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

Group Members:

1. How Yu Chern
2. Eugene Kah Chun Fan

Work Breakdown Agreement (WBA)

Version 2
Last Updated: 6/4/2022

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: 9/4/2022

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

Tasks for Eugene Fan
Review By: How Yu Chern
Deadline: 9/4/2022

Tasks:
REQ3: Enemies
REQ4: Magical Items

Group Tasks

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

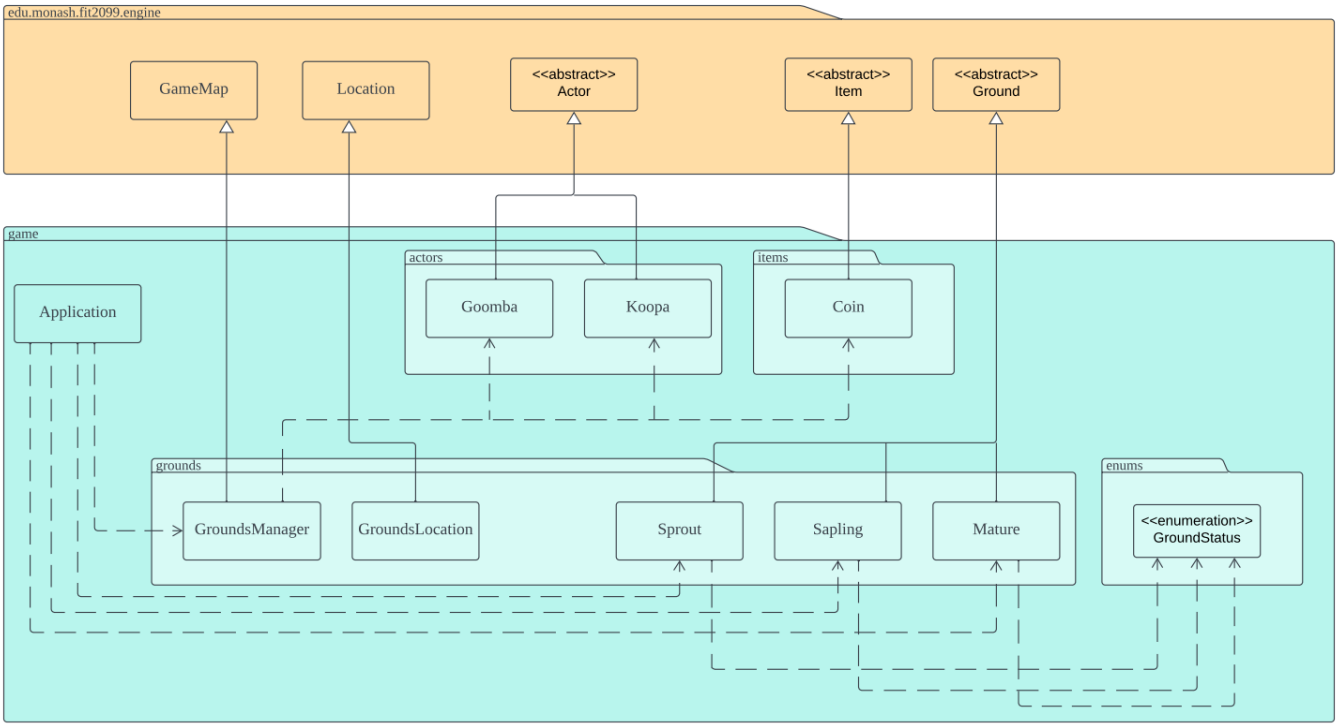
MA_Lab11Team1

- Deadline: 10/4/2022

Tasks:
REQ5: Trading
REQ7: Reset Game

UML Diagrams

REQ 1: Let It Grow



REQ1 Design Rationale:

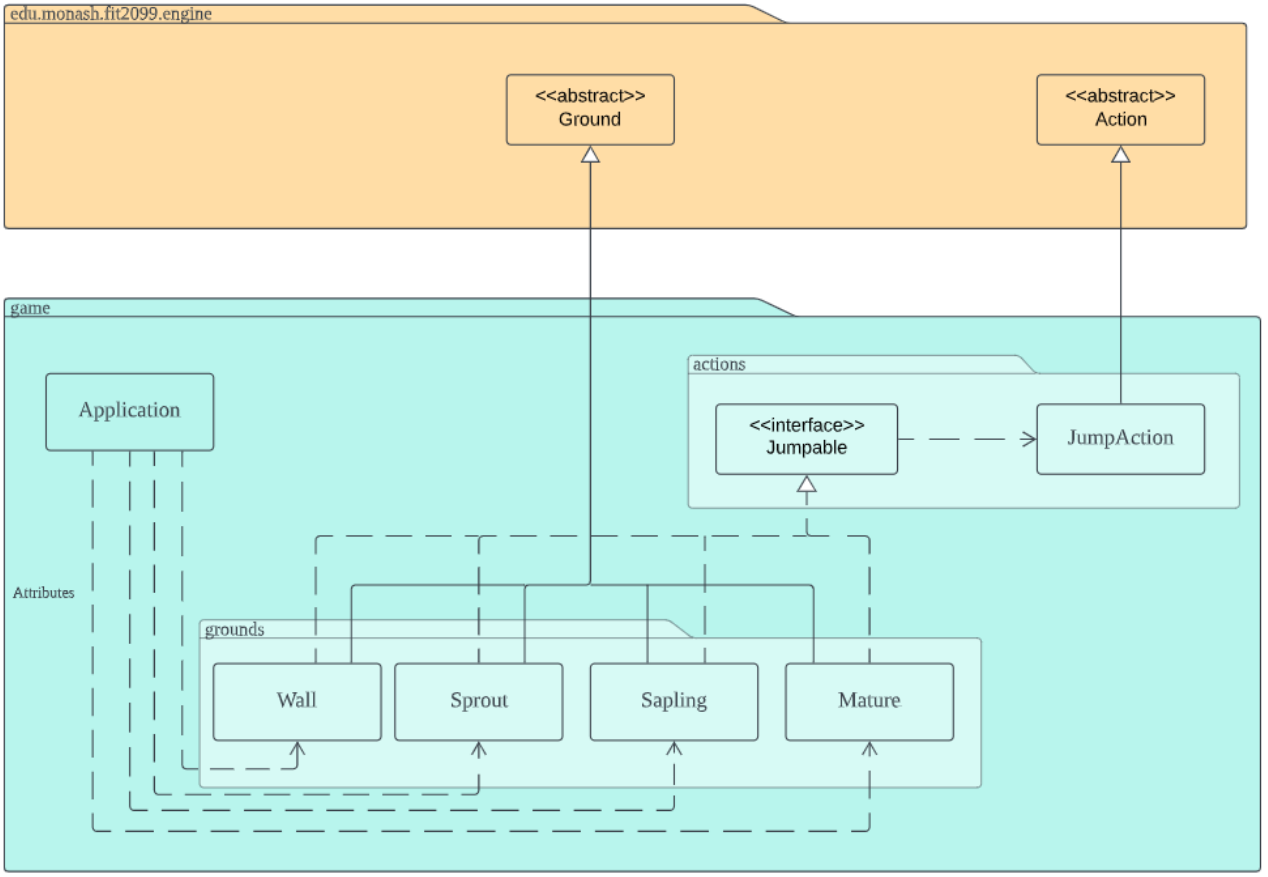
GroundsLocation is a class which extends location, and keeps track and manages a ground type at a specific location. It will keep track of the tick, and set the ground type accordingly. This allows trees on the map to change based on their life cycles.

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.

REQ2:



REQ2 Design Rationale:

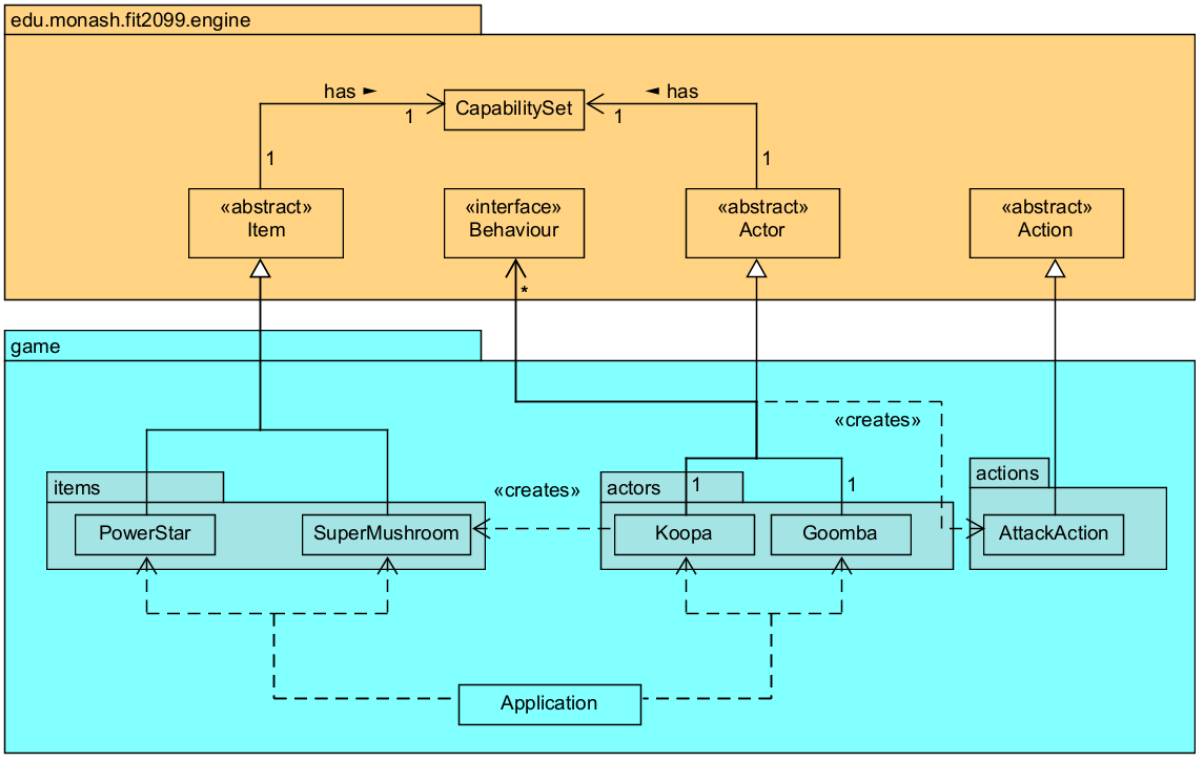
Wall, Sprout, Sapling, Mature (which extend Ground and inherit its properties) are considered higher ground, and thus implement the Jumpable interface, which has a default method that creates JumpAction for the player if the player is near any of the 4 ground types. JumpAction allows the player to go 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.

REQ3 & REQ4



***I have combined Req 3 & Req 4 ‘s UML together as the UML created is not too difficult to understand together.

REQ3 and REQ4 Design Rationale:

Koopa extends the actor class, inheriting actor’s methods and attributes. Koopa is an enemy actor who has different characteristics as compared to Goomba who is also an enemy actor. Both the enemies create an attackAction which allows the Player to attack the enemies given that the other actor has HOSTILE_TO_ENEMY capabilities (each actor has a capability set).

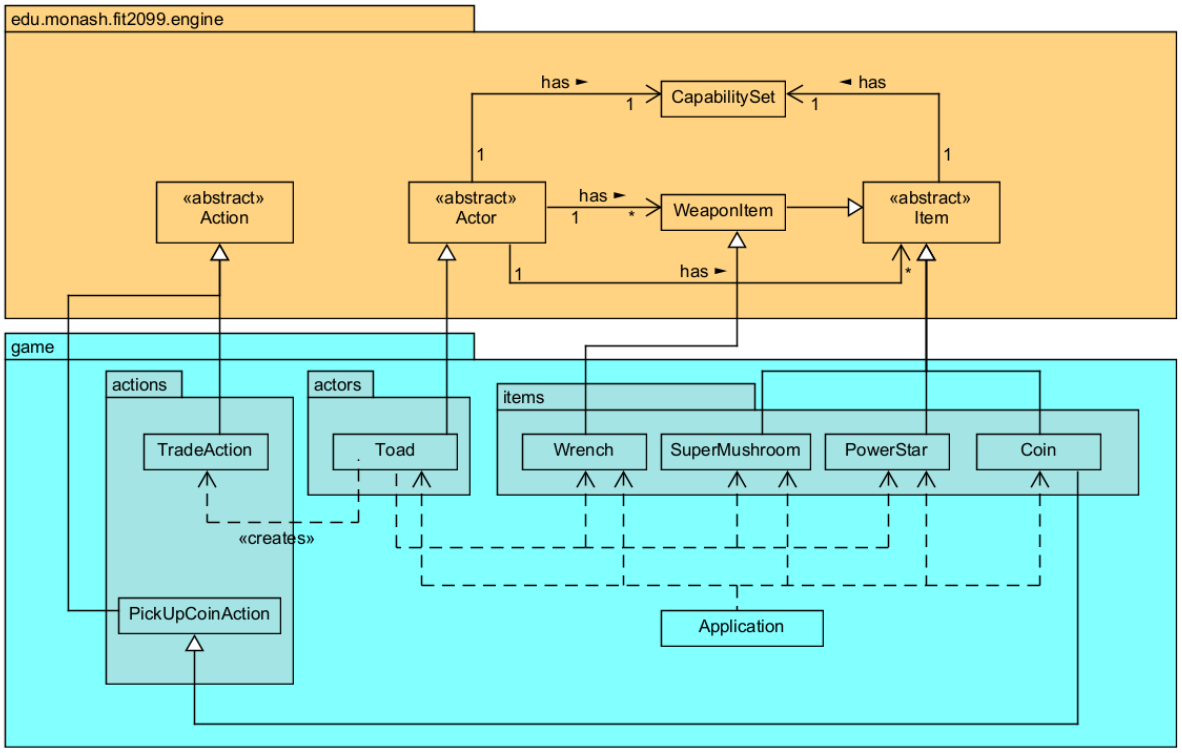
The enemies both have the ability to attack the Player too as they both have HOSTILE_TO_ENEMY capabilities. To have that ability to attack, the enemies each have a map (Map table) of Behaviour objects (which is basically what console is to the Player). Behaviours such as WanderBehaviour, FollowBehaviour and AttackBehaviour all help shape what enemies can do. These behaviour classes are examples of 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.

