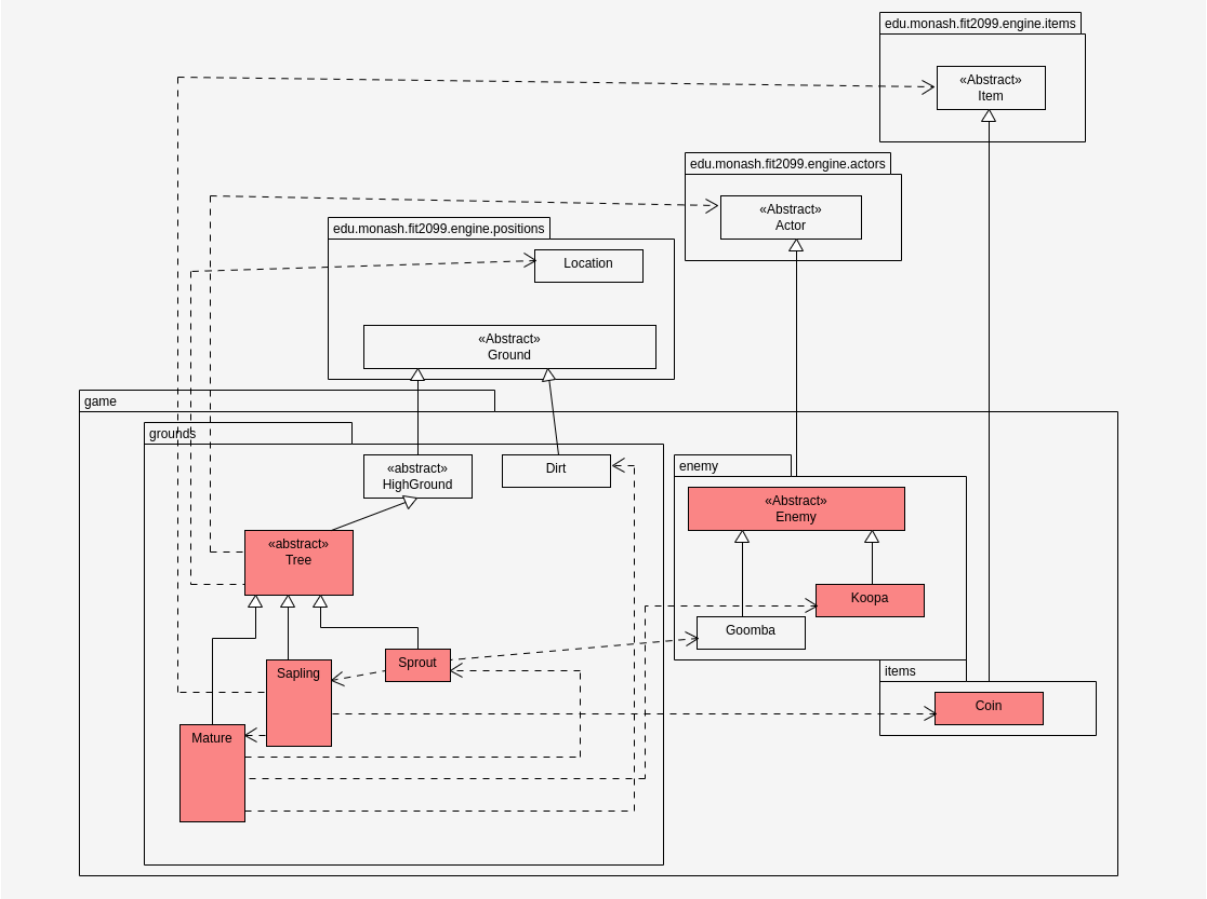
FIT 2099 Assignment 2 Design Rationale
Lab 7 Team 1

Lab Tutor: Dr. Tan Choon Ling
Group Members:
Khor Jia Shin 32356595
Lim Hong Yee 32455836
Fadi Alailan 31844936

REQ 1

Class Diagram



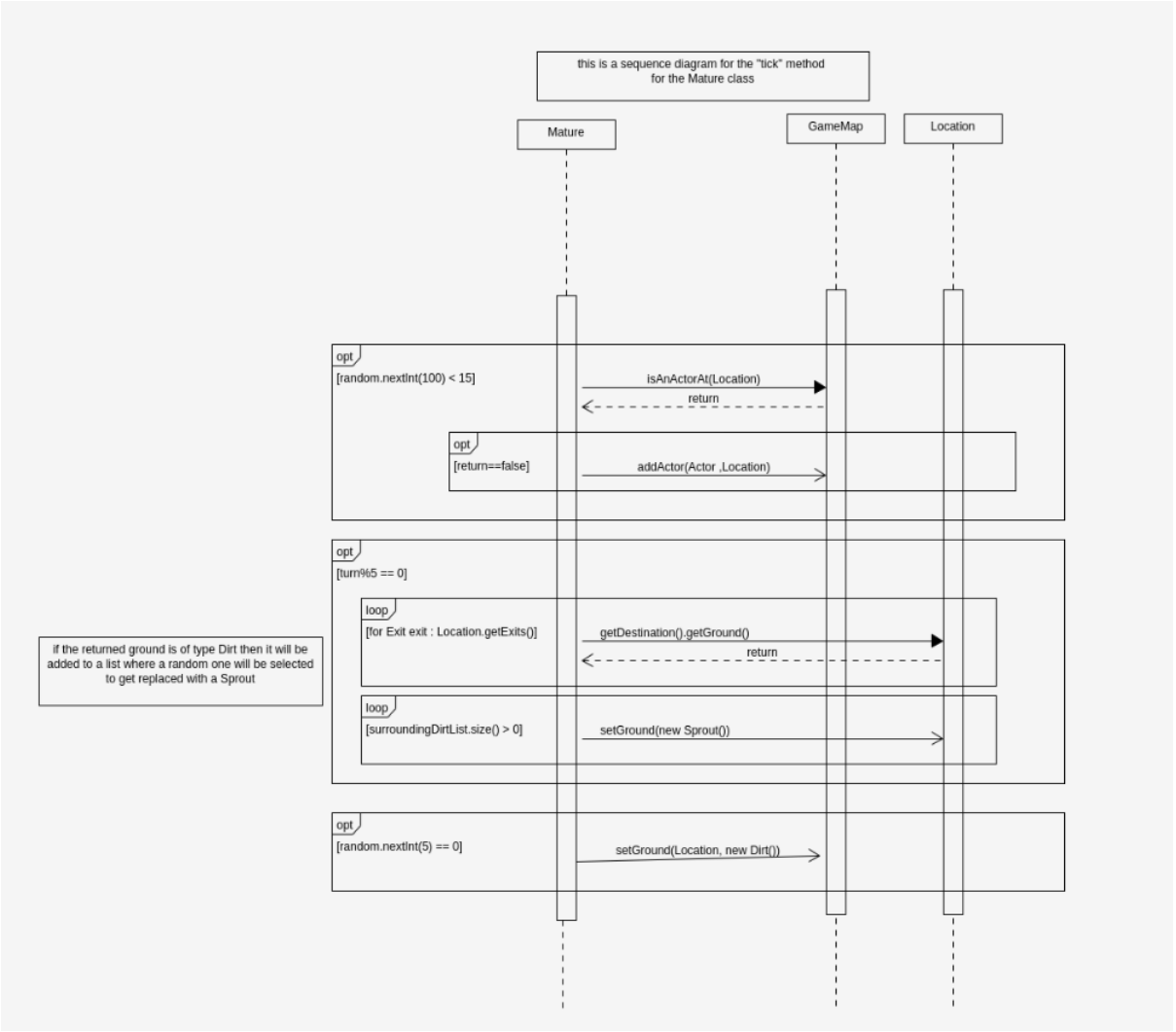
Tree:

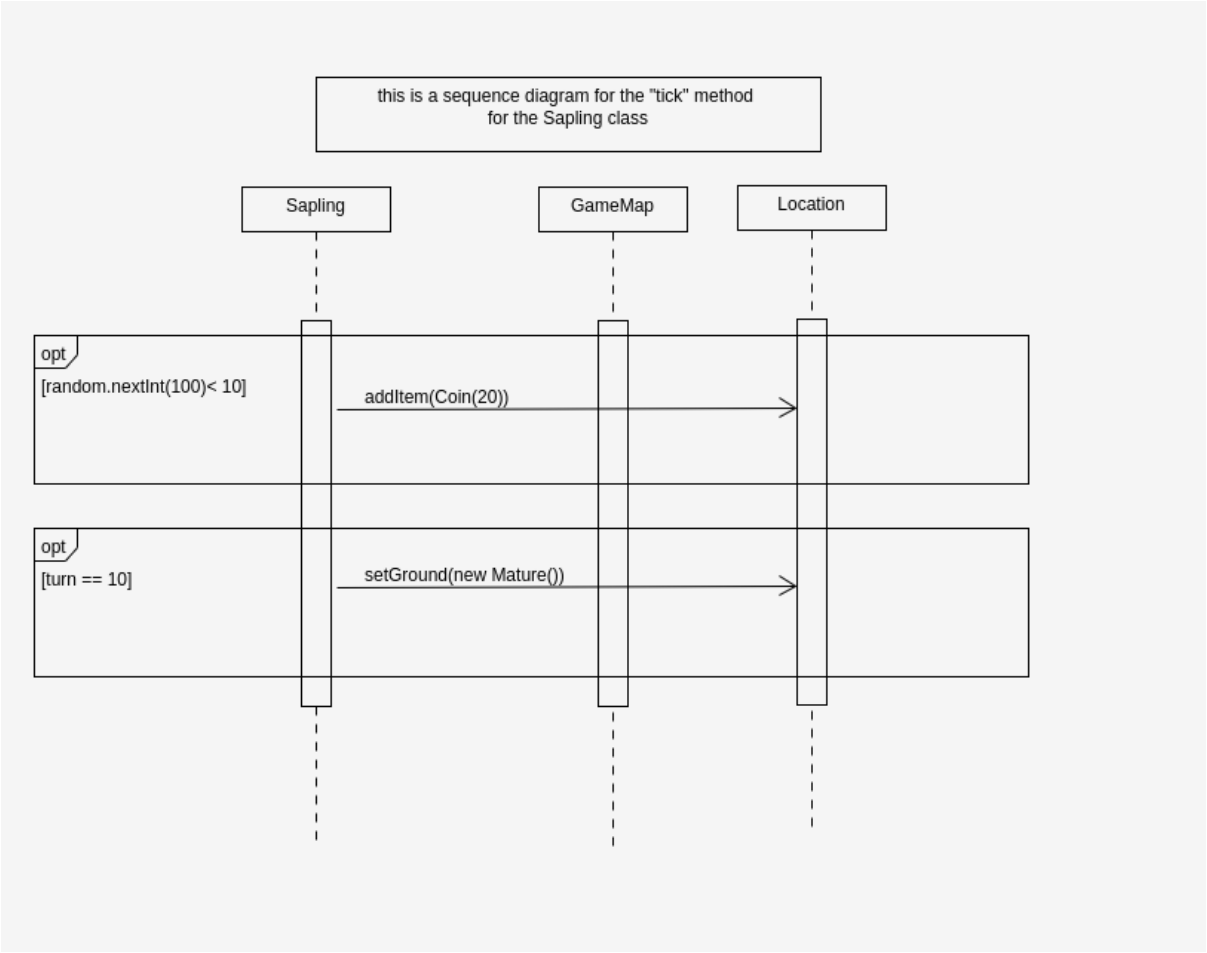
remove the dependency on GameMap
remove the dependency on Item
remove the dependency on Ground

