Design Rationale

REQ1. Let it grow!

Tree is made an abstract class so it acts as a base class which can be extended by other subclasses such as sapling, sprout and mature but the Tree class itself cannot be instantiated.

Spawn Enemy.
Since Sprout and Mature will spawn a Goomba and Koopa respectively, the former classes will have an association to the latter classes with a one-to-one relationship. Each of the Sprout and Mature will contain a Goomba and Koopa attribute as the spawn method in Sprout and Mature can potentially access the attribute, making it a strong relationship.

Growing of trees.
Since the trees (Sapling, Sprout and Mature) have a life cycle, we design it in such a way that they will have an association to the tree of their next cycle. For example, Sprout will have an association with Sapling which will have an association of Mature and Mature will have an association back to Sprout. It is done this way so that each tree will have an attribute of the next tree they will grow into. Dependency is not used as we need to store an attribute of the next tree cycle since a tree would need to know about the next tree it will grow into and access its method to instantiate that tree.

Coin.
As Sapling has a chance to drop a coin, we would just need to create a Coin object. It has an association as Sapling would need to specifically create a $20 coin, thus requiring access to the Coin object’s attribute, therefore making it a strong association.

REQ2. Jump Up, Super Star

JumpAction.
JumpAction has an association with the SuperMushroom class. It will get an instance of the SuperMushroom object to determine the success rate of each jump. It will also access the SuperMushroom attribute to get its capabilities which give the player 100% success rate in jumping as shown in REQ4. Hence it is a strong relationship because the JumpAction method can access this attribute. Since magical items are not the focus of this REQ, the implementation of magical items will be shown in REQ4.

We created a JumpAction in the ‘game’ package that extends MoveActorAction in the engine package which will inherit the methods to move the actor on top of the objects in the Ground class (eg: Sapling, Sprout and Wall) when it performs a jump action. Using the **Dependency Inversion Principle**, we implemented a Jumpable interface which is extended by the Ground, Sapling, Sprout and Wall so that instead of having an arraylist of jumpable high grounds in our JumpAction, we only need an arraylist of type Jumpable in our JumpAction class. So, if we ever need to create a new high ground class that is jumpable, the new class would just need to implement the Jumpable interface. Thus we do not need to modify the source code of the JumpAction class.

REQ3. Enemies

Actors attacks Actors
Since we know that player and enemy extends Actor abstract class and implement the actackedBy method, we would just need an association from AttackAction to the Actor abstract class so that any class object extended from the Actor abstract class can be attacked by each other. This is especially useful such that when a new enemy is added, it just needs to extend the enemy abstract class to apply the entire attack process, hence applying the **open-closed principle** where the enemy class can be extended without modifying the class itself.

Enemy Behaviours
Instead of directly extending the enemy base class to include behaviours, we added a more meaningful interface called Behavior which is extended by AttackBehavior, RetreatBehavior, WanderBehavior and a FollowBehavior. This would make our design more robust whenever we need to use Behavior at other classes such as for a new non-playable character class that is not part of the enemy. This allows any new classes to implement the behaviours as intended without modifying the base class itself. This is a part of the Liskov Substitution Principle when we ensure the robustness of polymorphism.r

Destroying Koopa's shell
Since Mario requires a Wrench to destroy Koopa's shell, we have an association from ActackAction to a Weapon class in order to access the capabilities(attributes) of the specific weapon used. If the weapon is a wrench, it will have the ability to destroy Koopa's shell.

List of Actions
Each actor would have a list of executable actions. Therefore, we made an abstract Action class to apply the open-closed principle. This is done so that a new class would just need to extend the ActionList class to obtain a list of executable actions instead of modifying the code in the class to include dependencies of a list of actions, thus making the ActionList class extendable without any modification.

REQ4. Magical Items

- Each magical item will have one and only one capabilities set.
- The player class will hold the consumed magical item until certain conditions are met. For example, 10 rounds limit has reached, or receive damage. Then the consumed item will be removed.
- The consumed magical item will be active hence we can use the capabilities set from the magical item after consumed.
- One way we can take care of disposal of consumed items is using tickers to count the round for power stars. For super mushrooms, we can add a method in player class to check whether there is an active super mushroom, if yes remove it.
- Some capabilities of the magical item might use actions. For example, the path of gold capability uses dropItemAction to drop coins.
- Each capability will be in at least 1 capabilities set but capabilities set may or may not contain each capability.
- Enemy is included in this UML as well because Koopa can drop super mushroom when destroyed

Clarification of Association and Dependency Used

- **Koopa — dropItemAction — Super Mushroom**

This uses dependency because Koopa instance will not contain dropItemAction class as an instance variable, It will simply use dropItemAction class in its method. dropActionItem instances contain item instances, hence association is used.

