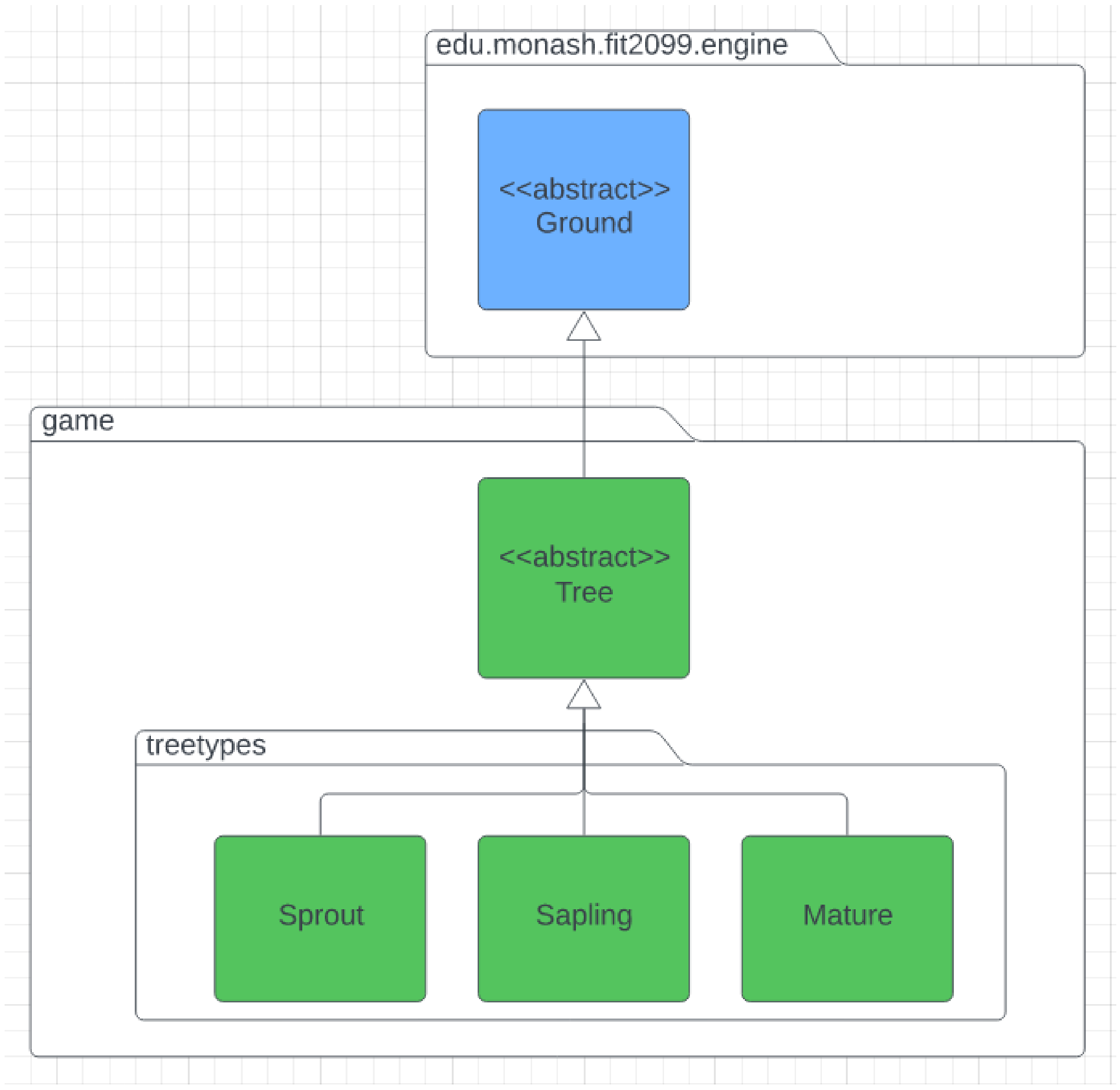
FIT2099 ASSIGNMENT 1

By Joel Atta, Jack Patton & Yepu Hou

Class diagram details:
Blue classes are classes that, as of starting that specific requirement, already existed in the project.
Conversely, green classes are classes that did not exist before starting that requirement and were added during it. Also, pre-existing classes that had their class header modified are green, such as when a non-abstract class that already existed is made abstract.
This means when a class is first created it will be in green, but if it is referenced in another UML diagram in a later requirement it will be in blue. Therefore, if you see a new class and don't understand its purpose, you can scroll up until you find the requirement in which that class is green, where it will be explained.

Req 1: Let it Grow!

Class Diagram:



Sequence Diagram:

Simple enough to not need a sequence diagram.