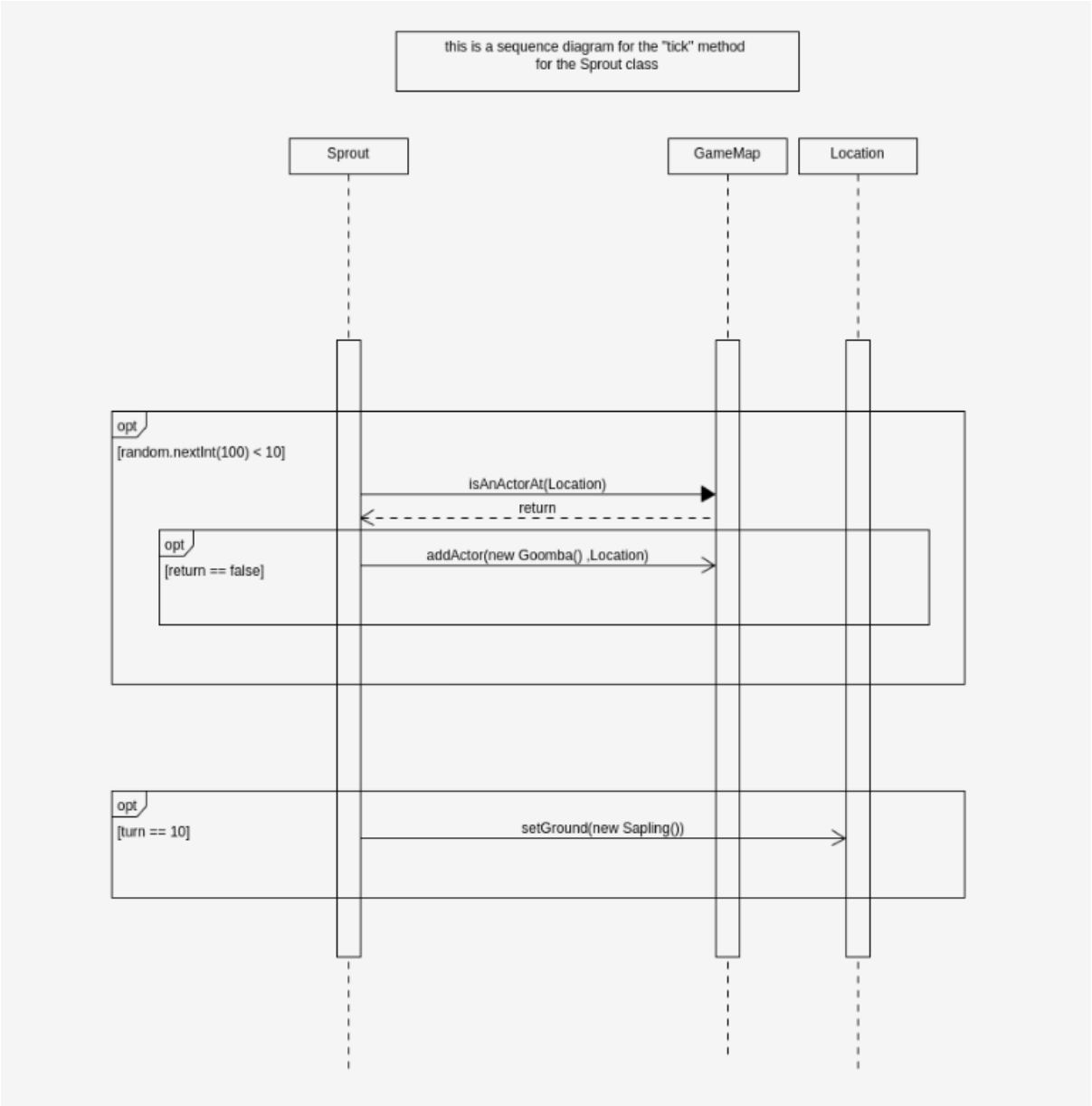
Sapling:

add a dependency on Item

Sequence Diagram


this is a sequence diagram for the "tick" method
for the Mature class
Mature
GameMap
Location
opt
[random.nextInt(100) < 15]
isAnActorAt(Location)
return
opt
[return==false]
addActor(Actor ,Location)
opt
[turn%5 == 0]
loop
[for Exit exit : Location.getExits()]
getDestination().getGround()
return
loop
[surroundingDirtList.size() > 0]
setGround(new Sprout())
opt
[random.nextInt(5) == 0]
setGround(Location, new Dirt())
if the returned ground is of type Dirt then it will be added to a list where a random one will be selected to get replaced with a Sprout


this is a sequence diagram for the "tick" method
for the Sapling class
Sapling
GameMap
Location
opt
[random.nextInt(100)< 10]
addItem(Coin(20))
opt
[turn == 10]
setGround(new Mature())



Design Rationale

Enemy:
an abstract class that will be added in REQ3, added here for consistency

HighGround:
an abstract class that will be added in REQ2, added here for consistency

Tree:
Tree is an abstract class that is meant to be used to group the 3 main Tree types (Sapling, Sprout. Mature), The main reasons for this class is to not violate the "Single Responsibility Principle" and to make the code more organised. Trees are part of the

terrain, so the Tree class inherits from Ground (HighGround in REQ2), it depends on Location because we use the "tick" method, and it depends on GameMap because it will need to access the Location's GameMap. It depends on actor and item because its children will need to spawn these objects

Why did I remove the dependency on GameMap from Assignment 1?

it turned out that all the methods I need were in the Location class, so there was no need to use the GameMap class

Why did I remove the dependency on Item?

because only Sapling has a dependency on Item and not Sprout or Mature

Why did I remove the dependency on Ground?

because the I noticed the class inherits from HighGround which inherits from Ground

Sprout:

it is a class that inherits from Tree because it is one of the 3 stages of a Tree's life (the other 2 being
Sapling and Mature). It spawns Goomba ,so it depend on them, and it depends on Sapling because it will turn into one after 10 turns

Sapling:

it is a class that inherits from Tree because it is one of the 3 stages of a Tree's life (the other 2 being Sprout and Mature). It spawns Coins ,so it depends on them, and it depend on Mature because it will turn into one after 10 turns

Why did I add a dependency on Item?

because sapling is the only one that makes use of Item to spawn Coins but not the other tree type, so it was moved from Tree to Sapling

Mature:

it is a class that inherits from Tree because it is one of the 3 stages of a Tree's life (the other 2 being Sprout and Sapling). It depends on Koopa, because it spawns them, and it depends on Sprout because it will spawn them every 5 turns in a random surrounding fertile ground, it depends on Dirt because it has a 20%
chance to turn into one each turn

Why did you implement the fertile ground system using a capability?

when I was thinking of how to implement the system, I came with 2 ideas at first:
a) make an interface called FertileGround and check if the current ground type is from that interface using instanceof
cons: we will be using instanceof which is a code smell
b) make an interface called FertileGround and a class called FertileGroundManager that keeps a List of all FertileGrounds
and can check if a ground is a FertileGround by looking at the list of FertileGrounds and checking if the current ground is there

cons: has the complexity of O(n) where n is the number of FertileGrounds, this is bad since approach (a) has a complexity O(1)

