A3 Design Rationale

Work Breakdown Agreement

Lab 12 Team 4

Group Member:

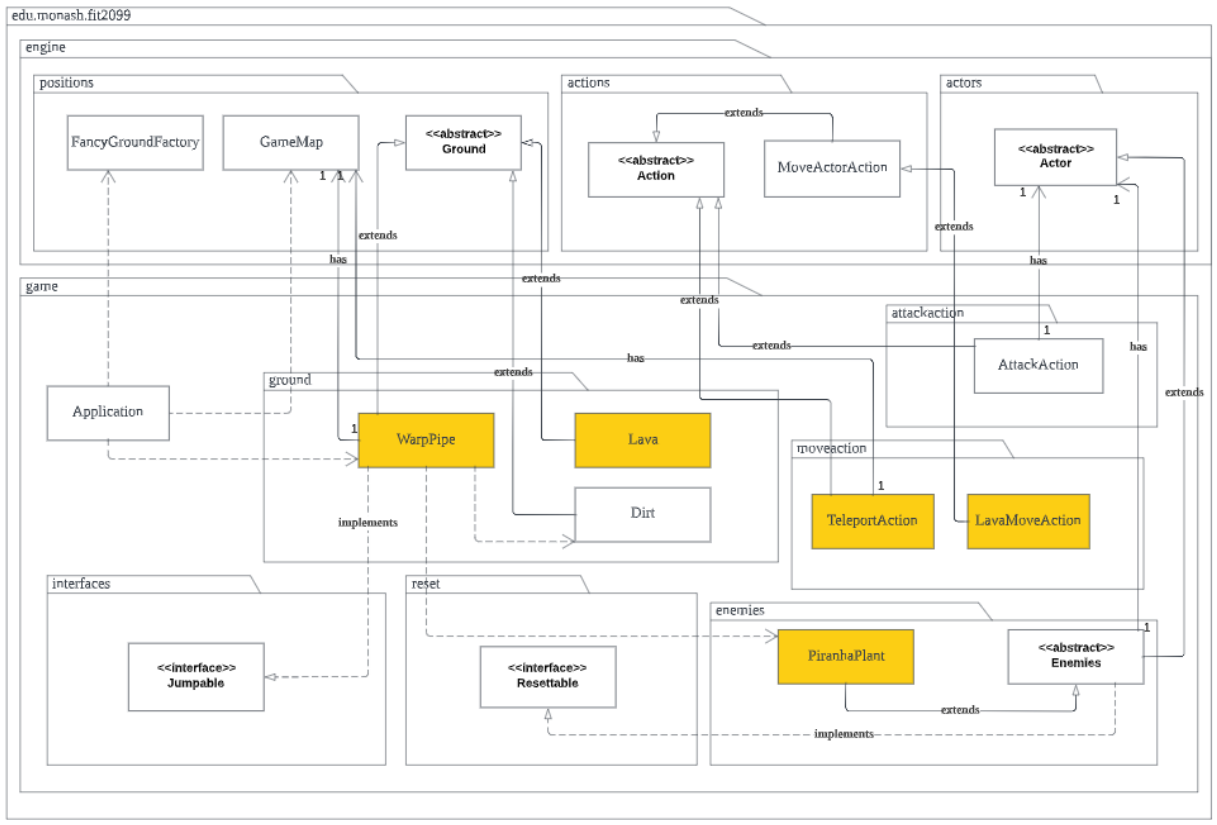
1. Seng Wei Han (32229070)
2. Chua Jun Jie (32022425)

Work Distribution

Name	Person In Charge	Status
REQ2, REQ3	Chua Jun Jie	✓
REQ1, REQ4	Seng Wei Han	✓

REQ1: Lava zone 🔥

REQ 1 CLASS DIAGRAM

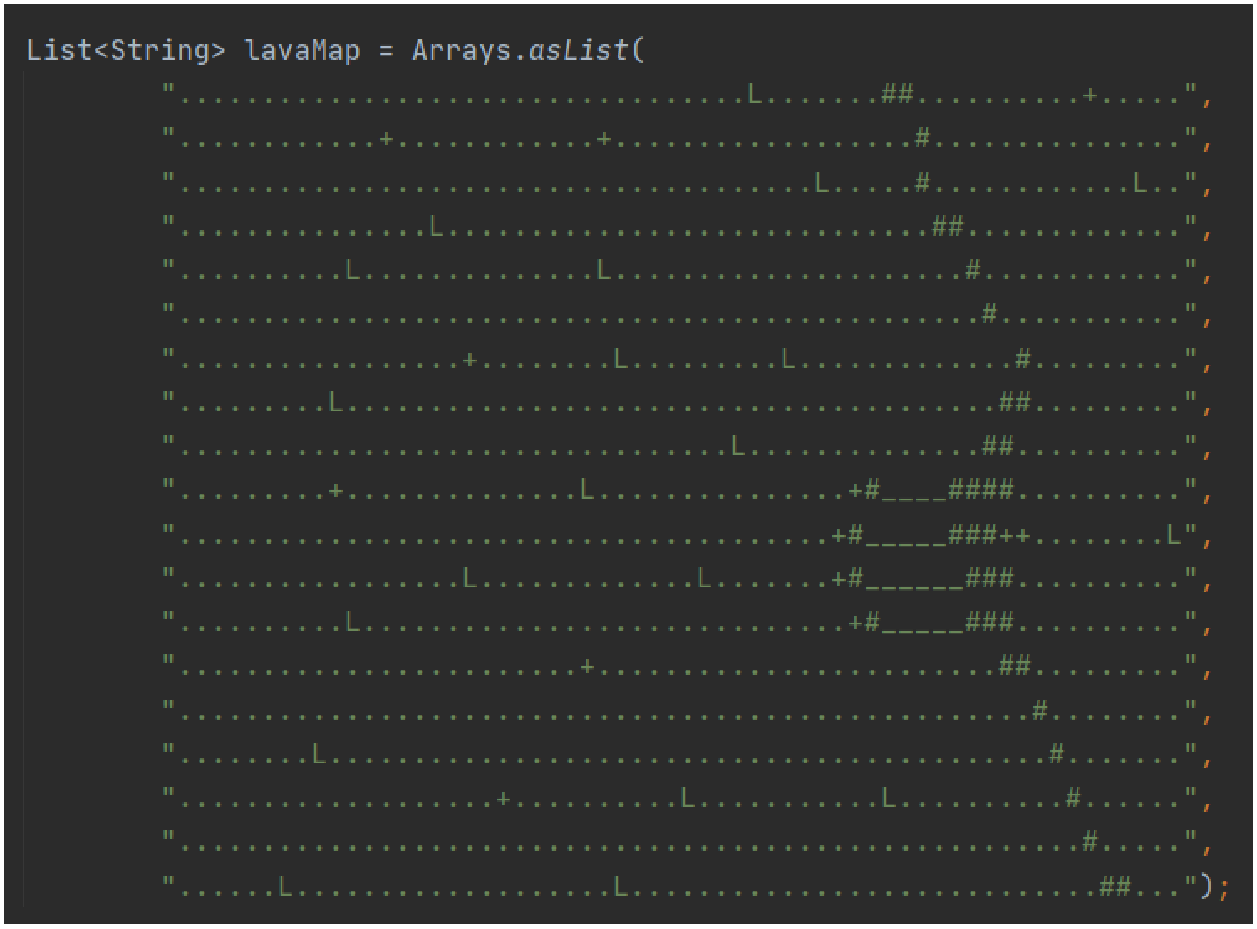


1. New Map 🗺️

Design Implementation:

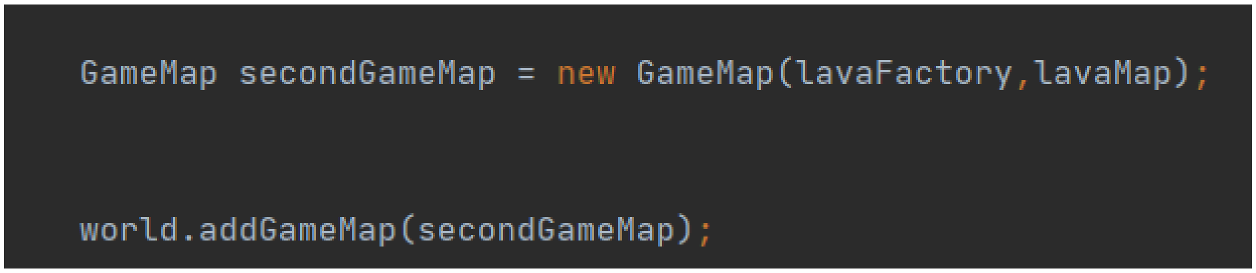
```
FancyGroundFactory lavaFactory = new FancyGroundFactory(new Dirt(),new Sprout(),new Wall(),new Lava(),new Floor());
```

Before we create the lava map, we first need to declare an instance of FancyGroundFactory and initialise all the instances of ground that will be present on the lava map into the constructor of FancyGroundFactory.



```
List<String> lavaMap = Arrays.asList(
        "..................................L.......##..........+.....",
        "............+............+..................#...............",
        "......................................L.....#............L..",
        "...............L.............................##.............",
        "..........L..............L.....................#............",
        "................................................#...........",
        ".................+........L.........L.............#.........",
        ".........L.......................................##.........",
        ".................................L..............##..........",
        ".........+..............L...............+#____####..........",
        ".......................................+#_____###++........L",
        ".................L.............L.......+#______###..........",
        "..........L.............................+#_____###..........",
        "........................+........................##.........",
        "...................................................#........",
        "........L...........................................#.......",
        "...................+..........L...........L..........#......",
        "......................................................#.....",
        "......L...................L............................##...");
```

Then, we can start to draw the lava map as shown above which in this case is slightly smaller as compared to the size of the original map. We also hardcoded the lava ground into the list of strings by randomly placing them. The next step is then to create an instance of the GameMap and finally add them into the world as shown below :



```
GameMap secondGameMap = new GameMap(lavaFactory,lavaMap);

world.addGameMap(secondGameMap);
```

Lava Class

1. A new blazing fire ground that can only be found in the Lava map.

2. Since enemies are not able to step on this lava ground, then we can set the method canActorEnter into false.
3. Has a status of BURN to notify that the current ground will inflict a damage of 15 to player who steps on them.

LavaMoveAction class

1. A moving action that will be triggered when the player is facing a lava ground.
2. When the player steps on that lava ground, 15 hit points will be deducted from the total hit points of the player.

2. Teleportation (Warp Pipe) C 🌀

WarpPipe class

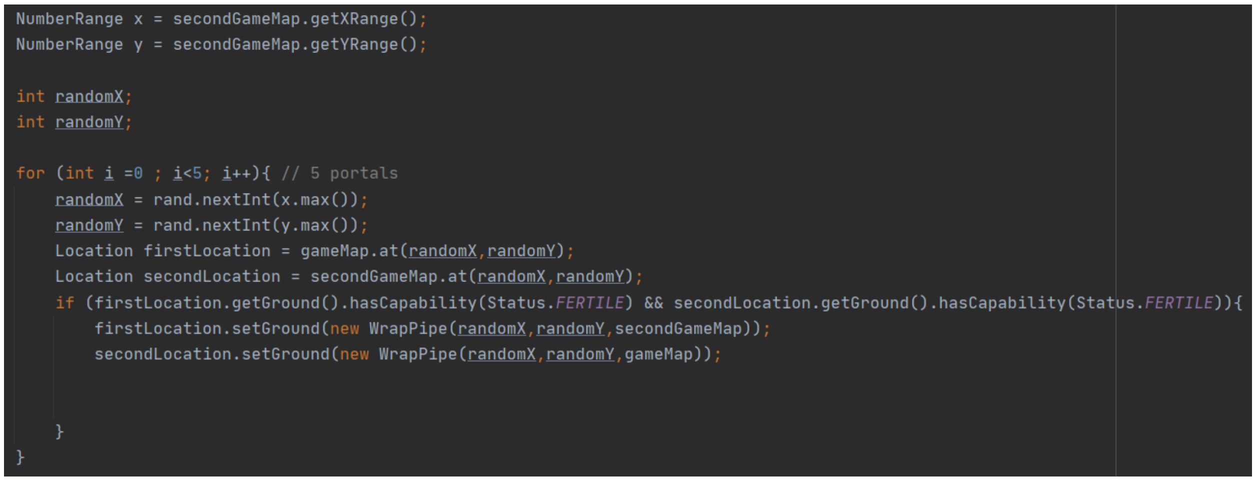
1. A type of ground that allows the player to teleport between two different maps when the player is standing on this pipe.
2. A piranha plant will be spawned on this pipe at the second turn of the game. Therefore, we can use a counter that will increment by one each time when the tick method is being called. Once the counter reaches two, then a new piranha plant will be spawned on top of this pipe.
3. In order to teleport , the player will need to jump on the pipe, hence this class will need to implement the Jumpable interface's method in which the method will be called when the player is facing a WarpPipe and at the same time there's no actor on top of the pipe. By implementing the interface's method, we are adhering to the Open-Closed Principle because we do not need to modify the existing code in the JumpAction when there's a new ground that is jumpable.

