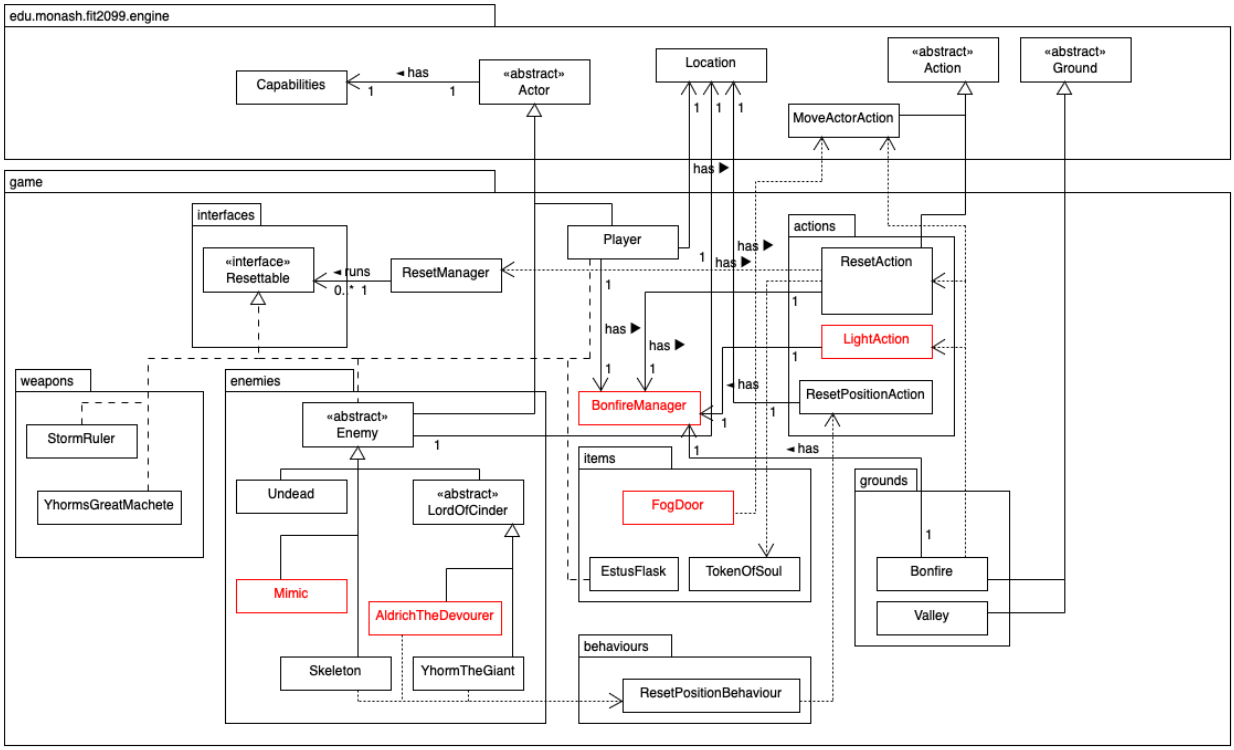
A3 (New Features) Design Rationale

Matthew Crick, Joshua Nung

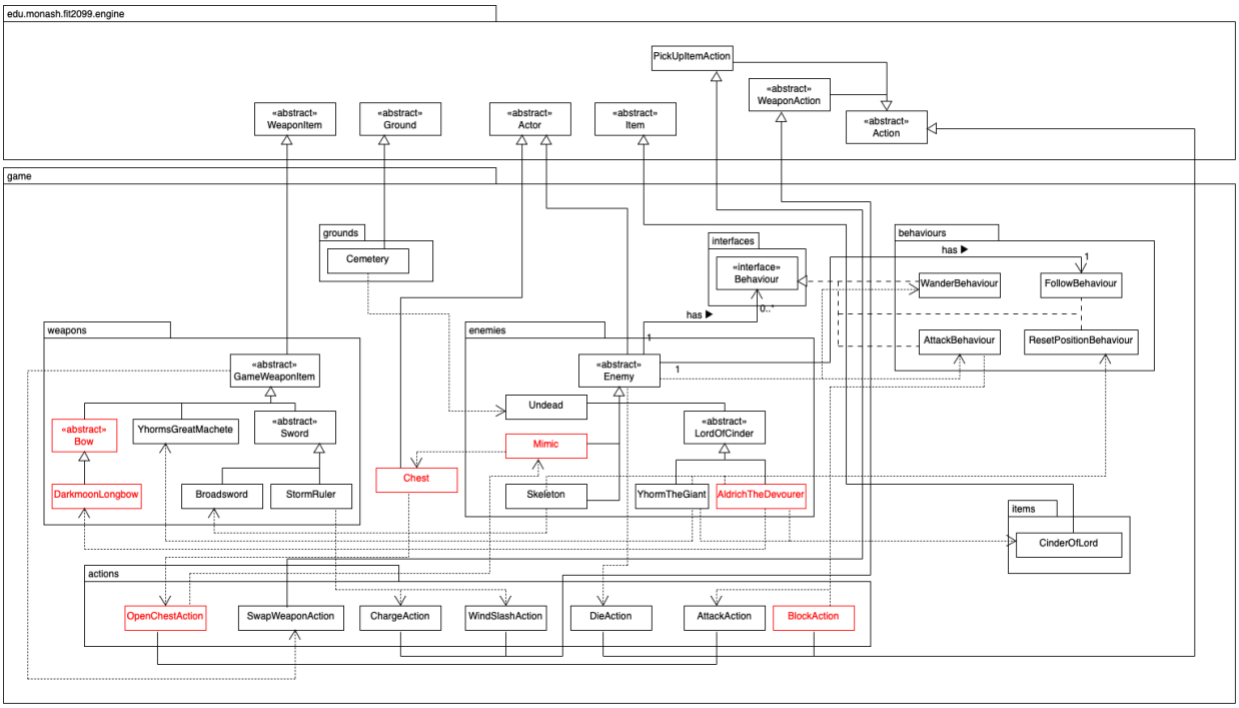
Requirement 1: New Map & Fog Door, and Requirement 2: Updated Bonfire



This UML diagram is an updated version of the Reset UML diagram from Assignment 2.

- To manage the different bonfires and whether they are lit or not, as well as MoveActorActions (teleporting actions) from the bonfires, a BonfireManager class with a hashmap of bonfires is used to handle this logic. This supports the **Single Responsibility Principle**, as the bonfire manager contains the methods required in a single class.
- The **Interface segregation principle** has been maintained with the Resettable and Soul interfaces, which have remained small to easily extend resetting and souls functionalities for the new enemies, Mimic and Aldrich.
- FogDoor is an item which provides a MoveActorAction to the player to another fog door.
- The bonfires now can produce different actions – LightAction, ResetAction, or a MoveActorAction (for teleporting to another bonfire).

Requirement 3: Aldrich the Devourer (Lord of Cinder), and Requirement 4: Mimic / Chest



