

# FIT2099 ASSIGNMENT 1

By Joel Atta, Jack Patton & Yepu Hao

## **Class diagram details:**

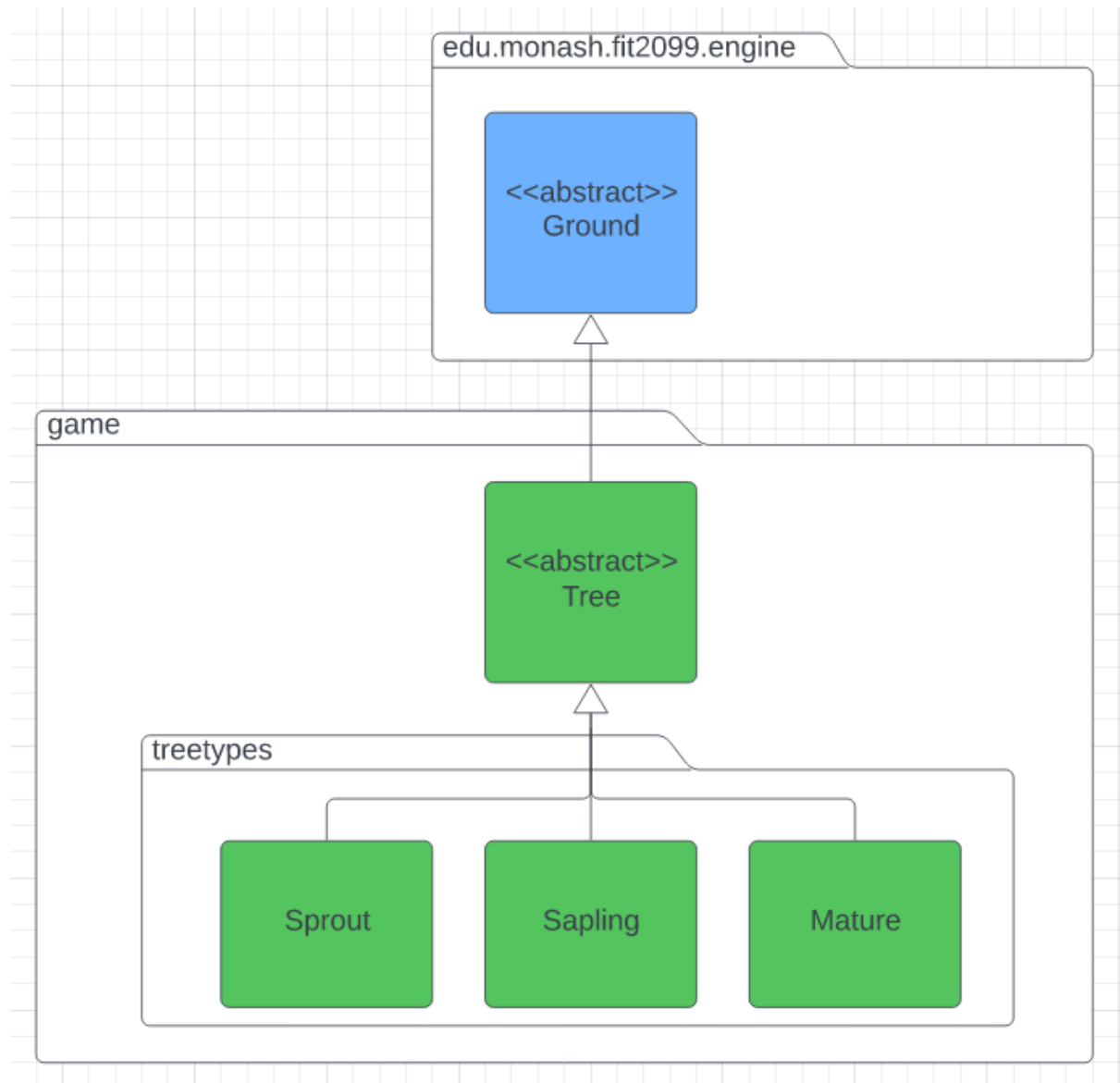
Blue classes are classes that, as of starting that specific requirement, already existed in the project.

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:

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 lifestage 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.

