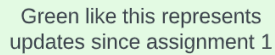


FIT2099 Assignment 2

Changes from assignment 1



Green like this represents updates since assignment 1

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 pickup 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. ~~There is a method that adds the consume action to the items allowableActions ArrayList when it detects the superMushroom in the players hand.~~ 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.

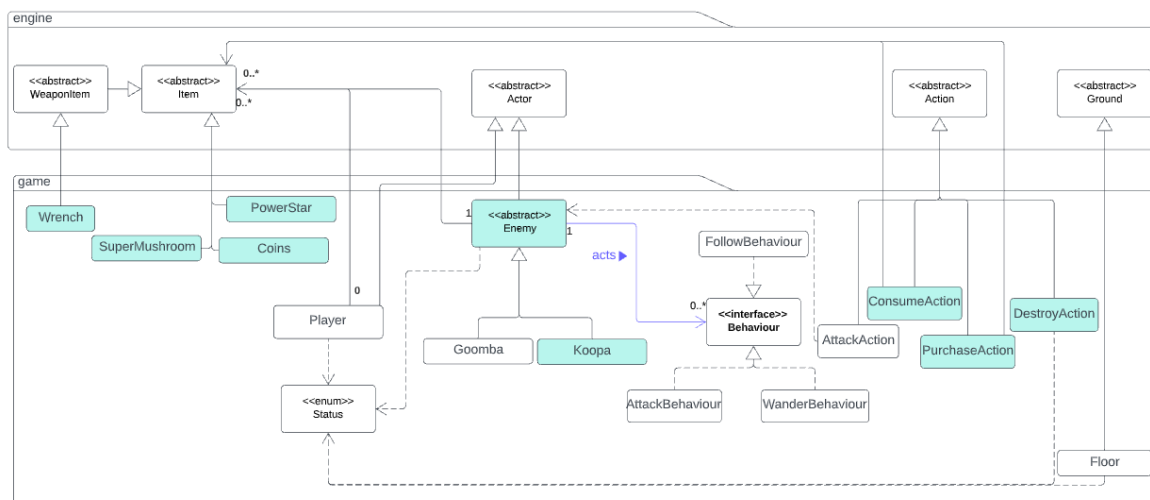
In increaseSuperMushroomHP method, it increases the players HP by an arbitrary high number, this is because throughout the game, the player can consume multiple superMushroom, permanently increasing their MaxHP, so having a number like 500 to increase HP may not be feasible in the long run of the game.

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, 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.

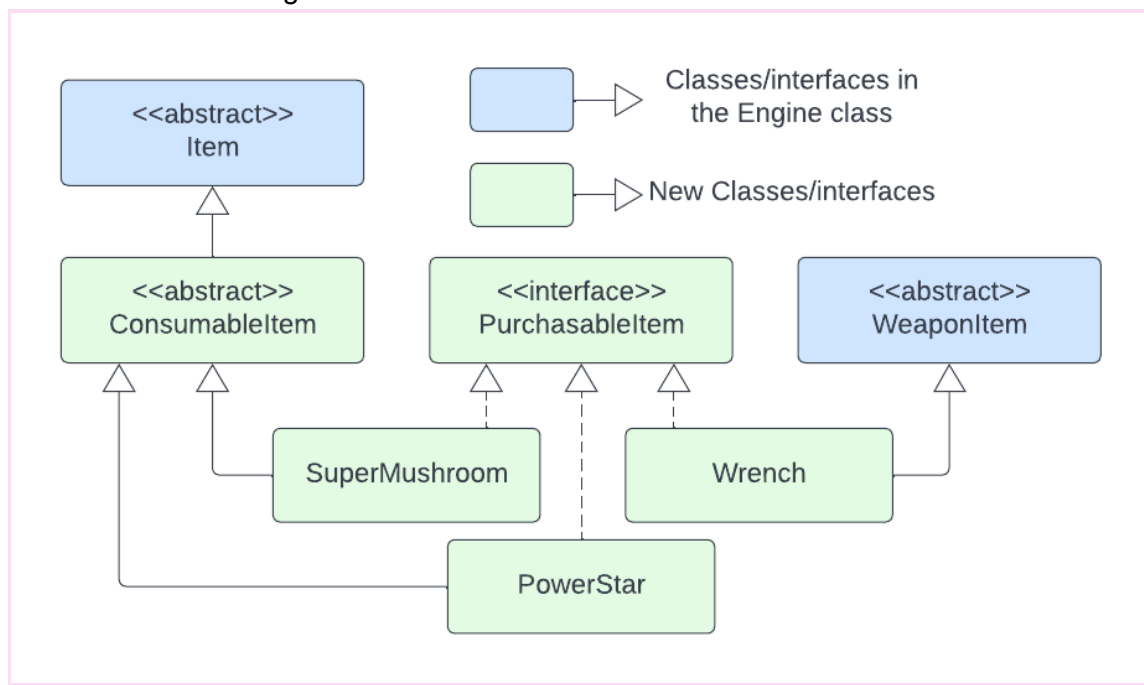
There is also a purchase method that is inherited from the PurchasableItem interface, which allows any purchasable item to interact with the players wallet and their respective PRICE.

