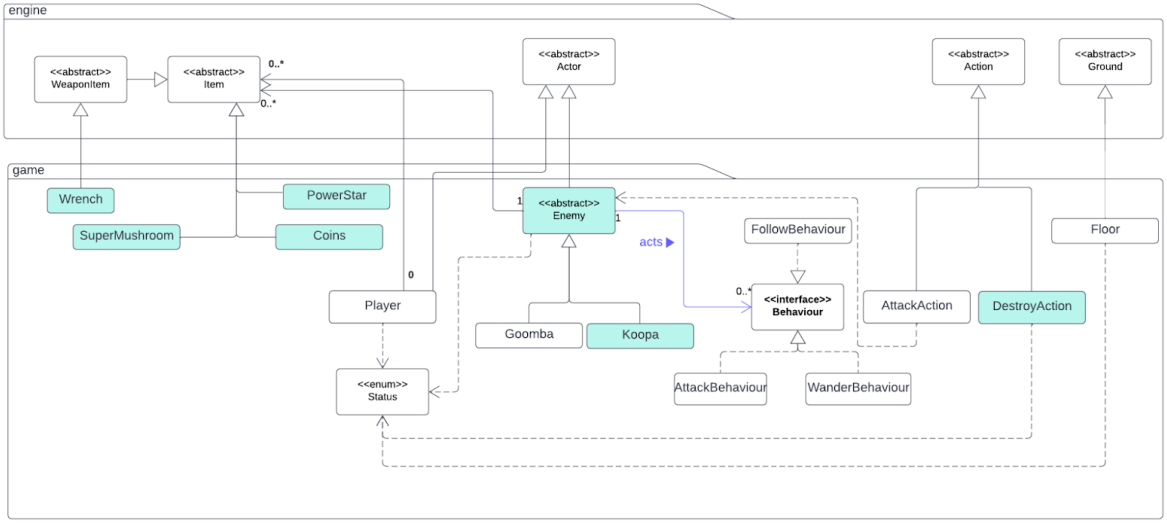
FIT2099 Assignment 1

SuperMushroom:

The sequence diagram makes the assumption that when walking over an item or booster, you automatically pickup that item and then it's 'in your inventory' and you then receive the action prompt to use the item.

The addCapability() updates TALL status that will be used in loops and methods to check that the player has picked up the SuperMushroom.

Advantages
Single Responsibility Principle is implemented in the SuperMushroom Class as it extends from the abstract item class so that the class only holds the responsibility of responding to the case where the player uses the SuperMushroom.



UML for enemies and items

Powerstar:

The sequence diagram from the powerstar also makes the assumption that when walking over an item or booster, you automatically pickup that item and then it's 'in your inventory' and you then receive the action prompt to use the item.

This will also add STAR as a capability as the Powerstar is picked up. And reset it at the end of the 10 turns

Override in player class the playTurn() to have a counter that runs when it detects invincible status.

Power Star will print to console the remaining turns active and the invincible status, handled in playturn().

The getIntrinsicWeapon method will be overwritten to give the player enough damage to kill any enemy.

Advantages	**Disadvantages**
Similarly, the PowerStar class extends from the <<abstract>> item class. Hence, it embodies the **Single Responsibility Principle** as it's only responsible for item actions related to the PowerStar Magical Item.	An observation and probably a change that will be made is the check____Balance() method, it's specific to each item but should be more general to receive input. To make it better, it will feed through the price from the item child class and then return Boolean if it's less than that balance etc.

Wrench:

The wrench extends the <<abstract>> class WeaponItem.
The logic is that when the player purchases a wrench, an instance of the wrench class will be created, initialised with static, final variables for hit rate and damage. And then when the player approaches within attack distance of an enemy, the player class will check for an instance of the wrench class in inventory. If it finds one, it will produce the prompt to use it. The wrench will be added as a capability when the player chooses to use the item to attack. Then the enemy can check the players status and take damage accordingly.

