REQ7: Reset Game

Overview:

To achieve this feature, there will be six new classes (i.e., Koopa, RemovableManager, ResetAction, ItemCapability, GroundStatus and Coin) created in the extended system, and four existing classes (i.e., Goomba, Tree, ResetManager and Player) 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 ResetAction class will be the last among all ten classes. The reason is that the implementation of ResetAction 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 ResetAction better.

1) ItemCapability class

ItemCapability is a new enumeration class with only one constant that represents whether the item on a particular location is a Coin or not, which is IS_COIN.

Description of constant:

- An item has a capability of IS_COIN if it is an instance of Coin.

Why I chose to do it that way:

For this requirement, when the reset action is selected, I must remove all the coins on the ground, hence I would need a way to identify whether the item is a Coin or not. My initial thought is to use “instanceOf” to determine whether an item is a Coin instance, however, I realized this is not a good design practice, therefore I had decided to create this enumeration class instead. Thus, for each Coin instance, IS_COIN capability will be added to them.

Advantages:

With this enumeration class, we can avoid excessive use of literals as we do not need any local parameter to keep track of whether an item is a coin instance. Besides that, this class is created to adhere to the Separation of Concerns principle as the enumeration class only focuses on a single aspect, that is the possible capability of an item could have. This kind of implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

Disadvantages:

As the game develops, more and more capability will be added to this ItemCapability class. If there are too many constants within this class, it might cause confusion when debugging.

2) 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 add the IS_COIN capability using addCapability() method within Coin’s constructor 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.**

3) GroundStatus class

Note: the design rationale for GroundStatus class in REQ7 will be the almost same as the one in REQ1 as they both referring to the same class, but the design rationale will focus on different aspect (i.e., REQ1 focus on determining whether a ground is a fertile ground whereas REQ7 focus on determining whether a ground is a tree), hence there will still be some repetitive content.

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

Description of constant:

- A ground has a status of IS_COIN if it is a Tree instance.

Why I chose to do it that way:

According to the assignment specification for this requirement, it says that when the user selects the reset option, Tree has a 50% chance to be converted back to Dirt. It means that in the ResetAction class, when I loop through the whole map (will discuss more in the ResetAction section), I would need to know what types of ground is at a particular location. Therefore, by adding this capability to each Tree instance, it helps to determine whether the ground on a particular location is a Tree, so that I can perform the corresponding task (i.e., 50% chance to be converted back to Dirt).

Advantages:

Using constant to specify that whether a ground is a Tree helps us to adhere to the “avoid excessive use of literals” principle as we do not have to create any literals to keep track of the status of a ground. This implementation also provides us the flexibility for future development. For instance, if we decided to remove more types of ground or convert one type of ground to another when reset action is selected in future, we can simply add more constant in this class to monitor the type of a particular ground. 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:

Similar to ItemCapability class, as the game develops, more and more status will be added to this Ground status class. If there are too many constants within this class, it might cause confusion when debugging.

4) 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 add the IS_Tree capability using addCapability() method within Tree’s constructor without considering any features of a Tree instance, therefore the design rationale for Tree class will not be available in this section. The usage of IS_TREE is explained in the design rationale of the GroundStatus class. **Hence, please go to the REQ1 and REQ2 section in the following pages to see the design rationale for Tree class.**

5) RemovableManager class

RemovableManager class is a global singleton manager that stores all the actors that will be removed after the reset action is selected. In this case, we are using the static factory method. Within this class, it has two class attributes, one constructor and three methods.

Description of attribute:

- removeableActorList is a private static array list that stores Actor instance that will be remove after the reset action is selected.
- instance is a private static RemovableManager that represents the unique instance of this class.

Description of constructor:

- the constructor of this class will initialize the removeableActorList to an empty array list.

Description of method:

- getInstance is a method that will return the only instance of this class if the instance is not null, otherwise it will create a new instance of this class and return it.
- appendRemovableActor will append the input parameter which is an Actor instance to removeableActorList.
- run takes an input parameter of a GameMap instance. After that, it will loop through the removeableActorList and call the removeActor(removeableActor) on the map instance to remove each Actor in the array list

Why I chose to do it that way:

By using this static factory method, it helps me to monitor all the Actors that are ready to be removed. This method is very convenient as I only have to call the run() method to remove everything efficiently. Besides that, with this design, if we had decided to remove more Actor instances when reset action is selected by the user, we could simply add that instance to this class, hence we do not have to modify the existing code to achieve this objective. Thus, according to the specification of this requirement, all Koopa and Goomba instances will be added to the array list in this class.

Advantages:

With the above design, we make our class open for extension (i.e., Actor that are removeable can be added to this class) but closed for modification (i.e., we do not need to modify RemoveableManager class), so we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are either a command or a query. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Moreover, we also follow the Single Responsibility Principle as this class is only responsible for storing the removeable Actor instance and performing remove action on it.

Disadvantages:

We store all the removeable actor instance in a single array list (i.e., both Goombas and Koopas are storing in a single array list), hence there is no way for us to identify the actual instance of the Actor (i.e., no way to identify whether the Actor is Goomba or Koopa). Thus, if we need to perform specific operations to each type of enemy in future implementation, this might be a serious problem.

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 adds the Koopa instance to the RemoveableManager class by using the appendRemovableActor method within Koopa's constructor 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) Goomba class

Goomba class is a class that represents one of the enemies in this game, which is Goomba. However, since Goomba 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 adds the Goomba instance to the RemoveableManager class by using the appendRemovableActor method within Goomba's constructor without considering any features of a Goomba instance, therefore the design rationale for Goomba 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 Goomba class.**

8) ResetManager class

ResetManager is a global Singleton manager that does soft reset on the instances. It is an existing class given in the base code. In this class, I had only modified the run method and the appendResetInstance method.

Description of method:

- when the run method is called, it will loop through the array list that stores the instances of Resettable and for each resettable instance, it will call the resetInstance() method.

- appendResetInstance is a method that appends the input parameter with the type of Resettable to the array list in this class.

Why I chose to do it that way:

According to the specification of this requirement, I would need to heal the player to maximum and reset the player status. Hence, in order to do this, I had decided to let Player class implement Resettable and it will then add the instance of Player to ResetManager by using appendResetInstance method. Besides that, inside the Player class, I will also implement the resetInstance() method (will discuss more in the Player section) to heal the player to maximum and reset the player status. Thus, when the run method is called, it will call the resetInstance() of the Player and perform the corresponding operations. With this design, it provides us the flexibility for future development. For instance, if there are more actors that need to reset their status once reset button is pressed in future, we could simply let the class of that actor implement Resettable and add it to ResetManager. Simply calling the run () method in this class will automatically reset all the actors.

Advantages:

With the above design, we make our class open for extension (i.e., Actor that are resettable can be added to this class) but closed for modification (i.e., we do not need to modify

ResetManager class), so we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are either a command or a query. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Moreover, we also follow the Single Responsibility Principle as this class is only responsible for storing the resettable Actor instance and performing resetInstance action on it. Besides that, it also allows each Resettable instance to have their own resetInstance method, which provides much flexibility for future implementation.

Disadvantages:

Similar to RemoveableManager class, we store all the resettable actor instance in a single array list (i.e., currently only Player is in the array list), hence there is no way for us to identify the actual instance of the Actor (i.e., no way to identify whether the Actor is Player or something else). Thus, if we need to perform specific operations to each type of resettable in future implementation, this might be a serious problem.

9) Player class

Player is a class that represents the Player in the Game Map, extends Actor class, and implements Resettable. It is an existing class given in the base code. In this class, I had created a new class attribute and only modified the constructor of the Player class, playTurn method and added a new method called resetInstance.

Description of attribute:

- oneReset is a Boolean which is initialized as false. After the user performs the reset action then it will then set the true. (Will discuss more in ResetAction section)

Description of constructor:

- For the constructor of the Player class, the only modification that I had done is to add the Player instance to the ResetManager by using this line of code, this.registerInstance()

Description of method:

- playTurn is a method to figure out what to do next for the Player. In this method, the only modification I have done is to use an if statement to check whether oneReset has been set to true. If oneReset is true, it means the user had performed the reset action hence reset action should not be available to the user anymore, otherwise, oneReset will be false and allow the user to perform the reset action on the menu.

- resetInstance is a method that must be implemented since the Player class now implements Resettable. In this method, it will use a for loop to loop through all the status of this Player instance and remove every single status except HOSTILE_TO_ENEMY. After that, it will heal the player to maximum by using this line of code, this.heal(this.getMaxHp()). Eventually, it will set oneReset to true.

Why I chose to do it that way:

For this requirement, I had to remove all the status of the Player except and heal the Player to maximum. Initially, I was thought to do these operations in the ResetAction class. However, I realized that this is not an efficient way. The reason is because what if there is more Player/Actor exists in the Game and needs to be reset after the reset button is pressed? If I reset each Player/Actor in the ResetAction class, the code will be very long and

not readable and maintainable. Therefore, by using the ResetManager and Resettable interface, it helps me to reset all the Player/Actor that are required to reset by just calling a single method, which is run () in ResetManager. This is an efficient way and makes our code readable and maintainable and easy to extend.

Advantages:

With the above design, we are adhering to the Single Responsibility Principle as the method within Player class only shows the properties of a Player and what a Player can do. Moreover, we also fulfil the Liskov Substitution Principle as all the methods in Player class still preserve the meaning from its parent class, which is Actor. Other than that, we also created a class attribute called oneReset to keep track whether the reset action has been used, this is to avoid excessive use of literals principle.

Disadvantages:

N/A

10) ResetAction class

ResetAction is a class that will reset the game when the user selects the reset option on the menu. It is a new class that extends Action class. In this class, it overrides the execute menuDescription and hotkey method from its parent class.

Description of method:

- For the execute method, it will only be called when the user selects the reset option on the menu. In this method, firstly, it will get the instance of the RemoveableManager and call the run method. In the previous section, we know that RemoveableManager stores all the Actors (i.e., currently Koopa and Goomba) that should be removed once reset option is selected, therefore, by using the run method, we can remove all the enemies from the map. After that, it will get the instance of the ResetManager and call the run method. In the previous section, we also know that ResetManager currently stores the Player instance. Hence, when we call the run method in ResetManager, it will reset the status of the Player and heal the Player to maximum. Next, in order to remove all the Coin instances from the map, we will loop through every single location in the map using the map.at(x,y) method by using two for loops where x and y are the coordinates for each location. Thus, for each location, we will check the itemList for that location, if there is a Coin instance (by checking whether the item has the capability of IS_COIN), then we will remove it (i.e., using removeItem()). In addition, based on the description of this REQ, each Tree instance will have 50% to convert back to Dirt. Therefore, similarly, we will use two for loops to loop through every single location on the map and get the ground instance of that location using getGround() method. After that, I will set that particular ground to a Dirt by using the setGround() method with 50% chance. Eventually, a string “The game is now reset” will be printed out. In conclusion, by doing all the operations above, we had successfully fulfilled the requirement for this feature.

- menuDescription will return a string that will appear on the command list in the console. The String that will be returned is “Reset the game”.

- hotkey returns a key used in the menu to trigger this reset action. In this case, the hotkey is ‘r’.

Why I chose to do it that way:

Since ResetAction is an action and we need the functions in the Action class, and thus we let ResetAction class extend Action class. However, since we need to ensure that when ResetAction is called, the corresponding operations will be performed (i.e., all the scenario listed in the assignment specification), hence we will need to override the execute method, menuDescription and hotkey to ensure our game logic works properly. The description and the method to perform the logic for the execute method is mentioned above. Besides that, the reason for me to override the menuDescription method is to let the user have a better understanding on what will happen if he/she pressed ‘r’ on the console.

Advantages:

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.

Disadvantages:

N/A