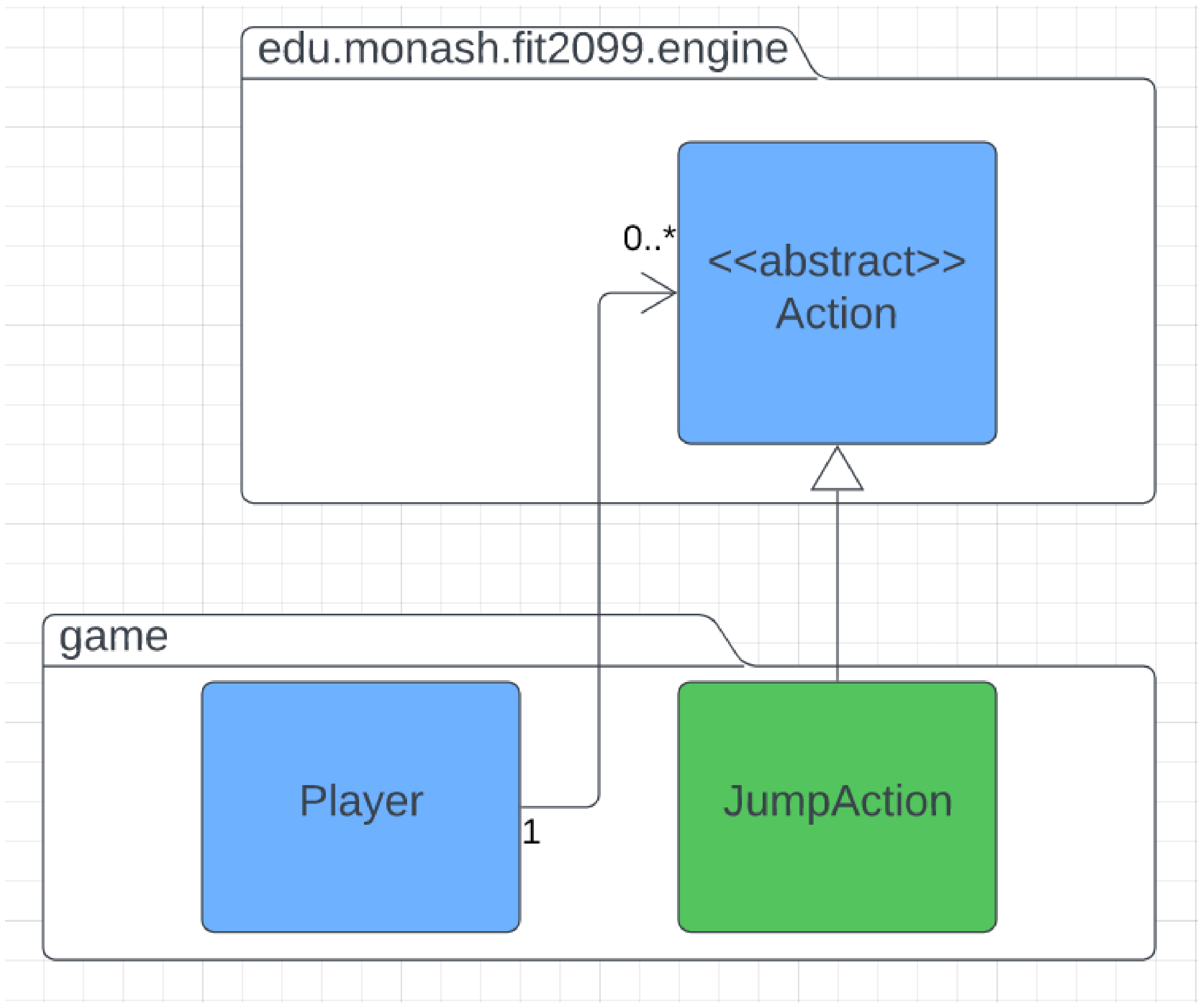
Design Rationale:

We could maintain a single Tree class and then choose which type of tree an instance of Tree is based on some internal counter tracking the turns since the tree was created. However, it would be better if there was an abstract Tree class and then 3 different classes representing each life stage of a tree that extended the main abstract tree class. This follows the already present methodology wherein Ground is extended by Dirt and Floor classes.
As Ground is an abstract class, and interfaces cannot extend an abstract class, Tree must also be an abstract class and not an interface.
For Req1, this means you can open the Sprout class and see that it has a 10% chance to spawn a goomba, while in the Sapling class the goomba spawning method is not present. In other words, code not relevant to that specific type of tree is not present in that specific type's class. This means we don't have to check the tree's current life stage to determine if it should, for example, spawn or goomba or spawn a coin.
These 3 classes follow the Liskov Substitution Principle, and there's an example of why based on requirement 2. Let's say Mario has a jump function that can be passed whatever object Mario is attempting to jump onto; it is legal for you to pass any of the classes derived from the abstract class Tree, which all have a different value for the success chance of jumping/damage taken on a failed jump.
Also, I found the 'game' package to be a bit cluttered and had issues finding what I was looking for quickly, so I moved all the tree types to their package within 'game' to aid in the clarity

Tree	Abstract class containing shared attributes and methods of all trees.
Sprout	Extends Tree, youngest tree stage.
Sapling	Extends Tree, middle tree stage.
Mature	Extends Tree, oldest tree stage.

Req 2: Jump Up, Super Star!

Class Diagram:



* Note that even though Player extends the Actor class I didn't show this dependency in the class diagram as Actor doesn't have any new direct dependencies, and this requirement is focused on the JumpAction class.

Sequence Diagram:

Simple enough to not need a sequence diagram. However, I will briefly describe the objects involved, to improve clarity.

Player stores an ArrayList of Actions that it can perform at any time. Relevant jump actions will be added based on the Player instances' position in the world, e.g. if Mario is next to a tree and a wall there will be two JumpActions in his available action list for each of the high grounds he can attempt a jump to.

However, in terms of the coding logic of how jumps would work with a super mushroom (100% jump success chance), the Player class would have a Boolean activeSuperMushroom tracking if the player is currently affected by a super mushroom and an int superMushroomTurnsLeft tracking how many turns are left on their active super mushroom. When Player calls JumpAction, it will pass it the value of activeSuperMushroom, which JumpAction will use to determine if it even has to calculate the success chance or can just skip straight to performing a successful jump. Again, this is not represented in the diagram yet as this isn't implemented until requirement 4.

Design Rationale:

Going up requires a jump, going down doesn't. Enemies cannot jump at all.

