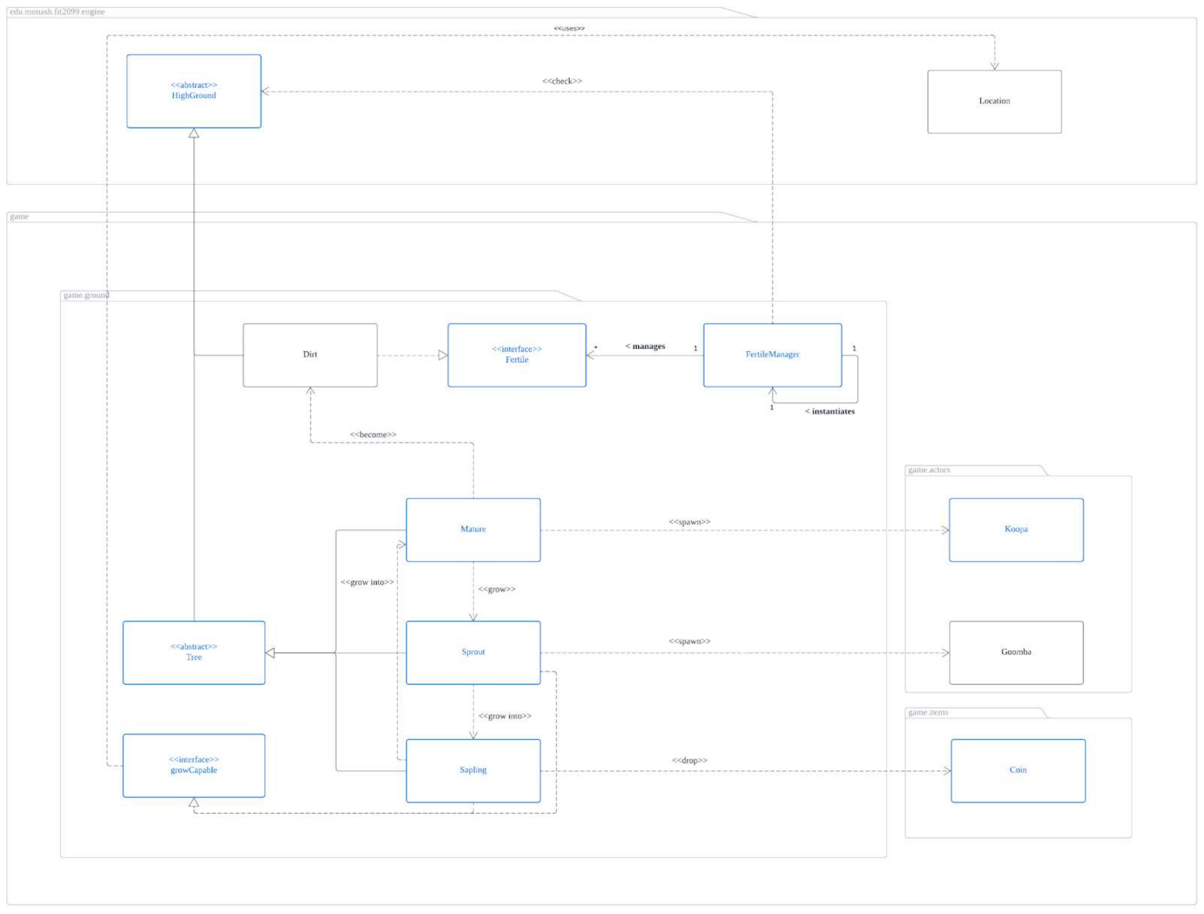
Design Documents (Assignment 1 & 2)

REQ1

Class diagram:



Requirement:

- Sprout & Sapling can grow into their next stage after they reach 10 turns
- Each stage of tree has their own unique spawning ability
  - ➢ Sprout: 10% chance spawn Goomba in every turn
  - ➢ Sapling: 10% chance to drop a coin ($20) in every turn
  - ➢ Mature: 15% chance to spawn Koopa in every turn
  - ➢ Mature: Can grow a new sprout on surrounding fertile ground every 5 turns (if there is one)
  - ➢ Mature has a 20% chance to wither and die (become dirt) in every turn

Design Rationale:

In requirement 1, to represent the 3 stages of the tree, we have decided to change the **Tree** class into an abstract **Tree** class that is a subclass of **HighGround**, we have also created 3 new classes which are **Sprout**, **Sapling** and **Mature** where all of them are the subclasses of **Tree** class.

