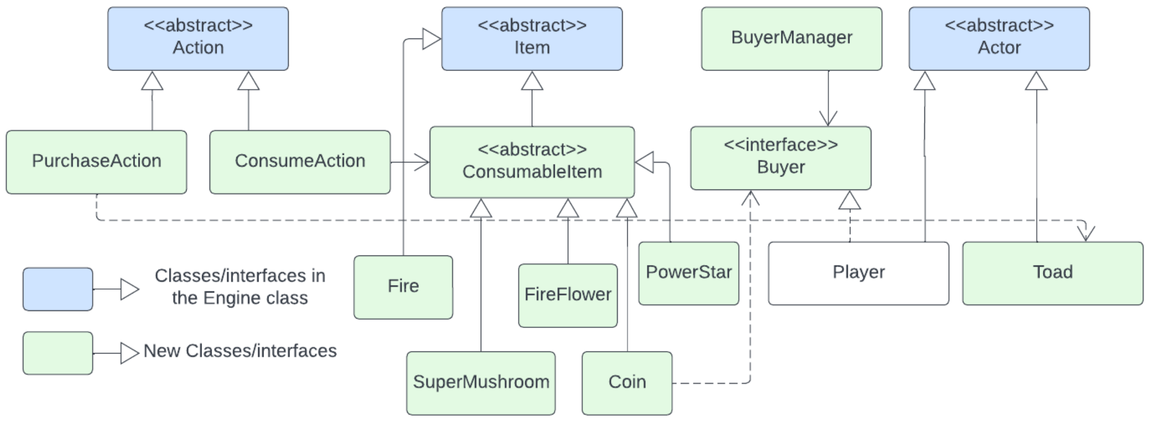
Group 5: Sara Hopkins, Peter Roberts, Timothy Moniaga

FIT2099 Assignment 3

Changes from previous assignments

**added massive bolded text headers for sections as seen below!!!

Magical Items



Above: Updated from previous assignment

SuperMushroom

The Supermushroom class now extends from the ConsumableItem abstract class.

SuperMushroom class is responsible for dealing with any changes to the player after consuming or picking up a SuperMushroom. The superMushroom has the ^ character when placed on the ground.

When a player walks over the SuperMushroom, the action PickUpAction will prompt the player to pick up the item, so they can add the magical item to their inventory. The user will also be prompted to consume the item straight away, to save an action that would be used to pick up the item and then consume it in another turn.

Upon picking up the item, the player will be prompted again with a consume SuperMushroom option, which will come from the ConsumeAction class. When the item is initialised, a consume action is added to the item's allowableAction ArrayList, which upon pickup, adds the item's allowable actions to the actor's allowable actions.

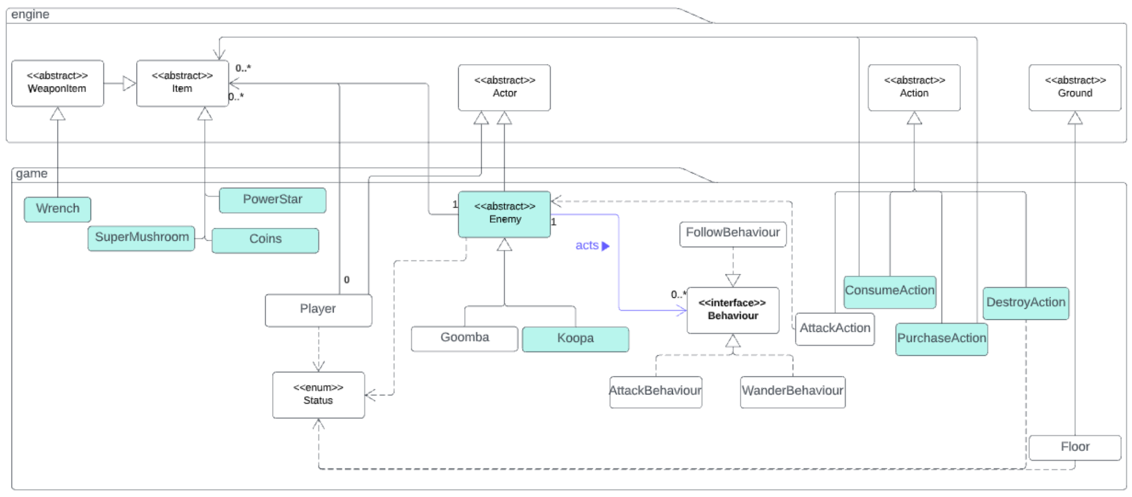
Alternatively, the player can purchase this item from Toad, by coming within 1 space of the Toad actor. The purchase action class is responsible for this action and will interact with the player wallet attribute (through getters and setters), to appropriately adjust the balance after purchase.

In increaseSuperMushroomHP method, it increases the players HP by an arbitrary high number, this is because throughout the game, the player can consume multiple superMushroom, permanently increasing their MaxHP, so having a number like 500 to increase HP may not be feasible in the long run of the game.

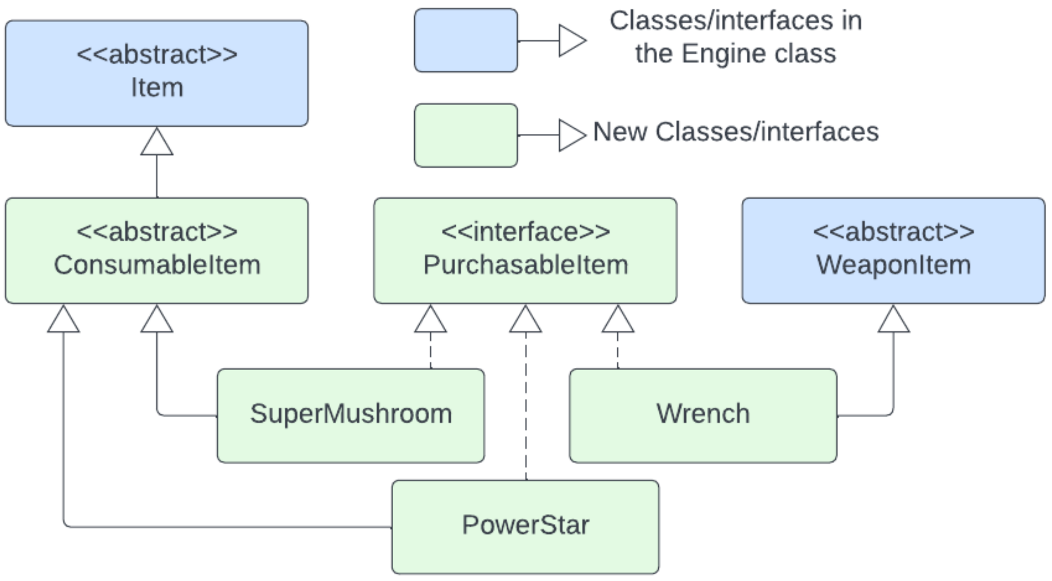
When an actor consumes the mushroom, their display character shifts to the uppercase version of whatever character is currently being used. Instead of manually hardcoding it to change ‘m’ to ‘M’, I implemented an update of the TALL status; the player class will check for this status and update to uppercase already. Added a toExecute() method that is overridden from the Consumable items class which is responsible for calling all relevant SuperMushroom methods that would be needed to apply to the player/gameMap when the SuperMushroom is consumed.

There is also a purchase method that is inherited from the PurchasableItem interface, which allows any purchasable item to interact with the players wallet and their respective PRICE.

OPEN CLOSE PRINCIPLE
The use of an abstract class, Consumable Item, which the consumable items inherit from, enforces the Open Close principle, as the addition of new Consumable items will require an extension from the abstract class. This will prevent the need to modify the existing item code in the future addition of other consumableItems.



Above: Previous UML for magical Items


Classes/interfaces in the Engine class
New Classes/interfaces
<<abstract>> Item
<<abstract>> ConsumableItem
<<interface>> PurchasableItem
<<abstract>> WeaponItem
SuperMushroom
Wrench
PowerStar

Above is the new UML diagram for magical items, it shows how SuperMushroom Item extends from a new ConsumableItem class, which extends from Item class.