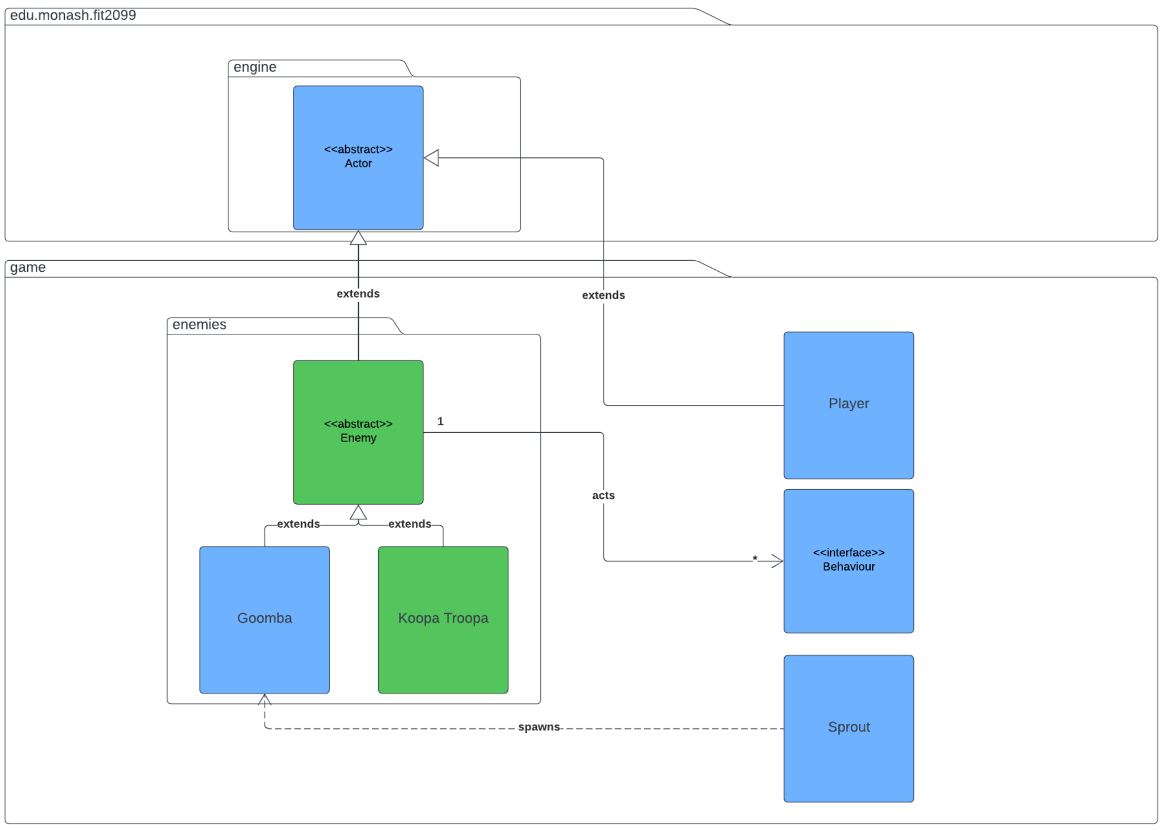
Therefore, we don't need to implement a jump feature for the enemies. The specifications state that enemies are intended to not be able to jump, so we don't need to worry about making it easy to add jump functionality to the enemies later down the line. (Not that it would be difficult to do, but it's good to keep in mind as it can make coding simpler if you don't need to worry about checking what actor is jumping).

In the same way that AttackAction is a class that extends Action, JumpAction is also a class that extends action. AttackBehaviour is only used by enemies to determine how they automatically attack, and as enemies can't jump we don't need to implement a JumpBehaviour class. Any "behaviour" for the jump can be kept in the JumpAction class itself.

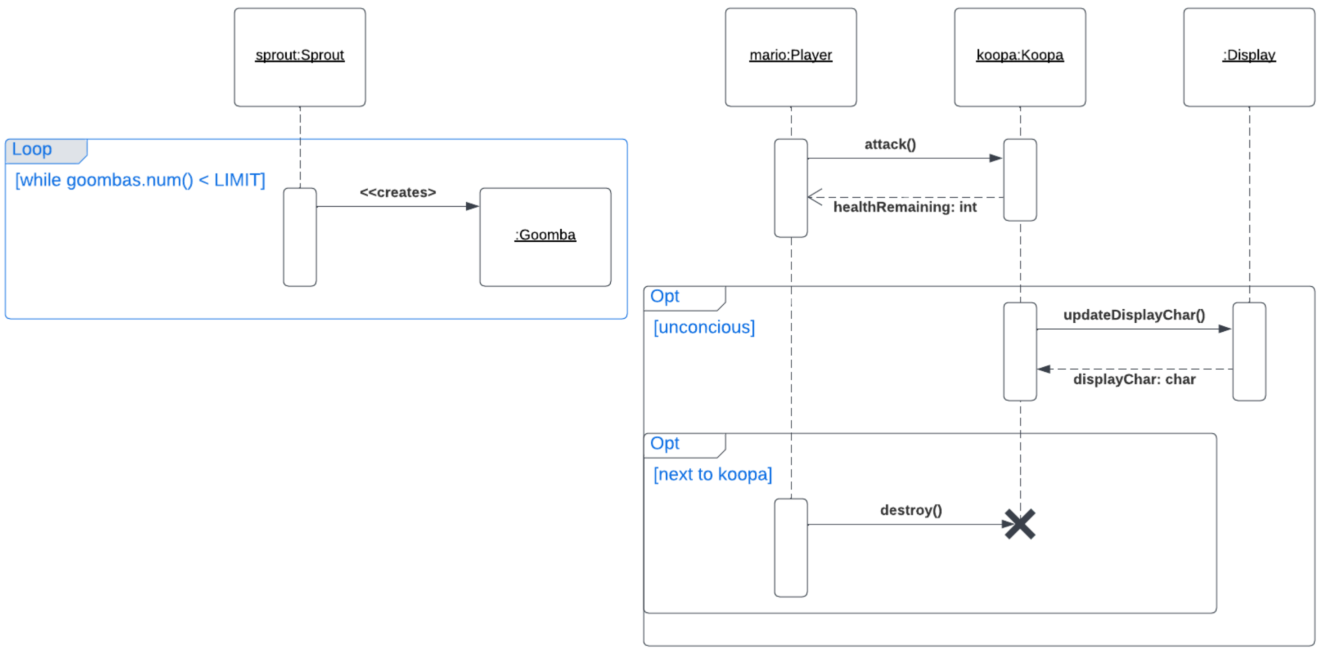
Action	Abstract class for actions an Actor can perform
Player	Adds an instance of JumpAction to its ArrayList of Actions for every jump the player can currently make.
JumpAction	Handles player jumping onto high ground action.

Req 3: Enemies

Class Diagram:



Sequence Diagram:



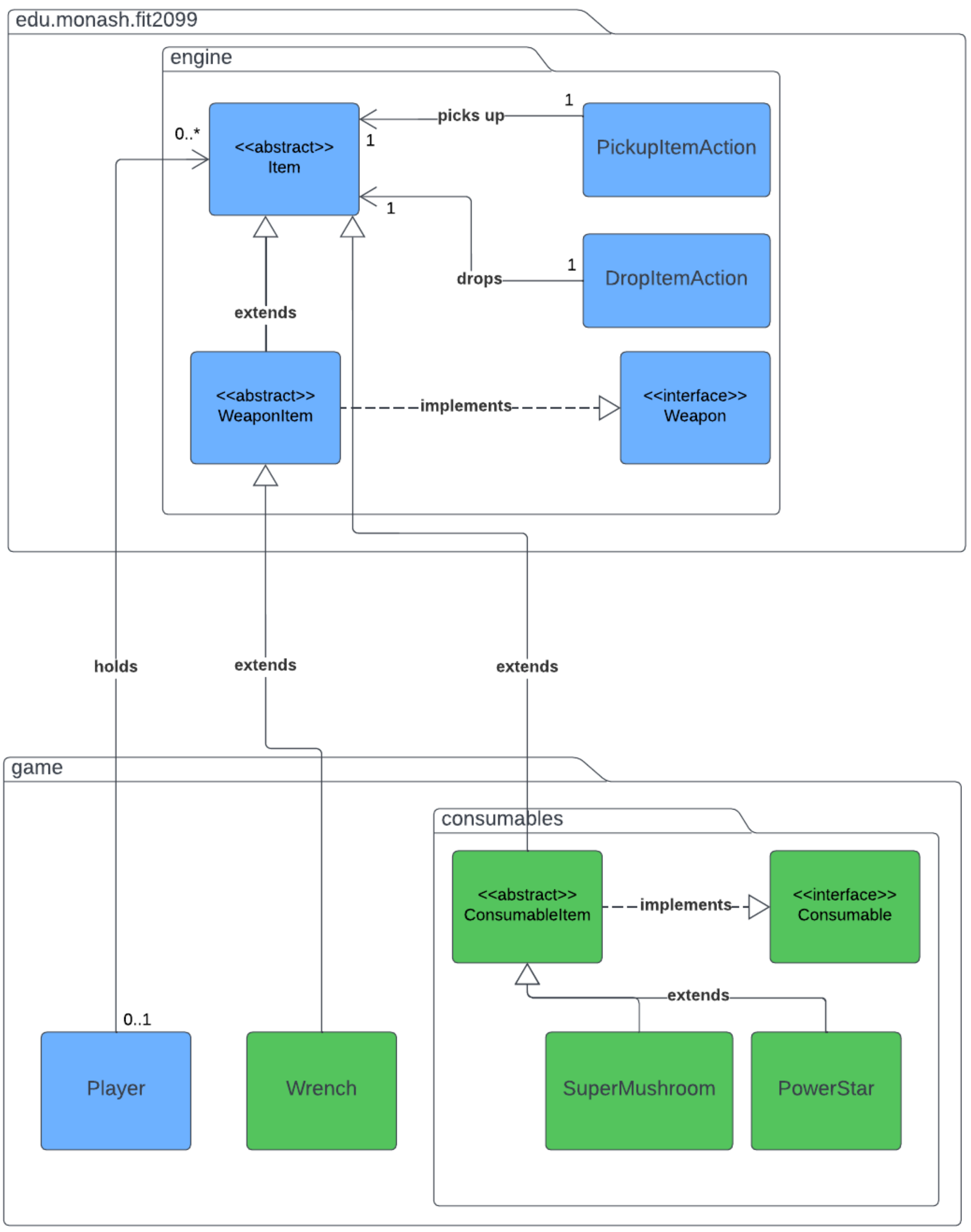
Design Rationale:

Enemy	Abstract class storing common attributes/methods to be extended by the many enemy types that may be in the game.
Koopa	A new enemy type within the Enemy abstract class. It will retreat to its shell when attacked by Mario and will only die if Mario hits it with a Wrench.

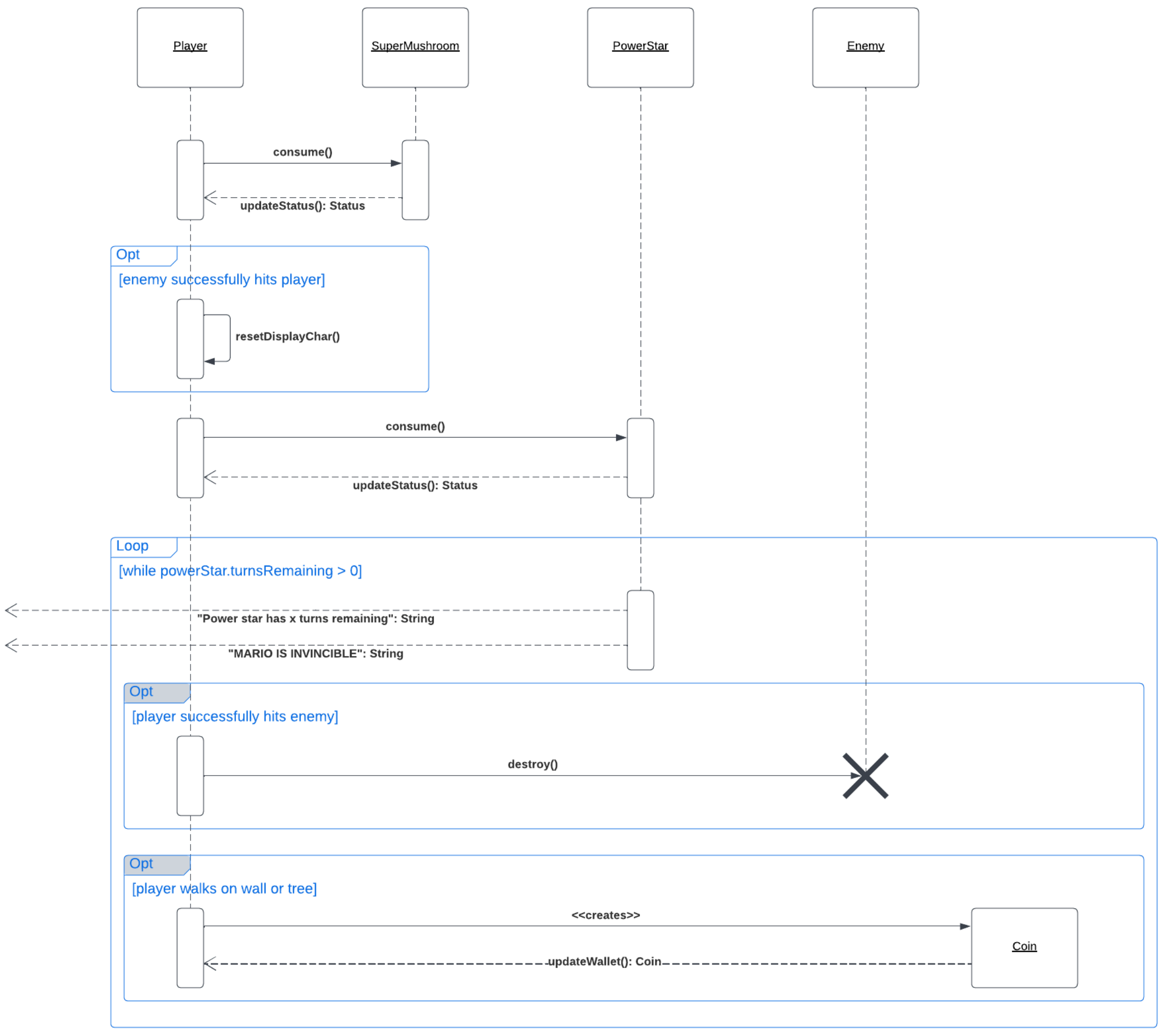
The starting point for my development of the class diagram for the Enemies requirement started by looking at the enemy that was already in the game, i.e. the Goomba. From this, I could see that the Goomba was inheriting from the *Actor* class within the engine package. As a result, I chose to make the new Koopa class also extend from the *Actor class* as the two enemies shared many attributes and methods. This is in adherence to the Open-closed principle which makes use of abstraction to ensure that the *Actor* class remains unchanged when adding a new enemy class. However, the Player class also extends from *Actor* and has a vastly different role, and along with that different behaviours from the enemies. To combat this I added an extra layer of abstraction with the *Enemy* class as a means of further separating the two types of *Actor*. Whilst in many cases multi-layered abstraction can make debugging more difficult, in a case where there are enough shared attributes/methods between two children of a node, it warrants another class. This will also make it flexible if we need to implement more enemies in A3. Additionally, I also added the *Enemy*, Koopa and Goomba classes to an 'enemy' package. The sequence is demonstrative of the implementation details for requirement 3, where some Sprouts spawn a Goomba up until a point, whilst also showing the interaction between Mario and the Koopa from its unconscious state up until its death. Note that occasionally I have simplified a group of actions to a single method call for the sake of simplicity such as the 'attack()' method, which in the main program would likely be a series of calls between classes.

