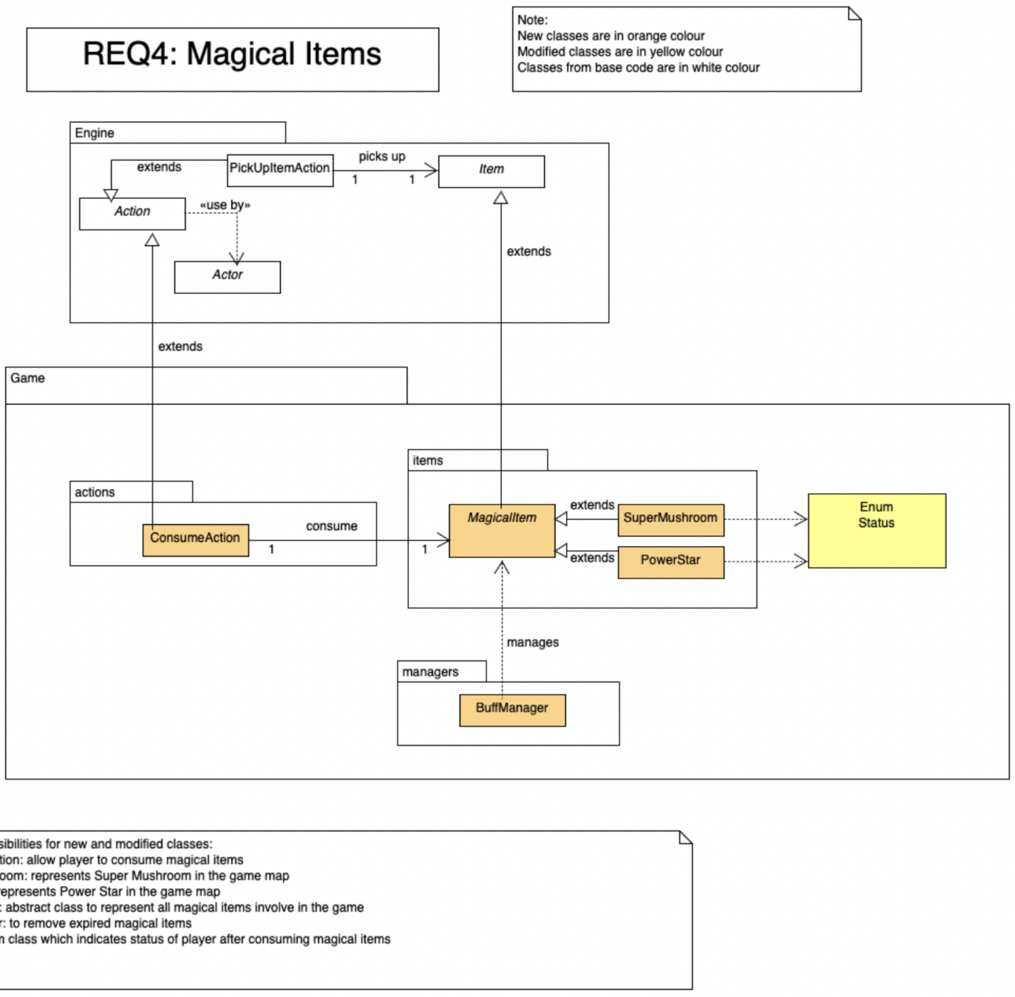
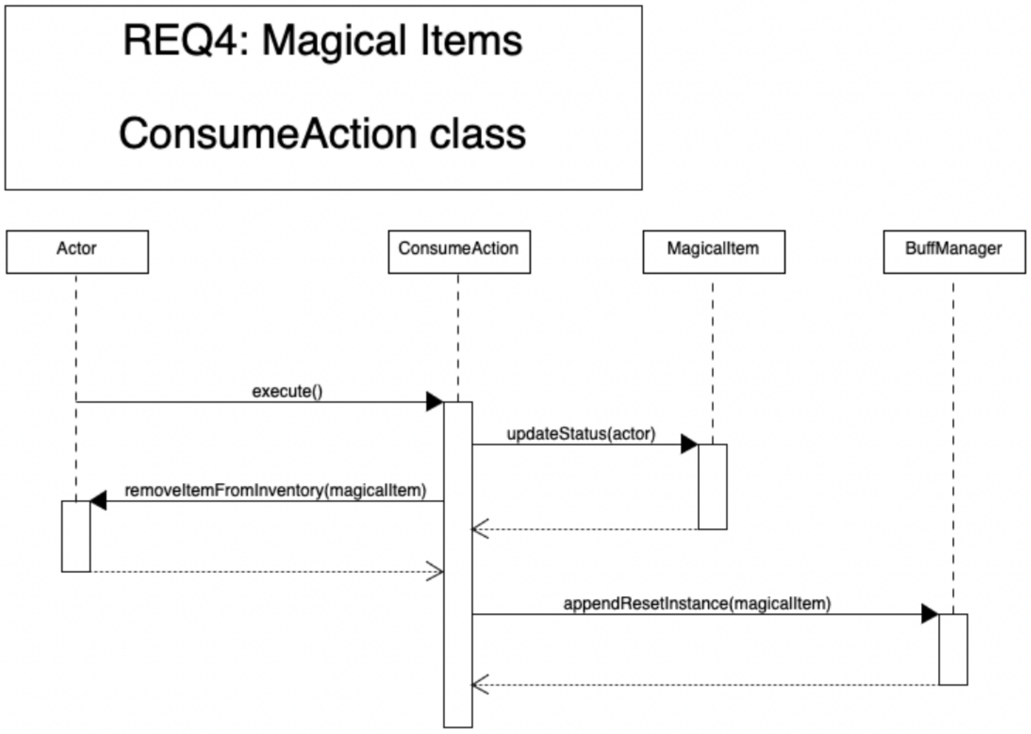
REQ4: Magical Items



