Assignment 1A - UML & Sequence Diagrams
MA Lab11 Team 1 Version 2

Group Members:

1. How Yu Chern
2. Eugene Kah Chun Fan

Work Breakdown Agreement (WBA)

Version 3 (Assignment 2 - Version 1)
Last Updated: 3/5/2022

Group Members Signature:
1) How Yu Chern - I accept this WBA
2) Eugene Kah Chun Fan - I accept this WBA

Task Breakdown and Summary:

- Each person in charge of a requirement (REQ) will be in charge of it's respective UML Diagram and Design Rational components. Each member's components will be combined to create the complete version of the UML Diagram, and Design Rational for all 6 Requirements. This will be done in each member's own time until the deadline. Each person's work will be reviewed by other members of the group, to ensure that it is complete and up to date.
- Each person will pick one interesting feature in one of the 6 requirements, and will create a sequence diagram out of it.
- For Group Tasks, members will meet up and collaborate, working together to complete the groups tasks of the requirements.
- This WBA, and all 6 Requirement's Diagrams and Design Rational will combined into a single PDF for submission.

Tasks for How Yu Chern
Review By: Eugene Fan
Deadline: 2/5/2022

Tasks:
REQ1: Add Trees to Terrain
REQ2: Jump Ability

Tasks for Eugene Fan
Review By: How Yu Chern
Deadline: 2/5/2022

Tasks:
REQ3: Enemies
REQ4: Magical Items

Group Tasks

- These tasks will be done and will be reviewed by both group member.

