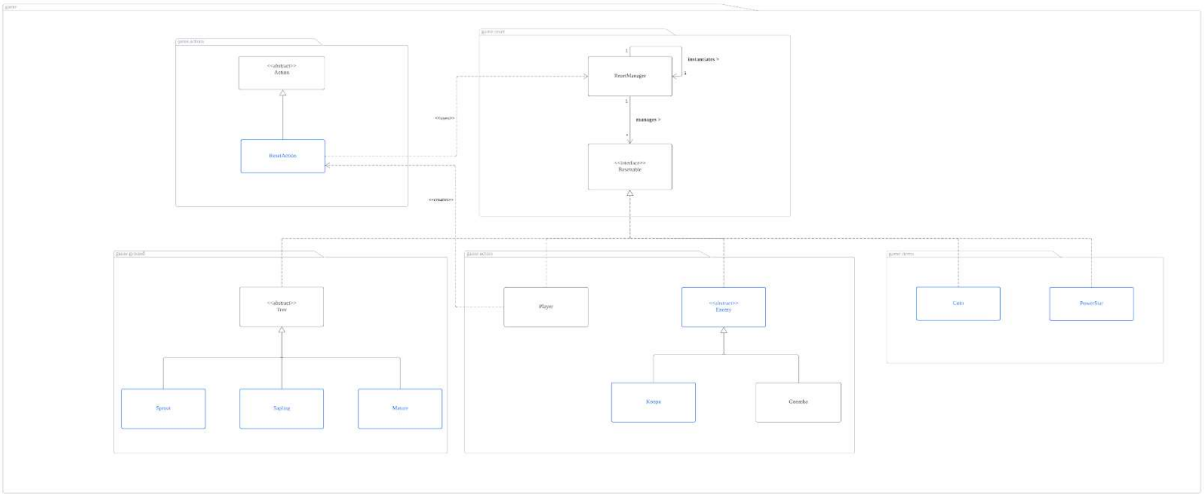
REQ7

Class Diagram:



Requirement:

- Add in an option for the player to reset the game
- Game can only be reset once
- Trees have a 50% chance to be converted back to Dirt
- All enemies are killed
- Reset player Status
- Heal player to maximum
- Remove all coins from the ground

Design Rationale:

In requirement 7, we would utilise the existing **Resettable** interface to reset all the things that stated in the requirement. To elaborate more on this, we would implement the Resettable interface in the abstract **Tree** class, abstract **Enemy** class, **Player** class, **PowerStar** class and the **Coin** class. We choose to implement the interface in the abstract Tree and Enemy class because all the subclasses should be reset if the reset option is chosen. Besides, in the constructor of each concrete class that implements the interface, we would use the method `this.registerInstance()` to register them to the ArrayList of type **Resettable** in the ResetManager class to simplify the reset process. The use of ArrayList of Resettable follows the **Dependency Inversion**

Principle (DIP) since the High-level modules (**ResetManager** class) does not depend on low-level modules such as **Player** class, **Coin** class and more. When we add in more classes that need to be reset, we do not need to modify the **ResetManager** class but just implement the **Resettable** interface in that class and the method in its constructor, and it will automatically be part of the reset cycle.

Moreover, in each of the classes that implement the **Resettable** interface, all of them will need to override the `resetInstance()` method and use `addCapability()` method to add the `Status.RESET` to all the **resettable** objects except **Player.** The actual resetting is done in the tick or playTurn method for all **resettable** objects except **Player.** The explanation is broken down to different sections:

Reset for all the Trees

We would override the `resetInstance()` method in the **Tree** abstract class, and would not override it again in the subclasses since all of them will have the same code implementation for this method, which has a 50% chance to be converted back to Dirt. This is an implementation that follows the **Don't Repeat Yourself Principle (DRY).**

The actual execution of reset happens in the `tick()` method as below:

1. When `tick()` method in subclasses of Tree is called, they will call the `super.tick()`.
2. In the `tick()` method, we would check whether the instance of Tree has the capability of RESET by using `this.hasCapability(Status.RESET)`. If yes, it will proceed to step 3, else it will continue the normal `tick()` routine.
3. It will check whether the **Tree** object meet the 50% chance of dying by using `if ((rand.nextInt(100) <= treeDieChance))`
4. If true, a new instance of **Dirt** will be created and set the ground to dirt (the Tree object has died). If false, it will continue with the next step.
5. It will remove every **Tree** object's capability of RESET by using `this.removeCapability(Status.RESET)` so that they will not be reset anymore.

We also added another layer of checking in the `tick()` method of Sprout, Sapling and Mature class to check whether the ground type of the current location is equal to the respective class. This implementation is needed to avoid the instance of these classes

executing the unique spawning ability after resetting since the `tick()` method will still continue to run for the current turn even after it has been removed from the current location

All enemies are killed

We would override the `resetInstance()` method in the **Enemy** abstract class. If the method is called, it will add the capability of RESET to the instance of the Koopa and Goomba.

