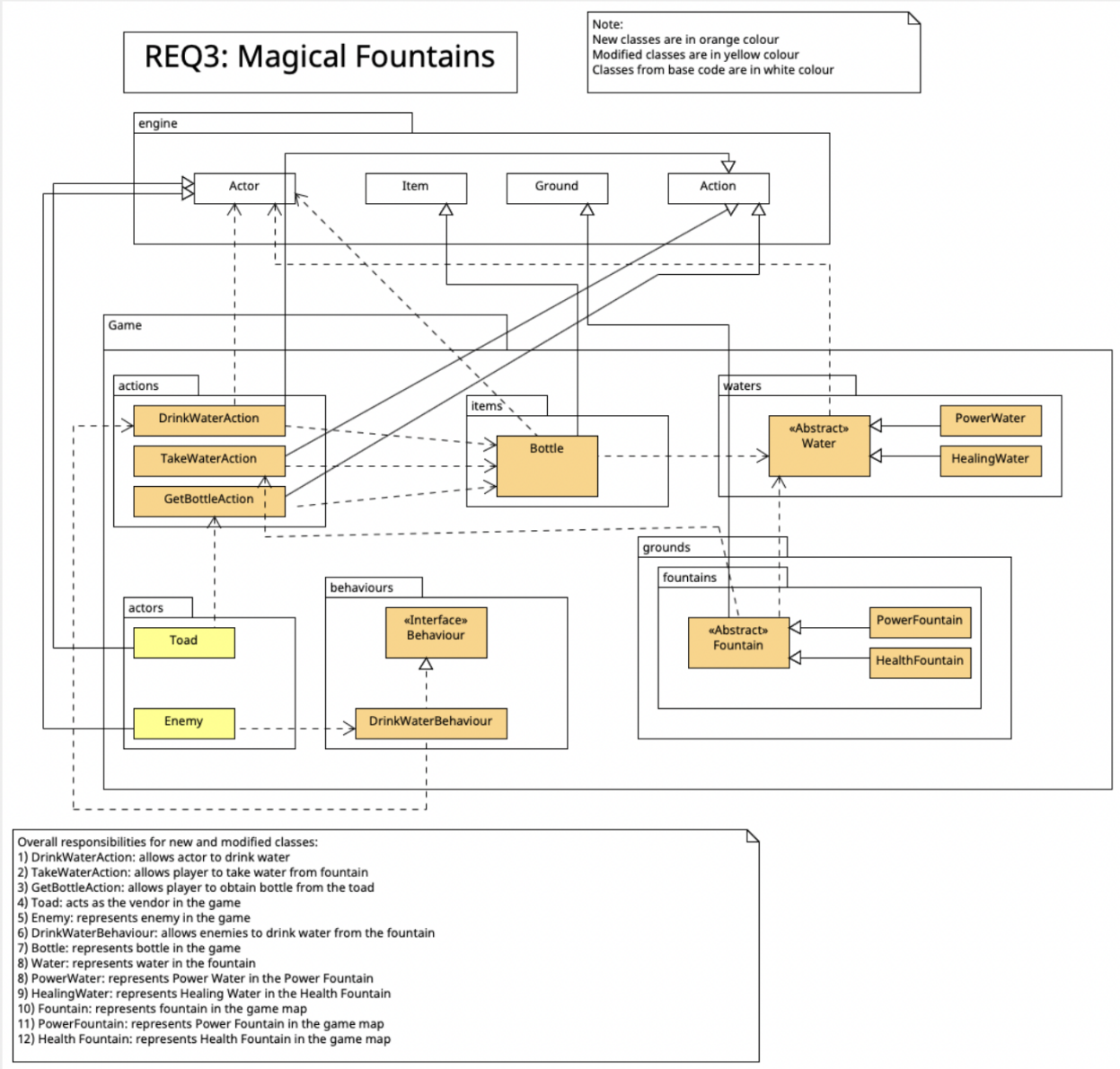
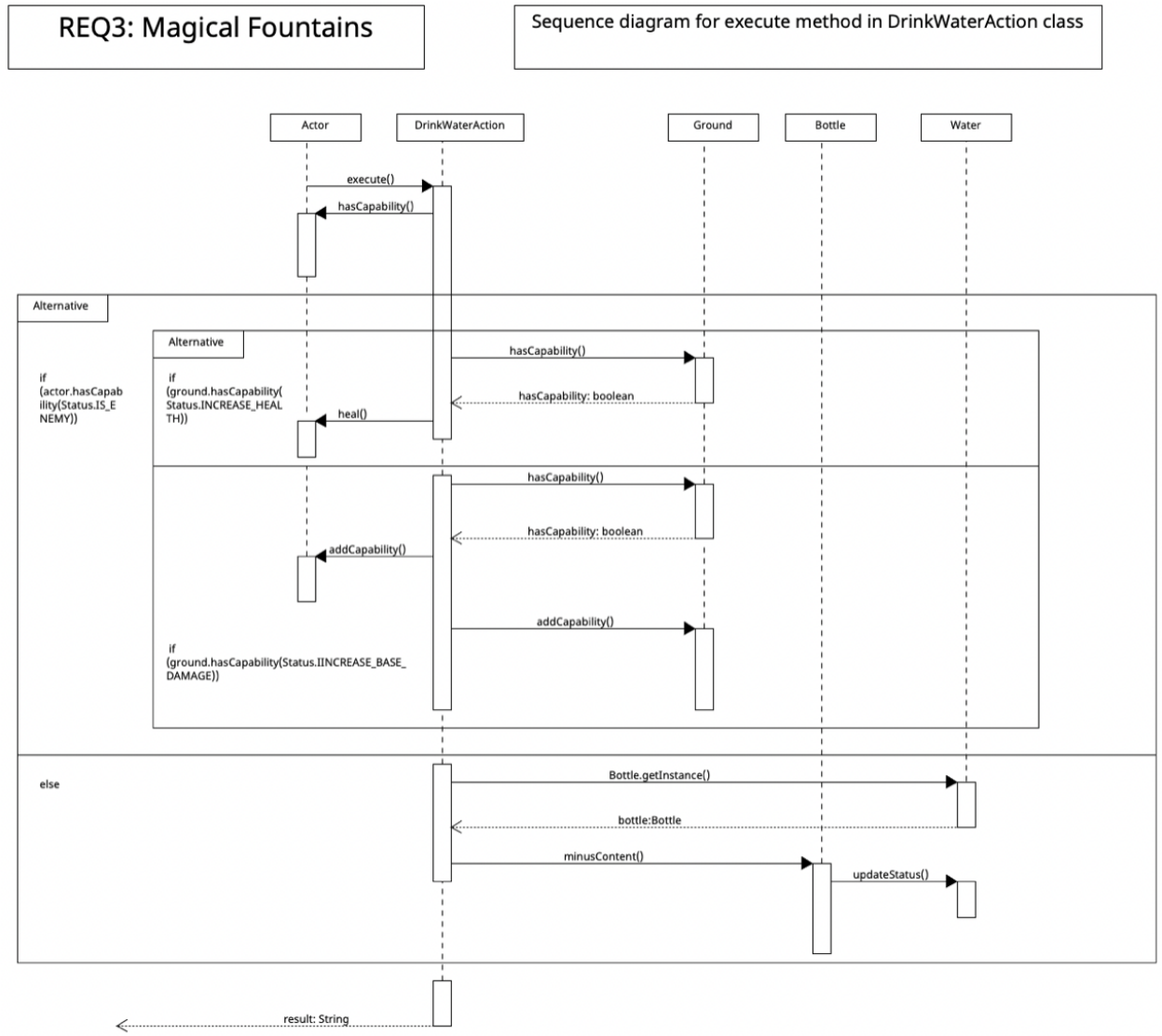
REQ 3: Magical fountains (with optional challenge)





