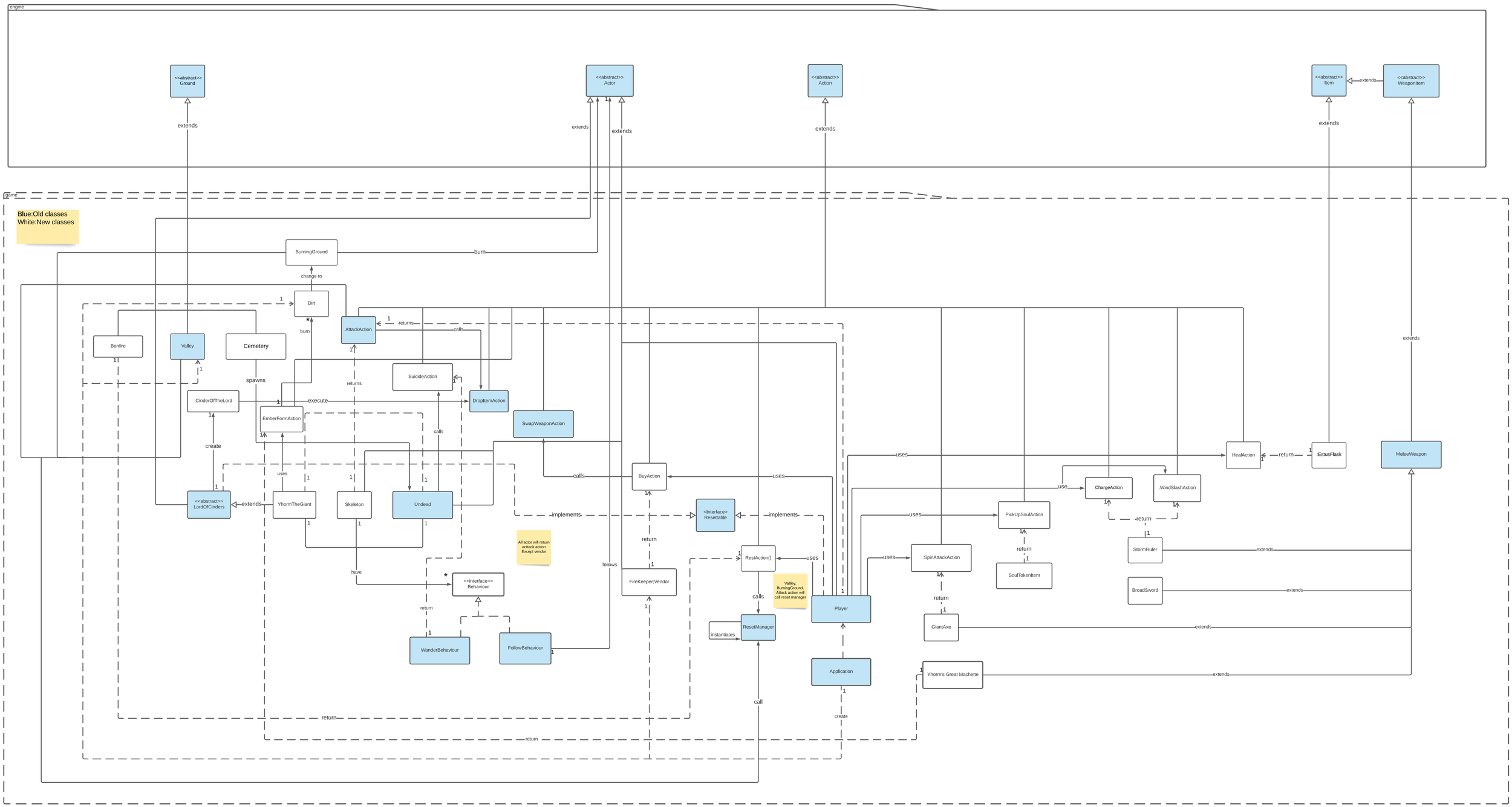
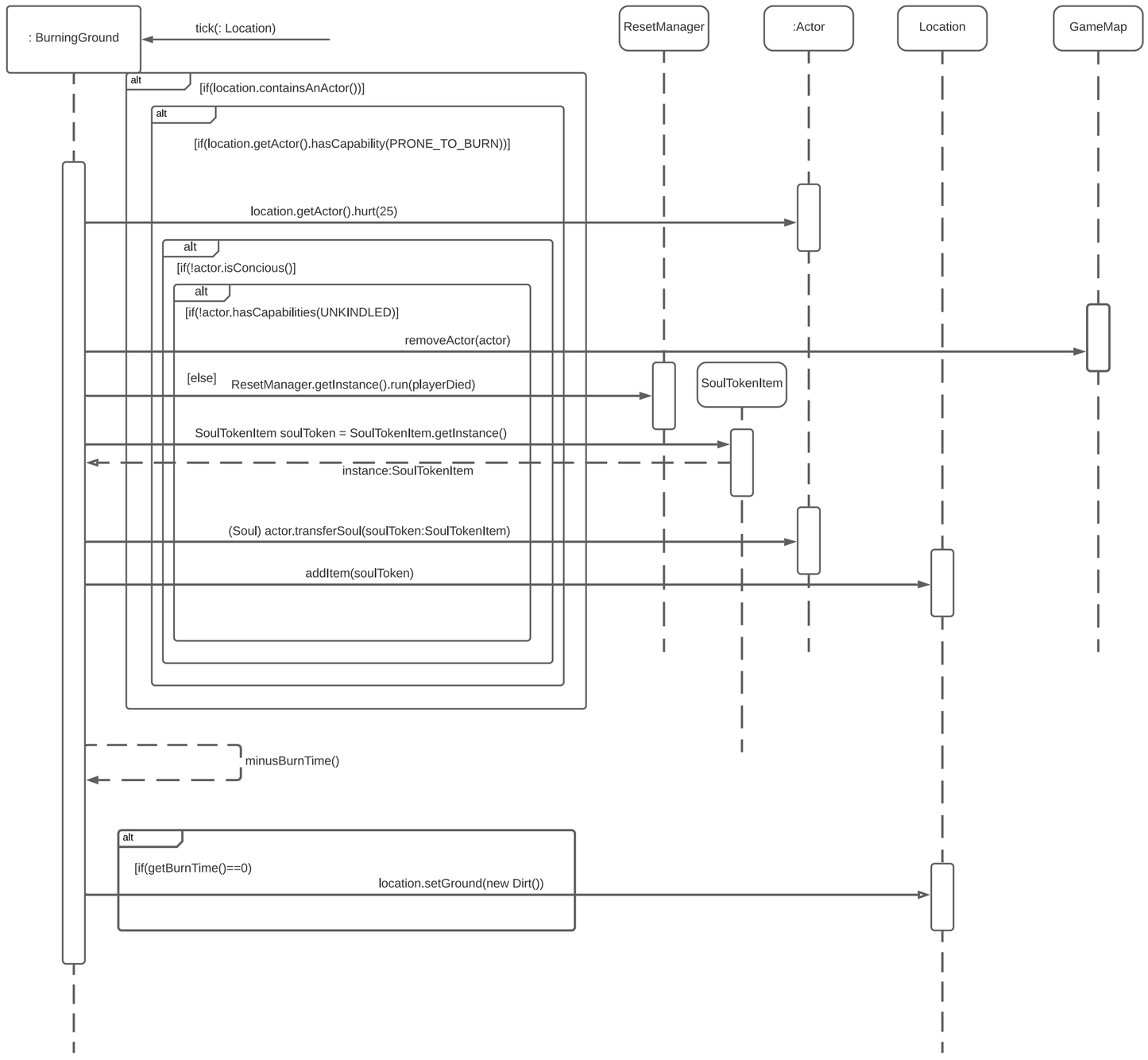
