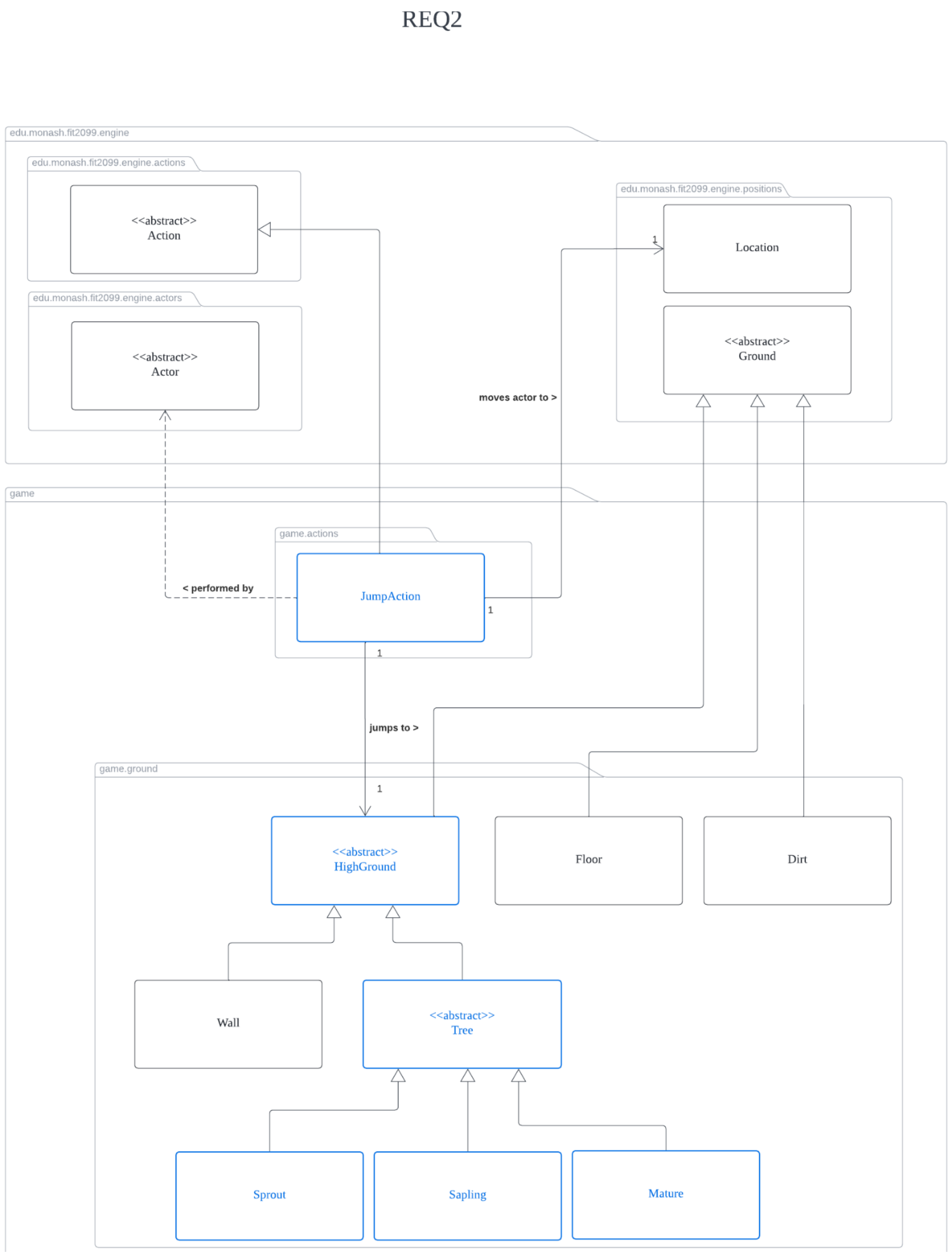
## REQ2

### Class Diagram:



## Requirement:

A Super Mario game will not be complete without a **"jump"** feature. When the actor is standing next to the high ground, the actor can jump onto it. Going up requires a jump, but going down doesn't require it (the actor can walk normally). Each jump has a certain success rate. If the jump is unsuccessful, the actor will fall and get hurt. The success rate and fall damage are determined by its destination ground as listed below:

- Wall: 80% success rate, 20 fall damage
- Tree:
    - Sprout(`+`): 90% success rate, 10 fall damage
    - Sapling(`t`): 80% success rate, 20 fall damage
    - Mature(`T`): 70% success rate,  30 fall damage

## Design Rationale:

In requirement 2, we are required to create the feature “jump” which allows the actor to jump to the place the actors cannot move into. We would create a **JumpAction** class, a **HighGround** abstract class and modify **Tree**, **Mature**, **Sapling**, **Sprout** and **Wall** classes.