Overall responsibilities for new and modified classes:
1) ConsumeAction: allow player to consume magical items
2) SuperMushroom: represents Super Mushroom in the game map
3) PowerStar: represents Power Star in the game map
4) MagicalItem: abstract class to represent all magical items involve in the game
5) BuffManager: to remove expired magical items
6) Status: enum class which indicates status of player after consuming magical items



ConsumeAction class

This class is use to update player’s status and remove magical items from players inventory whenever player consumes the item.

What changes in the design between assignment 1 and assignment 2:

- There is association between this class with the MagicalItem class because it has an attribute of MagicalItem in its constructor
- execute method in assignment 2 is modified to update status of actor depending on magical item consumed and add the item into actor’s inventory by obtaining the magicalItem returned from menuDescription method instead of using if-else statement (in assignment 1) to check which magicalItem is consumed

```
@Override
    public String execute(Actor actor, GameMap map) {
        magicalItem.updateStatus(actor);
        actor.removeItemFromInventory(magicalItem);
        BuffManager.getInstance().appendResetInstance(magicalItem);
        return actor + " consumed the " + magicalItem;
    }
```

Why i choose to do it that way:

I decided to make that change because,

the use of if-else statements as planned in assignment 1 will be a breach of Open Closed Principle because if a new item is added a new conditional statement needs to be added to the method, making the method not close for modification. As the game develops further, It might cause confusion during methods implementation as more and more if-else statements will be needed in the execute method. Liskov Substitution Principle(LCP) will also be breached in this case as the method will not work if a new instance of magical item subclass is pass in unless a new conditional statement is added, but that will not be a good programming practice.

By applying changes in assignment 2,

the OCP can be implemented as the method will remain the same regardless of what kind of and how many items are involved in the game. Besides, LCP can be implemented too as an instance of MagicalItem subclass can always be pass in and this ease the process of further development when more magical items are involve.

MagicalItem class

This class is an abstract class extended from the Item class to provide blueprint for magical items such as Super Mushroom and Power Star classes.