Req 4: Magical Items

Class Diagram:



Sequence Diagram:


Player
SuperMushroom
PowerStar
Enemy
consume()
updateStatus(): Status
Opt
[enemy successfully hits player]
resetDisplayChar()
consume()
updateStatus(): Status
Loop
[while powerStar.turnsRemaining > 0]
"Power star has x turns remaining": String
"MARIO IS INVINCIBLE": String
Opt
[player successfully hits enemy]
destroy()
Opt
[player walks on wall or tree]
<<creates>>
Coin
updateWallet(): Coin

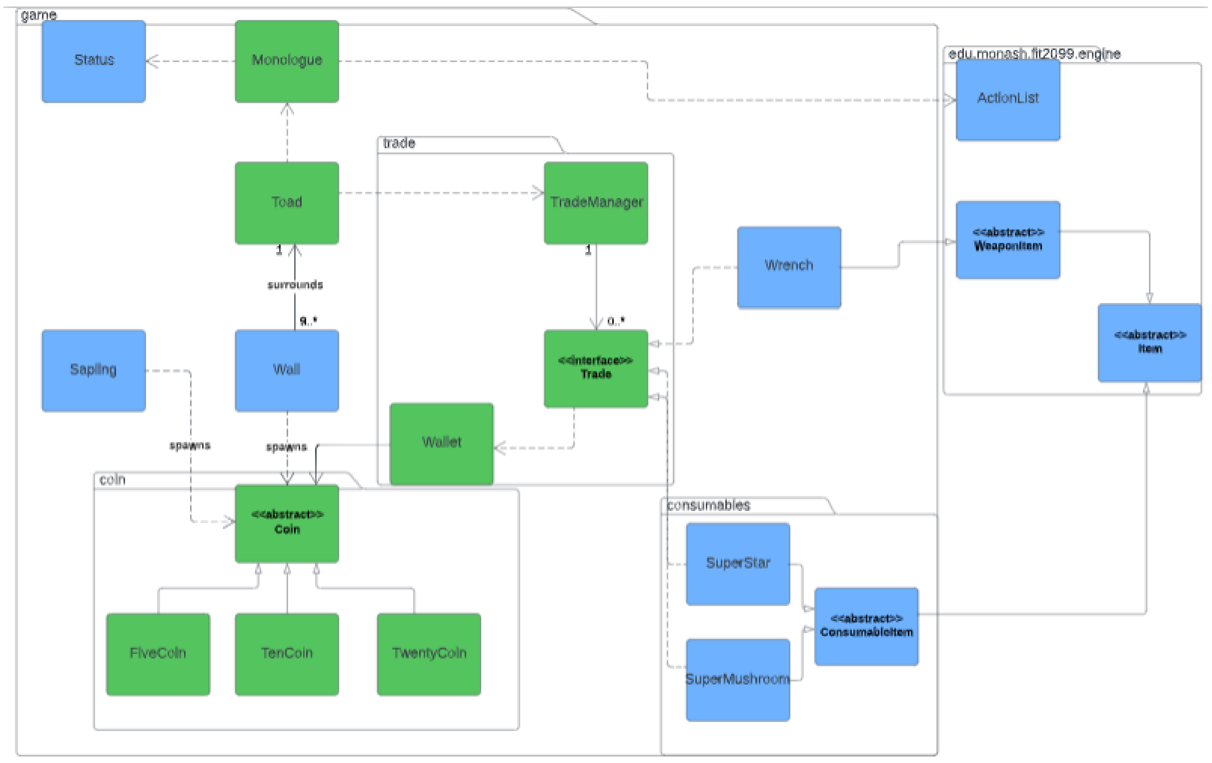
Design Rationale:

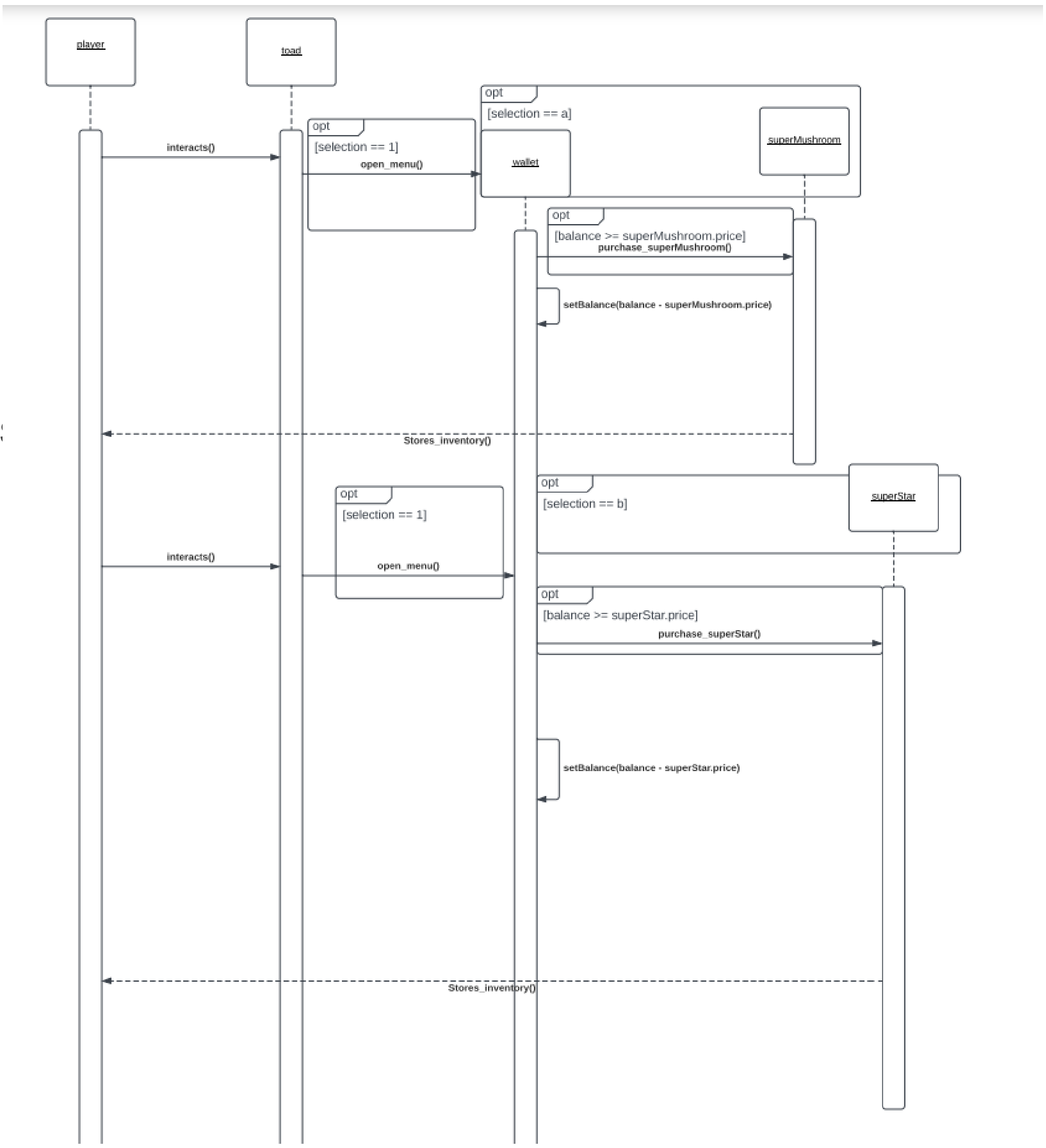
Consumable	A set of methods implemented by consumable items.
ConsumableItem	An Item that can be consumed (abstract).
Wrench	Used to break Koopa's shell (execute).
PowerStar	On consumption, makes Mario invincible, spawns coins on walls/trees and insta-kills enemies when hit. Resets after a set time.
SuperMushroom	On consumption, increases Mario's maximum health and changes his display char to be a capital 'M'. Resets when hit.

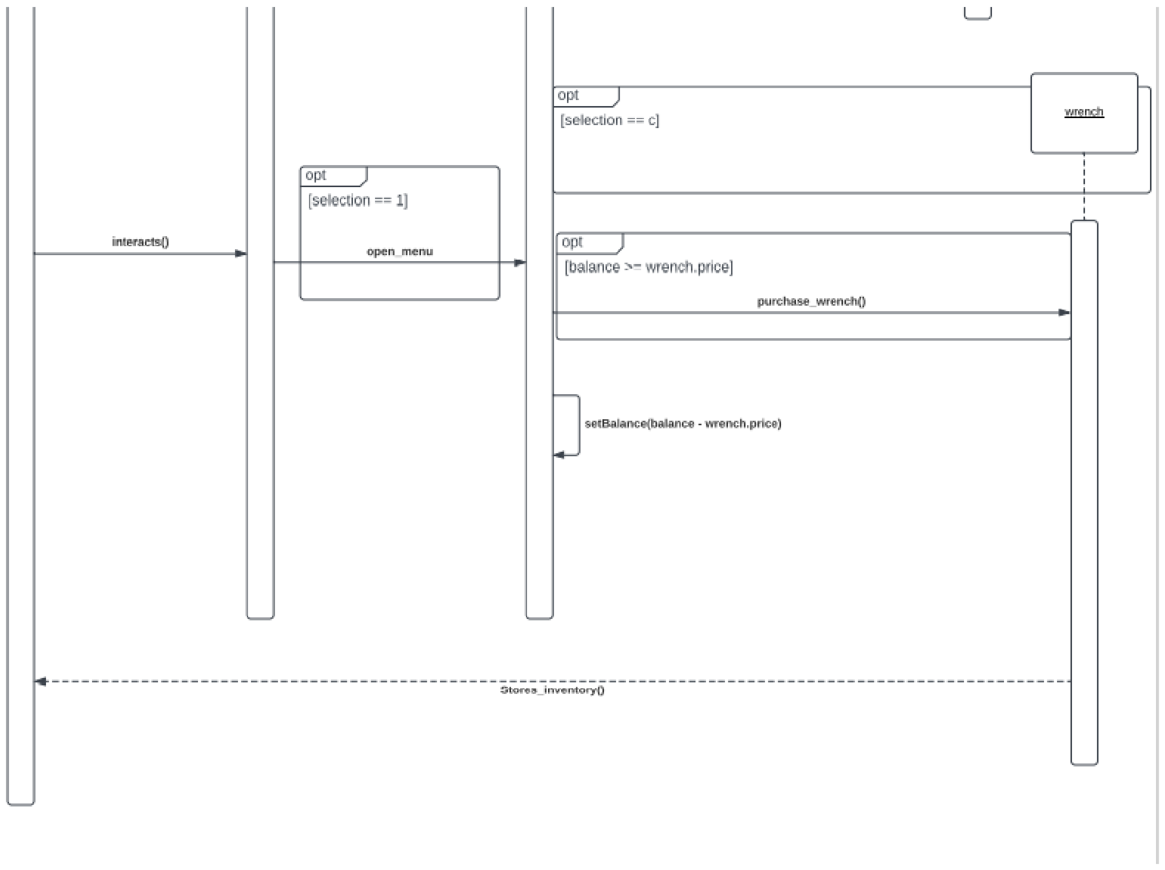
Initially, I started the class diagram for requirement 4 with the three items: Wrench, PowerStar and SuperMushroom all extending from the *Item* class. I quickly realised however that this is a bad design as it does not adhere to the Single Responsibility Principle. The items can be further subdivided into Weapons and Consumables. After I saw that within the engine there was a *WeaponItem* class that extended *Item* and implemented a Weapon interface, I sought to mirror this design by creating a *ConsumableItem* class that extends *Item* and implements *Consumable*. This gives a lot of flexibility when adding Items in future and adheres to the SRP, ISP and DIP. The sequence diagram shows the sequence of implementation details in requirement 4. Note that the updateStatus() method after the Player consumes() a *ConsumableItem* will change the Status enum for a player, which will then be checked in a tick and the corresponding actions/changes to attributes will take place. Additionally, the messages leading to no entity are Strings printed to the console.