First, the **HighGround** abstract class is created because there are many grounds which the actors cannot move into. Each highGround has their own value of jumpSuccessRate and fallDamge. In our implementation, highGround will extend ground. Two attributes jumpSuccessRate and fallDamage will be declared in highGround class, as well as their setters and getters. These values are to be defined in its subclass as different grounds have different values. In the class, we override the canActorEnter method in Ground class and make the method return false as all highGround are not available for actors to move in, actors should jump onto it. Also in the highGround, we would implement a getHighGroundName which would return a string value of the name of the ground which would be used in displaying the result of jumpAction. Next, we have allowableAction overridden from Ground class. This allowableActions method  would add a jumpAction to ActionList if the actor is a player. Parameters passed to the method include actor, direction, location and highGround. This implementation is following **Open-Closed Principle (OCP)** as by having a highGround abstract class, it is very easy to extend another type of highGround like rooftop, without changing the existing code. By using abstraction, we reduce the amount of effort to add new highGround in the future. The new type of highGround will have the same attributes jumpSuccessRate and fallDamage along with their setters and getter and allowableAction methods. This implementation also follows **Single Responsibility Principle (SRP)** as highGround only contains the attributes and methods which are needed. It does not implement any other new things which are not necessary for a highGround. This implementation follows the **Don't Repeat Yourself Principle (DRY)** because we do not have to repeat creating a setter and getter for attributes in different types of highGround.

We are going to discuss Tree, Mature, Sapling, Sprout and Wall in this paragraph. Tree is an abstract class extending HighGround. Mature, Sapling and Sprout are extending Tree, which is implemented in requirement 1. Wall is a class that extends HighGround. Tree class is left unchanged except extending the class. Meanwhile, sapling, mature and sprout override the method from HighGround. Two methods setFallDamge and setJumpSuccessRate are called during the initialization of sapling, mature and sprout via constructor. The values set are following the requirement, which are: 1) sprout – FallDamage:10, SuccessRate:90 2) sapling – FallDamage:20, SuccessRate:80 3) mature – FallDamage:30, SuccessRate:70. They also override the getGroundName methods which would return “sprout”, “sapling” and “mature” respectively. Similar implementation goes to wall class which directly extends HighGround class. Wall will override the method to set fallDamge:20, SuccessRate:80, and return “wall” string. We are choosing this implementation as the reason mentioned above. New and existing highGround can be added and managed easily because they all inherit the same class.

In this section, we are looking at the JumpAction implementation. JumpAction is extending Action class. In assignment 1, jumpAction has 5 instance variables: 1) actor 2) direction 3) destination 4)highGround 5) random. In assignment 2, change made is to remove the actor attribute because it is not required for the action. The change has changed the relationship between jumpAction and actor from association to dependency. Actor is only needed in the execute method as a parameter to indicate which actor is jumping. This will make the two classes to be more loosely coupled. This follows the principle that **classes should be responsible for their own properties to reduce**

**dependency** on each other. We first create a constructor which takes 4 parameters and initialises them, including actor, direction, destination and highGround. Next we would override the execute method from the Action class. In this method, we would first get the jumpSuccessRate and fallDamage using getJumpSuccessRate and getFallDamage methods of highGround. Change made here is to check if the player has the status of SUPER due to the effect of super mushroom. If it is true, we will set jumpSuccessRate to 100. Next, using the random which is previously initialised, we generate a number below 100 and see if the number is bigger than the range of jumpSuccessRate. If it is bigger, it means the jump action fails. We would reduce actor hit points by fallDamage value using the hurt method in Actor class and print a message saying the actor fails the jump and the damage received. We then check if the player is still alive using isConscious method in Actor. Message of "actor is dead" would be printed if the player dies. If the player succeeded in the jump, we would move the actor to that location using the moveActor method in GameMap class and print a message saying that the jump is successful. The other method we override is menuDescription which returns a string that "actor jumps to <<direction>>".  This is used to let the users know what options they should choose in the menu if they want to jump to a certain direction.

*Updates that have been done for REQ2 (class diagram & design rationale)

I. Remove actor attributes in JumpAction
II. Change the relationship between JumpAction and Actor from association to dependency.
III. Check if player.hasCapability(Status.SUPER). If the true super mushroom effect is active, set the jump success rate to 100.