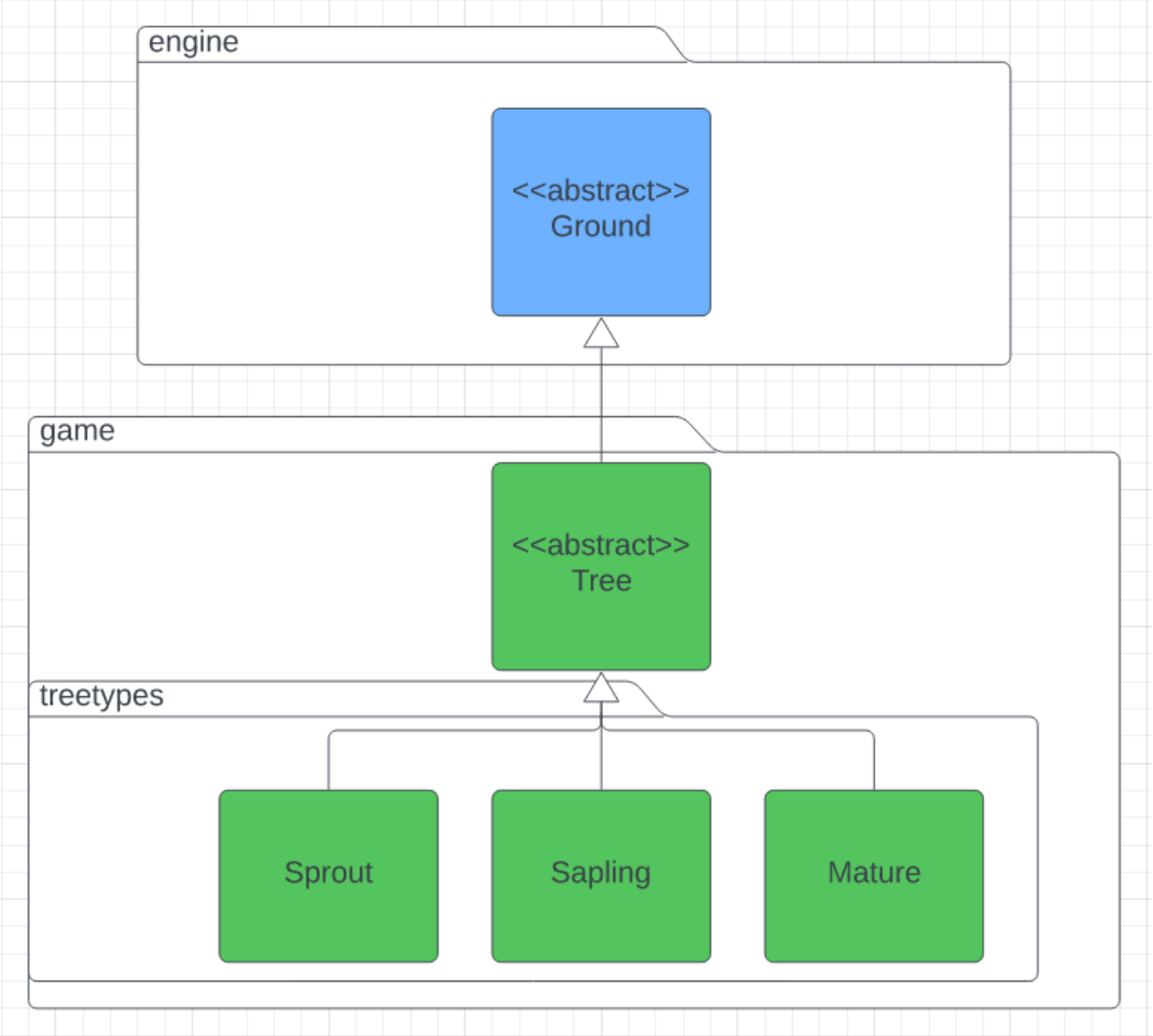
FIT2099 ASSIGNMENT 1

By Joel Atta, Jack Patton & Yepu Hao

Req 1:

Class Diagram:



Blue = pre-existing

Green = new/modified

Note that there is already a class named Tree, but it has been made abstract.

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.