Powerstar

PowerStar class extends from item class and will have a list of allowable actions which will ‘go to’ the player when they are on top of the item or they are holding it in their inventory.

There are two actions associated with the powerStar item, the consumeAction and the PurchaseAction. The consumeAction will be added to the player’s allowableAction list when the powerstar is initialised. The PurchaseAction will be added to the allowable actionList of Toad and when the player is in the vicinity of Toad, they will be able to receive this action.

The player can either use the PowerStar by consuming straight off the ground below them, placing it in their inventory and consuming within the remaining life of the item or purchasing the powerstar from Toad and once again, consuming the item before it ‘expires’.

The PowerStar will expire after 10 turns on the ground or in the players inventory, if it spends 3 turns on the ground, the player will only have the item in their inventory for 6 turns (keeping in mind that picking up the item counts as a turn) remaining.

Once consumed, the powerstar effects will last on the player for 10 turns. This is achieved by keeping the powerstar in the player inventory, if that's where it was consumed from. Or by adding the powerstar item to the players inventory. Then resetting the counter used by the tick method().

The portability of the powerstar will be changed at this point so it just sits in inventory, cannot be dropped, also consumeaction will be removed. Basically the actor/player should have no interaction with the item and its allowable actions after this point.

Power Star will print to console the remaining turns active for the powerstar when in inventory or on the surrounding ground and the “MARIO IS INVINCIBLE!” status, handled in playturn().

The attack action class will detect if the player has the POWERSTAR status/capability, and then adjust the attack damage accordingly.

The powerStar class has a method to add the item to the actor's inventory and remove the price from the actor’s balance when the purchase action is executed.

SINGLE RESPONSIBILITY
Similarly, the PowerStar class extends from the <<abstract>> ConsumableItem class. Hence, it embodies the **Single Responsibility Principle** as it’s only responsible for item actions related to the PowerStar Magical Item. It also embodies the same Open Close principle as SuperMushroom, not explained here due to repetition.

Wrench

The wrench extends the <<abstract>> class WeaponItem.
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 and whilst the player has the wrench, they can always break Koopa’s shell. The enemy can check the players status and take damage accordingly if the wrench is not used to break Koopa’s shell but for a normal attack.
The wrench cannot be lost by the actor once it has been purchased. It will stay in the actors inventory until they use the dropItemAction to drop it.
The Wrench class implements the purchasableItem interface and adds to the purchase method a call to update the actor’s capability when they have purchased a Wrench item.

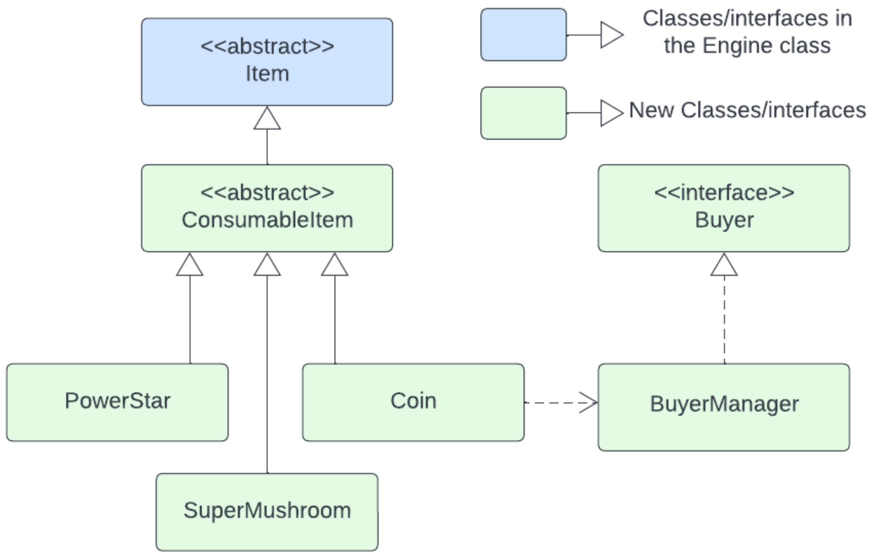
INTERFACE SEGREGATION PRINCIPLE
The **Interface segregation principle** is followed by the design of PurchasableItem and the way in which Wrench extends this. Wrench will implement all of the PurchasableItem methods in the PurchasableItem <<interface>> as they are applicable to the trading requirement. The Wrench class doesn’t implement or inherit any unused methods from the PurchasableItem 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.
When the coin is initialised, it is also added to the registerInstance ArrayList, which keeps track of items that need to be reset.

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, it's worth and its position on the map.
The coin class can be “consumed” by the player, and then the toExecute() method is called, it checks the supplied actor against the BuyerManager arraylist of buyers, and if the actor is in the list, it adds the coin’s value to the wallet balance.

SINGLE RESPONSIBILITY PRINCIPLE
The coin class is the only class that interacts with the player’s wallet. The coin class is responsible for all wallet and money balance interactions. Therefore, this class only has one job and supports the Single Responsibility Principle.

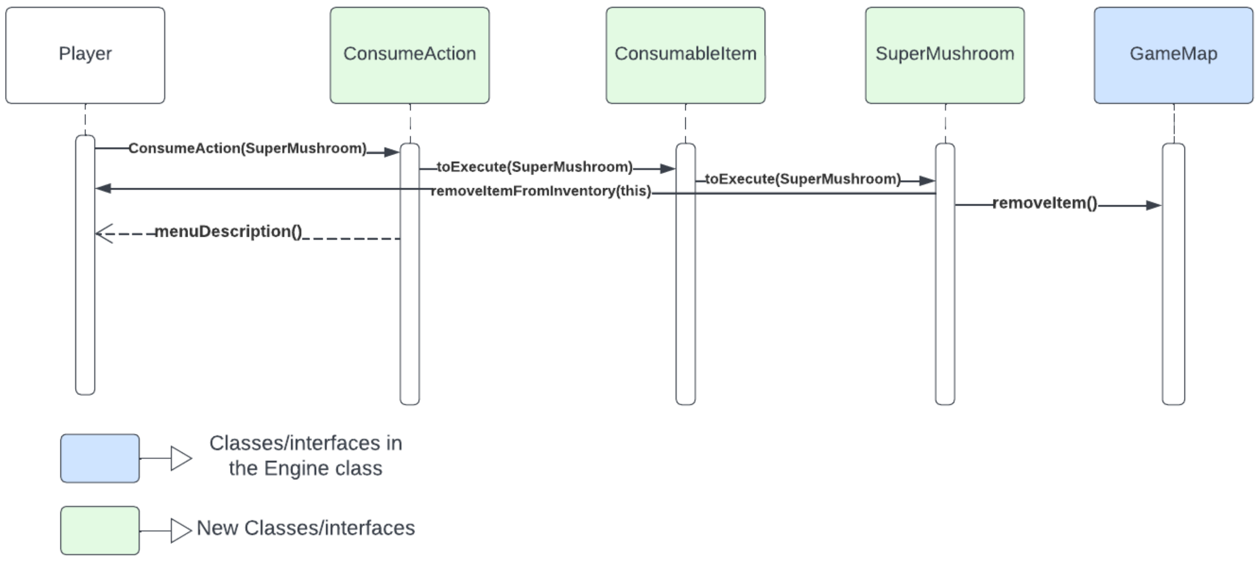


above: updated UML for coin class, depicting Coin dependency on BuyerManager

ConsumeAction

The consumeAction class was added when we realised that we needed to give the user the option to consume an item they purchased. We then decided to further extend it to allow the actor to consume a magical item on the ground instead of having to pick it up and then consuming it (reducing it from a multi-turn action to a single-turn action).

The consumeAction class has an instance of a consumableItem assigned to it through the constructor, which then allows it to call the abstract method toExecute().



The above sequence diagram depicts a player choosing to consume an item, selecting the consume item action from the menu.

OPEN CLOSE PRINCIPLE
The Consume action allows for further extension of the system, for more consumable items to be added, without needing to modify existing classes. The consumeAction and the ConsumableItem class both help to support this principle.