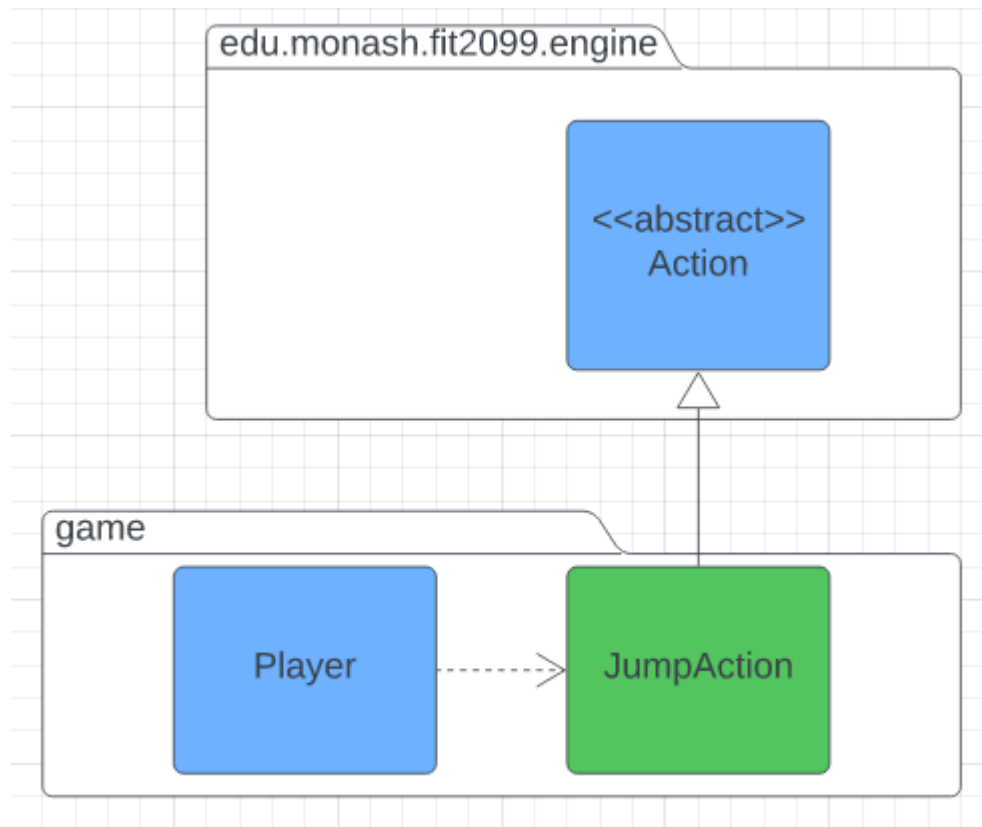
For Req1, this means you can open the Sprout class and see that it has a 10% chance to spawn a goomba, while in the Sapling class the goomba spawning method is not present. In other words, code not relevant to that specific type of tree is not present in that specific type's class. This means we don't have to check the tree's current lifestage in order 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 own package within 'game' to aid in clarity

## Req 2:

### 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, in order to improve clarity.

Player simply calls JumpAction which reads the contents of the tile that the jump is being made to and handles the success/failure logic.

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:

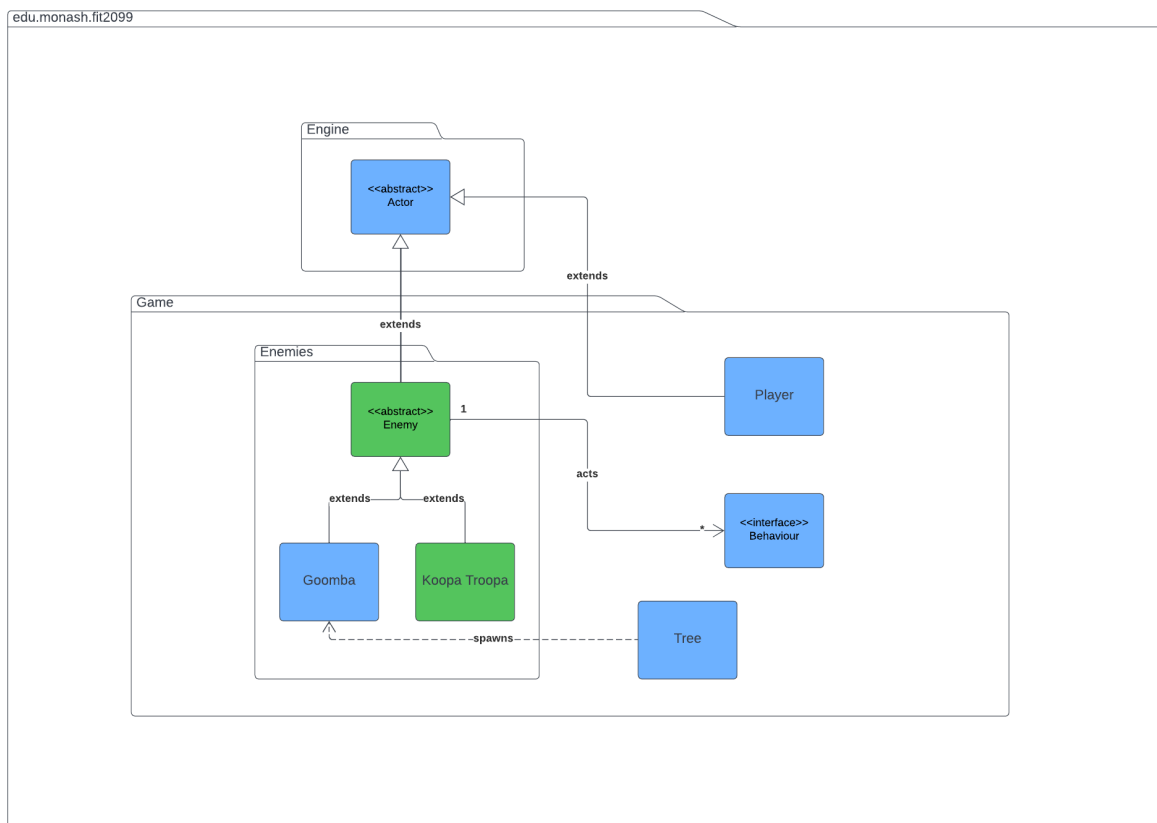
**Going up requires a jump, going down doesn't. Enemies cannot jump at all.**

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 its good to keep in mind as it can make coding simpler if you don't need to worry about checking what actor is jumping).