Advantages
The **Interface segregation principle** is followed by the design of WeaponItems and the way in which Wrench extends this. Wrench will implement all of the Weapon methods in the Weapon <<interface>> as they are applicable to the wrench requirements. The Wrench class doesn't implement or inherit any unused methods from the weapon interface.

Coins:

The coin class will extend the abstract Item class as they can be found on the floor but also used to purchase items. The coin's value will be kept in a private attribute.

The player will be given an attribute “wallet" which will update with each coin picked up. The coin class will be responsible for the instance of each coin, its worth and its position on the map.
Other classes will be responsible for creating coin instances.

Enemies Design Rationale

Key Decision: Enemy Abstract Class

With the addition of Koopa, we have decided to create an abstract class 'Enemy'.

Advantages	**Disadvantages**
- **Don’t Repeat Yourself:** There are various aspects common to both enemies, such as following the player and not entering the floor. By creating an abstraction of enemy we avoid repeated code in the Goomba and Koopa subclasses. - **Open/Closed Principle:** The true advantage of this approach will be realised if the game is extended with the addition of more enemies. The new enemy will not require new following or floor entry code, it can simply inherit the enemy class	- It can be argued that the abstract Enemy class doesn’t have enough function to justify its existence. Most functionality is handled by either parent or child classes. However, in the interests of avoiding repeated code and remaining extendable, the enemy class is added to the system.

Key Decision: DestroyAction Class

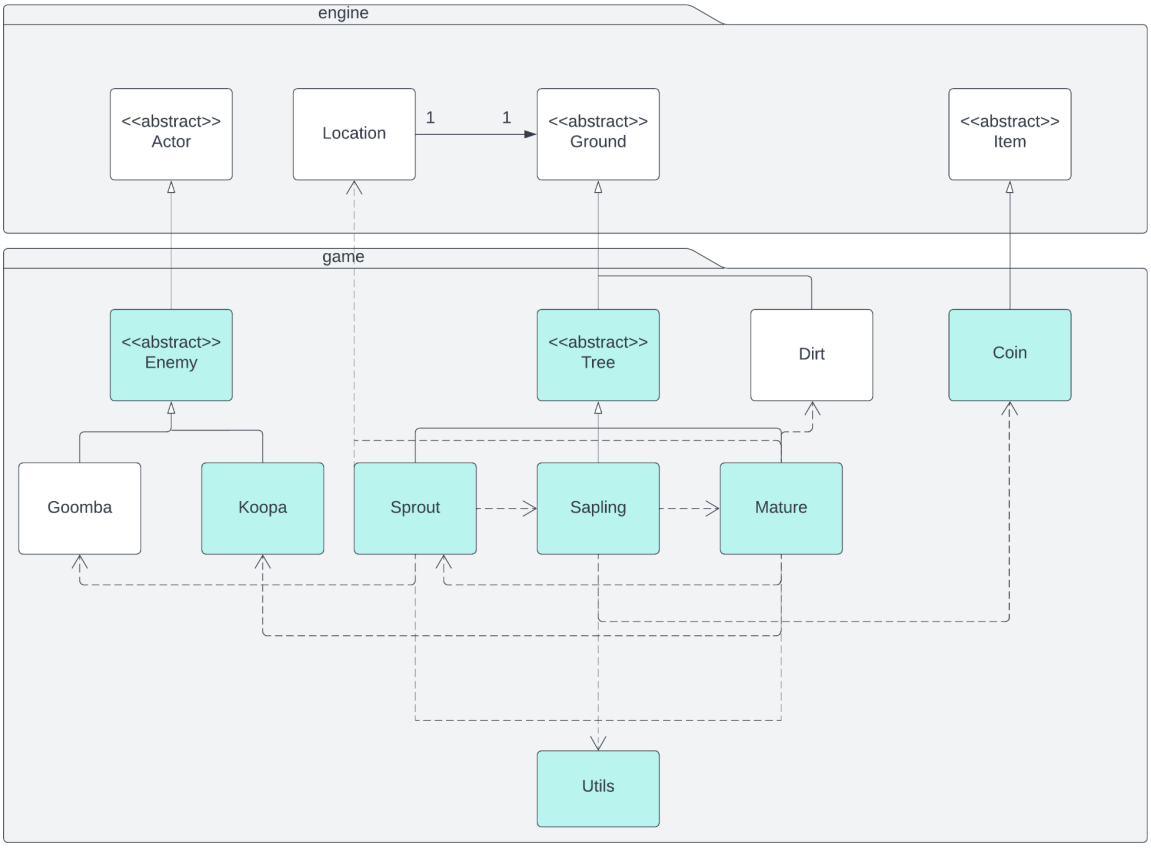
We have decided to modify the AttackAction class in order to incorporate player attacking Koopa as well as Goomba. The Koopa will have a capability CORPSE. If AttackAction’s target actor has this capability, it will not remove the target actor from the map upon death. When Koopa dies, its display character will be changed to D, it will lose CORPSE capability and gain DORMANT capability. Its allowable actions will no longer have AttackAction, in its place will be DestroyAction. DestroyAction will allow the Player to destroy the shell if the player has a wrench.