The implementation of our design follows the **Open-Closed Principle (OCP)** because the **Tree** class is open for extension but closed for modification. Since there will be different spawning ability for each stage of the tree, there will also be different implementation of code. To tackle this, we would add an abstract method named `spawn()` in the Tree class so that every subclass can perform their own unique spawning ability. The design of abstract **Tree** class can ensure that whenever there is a new stage of tree which has another unique spawning ability, the **Tree** class would not be modified to add in new features.

To meet the requirement of unique spawning ability, we would override the spawn() method in all the subclasses to different code implementations since they will have different spawning ability. The design of `spawn()` method in each class is shown below:

spawn() in Sprout

1. Create a local variable named goombaSpawnChance and assign a value of 10 to it
2. Use `rand.nextInt(100) <= goombaSpawnChance` to check the 10% chance of spawning Goomba
3. Use location.containsAnActor() to check if there is an actor on the current location
4. If steps 2 & 3 returns true, then it would create an instance of Goomba and use `location.addActor(goomba)` to spawn the goomba on its position.

spawn() in Sapling

1. Create a local variable named coinDropChance and assign a value of 10 to it

2. Use `rand.nextInt(100) <= coinDropChance` to check the 10% chance of dropping coin
3. If statement 2 returns true, then it would create an instance of Coin ($20) and use `location.addItem(coin)` to spawn the coin on its position.

spawn() in Mature

- **Spawn Koopa**

    1. Create a local variable named koopaSpawnChance and assign a value of 15 to it
    2. Use `rand.nextInt(100) <= goombaSpawnChance` to check the 15% chance of spawning Goomba
    3. Use location.containsAnActor() to check if there is an actor on the current location
    4. If steps 2 & 3 returns true, then it would create an instance of Koopa and use `location.addActor(koopa)` to spawn the koopa on its position.

- **Spawn sprout in surrounding fertile ground**

We would create a new interface called **Fertile** and a new class called **FertileManager** which aims to organise all the objects that implement the Fertile interface. For now, since the only class that is classified as fertile ground is **Dirt,** the **Fertile** interface is only implemented by the **Dirt** Class. In the constructor of every class that implements the **Fertile** interface, it will need to use the default `addToFertileManager()` method in the **Fertile** interface to add the object into ArrayList of Fertile in **FertileManager**.

The implementation of such design adheres to the **Dependency Inversion Principle (DIP)** as FertileManager does not depend on the low-level module such as Dirt. When new classes that implement Fertile are added in, we would not need to modify the FertileManager to keep track of all the objects that implement Fertile interface. This is because there is a layer of abstraction (**Fertile** interface) between all of them and it can ensure that the high-level module and low-level module would not affect each other.

1. Use `(turn != 0 && (turn % 5 == 0))` to check every 5 turns requirement where `turn = super.getAge()`
2. Use for loop on `location.getExits()` to get every exits available at the current location
3. For every exit, we use `exit.getDestination()` to obtain the destination (location) of the exit.
4. For every destination, we use `destination.getGround()` to get the ground type of the current location.
5. For every ground, we check if the ground in that location is a fertile ground by using `FertileManager.getInstance().getFertileGroundList().contains(ground)`
6. If the destination is a fertile ground, it will be added into a ArrayList of **Location** named surroundFertileList
7. After completing the for loop above, if there is any fertile ground around the mature, it will create a new object of **Sprout,** and use `surroundFertileList(rand.nextInt(surroundFertileList.size())` to randomly choose one of the fertile ground (location) and use the `setGround()` method to set the ground type of that location to **Sprout**.

Besides, we have added some common attributes such as `age` in the Tree class to track the age of the tree so we know when the tree should grow into its next stage. This implementation complies with the **Don't Repeat Yourself Principle (DRY)** such that we do not need to keep declaring the same attribute in every subclass of **Tree**.