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.



Previous UML for magical Items



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 they have the item in their inventory 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.

~~Override in player class the playTurn() to have a counter that runs when it detects invincible status.~~

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 getIntrinsicWeapon method will be overwritten to give the player enough damage to kill any enemy.~~ 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 dropltemAction 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.

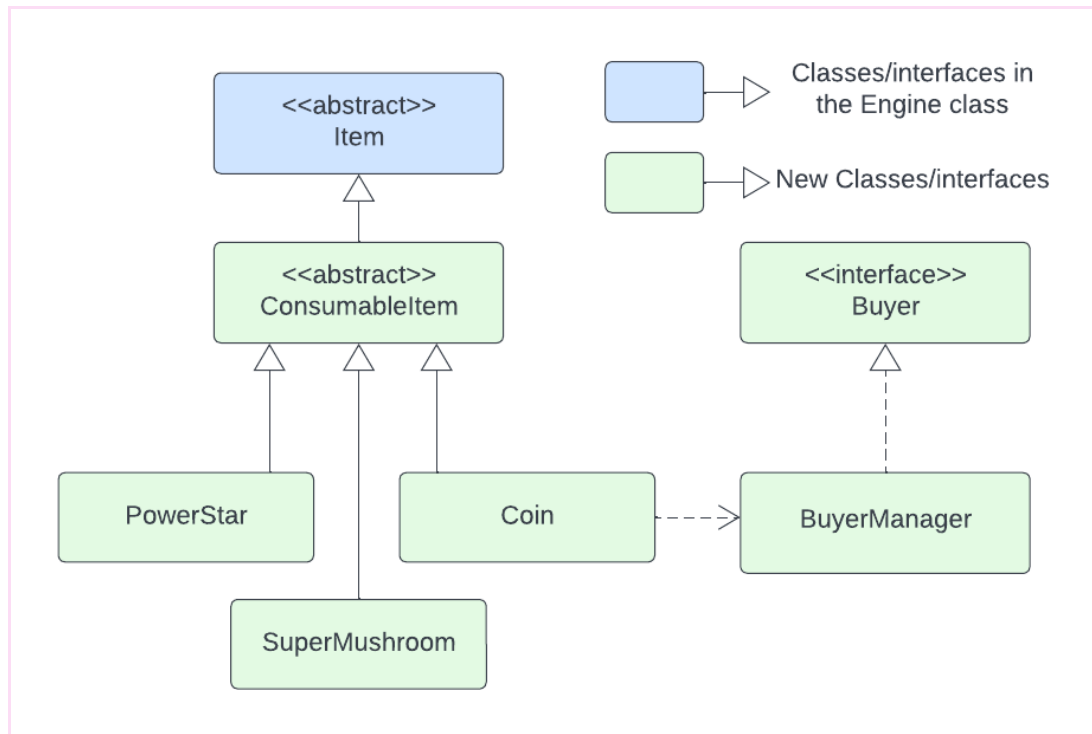
The player will be given an attribute "wallet" which will update with each coin picked up. The coin class will be responsible for the instance of each coin, it's worth and its position on the map.

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.