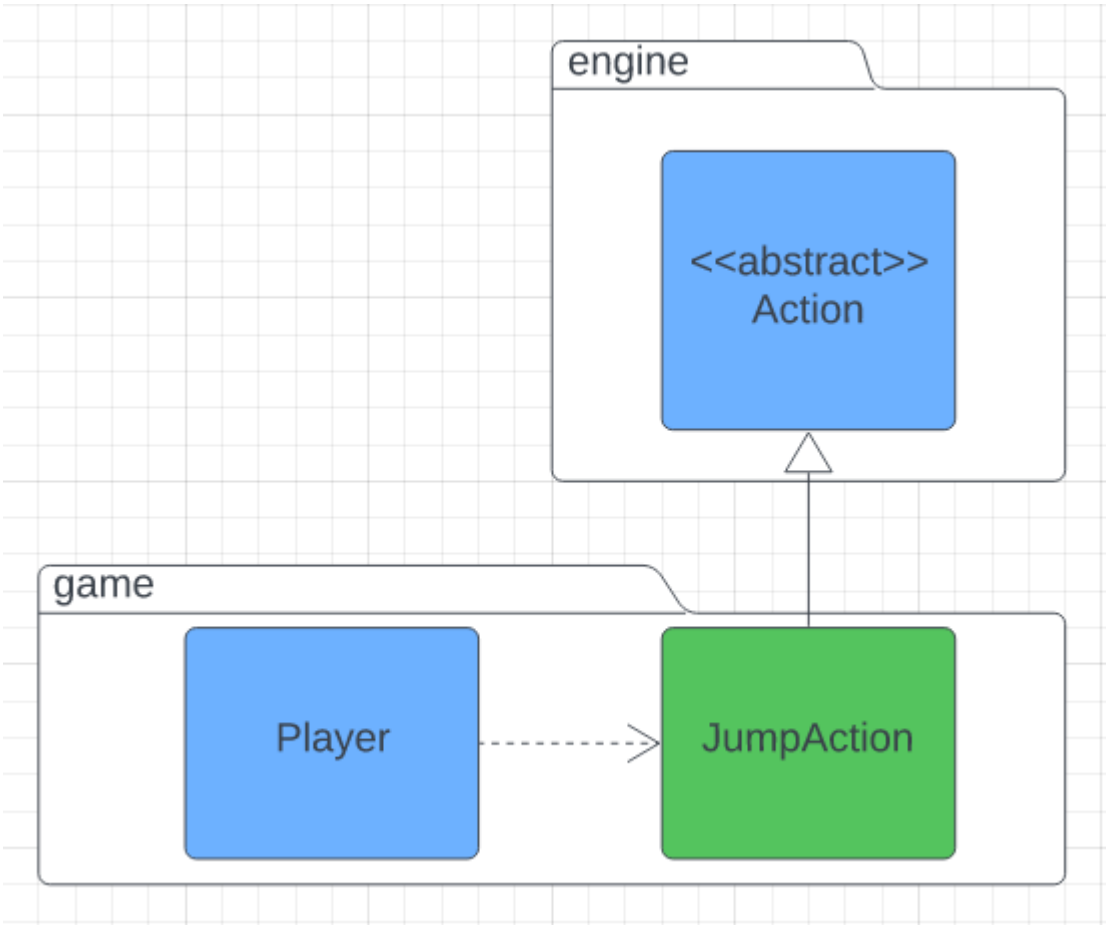
For Req1, this means you can open the Sprout class and see that it has a 10% chance to spawn a goomba, while in the Sapling class the goomba spawning method is not present. In other words, code not relevant to that specific type of tree is not present in that specific type's class. This means we don't 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 off 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 , 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.

*Class diagram only shows changes made in requirement 2 specifically, so this class diagram doesn't show the interaction with magic items.

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:

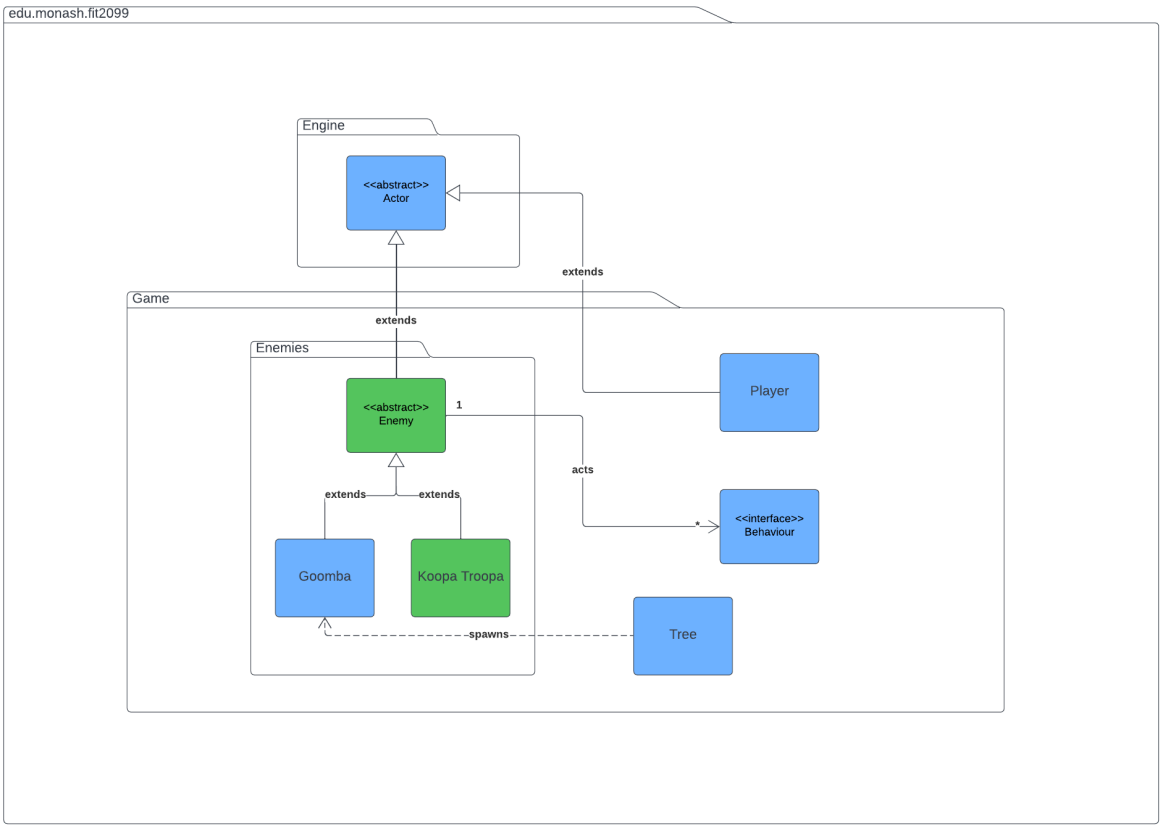
Going up requires a jump, going down doesn't. Enemies cannot jump at all.

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:



Complete class diagram with relevant entities as per the requirements.

Sequence Diagram:

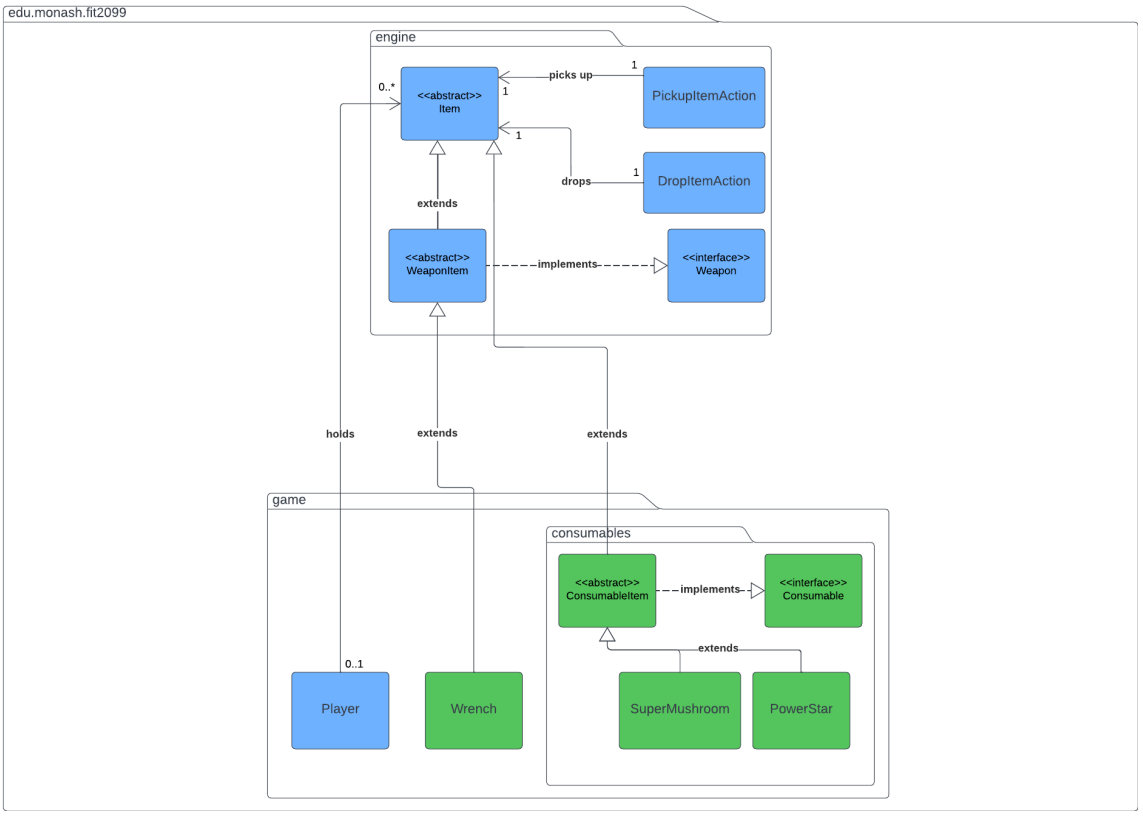
Design Rationale:

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:



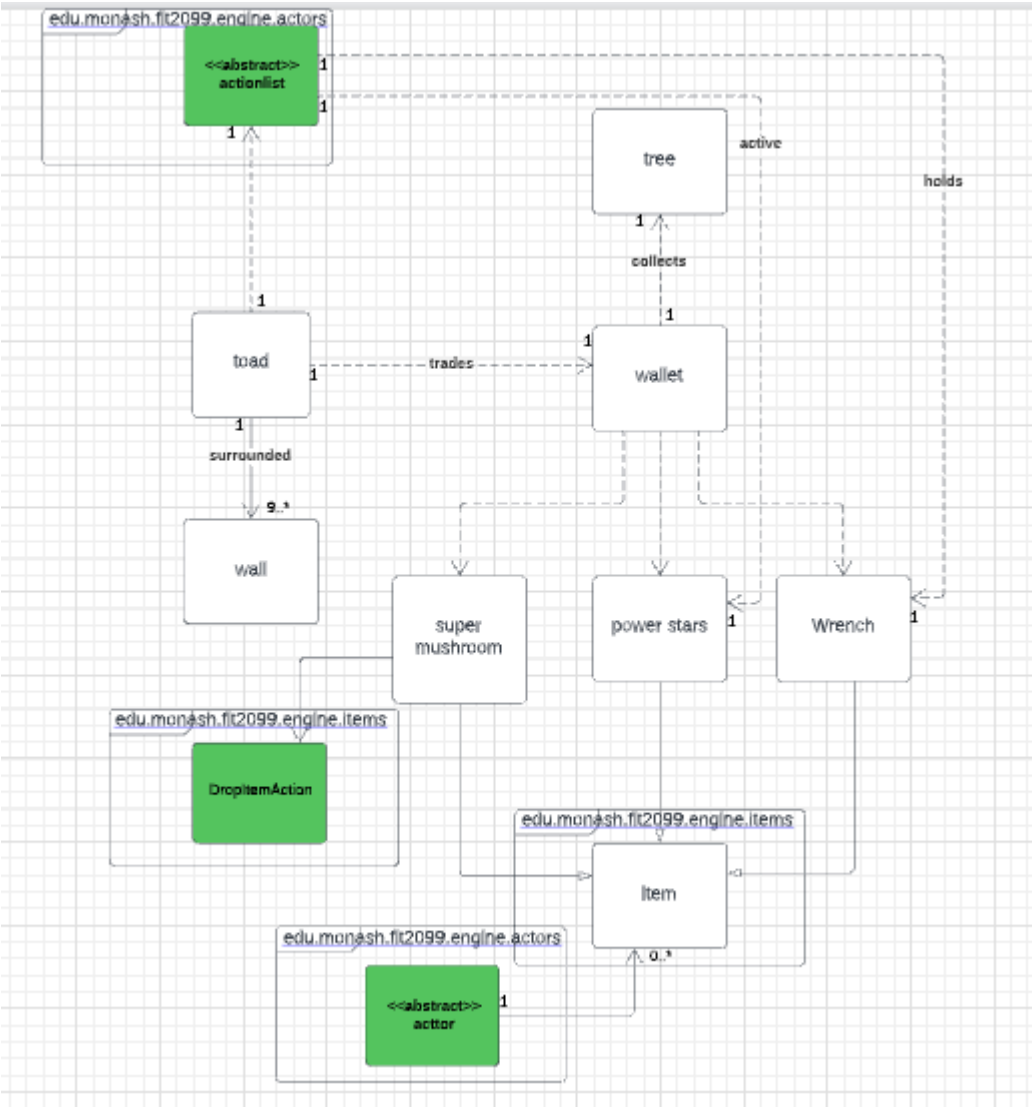
Complete class diagram with relevant entities as per the requirements.