Design Implementation:

Warp Pipe

1. In my implementation, I created the same number of Warp Pipe in both maps. Firstly, I created 2 warp pipes , one for each map and placed them at the position of x = 0 and y = 0 which is at the top left corner of the map. Then, I created a for loop that will run for 5 times to restrict the number of warp pipes in each map to be only 5

excluding the one that we created initially, as well as to randomly place the warp pipe at the same position in both maps which is as shown by the code below :



```
NumberRange x = secondGameMap.getXRange();
NumberRange y = secondGameMap.getYRange();

int randomX;
int randomY;

for (int i =0 ; i<5; i++){ // 5 portals
    randomX = rand.nextInt(x.max());
    randomY = rand.nextInt(y.max());
    Location firstLocation = gameMap.at(randomX,randomY);
    Location secondLocation = secondGameMap.at(randomX,randomY);
    if (firstLocation.getGround().hasCapability(Status.FERTILE) && secondLocation.getGround().hasCapability(Status.FERTILE)){
        firstLocation.setGround(new WrapPipe(randomX,randomY,secondGameMap));
        secondLocation.setGround(new WrapPipe(randomX,randomY,gameMap));


    }
}
```

Reasons and justifications for such implementation :

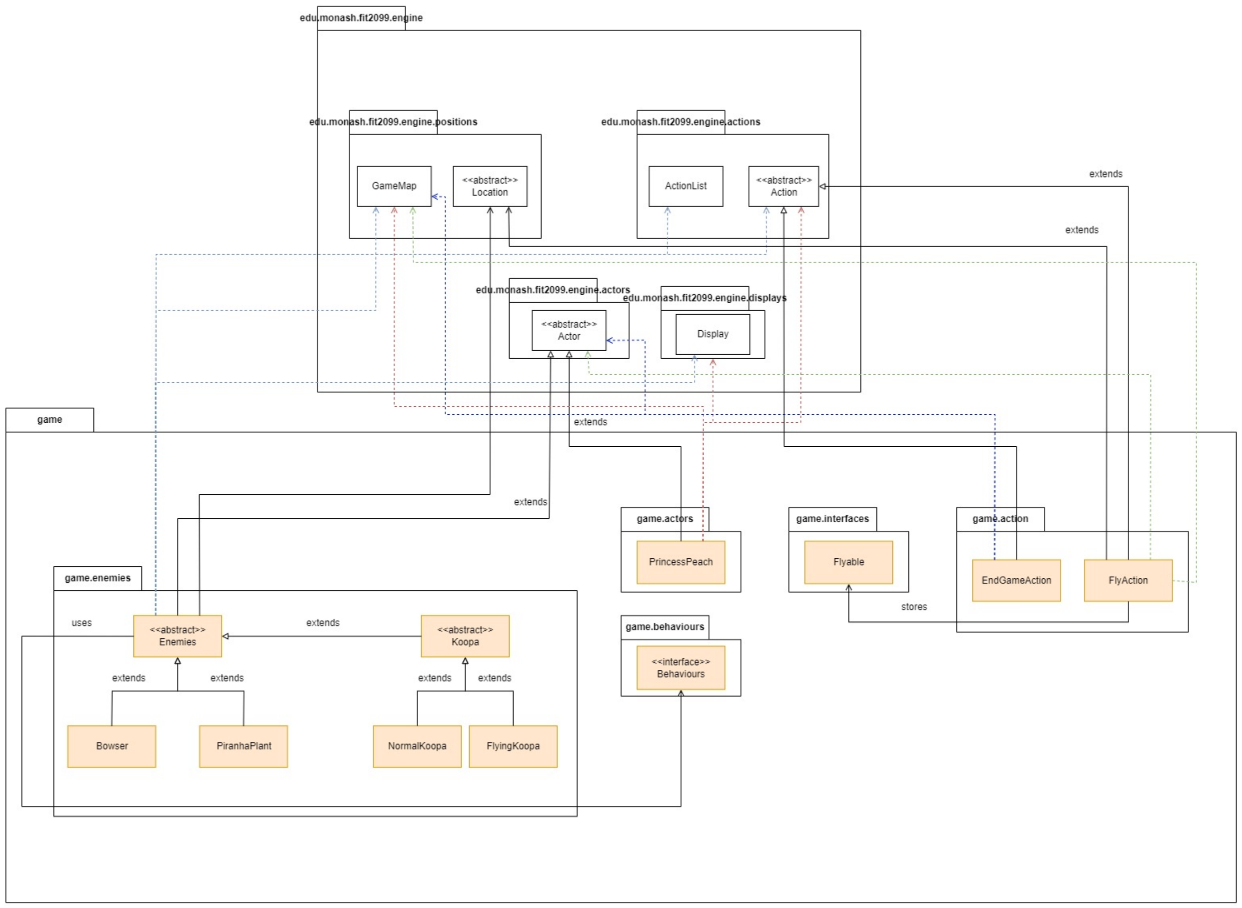
If we only allow the presence of one warp pipe in the second map (lava map) and imagine if mario (player) consumes the power star and destroy the only warp pipe in the second map, then the player will not be able to teleport back to the original map, instead the player will be stuck forever in the second map. Hence, by using my implementation of initialising the same amount of warp pipe in both maps, somehow give the player a chance to teleport back to the original map if one of the warp pipe is destroyed because there will still be other warp pipe available for mario to use it and teleport back to original map or vice versa.

TeleportAction class

1. An action that a player can take to teleport to another map when the player is standing on top of the Warp Pipe.
2. The execute method in the TeleportAction class will first check whether the target map that the player is going to teleport to contains an actor such as piranha plant, if it does then the actor will be removed from the map to allow the player to be teleport to the exact location of the warp pipe since a ground can only contain one actor. If the target map location of the warp pipe does not contain any actor , then the player will be removed from the current map and added to the target map.

3. By having this TeleportAction class, we are adhering to the Single-Responsibility Principle since this class is only responsible for teleporting a player between two maps and nothing else.

REQ2: More allies and enemies!



Implementation

Princess Peach (P)

1. Princess Peach is an actor that does nothing throughout the game. This is achieved by simply creating a PrincessPeach class and extending it from Actor abstract class. Then, its playTurn method will return a DoNothingAction. It’s allowableAction will contain an if-statement to check if the player has a key: if yes, player can end

the game by saving Princess Peach; else, player does not have the option to end the game.

Bowser (B)

1. Bowser is the final boss of the game that player needs to defeat in order to save Princess Peach. Thus, first of all the Bowser class is extended from Enemies class, because its an Enemy to player. Bowser does nothing until player is in its exits, thus, inside Bowser’s constructor, its behaviour TreeMap is cleared. Bowser is also given a special status called FINAL_BOSS.
2. Bowser has 500HP and deals 80 damage on successful attack. This is achieved by presetting its base HP in constructor, and overriding the getInstrinsicWeapon method to change its damage.
3. If Bowser lands a successful attack, it will also set the ground on fire. This is achieved by putting an if-statement inside NPCAttackAction to check if the actor using NPCAttackAction is Bowser using status: if yes, it will set the ground that the target is at on fire.
4. Upon resetting the game, Bowser will be healed to maximum and it will be moved back to its original position, beside Princess Peach. This is simply achieved by overriding resetInstance method.
5. Upon defeating Bowser, it will drop a key that allows the player to win the game by saving Princess Peach.

Key (k)

1. Key is an item in the game, thus it extends from Item abstract class. It is portable because it must be picked up from the player in order to win the game by saving Princess Peach.
2. It has a capability of WIN_GAME. This capability is used inside Princess Peach to check if the player has the key in his inventory.

Piranha Plant (Y)

1. Piranha Plant is an Enemy that is spawned at the second turn of the game, on top of the Warp Pipe. Since it is an enemy, PiranhaPlant class extends from Enemy class.

