Design Rationale

Requirement 1

Changes

Wallet Class

Originally, it was intended to have **Wallet** as an attribute inside the **Player** class instead of a class on its own. However, by having it as a class on its own, this adheres to the Single-Responsibility Principle, rather than having the **Player** class to keep track of the balance of coins.

Previous design

Coin --> <<stores>> --> Coin

The circular association between Coin objects was implemented with the intention to ease the process of resetting the game when implementing Requirement 7. However it was rather redundant after being able to understand and utilize the base code.

Tree* extends *Ground

All concrete classes that inherit the abstract *Ground* Class have the necessary methods to provide proper functionality for every ground in the game map.

Sprout, Sapling, Mature extend *Tree*

All concrete classes that inherit the abstract *Tree* Class have the necessary methods to grow, wither and spawn items or enemies.

Application --<<creates>>--> Sprout --<<creates>>--> Sapling --<<creates>>--> Mature

A tree in the game has three cycles with unique abilities at each stage. A naïve alternative implementation for this requirement would be similar to that in demo.conwayslife 's Tree class whereby, all of the logic is coded within the same class. The inclusion of special abilities at each stage means we would have a convoluted and un-modularised code.

Hence, instead of having **Application** create ***Tree*** directly, we have **Application** create **Sprout** and make ***Tree*** an abstract class since **Sprout, Sapling** and **Mature** are indeed Trees that share the same basic attributes and functionalities. This also allows for different implementations of certain shared functionalities like spawning enemies. We have thus managed these objects through separation of concerns, in adherence to the SRP (Single-Responsibility Principle) which states that each class should have a specific responsibility. By making the ***Tree*** class abstract, it avoids being a GOD class and conforms to both the Dependency Inversion Principle and the Open-closed Principle where in case of any changes that are necessary to be made to either subclass, there will be no need to make changes in the parent class. This increases maintainability and encourages extensions of new features to the Trees.

Requirement 2

JumpAction extends *Action*
Since JumpAction class inherits the Action class, it would have the necessary methods to perform the action and provide a text description before and after execution.

Ground objects (ie. Wall, Tree) are the ones that grant Mario, the player Actor, the action to jump onto them (be on the same location as the Ground objects). One of the possible approaches was to implement JumpAction as a method in the Player class. It would certainly work fine since this action is only used by the Player. However, it would mean that maintaining that action and its possible future extensions would be hard. Hence, it is better to extend Action and implement it as a separate class. By having the Actor class as an attribute in JumpAction (one-to-one association), we can allow this action to be carried out by other actors in the future if we need to. Besides, code management would be much cleaner. Our final design aligns with the Single Responsibility Principle as it serves only one purpose or functionality, which is an action to an actor.

As a general explanation of what this class does, it provides the player a way to move up to the high ground, which is also called jump. This can only be instantiated/added when the Player is near a Ground object of type Tree or Wall. When near, Player will have the option to execute the JumpAction. Depending on the obstacle ahead, the Player will be tested with different jump rates, meaning that jumping up also has a certain success rate. However, if Player has the 'SUPER' capability, obtained after eating the Super Mushroom, the Player will be able to jump freely with 100% success rate and no fall damage. Finally, when the Player successfully moves to the new position, it should not replace the Ground object.

Previous Design
The GameMap registers Player as an existing actor on that location, so when Player moves away, whichever Ground object that was there previously would be there again as usual.

Changes
The new JumpAction() method does not register Player to the location, it should just plainly shift the position of the Player.

In terms of actual changes, the initial implementation of this class was supposed to be done within the Player's class. However, it was not possible as it does not have a method to instantiate this action to itself, therefore, the alternative method was to instantiate JumpAction() in the Ground objects that can obstruct players, namely the Tree and Wall.

Requirement 3

Changes

Goomba, Koopa extends Enemy extends Actor

The initial plan was to let Goomba and Koopa to just extend its functionalities from Actor class since it was mostly using only Actor, however it was then made known that an Enemy class could make things more convenient in an event where certain class or methods are required to be implemented to only Goomba and Koopa, for example, Resettable class. This class still adheres to the Single Responsibility Principle.

Previous Implementation

Goomba, Koopa extends *Actor*

By inheriting Actor class, Goom and Koopa now have access to methods that allow them to function as characters (actor).

