REQ1: Let it grow!

Overview:

To achieve this feature, there will be four new classes (i.e., Koopa, TreeStatus, GroundStatus and Coin) created in the extended system, and two existing classes (i.e.,Tree and Dirt) will be modified. The design rationale for each new or modified class is shown on the following pages. However, the explanation of the design rationale for Tree class will be the last among all six classes. The reason is that the implementation of Tree uses all other classes, hence I would need to explain the design rationale for other classes first so that the marker can understand the implementation of Tree better.

1) TreeStatus class

Note: the design rationale for the TreeStatus class in REQ1 will be the same as the one in REQ2 as they both refer to the same class with the same functionality.

TreeStatus is a new enumeration class with three constants that represents the three stages of a Tree, which are SPROUT, SAPLING and MATURE.

Description of constant:

- A Tree instance has a status of SPROUT if its age is below 10.
- A Tree instance has a status of SAPLING if its age reaches 10.
- A Tree instance has a status of MATURE if its age reaches 20.

Why I chose to do it that way:

Since there are three stages for each Tree instance, hence I had decided to create a new enumeration to keep track of the stages for each Tree. By doing so, we are simplifying our code as within the Tree class, we do have to use attributes to keep track of it. This also makes our code more readable. Another reason is that the stages of a Tree are constant, hence there is no point of using any class attributes to keep track of the stages when we can use enumeration class to do so.

Advantages:

The purpose of creating this new enumeration class is that we can avoid excessive use of literal within the Tree class to check the tree's status. This implementation also provides us the flexibility for future development as we can simply add new tree status when there is any in the future. Besides that, we also adhere to the Single Responsibility Principle as this enumeration class is only responsible for storing the three possible stages of a Tree instance.

Disadvantages:

In future, if there are more stages for a Tree instance, it might confuse when debugging.

2) GroundStatus class

GroundStatus is a new enumeration class with only one constant that represents the status of ground, which is IS_FERTILE.

Description of constant:

- A ground has a status of IS_FERTILE if it is considered as a fertile ground.

Why I chose to do it that way:

According to the assignment specification for this requirement, it says that if a Tree instance is mature, then 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 create this new class with IS_FERTILE constant. 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.

Initially I was thinking of creating an interface (e.g., FertileGround) and using the static factory method (e.g., FertileManager) to store all the fertile ground in the array list. However, I realized that this would overcomplicate our code when we need to randomly grow the sprout and increase the time complexity as we need to search through the array list to find a specific fertile ground instance. Therefore, using this enumeration class would definitely simplify my code and make it more efficient.

Advantages:

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 possible statuses that a Ground can have.

Disadvantages:

N/A

3) 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.**

4) 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.**

5) Dirt class

Dirt is a class that represents the dirt in the Game Map. It is a class that extends from the Ground. In this class, I had only modified the constructor of the Dirt class.

Description:

- In the constructor of the Dirt class, it will call the parent's constructor by inputting it's display character. After that, it will add the capability of IS_FERTILE to itself to indicate that Dirt is a fertile ground.

Why I chose to do it that way:

Based on the specification for this REQ, only fertile ground is possible to grow sprout. Therefore, by adding this capability, it will be very useful for us to determine whether a ground is a fertile ground and perform specific tasks to it.

Advantages:

With this design, we are adhering to the Single Responsibility Principle as the method within Dirt class only shows the properties of a Dirt and what an actor can do to the Dirt. Moreover, we also fulfil the Liskov Substitution Principle as all the methods in Dirt class still preserve the meaning from its parent class, Ground.

Disadvantages:

N/A

6) Tree class

Note: the design rationale for Tree class in REQ1 will be the almost same as the one in REQ2 as they both referring to the same class, but the design rationale will focus on different aspect (i.e., REQ1 focus on what will happen when a Tree is in a certain stage, whereas REQ2 is focus on Player can jump to a Tree), although there will still be some repetitive content.

Tree is a class that represents the tree in the Game Map. It is a class that extends from the Ground. According to the assignment specification, a Tree has three stages, which are sprout, sapling and mature, hence there will be a significant amount of class attributes in this class.

private int age = 0;
private static final char *SPROUT_CHAR* = '+';
private static final char *SAPLING_CHAR* = 't';
private static final char *MATURE_CHAR* = 'T';

The meaning of the constants above are known by looking at the name of it.

In addition, Tree class has overridden 1 method which are tick and create two new methods, which are isSapling and isMature.

Description of methods:

- isSapling will return True if age reaches 10.
- isMature will return True if age reaches 20.
- tick method is called once per turn. For each turn, it will increment the age attribute by 1. After that, it will check if the age of this Tree instance reaches 10, if yes, it will remove the capability of SPROUT (these constants were introduced in TreeStatus class) and add the capability of SAPLING to indicate that it is now a sapling tree. Similarly, if the age of this Tree instance reaches 20, then it will remove the capability of SAPLING and add the capability of MATURE to indicate that it is now a mature tree. After updating the status of this Tree instance according to its age, it will now check the current status of this Tree and perform different tasks. If this Tree has a status of SPROUT, then it might have a 10% chance to create a Goomba object on that location only if there is no actor standing on that Tree in every turn. If this Tree has a status of SAPLING, it will have a 10% chance to create a Coin ($20) instance on that location at every turn. If the tree has a status of MATURE, then it will have 15% chance to create a Koopa object on that location only if there is no actor standing on that Tree in every turn, have 20% to wither and die in every turn (i.e., become Dirt) and grow s new sprout in one of the surrounding fertile grounds randomly for every 5 turns. If there is no available fertile ground, then it will stop spawning sprouts. In order to get the surrounding fertile ground, we would need to use the exits of that particular location. So, for each exit, we will get that ground instance and check if it has the capability of IS_FERTILE. After that, it will grow a sprout among all available fertile ground.

Why I chose to do it that way:

As you can see from above, we know that for each status of a Tree instance, we will need to perform different tasks, hence by using the constant in TreeStatus, it will help us to keep track of the status of trees efficiently. In previous pages, I had mentioned that the IS_FERTILE will help us to determine whether a ground is a fertile ground and hence this constant will be used when we need to randomly grow a sprout on its surrounding ground when the tree is mature. With this constant, we do not need to use “instanceOf” to check what type of ground it is. This is actually a good design practice as when more and more

types of fertile ground are introduced to the game, we do not need to use many if-else statements in our implementation.

Advantages:

With the above design, we are following the Open Closed Principle as when more types of fertile ground are introduced, we could just determine whether a ground is a fertile ground by looking at its capability instead of using “instanceOf”, hence we can extend our game with modifying any existing code. We also adhere to the Single Responsibility Principle as all the methods and attributes above tell the properties of a Tree without modifying any code in its parent class. Besides that, we also create several class attributes to store the fixed value. Hence, we are adhering to the “avoid excessive use of literals” principle.

Disadvantages:

Since all the requirements and logic need to be performed within the tick method, the method will be too long and hence less readable and maintainable. It might cause confusion to debug when an error occurs.