FIT2099 Assignment 1

Lab 13 – Team 33:

1. Ong Di Sheng (31109667)
2. Mark Gabriel Sta. Ana Manlangit (29350387)
3. Kennedy Tan Sing Ye (31108121)

Design Rationale

REQ1: Let it grow! 🌳

1. **Tree**
   Tree is an abstract class that is an extension from the Ground abstract class. This class represents a Tree on the map.

2. **TreeType**
   TreeType is an enumeration class that will be used to represent the different types of stages a tree can be in. It will also be used to retrieve the spawn rate information related to each stage of the tree.

3. **GrowCapable**
   GrowCapable is an interface that the various stages of the tree will implement. Each stage of the tree will be able to perform a grow function that represents a tree moving to its next stage.

4. **SpawnCapable**
   SpawnCapable is an interface that the various stages of the tree will implement. Each stage of the tree will be able to perform a spawn function that will be able to create another entity either on or around a tree.

5. **Sprout**
   Sprout is an extension of the Tree class which implements the GrowCapable and SpawnCapable interface. This will be the first type of tree that will be created on the map. This class will be responsible for randomly spawning a Goomba enemy on its location every turn as well as growing into a sapling once it has aged a certain amount.

6. **Sapling**
   Sapling is an extension of the Tree class which implements the GrowCapable and SpawnCapable interface. This will represent the second stage that a tree can be in on the map. This class will be responsible for randomly spawning a Coin item on its location every turn as well as growing into a mature tree once it has aged a certain amount.

7. **Mature**
   Mature is an extension of the Tree class which implements the GrowCapable and SpawnCapable interface. This will represent the last stage that a tree can be in on the map. This class will be responsible for randomly spawning a Koopa enemy on its location every turn, randomly spawning new sprouts in adjacent fertile tiles every 5 turns and will have a chance every turn to wither and die, which will turn it into dirt.

Implementing a single Tree class that would perform growing and spawning depending on the current stage the tree is at would be hard to maintain as it would have too many responsibilities and would not satisfy the Single Responsibility Principle. So instead of a single Tree class handling multiple different behaviours for each stage, we can instead make the Tree class an abstract class. This will mean that all tree stage classes that extend Tree will inherit the base functionality. By separating each tree stage as different classes that all extend the Tree class, the tree stages can inherit the Tree methods while also implementing their own versions of grow and spawn functionality through the use of the GrowCapable and SpawnCapable interfaces.

I separated the tree's growing and spawning functionality into the two separate interfaces GrowCapable and SpawnCapable. All the tree types implement the SpawnCapable interface but only Sprout and Sappling implement the GrowCapable interface. This is because a Mature tree has stopped growing but may still spawn entities. This satisfies the Interface Segregation Principle since the tree types do not have to implement methods they do not care about.

I used an enumeration called TreeType to store the spawn rate information of each stage that the tree is at. This was done to improve the maintainability and extensibility in the event of a change in a spawn rate or addition of a new tree stage.

REQ2: Jump Up, Super Star! 🌟

1. **JumpAction**
   JumpAction is a class that is an extension from the Action abstract class. This class allows an actor to attempt a jump to a high ground. The chances that an attempt is successful is dependent on the type of terrain the high ground is. It then returns a message of whether the jump was successful or not.

2. **JumpableGround**
   JumpableGround is an interface that is used to add the different types of high grounds into the HighGroundManager and retrieve a given high grounds type.

3. **HighGroundManager**
   HighGroundManager is a class that keeps track of all the high grounds on the map.

4. **HighGroundType**
   HighGroundType is an enumeration class that will be used to represent the different types of high grounds a ground can be. It will also be used to retrieve the success rate for each type of terrain.