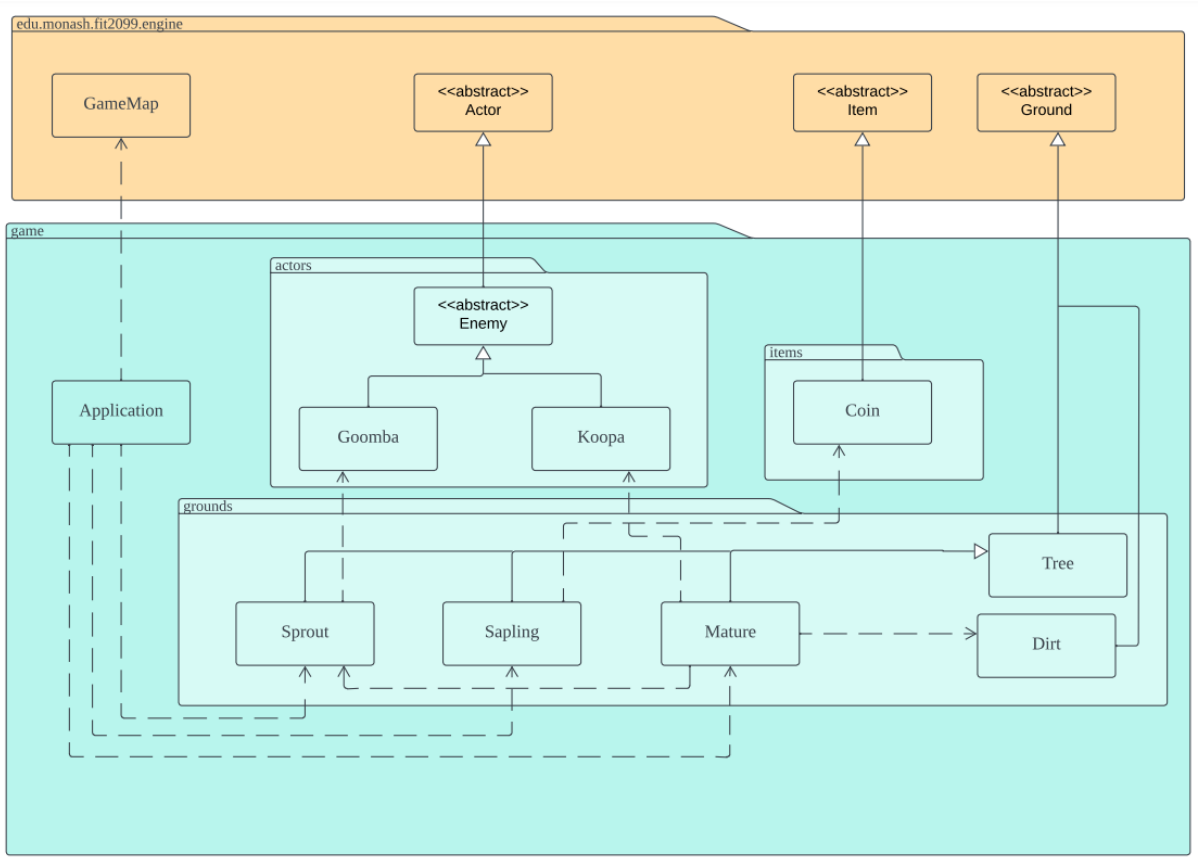
MA_Lab11Team1_V2

- Deadline: 3/5/2022

Tasks:
REQ5: Trading
REQ7: Reset Game

UML Diagrams

REQ 1: Let It Grow



REQ1 Design Rationale:

GroundsLocation is a class which extends location, and keeps track and manages a ground type at a specific location. It will keep track of the tick, and set the ground type accordingly. This allows trees on the map to change based on their life cycles.

A new class GroundsManager which extends Map is created for the purpose of keeping track of specific kinds of Grounds on the GameMap, such as Trees without removing or changing the original functionality of GameMap. It is initialised and used in application, similar to GameMap. It can also randomly pick places to create sprouts during the initialization of the map, as well as cap the total number of trees to prevent overspawning. GroundsManager keeps track of GroundsLocations (as GroundsLocations extends Location). GroundsManager will also aid in executing the behaviour of each type of Ground, such as the spawning of Goombas for Sprouts, Coins for Saplings, and Koopas for Mature, and many more. As GroundsManager keeps track of every Mature's location, Every 5 ticks (turns) it will also create Sprouts in its adjacent dirt squares if there are any, modelling the growth of new sprouts from Matures.

In this design, the tree class has been replaced with three classes of it's types, which are Sprout, Sapling, and Mature, which all extend Ground. Each type of tree keeps track of it's own age each tick, know's it's age limit, and also manages an enum that represents its status (GROW, NORMAL, DEAD). Once the tree reaches it's age limit, it would make a call

to GroundsLocation to change it to it's next stage of growth (or death in Mature case, which the ground will be changed to dirt). Even though Sprout, Sapling and Mature count as Trees, making Tree class an abstract class and extending it would create multiple layer inheritance, which has various problems, such as difficulty debugging and the Diamond Problem, so that option is not used here.

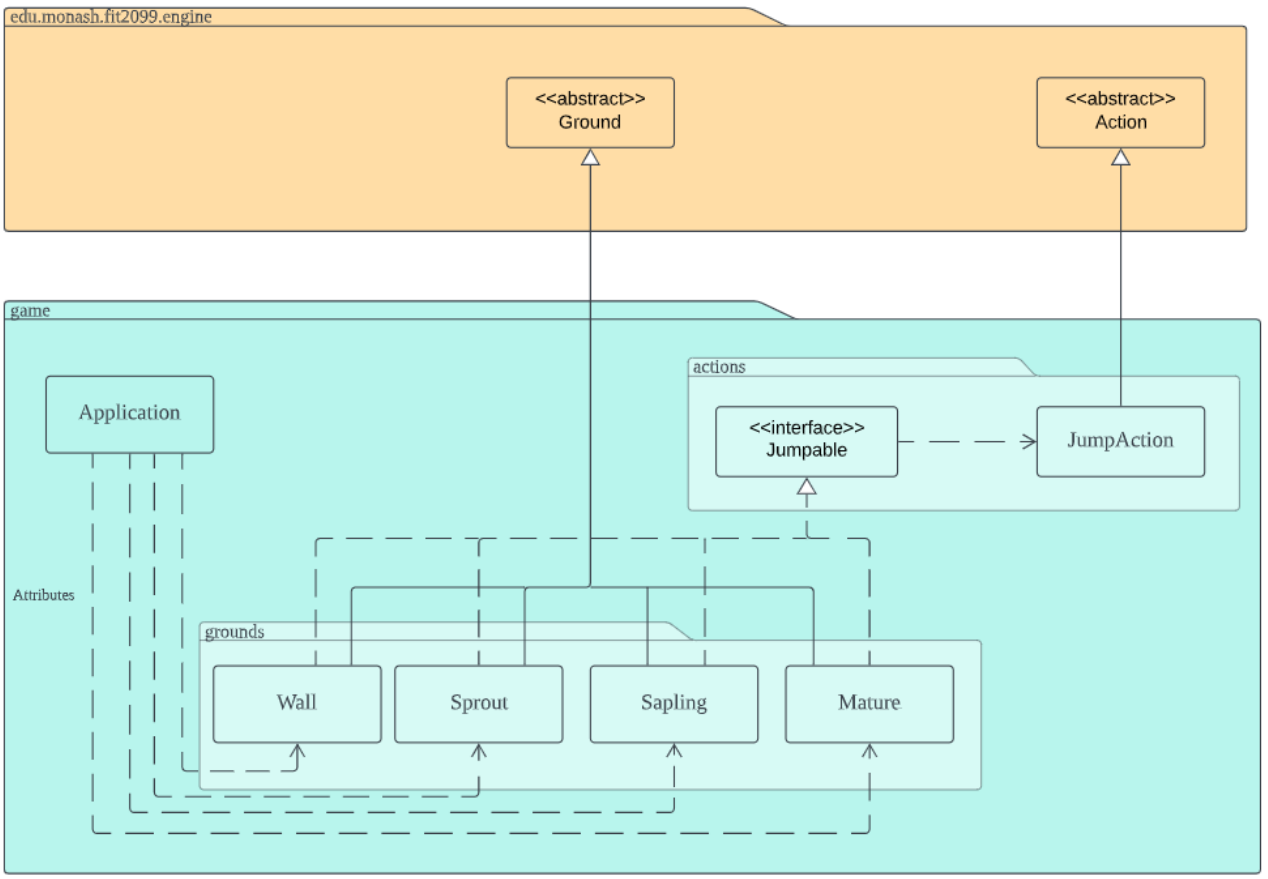
Main differences between Assignment 1 and Assignment 2

