeping must implement the method cleanUp() to remove themselves from the map, since each class knows the best way to remove itself from the map.

Weapons:

1) All classes that extends the **WeaponItem** must implement their passive and active skill accordingly (except some cases, see **WindSlashAction**)
2) All passive skills must be implemented within the child class itself where we use a method to manipulate the damage. Combining this with the overridden hurt method, we reduce the need to downcasting.

- GiantAxe:
    1. When the **Player** uses **Giant Axe** it calls the **SpinAttackAction** and it extends to **ActiveDamagingAction**. From **ActiveDamagingAction** it will then extend **to Action.** The **SpinAttackAction** deals 50%(aka 25 damage in this case) to enemies in the vicinity(i.e on adjacent tiles to the **Player**)
    - At this point the player can use
    2. We separated the logic of returning SpinAttackAction and executing action, and we let the SpinAttackAction deal with how to damage adjacent enemies. This is done to reduce dependency on the GiantAxe class.

- StormRuler
- Extending Melee Weapon to allow itself to implement the getActiveSkill method, this will simplify the implementation of allowable active skills towards the target
    1) Allow the **Player** to use **ChargeAction**, where it will increase the charge instance variable of **StormRuler**
    2) Once the charge reaches 3, the **Player** can unleash **WindSlashAction** towards **YhormTheGiant** only
    3) If the **Player** equip this Weapon and attack another **Actor** that is not weak to **StormRuler (!Status.WEAK_TO_STORMRULER)**, the damage will be reduced by half
        - In the future we might encounter more enemies that is weak to StormRuler
    4) The Player equipped with StormRuler can deal double damage towards the **Actor** at 20% rate.
    5) We have 2 different actions for this weapon, the charge action and wind slash action. This class will handle the returning of both the active skill and charge action so we don't have to check whether to return the action or not from the other actor class, and reduces unwanted dependency and repeating code.

- BroadSword
    1) The Player equipped with **BroadSword** can deal double damage towards the **Actor** at 20% rate.

- GiantMachete:
    1. Only **YhormTheGiant** can use this weapon for now

2. It returns that active skill **EmberFormAction**
3. If the Actor uses the **EmberFormAction**, the success hit rate of the holder increases to 90%.
4. In addition to this, upon using EmberFormAction, the holder burns **Dirt** which then changes to **BurningGround**. **BurningGround** hurts the Actor with **Capabilities(Status.PRONE_TO_BURN)** that stands on it by 25 hit points.**(see BurningGround)**
5. Again, we want to make sure that EmberFormAction will handle most of the logic of burning the ground and what not, this is to make sure that YhormTheGiant class doesn't implement more than what it should.

Vendor:

1) So for the **Vendor** first we set its location manually in the **Application** class of the **edu.monash.fit2099.game package**. We do this by writing the code GameMap.at(x,y).addActor(:Firekeeper).The coordinates(x,y) are the coordinates of a spot inside the Firelink Shrine.
To set the vendor's location manually we basically place its symbol inside the FireLink shrine and the **Vendor** is set to extend **Ground** instead of being an **Actor** as the **Vendor's** location on the game Map is permanent everytime the game resets.

2) Next we store the price of all the weapons the **Vendor** has to sell in the child class of **MeleeWeapon** class(ie BroadSword, GiantAxe). Each child class will implement their price.

3) We create a **BroadSword** class and a **Giant Axe** class which extend the **MeleeWeapon** class. This allows the classes to inherit all the attributes and methods from the **MeleeWeapon** class.

4) We retrieve the price of the weapon by overriding the **getPrice()** method from the **MeleeWeapon** class.
The prices are added manually for each weapon in the VendorSales class instead of using the getPrice() method (reducing dependency)

5) Next we create a new action(by extending the **Action** class) called **BuyAction**. This class has a method called **Transaction** which basically asks the player for their input and then processes the transaction accordingly.

6) If the player does not have enough souls to purchase the item we return a message telling the player that the transaction has failed. After a **weapon** is successfully purchased the **swapWeaponAction()** method is called which swaps the weapon currently in the player's inventory with the newly purchased weapon.All of these

actions occur during one turn and the player can purchase multiple options as long as he/she does not enter “Exit”.

- This is done so that we allow the Player to buy multiple items in one turn.

Changes made to Vendor:

- Instead of carrying out everything in one class I split the **Vendor** duties into two classes:(Single Responsibility Principle)
    1. **VendorSales**: This class was dedicated to only store the items and their corresponding costs which the vendor will be selling to the player
    2. **BuyAction**: This class is dedicated to only carrying out the transaction between the player and the vendor so it solely focus on retrieving the item and its cost from the VendorSales and then carrying out the purchase.

- I added a new **Action** called **VendorServiceAction**: This action is a service provided by the vendor to the player who can purchase it to get an increase of Hp and max HP by 25 hit points

- Another change made was that instead of having the **Player/Actor** buy everything in one turn (like it was decided during the first assignment) I changed it so that the player will be able to purchase one item/action in one turn. This helped to keep a fresh supply of items every time the player wanted to buy an item or an action.