What changes in the design between assignment 1 and assignment 2:

This class does not exits in assignment 1. In assignment 2, some important methods in this class are:

- updateStatus method to update status of the actor upon consumption
- tick method to keep track of the number of turns player makes while having the magical item
- setIsExpired to set isExpired to true when player loses the magical item effect

Why i choose to do it that way:

In assignment 2, this is an abstract class created to be the parent class of PowerStar and SuperMushroom class. Since Power Star and Super Mushrooms are magical items, they both share some same functions and hence an abstract MagicalItem class can be created to reduce duplicated codes as they can access methods from MagicalItem class by calling super. In assignment 1, without this abstract class, there are some repeated codes in the PowerStar and SuperMushroom class, leading to the breach of the Don't Repeat Yourself (DRY) principle.

The Don't Repeat Yourself Principle can be applied using approach in assignment 2 as common used methods don't have to be copy pasted in every magical item subclasses and this improve code readability and reduces possibilities of making errors.

PowerStar class

What changes in the design between assignment 1 and assignment 2:

- PowerStar class is extended from MagicalItem class instead of Item class
- a tick attribute is added into the PowerStar constructor
- There are two tick methods added into this class. One to keep track of the number of turns player makes when player is under the power star effect. INVINCIBLE capability will be remove from player after 10 turns and the power star will be remove from player's inventory, while the other one is to keep track of the number of turns it is left to be available in player's inventory or in the game map. The power star will be remove from the map once it reaches 10 turns.
- currentStatus method is added into this class to add INVINCIBLE capability to player while player haven't reach 10 turns while having the power star effect.

Why i choose to do it that way:

As explained in the MagicalItem class section, this class is extended from MagicalItem class instead of Item class in order to reduce duplicated code and hence obey the DRY principle. By doing so, default methods can be inherited from the its parent class (MagicalItem).

By adding tick method to the Power Star class, the SRP can be implemented since the power star effect is only available to actor 10 turns after consuming it. By having a tick method, each power star in the inventory, ground, and consumed power star will have their own "timer". This is important to deal with overlapping power star effect. For example, if actor consumes a power star, play 5 turns, and then consume another power star, the actor will suppose to have 15 turns with the power star effect.

SuperMushroom class

What changes in the design between assignment 1 and assignment 2:

- SuperMushroom class is extended from MagicalItem class instead of Item class

Why i choose to do it that way:

As explained in the MagicalItem class section, this class is extended from MagicalItem class instead of Item class in order to reduce duplicated code and hence obey the DRY principle. By doing so, default methods can be inherited from the its parent class (MagicalItem).

BuffManager class

The BuffManager class has dependency with the MagicalItem class because we need a list consisting of instance from MagicalItem class.

What changes in the design between assignment 1 and assignment 2:

This class does not exist in assignment 1

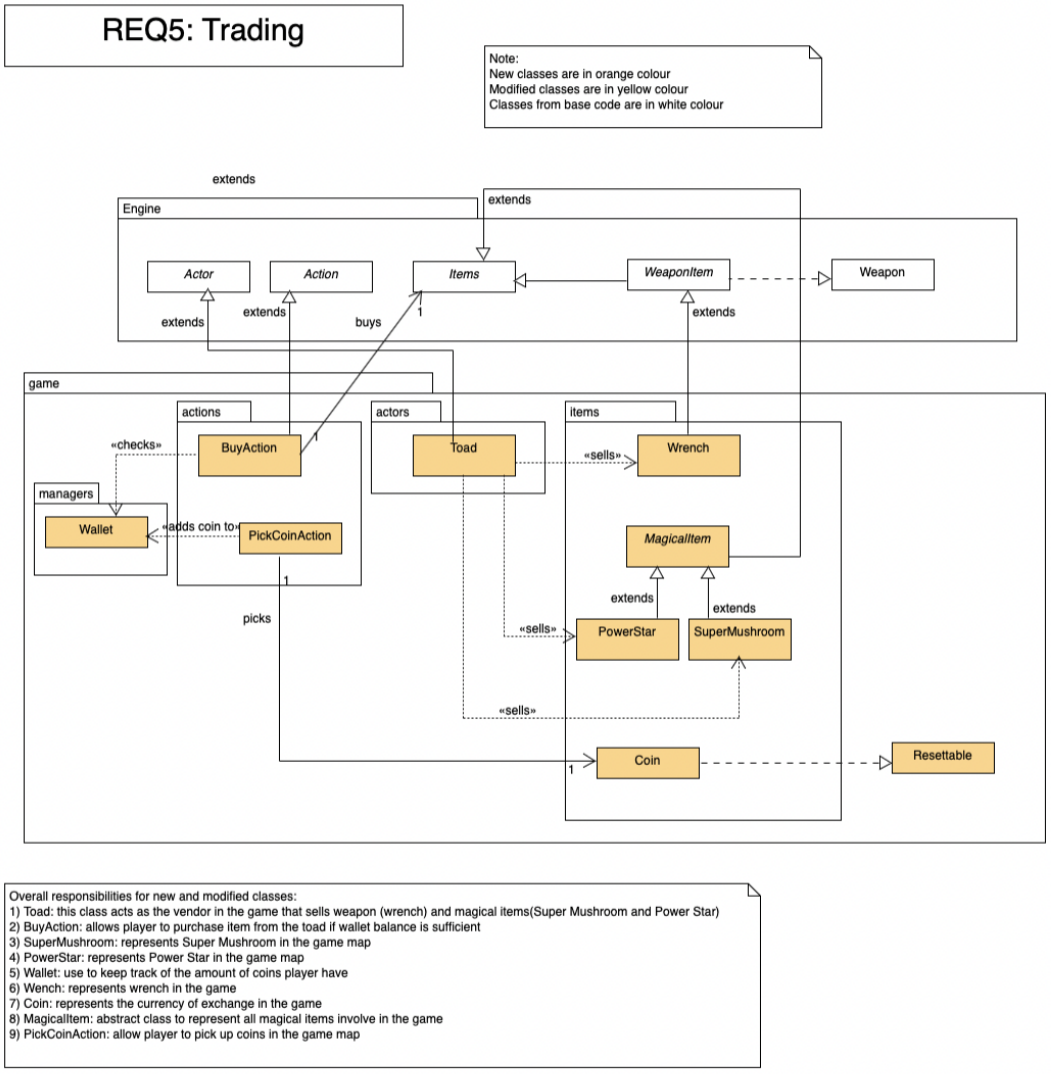
- run method to transverse through the list of magical items. This method is execute in every play turn of the players.
- appendResetInstance method to add resettable instance to the list

Why i choose to do it that way:

This class is added in assignment 2 to loop through the list of magical items and remove expired magical items from the list. By having a static factory getInstance method, the constructor of the BuffManager class can be call without creating a new instance of it. For example `BuffManager.getInstance().run(map.locationOf(this));`

This therefore implements encapsulation as the run() method can be call without getting internal information of the class, hence hiding implementation.