Advantages	Disadvantages
- **Reduce Dependencies:** These design decisions have been made in order to avoid the AttackAction class depending on Koopa. This also follows the **Liskov Substitution Principle:** We can pass *any* actor as target in AttackAction without breaking the code.	- **Single Responsibility Principle:** Koopa class handling both alive and dormant Koopa creeps towards Koopa becoming a God class. Implementing the “shell” as a separate class was considered, but ultimately rejected due to requiring too much refactoring of existing code and possibly engine code.

Key Decision: Give Koopa a Super Mushroom on instantiation

To allow Koopa to drop a mushroom when its shell is destroyed, Koopa will be given a SuperMushroom item on instantiation. The DestroyAction class will then handle the dropping of the item, in much the same way the AttackAction class does. This allows us to **use the existing engine methods** (dropItem) to drop the mushroom, rather than spawning one at a particular location.

Trees Design Rationale

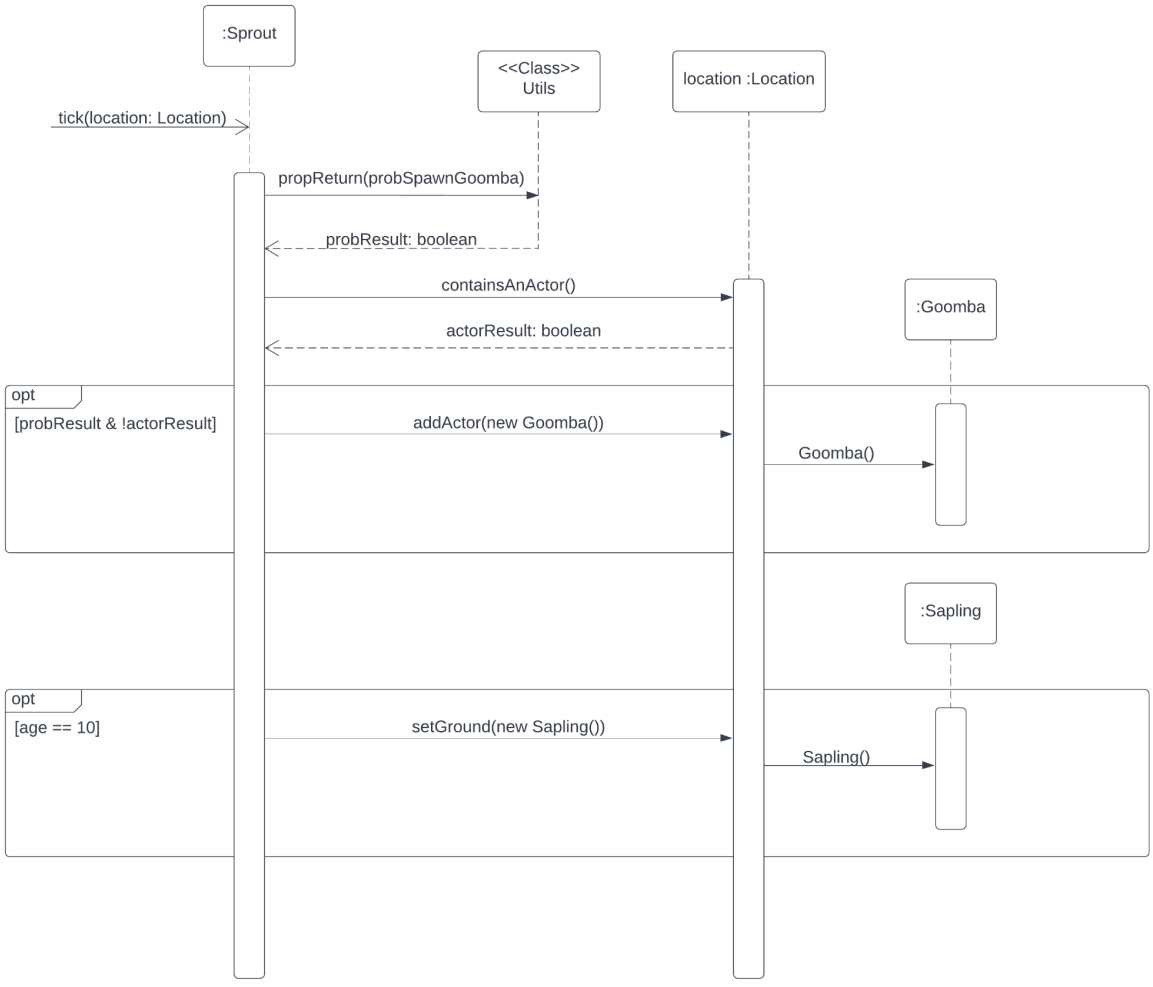


UML Class Diagram for Tree Features

Key Decision: Sprout, Sapling and Mature as Subclasses of Tree.

Each stage of the tree life-cycle has been implemented as a subclass of tree. Tree is now an abstract class. The Sprout, Sapling and Mature classes are now responsible for their own spawning and growing up.

Advantages	**Disadvantages**
- **Single Responsibility Principle:** Each subclass of tree is responsible for the aspects unique to that class. If Sprout, Sapling and Mature were implemented as a part of the Tree class (using Enums), this principle would be violated. These classes are separated because the different tree maturities change for different reasons.	- The tree 'growing up' now requires the subclass to call location.setGround() in the tick method, reinforcing a dependency on the Location class, violating the **Reduce Dependencies** Principle. This is justifiable as the **Dependency Inversion Principle** is still followed, lowly Sprout depends on lofty Location.



