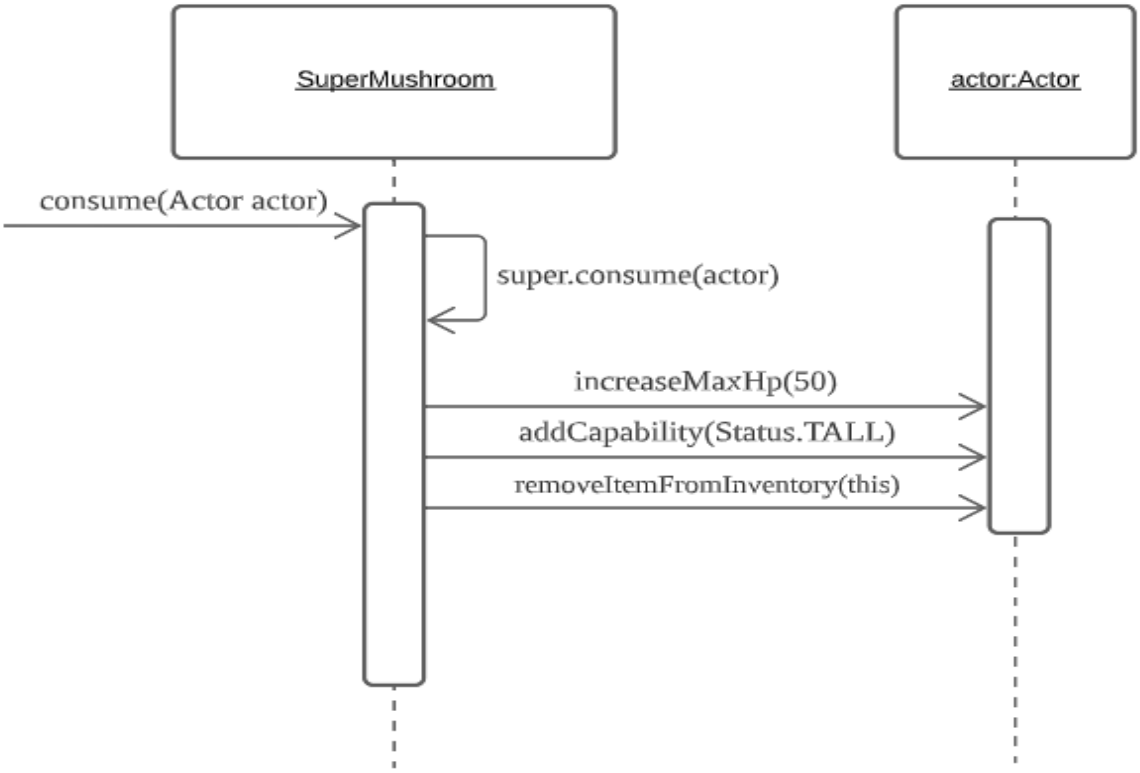
FIT 2099 Assignment 3 Design Rationale
Lab 7 Team 1
Structured Mode

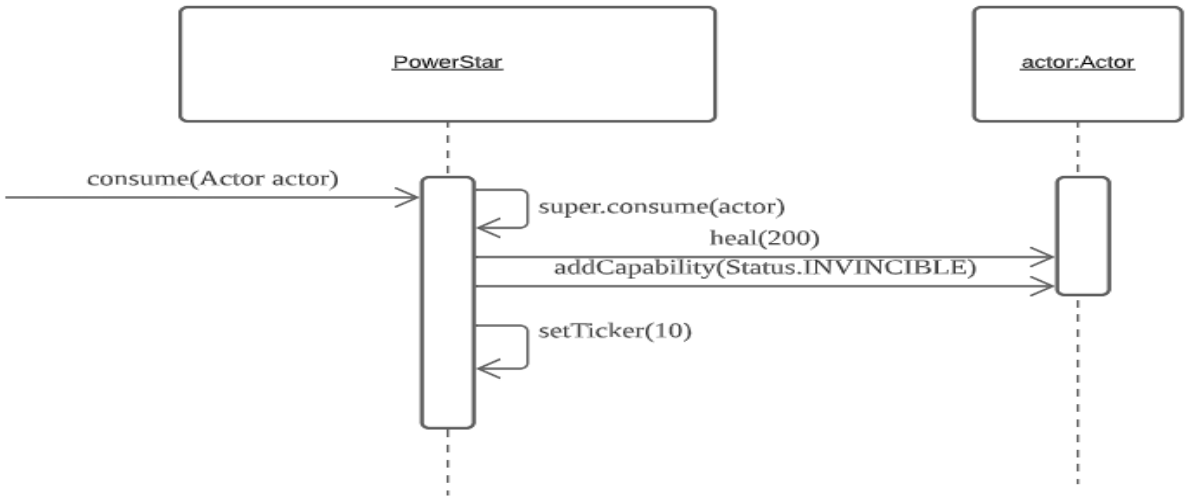
Lab Tutor: Dr. Tan Choon Ling
Group Members:
Khor Jia Shin 32356595
Lim Hong Yee 32455836
Fadi Alailan 31844936

Error to be corrected from Assignment 2

-SuperMushroom.consume()

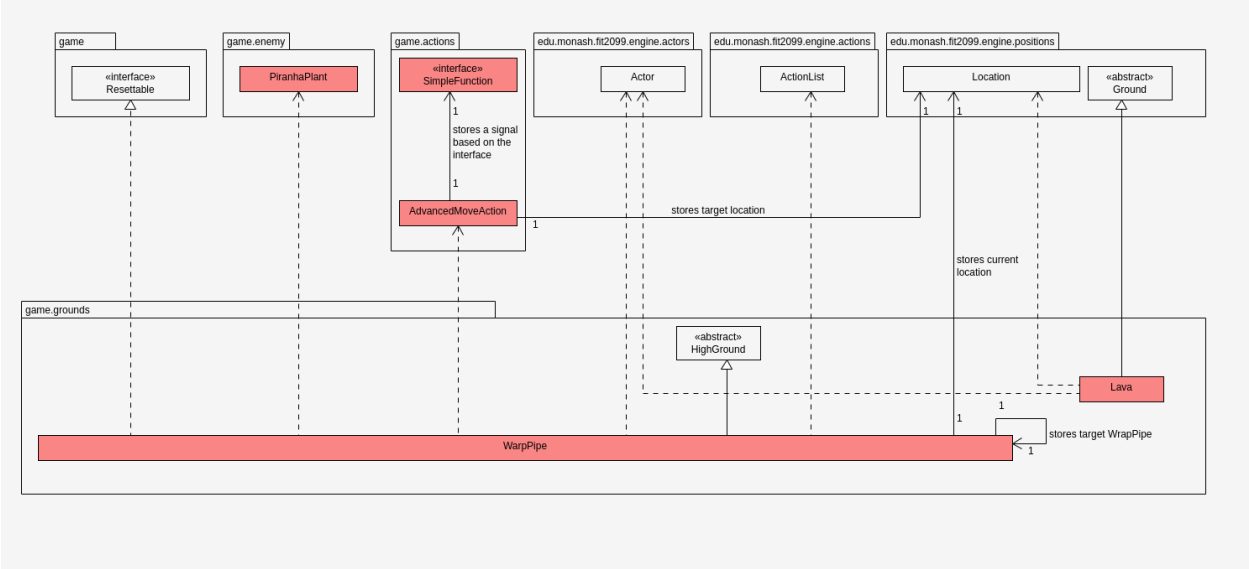


-PowerStar.consume()

