REQ1

We've built three new classes that extend from Tree Class: Sprout, Sapling, and Mature. We may add new classes and extend from Tree if we like, thus this complies with the Open-closed principle (OCP), which states that software entities (such as classes, modules, and functions) should be open for extension but closed for change.

Furthermore, we created a new actor class named Koopa, as well as a Goomba class, both of which are opponents. Sprout and Mature will produce a Goomba and a Koopa every turn, but each of them has a different spawn rate, therefore we constructed a Util class to assist us acquire a random number so that Sapling and Mature may spawn the foes according to their spawn rate.

Similarly, we constructed a Coin class that specifies the coin's object. We also designed a Wallet class to handle the player's wallet and an AddWalletMoneyAction action class to do the coin pick-up operation. These are in accordance with the Single Responsibility Principle (SPR). Each class is in charge of a certain aspect or function of the system. The Wallet class controls the player's wallet in the same way that our wallet does in real life. If the place where the player is standing holds coins, the AddWalletMoneyAction class is responsible for the player picking up the coin.

REQ2

We've chosen to make a new class called JumpAction that inherits the Action abstract class to meet this need. This stands to reason because the jump action is in addition to all of the other actions a player may conduct in the game. We'll need to update a handful of other classes, especially Wall and Tree, to make the proper things function for this need. More specifically, we created the 'HighGround' interface, which includes an empty method named JumpUpAction(). Since these two objects are considered high ground in the present implementation, the Wall and Tree classes implement this interface.

We specify the final attributes of SUCCESS_RATE and DAMAGE RECEIVED in the Wall class, implying that these values will not change, and then override the list allowableActions list. Similarly, we also decided to set the final SUCCESS_RATE and DAMAGE_RECEIVED values for every sprout, sapling and tree object so we can compare them to the random percentage, and if only they meet their requirements, the jump functionality will take place.

We override the method and set final attributes within the class to avoid violating the Single Responsibility Principle (SRP), which states that each module, class, or function in a computer programme should be responsible for a single part of the program's functionality and should encapsulate that part. As a result, the Utils class, which calculates our probability percentage, will not be liable for all success rates, from sprouting to jumping.

Furthermore, if we add new high ground objects, we can have their classes implement the HighGround interface and call getJumpAction in that object's class to override the allowableActions list in the same way we did previously without having to make any changes to the Jump Action class specifically. This demonstrates that the class follows the Open-Closed Principle (OCP), which states that software components should be extensible but not modifiable.

Finally, we've included a boolean called jumped and a JumpAction called lastJumpAction to ensure that if the player succeeds in jumping up, the player has already jumped up to high ground. As a result, our console will be less cluttered, and it will not recognise and display the player's high ground.

REQ3

Firstly, the player and enemies (Goomba and Koopa) will be created through the Actor abstract class. Goombas will be spawned from sprouts therefore Sprout class has an association relationship with Goomba class. Status enum class is to indicate the current status of the Actors. Player needs a Wrench to destroy Koopa's shell, therefore PLayer class has an association relationship with Wrench class and Wrench class is extends from WeaponItem abstract class.Since players and enemies can attack and have their own type of way to attack, therefore the AttackAction class has an association relationship with Actor abstract class. Since all enemies have their own behaviours, therefore a Behaviour interface is created and is implemented by AttackBehaviour and WanderingBehaviour class. This is Interface Segregation Principle.Whenever a fight is engaged between the player and the enemy, the enemy will follow the player, therefore the FollowBehaviour class is added and it implements Behaviour interface since it is part of the enemies' behaviour. Since player and enemies will also move around, therefore MoveActorAction class is created to enable player and enemies to move around. Therefore, Player class and WanderingBehaviour class will have dependency relationships with MoveActorAction class. Since the location will need to be updated when player and enemies move around, therefore MoveActorAction class will have a dependency relationship with Location class.