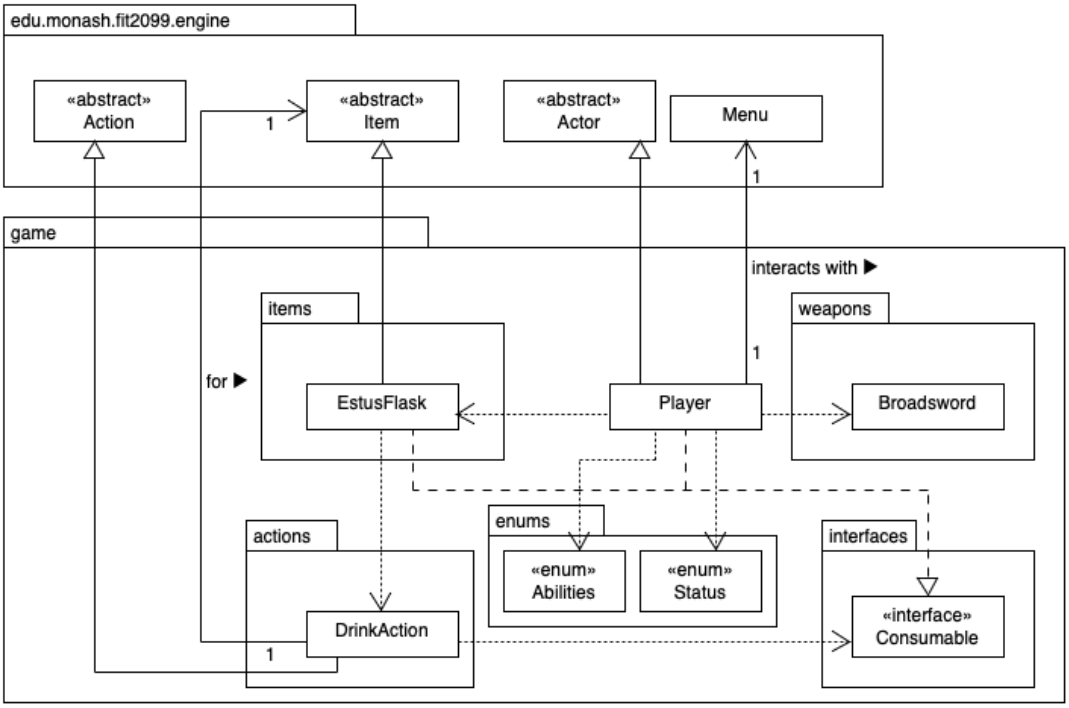
This UML diagram is an updated version of the Enemies UML diagram from Assignment 2.

- To implement the new enemies Aldrich the Devourer and Mimic, inheritance of the abstract classes LordOfCinder and Enemy has been done. This adheres to the **Open/closed principle**. For example, adding Aldrich is an example of opening for extension as logic specific to Aldrich such as giving Aldrich a Darkmoon Longbow is able to be done through inheritance in the subclass. This does not affect the parent LordOfCinder class, thereby adhering to the closed for modification principle.
- Inheritance also supports the **Liskov substitution principle**, such as the new Chest subclass which extends the Actor class. The Chest class behaves like any actor, such as providing actions to the player like other actors, or being added to the application as an actor.
- The DarkmoonLongBow class is a subclass of the abstract Bow class which extends GameWeaponItem.
- OpenChestAction and BlockAction are new actions to support the new chest feature and ranged attack logic for when a wall is between the player and enemy.