ConsumableItem

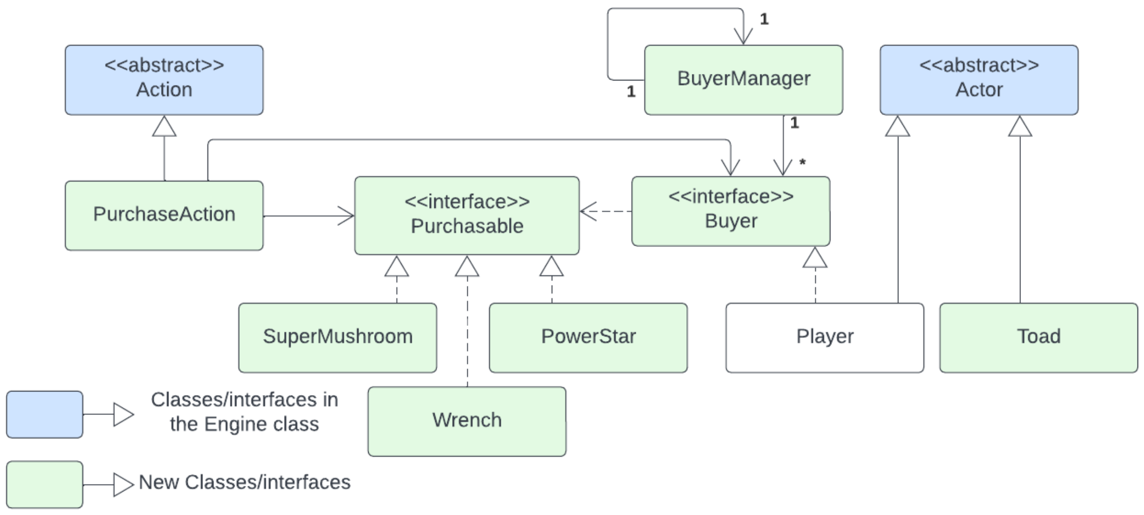
ConsumableItem is an abstract class that groups together the Magical items that can be consumed and the coin items. The main purpose of this class is to allow the ConsumableAction class to call a general method toExecute, that can be then overwritten by child classes of ConsumableItem.
This overriding process allows each subclass to write their own method, their own process of method calls that occurs when that item is 'consumed'.
This is very important as each of these 'consumable' items apply a different set of effects to the actor and the system when consumed.

DEPENDENCY INVERSION PRINCIPLE
The ConsumableItem class allows items to inherit consumable properties and the toExecute method but removes the need for these items to have a dependency on the consumeAction. Previously it was designed to have an instance of the ConsumeAction in each consumable item class. This is poor design as it commits the item classes to function a certain way instead of creating a flexible system that allows them to function in a consumable method when appropriate.

Trading

PurchaseAction

Purchase Action is a child class of abstract Action class. It extends the action mainly to implement the execute method, which will make a call to the purchasableItem interface and to its purchase() method. The execute method is run whenever the player selects the PurchaseAction in their turn.
Which is why it's important that this method be implemented throughout all item classes, where they can be purchased by the actor from Toad.
This class also has an instance of Purchasable item as an attribute, updated when the Action is instantiated in the constructor.



above: Depiction of purchasable Item system and class relations

OPEN CLOSE PRINCIPLE
Again, the use of an extension of the Action class allows us with a versatile system design that can implement many more purchasable items in future. All purchasable items can be action through this execute method which is designed with a general interface call, to allow for the extension not modification of this system.

Purchasable

This interface will be responsible for creating the general purchase() method that can be called by the PurchaseAction execute() method. This means that the PurchaseAction can avoid instanceOf and typeCasting comparisons and instead access the purchase action of whatever purchasable item it has been given during instantiation.

The PurchaseAction class will be instantiated with an instance of the purchasable item when initialised. This will allow the PurchaseAction, when executed, to make a general call to the method of whichever specific purchasable item supplied.

INTERFACE SEGREGATION PRINCIPLE
This class creates an interface with general methods needed by all classes that implement it. These methods that are implemented, are the only way the purchasableItems can be accessed by the purchaseAction. The interface is small and serves only the purpose it is needed for, and doesn't contain methods that are not needed for every class implementing it.

Toad

The Toad class extends from the Actor class, and Toad will have the display character 0. The Toad actor, on its turn, will check the surrounding exits to find an actor. If it finds an actor, it will add purchase actions for all purchasable items to its own allowable actions list, adding new purchase actions for SuperMushroom, PowerStar and Wrench items and a speakAction with the actor/buyer found. If there is no player in the adjacent spaces, Toad will take a doNothingAction().
Toad can add these actions to its own list because in the engine code, it specifies if the player is in the surrounding exits of another actor, it can access its allowableActionList.

To allow Toad to check if the actor within its exits is the player, not just another actor (an enemy), the Toad class will check to see if the player has the HOSTILE_TO_ENEMY status. This works since the player (mario) is the only entity in this game that will have that status.

Buyer

The buyer class was designed as an interface to allow certain classes and methods to access the buyer instance (the player Mario). This allows for classes to only allow certain characters to do certain things, but then can also access things like the player wallet as well. The buyer class is very important in accessing any attributes or methods that players would have that all actors would not have. This generally occurs due to a class only being able to see the player as an actor type, hence not being able to specify certain methods from the player class.

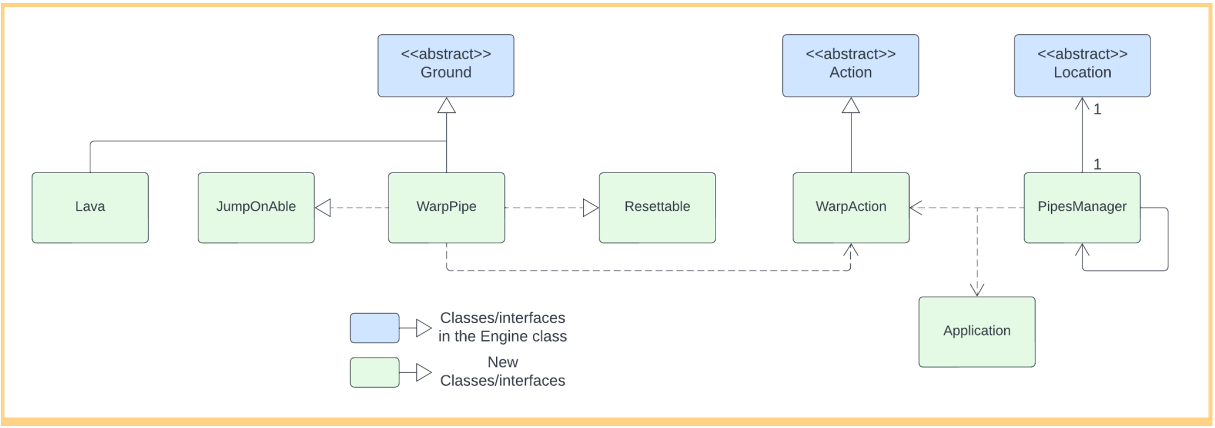
INTERFACE SEGREGATION PRINCIPLE
This class creates an interface with general methods needed by classes to access from the player. These methods that are implemented, are the only way that wallet and player bottle can be accessed by the other classes. The interface is small and serves only the purpose it is needed for, and doesn't contain methods that are not needed for every class implementing it. The interface contains any methods that separate the player class from other actors and are necessary for other classes to function.