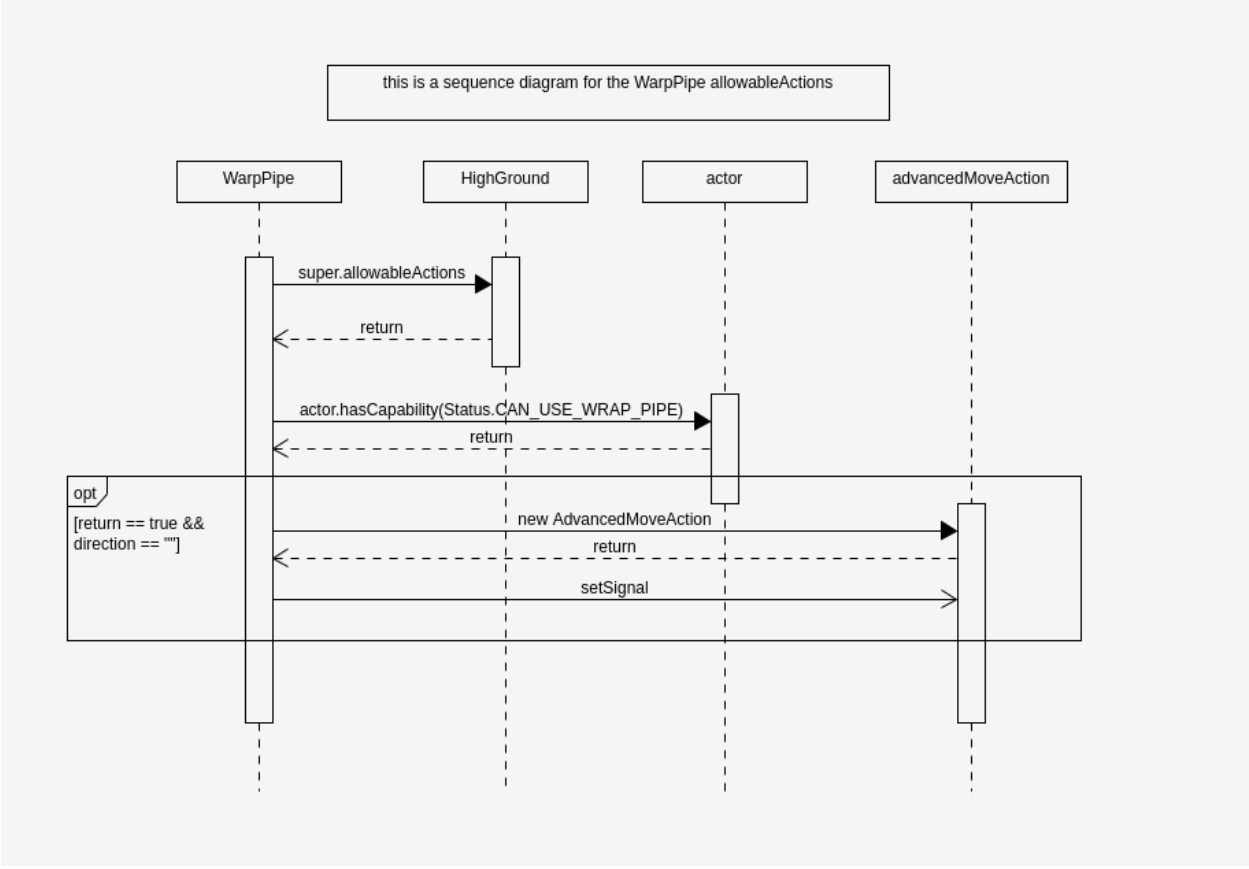


REQ1:

Class Diagram



Sequence diagram



Design Rationale:

WarpPipe:

the main class of the requirement it relies on the following classes for the following reasons:

HighGround (inherits): the assignment requires that the player has to jump to the WarpPipe, so it inherits from HighGround

WirePipe (association): it has an association with itself because it needs to store a reference to the WirePipe it will warp to

Location (dependency): the class needs to warp to another WarpPipe outside the tick method, so we can't access the location the tick method gives us, so we need to store it and update it every tick instead.

ActionList (dependency): this because of the allowableActions function which we need to give the player the advancedMoveAction

Actor (dependency): this is because of the allowableActions function which we need to give the player the advancedMoveAction

AdvancedMoveAction (dependency): this action is needed to warp the player to the other WarpPipe while also establishing a link with that WarpPipe

PiranhaPlant (dependency): not used in REQ1, will be explained in REQ2

Resettable (implements): not used in REQ1, will be explained in REQ2

AdvancedMoveAction:

this is an action that is an upgrade of the game engine's MoveActorAction , where now it has the ability to send a signal when executed (idea inspired from the godot game engine, a real game engine, though it can only send one signal, unlike the godot game engine, where it can send as many as it likes).

the main reason that this class was added was because of the open-closed principle (the O in SOLID), without this class every time we need to move an actor but also add a basic message or functionality, we will need to create a new class, and if we need to change this message or functionality then we need to modify said class, and none of these classes will support extensions, but for this class we can give it the message or functionality in the form of a signal, and if we want to change this signal, we don't need to touch a line of code in the class itself

it has an association with the following classes/interfaces:

SimpleFunction: the signal will be based on the SimpleFunction interface

Location: the class needs to store where the actor will be moved to

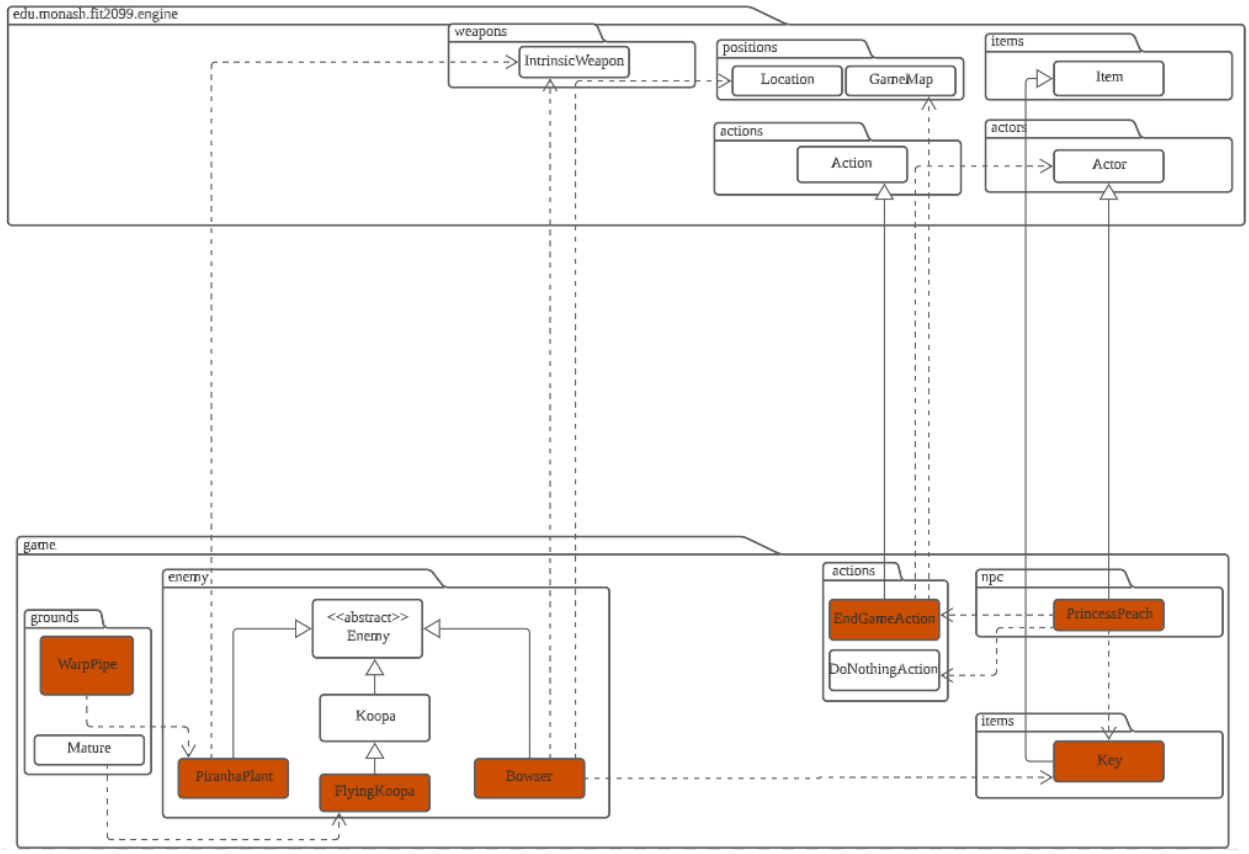
SimpleFunction:

this is an interface that is used to define a basic function with no arguments and no return

