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 as 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 use 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, it would need an association to the DropItemAction in the engine class to access its drop item method in order to drop a coin at the Sapling object's location.

REQ2.

REQ3.

REQ4. Magical Items

- Each magical item will have one and only one capabilities set.
- Player will store the capabilities set of magical items temporarily when consumed.
- Note that this stored capabilities set from magical items will disappear if a certain condition is met. For example, super mushroom capabilities set in player will disappear when player receives damage, power star capabilities set in player will disappear after 10 rounds, etc.

- Some capabilities of magical item might use actions. For example, 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 class will not contain Super Mushroom class as well, it might return the super mushroom object in its method hence dependency is used.

- Items — Player — magical items

This is a tricky one. Player instance will indeed contain magical items but we considered that as abstract class item instance. Hence player will have an association with item class. However, inside player class, we can implement a method that use 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 make sense for magical item 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 does not do a lot of things. Hence, association and multiplicity in the diagram is suitable.

REQ5. Trading

When player and toad are next to each other, trading can occur. Trading action doesnt really return an object, but instead it return a permission for 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 of deducting the balance while player are responsible of getting the item.

Clarification of Association and Dependency Used

- Toad — tradeAction — player

1 condition is needed to trigger trade action. Toad need 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 object inside the class, it will only use magical items object as method parameter to query for price.

- tradeAction — wallet

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

- Player— pickUpAction — coin — wallet

These 3 relationships are dependency because the player does not really need to store coins inside its class (coins has no meaning, only the wallet balance matters). But picking up coin means we will increase wallet balance. None of the instance here need to store instance of its adjacent class.

- Player — wallet

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

REQ6.

REQ7.