FIT2099 Assignment 1

Lab 13 – Group 33

Members: Ong Di Sheng, Kennedy Tan, Mark Manlangit

**Design Rationale**

REQ5: Trading

1. **PickUpCoinAction**

PickUpCoinAction is an extension from PickUpItemAction. This class allows an Actor (e.g. Player) to pick up a Coin if the current location where the Actor is standing on has a Coin. It has Coin as its attribute because when the Player executes this action, the Coin must be removed from the location and the value of the Coin which can be obtained by calling getValue method will be added to the Player’s balance in the Wallet.

Each instance of PickUpCoinAction can only have exactly one Coin stored as its attribute, so the multiplicity of PickUpCoinAction to Coin is one.

The reason why we do not choose to use PickUpItemAction to pick up the coin is because the PickUpItemAction will store the item in the actor’s inventory. In this case, we only want to get the value of the coin being picked up and add the value into the balance in the Wallet but not store the coin in the actor’s inventory. Due to a different implementation for the action of picking up a coin compared to other items, we choose to create a new class PickUpCoinAction that inherits from PickUpItemAction to deal with coin object only.

PickUpCoinAction satisfies the **Single Responsibility Principle**, because it only picks up a coin and not deals with any other items. This is to separate the action of coin being picked up from other items, which have their own implementations.

1. **Tradable**

Tradable is an interface in which all items that can be bought from the Toad such as Wrench, SuperMushroom and PowerStar should implement. In this interface, there is one getPrice method signature without the body and the concrete implementation for this method must be done by every Tradable items. This is because every Tradable items have their own price. The getPrice method should also be made as static so that we can obtain the price of the Tradable items without creating the item instance. The price should also be made as a constant in each of the Tradable item class to **avoid excessive use of literals** and this would make the understanding of the code and maintenance much easier in the future.

Every class that implements Tradable interface also satisfies **Open-Closed Principle** because they override the getPrice method and each of the Tradable item can have their own implementation or price without modifying the existing code.

1. **TradeAction**

TradeAction is an extension from Action abstract class. This class allows the Player to buy items such as Wrench, SuperMushroom and PowerStar from the Toad. It does not store anything as its attributes because when the Player executes this action, it will first get the balance of the Player in the Wallet and check whether the Player is able to buy the chosen items based on the character that the user inputs in the menu. The price of the chosen items can be obtained by using getPrice static method as mentioned earlier in (b). If the transaction is successful, the chosen item will be added to the Player’s inventory and the balance in the Wallet will be deducted too, otherwise an error message will be displayed to the user.

TradeAction satisfies the **Single Responsibility Principle**, because it only focuses on the trading process between the Player and the Toad. This is to separate the action of trading from other actions, which have their own implementations.

1. **Wallet**

Wallet is a class that holds an integer attribute called balance which is the amount of money that Player has currently. When the Player picks up a Coin, the value of the Coin will be added to the balance of the Wallet. The balance attribute should be made as static since it will be used in the static getBalance() and subtractBalance() method. This means that the balance of the Player can be retrieved without instantiating Wallet instance by using the static getBalance() method during TradeAction. The same goes for subtractBalance() when the item is purchased successfully.

Each instance of Player can only have exactly one Wallet stored as its attribute, so the multiplicity of Player to Wallet is one.

Wallet satisfies the **Single Responsibility Principle**, because it only deals with the current balance of the Player and not dealing with any other items.

REQ6: Monologue

1. **SpeakCapable**

SpeakCapable is an interface in which Toad should implement since Toad can choose a random sentence from a fixed list to speak to the Player when the Player interacts with the Toad. In this interface, there is one onSpeak method signature without the body and the concrete implementation for this method must be done in Toad class. In the Toad class, there will be a String array storing all the sentences which can be spoken by the Toad.

The reason why SpeakCapable interface is created even though currently there is only 1 Toad class implementing it is because there might be other actors where we might add or modify them to be speakable in the future.

1. **SpeakAction**

SpeakAction is an extension from Action abstract class. SpeakAction will store a SpeakCapable object as its attribute so that it can invoke the onSpeak method in the Toad class. Each SpeakAction is responsible for handling a SpeakCapable object, so the multiplicity for SpeakAction to SpeakCapable is one. When executed, it will undergo multiple checks before displaying the message to the user. The first check will determine whether the Player holds a Wrench in the inventory. If the Wrench is in the inventory, a constant number using static final keywords can be used to indicate that the first message should not be displayed by passing the constant number in the onSpeak method. The same goes for the situation when the Player is in the Power Star status so that the second message will not be displayed. The use of constant in SpeakAction class allows us to **avoid excessive use of literals**.

SpeakAction satisfies the **Single Responsibility Principle**, because it only focuses on the speaking process between the Toad and the Player. This is to separate the action of speaking from other actions, which have their own implementations.

REQ7: Reset Game

1. **ResetAction**

ResetAction is an extension from Action abstract class. ResetAction will call upon the ResetManager.getInstance().run() in the execute method to reset the game. Therefore, every Resettable instances that are added to the resettable list will execute their own resetInstance method.

ResetAction satisfies the **Single Responsibility Principle**, because it only focuses on the reset process when the user enters hotkey ‘r’. This is to separate the action of reset from other actions, which have their own implementations.

1. **Resettable**

Resettable is an interface where Enemy, Player, Coin and Tree should implement since they are involved during the reset process. Every class that implements the Resettable interface should override resetInstance method to reset their abilities, attributes or items. Since we make use of interface, any part of the program that accepts Resettable instance will be able to accept those classes that implements the Resettable interface (the **Liskov Substitution Principle**), so this modification will not fail the program.