The UML Diagram has been updated so that Sprout, Sapling and Mature extend the Tree class, and thus also indirectly extend Ground class through Tree. The Sprout Sapling and Mature inherit the properties of the Tree Class, and implement their own behaviours.

Sprout, Sapling and Mature now have dependencies to Goomba, Koopa, and Coin, where Goomba and Koopa now extend Enemy abstract class, which extends Actor abstract class. This is because Goomba, Koopa, and Coin are now created via their constructors from Sprout, Sapling and Mature respectively.

Mature also has a dependency to Dirt as it replaces itself with Dirt by calling Dirt constructor at a certain point in time. Groundsmanager and Groundslocation have been removed as the location of the ground can be obtained from tick, and the creation of random trees on the game map can be done from Application.

REQ2:



MA_Lab11Team1_V2

REQ2 Design Rationale:

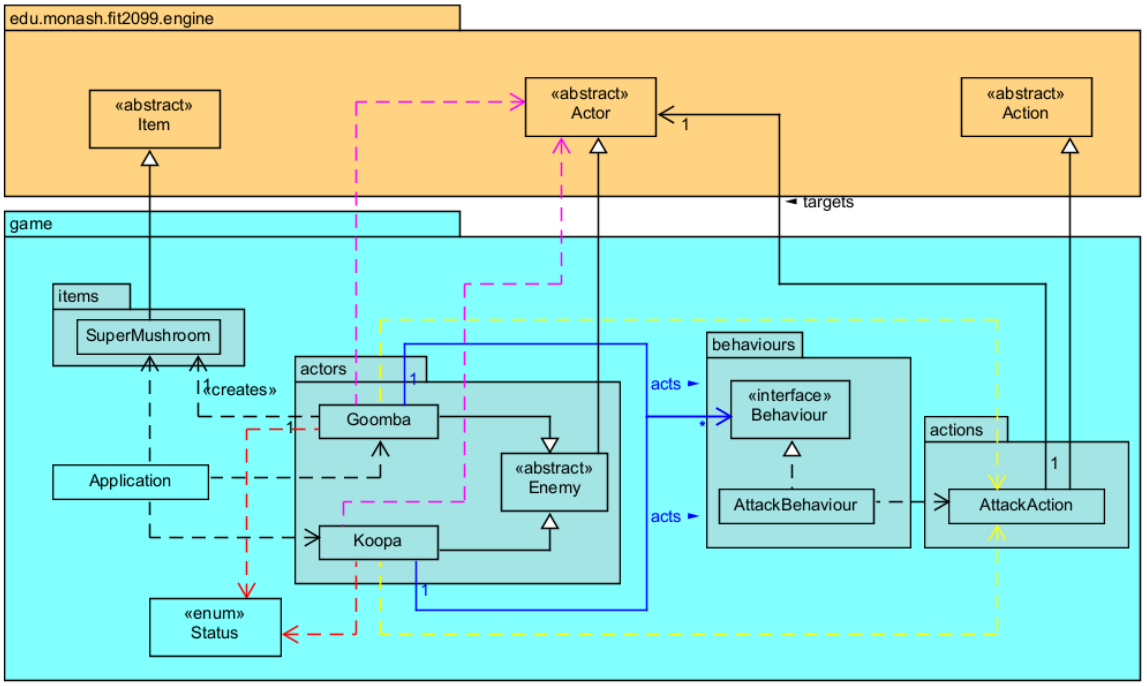
Wall, Sprout, Sapling, Mature (which extend Ground and inherit its properties) are considered higher ground, and thus implement the Jumpable interface, which has a default method that creates JumpAction for the player if the player is near any of the 4 ground types. JumpAction allows the player to go to the higher ground by jumping onto them. This is because the four ground types are higher than normal ground and require the player to jump onto them.