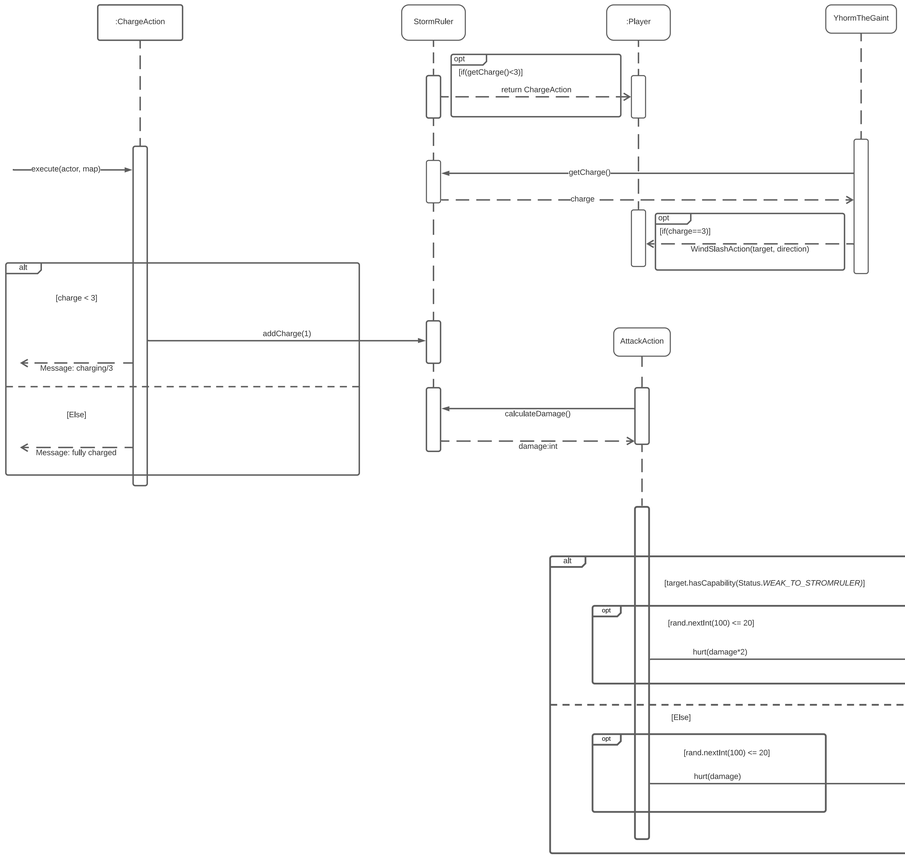
Class Diagram

