Requirement 1:

- Abstract class tree
    - Since trees all have similarities like their age, and ability to be jumped over, tree concrete is converted into an abstract class to allow 3 different types of trees (sprout, sapling and mature) to extend from it. This allows abstraction and the DRY rule. The tree abstract class has an attribute called age to facilitate growth.
- 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 spawn a sapling. Although this type of dependency is not needed to be shown, it was added to reflect the physical growth from sprout to sapling, something crucial in this requirement.
    - Similarly, it has a dependency to goomba as it can instantiate a goomba. This would also be implemented in tick() with the use of random.nextIn(100) <= 10 to create a goomba.
- 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, this will be implemented in the same way as sprout spawning goomba.
- Mature
    - Mature has three dependencies, one being with sprout as it can spawn a new sprout every 5 turns. This 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 use the way the processActorTurn() assesses an actor's surroundings but instead, using random.nextIn() with a bound of the location's exit list, and randomly look for fertile ground and instantiate 1 sprouts each turn.
    - The other dependency is with Koopa class as it can spawn a koopa. This would again be implemented in tick() similarly to sprout can spawn goomba, but would 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.

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 damage.

- JumpAction
    - As demonstrated in the sequence diagram of the execute(), the success of the jump would be changed into a 100%, if the actor has a Status.JUMP_WELL, from consuming a super mushroom (an Item). But, otherwise, it would use the random.nextInt(100) to set the jump success. If the jump succeeds, it would move the actor to the location of the location of the ground and return a message indicating the jump was successful. Otherwise, reduce the player's HP by the amount of fall damage and return a message that the jump was unsuccessful.
    - The jumpAction has an association with Ground. As the jumpAction happens to a ground, it needs a target of type Ground as an attribute.
    - It also associates with Location as the actor needs to move to the location of the ground.

Requirement 3

- Enemy abstract class
    - An abstract class called enemy is created to capture the similarity of the enemies and minimise redundancy. While this may seem like deep inheritance, as enemies all have a behaviour list attribute to determine their actions and need to have methods like getBehaviours() and similar traits such awander behaviour, this use of a deeper inheritance is justified.
    - To attack player and follow them after: Enemy class would override allowable actions and along with adding attackAction for player (actor with Status.HOSTILE_TO_ENEMY) to execute, it would also put a new AttackBehaviour for the enemy as first priority and new FollowBehaviour as second.
- 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 feature in an overridden playTurn().
- Koopa
    - Koopa is also an enemy and actor, and therefore extends the Enemy class. Its HP points and intrinsic weapon to attack will also be designed similarly to Goomba's HP and weapon.
    - As enemies have behaviours to return actions, a DormantBehaviour class is created and it returns the DoNothingAction as Koopa cannot follow,wander or attack in its dormant state. DormantBehaviour could also extend Action class and simply return itself, however, that would violate DRY principle as it would be very similar to DoNothingAction.
    - To allow its dormant state when unconscious, the koopa could have a goDormant method that changes its display char, clears its behaviour list. And to prevent the map to remove it when its HP is 0, Koopa would have a Status.Go_Dormant when instantiated, which would be used to prevent this in AttackAction. To allow Koopa to enter dormant stage, the isConscious method called in AttackAction would be calling Koopa's overridden isConscious(), following Open-Closed principle.

- 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.
- DestroyShellAction
    - The DestroyShellAction class is designed as the player cannot attack the Koopa once it's in its shell, it would need special action to destroy its shell with a Wrench. Therefore, DestroyShellAction extends from Action, following abstraction and SRP. Its implementation would be similar to AttackAttack.execute() however, since Wrench can destroy Koopa’s shell in one go if successful, it wouldn’t need to check if Koopa is conscious. Despite a level of repetition in DestroyShellAction and AttackAction, a generalisation to the latter would violate Dependency Inversion Principle.
    - If Koopa is unconscious and a wrench (the item with Status.DESTROY_SHELL) exists in the player’s inventory, the allowableActions() method in Koopa would add a DestroyShellAction to the return list.