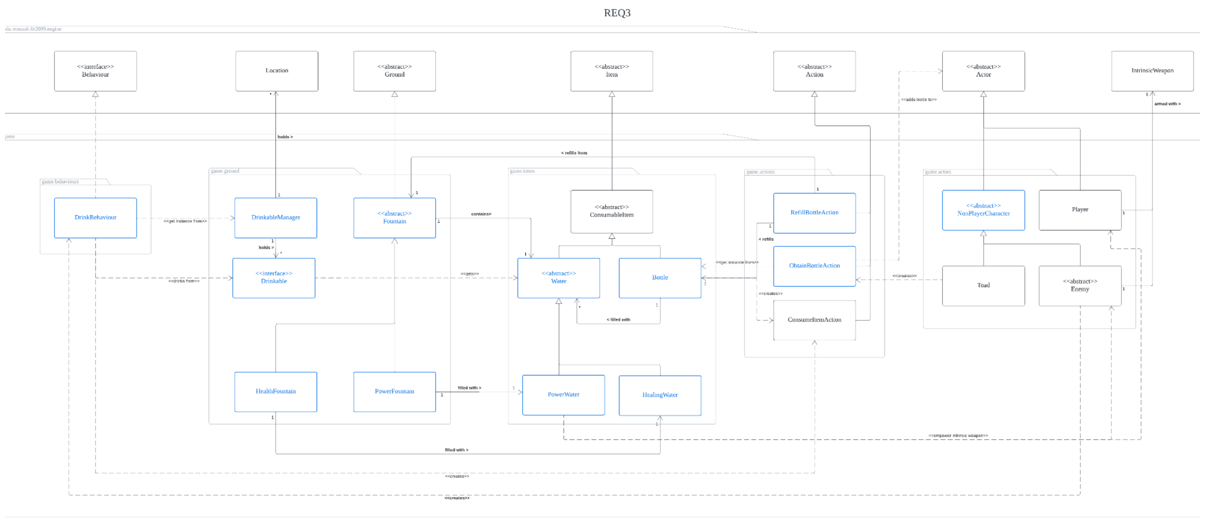
REQ3

Class Diagram:



* for clearer image refers to file in docs docs/A3/REQ3/UML_REQ3

Requirement:

- Bottle that can contain unlimited magical water. Mario can drink water in a bottle to gain a unique effect.
- Health Fountain that contains healing water which heals drinker by 50 hp
- Power Fountain that contains power water that increase drinker’s intrinsic weapon damage by 15
- Optional: Enemies standing on fountains can drink water and gain effect. Find a balance in behaviour priorities.
- Mario doesn’t have a bottle at the start of the game, he needs to obtain it from Toad.
- Fountain has limited water capacity. Refilling a bottle will use up 1 water slot, drinking water will consume 5 water slots. Once the fountain is exhausted, it will replenish in 5 turns.

Design Rationale:

In requirement 3, we make a **Bottle** class extending **ConsumableItem**. By extending it, **OCP** applies here as we don't have to repeat code for consumableItem. Initially we are not making bottles as consumable as we think it will be more reasonable to consume water and not bottle. Due to the sample output of menu description in requirement 3 which shows "mario consumes bottle[water]" and the fact that bottle is consumable is still acceptable, we extend the bottle as ConsumableItem. There are 2 attributes in Bottle, waterStack which stores a stack of water and static Bottle instance. We make a private constructor and public `getInstance()` method which returns the bottle instance. The reason behind this implementation is because accessing Bottle in Actor's inventory will require casting to call the method of Bottle that Item does not have access to. This applies **DIP** because high level modules (Item) in inventory do not need to depend on low level modules (Bottle). When obtaining Bottle from inventory we do not need to check if Item instanceOf Bottle, we access the method in Bottle. Bottle would depend on the abstraction on ConsumableItem and Item now, Item does not need to depend on the details of Bottle. Inside Bottle, we would have `getWater()`, `fillWater()`, `consumeWater()`, `consumeItem()` and `allowableActions()` which add consumeItemAction for Bottle. Talking about Bottle, we can relate it to **ObtainBottleAction** and **RefillBottleAction**. In ObtainBottleAction, it will add the Bottle instance to the actor's inventory. This applies **SRP** as this class has only 1 responsibility which is to add the bottle into inventory. This class also follows **LSP**. In the actor.addItemToInventory(x) method, x is expected to be an Item object. According to LSP, it can accept subclass objects of Item class, which means adding a Bottle instance into inventory is acceptable. The ObtainBottleAction will be added in **Toad** class when a player which has no bottle in inventory is beside Toad. In RefillBottleAction, there are 2 attributes, fountain and bottle. It follows **SRP** too because it only does 2 things in execute() method, decreasing water in fountain and adding water into bottle.