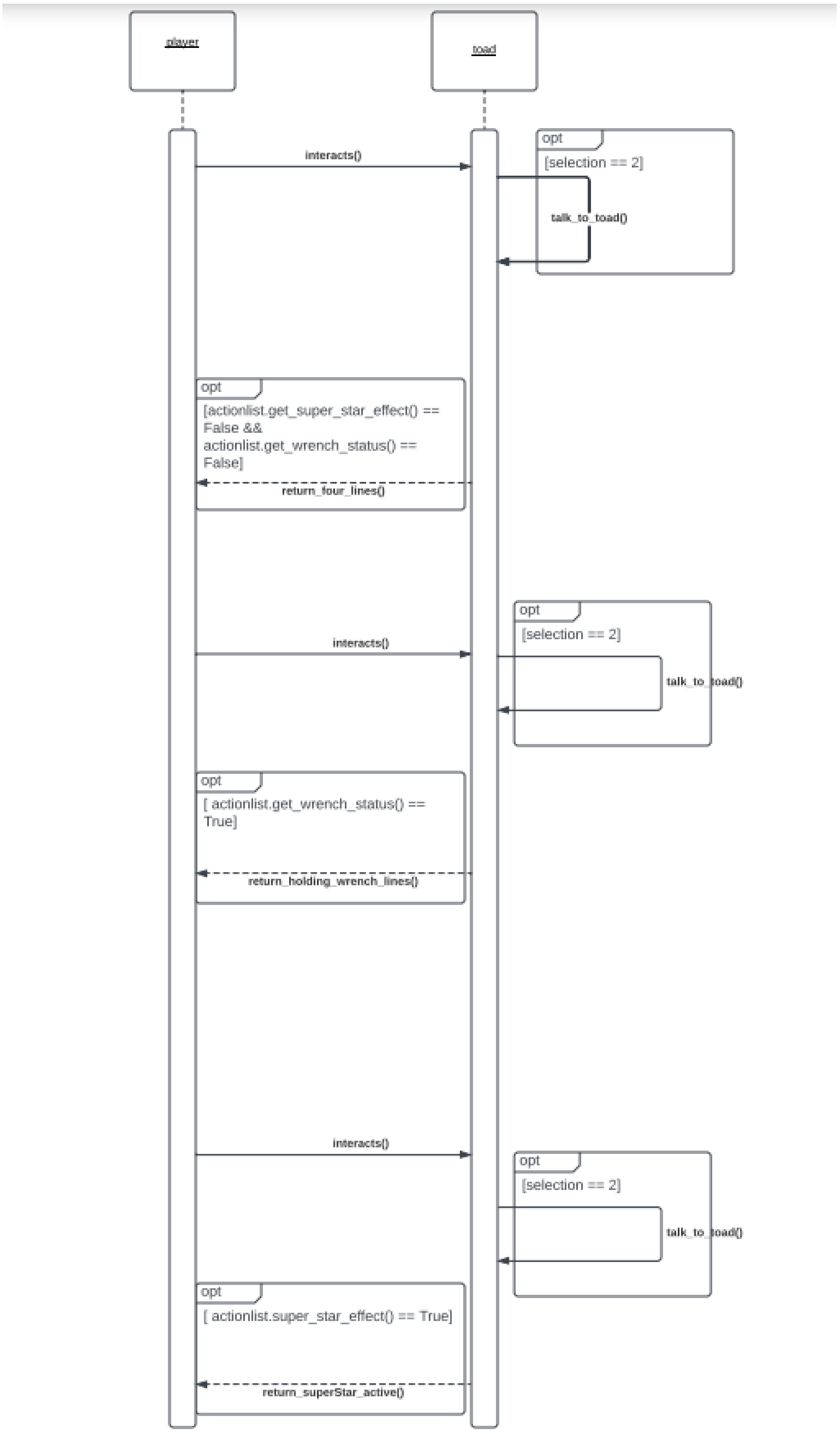
Req 5: Trading & Req 6: Monologue

Class Diagram:




player
toad
opt
[selection == a]
superMushroom
opt
[selection == 1]
interacts()
open_menu()
wallet
opt
[balance >= superMushroom.price]
purchase_superMushroom()
setBalance(balance - superMushroom.price)
Stores_inventory()
opt
[selection == 1]
opt
[selection == b]
superStar
interacts()
open_menu()
opt
[balance >= superStar.price]
purchase_superStar()
setBalance(balance - superStar.price)
Stores_inventory()


opt
[selection == c]
wrench
opt
[selection == 1]
interacts()
open_menu
opt
[balance >= wrench.price]
purchase_wrench()
setBalance(balance - wrench.price)
Stores_inventory()


player
toad
interacts()
opt
[selection == 2]
talk_to_toad()
opt
[actionlist.get_super_star_effect() == False && actionlist.get_wrench_status() == False]
return_four_lines()
interacts()
opt
[selection == 2]
talk_to_toad()
opt
[actionlist.get_wrench_status() == True]
return_holding_wrench_lines()
interacts()
opt
[selection == 2]
talk_to_toad()
opt
[actionlist.super_star_effect() == True]
return_superStar_active()