Moreover, we would create a new interface named **growCapable** to handle the growing of trees from one stage to another. This interface will be implemented by the **Sprout** and **Sapling** class only because they can grow into next stage such that

1. Sprout -> Sapling
2. Sapling -> Mature

In the `grow()` method from **growCapable**, we would override it in both classes that implemented it by creating a new instance of the class that will be the next stage of the tree, and pass it into the `location.setGround()` method to change the ground type of the current location. For instance, when the `grow()` method of **Sprout** is called, it will create a new Sapling object and pass it into the `setGround()` method which changes the ground type of the location from Sprout to Sapling that indicates the changing of stages of the tree.

This design follows the **Liskov Substitution Principle (LSP)** as we are only implementing the **growCapable** interface in **Sprout** and **Sapling** class instead of making it as an abstract method in abstract **Tree** class. This is because sprout and sapling can grow into their next stage while mature cannot, if we put the `grow()` method as an abstract method in Tree class, it will break the LSP as there is a subclass that can't do what the base class can.

In addition, the design is in line with the **Interface Segregation Principle (ISP)** where interfaces should be smaller, and the client should not be forced to depend upon interfaces that they do not use. In our implementation, the **growCapable** interface only has one method which is `grow()` to indicate the growing of the tree to another stage. It is small enough and would not force the class that implements it to use something that they do not need to.

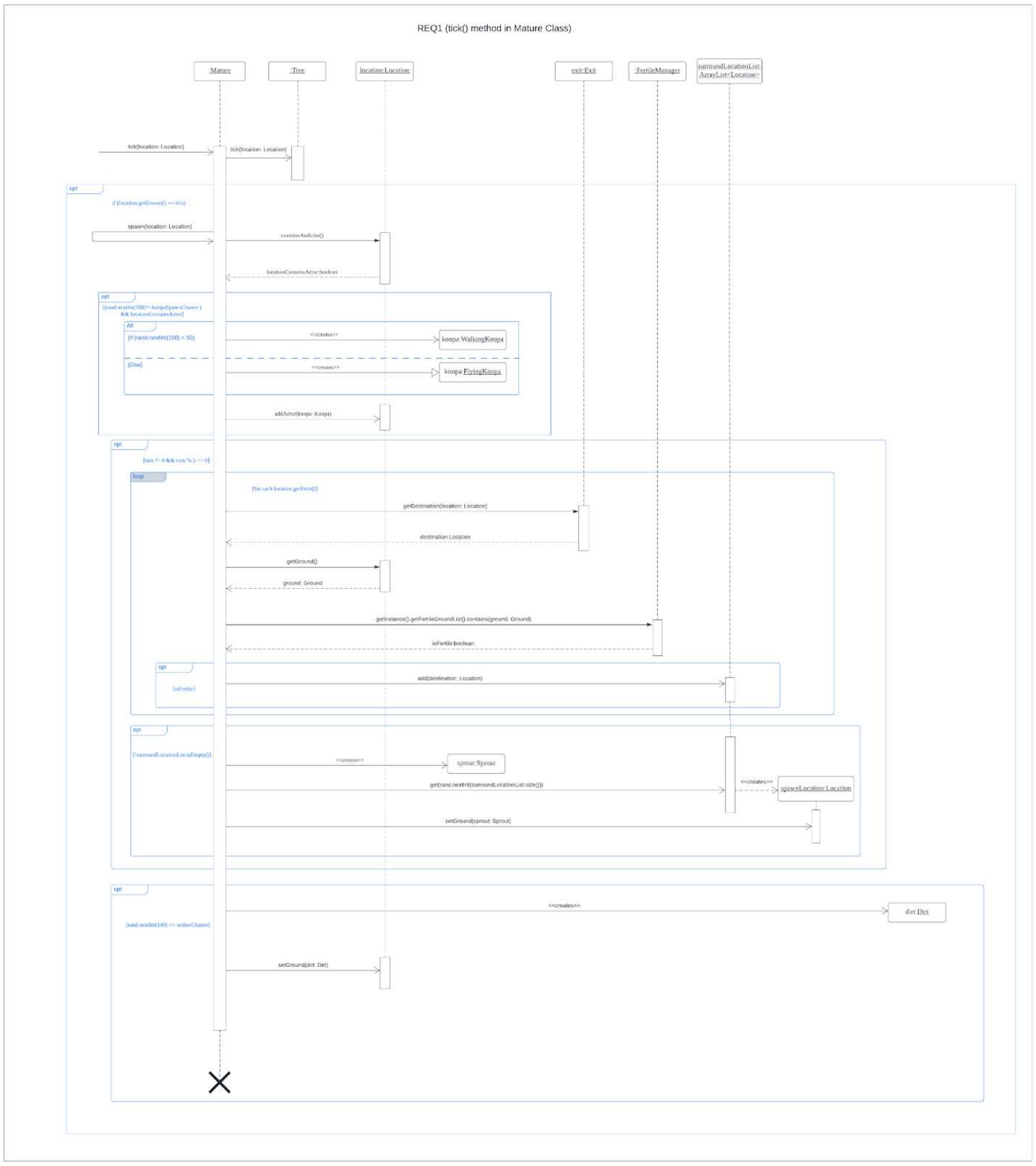
To implement the withering of Mature, we would use `(rand.nextInt(100)<= witherChance)` to check if it meets the 20% chance to be withered where the witherChance is a final attribute of Mature. If yes, then we would create a new Dirt object and use `location.setGround(dirt)` to change the ground type to Dirt.