BuyerManager

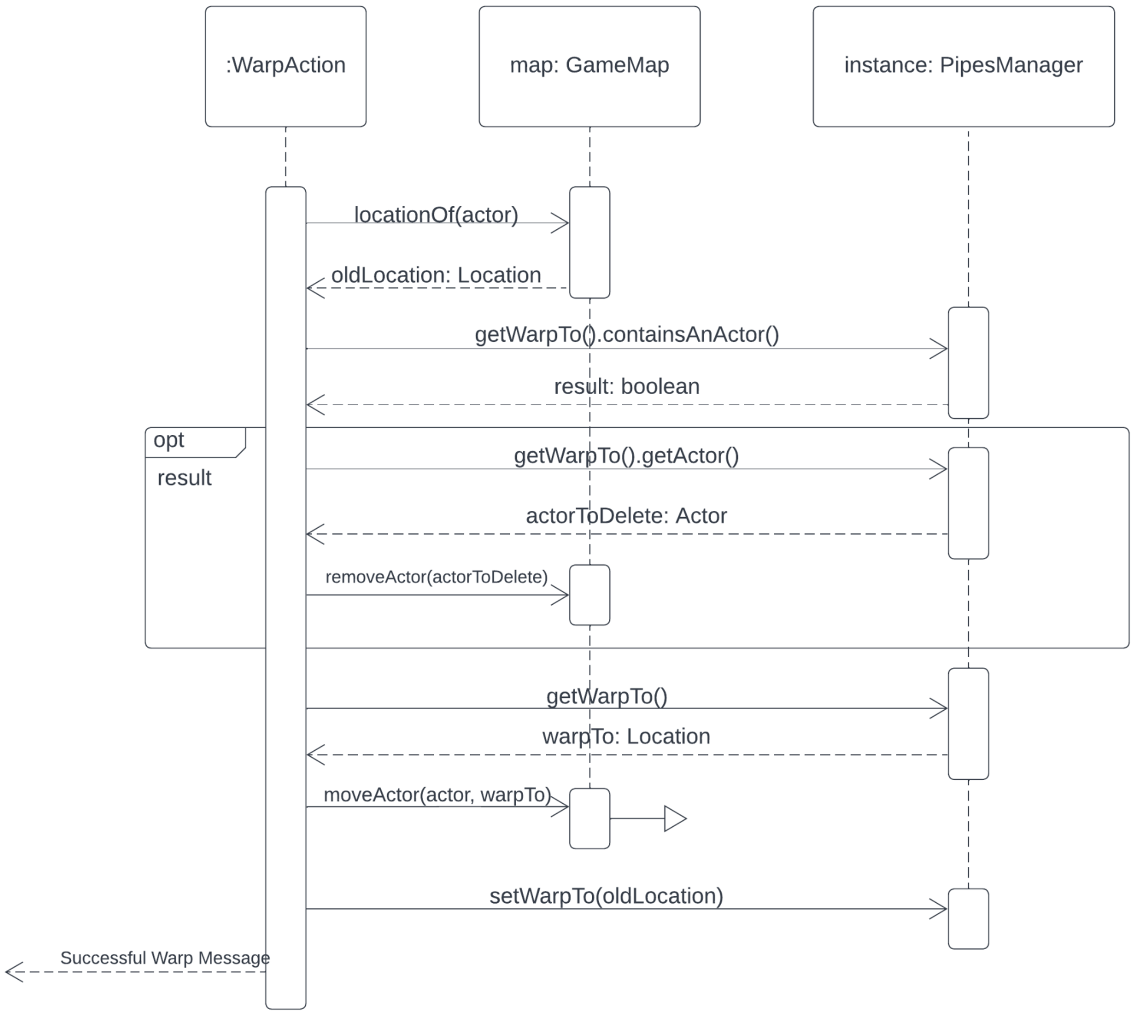
The BuyerManager class has one simple responsibility; to maintain a list of Buyers present in the current game. It was created for the aforementioned buyer issue, and allows classes to access a Buyer instance and all relevant methods by simply checking if the actor parameter the method uses, is contained in the BuyerManager arraylist.

SINGLE RESPONSIBILITY PRINCIPLE
The BuyerManager class has the sole responsibility of managing itself, creating a list of Buyer types in the game. And keeping track of Buyers and, upon request returning an instance of the Buyer.

Lava Zone



UML diagram for the Lava Zone requirement



Sequence Diagram for the warping requirement in Assignment 3

WarpPipe

New ground type to represent warp pipes. Adds a WarpAction to the Player when the player is standing on the WarpPipe. This class also handles the spawning of PiranhaPlants.

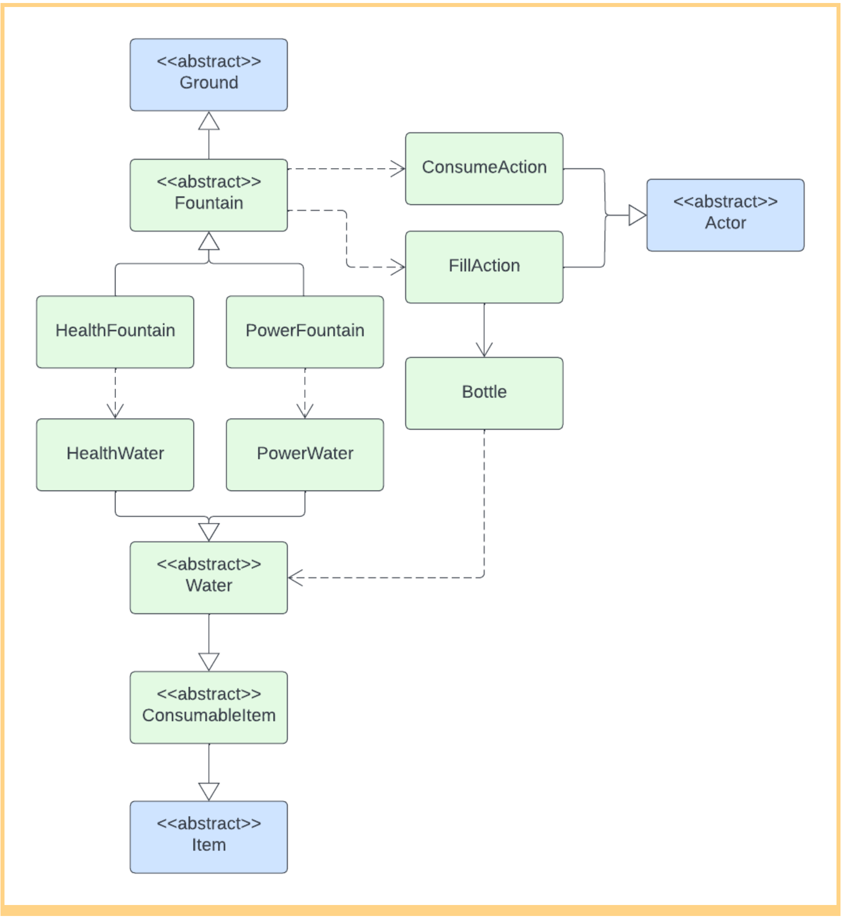
WarpAction

This action allows the player to move between the two maps when standing on the WarpPipe.

PipesManager

The PipesManager is a singleton responsible for holding the next warpTo location. For example, if Mario stands on Warp Pipe A (map 1), he can take the WarpAction. This will move him to Warp Pipe B (map 2). PipesManager will then update its warpTo location to Warp Pipe A, as mario’s next warpTo location will be Warp Pipe A. (He goes back to the pipe he came from). This design **reduces dependencies.** It allows only WarpAction and Application to depend on PipesManager, rather than having all the pipes interdependent, or having a separate class for the pipe on the second map. This also follows the **single responsibility principle.** The PipesManager is responsible for remembering the warpTo location, the WarpAction is responsible for making warping available to the player and the WarpPipe is responsible for representing warping on the map. The poorer design alternative of this would be to have one “God” class handle all three of these functions.

Magical Fountain



Above: new Magical Fountains UML

Fountains

An abstract fountains class was designed to house general methods that the health and power fountains would share. It extends from the ground and the fountain has been chosen to be a ground type. The main method is adding to the players allowable actions, in this method we can check the capabilities of the actor, to determine whether they are a player or an enemy.
It checks to see that the ground contains an actor, allowing the action only when the actor is on top of the fountain ground.

OPEN CLOSE PRINCIPLE
The use of an abstract fountain class will allow the addition of other fountain types in future without having to edit code or repeat methods in the new class. This fountain class allows other classes to easily extend the methods and properties of a fountain without requiring modification.

Health Fountain

The health fountain is a fountain at which actors can either drink health water (if they are not a player type) or they can refill their water bottle (if they are a player type).
The health fountain does not have any methods as its reason for its existence is simply to have its own display character ‘H’.
It extends from the abstract fountain class, inheriting its allowable actions method.