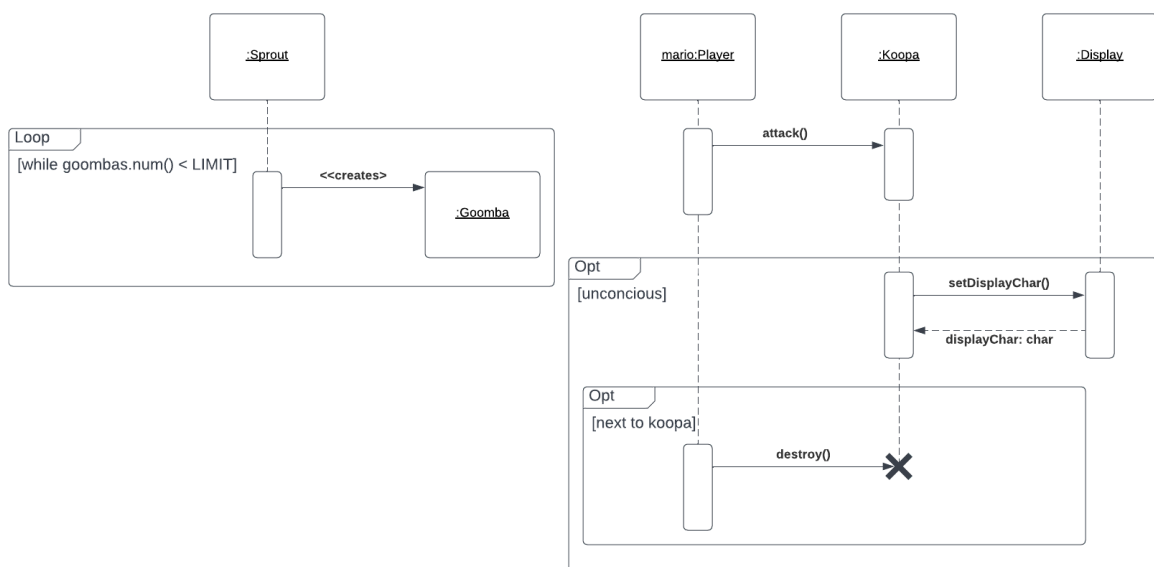
In the same way that AttackAction is a class that extends Action, JumpAction is also a class that extends action. AttackBehaviour is only used by enemies to determine how they automatically attack, and as enemies can't jump we don't need to implement a JumpBehaviour class. Any "behaviour" for the jump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