1) DrinkWaterAction class

This class which extends Action class is to enable player to drink water from the bottle and enemy to drink water from fountain. The water drank will minus one water slot from player's bottle.

Description of method:

- execute method is a method overridden from its parent class. It will check if the actor is enemy. if actor is an enemy, the actor can gain its effect straightaway from the fountain. Else, water will be added into bottle (if actor is a player). A description will then be display in the console showing what water is drank of which actor.

```
@Override
    public String execute(Actor actor, GameMap map) {
```

```
        String result= "";
        if (actor.hasCapability(Status.IS_ENEMY)){
            if (ground.hasCapability(Status.INCREASE_HEALTH)) {
                actor.heal(HEALED_HP);
                result = actor + " drank Healing water";
            } else if (ground.hasCapability(Status.INCREASE_BASE_DAMAGE)) {
                actor.addCapability(Status.POWER_WATER);
                result = actor + " drank Power water";
            }
            ground.addCapability(Status.DRANK_BY_ENEMY);

        } else {
            result = actor + " drank " + Bottle.getInstance().getLast();
            Bottle.getInstance().minusContent(actor);
        }
        return result;
    }
```

why i choose to do it that way:

By doing this, we are adhering to the Single Responsibility Principle(SRP) as this class is only in charge of allowing actor to drink water from fountains .