Two new classes (magical items) called SuperMushroom and PowerStar are created, which both extend the Item abstract class, and inherit it's attributes and methods. Since both magical items are Item objects, they also have their own capability set. This means when Player obtains these magical items, the Player will also inherit their capabilities. These items can be picked up from the ground or can be traded by the Toad.

The 3 OOP Principles are maintained here. The DRY principle is maintained as having all the items and all the actors 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.

REQ 5:



REQ5 Design Rationale:

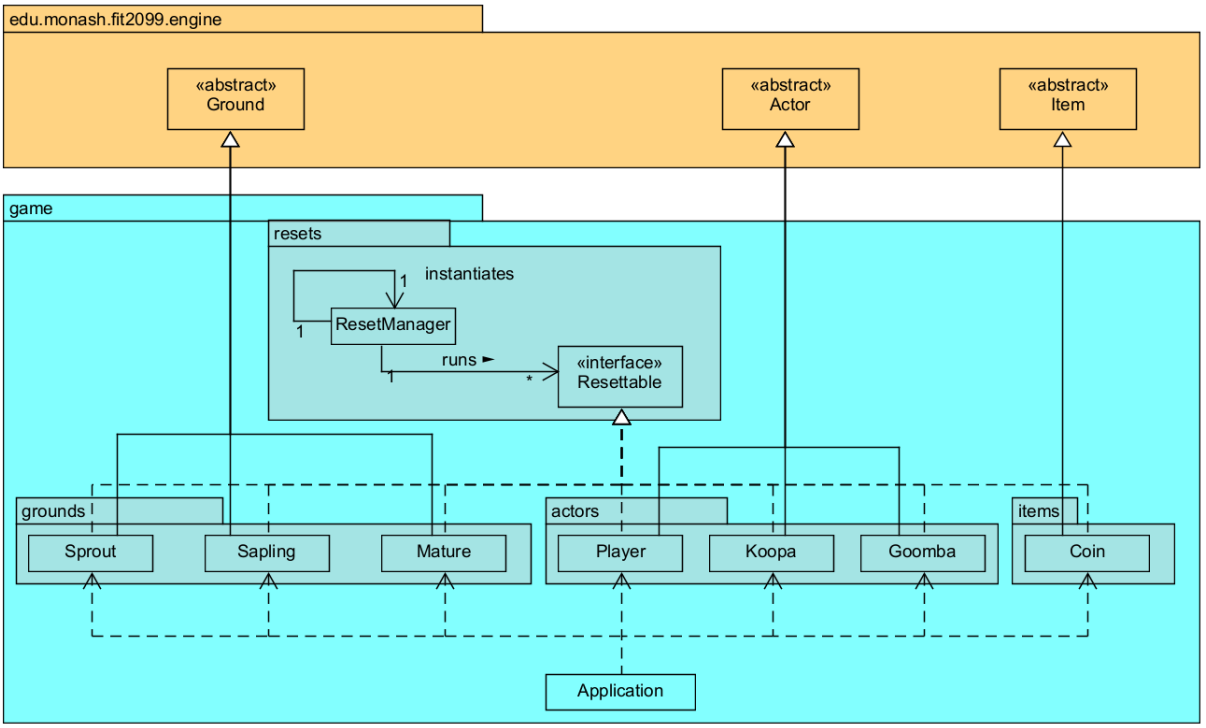
Coin extends Item, as it can exist on the terrain similar to other items such as PowerStar and SuperMushroom. Each coin will keep track of its own integer value, as according to one of The 3 OOP Java Principle - Classes should be responsible for their own properties. Coin creates a PickUpCoinAction which will handle the Player's action of picking up a coin. The player has it's 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 has its own wallet, separate from inventory, 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 WeaponItem class. 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 trading of items with the player. The items that are sold by the Toad are SuperMushroom objects, PowerStar objects and Wrench objects with their respective prices.