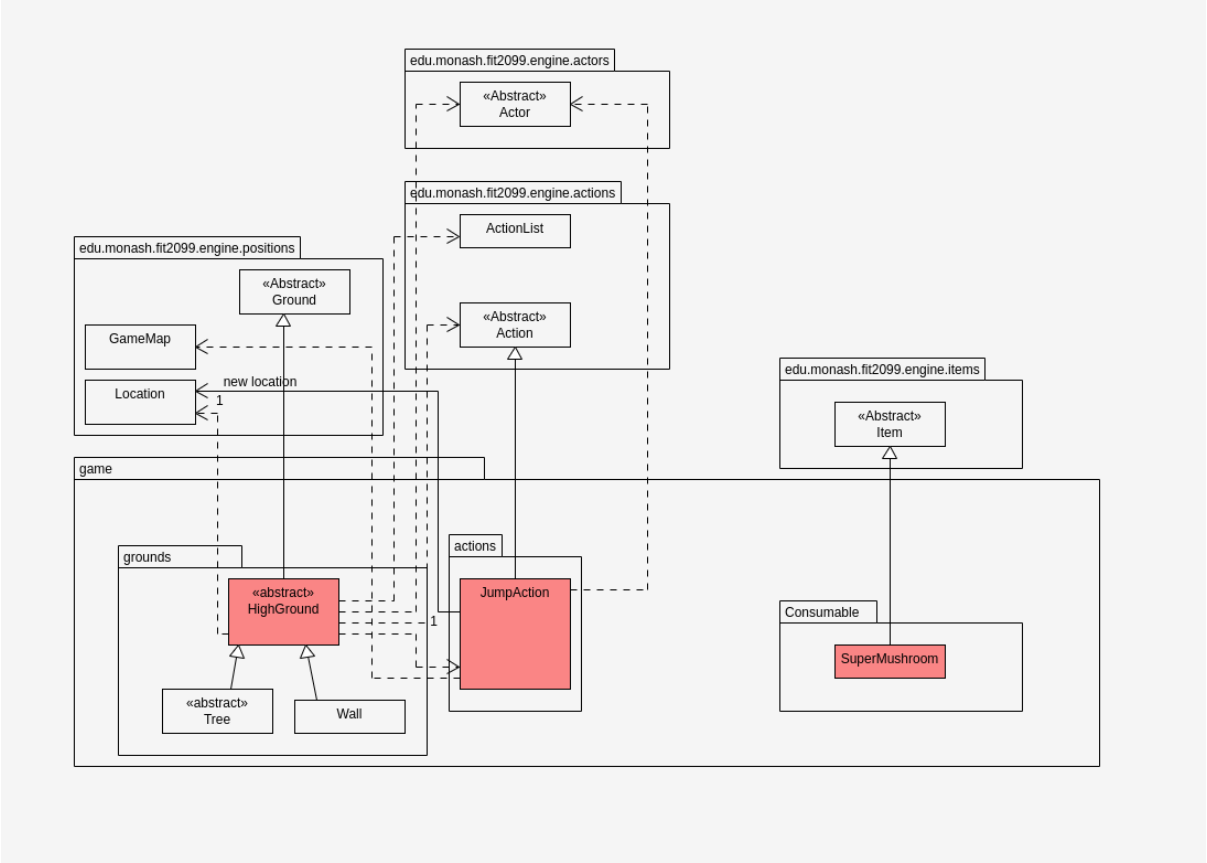
but then I realised that the Ground class supports capabilities, so I decided to add a capability called FERTILE that we can check if a ground has it, and this way we don't use instanceof or sacrifice performance.

Coin:

an Item that if collected gives the player money, it is an Item that can be picked up, so it seems natural that it inherits from Item

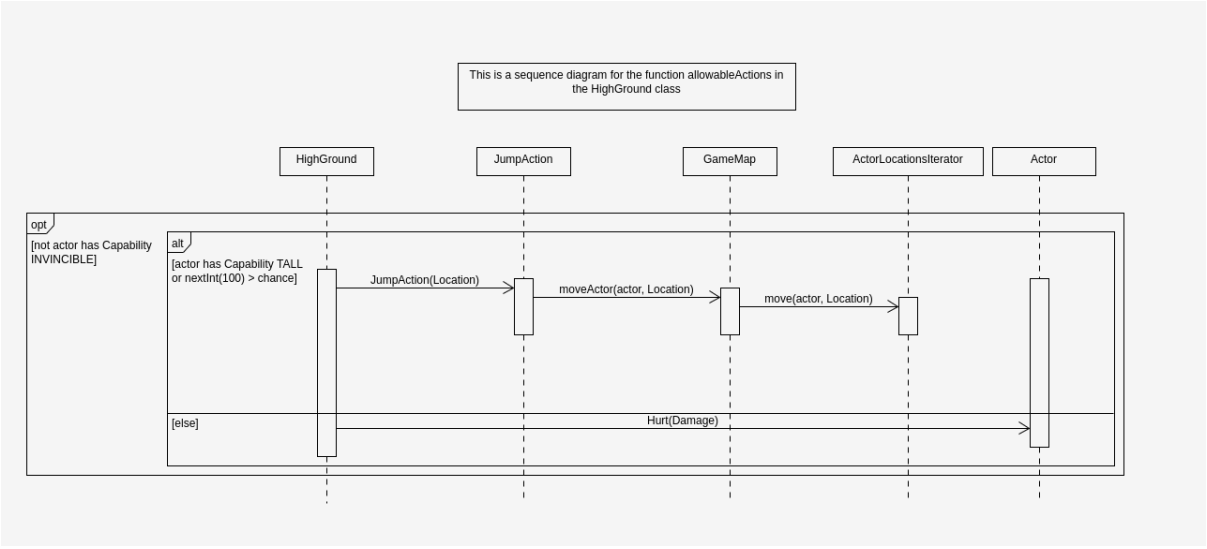
REQ 2

Class Diagram



changed package of SuperMushroom from Item to Consumable

Sequence Diagram



Design Rationale

JumpAction:

this is an action that the player can use when next to a HighGround, it will allow the player to jump to said high ground (move to that ground) even if that ground type blocks actors, it is an action, so it inherits from Action. it will need to move the player, so it needs to work with the Location class, so it will depend on it. because of the method "public String execute(Actor actor, GameMap map)" that it inherits from Action and we need to override, it will depend on Actor and GameMap

HighGround:

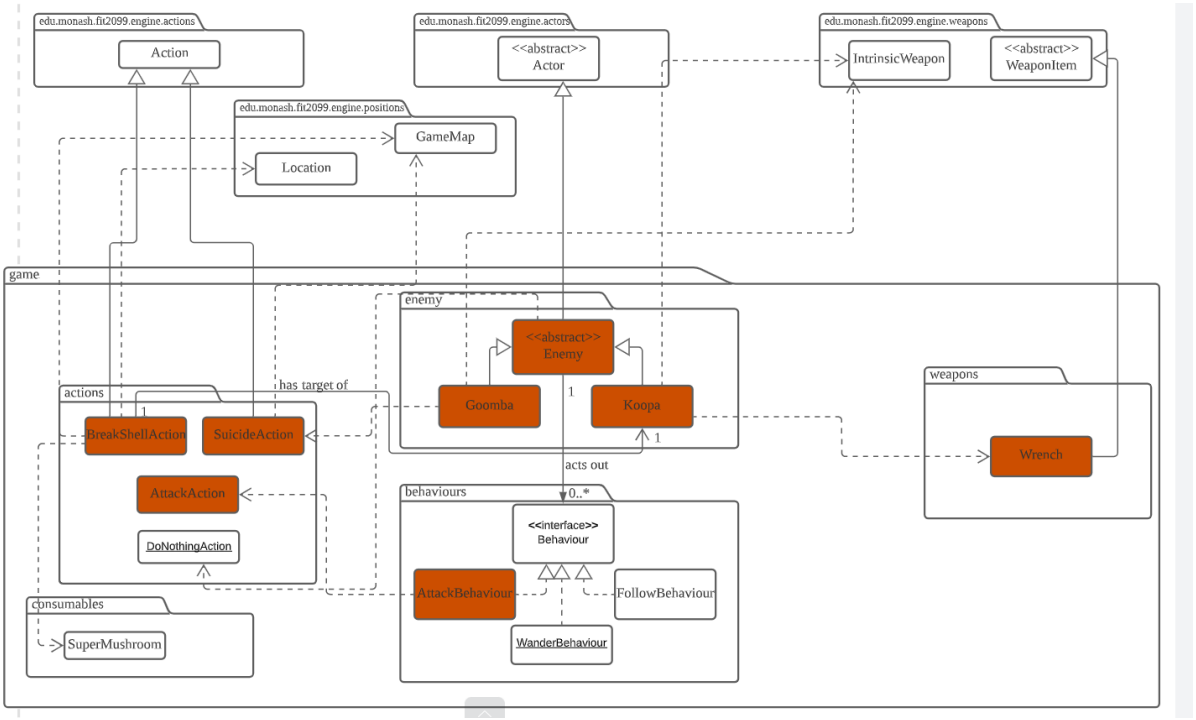
the HighGround Class is an abstract class that is meant to be used by Ground types that require the player to jump to them, it is meant meanly to make sure to follow the DRY principle. Because of the method "public ActionList allowableActions(Actor actor, Location location, String direction)" it will need to depend on Actor, Location, Action and ActionList. the method will return an ActionList containing the JumpAction, so it will need to depend on that as well

SuperMushroom:

it is an item that if picked up, will allow the player to have a 100% jump chance (it has other effects but they are not important for REQ2), we plan to make it give the player a status when consumed, so it won't depend or associate with any class except for an Enum class. it will only inherit from Item because it will be an item
that the player can pick up and store in its inventory