2) TakeWaterAction class

This class which extends Action class is to enable player to obtain water from the fountain. The water obtained will add one water slot into player's bottle.

Description of method:

- the execute method is to add one slot of water into player's bottle when player choose to take water from the fountain by obtaining the water returned from the menuDescription method.

```
@Override
    public String execute(Actor actor, GameMap map) {
        if (actor.hasCapability(Status.HAS_BOTTLE)) {
            Bottle.getInstance().addContent(water);
            fountain.minusContent();
        }
        return menuDescription(actor);
    }
```

Why i choose to do it that way:

By doing this, we are adhering to SRP as this class is only in charge of allowing player to refill water from fountains.

3) GetBottleAction

This class which extends Action class is to enable player to obtain bottle from the toad. The bottle obtained will be added into player's inventory

Description of method:

- execute method is to add bottle into player's inventory after player obtain bottle from the toad

```
public String execute(Actor actor, GameMap map) {
        actor.addItemToInventory(bottle);
        return menuDescription(actor);
    }
```

why i choose to do it that way:

By doing this, we are adhering to SRP as this class is only in charge of allowing player to obtain bottle from the toad.

4) Bottle class

This class represents bottle in the game. This bottle can contain unlimited magical waters. This bottle will be a permanent item that cannot be dropped or picked up. This bottle can be filled with water, and Mario can drink/consume the water inside the bottle. Each water will give a unique effect depending on its original source (fountains). Mario doesn't have a bottle in his inventory at the start of the game. Instead, Mario will obtain it from Toad.

Description of method:

- addContent method to add one slot of water into bottle
- minusContent method to minus one slot of water from bottle

```
public void addContent(Water water) {
        this.content.add(water);
    }

public void minusContent(Actor actor) {
        if (actor == null){
            throw new IllegalArgumentException("The input parameter (i.e., actor) cannot be null");
        }

        Water drankWater=content.get(content.size()-1);
        drankWater.updateStatus(actor);
        this.content.remove(drankWater);
    }
```

Why i choose to do it that way:

Since there can only be one bottle in the game, static factory method is used in Bottle class so that the constructor can be called directly instead of creating a new instance of it. By doing this, encapsulation can be use as behaviour of the Bottle class can be specify via parameters without needing to know the internal implementation of the Bottle class. This improves code readability. For example, content of the bottle can be obtained by `Bottle.getInstance.getContent()`.

Furthermore, exception is added into the minusContent method order to implement the Fail Fast principle, so that when the actor is null, the program will throw an error and stop running right after instead of continue to run until water is to be consume from the bottle. By doing this, this eases debugging process as it will be easier to identify the cause of error.

5) Water class

This is an abstract class that represents water in the fountain. Each water from a fountain provides different effects that will help Mario in his journey. Water class is an abstract class created to be the parent class of HealingWater and PowerWater classes.

Why i choose to do it that way:

This class is made to be an abstract class because this class is not use to create objects to represent anything in the game and this class is not instantiated throughout the code. Instead, it is to be inherited by HealingWater and PowerWater classes.

Since HealingWater and PowerWater are waters, they both share some same methods and hence a Water class is created to reduce duplicated codes as they can access method from Water class by calling super. By doing this, we are adhering to the Don’t Repeat Yourself (DRY) principle.

Open Closed Principle (OCP) can also be implemented as we are able to add new functionalities to the water without changing existing codes in the Water class. For example, if they exist a water which have exceptionally more functions, that water class can extend this Water class, implementing all methods from Water class, and override or create additional methods in it for additional functionalities.

Similar to as in the Bottle class, exception is added in order to implement the Fail Fast principle.

6) PowerWater class

This class represents power water that can be obtained from power fountain in the game. When the water is consumed, it increases the drinker's base
/intrinsic attack damage by 15.

Description of method:

- updateStatus method to add POWER_WATER capability to drinker.

```
public void updateStatus(Actor actor) {
        super.updateStatus(actor);
        actor.addCapability(Status.POWER_WATER);
    }
```

Why i choose to do it that way:

As explained in the Water class section, this class is extended from Water class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Water class.