Power Fountain

The power Fountain similarly extends from the abstract fountain class, inheriting its allowable actions method. The player type actors can go to a power fountain and fill their bottles with Power Water. And any other actor types can go to the fountain and drink Power Water from it.
With the power Fountain being a ground type, the allowable action to drink or fill up your bottle extends to the surrounding exits around the fountain instead of just on the fountain.

Health Water

Health Water can be obtained from the Health Fountain, and will increase the player's HP points by 50 or till max. The health water can be added to the player's bottle stack as it extends from the abstract Water class. The health water is also a consumable item as water extends from the ConsumableItem abstract, which means that the Health Water class makes use of the execute method to increase the player's HP.

Power Water

The PowerWater class has an identical setup to the above mentioned Health Water class, instead of altering the player's HP, it increases the player's intrinsic damage by 15 points. It also uses the inherited execute method from the ConsumableItem class to do this.

Water

OPEN CLOSE PRINCIPLE
The use of an abstract Water class will allow the addition of other fountain types in future without having to edit code or repeat methods in the new class. This fountain class allows other classes to easily extend the methods and properties of a fountain without requiring modification.

Bottle

The Bottle is a non-portable item designed for Mario. The non-portable means that Mario always has the bottle and cannot drop it. The bottle has a stack implemented, using the abstract water class, so that all types of water can be held in the bottle. The optional challenges for Magical Fountain have been completed meaning that this bottle can be obtained by standing in the surrounding spaces of Toad. Only the player type can receive this bottle, and once they have a bottle, they cannot receive another bottle.

Flowers

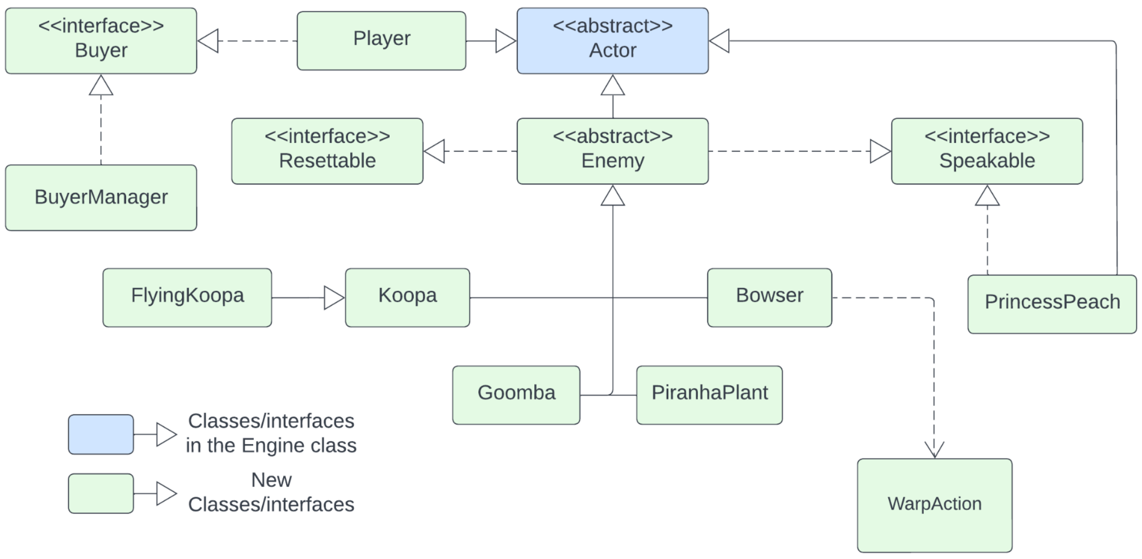
FireFlower

The fire flower has a method and its spawning is handled in the abstract trees class. The fire flower has a constant spawn chance of 50% from all types of trees hence why it is handled in the abstract to avoid violating DRY.
The Fire Flower item also functions similarly to the Powerstar however it cannot be held in inventory and must be consumed off the ground. The timer functions identically however, the fire attack booster places the fire flower in the actors inventory after consumption and sets the counter to 0, and sets the FIRE_ATTACK status.
This allows the other class, AttackAction, which will handle the attack with fire, to determine when an Actor has consumed the fire flower and is still within their 20 turns.
If AttackAction detects this status, it will add a fire item under the target.

Fire

The fire item class also tracks the turns of the fire on the floor, and if the internal fire counter ticks over 3 turns, then the tick method will remove the item from the ground.
Each fire instance has a counter that is assigned the value 0 when the item is initialised.

Enemies and Allies



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 Attack Action'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 a 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.

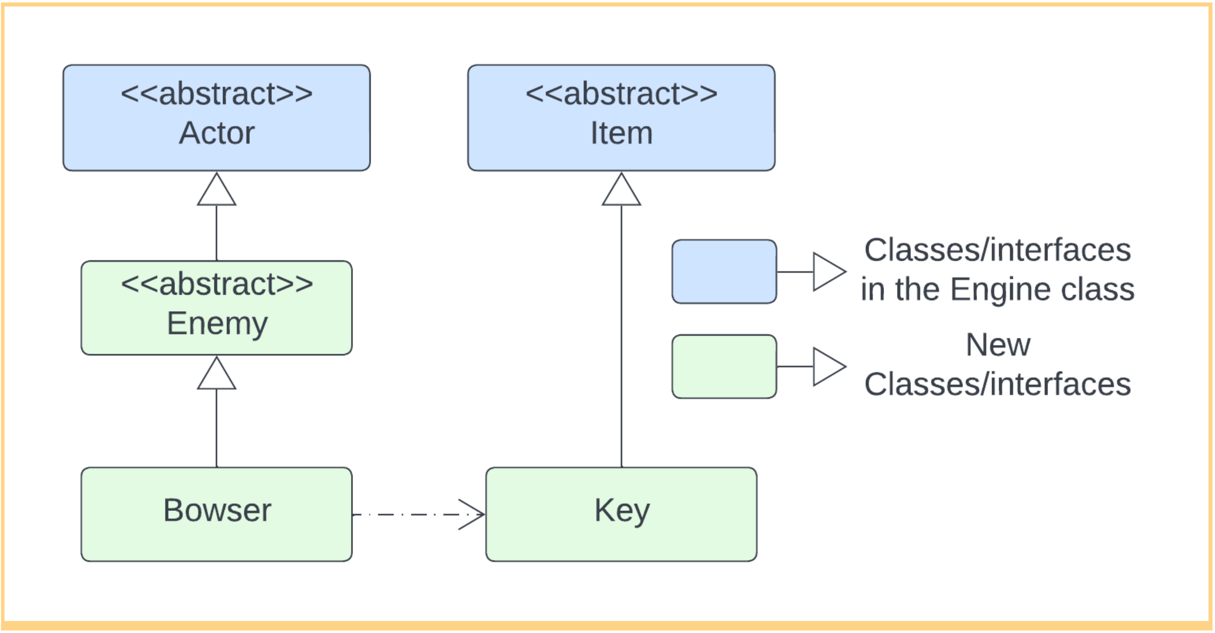
Princess Peach

The Princess Peach character has the class PrincessPeach, which extends from Actor. In Peach's playturn, she has the capability, based on a boolean, to speak a random string each second turn.
Peach checks if another actor is standing next to her on the map, if they are and they have HOSTILE_TO_ENEMY status, then Peach also checks to see if the player has the KEYHOLDER status.
Then Peach adds an UnlockCuffs action to her allowable actions, which can be accessed by any player standing in the surrounding spaces to the Peach character.

Bowser

The Bowser Actor extends from Enemy and has the main function of being an attacker of the player. Bowser is implemented with a TreeMap, which allows the actor to have a sorted behaviour list, meaning Bowser can have attack as its priority. The Bowser has a “home” location, a place on the ground where Bowser is initialised. This allows Bowser to return to his starting position upon reset.

Key



Above: a UML with Bowser and Key class

The key is an item that is held in the Bowser’s inventory until defeated by the player. The player, upon picking up this key, will obtain the KEY_HOLDER status.