The actual execution of reset will happens in the `playTurn()` method for both of the classes:

1. In their `playTrun()` method, it will firstly check if the instance of the class (**Koopa** or **Goomba**) has the capability of RESET or not.
2. If true, it will return a **SuicideAction** and it will be removed from the map when the action get executed.
3. If false, it will continue to perform normal `playTurn()` routine.

Reset player status

We would override the `resetInstance()` method in the **Player** class and **PowerStar** class.

The actual implementation of resetting will be implemented in the `resetInstancce()` method for **Player** and in the `tick()` method for **PowerStar**.

Player Class

1. When `resetInstance()` method is called, it will first reset player status by removing their capabilities from super mushroom using the following code
    a. `this.removeCapability(Status.SUPER)`

`PowerStar` Class

1. When `resetInstance()` method is called, it will check whether the PowerStar instance has the capability of INVINCIBLE.
2. If true, it will add the RESET capability to it
3. If false, it will just continue its normal operation

4. During the tick() method of the PowerStar instance, it will removed the PowerStar objects from the player's inventory if it has the RESET capability by using the following code,
    a. If (this.hasCapability(Status.RESET)) - for checking purpose
    b. consumedBy.removeItemFromInventory(this) - removing it from the means removing resetting the player's status.

Heal player to maximum

In the same resetInstance() method in the Player, we will add a few codes to reset the player's health.

1. It will heal the player to its maximum health by using `this.heal(this.getMaxHp())`
2. Lastly, we will remove the RESET capability of player by using this.removeCapability(Status.RESET) so that it will not be reset in the following turns again.

Remove all coins on the ground

We would override the `resetInstance()` method in the **Coin** class. If the method is called, it will add the capability of RESET to the instance of the **Coin** class.

The actual execution of reset will happens in the `tick()` method,

1. Firstly, it will check if the **Coin** instance has the RESET capability by using `this.hasCapability(Status.RESET)`
2. If true, it will remove the coin from its current location by using `currentLocation.removeItem(this)`.
3. If false, it will continue with normal execution sequence.

Furthermore, we would create a **ResetAction** class that is the subclass of **Action** which aims to provide the player a reset option in the menu. The implementation of ResetAction class obeys the **SRP** as this class has only one responsibility, which is to reset the whole game.

The actual execution of ResetAction will be as the following,

1. In the `execute()` method of ResetAction, it will triggers the resetting of all **Resettable** objects by using `ResetManager.getInstance().run()` which loop through all the instance of **Resettable** and triggers their `resetInstance()` method.
2. The player will be reset instantly once the resetInstance() method has been called and a RESET capability will be added into the other **Resettable** objects and execute the actual resetting in their `tick()` or `playTurn()` method.
3. After that we will use ResetManager.hasbeenReset() to set the attribute "ableToReset" to false so that the game cannot be reset again.
4. Lastly, return a String message to indicate the game has been reset.

We have also added a new static boolean attribute called **ableToReset** for **ResetManager** class that keep track of the reset status of the whole game, i.e., whether it has been reset before or not. ableToReset will be true as default and it will be changed to false through a method called `hasBeenReset()` after the **ResetAction** has been executed once.

To provide the reset action as one of the allowable actions to the player, we will check can we reset the game through `ResetManager.getAbleToReset()`. If it returns true, it will add ResetAction as one of the allowable actions, else it will continue to continue with normal checking.

Once the reset action has been executed, we would change the ableToReset attribute to false so that it will not be an allowable action anymore. Hence, the requirement of resetting the game once has been fulfilled.

*Updates that have been done for REQ7 (Class diagram & Design Rationale)

I. Class Diagram:
   A. Added in a new **PowerStar** class that implements the **Resettable** interface

II. Desgin Rationale:
   A. Added in detail code implementation explanations of resetting for each class that needs to be reset

B. Added in new explanations on how the reset action starts and triggers all the reset implementation
C. Reimplement the resetting of player's status for the **PowerStar** effect
    1. Reduce the responsibility of **Player** to reset the his status
    2. So that it is extensible in the future where more magical items with fading effect is added in, the player does not need to keep checking whether the item in the inventory has the capabilities to remove it.
D. Updated the implementation of tracking whether the game has been reset or not
    1. Older version – Handle by **Player** class
    2. Newer version – Handle by **ResetManager** class
    3. The update in implementation is to reduce the responsibility of **Player** handling the status of resetting. It should be done by **ResetManager** since it is a class that is meant for handling stuff related to resetting and the implementation also fulfilled the **SRP**.