If player chooses to jump onto the ground, it will call canActorEnter to check if the actor can enter it. The Jumpable interface will specify the behaviour that each ground type requires a certain success rate for the player to jump onto, and inflicts a certain fall damage to the player for a failed jump. Each of the four ground types stores its own success rate and fall damage.

If the player meets requirements, the player can jump. Or if the player's display character is "M", which the interface will also allow the player to jump onto any of the four grounds due to super mushrooms giving players 100% jump success rate.

Creation of the Jumpable interface and having the four ground types extend it aims to reduce excessive dependency of having 4 grounds use the default method of the interface to create jumpaction, rather than with repeated code. The Jumpable Interface also allows each of the four ground types to implement similar behaviour, but in their own unique ways, allowing each ground type to have varying success rates and fall damage. Thus, this design aims to fulfil the DRY Principle, and the Principle of Classes Should be Responsible For their Own Properties.

REQ 3



REQ3 Design Rationale:

Koopa and Goomba extends the Enemy abstract class, inheriting Enemy's methods and attributes. The Enemy class extends Actor class. Enemy class is created because Actor's who are Enemies will have some methods only enemies have and not all actors have these methods. Those who extend Enemy class will accept Actor class as a parameter in allowableActions() method and in classes like Goomba and Koopa, they will create an AttackAction which allows the other actors to attack the enemies given that the other actor has HOSTILE_TO_ENEMY capabilities (each actor has a capability set).

The enemies both have the ability to attack the Player too. To have that ability to attack, the enemies each have a map (Map table) of Behaviour objects (which is basically what console is to the Player). Behaviours such as WanderBehaviour, FollowBehaviour and AttackBehaviour all help shape what enemies can do. Short note that AttackBehaviour will basically use the codes already prepared in AttackAction, with small adjustments here and there. These behaviour classes are examples of Single Responsibility Principle (SRP) where each type of behaviour is a separate class.