FlyingKoopa

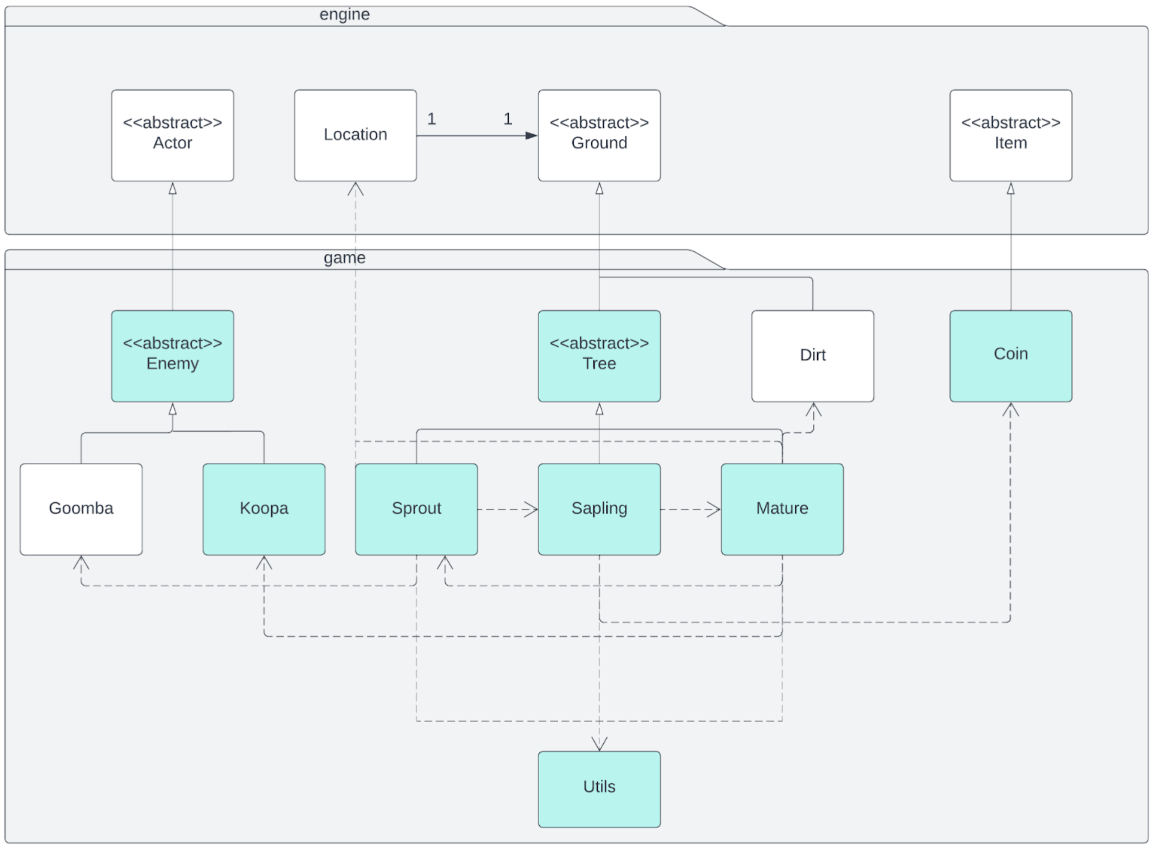
Flying Koopa is an enemy that extends Koopa, it has all the same functionality of the Koopa Enemy but it has a FLY status that allows it to have different move patterns. This FLY status allows the FlyingKoopa to move to higher ground without jumping, in the same way Player can when buffed with a powerstar. Flying Koopa also has an additional speak statement aside from Koopa.

Piranha Plant

The Piranha Plant is an enemy that is placed over the warp pipe and has to be defeated before accessing the secondary map. A counter in the tick method tracks the age, placing a piranha plant when the age is equal to 1, on the second turn. The Plant can’t move and only attacks if the Player stands on the surrounding spaces.

UnlockCuffs

Trees



UML Class Diagram for Tree Features

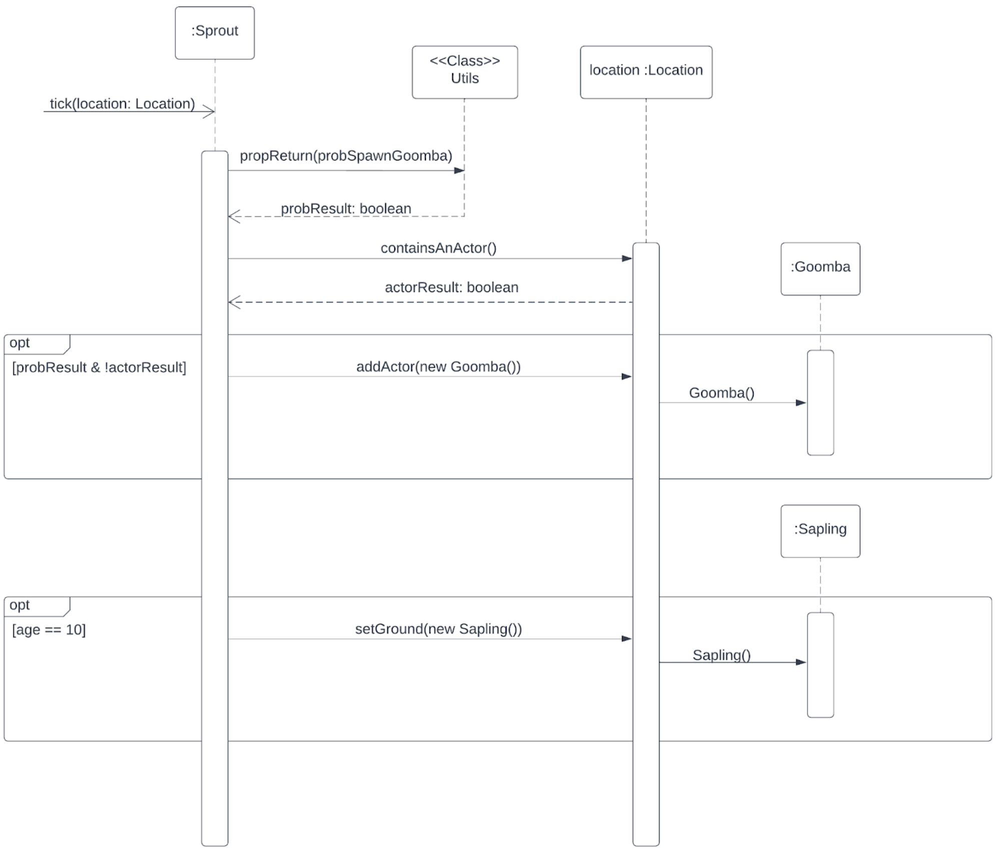
Key Decision: Sprout, Sapling and Mature as Subclasses of Tree.

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: Tree FireFlower Method.