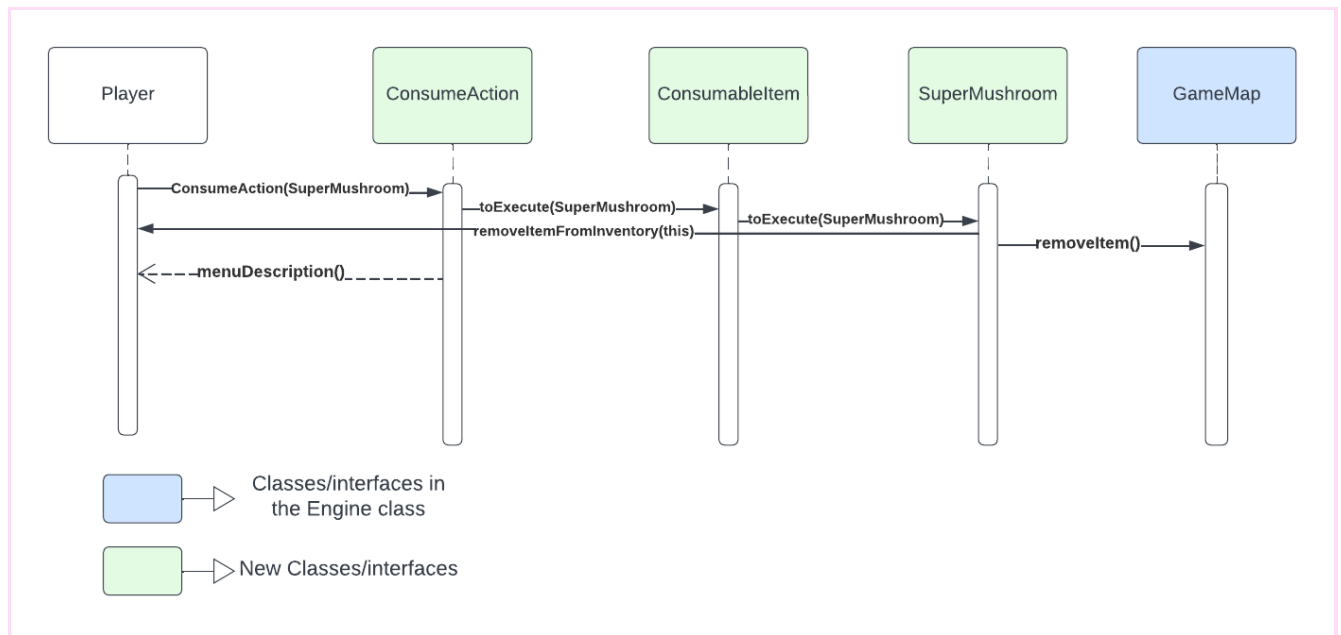


above: updated UML for coin class, depicting Coin dependency on BuyerManager

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

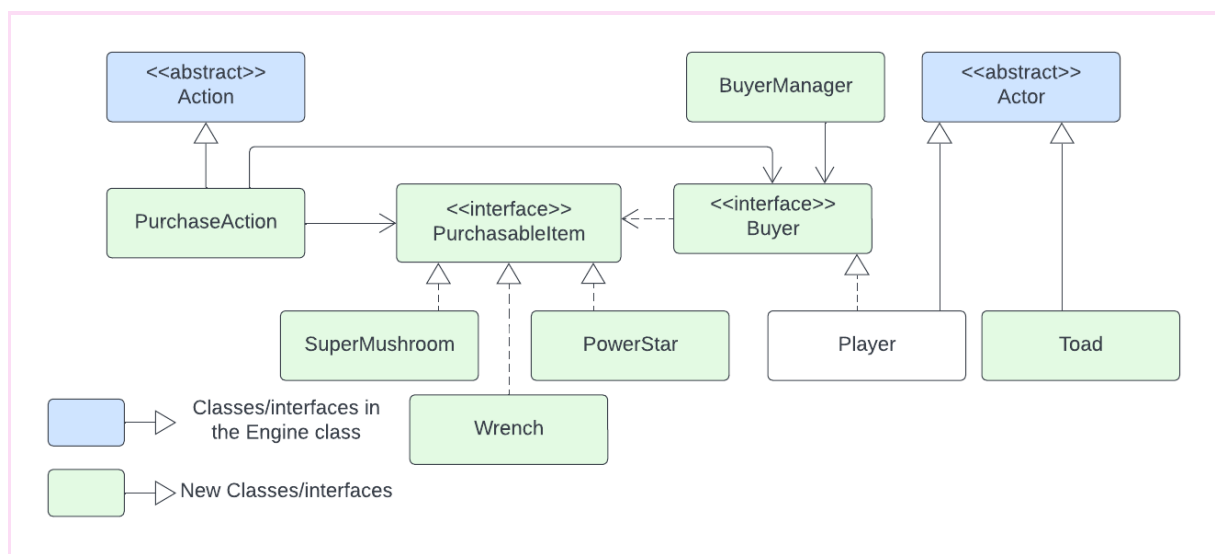
The ConsumableItem class allows items to inherit consumable properties and the toExecute method but removes the need for these items to have a dependency on the consumeAction. Previously it was designed to have an instance of the ConsumeAction in each consumable item class. This is poor design as it 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.