2. It’s behaviour is cleared at first inside constructor because it cannot move around nor follow player. Inside its playTurn method, it is coded that it can only attack when the player is around its exits.
3. Piranha Plant has 150HP, this is set inside the constructor. Piranha Plant deals 90 damage upon landing a successful attack, and this is done by simply overriding the getInstrinsicWeapon method and set it to 90 damage.
4. If the Piranha Plant is still alive when the player resets the game, its HP is increased by additional 50HP.

Flying Koopa (F)

1. Now that the game has Normal Koopa and a Flying Koopa, we first have to modify something before implementing Flying Koopa. First, original Koopa class is changed to an abstract class, and its constructor contains a new capability called KOOP. This is to separate itself with other enemies.
2. Since now the game has two types of Koopas, a new class is created called NormalKoopa that extends from Koopa abstract class. This is the Koopa that walks on land, and it has 100HP, initialised in its constructor.
3. The second type of Koopa is FlyingKoopa that also extends from Koopa abstract class. However, in its constructor, its HP is 150, and it has a new capability called FLYING.
4. Since FlyingKoopa can fly around the map, entering places that Normal Koopa cannot enter, this is achieved by creating an action called FlyAction, and an interface called Flyable.

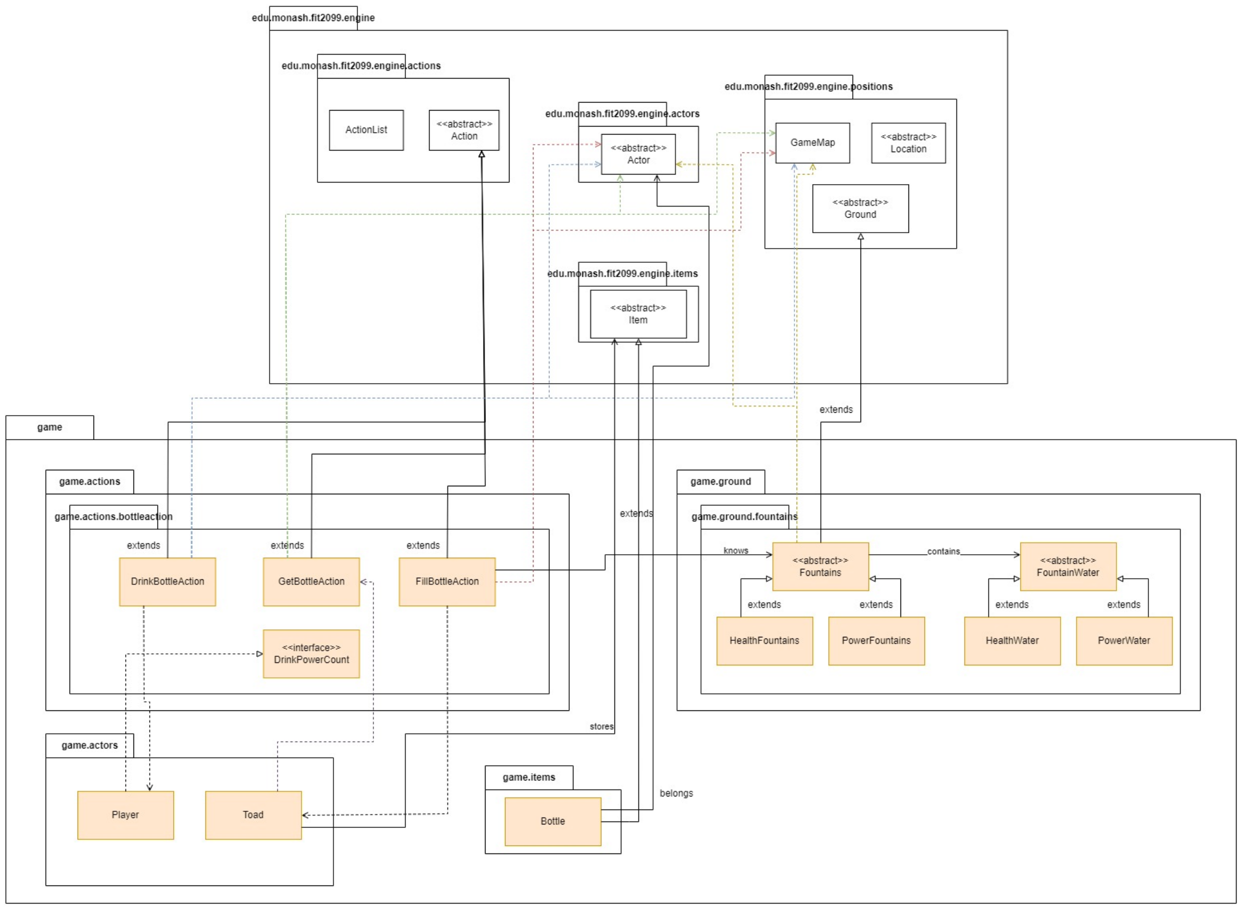
Flyable — An Interface

1. Flyable interface has a method called fly(). This interface is implemented by ground classes such as Wall, Floor, and Tree.
2. For the three classes mentioned, there is an if-statement added inside their allowableAction method, that checks if the actor has the capability of flying: if yes, it will return a FlyAction instead of a normal moving action. This is useful not only for FlyingKoopa, but for the future enemies/actors that can fly around the map.

FlyAction

1. Inside FlyAction, its constructor takes in classes that implements Flyable interface. For its execute method, it simply calls the fly method inside the classes that implements Flyable interface.

REQ3: Magical fountain



<u>Implementation</u>

Bottle

1. Bottle is designated to be a personal thing to the player, hence, a design similar to Wallet is taken. A Bottle class is created and it extends from Item abstract class. Inside the class, it has two static attributes: one is to store the water using ArrayList, another is to store a player’s information.

2. Inside the constructor, its details is initialised, and its not portable (portable == false), this is to make it so that the player cannot drop the bottle on the ground.
3. There are multiple static methods created, most notable ones are setPlayer, which sets the bottle to one player, addWater, which stores the fountain water in the ArrayList, and drinkWater, which gets the last water added into the ArrayList, removes it from the ArrayList, and “consume” the water.

Toad & GetBottleAction

1. Toad class is modified such that if the player does not have the bottle in his inventory, by checking if the player has the capability of HAS_BOTTLE, it will return an action called GetBottleAction.
2. GetBottleAction is a subclass of Action abstract class. It simply creates a Bottle object and adds it to the player’s inventory, and it runs Bottle class setPlayer static method. At last, actor now gains a capability of HAS_BOTTLE.

