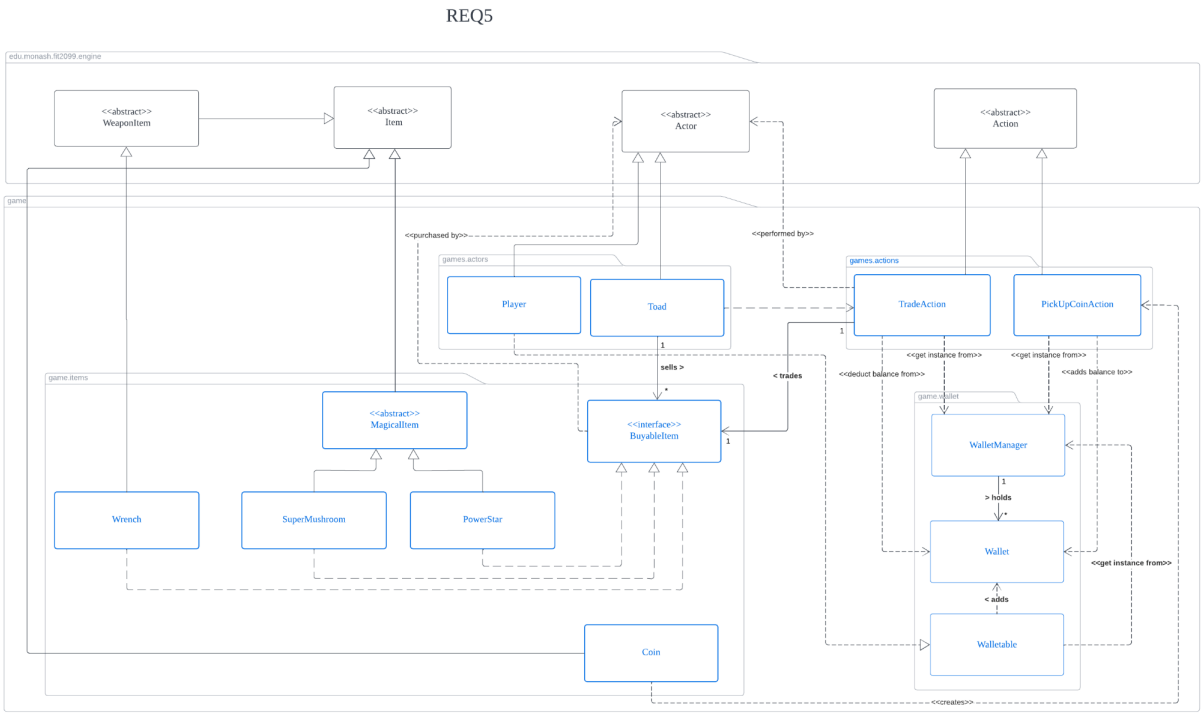
REQ5

Class Diagram:



Requirement:

The coin(`$`) is the physical currency that an actor/buyer collects. A coin has an integer value (e.g., $5, $10, $20, or even $9001). Coins will spawn randomly from the Sapling (`t`) or from destroyed high grounds. Collecting these coins will increase the buyer's wallet (credit/money). Using this money, buyers (i.e., a player) can buy the following items from Toad:

1. Wrench: $200
2. Super Mushroom: $400
3. Power Star: $600

Toad (`O`) is a friendly actor, and he stands in the middle of the map (surrounded by brick Walls).

Design Rationale:

In requirement 5, we are required to implement the functionality of trading. When the player is beside Toad, players are able to buy Wrench, superMushroom or PowerStar. Few classes and two interfaces are created or modified here, which are **TradeAction**, **PickUpCoinAction** extending Action class, **Wallet, WalletManager, Walletable interface, Toad**, **Coin**, **PowerStar**, **SuperMushroom**, **Wrench** classes, and **BuyableItem** interface.

We would first discuss BuyableItem, PowerStar, SuperMushroom and Wrench. BuyableItem is an interface for all the items which the actor can buy from toad. In this interface there is only one abstract method getBuyableItemPrice which would be overridden by the buyable items which have a price. In assignment 2, we added a purchasedBy(actor) method implemented by all the buyableItem to addItemToInventory(). The reason for creating this method is because we want to addItemToInventory(this) in the item class instead of calling addItemToInventory(buyableItem) which would be explained below. PowerStar, SuperMushroom extending MagicalItem and Wrench extending WeaponItem are implementing BuyableItem. They will override the getBuyableItemPrice methods to return their respective price value, which are 1) PowerStar $600 2) SuperMushroom $400 3) Wrench $200. This implementation follows **Interface Segregation Principle (ISP)** because the interface is small and it only returns buyableItemPrice and puts the item into inventory when purchase is done. The interface is kept as small as possible and has very specific tasks so buyableItems are not forced to implement other methods they do not require. It also obeys **OCP** as it is very easy to add an item which is buyable without modifying codes in other classes.