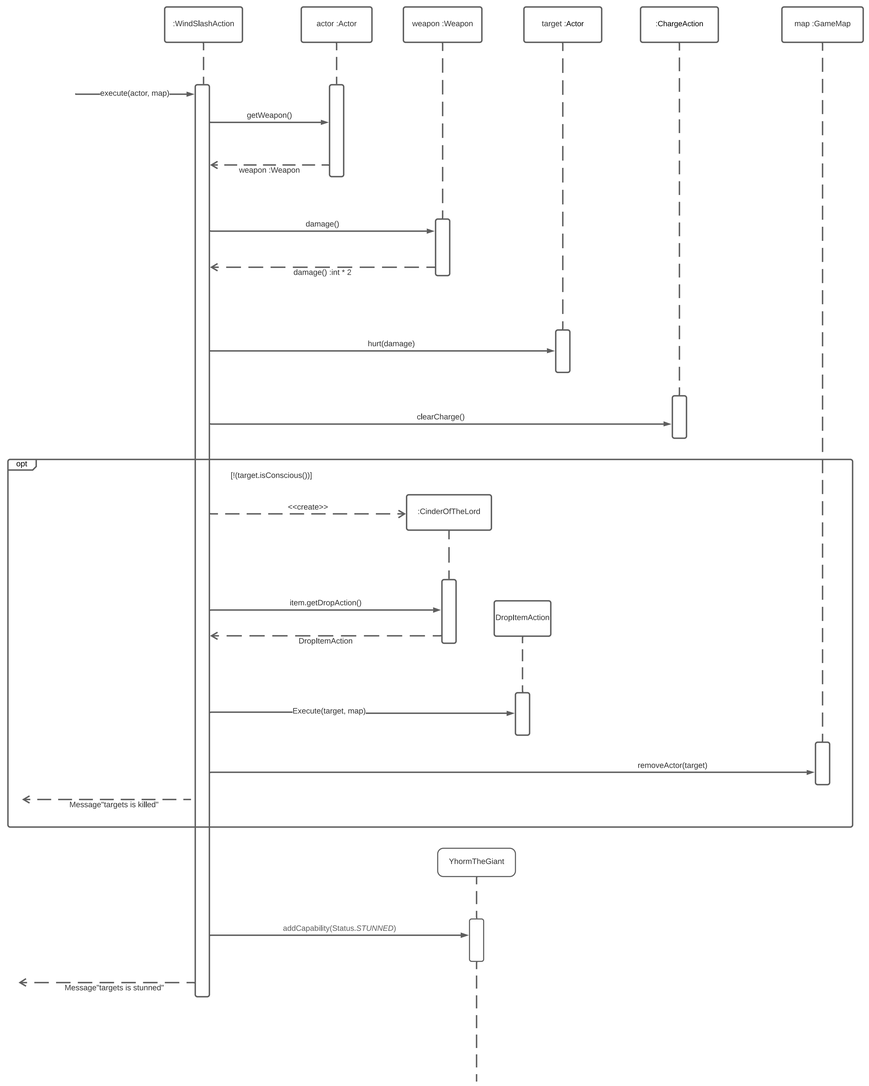


Interaction Diagram -
Burning

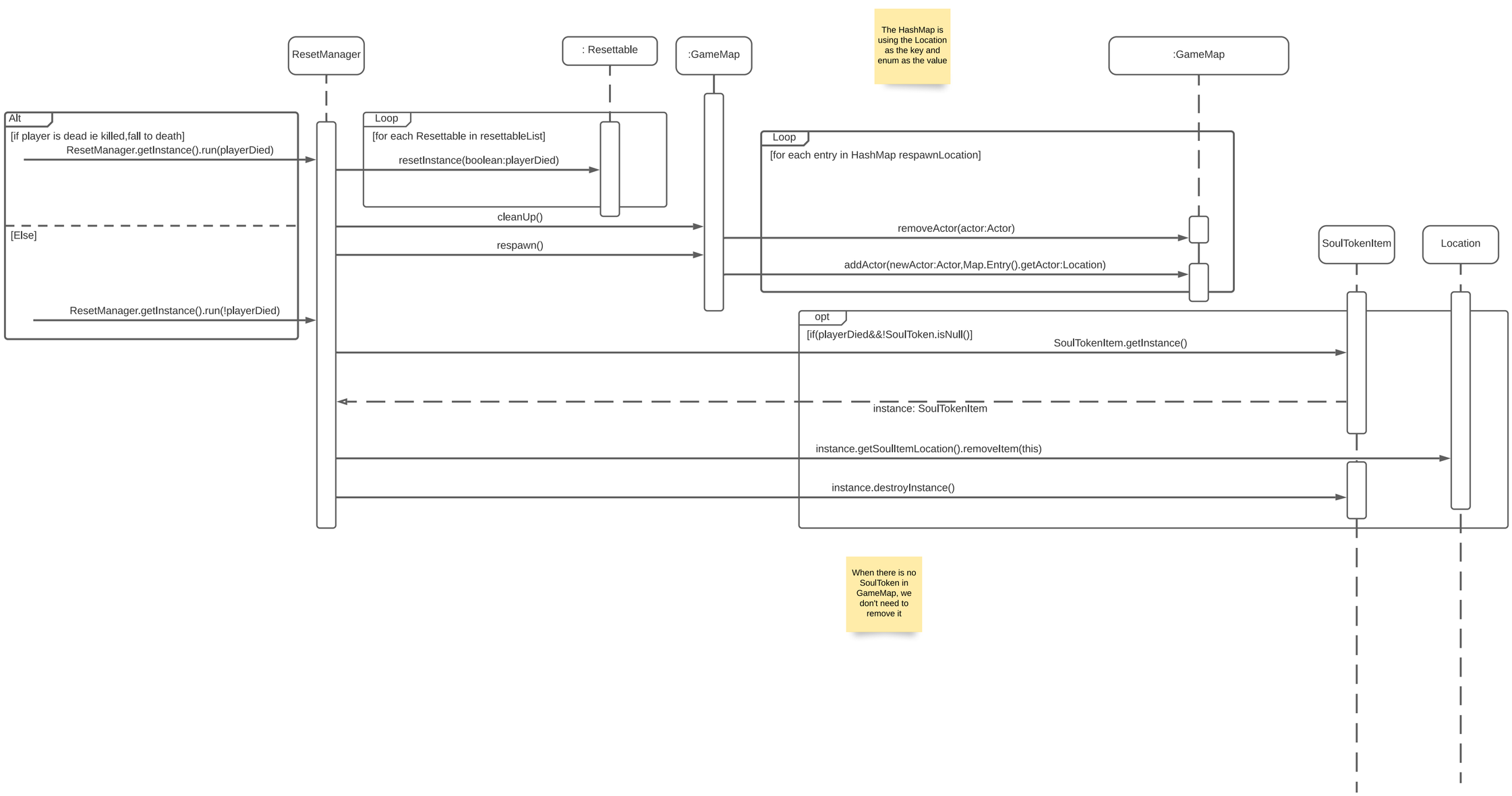


Interaction Diagram - Storm Ruler

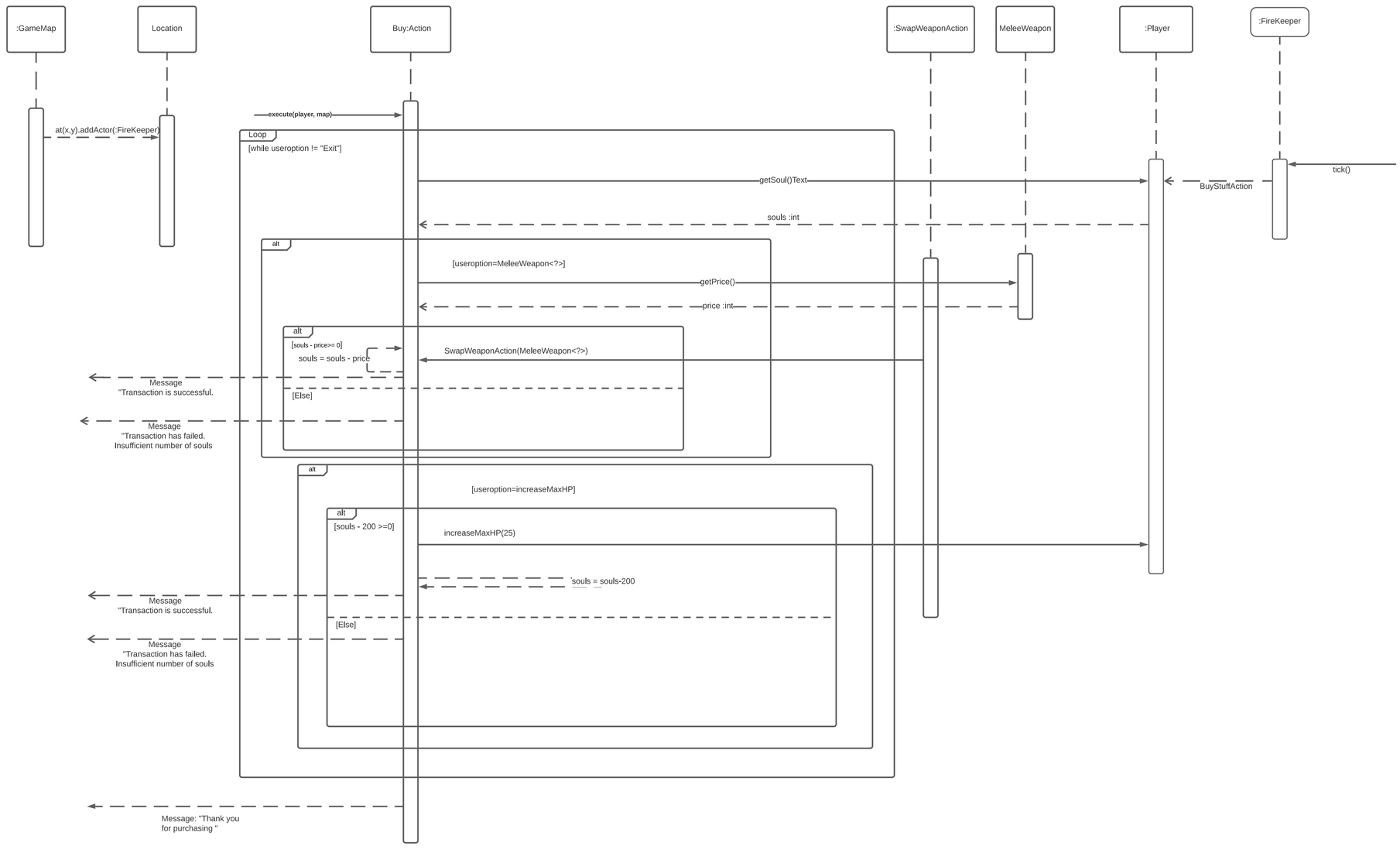




Interaction Diagram - Reset



Interaction Diagram - Vendor



Assignment 1 Design Rationale
designosouls

General Design Rationale

1) Since the engine is designed in such a way that every Actor, Item, Ground will return an Action for the user aka Player to do (Action based), most of our approach to get something done will be done through a specific child class that will extend the Action class.

2) We made more classes that will extend from the classes in engine package so that these child classes will implement their own specification, thus making the process of calling methods easier (overriding and overloading)

3) We try our best to make sure that this game is future proof by using Capabilities class for checking whether we should return any action or not, this is so that we can make our game more interesting in the future (we are not limiting ourselves!!, @see Enemies, Terrains, Weapons)

4) LordOfCinders will be a abstract class that must be extended by the bosses (for now, only YhormTheGiant), this is so that all bosses will implement their special feature such as YhormTheGiant will return WindSlashAction to the Player whenever possible.