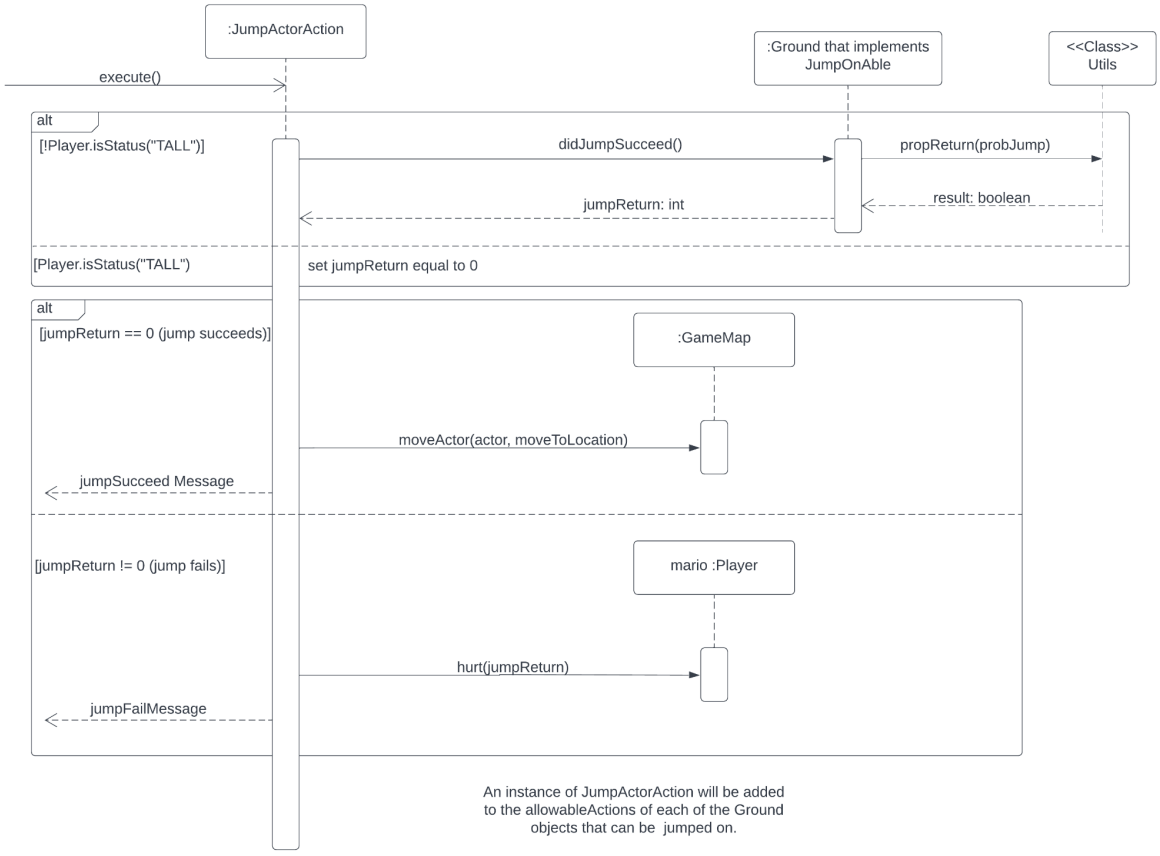
Sequence Diagram for the Sprout

Key Decision: Utils Class

In this project many methods in many classes require probabilities. E.g. Sprout has a 10% chance of Spawning Goomba. To handle this, we have a Utils class with a static method probReturn(probability) that returns a Boolean based on a random number generator.

Advantages	**Disadvantages**
- **Don't Repeat Yourself:** This Utils class allows us to avoid repeating the random event generation code in every method that deals with a probability	- **Reduce Dependencies:** Many classes will now depend on this Utils class.

