Requirements 1-3 Rationale

Requirement 1:

- Tree abstract class
    - Abstract class tree encapsulates the commonalities of all the tree phases, following OOP principles and SOLID
    - Changes from initial design: inherits from high ground abstract class, this allows the implementation of other requirements like power star. This may be a bit of a deep inheritance but as e.g., a mature is tree that is also a high ground, therefore it doesn't violate Liskov principle.
- Sprout
    - This class has a dependency on sapling as it can grow into (create) a sapling. The turns will be determined by overriding tick(), increasing the age as the tick method is called, and once the age reaches 10, it will create a sapling using location.setGround().
    - Similarly, it has a dependency to goomba as it can instantiate a goomba. This would also be implemented in tick() with the use of the general use Utils class static nextInt(100) method and to create a goomba.
    - Preconditions: is a tree
    - Postconditions: a sprout object
    - <u>Changes from initial design</u>: Usage of Utils class instead of random.nextInt(100), this avoids the use of having to declare a Random instance variable or local variable, abiding with ReD.
- Sapling
    - Similar to sprout, sapling has dependency with mature as it can grow into a mature, using the same implementation as sprout to become sapling.
    - It also has a dependency with coin as it can drop a coin, using similar implementation as sprout spawning goomba.
    - Preconditions: is a tree
    - Postconditions: a sapling object
    - <u>Changes from initial design</u>: same as Sprout
- Mature
    - Mature spawning sprout will be implemented in tick method where once the age % 5 == 0, it will look in its surroundings for fertile ground (which would have a Status.FERTILE, only Dirt right now) and grow one in that turn. To assess its surroundings it would get the mature's location exit list and use Util.nextInt() with a bound of the list size to randomly look for fertile ground and instantiate 1 sprout each turn.
    - To spawn a koopa, it would again be implemented in tick() similarly to sprout can spawn goomba, but also be calling containsAnActor from location to see if an actor is already standing on it.
    - The last dependency is with Dirt as it can wither and die, this is implemented similarly to spawning goomba as well, just with 20 and not 10.
    - Preconditions: is a tree
    - Postconditions: a sapling object
    - <u>Changes from initial design</u>: same as Sprout

Requirement 2:

- Grounds that allow jumps

        - Jumpable grounds like trees and walls will override allowableActions and JumpAction. To prevent enemies from jumping, it would first check if that actor has Status.HOSTILES_TO_ENEMY. To add each ground's own success rate and damage, the jumpAction would have damage and successRate attributes, and instantiation would allow each ground to pass their own success rate and damage in constructor. This allows SRP as the jumpAction class would only need to worry about executing the action and not each grounds' different success rates and damages.
    - JumpAction
        - It associates with Location as the actor needs to move to the location of the ground being jumped over
        - JumpAction.execute() is demonstrated in the sequence diagram
            - Pre-conditions: the ground allows a jumpAction; actor is not an enemy
            - Post-conditions: outcome of the jump
        - <u>Changes from initial design:</u> Added a direction instance variable for better game experience as coordinates can be confusing. Also added a hotkey instance variable so that the hotkey of that ground's direction in the player's menu would have the same key for the player as usual.

Requirement 3

- Enemy abstract class
    - An abstract class called enemy is created to capture the similarity of the enemies and minimise redundancy. As enemies all have a behaviour list attribute to determine their actions and have appropriate methods for them, this use of a deeper inheritance is justified.
    - To allow the player to attack: Enemy class overrides allowableActions and along with adding attackAction for player (actor with Status.HOSTILE_TO_ENEMY
    - To attack the player (first or not) and follow them: Enemy class has a method called findTarget where it would assess its exits for the player, if found it would attack the player by adding a new AttackBehviour and FollowBehaviour as highest priorities.
    - Preconditions: is an actor
    - Postconditions: an enemy object
    - <u>Changes from initial design:</u> Instead of adding attack and follow behaviours in allowableActions, findTarget() allows enemies to not only fight but also attack first.
- Goomba
    - Goomba is an enemy, therefore inherits the Enemy class. This follows Liskov principle as goomba is also an actor. Its 20 HP would be defined in the constructor. Meanwhile, since its weapons are intrinsic, Goomba class would just override getIntrinsicWeapon to create its own unique IntrinsicWeapon.
    - To add its 10% suicidal behaviours, this could be done through adding this in an overridden playTurn().
    - Preconditions: is an enemy and actor
    - Postconditions: a goomba object
    - <u>Changes from initial design:</u> None
- Koopa

    - Koopa is also an enemy and actor, and therefore extends the Enemy class. Its HP points and intrinsic weapon will also be designed similarly to Goomba.
    - To allow its dormant state when unconscious, the koopa class overrides the hurt() method and changes the koopa's display char to 'D', gives it a DORMANT Status and heals its HP by 1. This allows the Koopa to immediately enter a dormant state when attacked to consciousness and not be removed from the map. As it cannot do anything in this state, the playTurn will return a DoNothingAction if the Koopa has its DORMANT state.
    - Preconditions: is an actor
    - Postconditions: a koopa object
    - <u>Changes from initial design:</u> Removal of DormantBehaviour class as its sole purpose is to return DoNothingAction which could just be done through directly returning a DoNothingAction in playturn. Instead of isConscious(), the Koopa enters D mode from hurt() instead which allows it to heal by 1 HP so that it wouldn't be removed in AttackAction. This helped remove some if-else in attackAction where it has to deal with dormant actors which promotes SRP.
- Wrench
    - The wrench would inherit WeaponItem as it's an Item that can be used to destroy Koopa's shell. Since it can give the player a capability to destroy koopa's shell, it would get a Status.DESTROY_SHELL when instantiated.
    - Preconditions: is a WeaponItem
    - Postconditions: a Wrench object
    - Changes from initial design: None
- Destroying Koopa's shell
    - If Koopa has its GO_DORMANT status and a wrench (the item with Status.DESTROY_SHELL) exists in the player's inventory, the allowableActions() method in Koopa would add an AttackAction to the return list. This would reduce the redundancy of another attack-like action (e.g., DestroyShell), following DRY.
    - To drop a Super Mushroom, the mushroom is added to the Koopa's inventory since it's controlled by behaviours and is prevented from jumping on walls anyways. Therefore when it gets killed, the dropItems() method in AttackAction would simply drop its Super Mushroom on its location.
    - <u>Changes from initial design:</u> Instead of creating a dependency from AttackAction to SuperMushroom as it would make more sense for Koopa to depend on Super Mushroom, it simply adds the mushroom to the Koopa's inventory anyways, following both DRY and ReD.