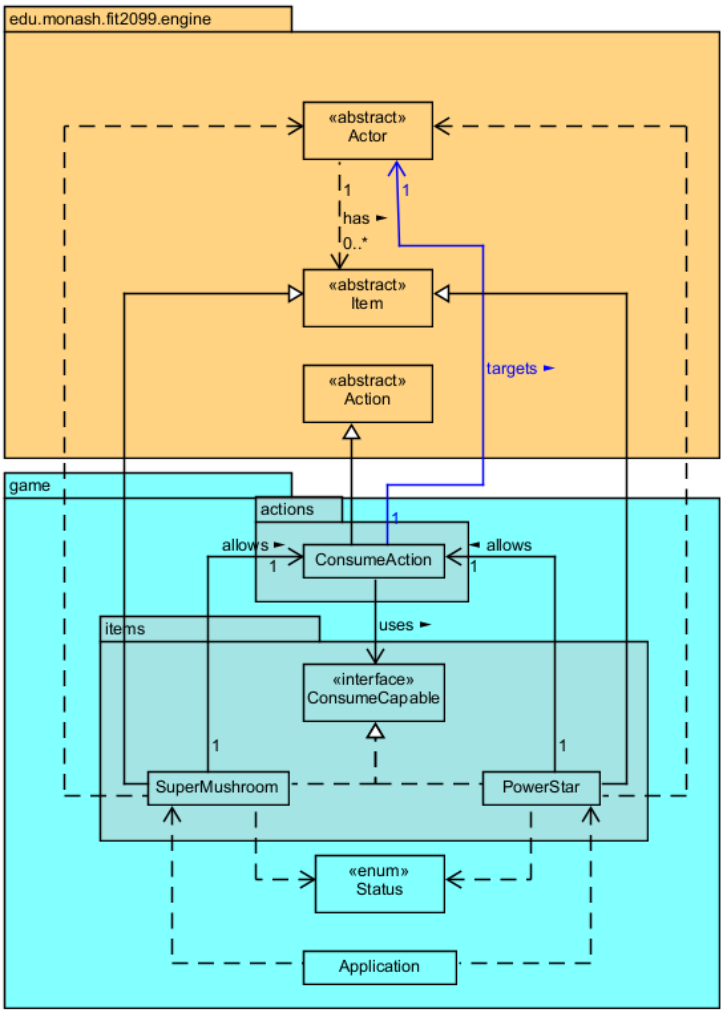
Koopa will also create a SuperMushroom and put it in their backpack. Once Koopa is killed, the SuperMushroom will be dropped for the player to pick up.
The 3 OOP Principles are maintained here. The DRY principle is maintained as having all the enemies extend their respective abstract classes reduces dependencies. Classes are in charge of their own behaviours, for example, Koopa and Goomba managing their own attack actions. There is a minimal amount of literals that are used in this design due to the inheritance factor also, as creating new literals is not necessary.

Main differences between Assignment 1 - V1 and Assignment 1 - V2:

Req3 and Req4 are now separate UMLs to increase readability. Koopa and Goomba now extends Enemy class instead of Actor class in V1 because Enemy will have some similar methods that non-enemy won't have. Enemies will have a hash table of Behaviours (Behaviour interface is now stored in the game folder instead of engine, this was just a mistake at V1, now corrected in V2).

Enemies also will be able to be attacked by otherActors as long as the actors have Status.HOSTILE_ENEMY capability. Enemies will also make use of Status enum to check for capabilities of either itself or otherActors, unlike V1 which does not have Status enum at all.