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.

PurchasableItem:

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 purchasableItem 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. 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.

Enemies Design Rationale

Key Decision: Enemy Abstract Class

With the addition of Koopa, we have decided to create an abstract class 'Enemy'.

Advantages	Disadvantages
<ul style="list-style-type: none">- 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	<ul style="list-style-type: none">- 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 AttackAction'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.

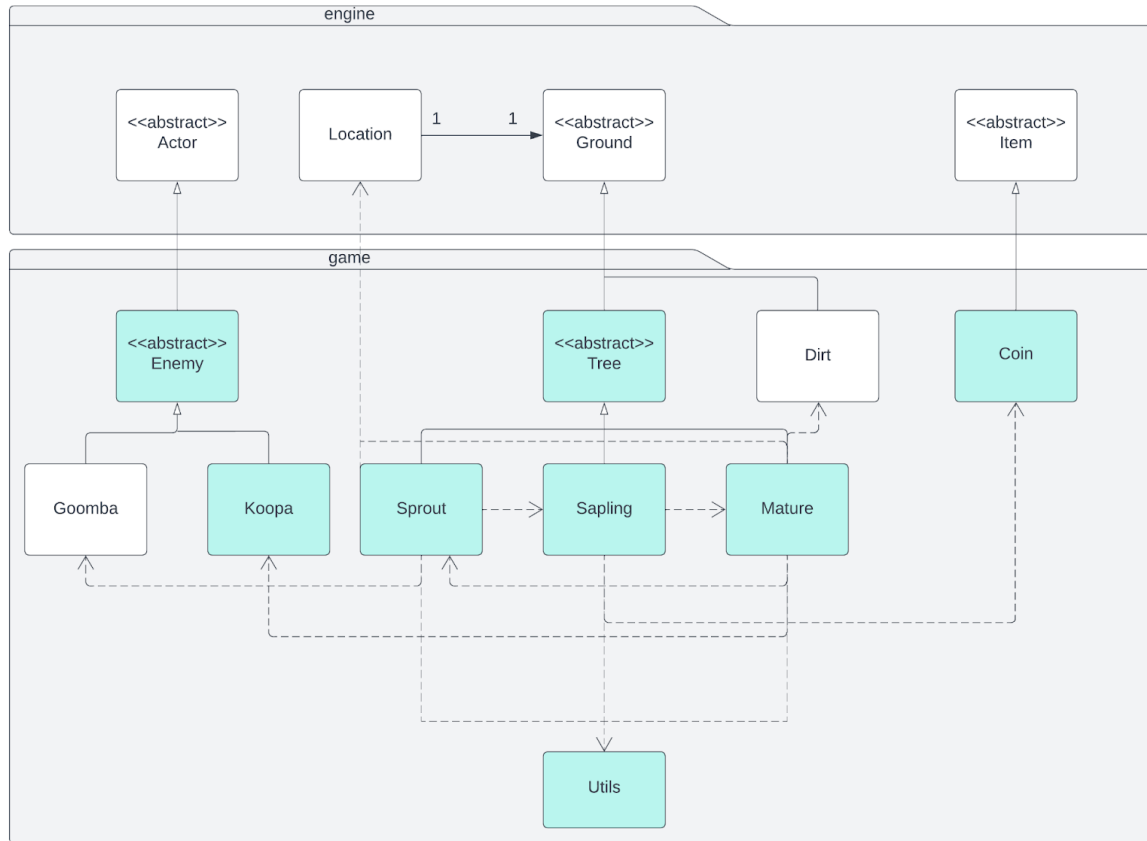
Advantages	Disadvantages
------------	---------------

<ul style="list-style-type: none">- 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 <i>any</i> actor as target in AttackAction without breaking the code.	<ul style="list-style-type: none">- 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** (dropltem) to drop the mushroom, rather than spawning one at a particular location.