no dependencies or associations

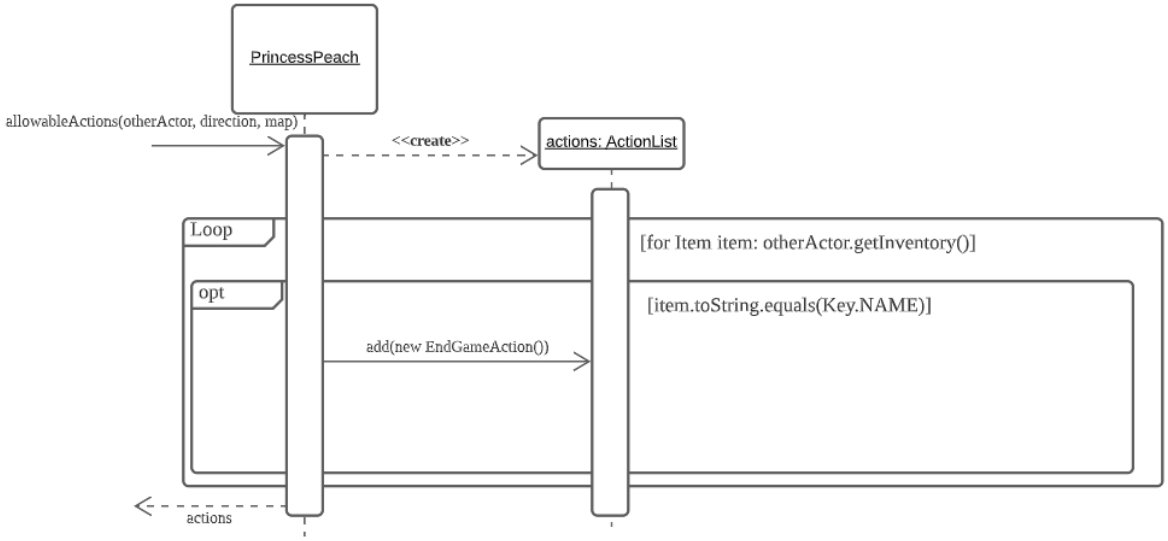
REQ2

Class Diagram

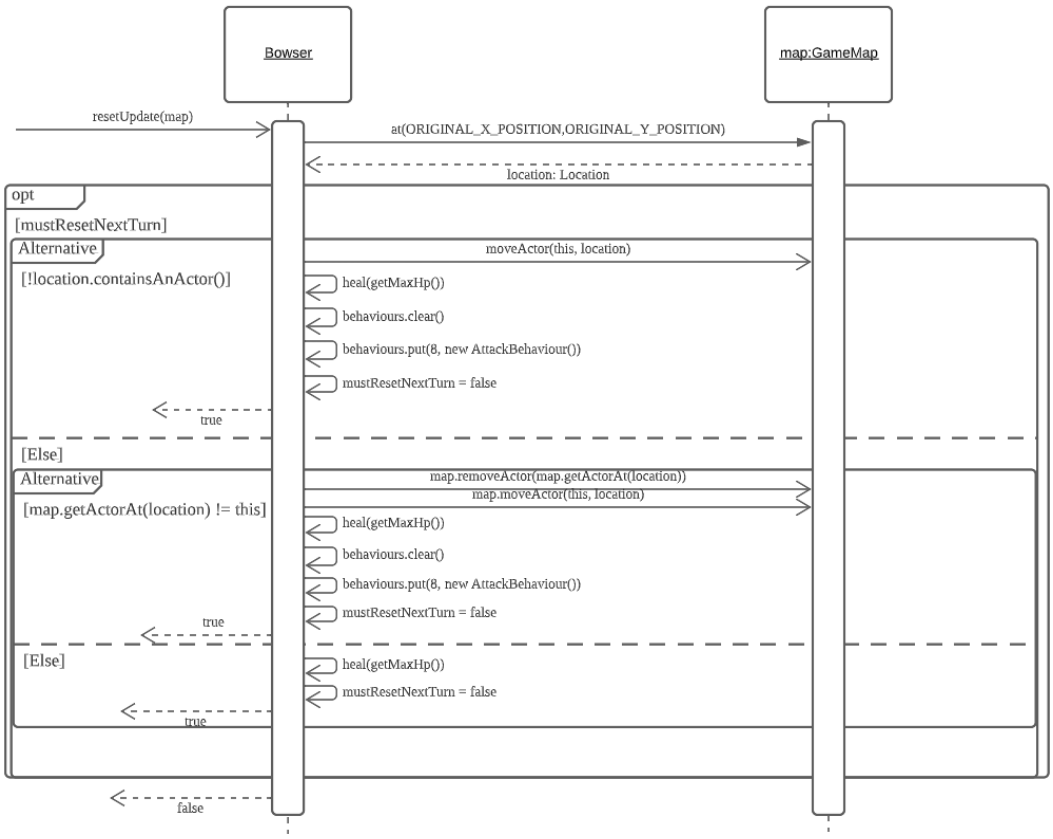


Sequence Diagram

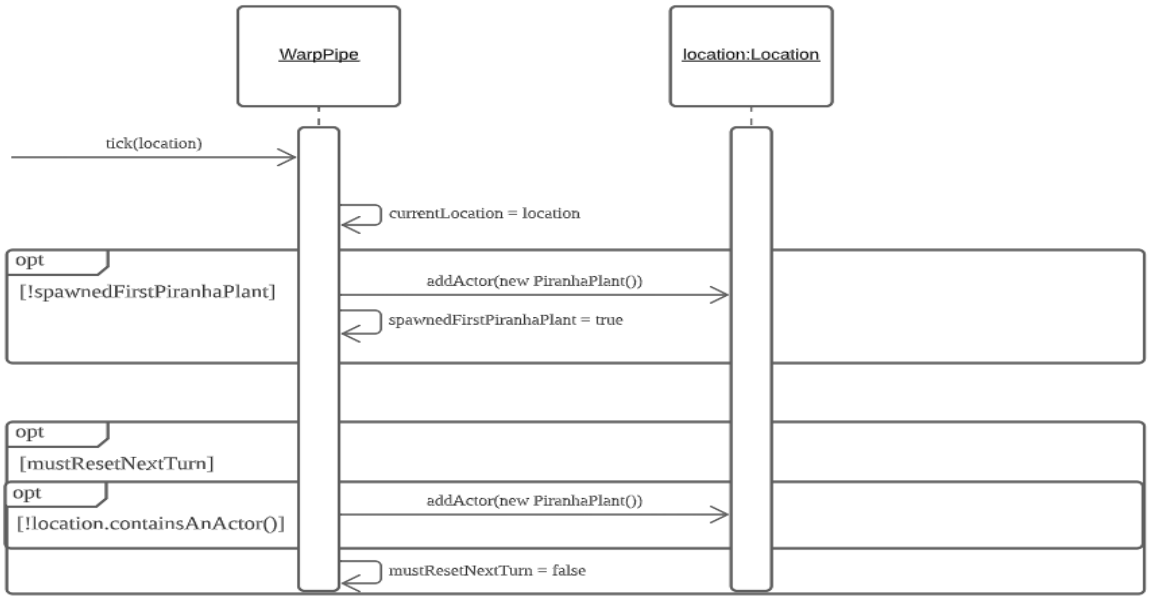
- PrincessPeach.allowableActions()



- Bowser.resetUpdate()



- WarpPipe.tick()



Design Rationale

Newly created/modified classes with new relationships

PrincessPeach

Responsible for representing PrincessPeach.

Key

Responsible for representing a Key item, in order to win the game.

EndGameAction

Responsible for ending the game.

Bowser

Responsible for representing the Bowser enemy. Extends enemy class to not violate DRY.

PiranhaPlant

Responsible for representing the PiranhaPlant enemy. Extends enemy class to not violate DRY.

WarpPipe

Modified to ensure PiranhaPlant spawns on the second tick of the game, and when resetted spawn another PiranhaPlant if there is no currently no actor on it.

Mature

Modified to have a chance to spawn FlyingKoopa

FlyingKoopa

Responsible for representing the FlyingKoopa enemy. Extends Koopa class to not violate DRY. Further explanation regarding Liskov Substitution Principle will be down below.

HighGround

Slightly modified to check if the actor has capability to fly

Features of PrincessPeach

"Once you have defeated Bowser and obtained a key, you can interact with her to end the game with a victory message!"

In PrincessPeach's allowableActions method, we can loop through otherActor(player)'s inventory and check whether he has a key or not, if so then we can use EndGameAction to end the game. In EndGameAction, it terminates the game by removing our main player from the GameMap.

Features of Bowser

"Whenever Bowser attacks, a fire will be dropped on the ground that lasts for three turns."
Bowser will be given the capability of fire breathing when instantiated. As to how the fire will be dropped, it will be explained in REQ4's design rationale.

"When the Bowser is killed, it will drop a key to unlock Princess Peach's handcuffs."
Bowser will be instantiated with a Key in his inventory.

