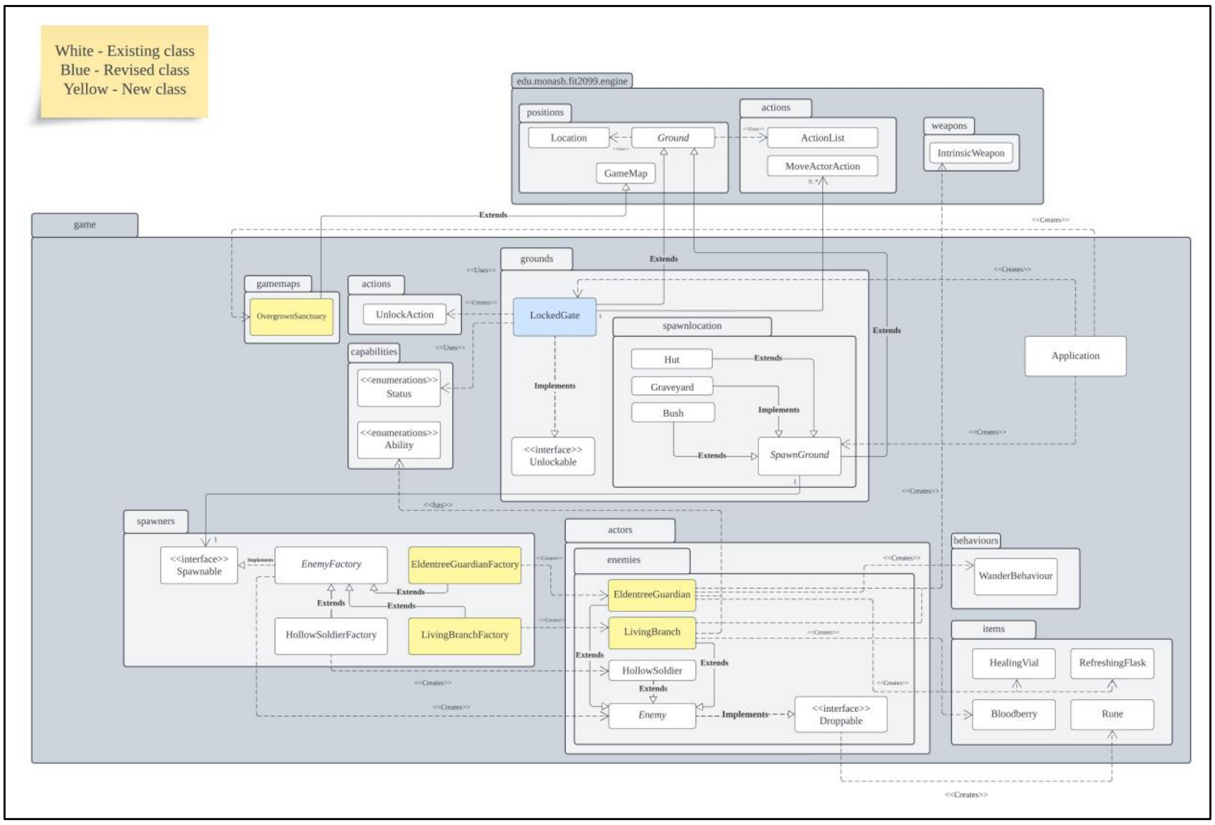
FIT2099 Assignment 3 Design Rationale

Requirement 1:



- Since the LockedGate can have multiple destinations in this requirement, the existing LockedGate class needs to be modified. In our design, we remove the original MoveActorAction attribute and replace it with an arraylist which consists of the MoveActorAction. A new method is created as well to allow the driver class to add extra destination to the locked gate (if needed). This helps to follow OCP as if there is any multiple destinations gate introduced in the future, we can simply use the new method to add those destination without modifying the existing code again.

- For the new enemies introduced in this requirement, the design will be the same as in assignment 2. What we need is just create a new enemy class extending from the abstract Enemy parent class (DRY). Then, a spawner is also created corresponding to the enemy and is pass to the spawn location (ground) through the constructor. This proves that our design in assignment 2 follows OCP as we are just extending our code without modifying any of the existing code.