Trees Design Rationale



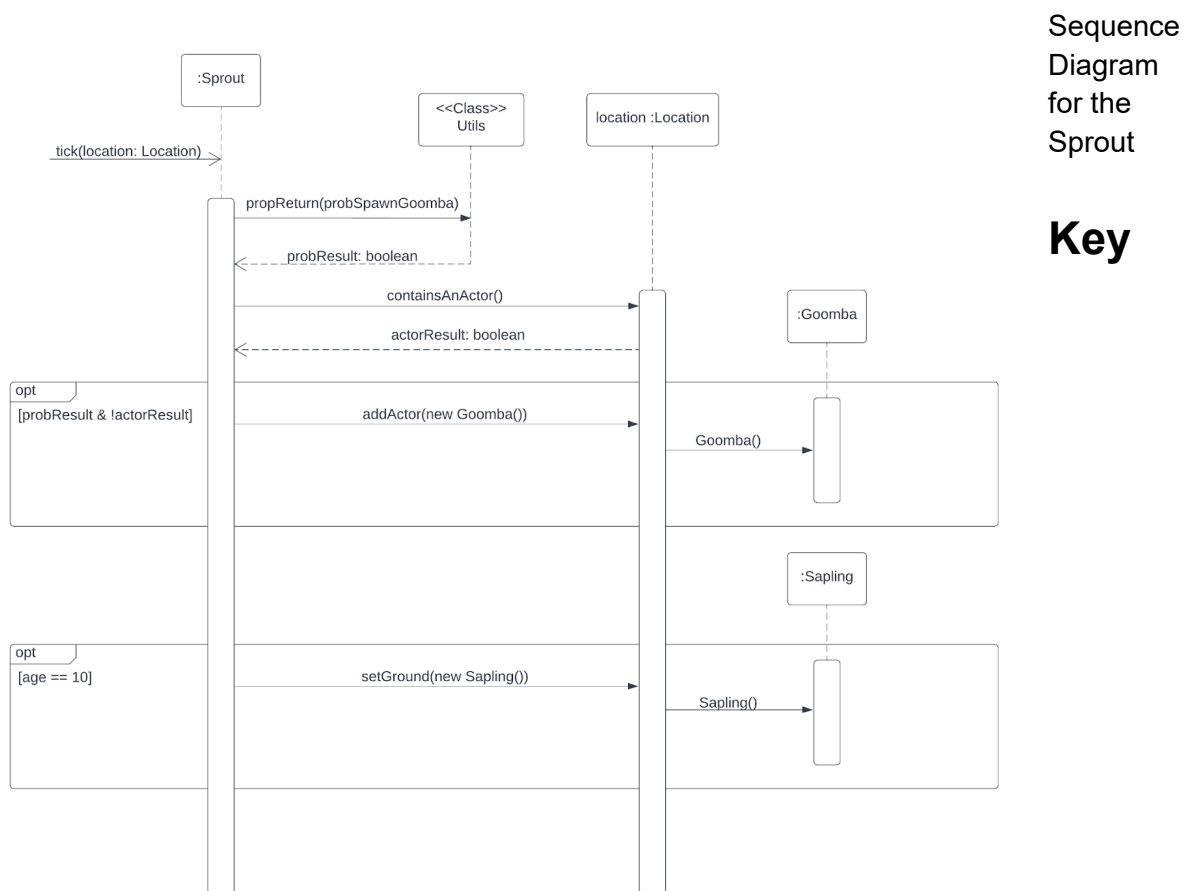
UML Class Diagram for Tree Features

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
<ul style="list-style-type: none"> - 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 	<ul style="list-style-type: none"> - The tree 'growing up' now requires the subclass to call <code>location.setGround()</code> 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

because the different tree maturities change for different reasons.	still followed, lowly Sprout depends on lofty Location.
---	---