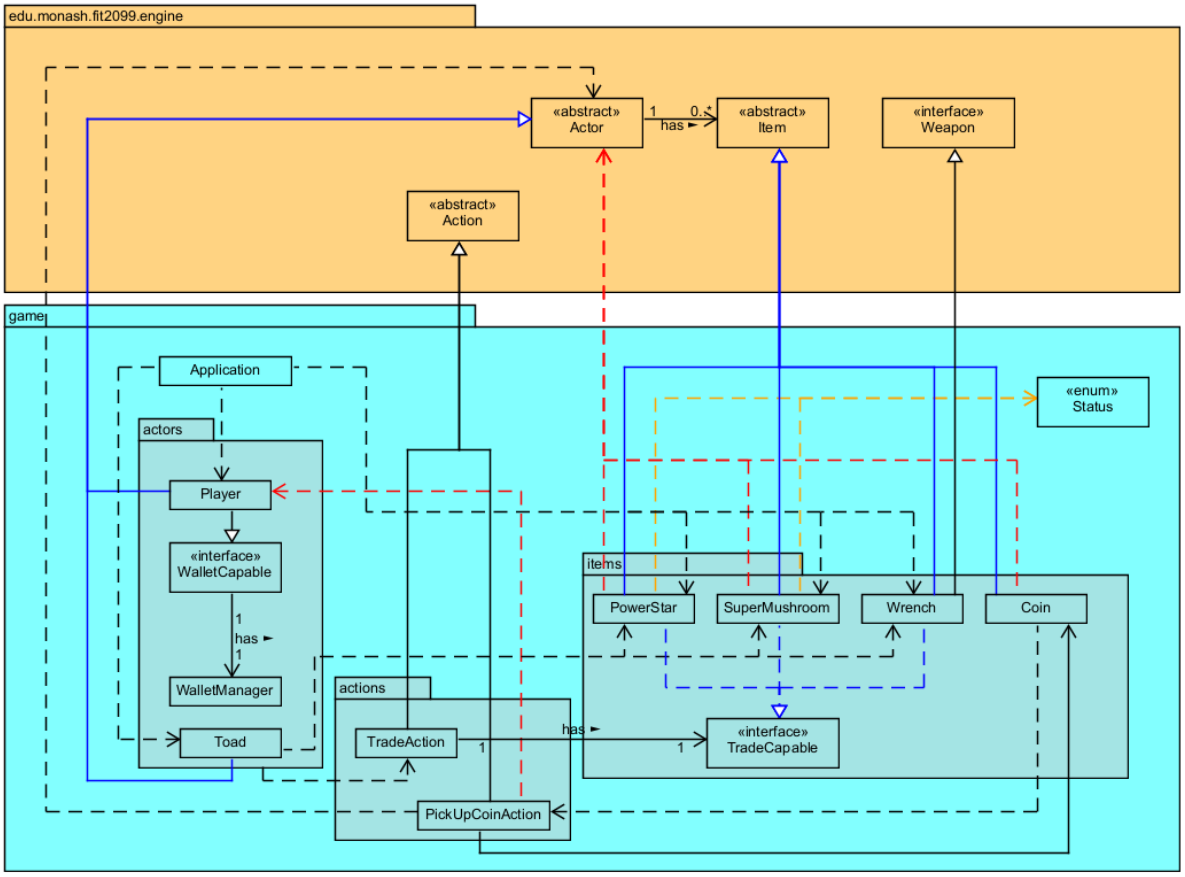
<u>REQ4</u>



REQ4 Design Rationale:
Two new classes (magical items) called SuperMushroom and PowerStar are created, which both extend the Item class and implement the ConsumeCapable interface, and inherit its attributes and methods. This ConsumeCapable interface is useful to identify which Items can be consumed and this adheres to Interface Segregation Principle (ISP) where certain Interfaces are in charge of certain capabilities. These items can be picked up from the ground or can be traded by the Toad (see Req5). Magical items allow other actors to use ConsumeAction upon it. ConsumeAction will target that actor and execute the ConsumeCapable Item's consume() method. Each of those items might have different effects on the actor if consume() method is executed.

Main differences between Assignment 1 - V1 and Assignment 1 - V2:
Again Req3 and Req4 are now separate UMLs to increase readability. PowerStar and SuperMushroom now implements the ConsumeCapable interface in addition to extending Item abstract class. This is to adhere to ISP. Now, the ConsumeCapable items allow ConsumeAction (totally not present in V1) to be executed by Actors. This change is important to allow Actors to reap the benefits of consuming ConsumeCapable items.

REQ 5:



REQ5 Design Rationale:

Coin extends Item, as it can exist on the terrain similar to other items such as PowerStar and SuperMushroom. Each coin will keep track of its own integer value, as according to one of The 3 OOP Java Principle - Classes should be responsible for their own properties. Coin creates a PickUpCoinAction which will handle the Player's action of picking up a coin. The player has its own wallet, and PickUpCoinAction will update the Player's wallet based on the value of the coin before removing the coin on the game map.

