Design rationale - req 4

Player Class

This class represents players in the game. Player class extends Actor class. It is associated with the Vendor class to allow players to purchase magical items. Player class uses PickItemsAction (which extends the Action class) to pick up magical items, which are Super Mushroom and Power Star; and use PickCoinAction to pick up coins.

ItemFound class

This class is use to manage magical items picked up by players from the ground. There is dependency between ItemFound class with the PowerStar and SuperMushroom class because we need to add a new instance of those items into the player's inventory when they are picked up by players from the ground. It also have dependency with the PickUpItemAction as a new instance of that class is needed by players to pick up items.

Description of method:

- itemIsFound method check if player found the item (if player's location== item's location). if yes, then it will call the execute() method from the PickItemAction class that will remove the item from the map and add the item into player's inventory, and then print a description into the console using the menuDescription method

```
PickUpItemAction pickup=new PickUpItemAction(item);

public void itemIsFound(Player player, Item item){
        int itemLocation= item.hashCode();
        int actorLocation=player.hashCode();

        if(itemLocation==actorLocation){
            pickup.execute(player,map);
        }
    }
```

```
/**
   * Add the item to the actor's inventory.
   *
   * @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) {
    map.locationOf(actor).removeItem(item);
    actor.addItemToInventory(item);
    return menuDescription(actor);
  }
```

```
/**
   * Describe the action in a format suitable for displaying in the menu.
   *
   * @see Action#menuDescription(Actor)
   * @param actor The actor performing the action.
   * @return a string, e.g. "Player picks up the rock"
   */
  @Override
  public String menuDescription(Actor actor) {
    return actor + " picks up the " + item;
  }
```

- itemIsConsumed method is use to update the player's status, adding capability to the player when player consumes the magical item. The item will then be removed from player's inventory.

```
public void itemIsConsume(Player player, Item item){
        if(item==powerStar){
            powerStar.updateStatus(player);
            player.removeItemFromInventory(item);
        }
        if(item==superMushroom){
            superMushroom.updateStatus(player);
            player.removeItemFromInventory(item);
        }
    }
```

Why i choose to do it that way:

This class is created to deal with items collected by players because items picked up by players will be added into the player's inventory first instead of carrying out their respective effect immediately. Hence this class is needed to add item into player's inventory upon pick up and remove it from player's inventory and update player's status after player consume the item.

Advantage:

Using this design, the Single Responsibility principle can be implemented as this class will only be used when used to manage magical items obtained by player’s in the game, hence higher cohesion.

Disadvantage:

As the game develops further, more item will be available for players to pick up and gain different capabilities. It might cause confusion during methods implementation as more and more if-else statements will be needed in the itemIsConsume method.

PickCoinAction

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 dependency relationship with the Coin class because a new instance of coin need to be added into this class.

Description of method:

- addToWallet method adds the amount of coins collected by player into the player’s wallet if player reaches location that contains coins. The coin will then be removed from the map after player collects it

```
public void addToWallet(Player player, GameMap map,Coin coin) {

        int itemLocation= coin.hashCode();
        int actorLocation= player.hashCode();

        if(itemLocation==actorLocation){
            map.locationOf(player).removeItem(coin);
            int current=player.getWallet().getBalance();
            player.getWallet().setBalance(current+ coin.getValue());
        }

    }
```

Why i choose to do it that way:

Since PickCoinAction is an action, functions in the action class are required and hence need to extend the action class. This class is created for players to pick up coins instead of using the PickUpItemAction in the engine package because the PickUpItemAction in the engine package stores item picked by player into the inventory.

However, this is not the case for coins. Hence I created this class to add values of coin into the player’s wallet instead of into player’s inventory.

Advantage:

Using this design, the Single Responsibility principle can be implemented as this class will only be used when used to pick up coins (which this class will then be used to increase the player’s wallet balance) hence higher cohesion. Open Close Principle can also be implemented as this class extends the Action class, by adding functionality to the Action class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Action class. This enables the Action class to support new functionalities as well as being added new methods easily. For example, the menuDescription method from the Action class can be override to display the amount of coin player received.

```
public String menuDescription(Player player) {
        return "Player receives " + coin.getValue() + " coins.";
    }
```

Disadvantage:

N/A

PowerStar class

This class is a subclass of items. Players that consume it will be healed by 200 hit points (hp) and become invincible. The invincible effect replaces fading duration (aka, fading turn's ticker stops), and it lasts for another 10 turns. It fades and disappears from the game within 10 turns. Sets the player’s status such that player does not need to jump to higher level ground, add 5 coins into player’s wallet for every destroyed ground, make player immune to damage and enable player to attack enemy successfully

Description of method:

- updateStatus method that add capabilities(effects of PowerStar) to the player. This method will be call when player consumes the Power Star

```
public void updateStatus(Player player){
        player.addCapability(Status.POWER_STAR_BUFF);
    }
```

Description of attributes:

- HEALED_HIT_POINTS is a public static integer attribute with value of 200 that indicates the hit points players can get healed by after consuming the Power Star

Why i choose to do it that way:

Since Power Star is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the effect it can bring to players and its ability to be traded for coins

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns and as little overlapping as possible. Which in this class, the class is only responsible for storing information about the Power Star. Excessive use of literals was also prevent by declaring HEALED_HIT_POINTS as private static attribute. This prevents confusion during coding process. Furthermore, if the value of HEALED_HIT_POINTS needs to be change, changes only need to be done at one place, which is at the line where that attribute is declared instead of going through entire code and changing the value all of the “200” . This minimise possibilities of producing errors too. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Item class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, the PowerClass can use the addCapability method from the Item class to add capability to players.

Disadvantage:

N/A

SuperMushroom class

This class is a subclass of items. The effect will last until it receives any damage (e.g., hit by the enemy). Once the effect wears off, the display character returns to normal (lowercase), but the maximum HP stays.

Description of attribute:

EXTRA_HP is a public static integer attribute with value of 50 that indicates the max hit points players can get after consuming the Super Mushroom.

Description of method:

updateStatus method that add capabilities to player such that:

- the display character evolves to the uppercase letter (e.g., from m to M).
- it can jump freely with a 100% success rate and no fall damage.
- increase max HP by 50

```
public void updateStatus(Player player){
        //display character evolves to the uppercase letter
        player.addCapability(Status.TALL);

        player.addCapability(Status.SUPER_MUSHROOM);
    }
```

Why i choose to do it that way:

Since Super Mushroom is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the effect it can bring to players.

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns and as little overlapping as possible. Which in this class, the class is only responsible for storing information about the Super Mushroom. Excessive use of literals was also prevent by declaring EXTRA_HP as private static attribute. This prevents confusion during coding process. Furthermore, if the value of EXTRA_HP needs to be change, changes only need to be done at one place, which is at the line where that attribute is declared instead of going through entire code and changing the value all of the “50” . This minimise possibilities of producing errors too. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Item class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example,

the SuperMushroom class can use the addCapability method from the Item class to add capability to players.

Disadvantage:

N/A