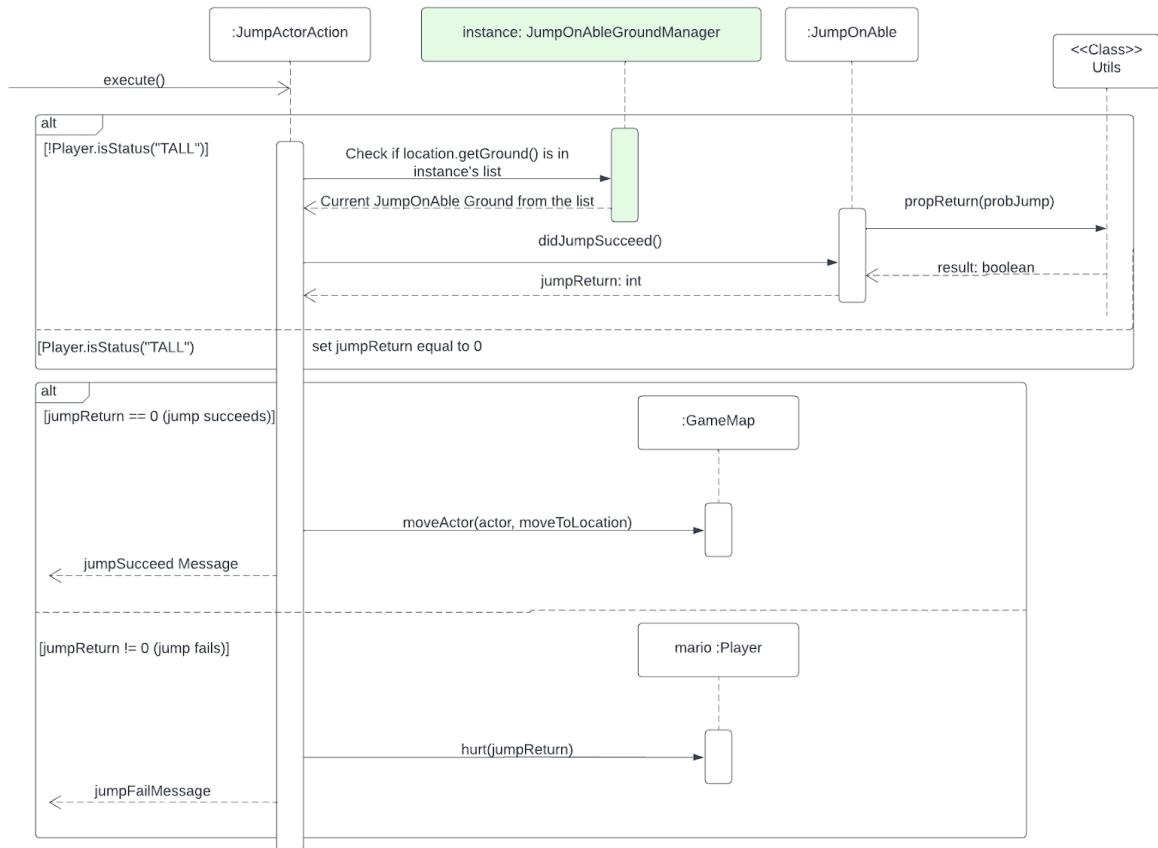


Decision: Utils Class

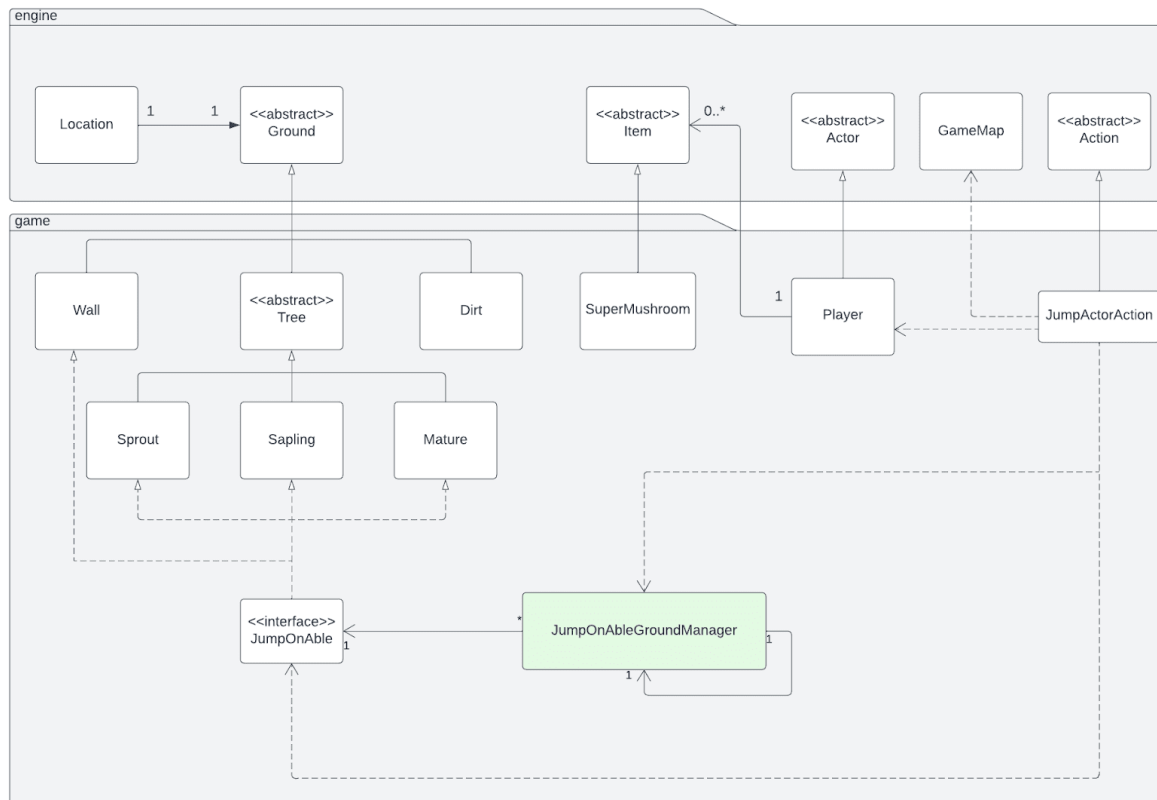
In this project many methods in many classes require probabilities. E.g. Sprout has a 10% chance of Spawning Goomba. To handle this, we have a Utils class with a static method probReturn(probability) that returns a Boolean based on a random number generator.

Advantages	Disadvantages
<ul style="list-style-type: none"> 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 	<ul style="list-style-type: none"> Reduce Dependencies: Many classes will now depend on this Utils class.