Fountains — Abstract class

1. Fountain is an abstract class created and extended from Ground abstract class. It is the base class to create specialised Fountain class in the future, hence adhering to Single Responsibility Principle and Open-Closed Principle (hereafter, SRP, OCP).
2. It has some attributes to be initialised in the constructor, most notable one to be one Fountain stores one FountainWater. Besides, it has a waterCount that represents how many water left in the fountain, and a refillCount to keep track of how many turns before the fountain is refilled with new water again.
3. In its allowableActions method, it is coded such as if the waterCount is not zero and refilling is false, it checks if the player has bottle in his inventory: if yes, it will add a FillBottleAction that takes in current fountain and fountain water as parameter, and add the water into the player’s bottle. Else if the refillCount is lesser than 4, it indicates that the fountain is refilling its water, hence refilling is set to true, and refillCount will increment by 1. Else, it means the fountain has fully replenished its water, refilling sets to false, refillCount is set back to zero, and waterCount is set back to 10.
4. Fountain abstract class implements Drinkable, thus the method inside Drinkable is needed to be overrode. The method is coded such that the fountain water is to be

consumed by player.

FountainWater — Abstract class

1. FountainWater is created as an abstract class for the purpose of SRP and OCP. It does not extend from any superclass and it contains an abstract method called consumeWater.

HealthWater & PowerWater — Subclasses of FountainWater

1. HealthWater and PowerWater are subclasses of FountainWater. Thus, consumeWater method must be overrode.
    a. In HealthWater class, its consumeWater method allows the actor to heal his HP by 50.
    b. In PowerWater class, its consumeWAter method adds a capability called DRANK_POWER to the player.
2. Besides that, both classes toString method are overrode such that it returns its name (”Health Water”, and “Power Water”)

HealthFountain & PowerFountain — Subclasses of Fountains

1. Both HealthFountain and PowerFountain are subclasses of Fountain abstract class. In their constructor, they both initiliase it with their designated values.
    a. HealthFountain’s display character is ‘H’, and it contains HealthWater.
    b. PowerFountain’s display character is ‘A’, and it contains PowerWater.
2. Both toString method contains its names and their remaining water count in the fountain so that the user can see how many times they can refill their bottle.

FillBottleAction

1. FillBottleAction is a subclass of Action abstract class. It contains a Fountain as its attributes. The attribute is initialised in the constructor as the constructor takes in one parameter as well (Fountain)
2. In the execute method, the fountain water count is decreased by calling a method from Fountain class called reduceCountByFilling, which reduces the water count by 2., allowing the player to refill his bottle 5 times per fountain. Then, Bottle class’s

static method addWater is called, and the water contains in the Fountain is added into the Bottle using the method getWater() in Fountain class.

DrinkPowerCount — An Interface

1. DrinkPowerCount, as mentioned, is an interface that allows the actor to drink specifically Power Water. This is created with the mind that in the future, actor or enemies that can drink Power Water should implement this interface. It contains a method header called drinkPowerWaterCount. This design also adheres to DIP.

Player

1. Inside Player class, in the playTurn’s method, it is coded such that if the player has the bottle in his inventory, and the bottle contains water, it will give the player the action to drink the water last added into the bottle. The drink action is called DrinkBottleAction. Player class now also implements DrinkPowerCount interface.
2. Besides that, Player class getInstrinsicWeapon method is modified such that after every time player drinks Power Water, it creates a multiplier such that the player’s base damage increases by 15 after every drink. This is coded such that if the player has the capability of DRANK_POWER, it will run a method called drinkPowerWaterCount, then the capability is removed.
3. Player class now contains a counter called drinkCount. After each execution of drinkPowerWaterCount method, which is an overrode method from DrinkPowerCount interface, drinkCount is incremented by one, and it calls getIntrinsicWeapon method. Inside getIntrinsicWeapon method, its base damage is increased with a multiplier using drinkCount.

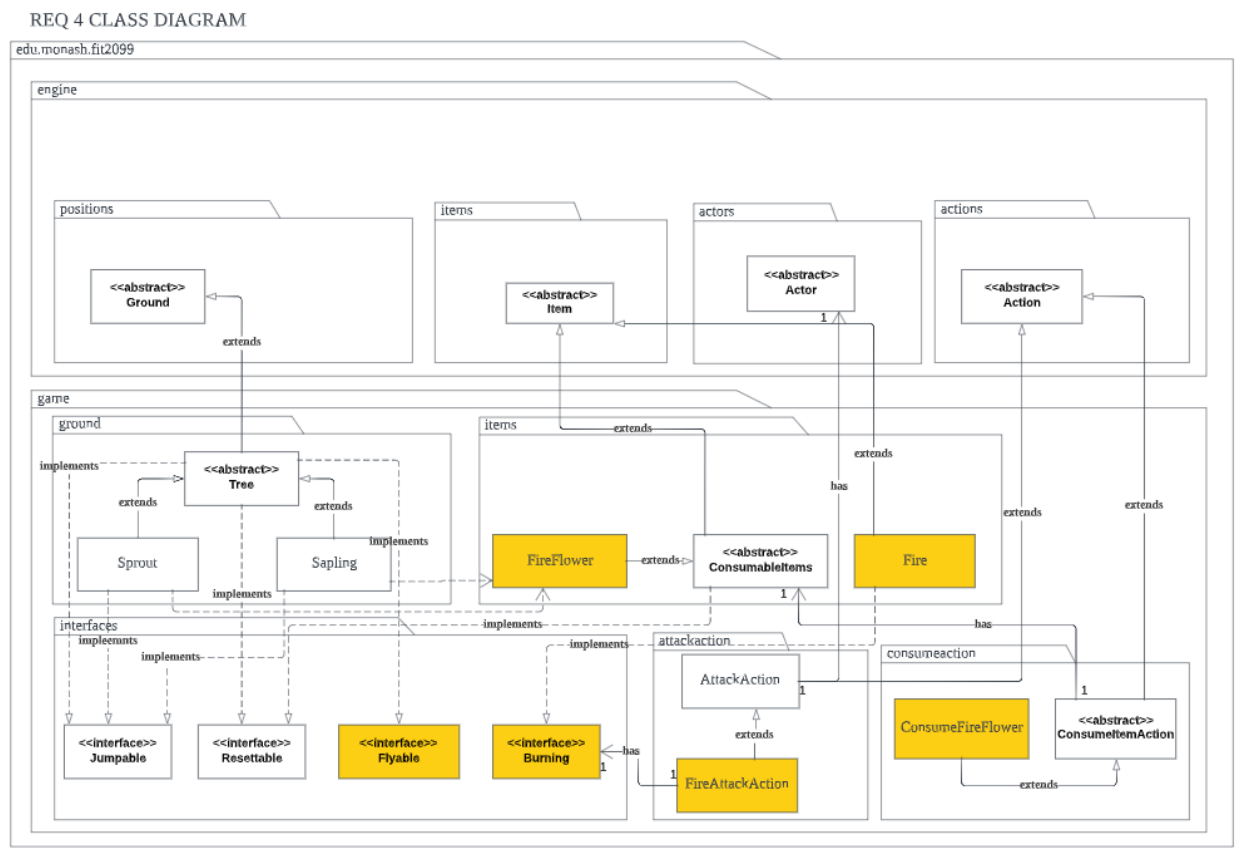
DrinkBottleAction

1. DrinkBottleAction is a subclass of Action abstract class. It doesn’t take in any parameters in its consturctor. Its execute method runs such that the last water added in the bottle is obtained using Bottle class static method called getWater. Then, Bottle class static method, drinkWater, is run, which it calls the FountainWater’s consumeWater method.
2. Its toString method is to return the water contained in the bottle.