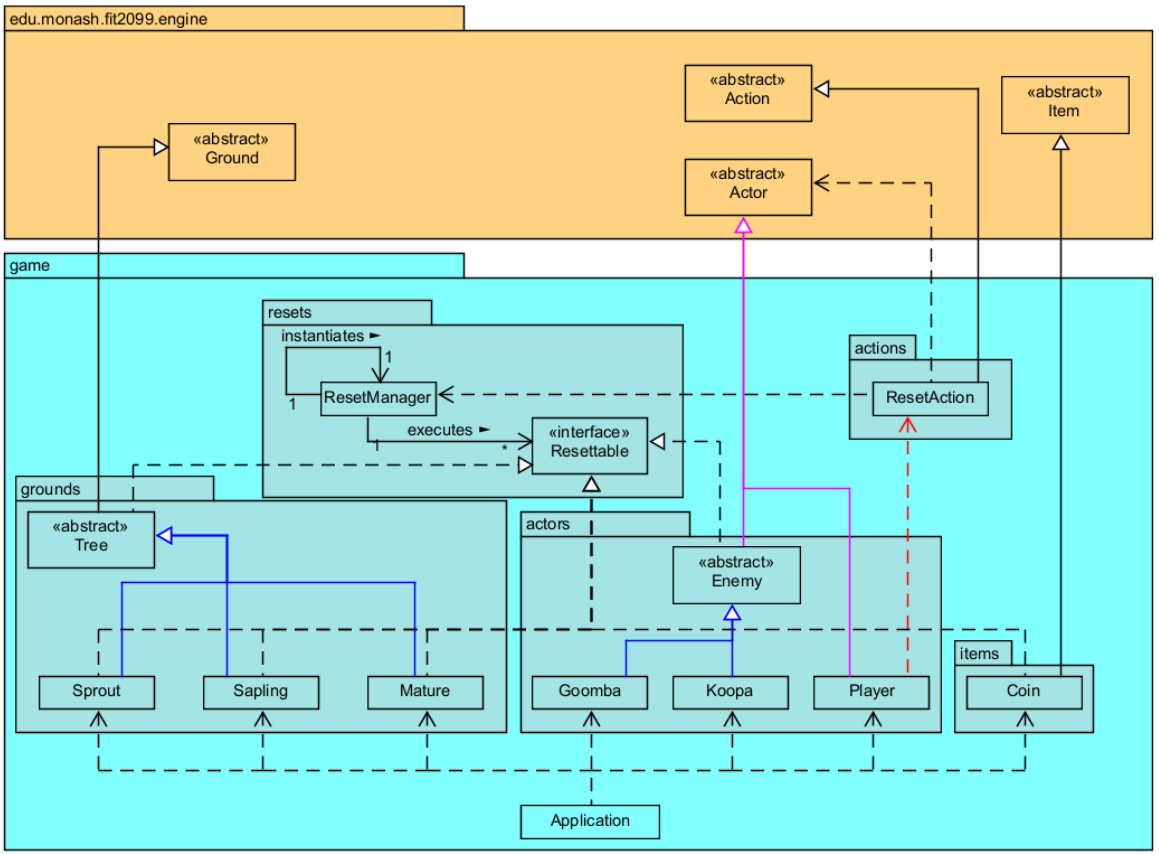
Player wants to have its own wallet, so it will implement WalletCapable interface. Those who implement WalletCapable interface will have a WalletManager class to manage the wallet values. By using interfaces to separate who can have a wallet, this adheres to the Interface Segregation Principle (ISP). WalletManager is used specifically to keep track of the balance of coins the player has picked up. This is because if the player stored coins in the inventory every time the player picked up a coin, the inventory would be occupied by coins, which would make managing other items such as Super Mushroom and Power Star tedious. This also serves to prevent unintended behaviour, such as a player dropping coins, as well as reduce complexity for not having to create a currency system to calculate how many coins to deduct from the inventory when buying items from toad. Thus, fulfilling the DRY Principle, and reducing excessive dependencies to cater for such cases.

Wrench class extends Item class and implements Weapon interface. The Wrench is a weapon item which means it can be used by the user to attack enemies. The Wrench also inherits a very important capability which is the ability to attack and kill Koopas which are in Dormant state.

Toad class extends abstract Actor class. The Toad is a friendly actor as it cannot attack anyone, however it does have the capabilities to initiate TradeActions of TradeCapable items with the player. The items that are sold by the Toad and implement TradeCapable interface are SuperMushroom objects, PowerStar objects and Wrench objects with their respective prices.

Main differences between Assignment 1 - V1 and Assignment 1 - V2:
The relationship between Coin and PickUpCoinAction has been changed from inheritance (wrong) to dependency. Application won't initialise Coin because Coin can only be gotten by defeating certain enemies or destroying certain grounds. There is now a TradeCapable (not present in V1) interface which is implemented by Wrench, SuperMushroom and PowerStar. This will prepare these items to be traded later on when TradeAction is called upon said item towards actors. Wrench now extends Item and implements Weapon interface (previously just inherits WeaponItem).

<u>Req 7:</u>



REQ7 Design Rationale:

MA_Lab11Team1_V2

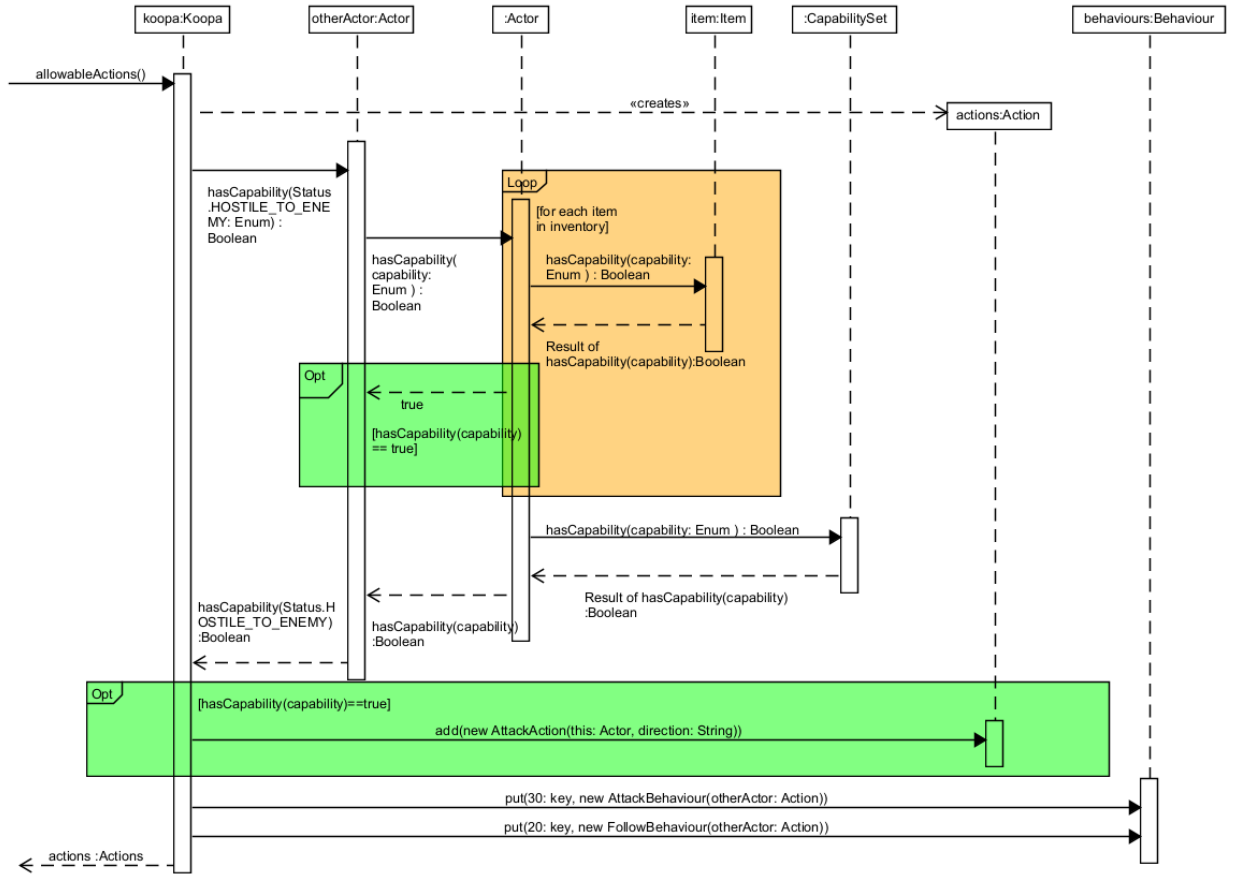
All classes which need to be reset implements the interface Resettable. In this case, all of the trees, enemies, player, and the item coin. This allows Resettable to keep initial instances of all these classes to be able to reset their current states back to their initial status later. As each of the classes of grounds, actors, and items are resettable, but are reset in different ways (Eg. Grounds have a 50% chance of being converted to dirt, Coins are deleted, enemies are removed from the map , the resettable interface is implemented by all the classes stated above that are required to be reset. Interface Segregation Principle (ISP) is practised as only classes that are resettable will need to implement that interface.

ResetManager will keep track of the reset instances and can run it. ResetAction will be in charge of displaying the reset option to the player.

Main differences between Assignment 1 - V1 and Assignment 1 - V2:
In V2, there are new classes like Enemy abstract class and also ResetAction class. These additions from V1 to V2 are because Goomba and Koopa have similar attributes and methods to each other but have different attributes and methods with non-enemy Actors. ResetAction will be used by Player to execute the reset process (no such thing in V1). By creating ResetAction, this action will be made available at Player's menu console.
In V2, there is a new Tree abstract class which Sprouts, Saplings and Mature extend.

Sequence Diagrams

1. Koopa's allowableActions()



2. PickUpCoinAction's execute()

MA_Lab11Team1_V2