We then move to the discussion of TradeAction class. In assignment 1, tradeAction has 3 variables: actor, map, and buyableItem. In assignment 2, we decide to remove actor and map attributes as they are not required. To run a tradeAction, actor and map just have to be inserted as parameters in the execute() method. This follows the principle that **classes should be responsible for their own properties to reduce dependency** of

different classes. We create a constructor to initialise these 3 variables so toad can add tradeAction. We also implement a menuDescription to let users know what items they are buying. An execute method is created to create a tradeAction for actors to buy BuyableItem. This method starts with getting the balance in the wallet using the getInstance and getBalance method in Wallet. Then, we obtain the price of buyableItem using the getBuyableItemPrice method. If the balance is higher than the price, trade is successful. We would call getInstance and deductBalance methods in Wallet to reduce the balance. Previously, we added the item to inventory in the execute() method. However, a problem arises when we cannot add buyableItem to inventory because addItemToInventory() method only accepts Item class and not BuyableItem. It can be solve by upcasting (Item)this.buyableItem but it is not a good design. So we assign the work of adding items to inventory to the buyableItem itself. We would call the purchasedBy(actor) method of the item to add the item into the inventory of the actor for current implementation. Then, the execute() method prints a message saying the actor buys the item. If the balance is not enough, we print a message indicating balance is not enough. This implementation follows **Don't Repeat Yourself Principle (DRY)** as in the buyableItem there is a method for retrieving price. By using this interface, we do not need to get the price of Wrench, PowerStar and SuperMushroom using 3 different getBuyableItemPrice methods, only 1 is needed.

For the trading process, another key implementation we had is when a Power Star is created inside Toad's inventory, the Power Star's turns left is set to 11 instead of 10, this is because we wanted the game to work in a way such that when a newly bought Power Star is passed from Toad to Player, the Power Star would have 10 turns left to consume in the Player's inventory, which is logically correct according to the specifications. If we had set the turns left to 10, after the Power Star is bought and added into Player's inventory, the Power Star would have only 9 turns left to consume, which we thought is not logical and a bad gaming experience.

Moving on to toad, it is a class extending Actor. It has an instance variable which is an arraylist of BuyableItem called buyableItemList. This list is used to store a list of all buyable items. We build a constructor which would call Actor (super) constructor. After that we initialise our buyableItemList by calling the refillBuyableItemList method. RefillBuyableItemList is a method used to create a new arraylist containing PowerStar,

SuperMushroom and Wrench and assign it to buyableItemList. This method is created because after TradeAction, we would add the item in the list to the inventory of actor. We then need to refresh the list using the method to prevent the situation of different players holding the same item instance. We also override the allowableActions from Actions. In this scenario, if a player is near toad, toad will add a speakAction to actionList. Then, it will refillBuyableItem list to create new instances of buyableItems, next loop through the buyableItemList to add TradeAction of every buyableItem to the actor and return the ActionList. We would then override the abstract method from Action and perform DoNothingAction in it because it is toad.It follows the principle of **Classes should be responsible for their own properties** because we are putting refillBuyableItemList right before adding TradeAction rather than anywhere else because they are related to each other.

In order for the buyer to be able to trade with the toad, we would need to have a wallet system that can track the total balance of the buyer's money. Thus, we created a **Wallet** class where the instance of the **Wallet** that stores the balance would be publicly accessible via a public static method, which we would be able to add balances to and also deduct balances from. We also wanted to make our game extensible and be able to accommodate for multiple players/actors, hence not only we have a **Wallet** class, we also have a **Walletable** interface that would be implemented by generally actors that can have a wallet, and also a **WalletManager** to handle all actor's wallets. Whenever a walletable actor is created, a wallet for the actor is created and registered to the wallet manager's HashMap with the actor as key, and the wallet as the value. This works because whenever we want to get the wallet of an actor, we could just access the WalletManager instance and use that actor as the key to get that wallet, and we could add or deduct balance from the wallet. This follows the **OCP** as our system is open for extension as we can handle other classes that might have a wallet in the future, thus we are not limited to just having wallet for a player,

As we already have our wallet system, the key question is how would we pick up Coins that are on the ground and add the credit into our wallet balance? We did this by implementing another new action called **PickUpCoinAction**. A coin would add **PickUpCoinAction** into its allowableActions list and whenever an actor executes this

PickUpCoinAction, we would get the wallet instance and add balance to the wallet according to the coin's value, then the coin would be removed from the map. The reason for creating a new **PickUpCoinAction** that extends **Action** class is because we wanted to follow **SRP** where it would handle situations where we pick up coins and interact with our wallet.

There was also another issue that we faced along the way, which is, as we overridden the toString method for PowerStar to also include the turns left, this results that when player is near Toad to purchase items, the turns left would appear in the console, for example "Mario buys Power Star ($600) (10 turns left)", which is very weird. Hence we added another checking in PowerStar such that when PowerStar's counter has 11 turns left, it would not return the turns left along with its name. This works because the PowerStar created in Toad has 11 turns left (as explained earlier).

*Updates that have been done for REQ5 (class diagram & design rationale)

I. Instead of just having one wallet for the entire game, we decided to make it extensible with a Walletable interface that is implemented by classes that can have wallets (generally it would be subclasses of actors), we also have a WalletManager to hold all these walletable classes' wallets.
II. Made sure that Power Star would have 10 turns left to consume when Player purchases a Power Star from Toad. This is done by creating a Power Star with 11 turns left before fading in Toad, so that when Power Star is added to Player's inventory upon purchase, Power Star would have 10 turns left to consume.
III. Remove actor and map attribute from tradeAction. Change relationship between actor and tradeAction in UML from dependency to association.
IV. Add purchasedBy action in BuyableItem interface to add item to inventory. Add dependency between BuyableItem interface and Actor.
V. Execute() method in tradeAction only handle check balance, deduct balance and call purchasedBy Action instead of add item to inventory in execute method
VI. Add a player class which would implement a walletable interface. Walletable interface will have dependency with wallet and walletManager. WalletManager will have dependency with tradeAction and pickUpCoinAction.WalletManager which holds wallets will have association with wallet. Also, wallet class will have

dependency with tradeAction and pickUpCoinAction as they will check, add and deduct balance from wallet..