"Bowser punches with 50% hit rate and 80 damage"
We can override the getIntrinsicWeapon method and change the damage value and verb when creating a new IntrinsicWeapon. Hence, the dependency from Bowser to IntrinsicWeapon.

"Resetting the game (`r` command) will move Bowser back to the original position, heal it to maximum, and it will stand there until Mario is within Bowser's attack range."
In Bowser's resetUpdate method, we can check if Bowser needs to be reset. If so, check if Bowser's original spawn point has any actors, if not just move Bowser back to his original spawn point and heal him to maximum. If so, remove the actor currently at Bowser's original spawn point and move Bowser there, and heal him to maximum.

Features of PiranhaPlant

"Piranha Plant will spawn at the second turn of the game"
In WarpPipe's tick method, we can check if we have already spawned a PiranhaPlant, if not then spawn it.

"PiranhaPlant chomps with 50% hit rate and 90 damage"
We can override the getIntrinsicWeapon method and change the damage value and verb when creating a new IntrinsicWeapon. Hence, the dependency from PiranhaPlant to IntrinsicWeapon

"Once the player kills it, the corresponding `WarpPipe` will not spawn Piranha Plant again until the player resets the game (`r` command)."
In WarpPipe's tick method, we can check if WarpPipe needs to be reset. If so, if there is no actor on the WarpPipe, spawn a new PiranhaPlant.

"Resetting will increase alive/existing Piranha Plants hit points by an additional 50 hit points and heal to the maximum."
In PiranhaPlant's resetUpdate method, we can check if PiranhaPlant needs to be reset. If so, use increaseMaxHp to increase max HP and heal to maximum.

Features of FlyingKoopa

"Furthermore, it can walk (fly) over the trees and walls when it wanders around (incl. other high grounds)."

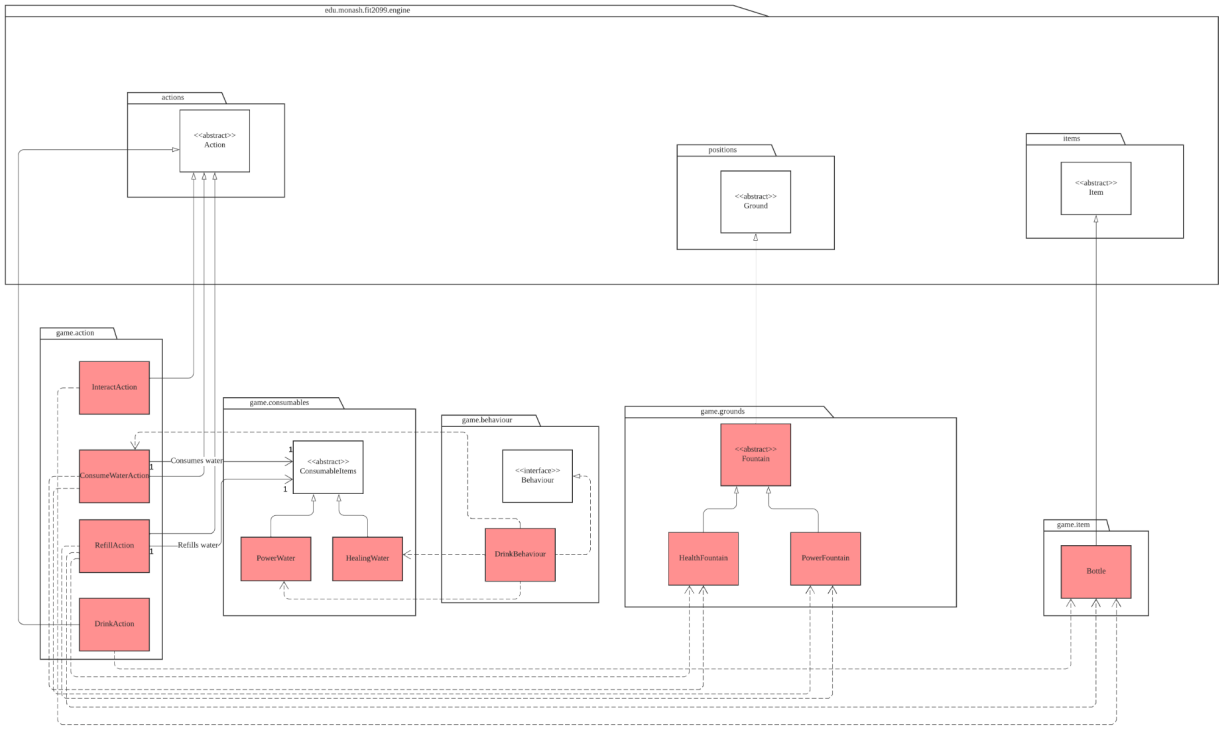
Every FlyingKoopa will be given the capability of being able to fly when instantiated. In HighGround's tick method, if the actor on it has capability to fly, then return true.