Clarifications

1. Even though we didn’t plan to implement optional challenge for REQ3, we only implemented showing water count, and obtaining bottle from Toad, with the idea of balancing the game difficulty. Endlessly filling up the bottle without a counter from fountain makes the game too easy for the player, and we force the player to journey to find the Toad and obtain the bottle from Toad to make the game more interesting.

REQ4: Flowers 🌻 (Structured Mode)



1. Fire Flower (f) 🌻

Design Implementation:

FireFlower class

1. An item that can be consumed by Mario to use fire attack.

2. Since it is a consumable item, it will extends the ConsumableItems abstract class to promote polymorphism.
3. For every growing stage of the tree, it has 50% chance to spawn a fire flower in its location and this can be implemented in the tick method of sprout and sapling to do an if statement checking to check whether the random value <= 0.5, if it does it will spawn a fire flower on top of it.

ConsumeFireFlower class

1. An action that can be taken by the player to consume fire flower.
2. Its constructor will take in an instance of ConsumableItems as input and will call the instance of ConsumableItems's consumeItem method in its execute method body .By doing so, we are adhering to the Open-Closed Principle by preventing the need to modify the existing code (such as adding multiple if-else statement) when there's a new item that is consumable.
3. When the player consumes the fire flower, the player will obtain a status called Status.FIRE_ATTACK and this status will allow the player to use fire attack when encountering enemies.
4. If the player has already consume one fire flower and he wanted to consume more fire flower, a string message will be printed out in the console saying that he had already consumed one fire flower from previous ConsumeFireFlower action to restrict the player to only consume one fire flower if the fire attack effect is still there.

2. Fire attack (v) 🔥

<u>Design Implementation:</u>

Fire class

1. An item that will be dropped at the target's ground when the player is using fire attack to attack the target or when the bowser is attacking the player.
2. Implements a burning interface and its method burn that will be called inside the FireAttackAction's execute method.
3. Fire will stay on ground for 3 turns and this can be implemented by using a counter inside the fire class's tick method. Everytime the fire class's tick method is being

called , the counter will increment by one and once the counter reaches 3, this item will eventually be removed from the map else if the counter haven't reached 3, then for every actor that steps on the ground provided that the actor does not have the immunity status , the actor hit points will get deducted by 20 due to the burning effect from the fire.

Burning interface

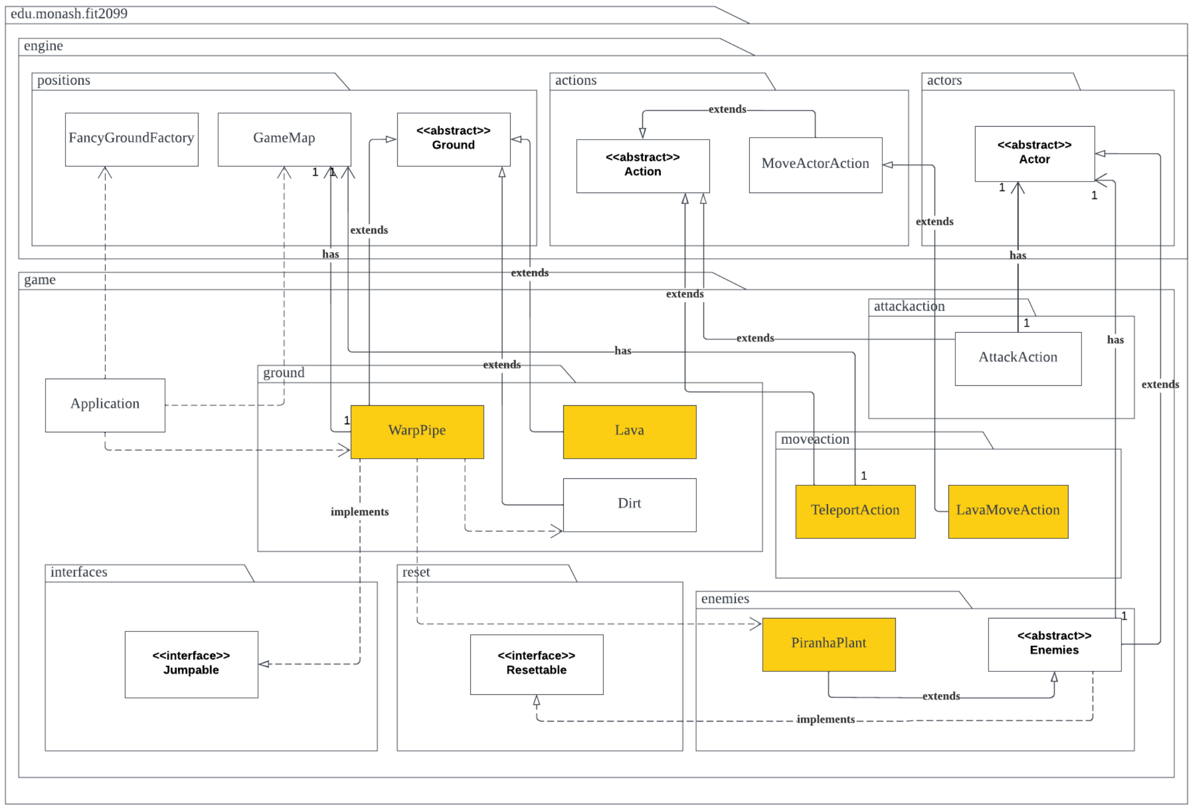
1. An interface containing a burn method which will be implemented by the fire class.
2. The burn method implemented by the fire class will contain all the logic of what will happen when the player uses a fire attack on the enemies.