REQ 3

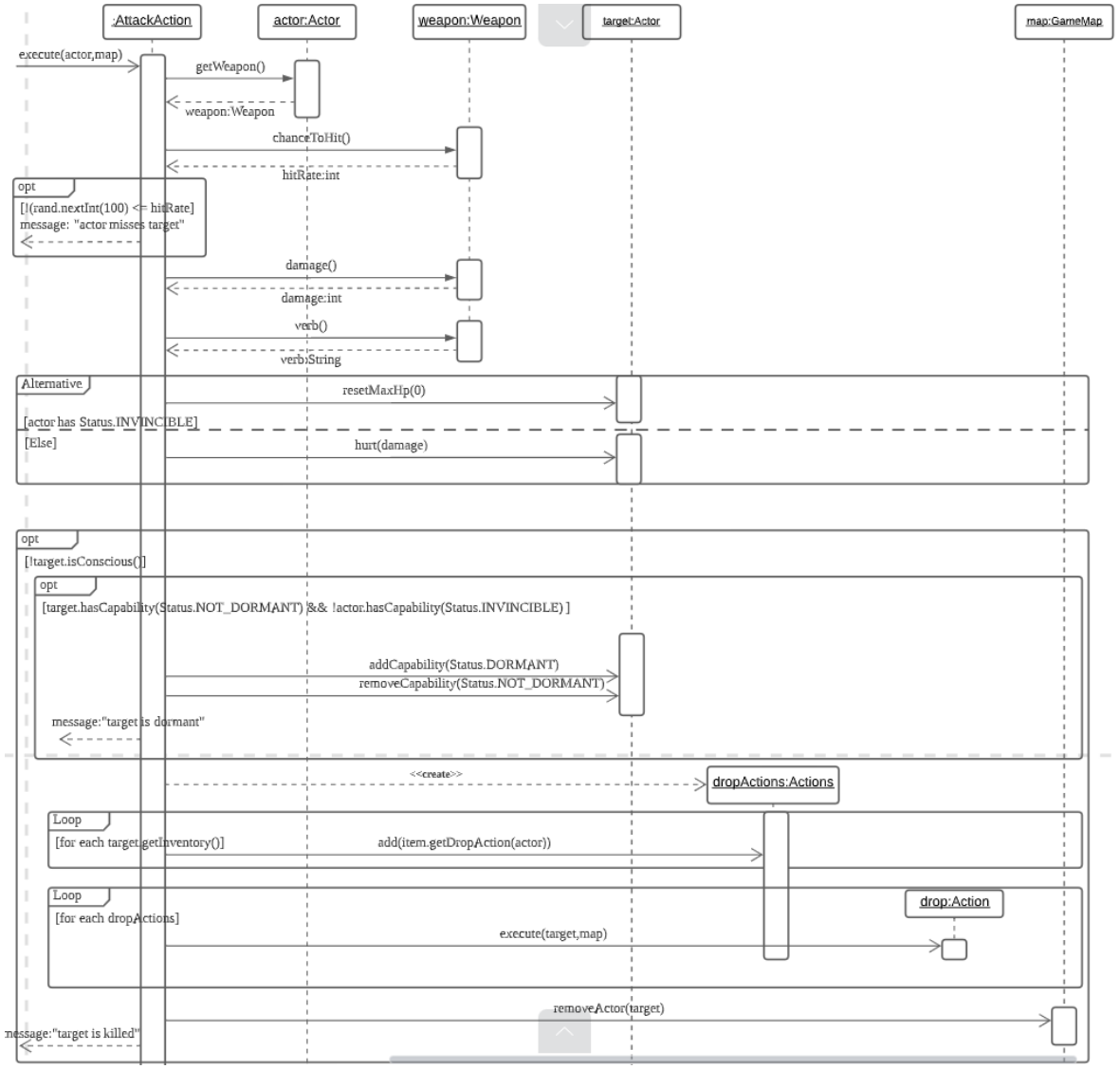
Class Diagram



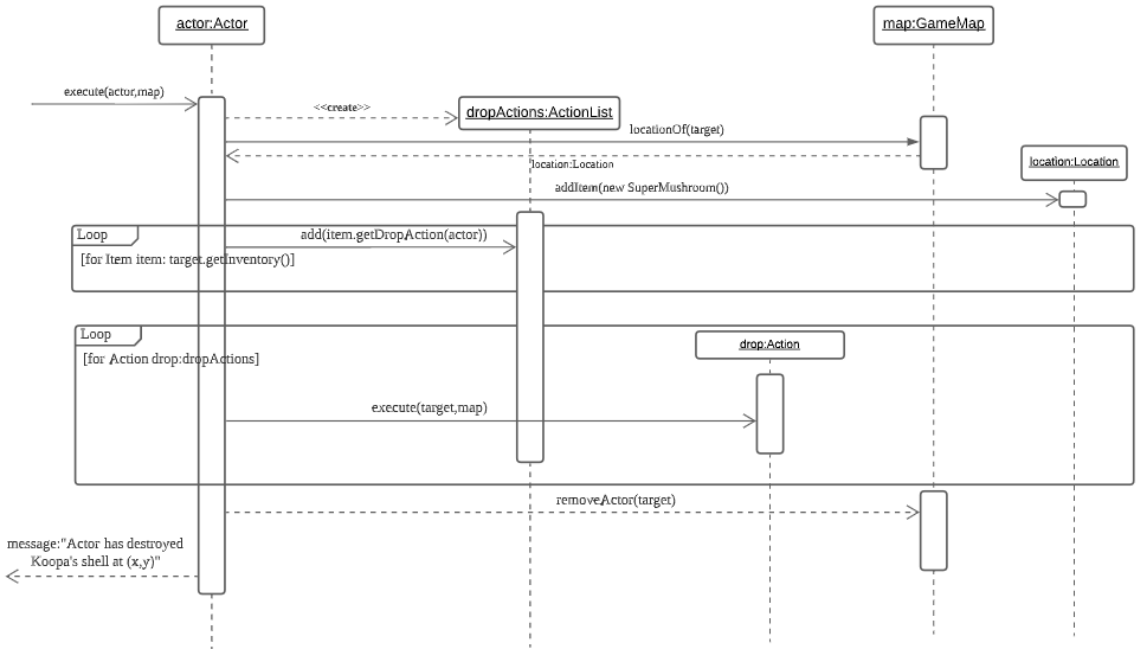
- Added BreakShellAction association with Koopa
- Added BreakShellAction dependency with SuperMushroom and Location
- Removed Koopa's dependency with SuperMushroom

Sequence Diagrams

- AttackAction.execute

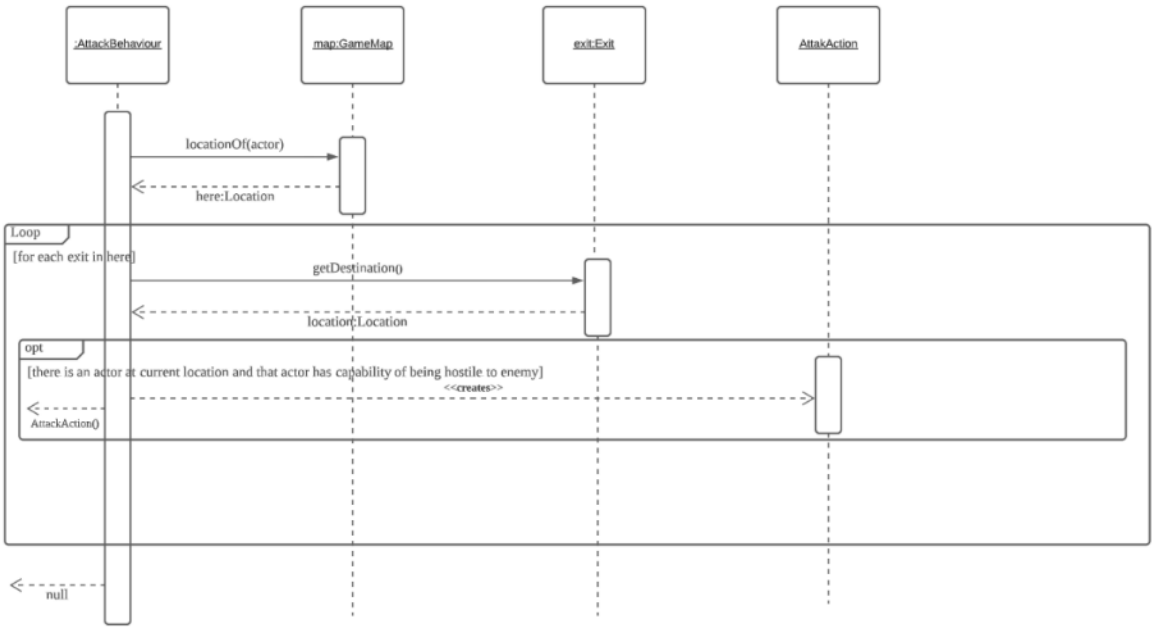


- BreakShellAction.execute

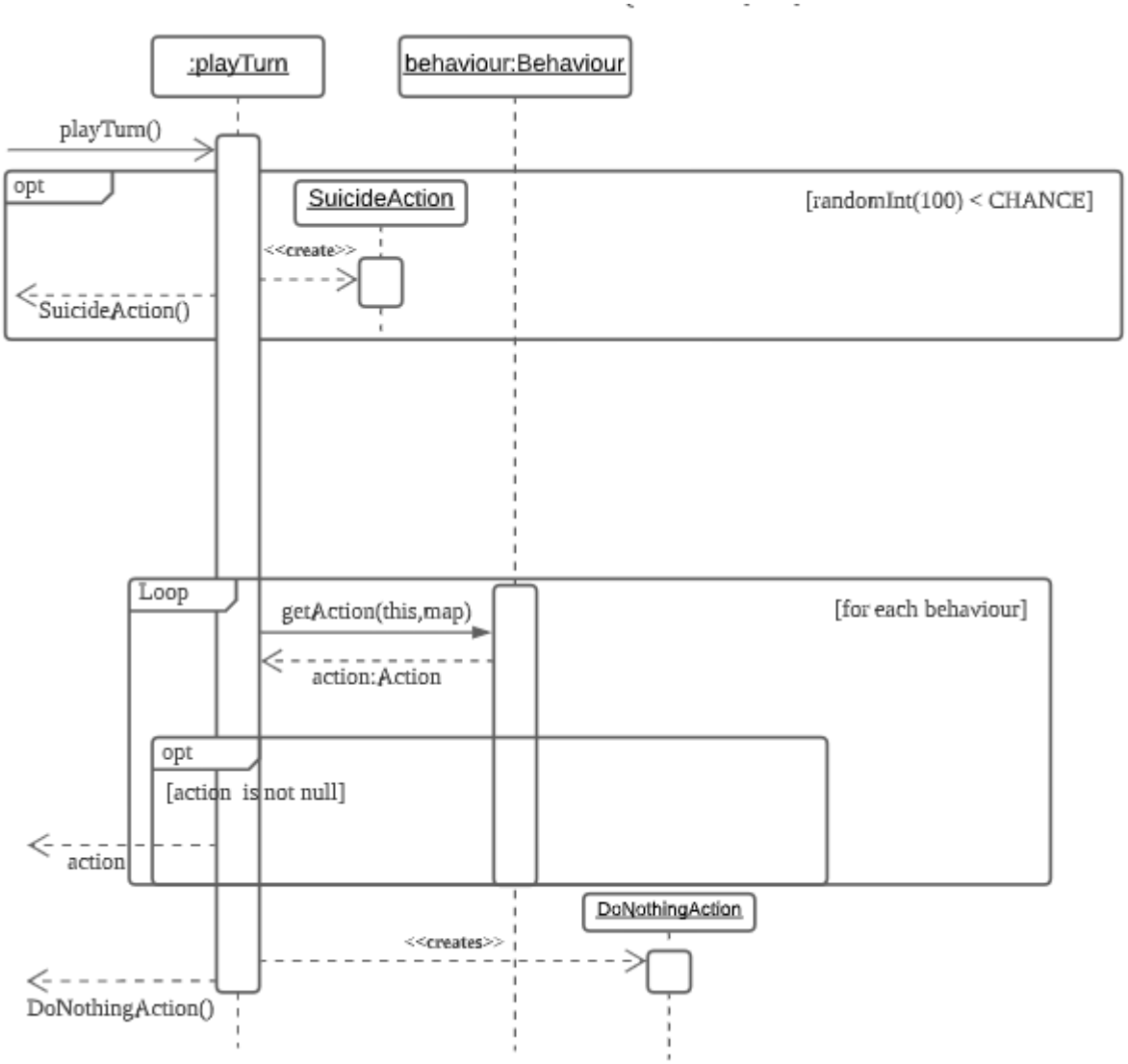


  - Modified to now spawn a SuperMushroom at where Koopa's shell will be broken at

- AttackBehaviour.getAction()



