Design Rationale

Requirement 1: Ali

- Tree will be an abstract which was not the case in assignment 1, with Sprout, Sapling and Mature be abstracted from the tree class, this will help minimise the responsibilities that the tree class will have as all three tree types will have their own unique features.
- We will add a feature class in the tree that's called treeFeatures that will be used in each of the tree child classes to implement their own features.
- We will also implement a function that will represent the tree's age, and age is representing the number of turns the game is going to take. When the play turn has reached a certain amount of turns the age should be invoked accordingly and incremented by 1 for each turn.
- In the Sprout class we will call the treeFeature function that will do all the necessary things that sprout is supposed to do like produce a Goomba if the location does not have an actor in it and with 10 % chance.
- In Sapling the treeFeature function is overridden from the parent class and implements the features of Sapling such as producing a coin in its place so that mario can pick it up with a 10% chance.
- In Mature the treeFeature function will produce Goomba and initialise a new Sprout if the ground is fertilised.

Requirement 2: Sujin

- HighGround abstract class has been created; Tree and Wall class extend HighGround class.
- HighGround did not exist in assignment 1, however, according to DRY principle, Jumping should be implemented by HighGround (including Wall and Tree) only once, not each wall and tree class.
- HighGround implements a Jumpable interface since HighGround (wall and tree) can be jumped by the player. The methods dedicated to jumping.
- Depending on the type of terrain and the status of Tree, each of them has a different jump success rate, and damage (with player.hurt(damage)) when the jump is not successful. Due to this, each of them has getDamage() and getSuccessRate() methods so that jumping is implemented with their own damage and success rate.
- If the status of the player is TALL, which means the player has consumed super mushroom, ‘jump’ action is implemented with a 100% of success rate and a 0 damage regardless of getDamage() and getSuccessRate() in each class.
- JumpAction will be added to the ground unless the status of the player is ‘consumed_powerstar’ in the allowableActions method in HighGround class.
- The player does not get to have ‘jumpAction’ for the ground that player stands on at the moment (if the ground contains any actor (including the player), jumpAction option will not appear in the console).

Requirement 3: Madeline

- Koopa and Goomba are abstracted from the abstract class, Enemy.
- Both of them implement behaviours from the Behaviours interface, being WanderBehaviour, FollowBehaviour and AttackBehaviour.

- For Koopa, I've created two capabilities; KOOPA_ACTIVE status and KOOPA_DORMANT status, which are applied accordingly in their suitable situations
    - KOOPA_ACTIVE serves as Koopa's standard and alive state, where they can roam freely, attack and follow the player when engaged in battle.
    - Once Koopa's HP is reduced to 0, all capabilities will be cleared and replaced with the KOOPA_DORMANT status. Making them immobile on the map and unable to attack the player.
    - The player will be able to attack the dormant koopa as long as they possess the wrench in their inventory. In this case, upon executing AttackAction, a dormant Koopa's KOOPA_DORMANT capability will be removed. Following this, within the Koopa class's execution, if both KOOPA_DORMANT and KOOPA_ACTIVE capabilities don't exist on the current instance of Koopa, it will be removed from the map and replaced with a Super Mushroom - ready for the player's next turn.
- Goomba has a 10% chance every turn to self-destruct. Therefore, a method has been placed in the Goomba class which will generate a random number to be used to determine the Goomba's fate for the next turn. (whether they will live or self destruct, resetting the space to dirt)

Requirement 4: Sujin

- MagitalItem abstract class (extending Item class) will be created; Super Mushroom and Power Star class will extend the Magical Item class.
- In assignment 2, there is no point in having the abstract magical item class as super mushroom and Power Star have different features in Consume Action, however, I made it for the future assignment in case they have common features.
- Magical Item implements Consumable interface as magical items can be consumed by the player.
- Magical items have an attribute of consume action since one magical item can get only one consume action (one item can be consumed only once).
- For Super Mushroom:
    1. Increase max HP by 50 (actor.increaseMaxHP(50))
    2. Add capability of TALL (for the uppercase of displaychar)
    3. Add capability of CONSUMED_SUPERMURHSOOM (if the player got damaged from enemies in the status of CONSUMED_SUPERMUSHROOM, the player lost all of the superpowers and the item will be removed from the inventory)
    4. The Consume Action will be added just once when it is picked up by the player. When it is dropped by the player, it loses the consume action.
- For Power Star:
    1. If the actor steps on the high ground, the ground will be destroyed and converted to coins (value $5).
    2. All the attacks from the enemies will be useless (in the player.hurt() method, if the player has a capability of CONSUMED_POWERSTAR, the player will be damaged by 0).
    3. When the player attacks enemies, the target (the enemies) will be dead in target.resetMaxHP(0); instantly kill them.
    4. If the item is on the ground or in the inventory for more than 10 turns, it will disappear from the game.
    5. After being consumed, the duration of the superpowers is 10 turns.
    6. Counter for the inventory and the other one for the consumer will be reset every time the item gets picked up by the player.

Requirement 5: Madeline

- I've created a class called Wallet, which represents the player's money in the game.
- A PurchaseAction class has been created to manage the player's finances.
    - This involves checking to see whether the player has enough money in their wallet to purchase the selected item. If so, the value of the item will be subtracted from the player's money, and the item will be added to their inventory.
    - In an opposite case, where there's insufficient funds in their wallet, nothing will happen, and the appropriate message will be displayed.
    - Within the Toad class, I've added the available options to purchase items, which will show up in the actionList alongside speakAction

Requirement 6: Ali

- Toad the non playable character and is going to shout out a message if the player speaks to him by implementing an arrayList of messages that he can say.
- He also needs to have a speaking action class which he will use to execute the speaking action to the player in the console.
- The speaking action will require a speakBehavior interface that the class will implement to be able to get the action of speaking.

Requirement 7: Sujin

- Resetting the game requires the changes in Tree class, Enemy class, MagicalItem class, coin class and player class (since they are going to be removed from the game including the related player status).
- For the reset action to be executed, I have made a reset action which triggers run() method to run in the resetmanager class. ResetInstance() method in the resettable interface will be gone through in the run() method of the reset manager class. Every resettable instance will be reset by the ResetInstance() method.
- If the player wants to reset the game, reset action will be implemented and all the related classes to reset action get to have a status of RESETTED.
- To make sure the player implements Reset Action only once, RESETCOMPLETE status will be given to mark the game as resetted. After resetting, the related classes will lose the status of RESETTED since the reset implementation is over.