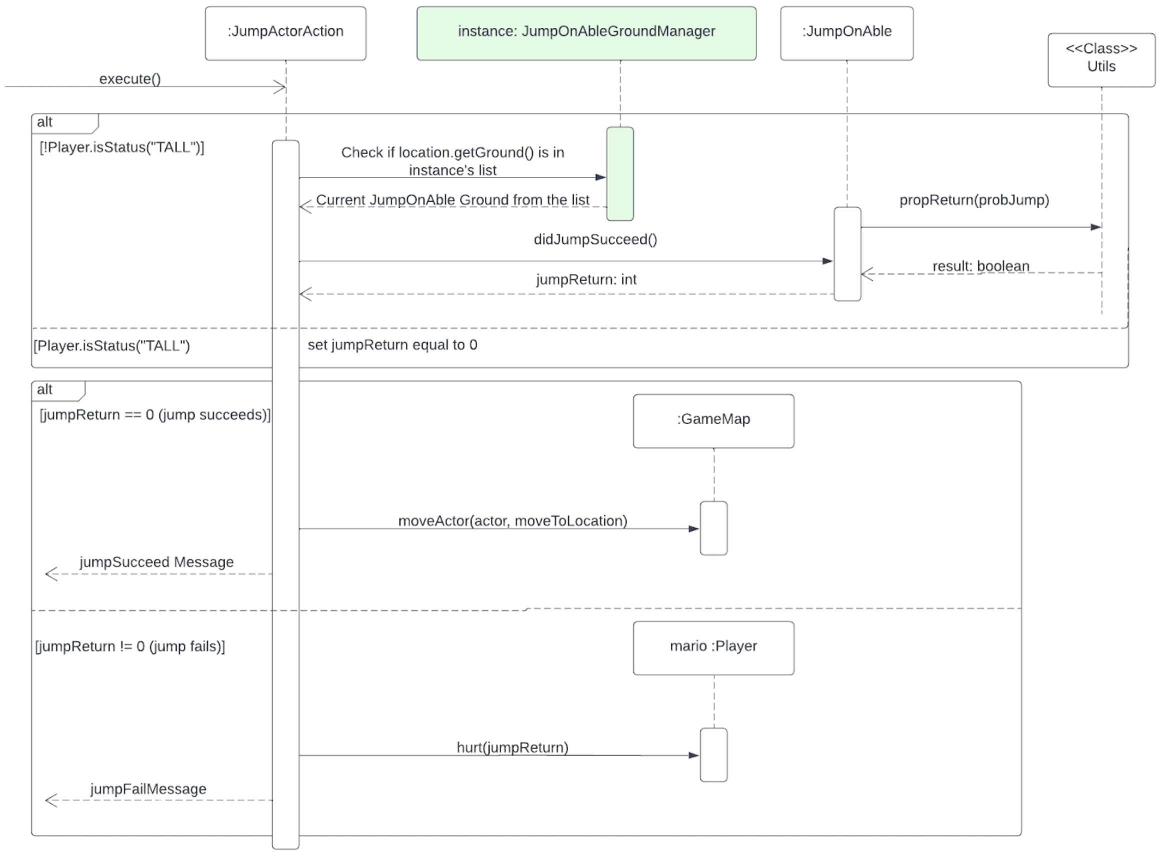
In the abstract Tree class, a fire_flower_spawn method was added. The purpose of this method was to meet the requirement of all types of tree having a 50% chance of spawning a fire flower each turn. By placing this method in the tree attract class, and then calling it in the tree's tick method, all subclasses of tree will experience this during their tick.
DRY: this supports the don't repeat yourself principle by avoiding the repetition of a common method through all subclasses.

Key Decision: Utils Class

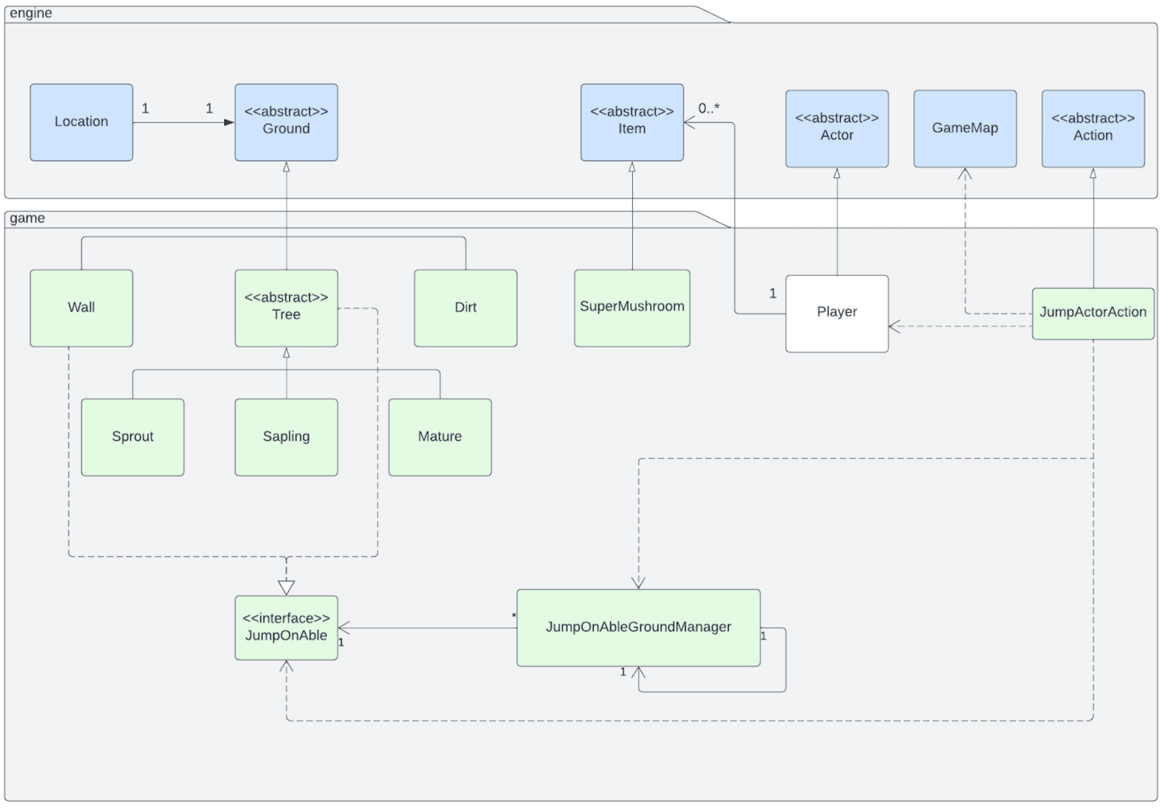
In this project many methods in many classes require probabilities. E.g. Sprout has a 10% chance of Spawning Goomba. To handle this, we have a Utils class with a static method probReturn(probability) that returns a Boolean based on a random number generator.

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


:JumpActorAction
instance: JumpOnAbleGroundManager
:JumpOnAble
<<Class>> Utils
execute()
alt
[!Player.isStatus("TALL")]
Check if location.getGround() is in instance's list
Current JumpOnAble Ground from the list
propReturn(probJump)
didJumpSucceed()
result: boolean
jumpReturn: int
[Player.isStatus("TALL")
set jumpReturn equal to 0
alt
[jumpReturn == 0 (jump succeeds)]
:GameMap
moveActor(actor, moveToLocation)
jumpSucceed Message
[jumpReturn != 0 (jump fails)]
mario :Player
hurt(jumpReturn)
jumpFailMessage

Sequence Diagram for the Jump Feature



UML Diagram for the Jump Feature

Key Decision: JumpOnAble Interface and Jump~~Actor~~Action Class

A JumpOnAble interface will be implemented by all Ground types that can be jumped on. Ground types that implement the interface will be managed by JumpOnAbleGroundManager. They will implement a didJumpSucceed() that will return the fall damage incurred by the jump (0 if the jump succeeds). The JumpOnAble interface also has a default method destroyedByPowerStar. This default method runs in tick() for the Ground types, replacing the ground type with dirt and spawning a coin if an invincible actor is standing on it.

The JumpAction 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 JumpAction code. The new ground type will simply implement the JumpOnAble interface. - **Don't Repeat Yourself:** The default method in JumpOnAble interface minimises repeated code, taking advantage of the fact that all JumpOnAbles are destroyed by PowerStar in the same way.	- **Don't Repeat Yourself:** This implementation means we have to add the JumpAction 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.

Speaking

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.~~

The monologue ideas and set up from the previous assignment were completely scrapped. The previous SpeakWithToadAction class wasn't designed in an OO manner, violating SOLID principles, making it hard to extend.

Instead, the SpeakWithToadAction class was replaced with a SpeakAction class. And then to allow for the extension of this action to all different prompts for different actors, a speakable interface was created.