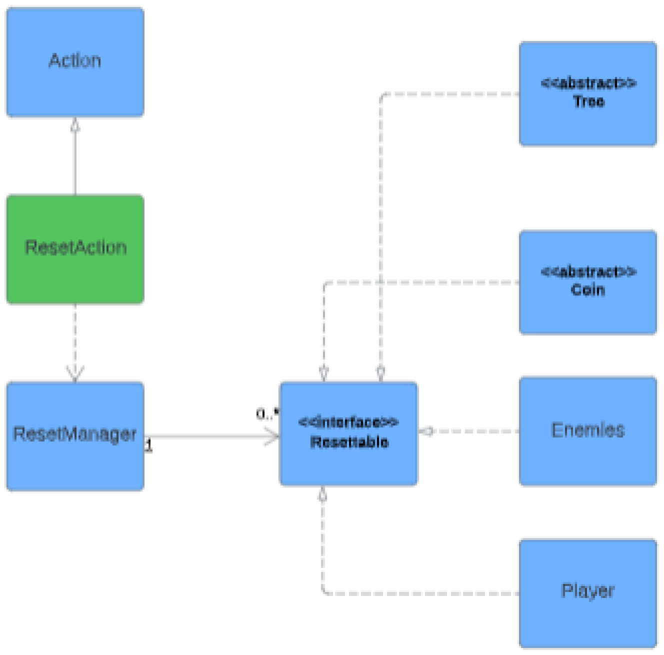
Design Rationale:

1. Toad - A character to trade and talk to
2. Monologue - Lines to say to the character
3. TradeManager - Manages all the trade
4. Trade (interface) - Tradeable items
5. Wallet - Wallet system to trade between coin and item
6. Coin (abstract) - Coins to store different types of coins
7. FiveCoin - Different types of coins
8. TenCoin - Different types of coins
9. TwentyCoin - Different types of coins

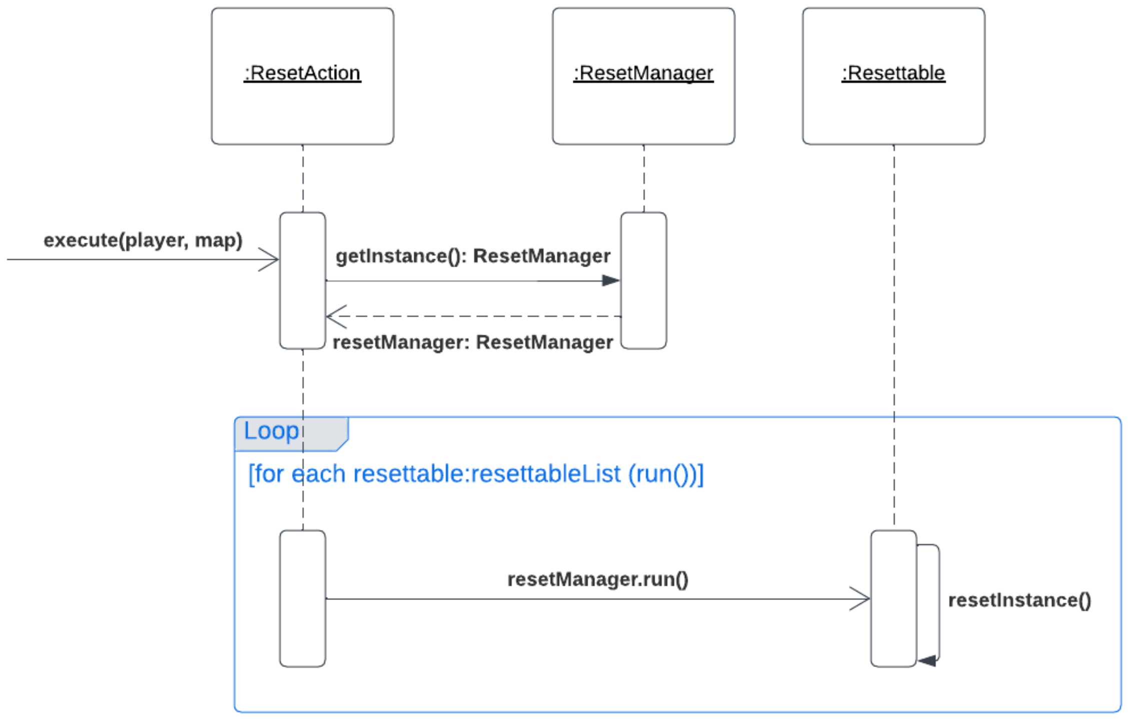
Although we can just create a wallet class and do an if-else statement to do the trading, by the rule of single responsibility each class needs to have an independent purpose. Toad is a class that has a dependency with both Monologue and TradeManager. Monologue has a dependency with ActionList because the Monologue is dependent on whether an actor has a wrench or SuperMushroom effect. The TradeManager manages each trade that has happened and Trade will be created as an interface for which Wrench, SuperStar, and SuperMushroom can be traded using Coins. Coins are randomly spawned from Sapling and break the Wall. The Coin itself is an abstract class that has FiveCoin, TenCoin, and TwentyCoin objects extended from the abstract Coin class. A Wallet stores an integer value walletValue which represents the total of all the Coins in the wallet, which is used by Trade.

Req 7: Reset Game

Class Diagram:



Sequence Diagram:


:ResetAction
:ResetManager
:Resettable
execute(player, map)
getInstance(): ResetManager
resetManager: ResetManager
Loop
[for each resettable:resettableList (run())]
resetManager.run()
resetInstance()

Design Rationale:

Class ResetAction extends Action, adding a new action available to the player at any time which will reset most aspects of the game. Tree, Coin, Enemies, and Player all implement the Resettable interface, and all have their specific resets. When resetInstance is called, all Enemies and Coin(specifically, the coins that extend the abstract Coin) instances are completely deleted. Note that they are deleted and not killed, as we don't want them to proc any on death effects such as dropping coins. When resetInstance is called on Tree's subclasses, trees are deleted and replaced with dirt 50% of the time on a per-instance basis. Finally, when resetInstance is called on the Player, their current power-up status is reset.

While each of the four classes (Tree, Coin, Enemies, and Player) implement their resetInstance method and could theoretically be called independently e.g. resetting Enemies but none of the other classes, this won't occur as the ResetManager, and by extension, the ResetAction will call resetInstance on all instances of all four classes.