FireAttackAction class

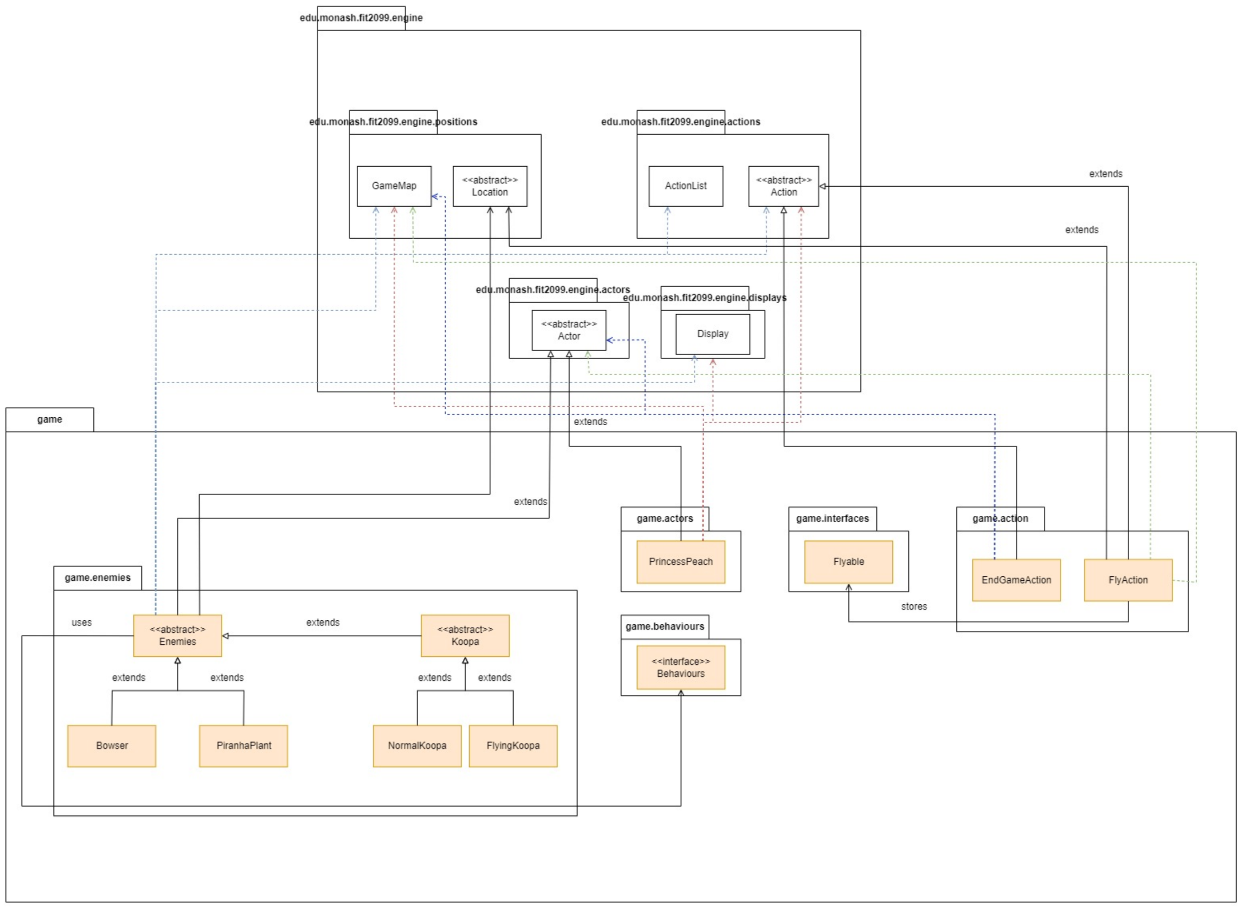
1. An action that can be taken by the player when the player has already consumed a fire flower and at the same time an enemy is approaching the player.
2. Has a direct association with the burning interface. By doing so, we can prevent the usage of if-else statements in the execute method whenever there's a new item which will cause a burning effect instead we can just call the burn method that is implemented by those classes which implements the Burning interface. Such design satisfies the Open-Closed Principle.
3. This class also adheres to the Single-Responsibility Principle because it is only responsible for the attack action of the player when the player consumes fire flower and nothing else.
4. This class will extends the AttackAction class instead of Action class because when in both classes there are having similar logic hence there will be repeated codes, for instance when the target (enemies) died from the attacks, they will drop certain items. Since they are having repeated codes, I have chosen to allow this class to extends AttackAction to adhere to OOP principles which is to reduce the amount of repeated codes.