- Goomba.playTurn()



Design Rationale

Newly created/modified classes with new relationships

Enemy
We made an abstract Enemy class that extends Actor class. Then, Goomba and Koopa will extend Enemy class. At first, we thought about just having Goomba and Koopa extend the Actor class itself,since it seemed straight forward. However, after looking through the instructions, there are plenty of features that Goomba and Koopa share with one another, such as them being hostile to the player, or how both have similar behaviours which they can act on. So, that's why we decided to create an abstract parent class of both of them called Enemy, in order to not violate the Don't Repeat Yourself principle. The biggest advantage by doing so is that in the future, we will assume many more actors will be hostile towards the player, so we can make them extend the enemy class in order to prevent redundant codes. Responsible for handling all hostile enemies. Has association with Behaviour since every enemy will be able to act out behaviours.

Goomba
A class that extends Enemy class. Responsible for representing a Goomba enemy.

Koopa
A class that extends Enemy class. Responsible for representing a Koopa enemy.

AttackBehaviour
A class that implements the Behaviour interface. Responsible for knowing when it is appropriate to launch an attack to a hostile actor.

BreakShellAction
A class that extends Action class. Responsible for executing the action to break Koopa's shell.

SuicideAction
A class that extends Action class. Responsible for executing the action to cause Goomba to suicide.

AttackAction
Modified to adhere to the feature of Koopa going into dormant state and not dying, and from REQ4's feature that if player is invincible, can one-shot Koopa even without a wrench

Wrench
A class that extends Item class. Responsible for representing a Wrench weapon.

Floor
Modified to adhere to the features of enemies not being able to enter it

Basic features of enemy

"Once the enemy is engaged in a fight, it will follow the Player"

After looking through ED, Dr Riordan stated that as long as the enemy is in the vicinity(one of the 8 exits) of the player, it will start to follow the player. So, this can be done by adding a new FollowBehaviour to the enemy's allowableActions method, where if the otherActor has capability of being hostile to the enemy, then it will start following them.

"The unconscious enemy must be removed from the map."

This is already implemented in the original source code given in AttackAction.

"All enemies cannot enter the Floor."

For this, we can override the canActorEnter method in Floor class, to check if the actor is hostile to the player, false will be returned, true otherwise.

Features of Goomba

"Goomba starting with 20 HP"

In the constructor itself, we can initialise Goomba with 20 hitpoints.

"Goomba attacks with a kick that deals 10 damage with 50% hit rate."

This is easily achievable by overriding the getIntrinsicWeapon method and changing the damage value and verb when creating new IntrinsicWeapon. Hence, the dependency from Goomba to IntrinsicWeapon.

"In every turn, Goomba has a 10% chance to be removed from the map (suicide)."

This can be done in Goomba's playTurn method, where we could generate 100 numbers ranging from 0 to 100 exclusive, if the number generated is less than 10, a SuicideAction will be returned, else process Goomba's behaviour like normal. In SuicideAction's execute method, we can utilise GameMap's removeActor method. Originally, we thought of creating a SuicideBehaviour, but we thought it wouldn't make sense since behaviours are something that an actor, depending on the priority of behaviours and the current condition, return an action. But for suiciding, it is completely random, hence that's why we scrapped the idea of having a SuicideBehaviour. Hence, the dependency from Goomba to SuicideAction.

Features of Koopa

"Koopa starts with 100 HP"

In the constructor itself we can initialise Koopa with 100 hitpoints.

"Koopa attacks with a punch that deals 30 damage with a 50% hit rate."

This is easily achievable by overriding the getIntrinsicWeapon method and changing the damage value and verb when creating new IntrinsicWeapon. Hence, the dependency from Koopa to IntrinsicWeapon.

"When defeated, Koopa will not be removed from the map. Instead, it will go to a dormant state (D) and stay on the ground"

Koopa will be initialised with Status.NOT_DORMANT in the constructor. In AttackAction, when target is not conscious, we can check if the target has capability of not being dormant, and if the player does not have invincibility(from req4), Koopa target will be given the capability of being dormant, and the not dormant capability will be removed. All of Koopa's current behaviours will be removed as well since it can't do anything in its dormant state. For the display character, we can override getDisplayChar method, and check if Koopa is in dormant state or not, if yes then replace the char with 'D'.

"Mario needs a Wrench (80% hit rate and 50 damage), the only weapon to destroy Koopa's shell."

In Koopa's allowableActions, if the player is holding a wrench(checked using getWeapon method), hence the dependency from Koopa to Wrench, and Koopa has the capability of being dormant, then the player can do a new action to it called BreakShellAction. It will get every item in Koopa's inventory and drop it, and then picked up by the player, and lastly utilise GameMap to remove the dormant Koopa from the map. Originally, we thought of just having more if statements to check if player is using a wrench and if it's dormant then kill it, but we think it violates the OOP principle of Single Responsibility Principle, as the AttackAction will have too many responsibilities at this point as it will bear the responsibility of attacking actors as well as breaking Koopa's shell.

"Destroying its shell will drop a Super Mushroom."

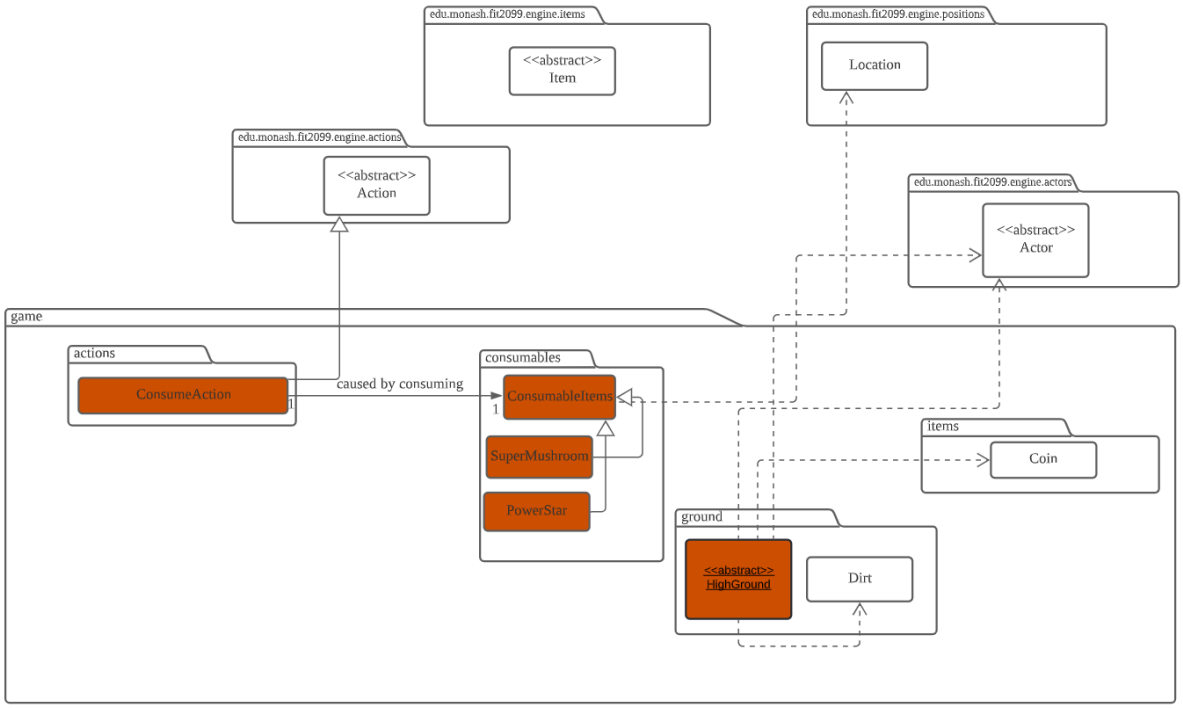
In BreakShellAction, a SuperMushroom item will be created and added to the location where Koopa's shell is broken.

AttackBehaviour's design

AttackBehaviour will be added to every enemy's behaviour list during construction, hence the dependency from Enemy to AttackBehaviour. Firstly, location of enemy will be obtained using GameMap's locationOf method. For that location, we will get all its exits using the getExits method. For each exit, we will get its location using the getDestination method. For each of those locations, we will check if there is an actor at a current location and whether that actor has the capability of being hostile to enemies, if so, then create an AttackAction in that direction.