A2 Design Rationale

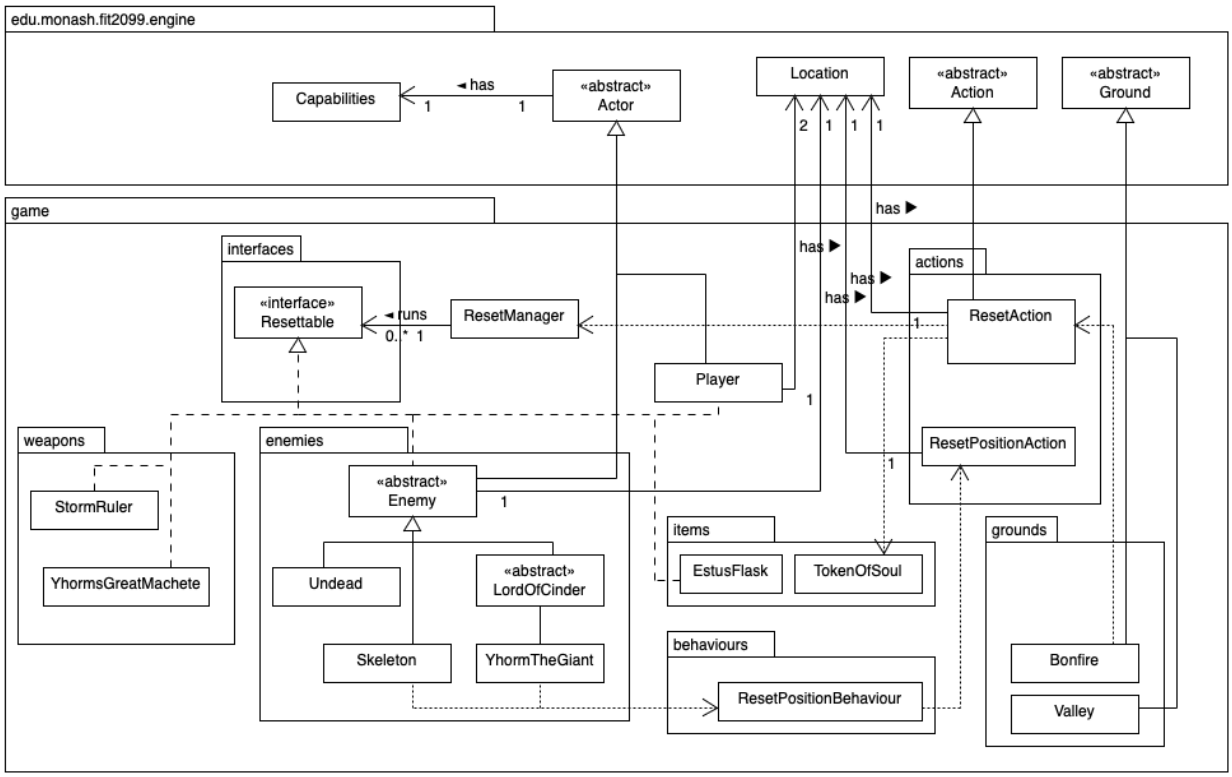
Matthew Crick, Joshua Nung

Requirement 1: Player and Estus Flask



- The player has a Menu object to make choices from
- The Broadsword is added to the Player's inventory, as well as an EstusFlask.
- To implement the EstusFlask feature, a Consumable interface is created that Actors and Items can implement – this is a form of **Dependency inversion principle** because although the Player and EstusFlask will implement Consumable, in the future other Actors or consumable Items can also implement the Consumable interface and be used similarly, reusing the methods of the Consumable interface.
    - This avoids a tight coupling between the Player and the EstusFlask. The Estus Flask provides a DrinkAction that depends on Consumable, to the Player.
- For different features of the game, various capabilities through the Abilities and Status enums can be given to the Player as well as other Actors and Items to indicate certain properties or statuses. These are reusable and can be checked by different classes.

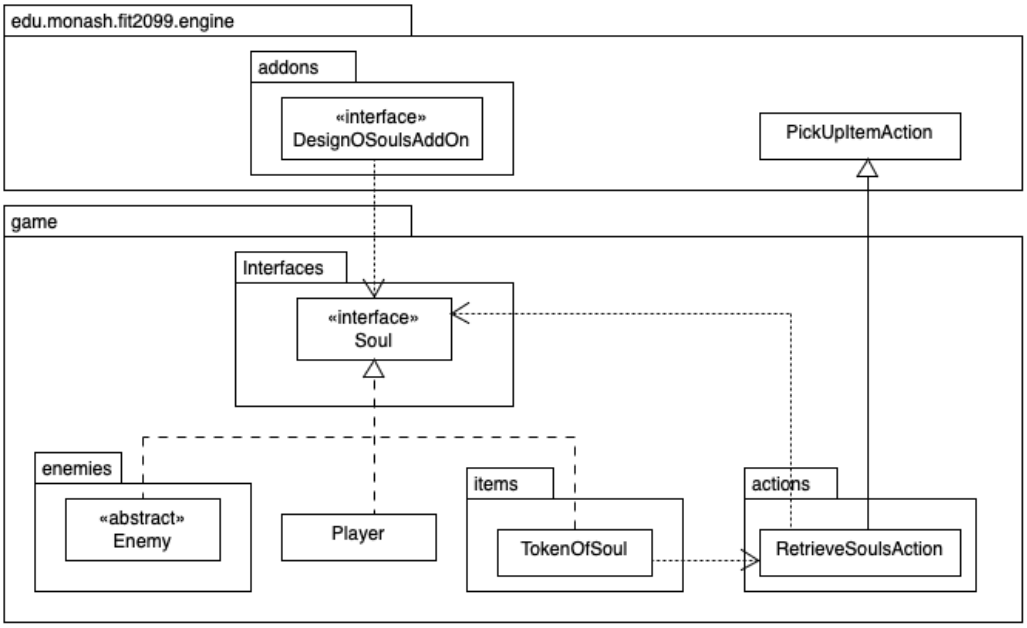
Requirement 2: Bonfire, Requirement 6: Dying in the game / Soft reset, and Requirement 5: Terrains (Valley)