Jump Design Rationale



Sequence Diagram for the Jump Feature



UML Diagram for the Jump Feature

Key Decision: JumpOnAble Interface and JumpActorAction 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.

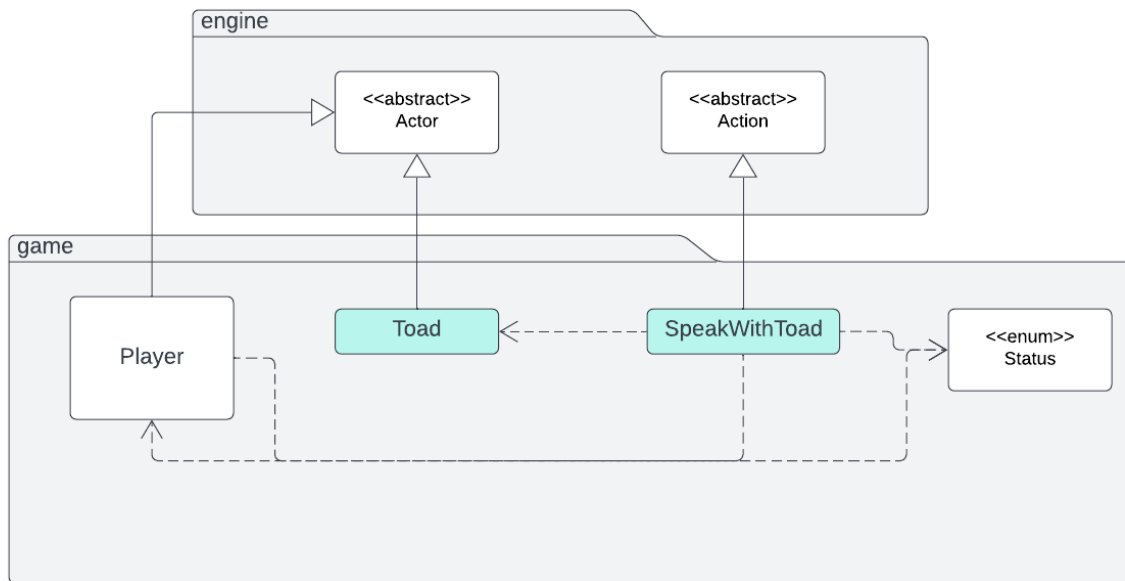
The JumpActorAction class will extend the action class. Its execute method will be responsible for performing the jump – deducting player HP if the jump fails and moving the player if the jump succeeds.