7) HealingWater class

This class represents healing water that can be obtained from health fountain in the game. Drinking this water will increase the drinker's hit points/healing by 50 hit points.

Description of method:

- updateStatus method to heal drinker by 50 hp upon consumption

```
@Override
    public void updateStatus(Actor actor) {
        super.updateStatus(actor);
        actor.heal(HEALED_HP); // HEALED_HP = 50
    }
}
```

Why i choose to do it that way:

As explained in the Water class section, this class is extended from Water class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Water class.

8) Fountain class

This class represents fountain in the game. Mario can refill the bottle when he stands on the fountain. A fountain produces an endless amount of water. Each water from a fountain provides different effects that will help Mario in his journey. Fountains have limited amount of
water(10 slots of water capacity).

Description of method:

- allowable action method to allow player to refill water from the fountain if the fountain is not empty.

```
@Override
    public ActionList allowableActions(Actor actor, Location location, String direction) {
        if (location.containsAnActor() && actor.hasCapability(Status.HAS_BOTTLE) && !this.isEmpty()) {
            return new ActionList(new TakeWaterAction(this.getWater(),this));
        }
        return new ActionList();
    }
```

- tick method to keep track of number of turns in the game and minus one slot of water whenever water in it is drank my enemies or refilled by players. When content of the fountain runs out, the “turnWhenWaterRunOut” will be use to store that particular current turn, and content will be set to -1 to indicate that the fountain is empty. Content of the fountain will then be reset to 10 five turns after “turnWhenWaterRunOut”.

```
@Override
    public void tick(Location location) {
        this.currentTurn++;

        if (this.hasCapability(Status.DRANK_BY_ENEMY)){
            this.minusContent();
            this.removeCapability(Status.DRANK_BY_ENEMY);
        }

        if (this.content == 0){
            this.addCapability(Status.IS_EMPTY);
            this.turnWhenWaterRunOut =this.currentTurn;
            this.content=EMPTY_INDICATOR;
        }

        if (this.content==EMPTY_INDICATOR && (this.currentTurn - this.turnWhenWaterRunOut ==REPLENISH_TURN)){
            this.turnWhenWaterRunOut =INITIAL_TURN;
            this.setContent(MAX_CONTENT);
            this.removeCapability(Status.IS_EMPTY);
        }

    }
```

Why i choose to do it that way:

Similar to Water class, this class is made to be an abstract class because this class is not use to create objects to represent anything in the game and this class is not instantiated throughout the code. Instead, it is to be inherited by HealthFountain and PowerFountain classes.

Since HealthFountain and PowerFountain are fountains, they both share some same methods and hence this Fountain class is created to reduce duplicated codes as they can access method (such as allowableActions, tick, setContent, minusContent etc methods) from Fountain class by calling super. By doing this, we are adhering to the Don’t Repeat Yourself (DRY) principle.

Open Closed Principle (OCP) can also be implemented as we are able to add new functionalities to the fountain without changing existing codes in this class. For example, if they exist a fountain which have exceptionally more functions and characteristic, that class can extend this Fountain class, implement methods from Fountain class, and override or create additional methods in it for additional functionalities.

9) HealthFountain class

This class represents health fountain (’H’) in the game. This fountain contains HealingWater which players can refill it into the bottle while enemies can consume it straight away from the fountain.

Why i choose to do it that way:

As explained in the Fountain class section, this class is extended from Fountain class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Fountain class.

10) PowerFountain class

This class represents power fountain (’A’) in the game. This fountain contains Power Water which players can refill it into the bottle while enemies can consume it straight away from the fountain.

Why i choose to do it that way:

Similar to HealthFountain, this class is extended from Fountain class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Fountain class.

11) DrinkWaterBehaviour class

This class is to enable enemies to drink water from magical fountains. it implements the Behaviour class.

Description of method:

- getAction method checks if enemy is located on a fountain. If yes, allow enemy to drink water from the fountain by returning the DrinkWaterAction.

```
@Override
public Action getAction(Actor actor, GameMap map) {
Ground currentGround = map.locationOf(actor).getGround();
    if(currentGround.hasCapability(Status.IS_FOUNTAIN)){
return new DrinkWaterAction(currentGround);
}
return null;
}
```

Why i choose to do it that way:

By doing this, we are adhering to the Single Responsibility Principle(SRP) as this class is only in charge of allowing enemy to drink water from fountains, by returning a DrinkWaterAction to enemies if the ground they are located on is a fountain.