Pros:

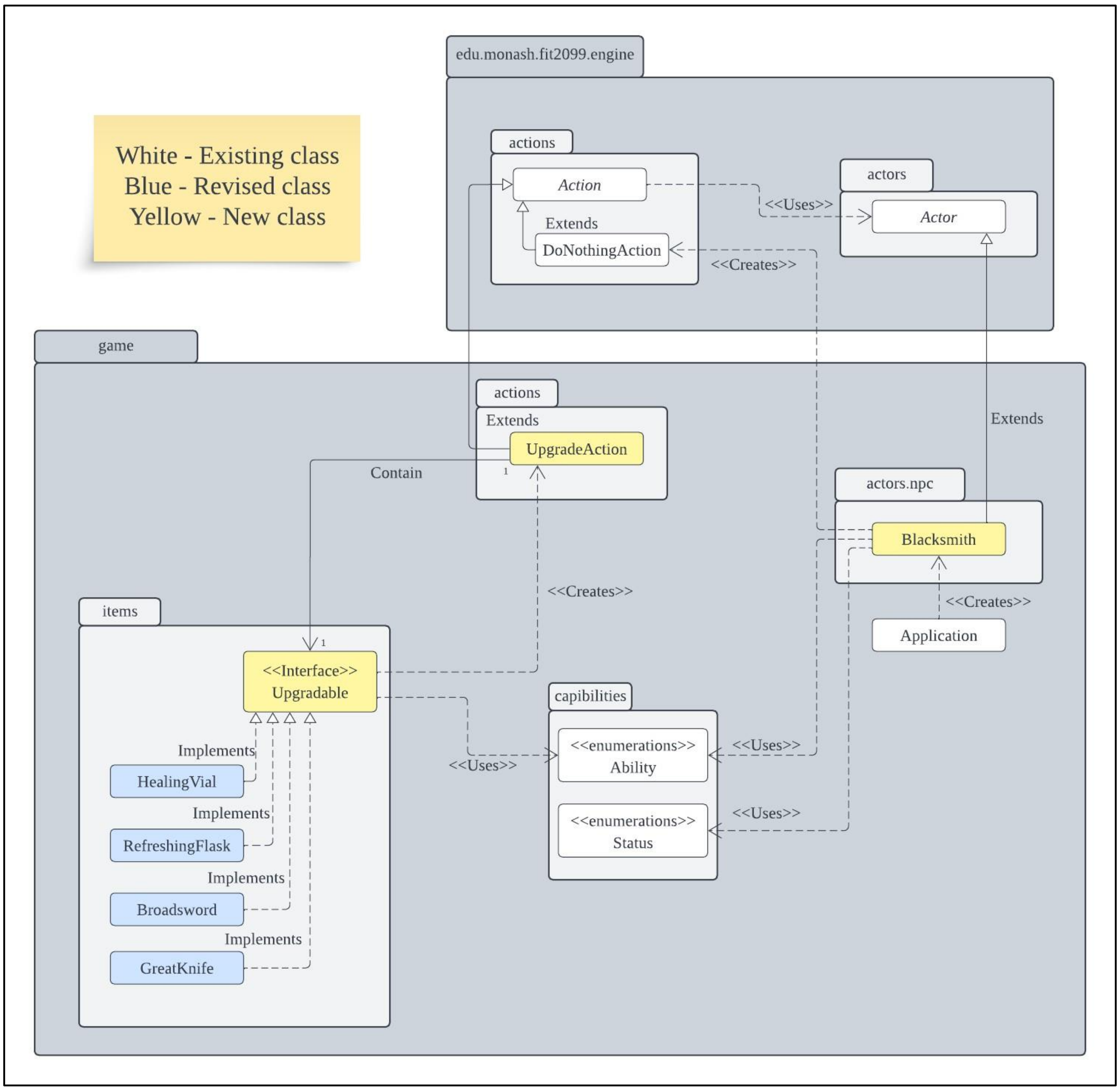
- Follows SRP. The locked gate only focuses on moving the player to a new map. The spawner only focuses on spawning the enemy and the concrete enemy class focuses on the characteristics and behaviors of the enemy.
- Easier to extend without needing to modify the existing code. Follows OCP.

- The new enemies and factories were simply implemented by only extending the classes we had before, which means we have a strong foundation for some of the SOLID principles such as OCP.