Advantages	Disadvantages
------------	---------------

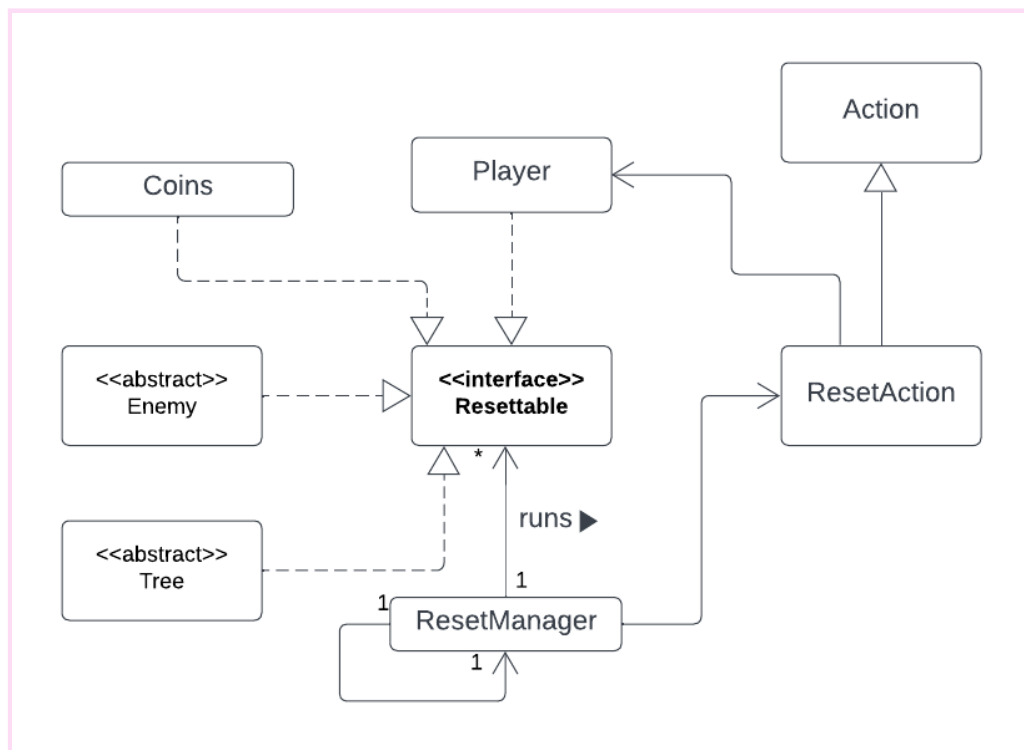
<ul style="list-style-type: none">- Open Closed Principle: Each ground type is responsible for its success rate and fall damage. This means that if we want to extend our system by adding more jump-on-able ground types, we don't need to change the Player or JumpActorAction code. The new ground type will 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.	<ul style="list-style-type: none">- Don't Repeat Yourself: This implementation means we have to add the JumpActorAction 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.
---	--

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.



Reset Design Rationale



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
<p>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.</p> <p>It also follows the Interface Segregation Principle: only classes that are affected on reset have a reset method, others do not.</p>	<p>The largest disadvantage to this method is the requirement for many classes to implement the resettable interface. This has the potential for repeated code.</p>