Following our design we have a **Water** class which extends **ConsumebleItem**. We also have **PowerWater** and **HealingWater** extending Water class. **OCP** is reflected in the design here. By using inheritance and abstraction, it is very easy to add new water without changing the implementation of other water or items. Water class override constructor from ConsumableItem. Healing Water and PowerWater inherit constructors and override `consumeItem()` method from ConsumableItem. In healing water implementation, consumeItem() would heal player by 50hp. For PowerWater, we check if actor has the capability of `Status.HOSTILE_TO_ENEMY.` If yes, we would cast to player `((Player) actor)`, else we cast to enemy `((Enemy) actor)` and then call `empowerIntrinsicWeapon()` method. In player and enemy classes, we would make an **intrinsicWeapon** attribute and an `empowerIntrinsicWeapon()` method which increase damage by 15. We have our only use case of casting in this method under the limitation that engine code is unchanged. This is because we need to keep track of number of times the player or enemy consume powerWater or keep track of how strong the intrinsic weapon has grown into. So we need to have an attribute in player and enemy classes. However, the actor class has no access to intrinsicWeapon attribute and `empowerIntrinsicWeapon()` method, so casting is needed here. It is possible to prevent casting here by making an ableToEmpower interface and a manager. But this would just add all the player and enemy into the manager and we think that the effort is not worth to put some many extra work on adding all player and enemy to the manager and maintaining them to prevent small use case of casting in the current stage.

Moving on, we have **Fountain** class extending ground, followed by **HealthFountain** and **PowerFountain** extending the Fountain. This follows **OCP** as adding a new type of fountain is easy and will not affect codes for other fountain or any other parts. This also implements **DRY** principle because we have 4 attributes and 9 methods in fountain class. They will be inherited or overridden by HealthFountain and PowerFountain. This reduces repetition of code by a lot. Inside fountain class, we have water, waterAmount, capacity and turnsToReplenish attributes. During initialization in the constructor we will have waterAmount set to 10, capacity set to 10 and turnsToReplenish set to 5. In the tick() method overridden, we would check if the waterAmount is 0. If it is, we would start the counter of turnsToReplenish by decrementing the counter. Once turnsToReplenish reaches 0, we set the turns back to 5 and call `replenishFountain()`method which sets waterAmount to maximum capacity. In the allowableActions, we check if the current location contains an player, if the player has a bottle in inventory and if the waterAmount of fountain > 0. If all conditions are passed, we would call refillWater() and add a refillBottleAction(fountain, bottle). RefillWater() is a method to add a new instance of water to fountain as long as waterAmount > 0. It is an abstract method in fountain. It is overridden in PowerFountain to return new instance of PowerWater and overridden in HealthFountain to return a new instance of HealingWater.

We also have a drink() method which will call refillWater() and decrease waterAmount of fountain by 5, which would be used when enemy drinks water directly on fountain.

Next, we have **DrinkBehvaviour** which would be executed by enemy only. The drinkBehaviour will be added to enemy during initialisation of enemy by calling constructor of DrinkBehaviour class. For all behaviours, there are priorities and the priorities list including drinkBehaviour is as speak > attack > follow > drink > wander. In drinkBehaviour getAction(actor, gameMap) method, our team faces issue as our original implementation is to check if the location is an instanceOf fountain, then cast the location to get the fountain and check if the fountain has enough water. If yes, we would call drink() method in fountain class and add consumeItemAction of the water. To prevent this, we added **Drinkable** interface and **DrinkableManager** class. Fountain would implements Drinkable interface. Drinkable interface has 3 methods, getWaterAmount(), getWater() and drink() where all of them have no implementations and need to be overridden. This follows **OCP** as adding new drinkable Ground is easy due to the abstraction. This also applies **ISP** as the interface is kept as small as possible to have only the necessary features of a drinkable ground. In the application driver class, we would initialize one powerFountain and one HealthFountain and add it into DrinkableManager. Inside DrinkableManager we have a static instance of manager. We also have a hashMap which records location of drinkable ground as key and drinkable ground as value. With these implementations, instead of using instanceOf to check if the location is fountain in drinkBehaviour, we would call `DrinkableManager.getInstance().getDrinkableGroundCollection().containsKey(map.locationOf(actor))` to check the location. Then get the drinkableGround from the hashMap in drinkableManager. Check if the water is enough using drinkable.getWaterAmount() and use drink() method to refill and decrease water in drinkable ground. This implementation follows **DIP** as high level modules (DrinkableManager) does not depends on low level modules (PowerFountain and HealthFountain). Drinkable ground does not need to depends on details of fountain, fountain would depends on Drinkable ground via abstraction.