REQ5: Trading

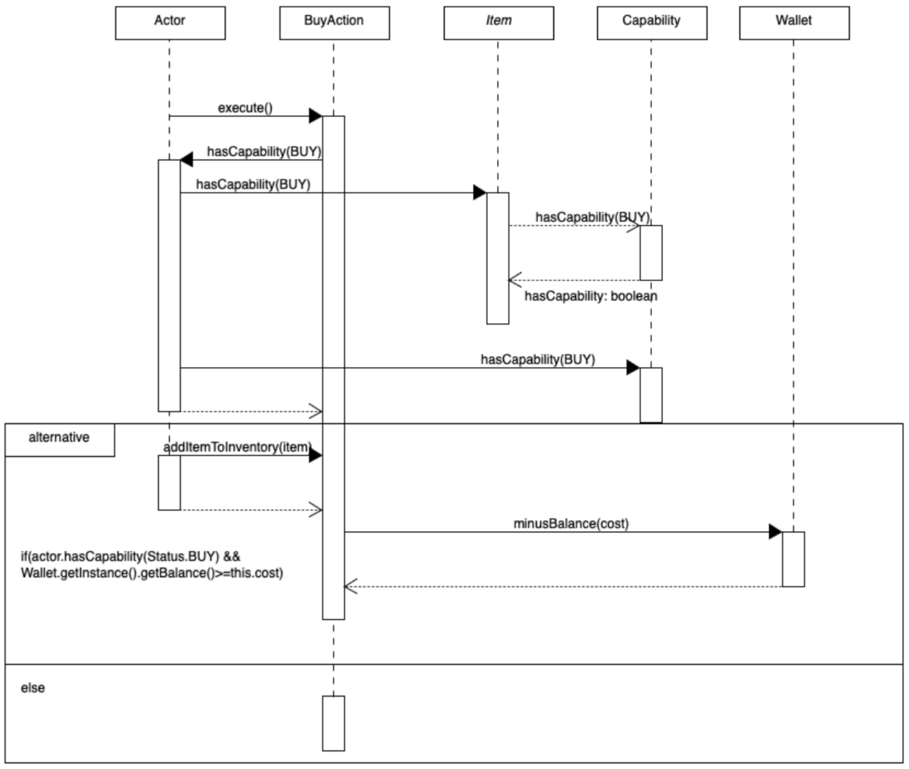

REQ5: Trading

Note:
New classes are in orange colour
Modified classes are in yellow colour
Classes from base code are in white colour

Overall responsibilities for new and modified classes:
1) Toad: this class acts as the vendor in the game that sells weapon (wrench) and magical items(Super Mushroom and Power Star)
2) BuyAction: allows player to purchase item from the toad if wallet balance is sufficient
3) SuperMushroom: represents Super Mushroom in the game map
4) PowerStar: represents Power Star in the game map
5) Wallet: use to keep track of the amount of coins player have
6) Wench: represents wrench in the game
7) Coin: represents the currency of exchange in the game
8) MagicalItem: abstract class to represent all magical items involve in the game
9) PickCoinAction: allow player to pick up coins in the game map

REQ5: Trading

BuyAction class



PickCoinAction class

This class extends the Action class to allow player to pick up coins and then add the collected coins into the users wallet. This class has association relationship with the Coin class as there will be a coin attribute initialised in the PickCoinAction constructor so that it knows which coin (of what value) is picked.

What changed in the design between assignment 1 and assignment 2:

- execute method is a method that is override from its parent class (Item class) to add the value of coins collected by player into the player’s wallet balance if player picks up the coin, and then display the amount of coin picked up by player in the menu description.

```
// assignment 2
@Override
public String execute(Actor actor, GameMap map) {
        map.locationOf(actor).removeItem(coin);
        Wallet.getInstance().addBalance(coin.getValue());
        System.out.println("Wallet balance: " + Wallet.getInstance().getBalance());
        return menuDescription(actor);
    }
```

Why i choose to do it that way:

Changes were applied in assignment 2 because I realised there is no need to obtain the hashCode of the actor and coin's location. I also decided to write those lines of code in the override execute method because the method will be invoked after actor enters the hotkey, hence it is not needed to create a separated method to add value of coins into wallet balance.

Using the new approach, the Single Responsibility principle can be implemented as this class will only be used when used to add value of the picked up coin into the actor's wallet balance.

Toad class

What changes in the design between assignment 1 and assignment 2:

- tradeItem method is removed
- playTurn method is a method that is override from the parent class (Actor class). This is to add new instance of items into the saleItem list in every play turn of the toad.
- allowableAction method is another method that is override from the parent class (Actor class). This method is to add buyAction to the toad's customer (player) so that they can purchase item from the toad based on the hotkey keyed in.

Why i choose to do it that way:

tradeItem method is removed because that process was moved to be carried out by the BuyAction class. Having the tradeItem method in assignment 1 will cause the Toad class to breach SRP as the Toad should only act as an "item provider" in this requirement.

