REQ1: Let it grow!

Overview:

To achieve this feature, there will be six new classes (i.e., Koopa, Mature, Sapling, Sprout, Enemy and Coin) created in the extended system, and three existing classes (i.e., Status, Tree, and Dirt) will be modified. The design rationale for each new or modified class is shown on the following pages. Please note that although Tree class extends HighGround (i.e., a new class), however, the purpose of creating HighGround is to achieve the feature in REQ2, therefore HighGround class will not be considered in the design rationale and UML diagrams for REQ1.

1) Tree class

What changed in the design between Assignment 1 and Assignment 2 and Why:

In assignment 1, I did not create Mature, Sapling, Sprout class for each stage of the Tree, instead I only used the Tree class to handle all properties of each stage of a Tree. To keep track of the status of a Tree instance, I use the constants in enum class. As a result, there are too many if-else statements, repeated code, overused of addCapability, removeCapability and hasCapibility methods. This makes my code too complicated and less maintainable and readable. Therefore, in assignment 2, I had decided to make this Tree class to be an abstract class. The reason to make this class abstract is because we should not create any instance of Tree as Tree class is only used to provide a common, implemented functionality for its subclasses. After that, I will create the Mature, Sapling, Sprout class to be the subclass of Tree class. This approach is better since each stage has a unique spawning ability and their properties are different (e.g., display character, success rate of a jump, fall damage etc), hence it is better to have different classes to handle this situation. Besides that, by doing so, whenever any of these subclasses have common code, I can avoid repeated code by putting that piece of code in their parent class (i.e., Tree class).

Why I chose to do it that way:

With this design, we are following the Open Closed Principle as when more stages of a Tree are introduced to the game, we can let that new class extend this class without modifying any existing code. Hence, we are allowing our class to be open for extension and close for modification. Besides that, we also create a class attribute to store the fixed value. Hence, we are adhering to the “avoid excessive use of literals” principle. Moreover, we are adhering to the DRY principle as all the subclasses of Tree class will not have any repeated code. In addition, we are following the Reduce Dependencies Principle as in REQ7, when reset action is called, we can remove the Tree instance (i.e., with 50% chance) by directly letting the Tree class implement Resettable. Since we do not need to let any of these subclasses implement Resettable, we are reducing the dependencies in our design.

2) Sprout class

What changed in the design between Assignment 1 and Assignment 2 and Why:

In assignment 1, instead of creating a Sprout class to indicate a tree in the map to become a sprout tree, I add the capability SPROUT to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class.

Therefore, in assignment 2, I will create a Sprout instance if a tree in the map becomes a sprout tree at that location. Details to keep track of the age of a sprout is shown in the code.

Why I chose to do it that way:

By doing so, we are adhering to the Single Responsibility Principle as the method within Sprout class only shows the properties of a Sprout. Furthermore, for each of the properties of a Sprout instance (e.g., success rate of a jump, display character, the rate to spawn a goomba and so on), I store them as private constant class attributes. Consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Sprout class still preserve the meaning from its parent class, Tree.

3) Sapling class

What changed in the design between Assignment 1 and Assignment 2 and Why:

Like sprout class, in assignment 1, instead of creating a Sapling class to indicate a tree in the map to become a sapling tree, I add the capability SAPLING to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class. Therefore, in assignment 2, I will create a Sapling instance if a tree in the map becomes a sapling tree at that location. Details to keep track of the age of a sapling is shown in the code.

Why I chose to do it that way:

Similarly, we are adhering to the Single Responsibility Principle as the method within Sapling class only shows the properties of a Sapling. Furthermore, for each of the properties of a Sapling instance (e.g., success rate of a jump, display character, the rate to spawn a coin and so on), I store them as private constant class attributes. Consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Sapling class still preserve the meaning from its parent class, Tree.

4) Mature class

What changed in the design between Assignment 1 and Assignment 2 and Why:

Similar to other subclasses of Tree class, in assignment 1, instead of creating a Mature class to indicate a tree in the map to become a mature tree, I add the capability MATURE to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class. Therefore, in assignment 2, I will create a Mature instance if a tree in the map becomes a Mature tree at that location.

Why I chose to do it that way:

Likewise, we are adhering to the Single Responsibility Principle as the method within Mature class only shows the properties of a Mature. Furthermore, for each of the properties of a Mature instance (e.g., success rate of a jump, display character, the rate of wither in every turn and so on), I store them as private constant class attributes. Consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable.

Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Mature class still preserve the meaning from its parent class, Tree.

5) Enemy class

Enemy class is an abstract class that represents the enemies in this game. However, since Enemy class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ3) and in this REQ, it only creates the instance of Enemy's subclasses without considering any features, therefore the design rationale for Enemy class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Enemy class.**

6) Koopa class

Koopa class is a class that represents one of the enemies in this game, which is Koopa. However, since Koopa class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ3) and in this REQ, it only creates the Koopa instance without considering any features of a Koopa instance, therefore the design rationale for Koopa class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Koopa class.**

7) Coin class

Coin class is a class that represents the coin item in this game. Similar to Koopa class, Coin class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ5) and in this REQ, it only creates the Coin instance without considering any features of a Coin instance, therefore the design rationale for Coin class will not be available in this section. **Hence, please go to the REQ5 section in the following pages to see the design rationale for Coin class.**

8) Dirt class

What changed in the design between Assignment 1 and Assignment 2 and Why:

No changes are made between Assignment 1 and Assignment 2.

9) Status class

What changed in the design between Assignment 1 and Assignment 2 and Why:

In the previous assignment, I created a class GroundStatus to store the constant IS_FERTILE. However, I realized the GroundStatus class is redundant as I could simply put IS_FERTILE in the status class. Hence, by doing so, I am able to avoid creating purposeless classes.

Why I chose to do it that way:

According to the assignment specification for this requirement, it says that for each Mature instance, for every 5 turns, it can grow a new sprout in one of the surrounding fertile grounds and the only fertile ground currently is dirt. However, it means that we need to figure out a

way to check if the surrounding ground is a fertile ground, otherwise based on the game logic, we should not spawn the sprout if it is not a fertile ground. Hence, I had decided to add this IS_FERTILE constant in Status class. Hence, whenever we initialise a fertile ground, we can add this constant to its capability to indicate that it is a fertile ground. Thus, if we want to grow a sprout on a particular ground, we can determine if it is a fertile ground by checking whether it has the capability IS_FERTILE.

In this case, we are using constant to specify that a Ground is a fertile ground. By doing so, we can avoid excessive use of literals as we do not have to create any local attributes within Dirt class for example to indicate that all Dirt instances are considered as fertile ground. This implementation also provides us the flexibility for future development. For instance, if there are more types of fertile ground that extends Ground class introduced to the game, we could simply add this constant as the capability of it. We also adhere to the Single Responsibility Principle as this enumeration class is only responsible to store the constant that will be used during our implementation.