REQ 4

Class Diagram

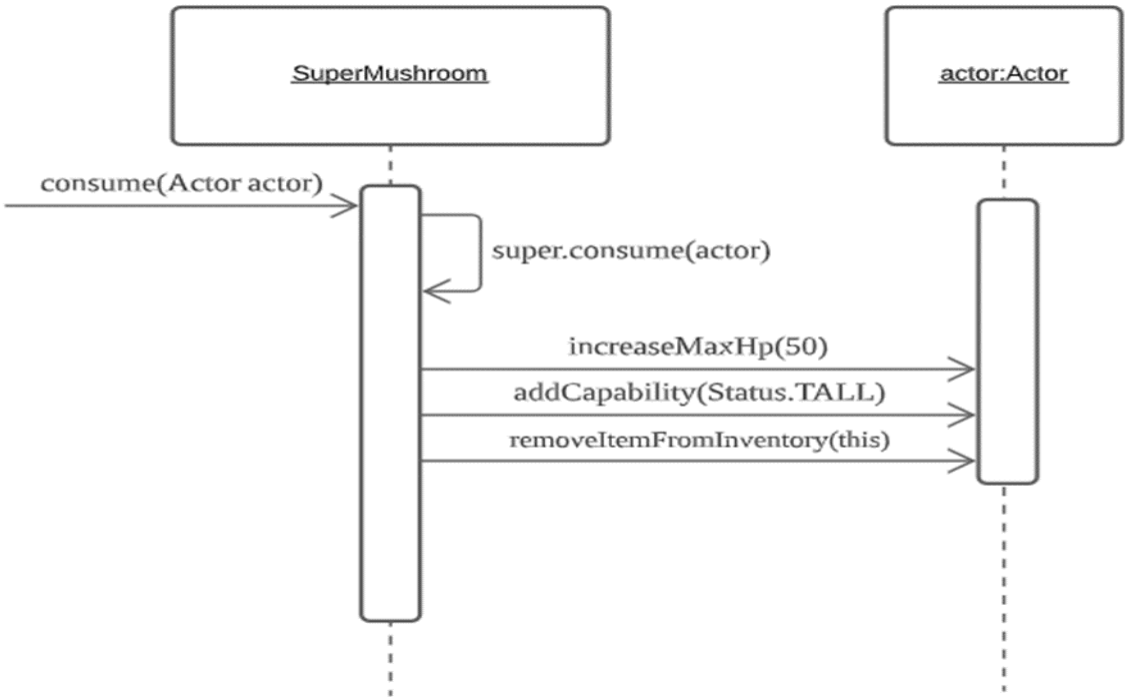


- SuperMushroom and PowerStar now extends an abstract class called ConsumableItems
- Instead of having a separate action for consuming each item, there is now only one action called ConsumeAction that is responsible for handling the consumable item's effects once consumed.
- ConsumeAction has an association with ConsumableItems

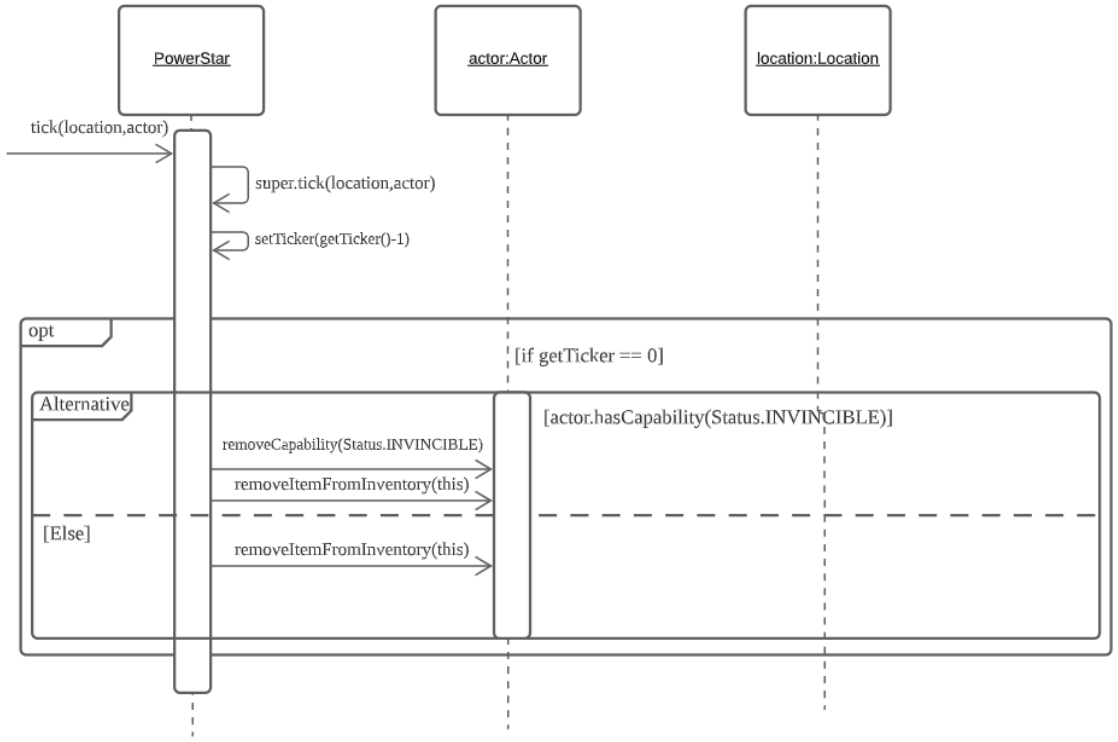
Sequence Diagrams

- SuperMushroom.consume()

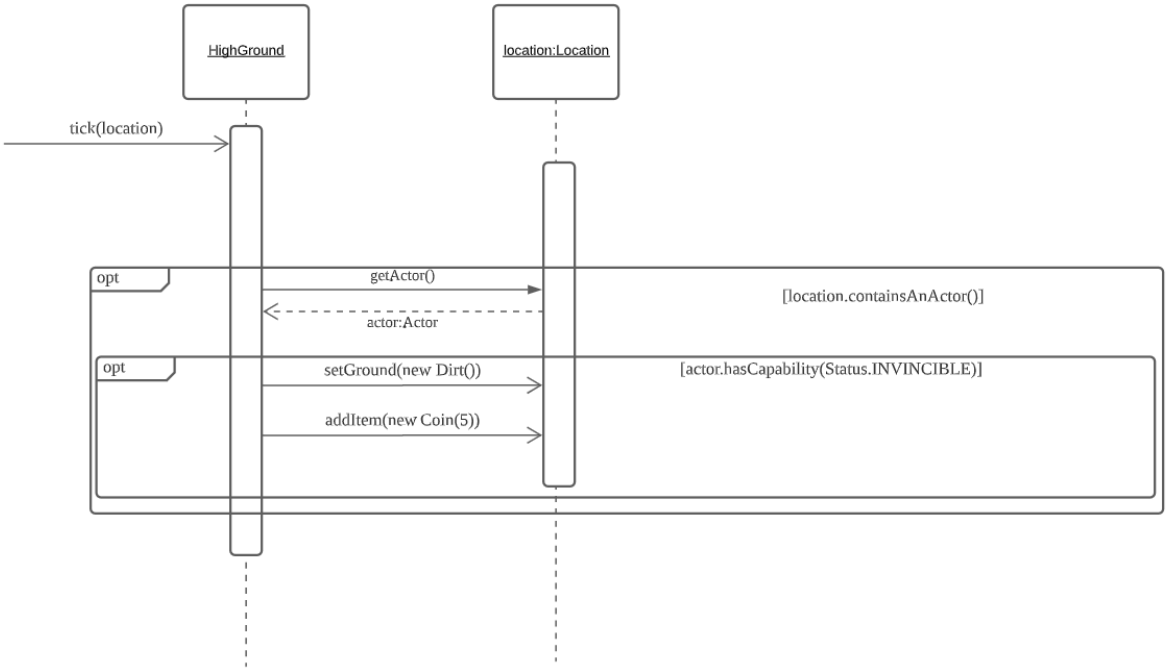
- PowerStar.consume()



- PowerStar.tick()



- HighGround.tick()



Design Rationale

Newly created/modified classes with new relationships

ConsumableItems(new)
An abstract class that extends Item class. Used to represent items that can be consumed.

SuperMushroom
A class that extends the ConsumableItems class. Responsible for representing a SuperMushroom object

PowerStar
A class that extends the ConsumableItems class. Responsible for representing a PowerStar object

ConsumeAction(new)
A class that extends Action class. Responsible for handling consumable item's effects when consumed. This class was created to avoid breaking the Don't Repeat Yourself principle, so there's no need for separate action for each consumable item.

HighGround
This is changed because of some changes that need to be done to its allowableActions method. And, it can be destroyed now after having the effects of consuming a power star. Was created to avoid breaking the Don't Repeat Yourself principle

Effects of consuming a super mushroom
"Increase max HP by 50"
We can utilise actor's increaseMaxHp method to heal players by 50 hitpoints in SuperMushroom.consume()

"The display character evolves to the uppercase letter"
This is already done for us in original source code for Player class, in getDisplayChar method

"It can jump freely with a 100% success rate and no fall damage"
Players will be given the status of TALL. Jumping freely with a 100% success rate can be done in HighGround's allowableActions method, where we can implement a check to see if an actor doing the jump has TALL status or not, if it is then JumpAction can be executed without even checking for percentages.

After all this is done, the SuperMushroom that was consumed will be removed from the player's inventory, in SuperMushroom's consume method using removeItemFromInventory method.

"The effect will last until it receives any damage"
We can override the hurt method in Player class, where if a player does have TALL status, then remove it.

Properties of Power Star

"It will fade away and be removed from the game within 10 turns (regardless it is on the ground or in the actor's inventory)"

We can override both tick methods from Item class, the first tick method is the one with only location as parameter. This means that it is on the ground, so in every tick the ticker will decrease by 1. If the counter reaches 0, we can utilise location's removeItem method to clear the PowerStar from the map. The second tick method is the one with location and actor as parameters, this means that it is in the player's inventory. Ticker will be decreased by 1. If ticker reaches 0, we check if actor has capability of invincible, if it does, then remove it and remove the item from actor's inventory. If not, just remove the item from the actor's inventory.