Sequence Diagram:

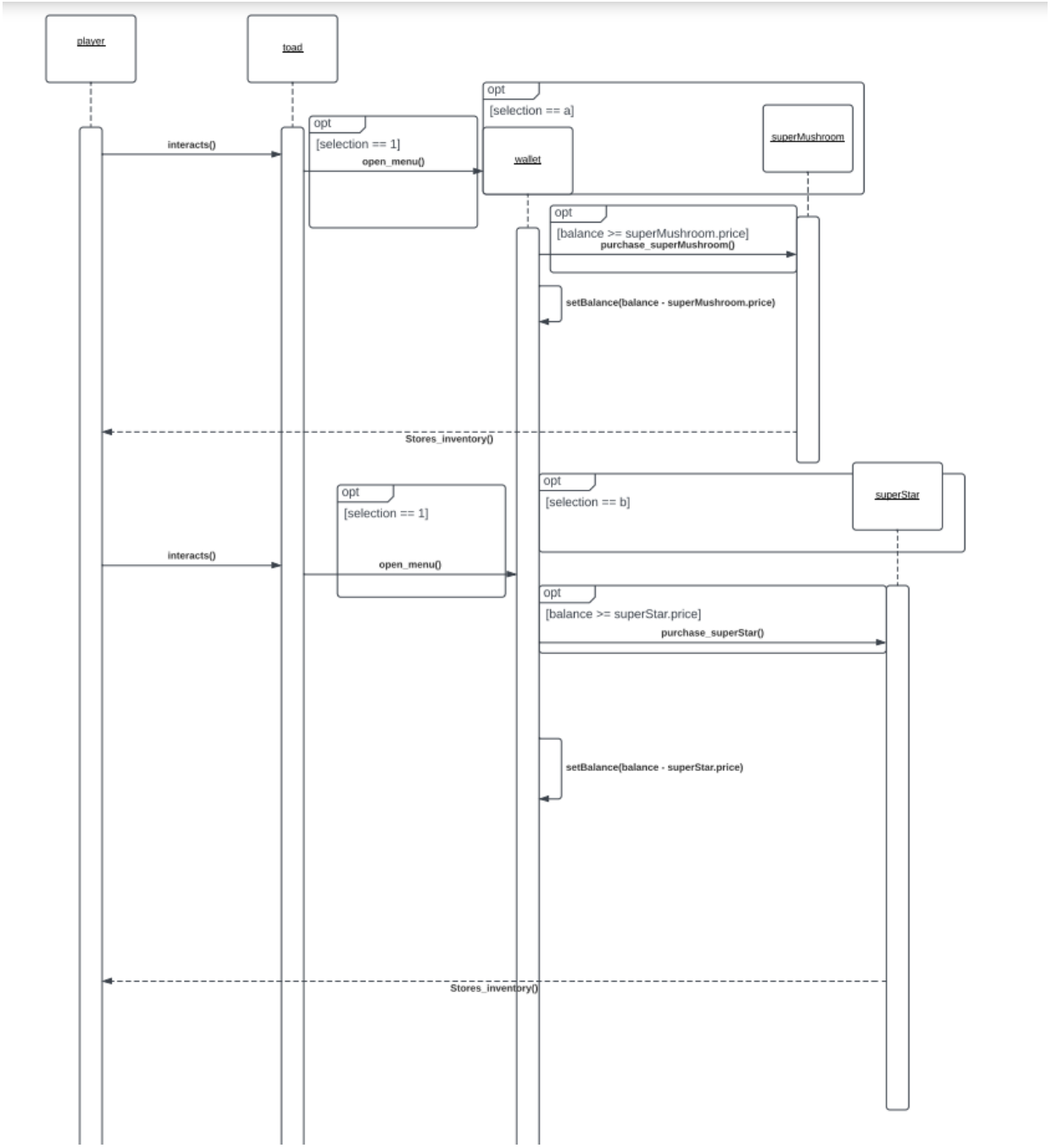
Design Rationale:

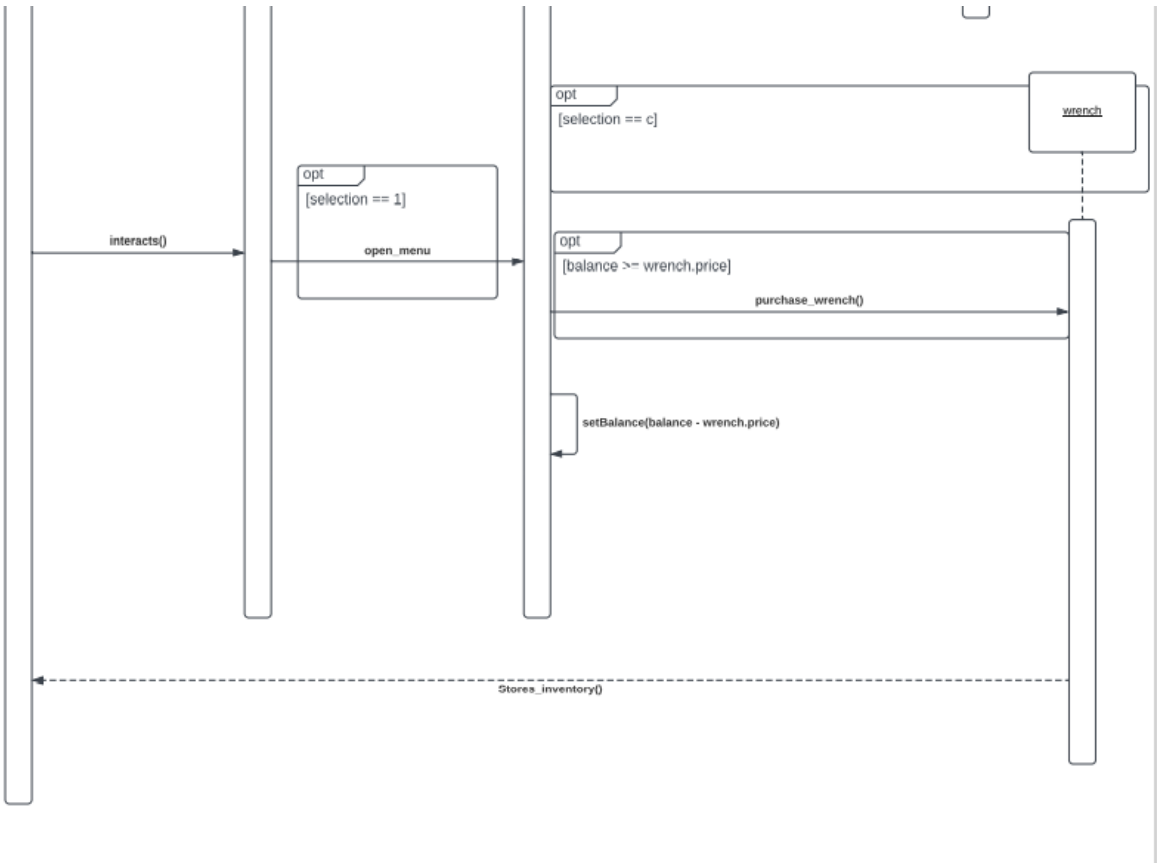
Req 5 + Re6:

Class Diagram:

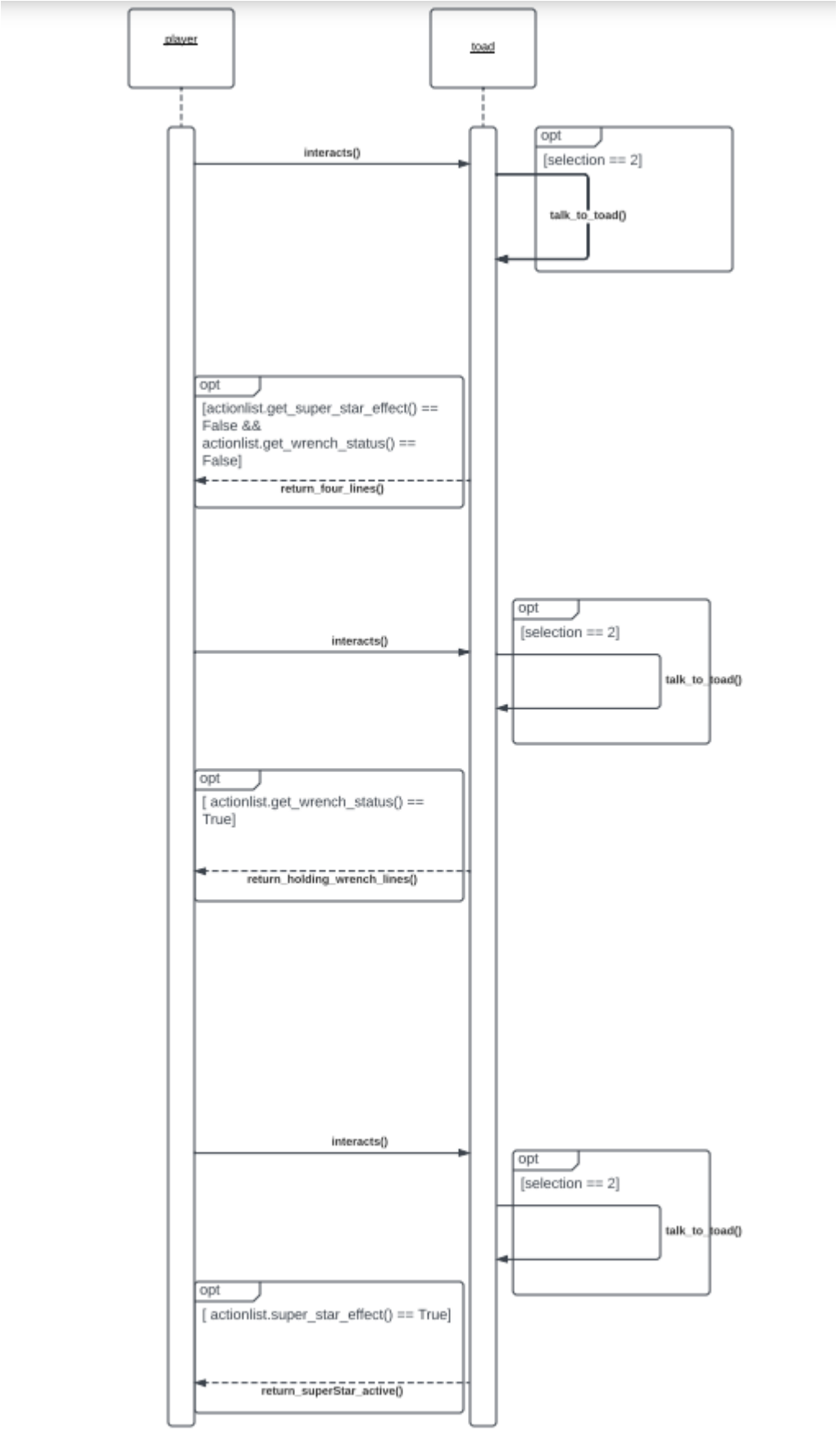


Sequence Diagram (re5):


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()

Sequence diagram (re6)


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:

Design rationale

Toad:

- We need to have a toad class
- There needs to be 2 selections
- Selection 1: trading
- Selection 2: monologue
- This toad is surrounded by at least 9 walls

Wallet:

- This section will print a list of items a player can buy for example.

A: Mario buys Power Star ($600)

B: Mario buys Super Mushroom ($400)

C: Mario buys Wrench ($200)

If any of the option is chosen, this will call a public method within the wallet class corresponding to each choice. This public method will do a check between balance and price. Then it will add the item in to the item list. Then it will do a deduction of the balance.

Each item extends from the parent class which is item. An actor will have an association with item.

Super mushroom will have a direct association with drop item action.

Tree will spawn coin randomly hence we need to use the method from the tree class which will be a dependency.

Monologue:

- Monologue will not have a class of itself. This will be implemented within the toad class.
- After pressing selection 2 this will call the monologue method.

- Inside the monologue method this will call an action list to check whether the player is holding wrench or having power stars active. In either case this will call other methods which will print different lines corresponding to each case.

Req 7:

Class Diagram:

Sequence Diagram:

Design Rationale: