REQ7: Reset Game

Overview:

To achieve this feature, there will be three new classes (i.e., Enemy, ResetAction, and Coin) created in the extended system, and four existing classes (i.e., Status, Tree, ResetManager and Player) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) Status class

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

In assignment 1, I created a lot of constants in different new classes such as ItemCapability and GroundStatus to perform the corresponding operation to Coin and Tree when reset is called. I realized that this is a bad design practice as it overcomplicated my code and I could do all this by just adding one constant to all the instances of these classes. Therefore, in assignment 2, I created a constant called RESET_CALLED. This constant will be added to the instances that their class implements Resettable and resetInstance is called. Therefore, this constant will help me to perform corresponding operations to those instances in either playTurn or tick method.

Why I chose to do it that way:

With this design, we will not use any more literal to specify whether a Player/Actor has a particular capability in another class as we can make use of these constants. Hence, we are adhering to the “avoid excessive use of literals” principle. This kind of implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

2) 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 let Enemy class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, when playTurn is called, it will remove that enemy instance from the map. Thus, 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.**

3) Tree class

Tree class is a class that represents the tree in this game. However, since Tree class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ1 and REQ2) and in this REQ, it only let Tree class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, if it has this capability, the this tree instance will have 50% chance to convert to dirt, without considering any features of a Tree instance, therefore the design rationale for Tree class will not be available in this section. **Hence, please go to the REQ1 and REQ2 section in the following pages to see the design rationale for Tree class.**

4) Coin class

Coin class is a class that represents the coin item in this game. However, since 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 let Coin class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, if it has this capability, then this item will be remove from the map, without considering any features of a Coin instance, therefore the design rationale for Coin class will not be available in this section. The usage of IS_COIN is explained in the design rationale of ItemCapability class. **Hence, please go to the REQ5 section in the following pages to see the design rationale for Coin class.**

5) Player class

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

The only changes I made between Assignment 1 and Assignment 2 is that in assignment 1, I not remove HOSTILE_TO_ENEMY when resetInstance is called, but in assignment 2, I will not remove the following three constant, which are HOSTILE_TO_ENEMY, BUY and SPEAK. The reason is because even if the game is reset, the player should still be able to speak to toad, buy from toad and be hostile to the enemy. Rest of the part remains the same as Assignment 1. Thus, please refer to the design rationale for this class in Assignment 1 for more details.

6) ResetManager class

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

No changes are made between Assignment 1 and Assignment 2.

7) ResetAction class

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

In assignment 1, I create a RemoveManager class to store all enemies that will be removed when reset is called. In addition, I use two nested for loops in the execute method of ResetAction class to remove coins and convert trees to dirt by looping through every single location in the map. However, I realized this is a very bad design practice as it violates Single Responsibility Principle. After that, I realized I can do all these operations by letting those classes implement Resettable and call its resetInstance method using ResetAction. This makes our code more readable, maintainable, and allow for extension.

Why I chose to do it that way:

With the design above, we are adhering to the Single Responsibility Principle as the ResetAction class only focuses on what will happen when the user selects option 'r'. Besides that, we also follow the Open Closed Principle. The reason is because based on our current implementation, when there is more Resettable instance or removeable instance created in future, we do not have to modify any code in ResetAction as we could just simply call the run method in ResetManager class and RemoveableManager class. In addition, we also fulfil the Liskov Substitution Principle as ResetAction preserves the meaning of execute and menuDescription method behaviours from Action class.