- **Items — Player — magical items**

This is a tricky one. Player instance will indeed contain magical items but we considered that as an abstract class item instance. Hence players will have an association with item class. However, inside the player class, we can implement a method that uses a magical item as parameter/ uses it somewhere in the code hence the dependency.

- **Capabilities — capabilities set**

This is obviously an association because the capabilities set will contain some of the capabilities. For example, capabilities set in power star class will contain invincible capability, instakill capability, etc.

- **Capabilities set — item**

Each item instance may or may not CONTAIN a capabilities set. For example, it makes sense for magical items to contain capabilities set because they can do many things. But it does not make sense for items like coins to have a capabilities set because coins do not do much. Hence, association and multiplicity in the diagram is suitable.

- **Enemies — player**

This is an association because enemy attacking players simply notify the player to cancel the super mushroom effect. It does not store the player instance nor does the player store enemy instances.

REQ5. Trading

When player and toad are next to each other, trading can occur. Trading action doesn't really return an object, but instead it returns a permission for the player to buy an item. If insufficient wallet balance, player cannot buy that item, else balance is deducted from getting the item. TradeAction is responsible for deducting the balance while players are responsible for getting the item.

Clarification of Association and Dependency Used

- **Toad — tradeAction — player**

1 condition is needed to trigger trade action. ‘Toad’ needs to be beside the player. But trade action class does not contain toad and player. Hence, In our diagram, we include toad and player as a dependency to trade action. Trade action will also need to decide whether the player has enough balance for an item, hence a dependency of player to trade action is needed.

- **tradeAction — sellable items**

This is a dependency relationship because trade action will not contain these items objects inside the class, it will only use magical items objects as method parameters to query for price.

- **tradeAction — wallet**

This is a dependency as well because tradeAction will deduct balance from wallet but wallet is not an instance variable of trade action class.

- **Player— pickUpAction — coin — wallet**

Player to pickUpAction is dependent because the player will only call the method but not store the instance. pickUpAction to coin is an association because it will store the item instance inside. Coin to wallet is an association because the waller does not actually store coins (coins have no meaning, only the wallet balance matters), picking up coins will increase wallet balance but not storing it inside wallet.

- **Player — wallet**

This on the other hand is an association because the player will hold 1 wallet. (Stored inside the player instance)

<u>REQ6. Monologue</u>

Toad will speak a total of four different sentences which is dependent on what item the player holds (if any). The four different sentences are stored in the monologue class.

<u>Clarification of Association and Dependency Used</u>

- **Toad — MonologueAction — PowerStar**

There is an association between Toad and MonologueAction class as an instance of Toad will be implemented in MonologueAction. If the player has a power star active, the Toad must not print out the second sentence “You better get back to finding the Power Stars.” . This would require a dependency in our diagram between PowerStar class and MonologueAction.

- **Toad — MonologueAction — Wrench**

There is an association between Toad and MonologueAction class as an instance of Toad will be implemented in MonologueAction. If the player has a Wrench and interacts with the Toad, the Toad must not print out the first sentence “You might need a wrench to smash Koopa's hard shells.” . This would require a dependency in our diagram between Wrench class and MonologueAction.

- **MonologueAction — Monologue**

This is an association between MonologueAction and Monologue as MonologueAction class will create instances of Monologue class.

REQ7. Reset Game

Based on our UML class diagram, in order to reset the game, we will implement four abstract classes (Action, Actor, Ground, and Item). Each abstract class will be extended based on the requirements as will be explained below.

Clarification of Association and Dependency Used

- **Player - ResetAction - ResetManager - Resettable <Interface>**

When a player clicks on 'r' and activates resets, it will activate ResetAction via an Association which then activates the ResetManager via another Association. ResetManager is implemented by the Resettable Interface.

- **Trees have a 50% chance to be converted back to Dirt**

In order to reset the trees (Sprout, Mature, and Sapling) into Dirt, ResetManager which connects to the tree classes via an association will destroy them. Each of the tree classes (Sprout, Mature, and Sapling) are connected to Dirt class via an association as they may implement an instance of Dirt class to be converted to dirt.

- **All enemies are killed.**

Koopa and Goomba classes will each have an association with the ResetManager class as they will implement a method which will kill the enemies.

- **Reset player status**

This is an association between Player class and ResetManager. ResetManager will reset the status of players.

- **Heal player to maximum**

This is an association between Player and ResetAction class, ResetAction class will have an attribute of the ResetManager class (an association relationship which is shown in UML class diagram).

- **Remove all coins on the ground (Super Mushrooms and Power Stars may stay).**

This is an association between the Coin class and the ResetManager class, this is to reset the coin count. The ResetManager class has an attribute of type Coin in order to reset the coin count.