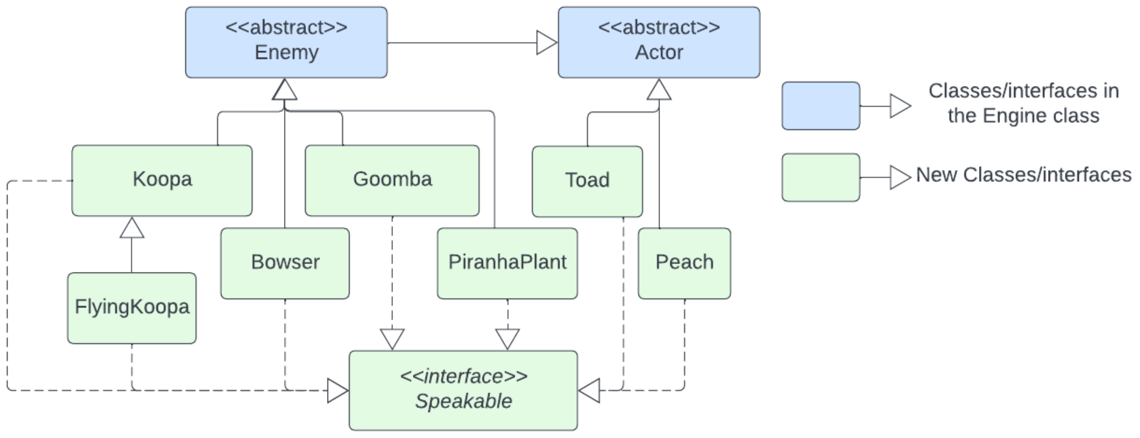
Any actors who can use the speakAction also implement the Speakable interface. This allows a general speak() method in the interface to be called dependent on the actor speaking.

Two constructors are used for the two cases in which the action can be used, an actor choosing a speak action or the player choosing a speak with Toad action, which is now handled through the SpeakAction class. The second constructor handles the second case by allowing the input of an actor.

When the actor instance is not none, the speak method in Toad will add the option of the player to speak with Toad, else toad will speak every second turn.

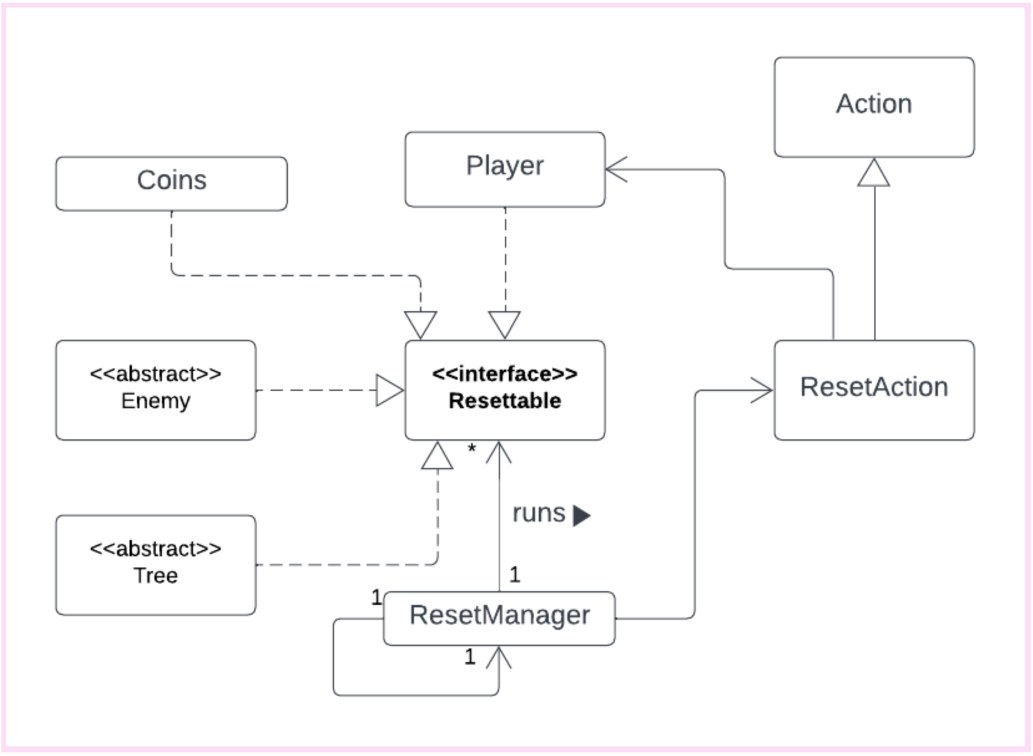
Another piece of logic was ensuring that all actors speak on every second turn, so in each actor, they have a boolean which is updated to the opposite every turn. And then an if-else that runs through a speak action or all the other logic for the Actors turn.

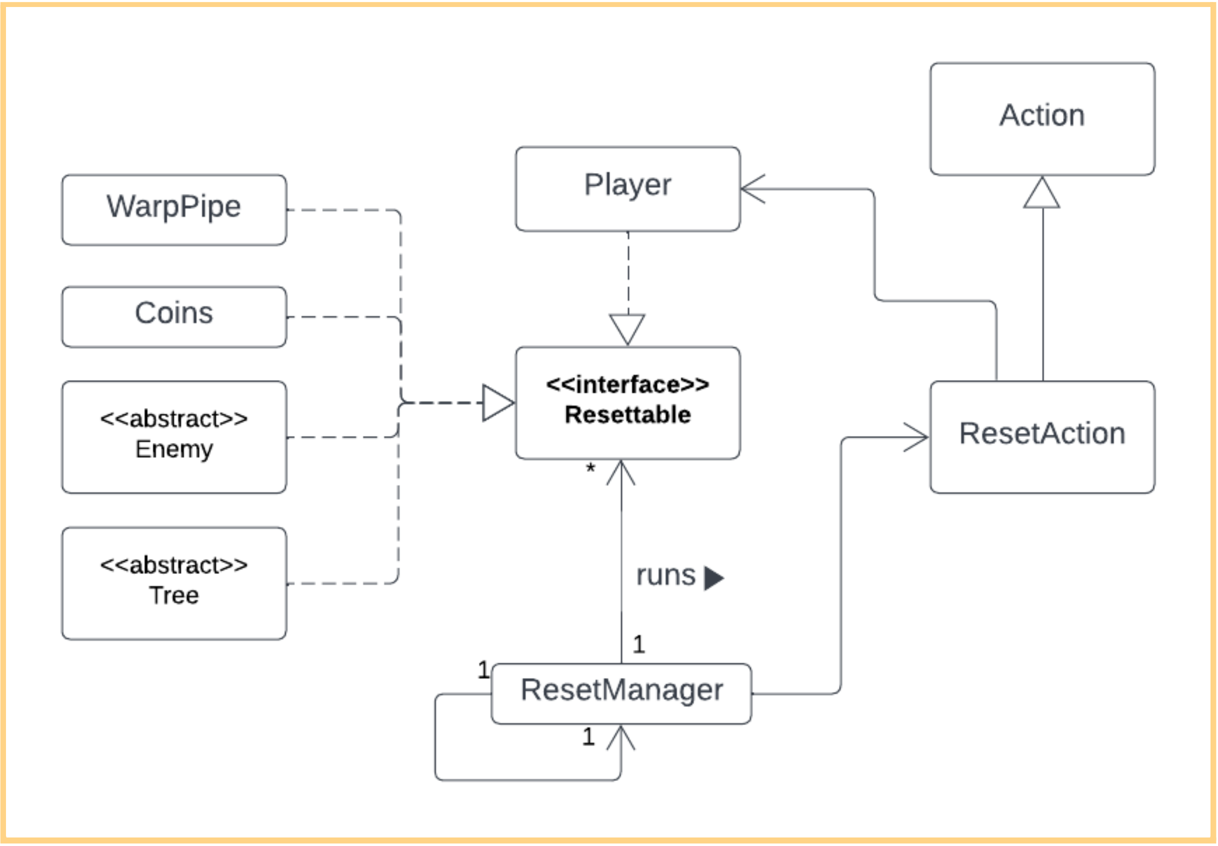
OPEN CLOSE PRINCIPLE
The use of an Speakable interface makes the SpeakAction open for extension as it only calls an interface method not individual, specific class methods.



Above: the updated UML diagram for Speak/Monologue

Reset





All classes that are required to be reset will be implementing <<interface>> Resettable rather than creating an association towards a Resettable class which would store ArrayLists of all of the objects that are required to be reset. This is done to adhere to both the Interface Segregation Principle (ISP) and Dependency Inversion Principle (DIP).

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. Reset follows the **Dependency Inversion Principle:** all classes that are required to reset depend on the abstraction of <<interface>> resettable.	The largest disadvantage to this method is the requirement for many classes to implement the resettable interface. This has the potential for **repeated code.**