5) ResetManager will be our big boss who will take care of our soft reset, any classes that implement Resettable must implement their resetInstance method so that ResetManager will just call resetInstance() for each Resettable, ResetManager don't need to know what is done, they just know they will do something to reset themselves. Then they will just clean up the map, and respawn the monster back to their position (except Undead because Cemetery will be responsible to spawn them)

6) If any of the LordOfCinders died, they will be ignored when calling ResetManager to reset again, using isExist(), so we don't respawn the boss(unless you like to torture yourself with endless loop of killing bosses :?). This concept applied to all classes that implement the Resettable interface.

7) If our Player died, the specific entity that kills our Player will call the ResetManager, then it will insert the SoulTokenItem into the map AFTER RESET SINCE RESET WILL CLEAN UP THE MAP

8) We design our SoulTokenItem as a singleton class, such that only ONE SoulTokenItem will exist at a time, this is to facilitate the cleaning up of soul (if player died again before retrieving the soul)

9) We uses tick() method for implementation of checking whether any Actor standing on it, please see BurningGround and Valley, they will be used to kill/damage the Actor with the right Capabilities, as well as calling the ResetManager when needed.

10) StormRuler will only return ChargeAction and YhormTheGiant will only return the WindSlashAction, this is because we only can use WindSlashAction against YhormTheGiant.

11) FireKeeper is designed to let the Player to purchase stuff, but all purchasing will be done in one turn, this is to facilitate Player who like to save up a lot of soul to buy things at once.

Below will be explaining the requirement in a specific manner

Player and Estus Flask:

1) **HealAction** extends **Action** and is used to increase the player's health
2) Everytime the player uses the **Estus Flask** it returns the **HealAction**
3) **Estus Flask** will return **HealAction** if charge > 0
4) 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

Souls:

1) Souls are like currency in the game, **Player** will have instance variable soul
2) Player can use soul to buy item from **FireKeeper**
3) **Player** can get soul by killing monster

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**

- 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.
- 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**
2) They will return a suitable action if possible

- Valley
    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**
4) The **SoulTokenItem** will be placed at the tile where the **Player** was before stepping into Valley

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

- BurningGround
  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**
  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 **respawn()** method.

   - cleanUp() will be in charge of removing actors that need to be removed from the map
   - respawn() will respawn the actor at 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.

Weapons:

1) All classes that extends the **WeaponItem** must implement their passive and active skill accordingly (except some cases, see **WindSlashAction**)

- GiantAxe:
    1. When the **Player** uses **Giant Axe** it calls the **SpinAttackAction**(which is an extension of **Action**). The **SpinAttackAction** deals 50%(aka 25 damage in this case) to enemies in the vicinity(i.e on adjacent tiles to the **Player**)

- StormRuler
    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.

- 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)**

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.

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.

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.