As Goomba and Koopa are supposed to be individual objects that have different characteristics and traits (which require different implementation of methods), it is only logical to create them separately as different classes. This allows us to maintain them individually. Besides, they do not share similar functionalities when it comes to non-inheritable functions. It does not justify the creation of an abstract Enemy class as the majority of the codes involve those of the Actor anyway. If an Enemy class is used, then further changes to some implementations or even additions are necessary, which would not adhere to the DRY principle. Therefore, we are able to reduce potential dependencies on other classes by inheriting Actor instead. Hence, this shows that both classes apply the Single Responsibility Principle.

Application ---<<creates>>---> Goomba
Application ---<<create>>---> Koopa

The application creates both Goomba and Koopa at the start of the game. The existing Application class needs to be modified to accommodate this. Koopa, just like Goomba, has a hashmap of <Integer, Behavior> pair as its attribute (hence, the association relationship to behavior interface). This allows Koopa to have more complex actions such as the FollowBehavior which allows to follow the player around the map.

Sprout ---<<create>>---> Goomba
Mature ---<<create>>---> Koopa

Moreover, Application also creates Sprout. Sprout experiences growth and follows a cycle and has a certain percentage of possibly spawning Goomba in its location at every turn. Similarly, Koopa may be spawned by Mature tree.

Requirement 4

SuperMushroom, PowerStar extends *Item*

SuperMushroom and PowerStar are best implemented as items in the game by inheriting the Item class as they are meant to be portable and picked up or dropped by the actor. They both now have access to the necessary methods to be displayed on the map and hold their own separate characteristics.

ConsumeAction extends *Action*

Since our Player needs a way to interact with the items and gain powers, it is better to implement it as a separate action than to code the logic in the Player's class. This would significantly improve readability and ensures the separation of concerns. Inheriting the Action class also means that we will be able to display the action option through the menuDescription() method and return a string after the execution. The main logic can be housed inside the execute() method, where we could call Player's methods such as resetMaxHp() and increase their capabilities. Instead of having a direct association with only SuperMushroom and PowerStar classes, we opt to have the Item class as one of the attributes (one-to-one association relationship), as it allows us to have a general implementation to other items that may require similar action in the future. Maintaining and expanding the game functionality would be much easier. This ConsumeAction would then be added to the allowableActions list of SuperMushroom and PowerStar.

SuperMushroom, PowerStar --<<Uses>>--> Status

Status is an enum class that contains statuses which can be used as a means of indicating actions or abilities that can be attached or detached. By using this class, we can know when the Player has consumed an item. For example, in the execute() method, if the Player consumes a SuperMushroom, then we can add a capability stating that the Player is 'SUPER'. This indication of the Player's status can be used to determine the outcomes of other actions such as JumpAction where the player could jump freely when he has the 'SUPER' capability (consumed a mushroom). For PowerStar, we can use an enum to indicate whether it has been consumed or not. This would help us to reset the life counter as Power stars have a fixed life span of 10 turns until it disappears or fades from the Player. While this may not seem as straightforward, it reduces the amount of attributes need to check for the statuses of the actor or items. This is in line with the Single Responsibility Principle where classes should not have too many responsibilities.

Requirement 5

Previous design:
Coin extends Item
Coin is an item that the Player could collect which would increase their wallet balance. The coin class would be responsible for creating coins that hold a certain value. We have decided to keep track of all Coin objects that are on the ground by using an ArrayList. This is because, it would be easier to have the location of all these coin objects when resetting the game as they would need to be removed.

Changes in Coin Class:
As mentioned in Requirement 1, there is no need to store the location of all coin objects to implement Requirement 7 (Resetting the game) as we can implement the removal of all Coin objects on the ground by overriding the tick() method which is already implemented in the ***Item*** class.

Wrench uses <<Enum>> Status
We use the status enum to give capability to wrench since it is capable of destroying shells. Thus the actor who has a wrench in their inventory will get the capability too.

TradingAction association with Item, Actor
TradingAction class now has a one to one association with Item and Actor objects as they as used as attributes in its class. This is so we are able to keep track of the actor who possesses the item which would allow use to add or remove capabilities accordingly. Next, the Item is stored to keep track of the type of Item that is being traded such as PowerStar, SuperMushrrom and Wrench.