Future extensions support:

- If a new gate with multiple destination is introduced, construct a locked gate with one of the multiple destination first, then add the remaining destination to the gate using the addTravelAction method.

Requirement 2:



- In this requirement, a Blacksmith class is created to act as a blueprint for all blacksmiths in this game. The Blacksmith class is extended from the Actor class to inherit the methods from the Actor class (DRY). In this design, we don't have an abstract parent Blacksmith class, which is different with the merchant design in assignment 2. The reason is that the merchant is said to have different selling prices depending on which merchant is interacting with the player. However, for the blacksmith, we don't have this feature, and our assumption is that all the price offered by all the blacksmiths in the game is consistent and any weapons or items can be upgraded by all the blacksmiths in the game if the player have the weapons or items. This assumption follows the logic of most of the roguelike game. Hence, all the blacksmiths in this game should be instantiated from only this class. By following this design, this helps to reduce unnecessary complexity for the code as having multiple inheritance may be confusing and hard to understand for the developers. So, multiple inheritance should be avoided if we have enough reason to reject the use of it.

- In this requirement, we introduce a new interface, Upgradable. This interface is only implemented by those items or weapons that can be upgraded by the blacksmith. By introducing an interface, it helps to achieve the OCP as if there is a new items or weapons that can be upgraded, the new items or weapons can have this feature by implementing the Upgradable interface. The Upgradable interface only consists of two methods which are highly related to the upgrade feature, this follows ISP as every interface should only have methods that are applicable to all child class. Besides, by having these two methods in the interface, it enforces the child class to implement the complete code for the method with more specific details.

Pros:

- Follows OCP. Every new item or weapon that is upgradable can achieve this feature by simply implementing the Upgradable interface and complete the two methods in it.
- Easier to extend without needing to modify the existing code.
- Reduce code complexity by removing unnecessary multiple inheritance for blacksmith class (this is a concrete class compared to the merchant in assignment 2 which is an abstract class).

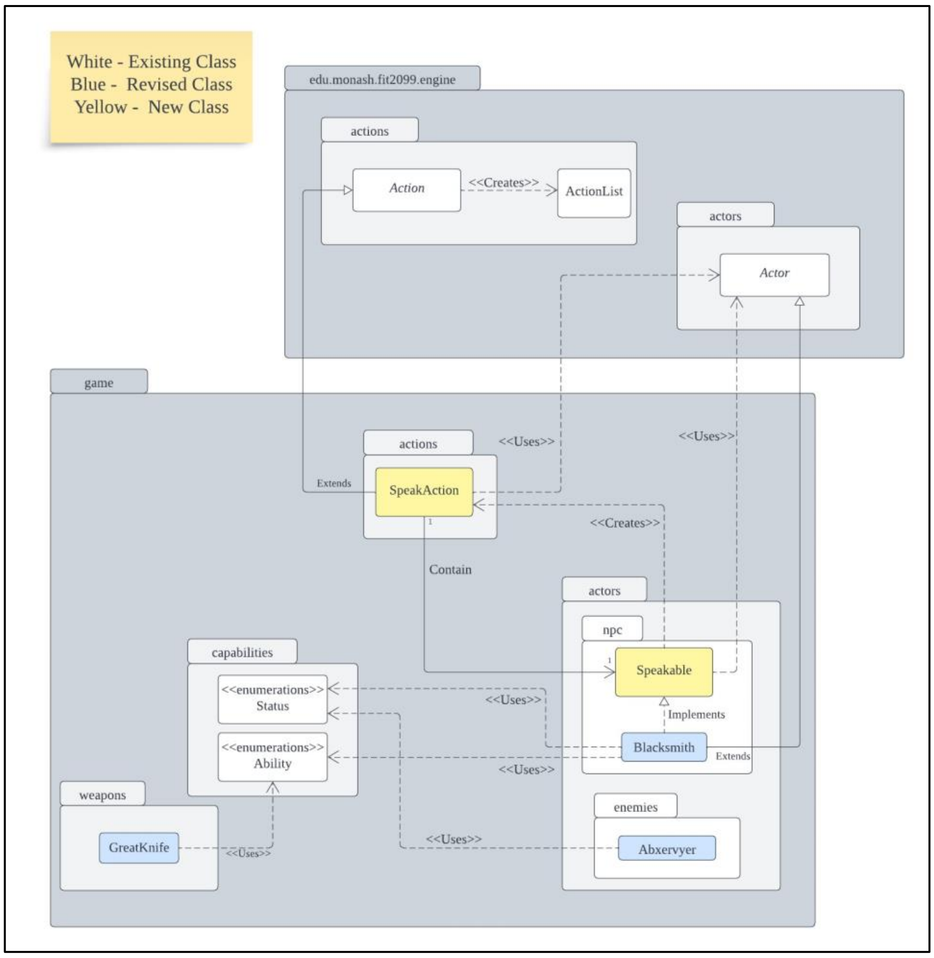
Cons:

- If every blacksmith in the game has different available upgradable item or have different upgrading cost, then the current blacksmith class should be modify to become an abstract class and the concrete blacksmith class should extends from this abstract class.

Future extensions support:

- All the blacksmiths in this game can be directly instantiated from the blacksmith class. All items or weapons that are upgradable can be upgraded by all the blacksmiths in the game.
- New items or weapons that can be upgraded by the blacksmith can achieve this feature by simply implementing the Upgradable interface. More specific details regarding the upgrade can be done in the upgrade method from the Upgradable interface.

Requirement 3:



- A new Speakable interface is introduced for this requirement. This interface will be only implemented by those npcs that can speak to the player. Having this interface, we follow SRP as this interface only focuses on the speaking feature for those npc that implements it. ISP principle is also achieved as this interface only has a speak method which can be used by all the classes that implement it. Having this interface also forces those classes that implement it to complete the speak method by providing a more specific details in the method. In the speak method, every Speakable object should implement this method with specific details such as what monologues will be returned when the player chooses to listen to them. This interface also helps us to achieve OCP as if there is any actor that can speak, they can simply just implement this interface and complete the speak method based on the condition given without modifying the existing class.

- A new ability and status are introduced for this requirement as well. The ability StabStep is used to indicate that the player carries a GreatKnife. The Abxervyer killer status is used to indicate that the player has defeated the boss. This helps us to prevent from using instanceof when choosing which monologues to be spoken by the Speakable as some monologue only exist when certain condition is met.

- A SpeakAction is introduced as well to manage the process after the player chooses to listen to any Speakable object. This follows SRP as the main responsibility of SpeakAction only focuses on performing the speaking feature of the npcs. This SpeakAction is extended from the Action class to avoid any code duplication (DRY).

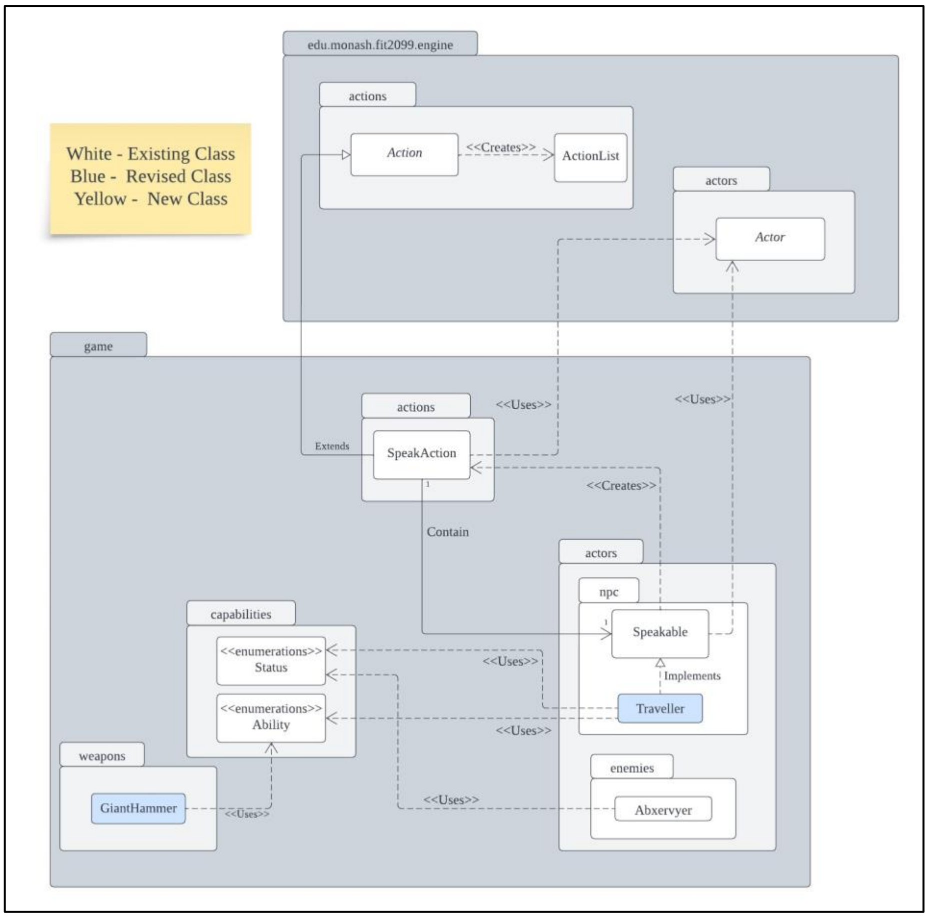
Pros:

- This design follows OCP principle as if there is a new actor that can speak, we do not need to modify the existing code but simply just make the class implement the Speakable interface.
- Follow SRP. Every new class that is introduced in this requirement only focuses on one responsibility. This helps the code to be easier to maintain.

Future extensions support:

- If there is any new actor that can speak, they can simply just implement this interface and complete the speak method based on the condition given without modifying the existing class.
- The monologues that can be spoken by the Speakable should be listed in the speak method. For those monologues that only appear when some specific conditions are met, those conditions should be specified in the speak method as well.

Requirement 4:



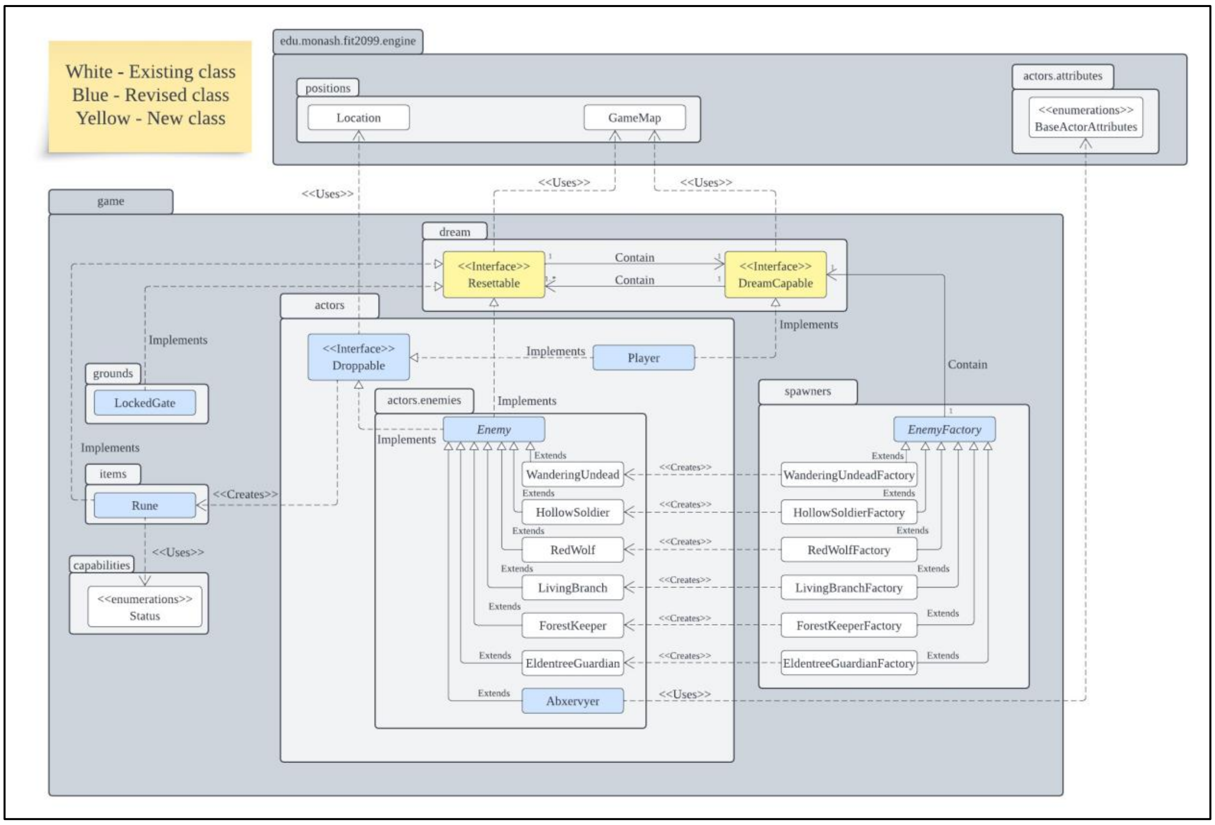
- The implementation of this requirement is pretty much the same as in requirement 3. In this case, the Traveller implements the Speakable interface as it can speak to the player. The details implementation of the monologue spoken by the Traveller is done in the speak method.