Explanation on FlyingKoopa extending Koopa

The only different feature a FlyingKoopa has from Koopa is that it can fly. So, every other feature of FlyingKoopa is the same as Koopa. Hence, FlyingKoopa extends Koopa class as mentioned earlier to adhere to DRY. The reason why this still adheres to Liskov Substitution Principle is because FlyingKoopa does not overwrite any of Koopa's methods, hence it will not result in any unexpected behavior. We can replace FlyingKoopa with Koopa and it still will not result in any unexpected behavior. The only method that will return a different output depending on if it's FlyingKoopa or Koopa is HighGround's tick method. But, replacing FlyingKoopa with Koopa will just return false instead of true since Koopa can't fly, hence not an unexpected behavior. Of course, down the line if there are more different features of FlyingKoopa compared to Koopa, Liskov Substitution Principle might be broken, but given the assignment instructions so far, this is not the case.

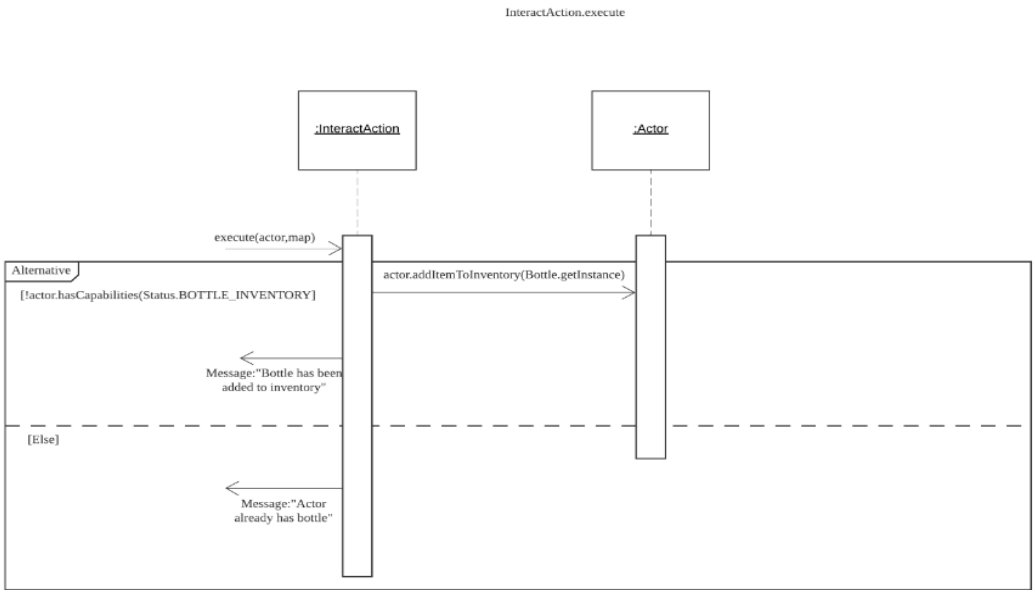
REQ 3

Class Diagram

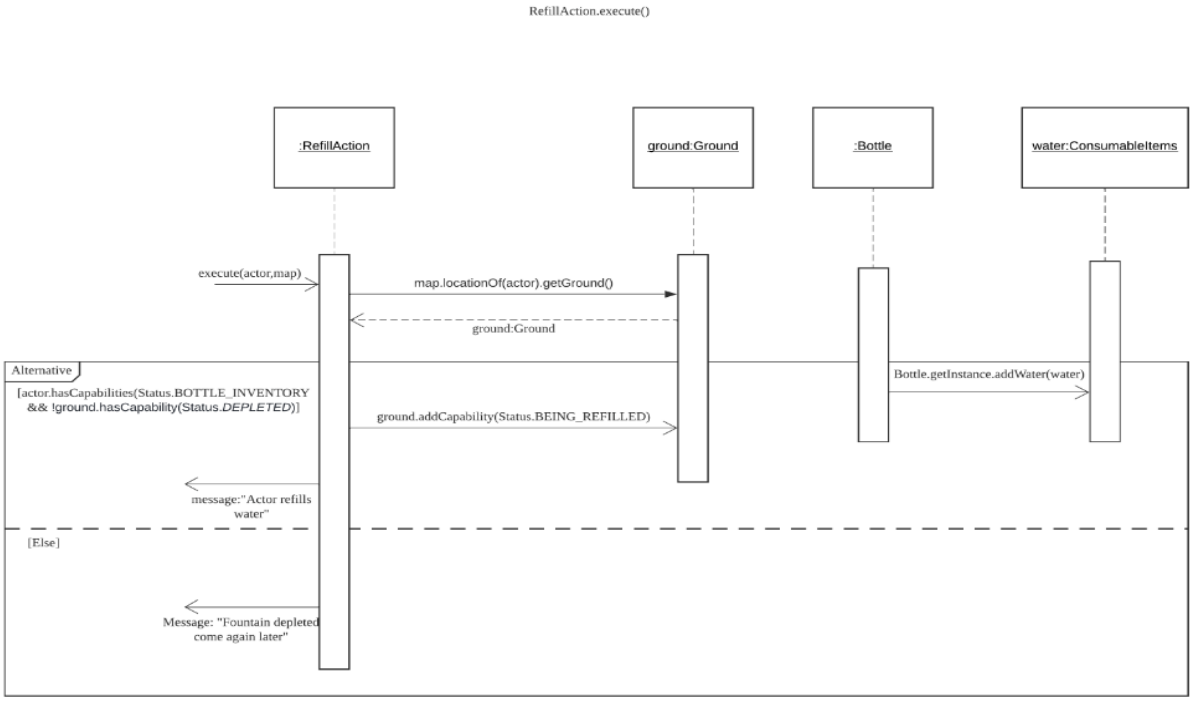


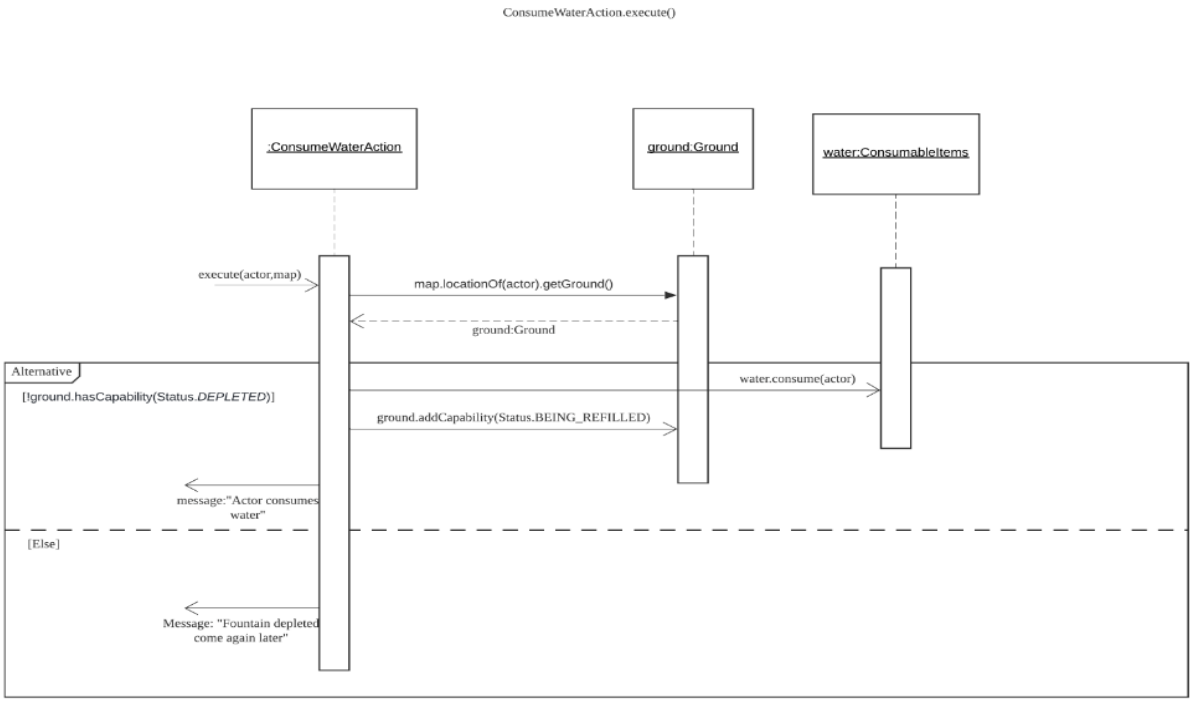
Sequence Diagram

InteractAction.execute



RefillAction.execute() & ConsumeWaterAction.execute()





Design Rationale

Newly created/modified classes with new relationships

InteractAction

An action for Mario to get bottle from Toad. New InteractAction that gives mario the bottle instance.

ConsumeWaterAction

An action for actor to consume water from fountain. Called in DrinkBehaviour so that actors can drink directly from fountain.

RefillAction

An action for Mario to refill bottle with PowerWater or HealthWater.

DrinkAction

An action for Mario to drink water from bottle.

Fountain

A new abstract class that extends ground class. To achieve the DRY principle, this new abstract class is created. If this class is not created, we would have to repeat the same code for both HealthFountain and PowerFountain with very little difference between them.

HealthFountain

A new class that extends Fountain class so that it does not violate DRY principle.

PowerFountain

A new class that extends Fountain class so that it does not violate DRY principle.

Bottle

A new class that extends Item class that could store water in it. I chose to use a singleton class for bottle because there is only one bottle in the game. This might be bad in the future if we want to implement more bottles, I would suggest to get the bottle from actor's inventory.

HealthWater

A new class that extends ConsumableItems to heal actor for 50 HP. I extended ConsumableItems for both Health and Power Water because once it is consumed, the water is gone from the inventory. This shows that it has similar methods and properties as Consumable Items. My implementation follows the open-closed principle because we do not have to modify our existing code from assignment 2 but instead we extended from Consumable Items.

PowerWater

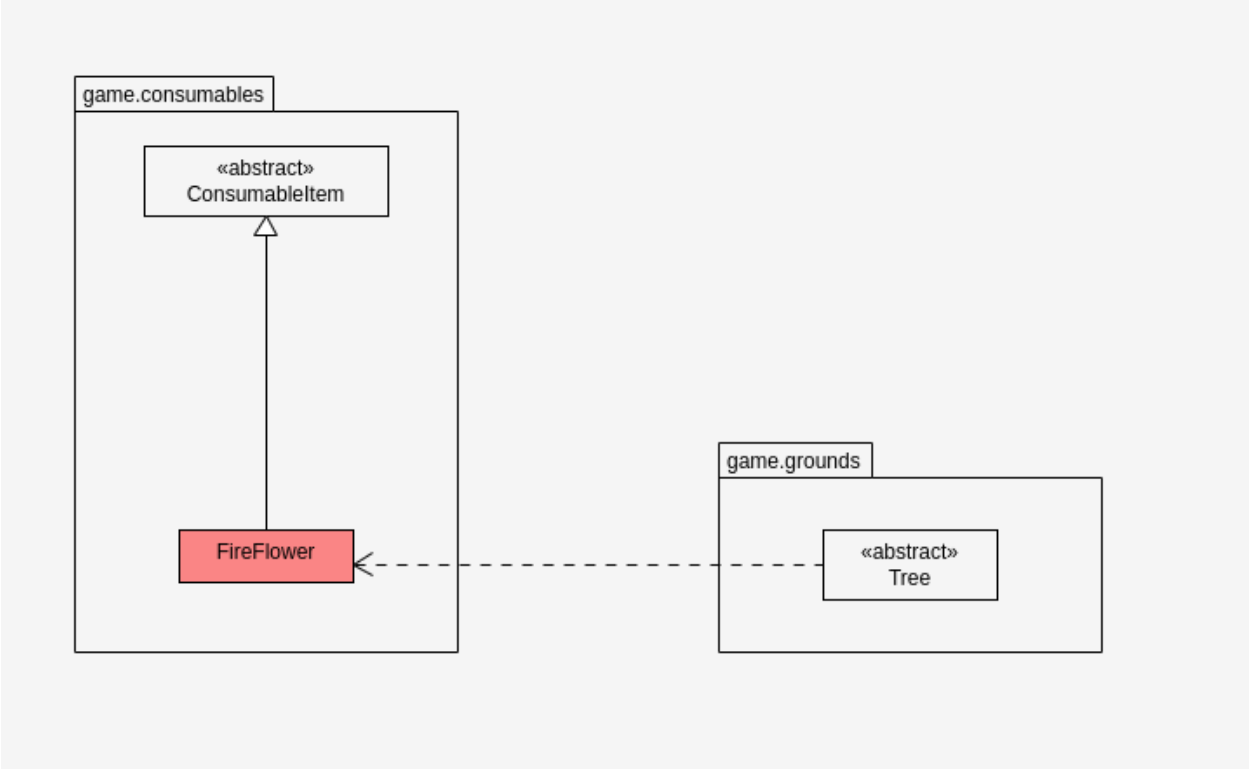
A new class that extends ConsumableItems to increase 15 base damage of an actor.