- Resetting occurs when the player either rests at the bonfire or dies, and this occurs through a ResetAction, which is created by the Bonfire ground, and also created when the Player dies. A future consideration could be to make these two ways of resetting occur through separate actions.
- To manage resetting, any entities that need to be reset to a certain state when the map is reloaded implements the Resettable interface. This includes the player, enemies, EstusFlask, and some weapons. This allows each type of entity to provide its own unique logic for resetting. The ResetAction then calls the ResetManager to loop through the Resettables and run each instance's particular implementation of resetInstance().
- Some enemies also need their positions reset, namely Skeletons and YhormTheGiant. They have a ResetPositionBehaviour as one of their behaviours, that generates a ResetPositionAction which is activated when a reset occurs. The ResetPositionBehaviour class can be reused for future enemies that need to have their positions reset.
- To manage the initial locations that an Actor should be returned to, an attribute storing the initial location can be made for each Actor, and this is passed through the constructor of an Actor when creating such an Actor. The ResetPositionAction also makes use of a initial location attribute, and an Actor can inject their initial location into the constructor of the ResetPositionAction.

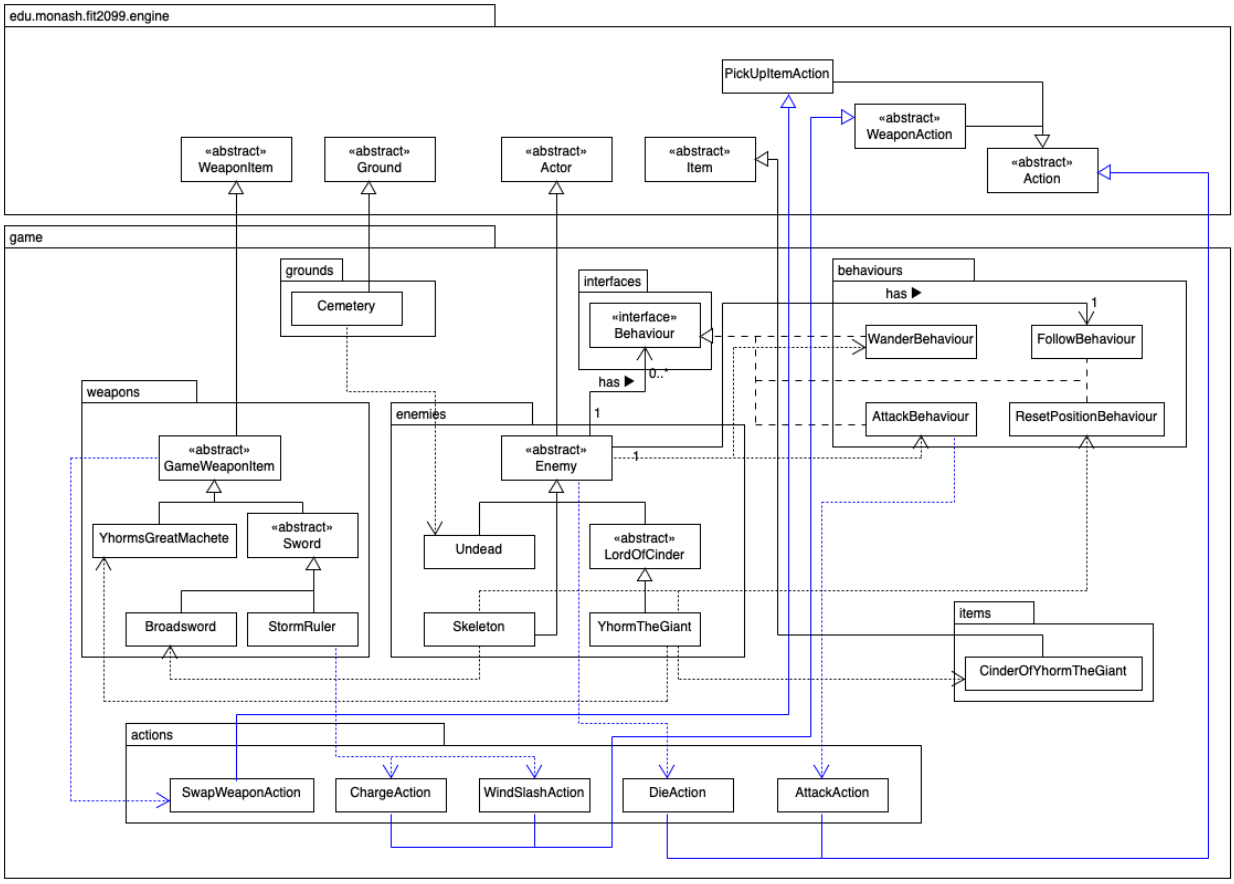
- The ResetAction is also responsible for generating a TokenOfSoul which requires knowledge of the Player's previous location, so the Player also has an attribute for their previous location, which is used by ResetAction to place the TokenOfSoul at the location the Player was at before they died.
- The Valley causes the Player to die; this is handled through a check for the CAN_FALL capability that only the Player has – this capability allows for a distinction between the player and enemies from falling in the valley.

Requirement 3: Souls



- To implement the souls feature, a Souls interface is used that can be reused for any instance that needs to use souls, including the Player, enemies, and token of soul, and any new classes in the future that need to use souls can quickly implement the Souls interface also. Using the asSoul() method allows upcasting to the interface that reduces dependencies between the actual classes transferring souls to each other.
- Since the TokenOfSoul needs to give souls to the Player when the Player picks it up, a RetrieveSoulsAction is made that extends the PickUpItemAction to include the asSoul() method and logic related to souls.

Requirement 4: Enemies, Requirement 5: Terrains (Cemetery), and Requirement 7: Weapons



- An enemies package was created. The abstract Enemy class that extends the abstract class Actor from the engine is made so that attributes and methods that are similar for all enemies can be created and reused.
    - To handle the playTurn() for enemies, behaviours are created including attack and wander behaviours that have their own logic for determining actions.
    - The floor restrictions that are to be shared amongst all forms of enemies can be implemented by using a single capability BLOCKED_BY_FLOORS that can be added to the Enemy class and inherited by all enemies.
    - The abstract class LordOfCinder has similar advantages to Enemy but allows for a distinction between bosses and regular enemies, so that if future bosses have similar features, they can be shared in the LordOfCinder parent class.
- The cemetery ground has a dependency relationship creating undead objects under a certain chance given there is not an actor standing upon it
- GameWeaponItem extends WeaponItem and has a SwapWeaponAction to override its PickUpItemAction as in this game, there is a particular functionality where an Actor can only have one weapon at a time.

- Enemies including YhormTheGiant and Skeleton have a dependency relationship with YhormsGreatMachete and Broadsword respectively. This relationship implies that these classes interact with each other as the enemy receives an instance of the weapon. The design rationale here is to allow enemies to have weapon objects and these weapons can be reused for different enemies and have features added to the weapons that are usable by any Actor holding the weapon.
- A Sword abstract class has been made that extends GameWeaponItem, since the two swords in the game both have a critical passive that can be placed in the Sword class. The advantage of this can be seen if more weapons were to be added that want to share the properties of Sword or rather if the abstract class Sword was to be upgraded to its extended subclasses to all share a new feature that are not wanted for non-boss weapons.
- StormRuler creates ChargeAction and WindSlashAction which extend the WeaponAction class, so that the getActiveSkill() method of the Weapon interface can be used.