BuyAction class

BuyAction is an action that allows the player to use coin to purchase items from the toad. It is trigger by the Toad class when players use buy action to purchase items from the toad. This class is associated with the Item class as Item instance needs to be created in the BuyAction class constructor in order to pass in item bought by players.

What changes in the design between assignment 1 and assignment 2:

- execute method checks if actor’s (which has BUY capability) wallet has enough money, if yes, add the item into actor’s inventory and subtract the item price from wallet, and then return a string indicating item bought by player. Else, inform player that balance is insufficient.

```
/**
     * Add the item to the actor's inventory and subtract the cost from actor's wallet balance
     *  display "You don't have enough coins!" is actor's balance is insufficient
     * @see Action#execute(Actor, GameMap)
     * @param actor The actor performing the action.
     * @param map The map the actor is on.
     * @return a suitable description to display in the UI
     */
    @Override
    public String execute(Actor actor, GameMap map) {
        String output="";
        if(actor.hasCapability(Status.BUY) &&
                Wallet.getInstance().getBalance()>=this.cost){
            actor.addItemToInventory(item);
            Wallet.getInstance().minusBalance(cost);
            output+= actor + " purchased a " + this.item;
        }
        else{
            output+="You don't have enough coins!";
        }
        return output;
    }
```

Why I choose to do it that way:

In assignment 2, the execute method which is override from the action class is use to process the actor’s buying action instead of using a separate method as designed in assignment 1 because the execute method will be invoke once the actor enters the hotkey corresponding to the BuyAction class. In the execute method, the actor needs to be check if it has the BUY capability to ensure that it is a player.

Furthermore, by having Item in the BuyAction constructor, the LCP can be implemented as as an instance of Item subclass can always be pass into the execute method. This approach is important because the Toad not only sells magical item but weapon such as wrench too (or even other types of item in the future). Hence the execute method can take in any type of item as long as the class is a subclass of Item class.

Wallet class

This class is used to store and keep track of the amount of coins players have, using static factory method.

What changes in the design between assignment 1 and assignment 2:

- Wallet class is made to have static factory methods
- receiveCoin method is removed
- getBalance method to return the current amount of coins in player's wallet
- minusBalance method to subtract item cost from player's wallet balance
- addBalance method to add value of coin into player's wallet balance

Why i choose to do it that way:

Static factory methods are use in Wallet class so that the constructor of the Wallet class can be call directly instead of creating a new instance of Wallet class. By doing this, encapsulation can be use as the behaviour of the Wallet class can be specify via parameters without knowing the Wallet class hierarchy. Also, static factory methods eliminates the need to create a new instance of Wallet for every player instantiated. Code readability can also be improved in this case. For example, the wallet balance can be obtained by `Wallet.getInstance().getBalance()`.

Wrench class

What changes in the design between assignment 1 and assignment 2:

- HAVE_WRENCH capability is added to the Wrench in the constructor of the Wrench class instead of creating an additional updateStatus method as in assignment 1
- Wrench class extends WeaponItem class instead of Item class

Why i choose to do it that way:

Based on the Actor class given in the engine, the hasCapability method will iterate through the actor's inventory and returns true if the item has the required capability, and if the item has that capability, the capability will be added to the actor too. Which means, actor will have capabilities that items in the inventory have. Hence there is no need to create an additional method as it may be redundant. By using this approach, the SRP can be implemented as the Wrench class will only be responsible to store information of itself and add capability to actors having it. It also reduces lines of code and therefore improve readability.

Furthermore, Wrench was decided to be extended from WeaponItem class because wrench is a weapon and instance variables from the WeaponItem constructor is needed for Wrench to contain additional instance such as damage, hit rate and verb. By extending from WeaponItem, Wrench class will be able to access those instance variable just by calling super, hence reducing lines of repeated code. Extending Wrench class from Item class as did in assignment 1 was an incorrect approach as additional instance variables need to be declare and it may cause confusion in the BuyAction class.

Coin class

What changes in the design between assignment 1 and assignment 2:

- no changes