Jump Design Rationale



Sequence Diagram for the Jump Feature

Key Decision: JumpOnAble Interface and JumpActorAction Class

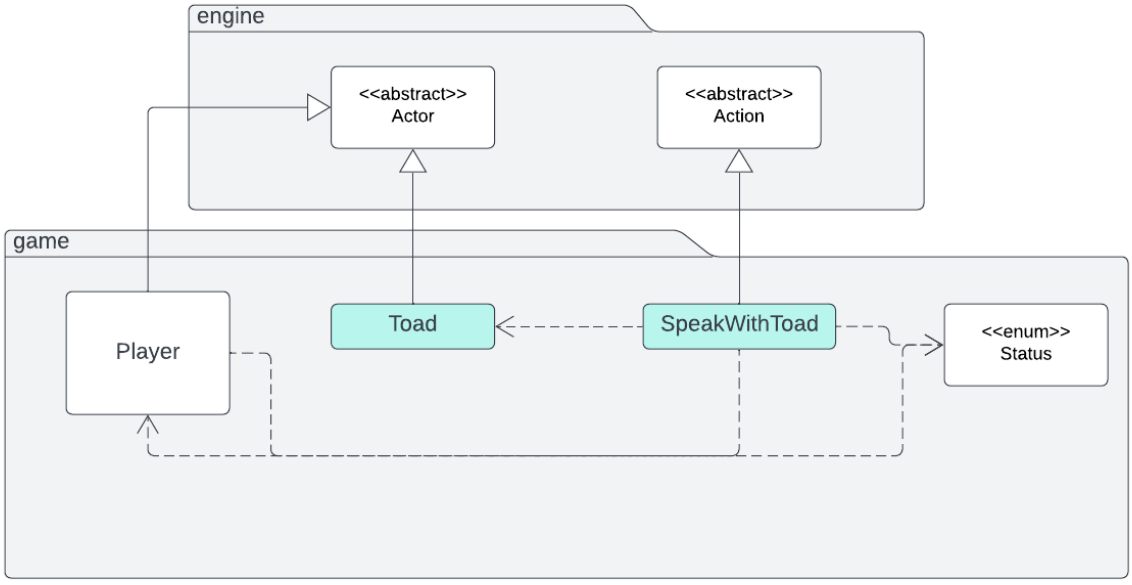
A JumpOnAble interface will be implemented by all Ground types that can be jumped on. They will implement a didJumpSucceed() that will return the fall damage incurred by the jump (0 if the jump succeeds). **It will also check player capability to allow for any special abilities regarding jumping success and damage.