Req 7:



REQ7 Design Rationale:

All classes of all grounds, actors, and the item coin which need to be reset implements the interface Resettable. 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, Goombas are removed from the

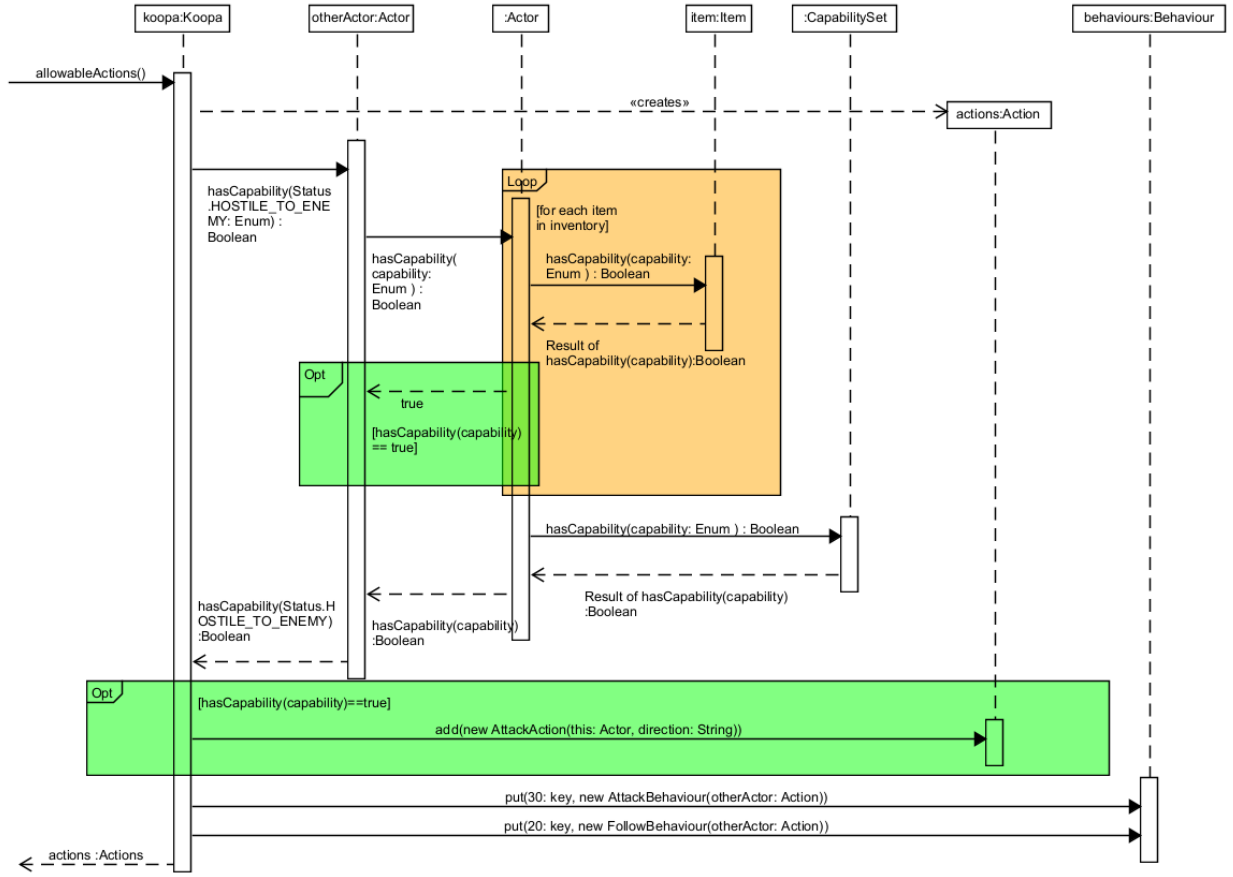
map via suicide, Koopas are killed and retreats into their shells), the resettable interfaces is extended by all the classes stated above that are required to be reset.

ResetManager will keep track of the number of resets to limit the reset to one time only. It will also be in charge of displaying the reset option to the player.

MA_Lab11Team1

Sequence Diagrams

1. Koopa's allowableActions()



2. PickUpCoinAction's execute()