# Req 3: Enemies

## Class Diagram:



## Sequence Diagram:

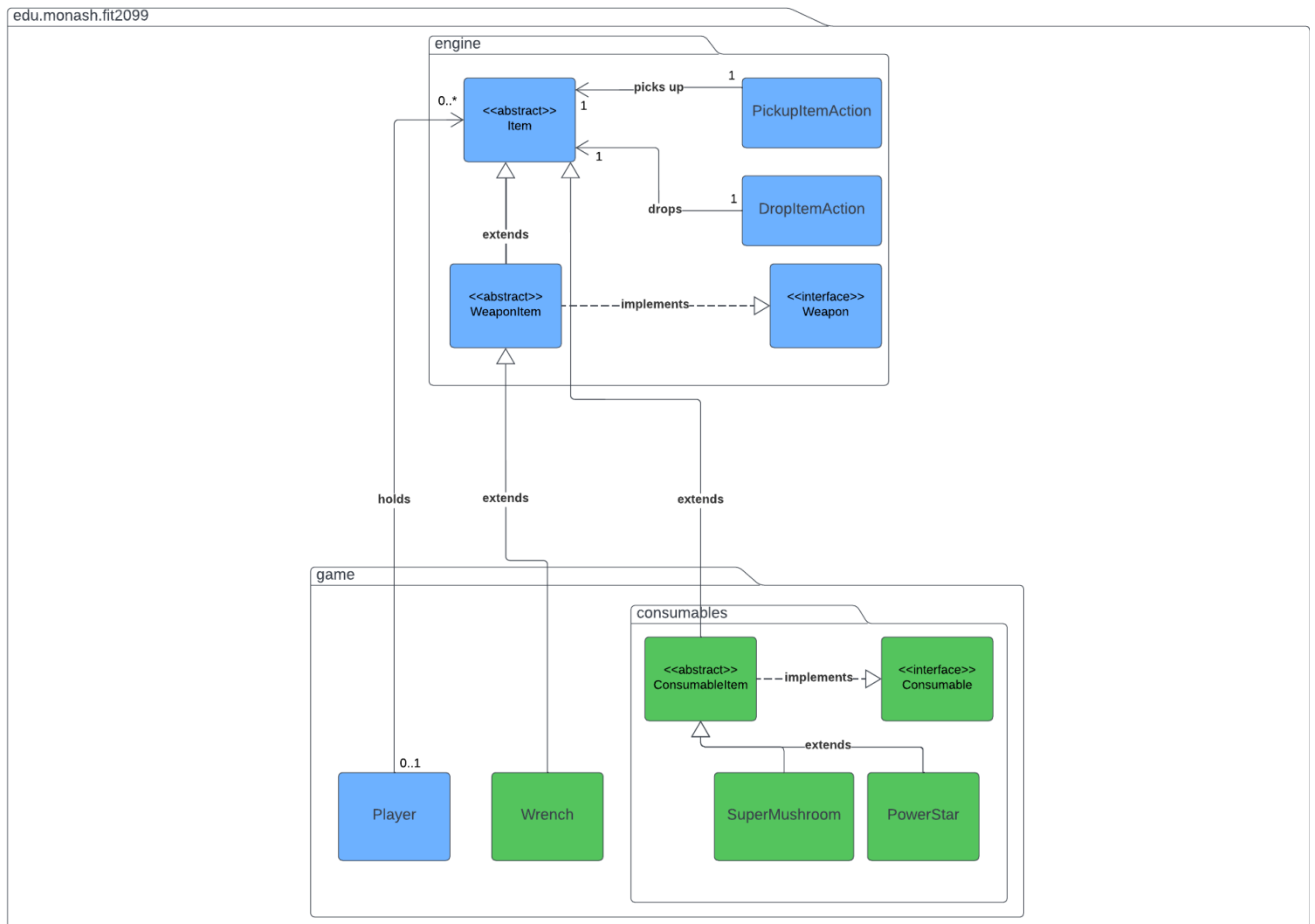


## Design Rationale:

Within the program, the other enemy type (Goomba) already inherits from the Actor class. Therefore, it only makes sense that the other enemy within the game also inherits from the Actor class as it shares many of the same attributes with the other enemy. It is possible to create an Enemy abstract class that extends from the Actor class for further abstraction, but generally multi-layer abstract classes are not the best idea and so we chose to avoid it in this scenario. We may however implement an interface that enforces some of the common behaviours that the enemy classes share as a means to differentiate them from the player. This can also be achieved by placing them within the same package.