- A new ability, Giant Slam, is added. This ability is useful to check whether the player carries a Giant Hammer or not.

Pros:

- This requirement is done easily without needing to modify the existing code but just extending it for the Traveller class. This proves that our initial design follows the OCP principle.

Requirement 5:



- Similar to requirement 5 in assignment 2, for this requirement, two new interfaces, Resettable and DreamCapable, are introduced. DreamCapable interface is implemented by the player. This interface mainly focuses on managing the respawn functionality of the player (i.e., how is respawn function works and what happens after the player is respawned). Meanwhile, Resettable interface is implemented by those game entity that are affected after the player is respawned. This interface only consists of one method which requires those that implement this interface to specify the actual details when the player is respawned. By having these two interfaces, it can help to follow ISP as each interface consists of methods that are applicable to all the classes that implement it. This also follows SRP as each of the interfaces only focuses on one responsibility. This helps the developers to maintain the code easily. Besides, having the two interfaces also allows us to follow OCP as the code can be easily extended without needing to modify the existing code. For example, if the weapons on the ground will also be removed when the player is respawned, we can simply let the weapon implements the Reettable interface to obtain this functionality.

- All the Resettable contain a DreamCapable as they need to subscribe to the DreamCapable. The spawners (enemy factories) also contain a DreamCapable so they can pass it to the enemy through the enemy's constructor. Meanwhile, the DreamCapable will consist of a list of Resettables so that it can inform them to reset when the player is respawned.

- The reset method in the Resettable interface need to use the GameMap in order to remove all the enemy from the map. Meanwhile, for the rune, the GameMap cannot be used as we need to track the actual location of the rune and the only way to do this is in the tick method. Hence, when the reset method is called for the rune (or all the item type), a reset status is added to the rune. The tick method will then check whether it has the reset status every turn, if the reset status is present, the rune will be removed from the ground.

- Since the player will drop runes when he is respawned, the player will also implement the Droppable interface. The developer will only need to specify the actual details of the drop method without needing to modify the existing code. This proof that using a Droppable interface is extensible and follows OCP.

- The drop method in the droppable interface now uses Location instead of GameMap. This is because our implementation is designed to let the player respawn to the first map before dropping the runes. If the argument of the drop method is GameMap, the location of the player we get to access will be the respawn location. Hence, instead of using GameMap, we use Location as the argument to access to the dead location of the player.

Pros:

- Achieve OCP as it is easier to extend without needing to modify the existing code by having the DreamCapable and Resettable interface.
- Each interface only focuses on one responsibility (SRP) which helps to make the code easier to maintain.

Cons:

- Each of the resettables will contain a DreamCapable and the DreamCapable also consists of a list of resettables. This may cause tight coupling and cause the code to be harder to maintain.

Future extensions support:

- Implements Resettable for game entity that is affected when the player is respawned.
- For new enemy that implements Resettable, directly use the GameMap to remove the enemy from the map in the reset method.
- For new items that implement Resettable, add a reset status to the item in the reset method. Then, check for the status in the tick method for items located on the ground to decide whether to remove the item from the ground.
- If the player may drop other items in the future after respawning, implement the specific details in the drop method from the Droppable interface in the player class.