If Goomba is unconscious, it will be removed from the map, therefore no additional class is needed. However, when Koompa is unconscious, it will go into dormant state, therefore the display method will display D instead of K, and the rest of the method will change accordingly when isConscious becomes false. When the shell of Koopa is destroyed, it will drop SuperMushroom, therefore Koopa class knows about SuperMushroom class and SuperMushroom class will have dependency relationship with Player class since it will update attributes of Player class.

REQ4

MagicalItem abstract class will extends from abstract Item class. SuperMushroom and PowerStar class will both extend from MagicalItem abstract class. ConsumeAction class is created for the Player to consume magical items. Both PowerStar and SuperMushroom class has method to buff or debuff players. This is the Single Responsibility Principle, this is because SuperMushroom and PowerStars each have their own way to buff and debuff the Player. ConsumeAction class has a dependency relationship with SuperMushroom, PowerStar and Actor class. Wall and Tree class will both become Dirt by being removed from the map and replaced from to Dirt class, and then when an actor that has status INVINCIBLE steps on dirt, it will create a new Coin class that returns a coin when the player steps on the Dirt, and the Player's wallet will increase.

REQ5

We have chosen to create a TradeAction to perform trading that inherits Action abstract class. We instantiate the cost for Wrench, Power Star and Super Mushroom and make it as the final attribute. We'll need to update a handful of other classes, such as Wallet class and Toad class to make the proper things function for this requirement.

In this requirement, I have implemented another menu for trading interaction between toad and player. I have used the scanner library in this class to open up a menu. In this menu, there are options to buy Wrench, Powerstar and SuperMushroom.

We override the method and set final attributes within the class to avoid violating the Single Responsibility Principle (SRP). In execute() method, I have overwritten the execute method

in engine package and decide to code my conditions for the options to buy Wrench, PowerStar and SuperMushroom from Toad.

For each of the conditions, I will have to get my Wallet balance and see if I have enough coins to buy the required item. If the required coins are enough, my wallet balance will be deducted with the method of points.deductBalance(), and the item will be added into my inventory with the method player.addItemToInventory().

REQ6

We have chosen to create a MonologueAction to handle the monologue that will be output if we interact with the Toad. In Toad class, we instantiate a java list to store the monologue that the toad has to output. Then, in Toad class, we will create a method called getSentence to get the required monologue that can be output based on the conditions that are stated in MonologueAction class.

In MonologueAction class, we overwrite the execute method in order to fulfil the Single Responsibility Principle (SRP). We use if else method to give the conditions of monologue that is going to be output. If the player contains a Wrench in his inventory, the index 0

monologue will not be output. If the player is affected by the power of PowerStar, the index 1 monologue will not be output.

REQ7

We created the ResetAction action class, which will reset our game if we push the "r" reset button. Then, in the Status class, we added three enums ("RESET", "RESET GAME",”STOP_SPAWN). RESET is used in the tree and coin classes. If a tree or coin has the capacity to RESET, it will be removed from the map partially (tree) or entirely (coin). One of the player's talents is RESET GAME. When the reset button is clicked, the game will restart with the player's current status and all enemies removed. STOP_SPAWN is to prevent the trees from spawning an enemy or coin at the same time with the reset game turn. Hence, every tree will stop spawning for one turn which is the reset game turn.

As a result, the Single Responsibility Principle (SRP) is satisfied since the resetAction class is one of the action classes and will be in charge of the game's reset function.

In addition, there is a resettable interface class. When the reset button is pushed, any class that has to be deleted or modified will be implemented as resettable. The Dependency Inversion Principle (DIP) asserts that rather than real implementations, we should rely on abstractions (interfaces and abstract classes). Details should not be dependent on abstractions, rather, abstractions should be dependent on details. As a result, we've met DIP since we utilise a resettable class to implement all of the needed classes.

Finally, there is a class named ResetManager that will obtain the instance of resettableList and run the reset programme, which will reset the game.