Every method mentioned above such as `grow()` and `spawn()` will be implemented in the `tick()` method of their own class since the `tick()` method aims to let the location experience the passage of time, in order words, it is the turn of the game.

*Updates that have been done for REQ1 (class diagram & design rationale)

I. Class diagram:
   A. Tree extends from HighGround now
II. Design rationale:
   A. Changed Tree extends from Ground to Tree extends from HighGround

Sequence Diagram:


REQ1 (tick() method in Mature Class)
:Mature
:Tree
location:Location
exit:Exit
:FertileManager
surroundLocationList:
ArrayList<Location>
tick(location: Location)
tick(location: Location)
opt
if (location.getGround() == this)
spawn(location: Location)
containsAnActor()
locationContainsActor:boolean
opt
[(rand.nextInt(100)<=koopaSpawnChance)
&& locationContainsActor]
Alt
[if (rand.nextInt(100) < 50)
<<creates>>
koopa:WalkingKoopa
[Else]
<<creates>>
koopa:FlyingKoopa
addActor(koopa: Koopa)
opt
[turn != 0 && turn % 5 == 0]
loop
[for each location.getExits()]
getDestination(location: Location)
destination:Location
getGround()
ground: Ground
getInstance().getFertileGroundList().contains(ground: Ground)
isFertile:boolean
opt
[isFertile]
add(destination: Location)
opt
[!surroundLocationList.isEmpty()]
<<creates>>
sprout:Sprout
get(rand.nextInt(surroundLocationList.size()))
<<creates>>
spawnLocation:Location
setGround(sprout: Sprout)
opt
<<creates>>
dirt:Dirt
[rand.nextInt(100) <= witherChance]
setGround(dirt: Dirt)

This sequence diagram will describe the operation of Mature in every turn whereby Mature would have a 15% chance to spawn Koopa, with 50% chance to spawn either Walking or Flying Koopa, grow a new sprout in one of its surrounding fertile squares randomly for every 5 turns and also have a 20% chance to wither.

The starting point of the sequence diagram above is when the `tick()` method in the Mature class is called. The `tick()` method will separate into 3 parts:

1. super.tick()

➔ Firstly, it will call the tick() method from its super's class and perform any respective action without returning anything.

2. spawn() method

    ➔ From then on, we will check whether the current location ground type is equal to Mature. This layer of checking is needed because after the reset action has been done, the tick() method of Mature will still be ongoing although the type of ground of the current location has changed to Dirt. Without this checking, this might leads to a problem where the Koopa is spawned above a ground where there is no more Mature on it.

    ➔ If true, we will call the spawn method which is located in the Mature class itself. For more detailed information about the code in the spawn method in the sequence diagram, please refer to the **spawn sprout in surrounding fertile ground** section in the design rationale of REQ 1.

3. 20% chance of wither

    ➔ In this part, it will check whether it meets the 20% chance to wither. If yes, it will create a **Dirt** instance and pass it as the method argument into `location.setGround(dirt)` to change it to dirt which indicates that the mature has withered.

*Updates that have been done for REQ1 (sequence diagram & explanation)

I. Added in checking for 50% chance to spawn either WalkingKoopa or FlyingKoopa while spawning Koopa