In this game, a high ground will give an Actor (i.e., the Player) an action that it can be jumped to. In order to perform the jump, the action will first have to retrieve the odds of performing a successful jump depending on the terrain. Implementing multiple if statements to check the type of high ground and retrieving the relevant odds would lead to extra dependencies and would lead to high coupling. Instead, we implement a HighGroundManager that keeps track of all the high grounds on the map. We can then extract the current high ground type and retrieve the relevant odds through the HighGroundType enumeration, which stores all the odds of success for each type of high ground. This will satisfy the Dependency Inversion Principle as the JumpAction class will not have to depend on all the types of high ground and instead implements an abstraction through the use of the JumpableGround interface and the HighGroundManager.

This implementation will also satisfy the Open-closed Principle as new types of high ground can be added by implementing the JumpableGround interface. This will allow the new ground to be considered high ground and has allowed us to extend the JumpAction class without modifying the class itself.

REQ3: Enemies

a) **Enemy**

Enemy is a new abstract class that extends from Actor abstract class. It contains the general characteristics of the actors that can deal damage to the player. With abstraction, we managed to extend the code functionality without modifying the existing Enemy class. This follows the **Open-Closed principle** which has define as, classes should be open for extension but closed for modification. In our assignment, currently we have two enemies, which are Goomba and Koopa. Both having AttackBehaviour, but they still having different characteristics. Therefore, Goomba and Koopa now inherit from Enemy abstract class, then we can just add on their characteristics in their own classes. For instance, Koopa will go to a dormant state when defeated, but Goomba will not. In the future, if we want to create other enemies, we can simply create one more class which inherit from the Enemy abstract class. If we want to add the abilities or characteristics of certain enemy, we simply just add on in their own classes.

b) **Behaviour**

Behaviour is an interface that included in the based code which determines the current behaviour of the enemy. It is split into three behaviour which are FollowBehaviour, WanderBehaviour and AttackBehaviour. So that the enemy can only implement the methods they need at that specific situation. This follows the **Interface Segregation Principle** which is defined as larger interface should be split into smaller ones.

c) **Koopa**

Koopa is an extend of Enemy class, which extends from Actor class. Koopa is one of the enemy in this system, but it has different characteristics compared to Goomba. Koopa will go to dormant state by using Status enumeration and it is not allow to move around and when it is destroy by the player using wrench, it will drop a SuperMushroom by DropItemAction. These characteristics make it much different compared to Goomba. So, instead of putting all types of enemies inside Enemy class and using if statement to implement their characteristics, we create classes for every enemy and extends from Enemy class. In this way, also their general characteristic is defined in the Enemy abstract class. In the future, if we need to ass enemies to the system, we simply just create classes for them and extends from Enemy abstract class. This satisfied the solid principle, **Open-Closed Principle**. Enemy is the abstract class closed for modification, characteristics for enemy should only present in their own classes.

REQ4: Magical Items

a) ConsumeAction

ConsumeAction is an extension of Action abstract class. This method is invoked when the player wants to consume the ConsumableItems in the inventory. Each items consumed have their own effects to the player. Player can obtain capabilities from these items by setting some status to the player. For instance, the player can get an Invincible status once they consume the power star and they will become invulnerable for a period of time. By creating ConsumeAction method, we follow the principle **Single Responsibility Principle**, which defines as, each class should be responsible for a single part or functionality of the system. In this case, ConsumeAction only focuses specifically on the consumption of items process of the player. Instead of having all actions in one Action class, we separate ConsumeAction from other actions, which have their own implementation.

b) Destructible

Destructible interface is created to determines which object can be destroy and what will happen if they were destroyed. When player consumed some items, they will be in a status that it can destroy the ground and drops a Coin. Instead of creating a destructible method in every object on the ground, we created a Destructible interface. Then we invoke the methods in the interface to determine whether when Tree and Wall can be destroyed and when it cannot be destroyed. This follows the principle of **Don't Repeat Yourself** and **Write Every Time**. This can make the code reusable and easier to maintain in the future.

REQ5: Trading

a) 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.

b) 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.

c) 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.

d) 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

a) 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.

b) 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

a) 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.

b) 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.