DrinkBehaviour

A new class that implements Behaviour interface so that enemies could drink from the fountain.

REQ4 PART 1 (Done by Fadi)

Class Diagram



Sequence Diagram:

there is no sequence diagram for this part because the communication between the classes is simple

Design Rational:

FireFlower:

it inherits from ConsumableItems, since it can be consumed by the player

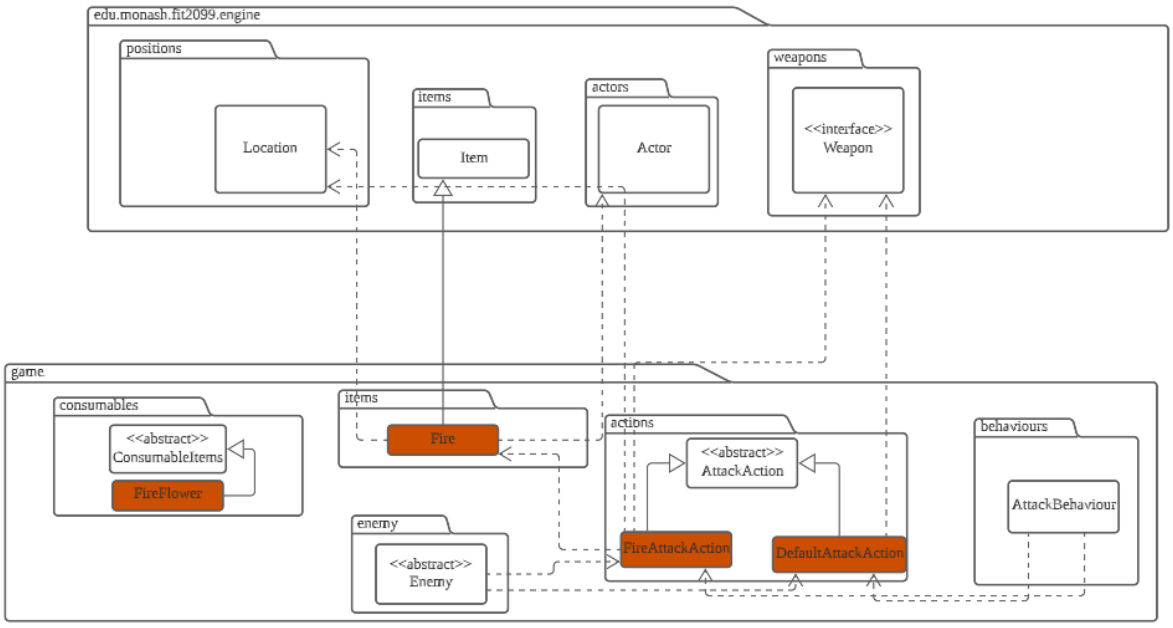
Tree:

the tree's children (sapling, sprout and mature) when they grow, they have a chance to spawn a fire flower, but because of DRY, I decided to make a function called "attemptSpawnAFireFlower" in the tree class that is called by the children, so they call it when growing making it so that they can spawn a fire flower, but they don't know about its existence only the tree class does

most of the requirement logic is in part 2, so the more interesting design decision are made there

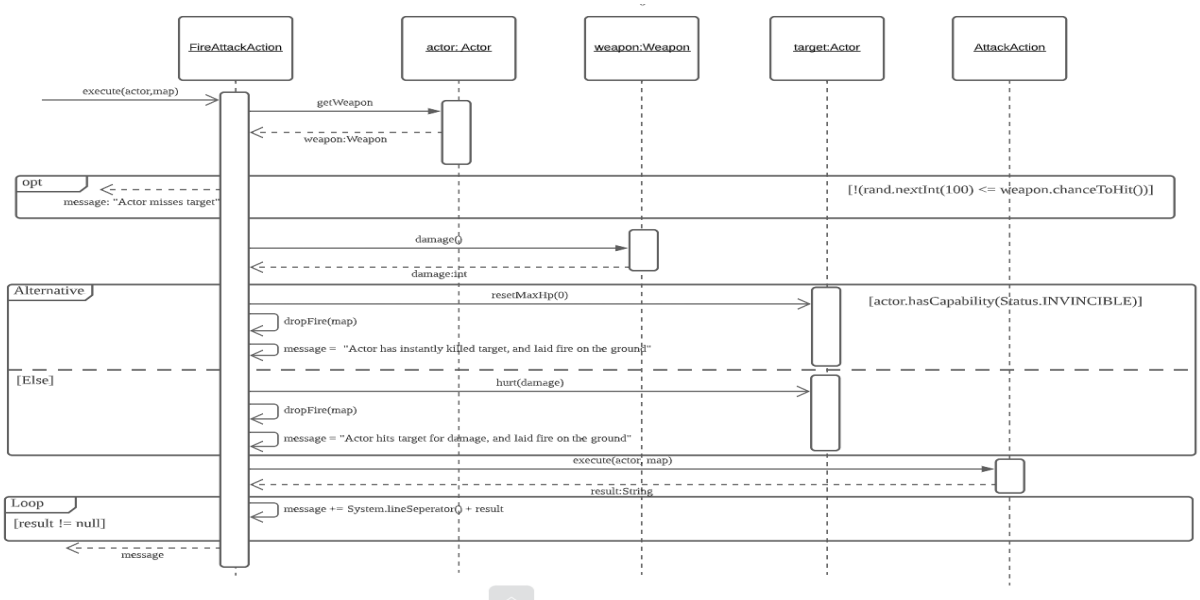
REQ4 Part 2 (Done by Jia Shin)

Class Diagram

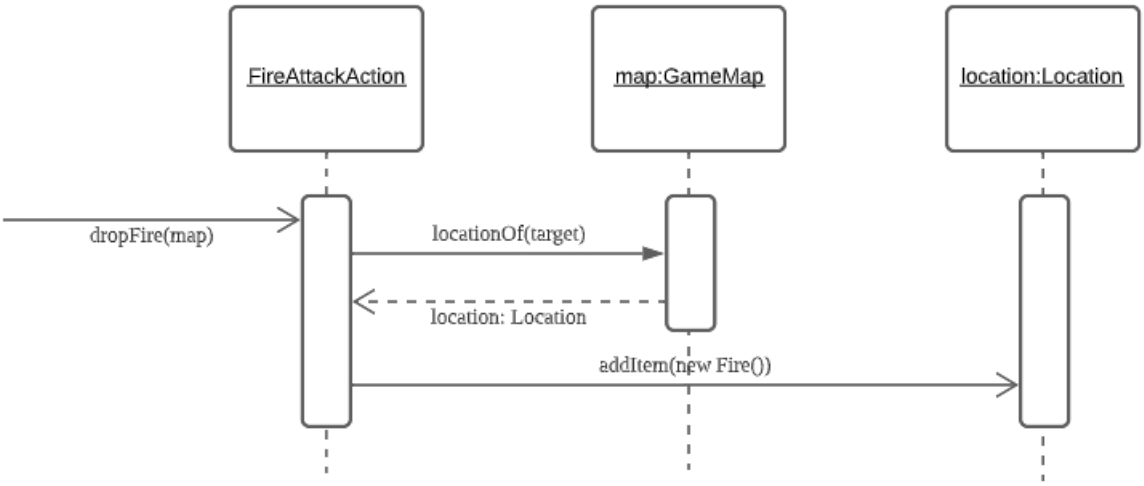


Sequence Diagram

- FireAttackAction.execute()



- FireAttackAction.dropFire()



Design Rationale

Newly created/modified classes with new relationships

FireFlower

Extends the ConsumableItems class. Allows actors to consume it to have fire breathing capabilities

Fire

Extends Item class. Originally we implemented it to extend Ground class, but by doing so we need to keep track of the ground overwritten, to spawn back the ground overwritten after the fire has dissipated after 3 ticks, so we avoided association between Fire and Ground.