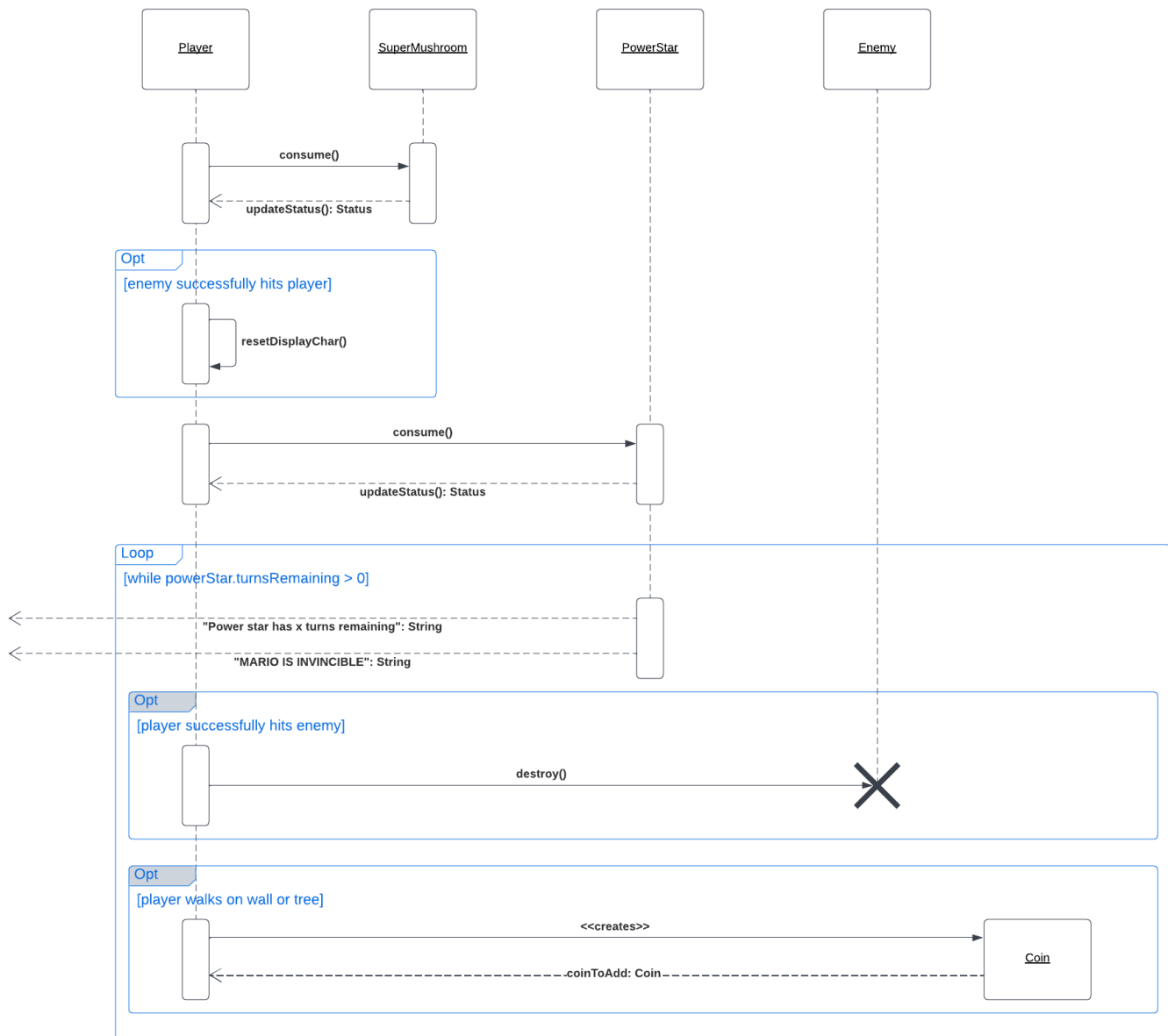
# Req 4: Magical Items

## Class Diagram:





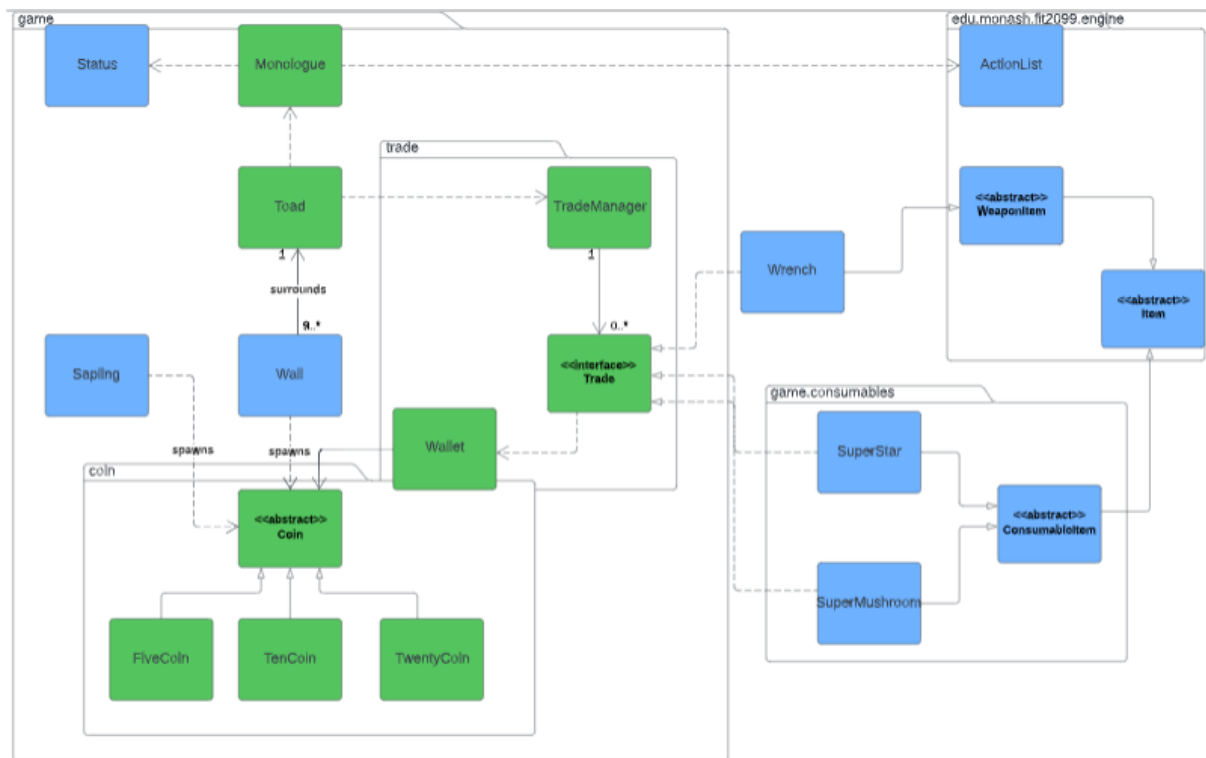
## Sequence Diagram:



Design Rationale:

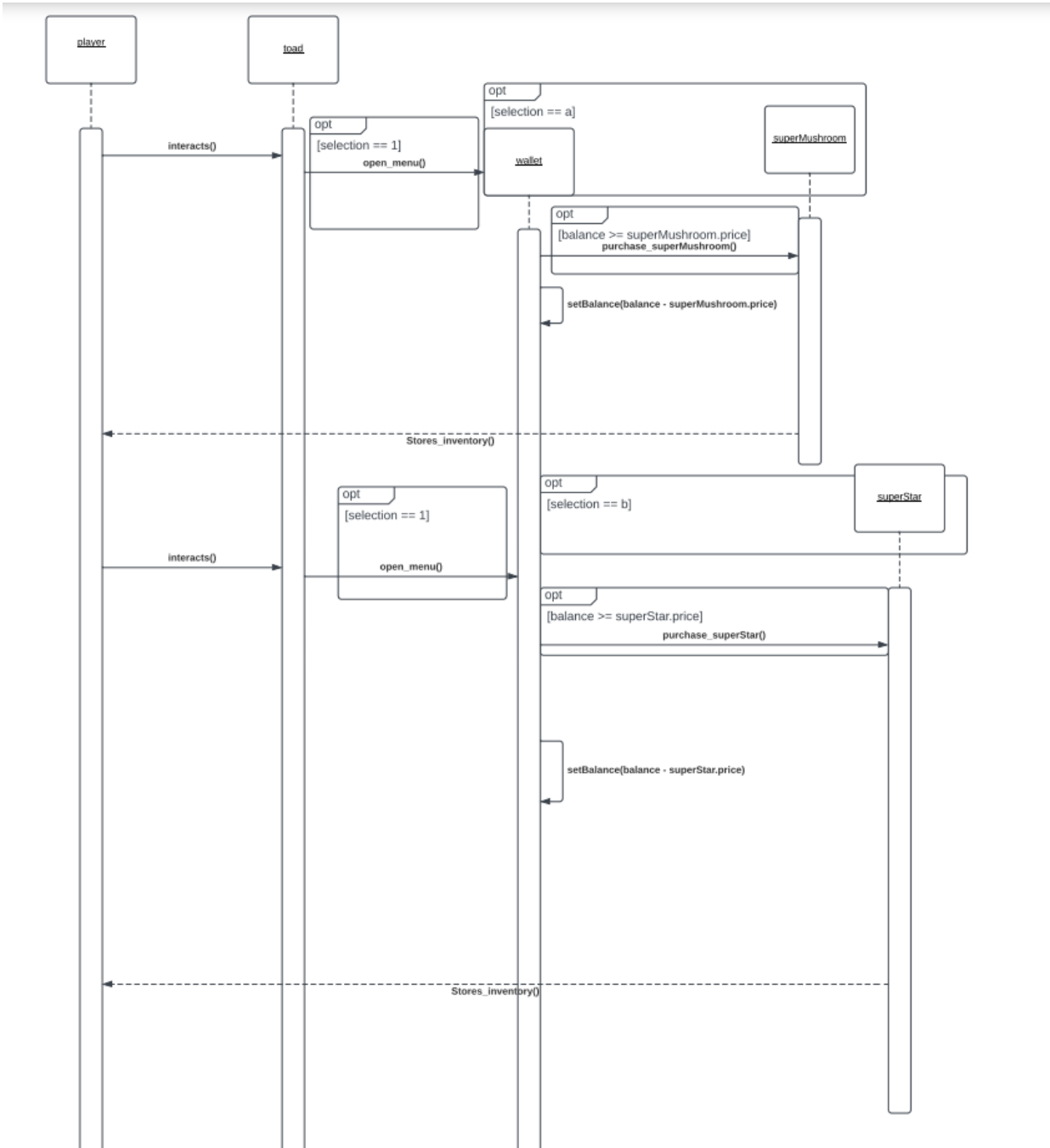
Req 5 + Re6:

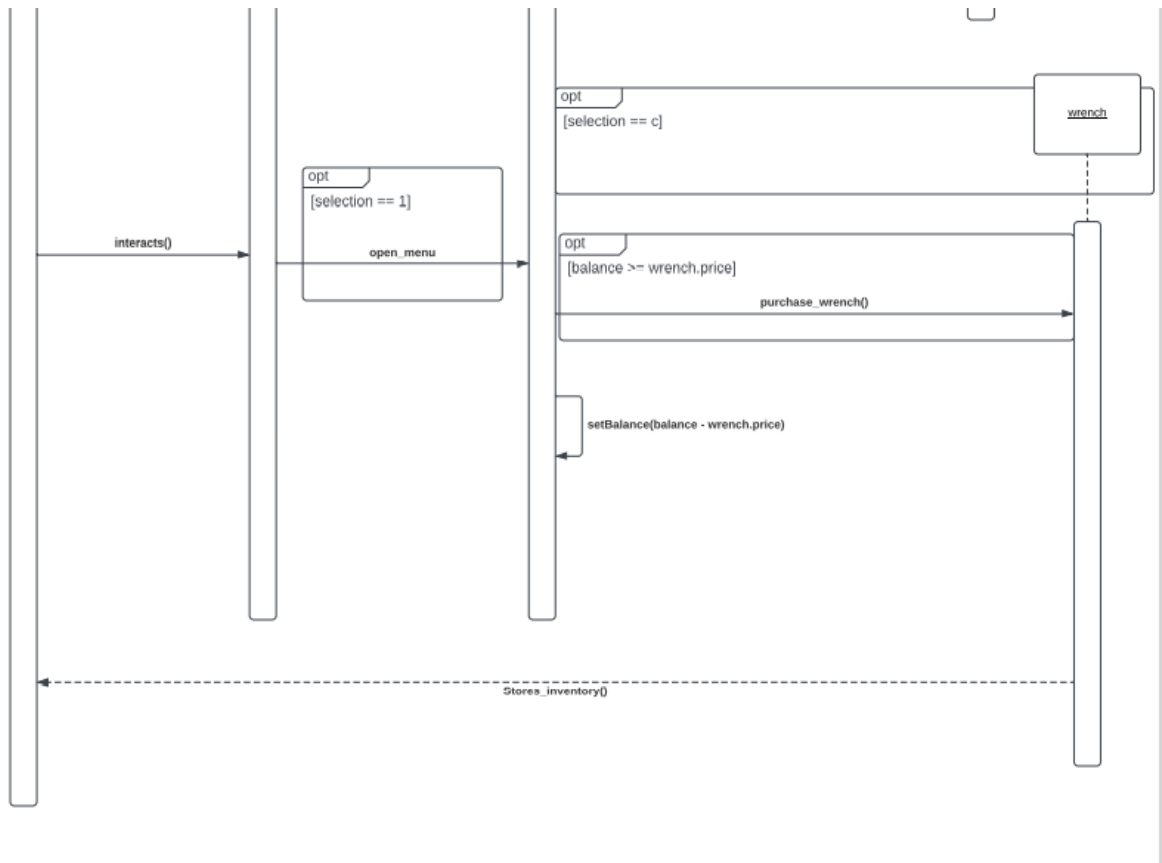
## Class Diagram:



## Sequence Diagram (re5):

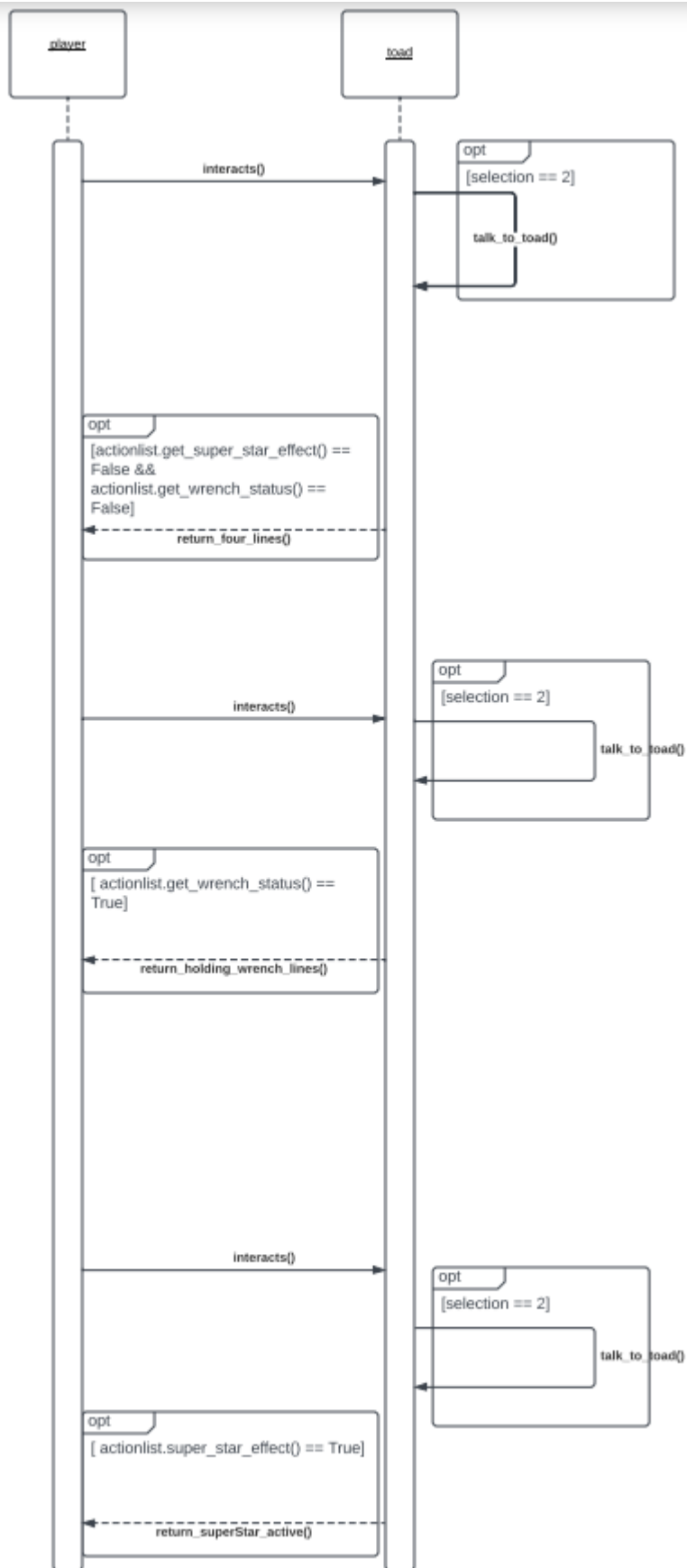






Sequence diagram (re6)





## Design Rationale:

### Classes that are newly created:

1. Toad
2. Monologue
3. TradeManager
4. Trade (interface)
5. Wallet
6. Coin (abstract)
7. FiveCoin
8. TenCoin
9. TwentyCoin

Although we can just create a wallet class and do an if else statement to do the trading, but by the rule of single responsibility each class needs to have each independent purpose.

Toad is a class which has dependency with both Monologue and TradeManager.

Monologue has dependency with ActionList, because the Monologue is depending on whether an actor has a wrench or supermushroom effect.

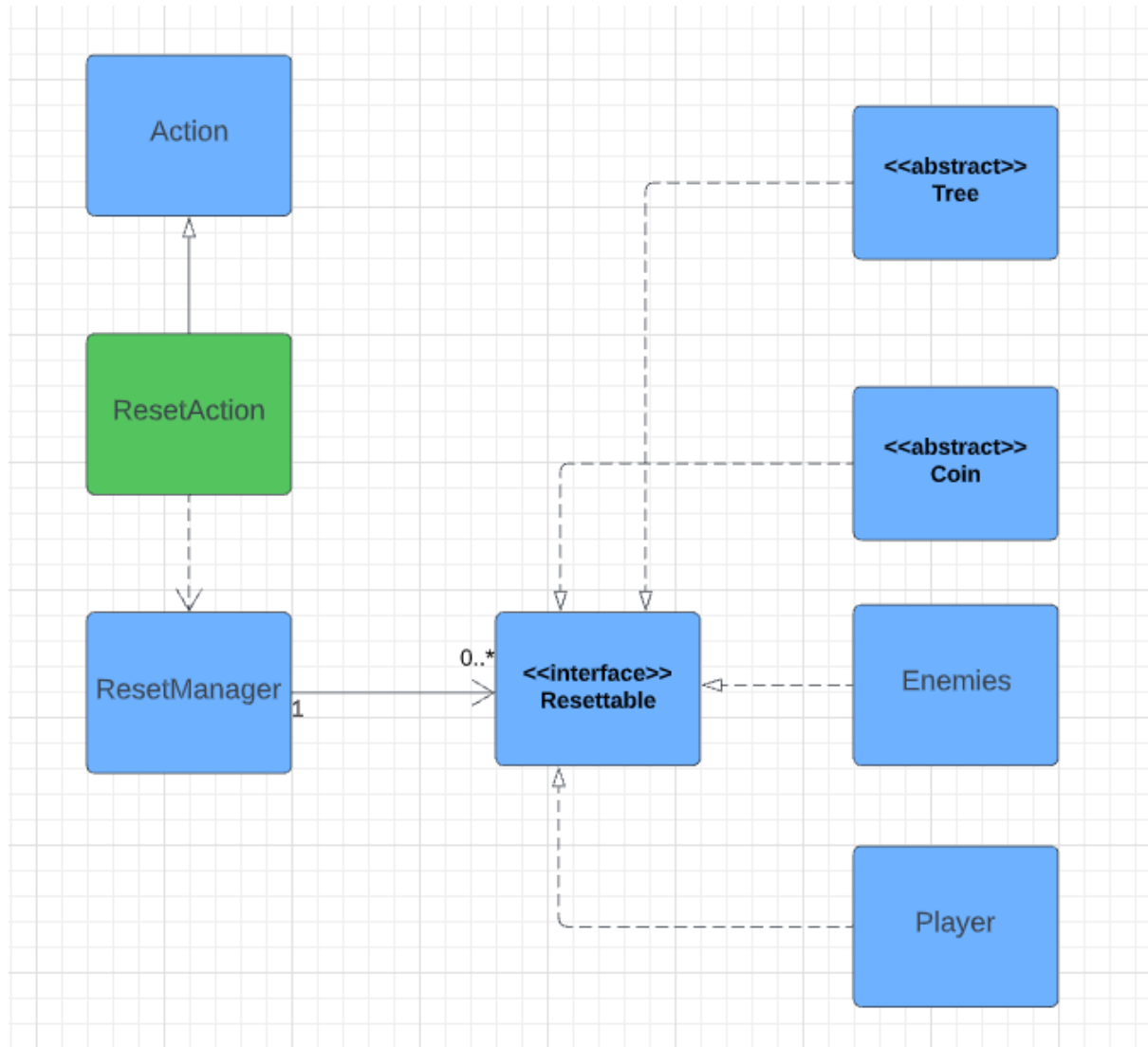
The TradeManager manages each trade that has happened and Trade will be created as an interface for which Wrench, SuperStar, and SuperMushroom can be traded using Coins.

Coins are randomly spawned from Sapling and breaking the wall. The Coin it self is an abstract class which has FiveCoin, TenCoin, and TwentyCoin objects stored within the abstract Coin class.

A Wallet stores Coin as an integer and is used by Trade.

## Req 7:

### Class Diagram:



### Sequence Diagram:

### 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 own 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 own `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.