Wrench extends Weapon*Item*
WeaponItem extends *Item.* Hence, it is wise for us to implement Wrench as a class that extends the WeaponItem class, instead of the Item class. This is because, in order for us to use the Wrench as a weapon against enemies, we would have to implement certain attributes that would provide damage. Since these extra capabilities are already provided by the abstract WeaponItem Class, we inherit that instead.

TradingAction extends *Action*
By inheriting the Action class, TradingAction has access to the necessary methods that facilitate the execution of the trade logic while also displaying the menu description and text of the action that transpired during the game.

The TradeAction is responsible for the trade logic whereby, in the execute method, it checks if the Player has enough money in their wallet for the item that they would like to purchase. If it is enough, then the item is added to the Player's inventory, if not a message is printed. By having a dedicated trade class, we can perform similar logic to any item that the Player buys. An alternative way would be to have this method included in each item that could be traded which involves a lot of repeated codes and it will not be easy to maintain or make changes. By having a general trade action that has the Item class as its attribute (one-to-one association), we are able to implement this action to any item if needed. Besides that, we

have opted for a simple wallet system where the Player's money is kept as an integer in an attribute. This is because the transaction of coins does not require the object itself and creating a separate wallet class would be redundant for its simple intended usage.

Requirement 6

Changes

Monologue Class

An addition of the **Monologue** class to store all the possible monologues that **Toad** will display according to certain conditions in an array list. The list of possible monologues was supposed to be stored in **Toad** class itself, but this does not accommodate for future extensions, for example adding new possible monologues for Toad to display when interacted with.

InteractAction → <<uses>> → Monologue

InteractAction has an instance of the Monologue class in order to filter out the monologues to randomly choose from under certain conditions.

Previous design

InteractAction extends *Action*

All concrete classes that inherit abstract *Action* Class have the necessary methods to execute a certain action while providing a written text before, through menuDescription() method, and after the execute() method, by returning a string of the outcome.

InteractAction:

A general explanation of how this class works is that it provides a function where the **Player** can interact with Toad. ***Toad*** extends the actor class which allows it to inherit its methods. **InteractAction** would be added to Toad's list of allowable actions for the player to execute. The **execute()** method in this class will check whether the actor or in our case, the player has a Wrench in their inventory or whether they have consumed a Power Star. it will then return a String from an array list that stores the four possible sentences that Toad will respond with when the player chooses to interact with Toad. We could add more sentences if required, by using the add() method for ArrayList compared to manually increasing an array's size. Alternatively, this functionality could be implemented directly inside the **Toad** class or inside the Player's playTurn() method, but the action would only be applicable to Player interacting with Toad. Not to mention, that it would violate the Single Responsibility Principle where each class should have a focused responsibility. Implementing the InteractAction as a separate class and having an association with the *actor* will definitely make it easier to maintain and accommodate for extensions to have other actors interact in the future.

Requirement 7

Changes

***Tree,* Goomba, Koopa, Player, Coin, PowerStar implement Resettable:**

The resetInstance() implemented in the respective classes that implements the Resettable interface is just so that the items / players have the status to be resetted (which in other words means that ResetAction has been executed by Player). The actual implementation of how the item/actor is resetted is in either tick() for Items or playTurn() for Actors.

The **ResetManager** will then call resetInstance() for all Resettable Objects that are stored in the resettableList when **ResetAction** is executed. The main game loop will then handle the actual resetting of Items/Actors.

Previous design

***Tree,* Goomba, Koopa, Player, Coin --<<implement>>--> Resettable**

All classes that implement the Resettable interface will have the necessary methods to provide the functionality to reset the objects of each respective class.

ResetAction extends *Action*

All concrete classes that inherit the abstract *Action* class have the necessary methods to execute a certain action and provide a text description before (menuDescription) and after execution.

ResetAction:

A general idea of how this class works is that it creates an instance of the ResetManager in order to access the array list of Resettable Items. Subsequently, in the **execute()** method, it can call **run()** on all the items in the Array List.

***Tree,* Goomba, Koopa, Player, Coin implement Resettable:**

Instead of all the above classes having a direct relationship with **ResetManager**, we have decided to make use of the **Resettable** interface in accordance with the Liskov Substitution Principle. This is because all the classes above have different requirements when it comes to implementing the reset function in their respective classes. The functionality of ResetManager remains unchanged even though more classes require a reset function in them.