UML Class Diagrams

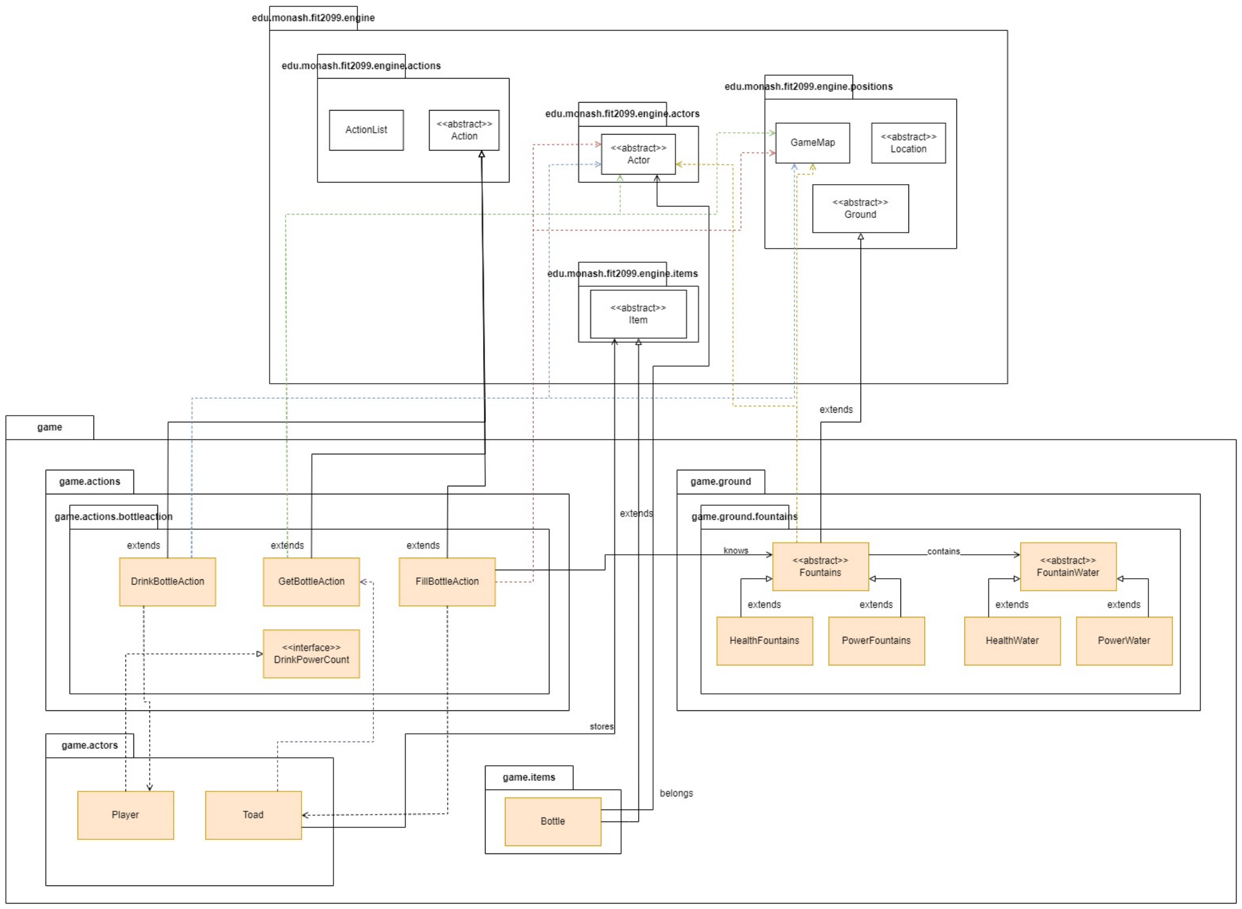
REQ 1 CLASS DIAGRAM



Requirement 1 Class Diagram

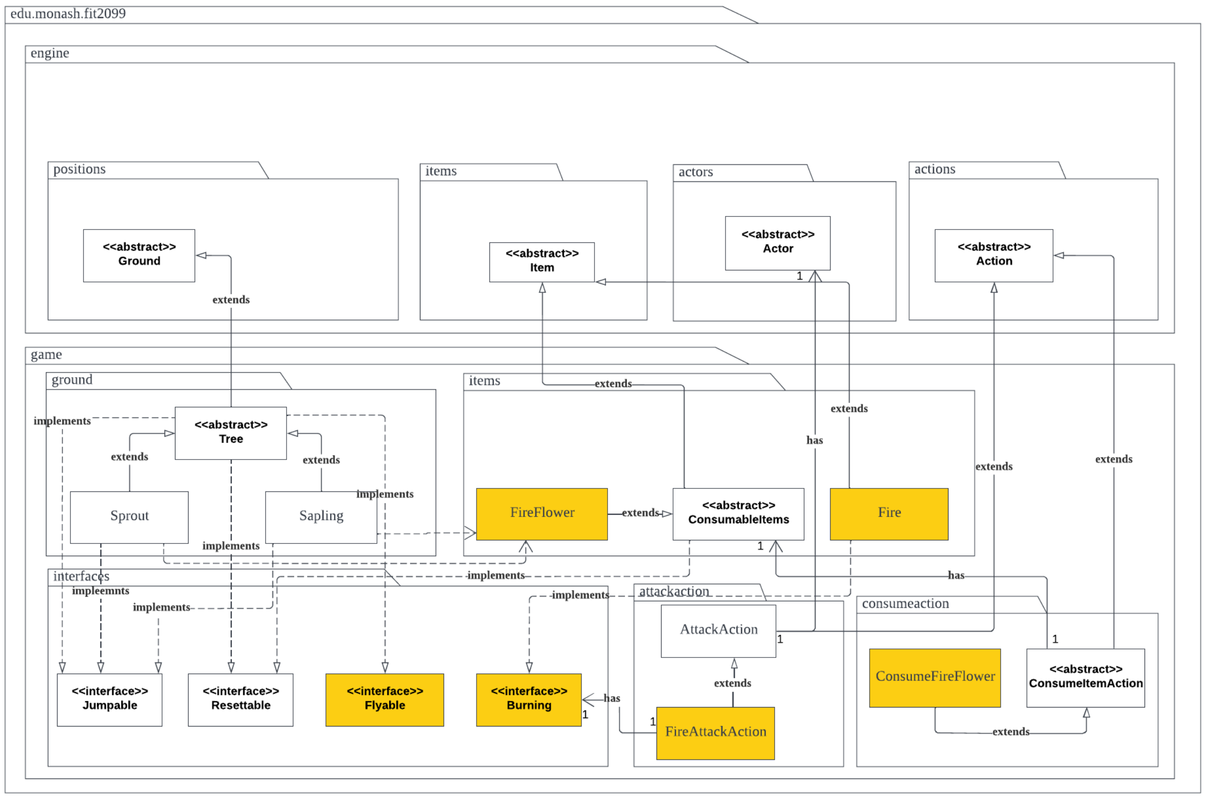


Requirement 2 Class Diagram



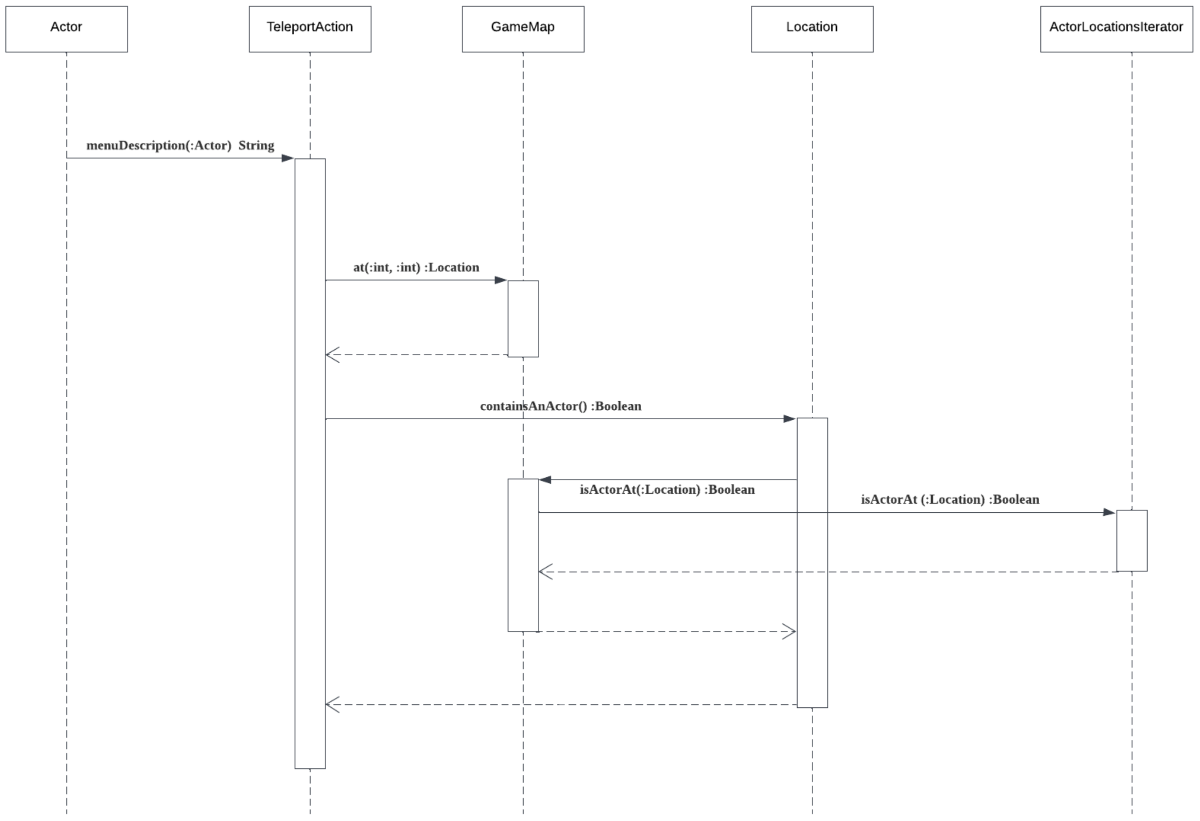
Requirement 3 Class Diagram

REQ 4 CLASS DIAGRAM



Requirement 4 Class Diagram

Interaction Diagrams


Actor
TeleportAction
GameMap
Location
ActorLocationsIterator
menuDescription(:Actor) String
at(:int, :int) :Location
containsAnActor() :Boolean
isActorAt(:Location) :Boolean
isActorAt (:Location) :Boolean

Sequence Diagram for TeleportAction