Effects of consuming a Power Star

"Anyone that consumes a Power Star will be healed by 200 hit points"

We can utilise actor's heal method to heal player by 200 hit points

" Will become invincible"

We can use addCapabilities to give the player Status.INVINCIBLE

“The actor does not need to jump to higher level ground (can walk normally)”

We can override the canActorEnter method in HighGround, checking that if an actor has the capability of invincible, it will just return true, while normally it returns false.

“If the actor steps on high ground, it will automatically destroy (convert) ground to Dirt. For every destroyed ground, it drops a Coin ($5).”

In HighGround’s tick method, we will first check if the location contains an actor using Location’s containsAnActor method. If so, we will get the actor in that current location, using the getActor method from Location, and check whether the actor has invincible capability or not. Therefore there’s a dependency between HighGround and Actor. If so, we can utilise Location’s setGround method to set current location to dirt and add a coin of value $5 using addItem method.

“All enemy attacks become useless (0 damage).”

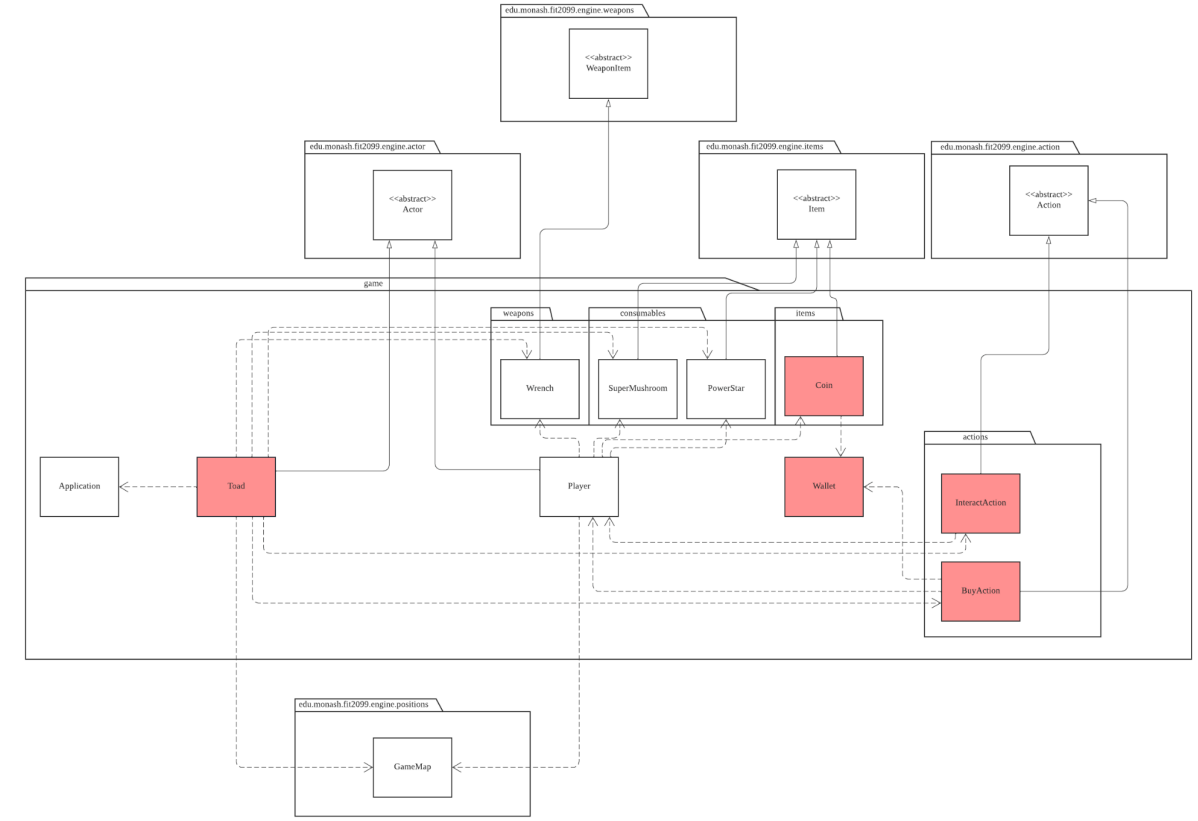
Override player’s hurt method, if player has invincibility status, hitPoints -= 0

“When active, a successful attack will instantly kill enemies.”

Referenced in REQ3’s AttackAction sequence diagram, we can check if an actor has the capability of invincibility. If so, reset the target’s max hp to 0 using resetMaxHp method.

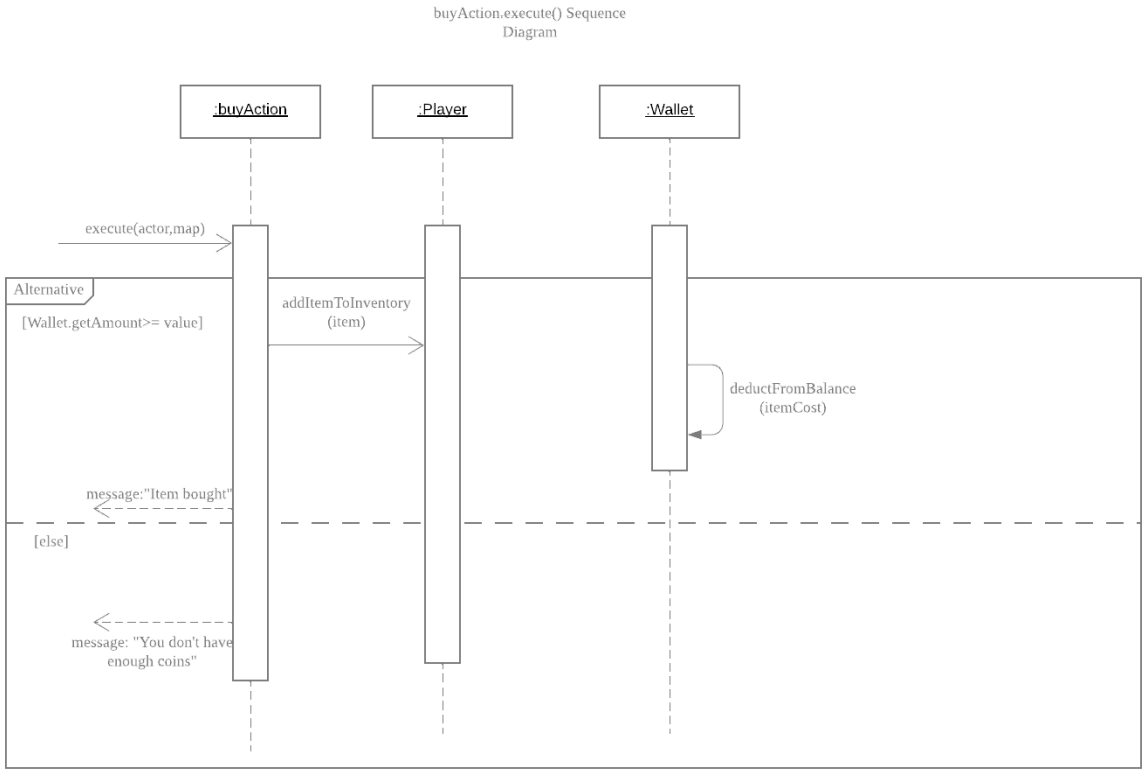
REQ 5&6

Class Diagrams



- Added dependency between InteractAction and Player, BuyAction and Player.
- Removed association between player and wallet because wallet is not instantiated and declared in Player class.
- Removed dependency between player and toad. They have dependency via BuyAction and InteractAction but no direct dependency.
- Edited packages for items to consumables, items and weapons.

Sequence Diagram



Design Rationale

Newly created/modified classes with new relationships

Coin(new)
A class that extends Item. Used to represent the coins on the ground. Has a constructor that will initialise the value of the coin.

Wallet(new)
A new class to keep track of the amount of money that the player has.

Toad(new)
This is a class for the Toad actor that extends the actor class.

BuyAction(new)
A class that extends Action class, in charge of selling items to the player

InteractAction(new)
Class that extends Action class, is in charge of having a conversation with the player. An action that will be called when player is at toad's surrounding

Wrench, Super Mushroom and Power Star
Added value to wrench, super mushroom and power star

Properties of Coin and effects on picking coin up

"The coin($) is the physical currency that an actor/buyer collects. A coin has an integer value (e.g., $5, $10, $20, or even $9001)."
Whenever a player picks up an item, it will automatically be in the player's inventory, but for coin, we do not want it to be in the player's inventory. We want it to add to the player's wallet when it is picked up. Our implementation is, we are going to override the item's tick method and remove the coin that is picked up from the inventory and add it into the player's wallet.