DefaultAttackAction

Extends AttackAction. The normal attack done by either the player or most of the enemies. Adheres to DRY principle

FireAttackAction

Extends AttackAction. The fire attack done by Bowser or when the player has consumed a FireFlower that will launch a Fire at the target's location. Adheres to DRY principle

AttackBehaviour

Slightly modified to check if actors have fire breathing capabilities, if so then launch a fire attack, else default attack

Enemy

Modified to allow different types of attacks to be done to it depending on capability of the player

AttackAction

Modified to be an abstract class. In assignment2, we did not foresee that there would be different types of attacks available. Back then, our implementation was split into 2 parts: how to hurt the enemy, and what to do if the enemy is dead after hurting it. We know that there is only one way that an actor can die, which is when it is not conscious. We also know what happens when it dies, which is to drop all its items and remove them from the map. The only thing that differs between different types of attacks is the way the attack is carried out. For example, normal attacks just hurt the target, whereas fire attack hurts the target and launches a fire at them. Hence, AttackAction's execute method just has the implementation to check for when the target is not conscious, drop all its items and remove them from the map. This way, we don't violate the Open-Closed Principle, because different types of attacks can just implement different logic on how to hurt the enemy, and use AttackAction's execute method to implement dropping items when the enemy is dead and removing them from the map without changing anything from AttackAction's execute method. Also, since we are not lumping up all different types of attack into just one single AttackAction class, it also adheres to Single Responsibility Principle, since each child class of AttackAction only has the responsibility of implementing how to hurt the enemy, rather than one AttackAction that needs to check all sorts of logic to attack enemies in different ways.

"Mario can consume this fire flower to use Fire Attack."

Since player can consume it, we made FireFlower extend ConsumableItem. In FireFlower's consume method, we make the actor have the capability of fire breathing.

"Once the actor consumes the Fire Flower, it can use fire attack action on the enemy. Attacking will drop a fire ▽ at the target's ground."

In the enemy's allowableActions, if the player has the capability of fire breathing, launch a fire attack. In FireAttackAction, launch a fire at the target's location.

"This "fire attack" effect will last for 20 turns"

FireFlower's ticker is set to 20, and will decrease when consumed, by 1 in tick method

"The fire will stay on the ground for three turns"

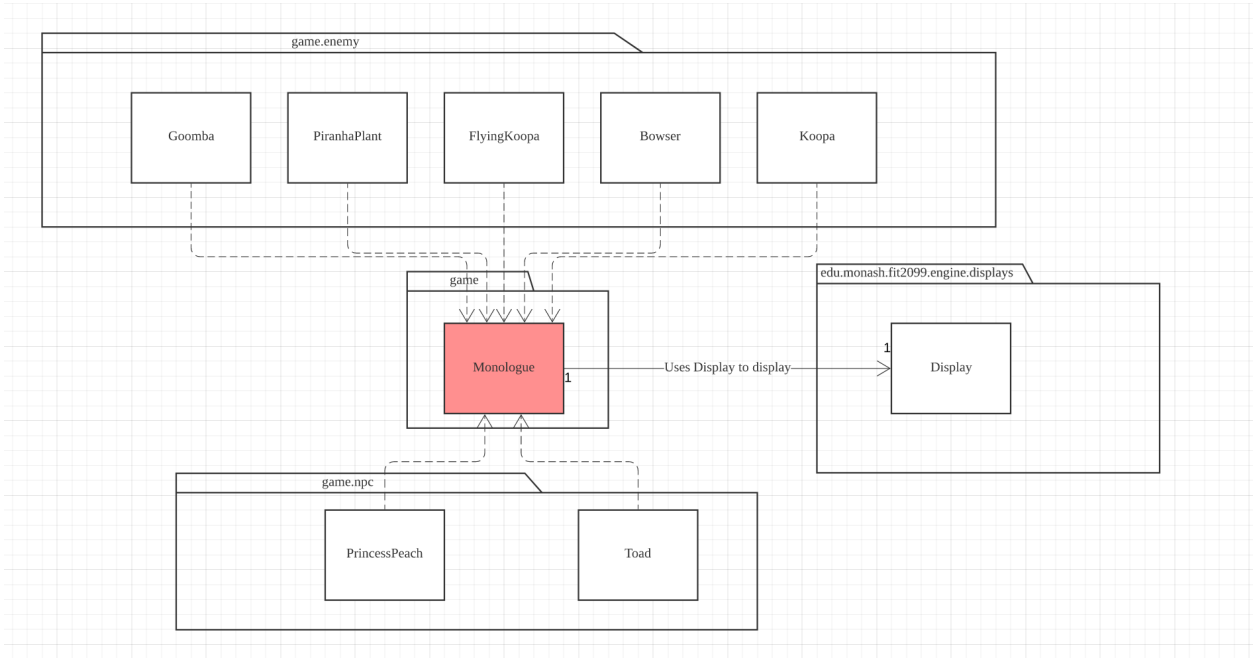
Fire's ticker is set to 3, and will decrease every tick by 1

"The fire will deal 20 damage per turn. "

Can utilize Fire's tick method. If there is currently an actor standing on it, hurt it by 20 hp.

REQ5

Class Diagram



No sequence diagram for this req as this is fairly straightforward.

Design Rationale

A new class Monologue is created to manage all of the strings that the actors are supposed to say. If we made all the strings in their respective class, it would make the class have too many responsibilities. By doing this, we can achieve the Single Responsibility Principle by having a class to be responsible for only the monologues of the actors.

WORK BALANCE AGREEMENT

Task: REQ1 implement, design, document and test the Lava zone and Warp Pipe
Teammate responsible: Fadi Alailan

Task: REQ2 implement, design, document and test Princess Peach, Bowser, Piranha Plant, and Flying Koopa
Teammate responsible: Khor Jia Shin

Task: REQ3 implement, design, document and test the Magical fountain
Teammate responsible: Lim Hong Yee

Task: REQ4 implement, design, document and test spawning the FireFlower(REQ4 part 1)
Teammate responsible: Fadi Alailan

Task: REQ4 implement, design, document and test the FireFlower and FireAttackAction(REQ4 part 2)
Teammate responsible: Khor Jia Shin

Task: REQ5 implement, design, document and test Speaking
Teammate responsible: Lim Hong Yee

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