The JumpActorAction class will extend the action class. Its execute method will be responsible for performing the jump – deducting player HP if the jump fails and moving the player if the jump succeeds.

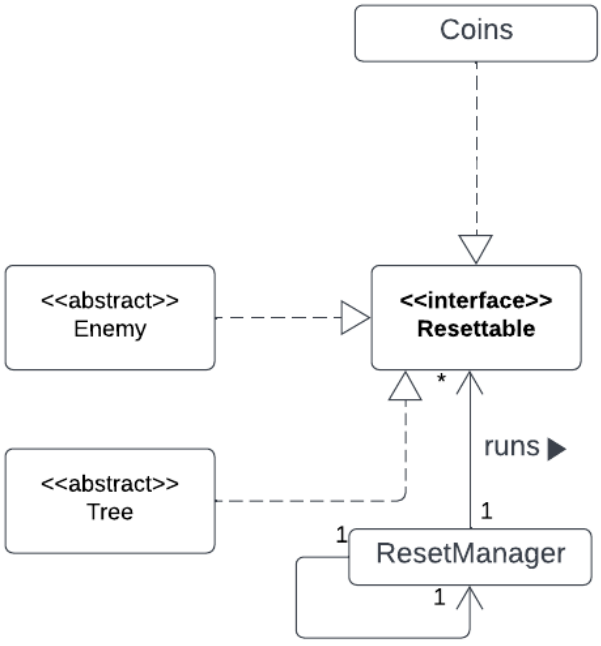
Advantages	**Disadvantages**
- **Open Closed Principle:** Each ground type is responsible for its success rate and fall damage. This means that if we want to extend our system by adding more jump-on-able ground types, we don't need to change the Player or JumpActorAction code. The new ground type will simply implement the JumpOnAble interface.	- **Don't Repeat Yourself:** This implementation means we have to add the JumpActorAction to allowable actions for each JumpOnAble ground. This will result in a small amount of repeated code in the allowableActions method for each JumpOnAble Ground.

Monologue Design Rationale

The monologue of Toad will be contained within the SpeakWithToad class that is inherited from action as this option must be available when Mario is standing adjacent to Toad. This class will hold a randomly generated number from 1-4 and pick a line depending on that number using a switch case. However, the special cases with the wrench and super star would change the possible outcomes as outlined in the brief. To obtain this information the SpeakWithToad class must have a dependency on the Player class so it can read the status and item of the player.



Reset Design Rationale



The reset will be conducted by the ResetManager class which contains an arraylist of objects that are resettable. Objects that are resettable will implement the Resettable <<interface>>. When the reset function is called from the list of actions. The class will run through the array list and depending on the type of class it will remove or do some other

function that is specified in the brief. This will be controlled by a switch case as it would be easier to debug and understand the code.

Advantages	**Disadvantages**
By creating an association it prevents the ResetManager from having to iterate through every single map location and checking for what object is within that location and adjusting it, which would be much slower. It also follows the **Interface Segregation Principle:** only classes that are affected on reset have a reset method, others do not.	The largest disadvantage to this method is the requirement for many classes to implement the resettable interface. This has the potential for **repeated code.**