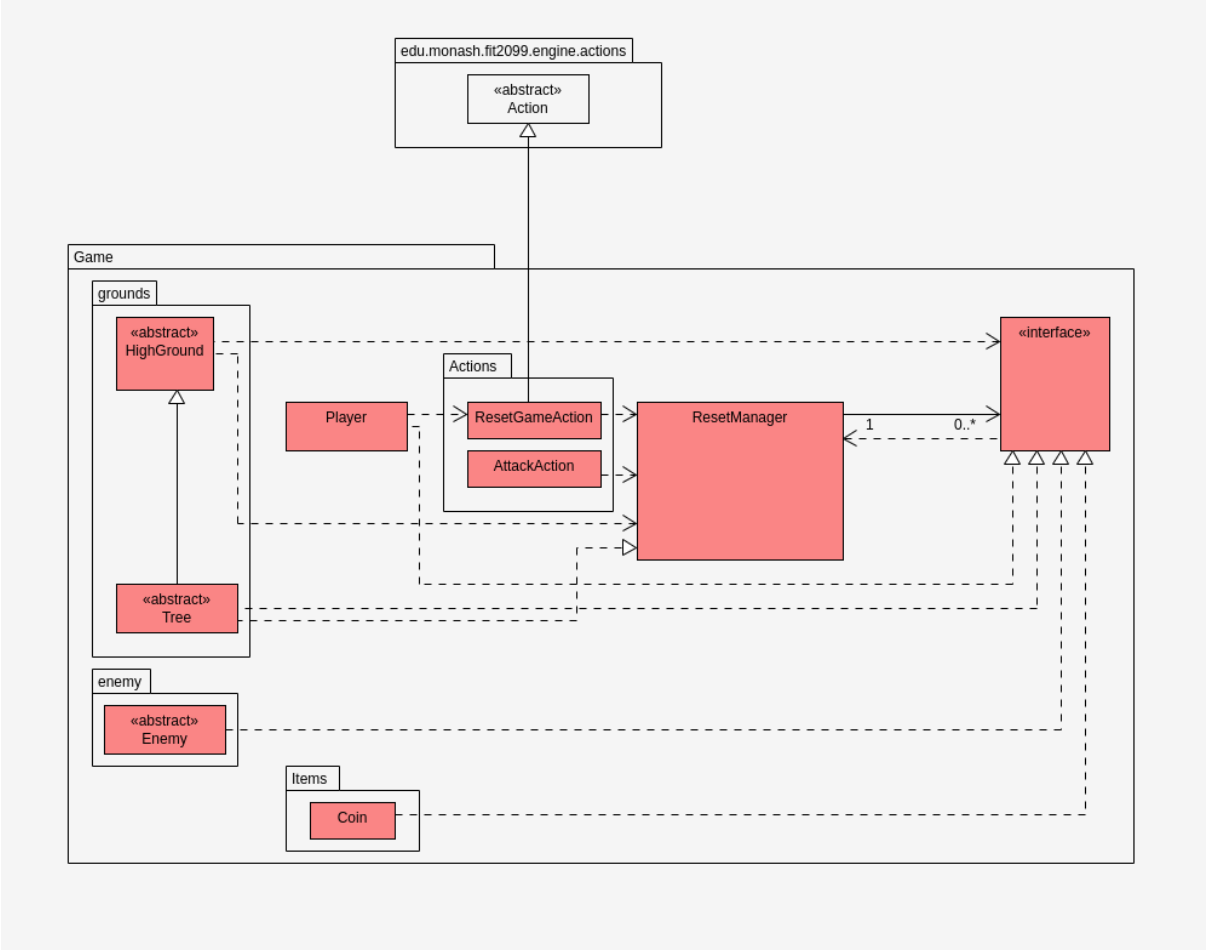
"Coins will spawn randomly from the Sapling (t) or from destroyed high grounds."
- Implemented in req2 and req5.

"Collecting these coins will increase the buyer's wallet (credit/money)."
When a player picks up a coin, it will automatically be added into the player's wallet. We are assuming that during the game, there will be only one player, so we used the singleton design pattern to create this wallet class. When a coin is picked up by the player, we remove the coin from the player's inventory, access the wallet and add coins to the wallet.

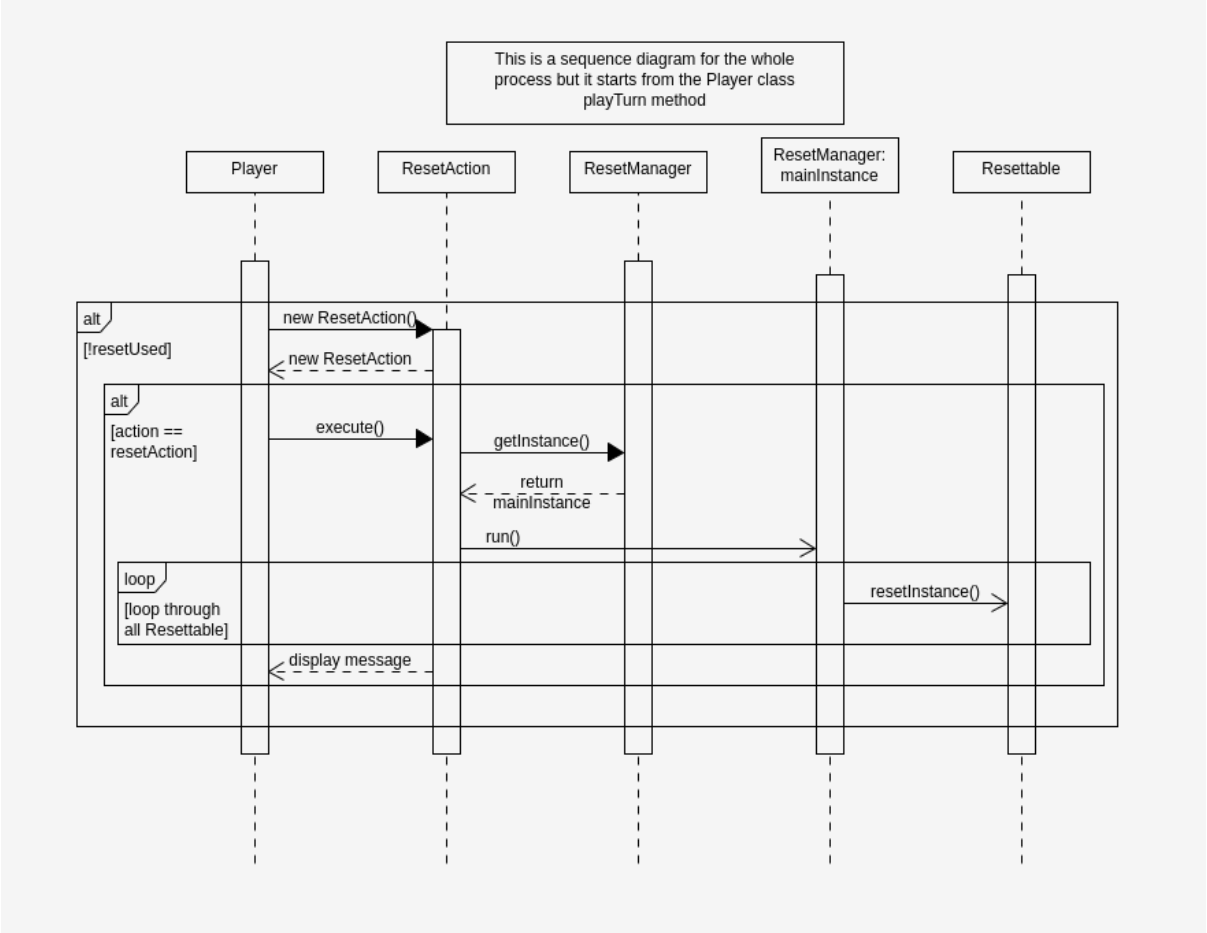
"Using this money, buyers (i.e., a player) can buy the following items from Toad:,Wrench: $200,Super Mushroom: $400,Power Star: $600"
. In this game, the ideal way to interact with the object is by attaching appropriate action to its corresponding object. Therefore, this action will have a dependency with Toad instead of Player. An action that will be called when the player is at Toad's surrounding.When a player wants to buy an item, Toad will check the player's wallet if it has sufficient balance to buy the item. If the player has sufficient balance, it will deduct the balance and add the item to the inventory. If it doesn't have sufficient balance, it will tell the player that it does not have enough money.
"Toad (O) is a friendly actor, and he stands in the middle of the map (surrounded by brick Walls)."
We add Toad manually in the application class as the specifications stated that we could add it manually.
"You will have an action to speak with Toad. Toad will speak one sentence at a time randomly:"
Toad has a dependency with InteractAction and BuyAction because Toad is the object that gives the other actor(Player) an action to buy and interact. Alternatively, we can create 3 separate BuyActions for Wrench, Power Star and Super Mushroom. However, doing so will require us to implement multiple methods that are the same inside different classes. This would violate the Don't Repeat Yourself(DRY) principle in object-oriented programming. In this game, the ideal way to interact with the object is by attaching appropriate action to its corresponding object. Therefore, this action will have a dependency with Toad instead of Player. Toad will check the player's inventory if it contains a wrench. If it contains a wrench, it won't say the first phrase. If player has the capability of invincibility, using the hasCapability method to check, it won't say the second phrase. If it contains both, it will only say the third and fourth phrase.

REQ 7

Class Diagram



Sequence Diagram



Design Rationale

Resettable:

an interface for all resettable objects, if an object would react in someway to the game reset, then it needs to implement this class

ResetManager:

a singleton that manages a list of all Resettable objects in the game and can remove objects from the list if they die,vanish or get used up before the game resets

ResetGameAction:

This is an action that the player can use to reset the game, it inherits from Action since it is an action the player can perform. It depends on ResetManager since it needs to call the run() method in order to reset the game

AttackAction:

Now when an enemy becomes unconscious, they are removed from the reset manager.

Enemy:
when the game resets all enemies must die, so when the game resets, each instance will kill itself

HighGround:
when the player stands on a HighGround after eating a powerstar, it turns to dirt, so if that current instance of a HighGround happens to be Resettable, it needs to be cleaned up

Tree:
a tree has a 50% chance to turn to dirt when resting, so it needs to be Resettable

why did you implement the die() method? to follow the DRY principle

Coin:
all coins disappear when the game resets, so it needs to be Resettable

WBA

Task: REQ1 implement, design and test the Tree class and its children
Teammate responsible: Fadi Alailan

Task: REQ2 implement, design and test Jumping
Teammate responsible: Fadi Alailan

Task: REQ3 Designing the design for Goomba and Koopa, as well as implementing it and testing it
Teammate responsible: Khor Jia Shin

Task: REQ4 Designing the design for SuperMushroom and PowerStar, as well as implementing it and testing it
Teammate responsible: Khor Jia Shin

Task: REQ5 and 6 Added Toad, Wallet and added InteractAction and BuyAction, as well as testing them.
Teammate responsible: Lim Hong Yee

Task: REQ7 implement, design, document and test the game reset action
Teammate responsible: Fadi Alailan

I accept this WBA(Lim Hong Yee)
I accept this WBA(Fadi Alailan)
I accept this WBA(Khor Jia Shin)