REQ2: Jump Up, Super Star!

Overview:

To achieve this feature, there will be three new classes (i.e., JumpAction, HighGround, and Destroyable) created in the extended system, and three existing classes (i.e., Wall, Tree, and Status) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) HighGround class

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

In assignment 1, I let those high-level grounds implement the Jumpable interface (the details of Jumpable interface can be viewed in the pdf of Assignment 1). However, I realized this is not a good design practice as using an interface does not reduce the repeated code of high-level grounds efficiently. Therefore, I had decided to create this abstract class called HighGround which extends Ground. In addition, I will let Tree and Wall class to extend this class as they are the current high ground in the game. By doing so, they can share the same code such as canActorEnter, getJumpRate and so on.

Why I chose to do it that way:

Currently based on the requirement of this feature, the Wall class and Tree class will be the extension of this class. This indicates that Wall and Tree on the map are jumpable for the Player as they are high ground. The reason for us to create an abstract class is that I realized Wall and Tree have a lot of repeated code and they both extend from the Ground. Hence, I decided to let HighGround extend Ground and let Wall and Tree class extend HighGround. Besides that, with this design, if we had to add a new type of ground that is also jumpable for the Player, we do not have to modify the existing code as we could just let this new class extend HighGround. By doing so, we are adhering to the Open-closed Principle (OCP).

In addition, we can see that all the methods in the class are either a query or command but not both. Hence, we are following the Command-Query Separation Principle. Moreover, our implementation also fulfils the Reduce Dependencies Principle. It is because instead of having a JumpAction class to have an attribute of each high ground (i.e., Wall or Tree) (we will discuss this more in the JumpAction section), we can have an attribute of type HighGround. By doing so, the concrete class JumpAction will not directly depend on the Wall and Tree class. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the HighGround class still preserve the meaning from its parent class, Ground. Additionally, within the allowableAction method in the HighGround class, we pass in the instance of HighGround to the JumpAction class, which is known as constructor injection. The benefit of using dependency injection is to ensure the reusability of code and ease of refactoring.

2) Wall class

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

Instead of implementing the Jumpable interface (which is already removed in Assignment 2) as it does not reduce the repeated code efficiently, I created a new abstract class called HighGround and let this class extend it. The reason is because I realized that all high ground classes (i.e., currently only Wall and Tree) share some part of the code. To avoid any

repeated code, I had decided to let Wall class extend HighGround. Therefore, in Wall class, it only overrides one method which is blocksThrownObjects, as Wall can block thrown objects.

Why I chose to do it that way:

With the above design, we are following the Single Responsibility Principle as all the methods and attributes above tell the properties of a Wall and what an actor can do to the Wall without modifying the implementation of its parent class (i.e., HighGround). We also obey the principle of Command-Query Separation Principle as none of the methods above do both command and query. Besides that, we are also adhering to the Reduce Dependencies principle as JumpAction will not have to depend on the Tree and Wall since we make use of the HighGround abstract class. Besides that, since I am storing all the constants as private class attributes, consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Furthermore, we also fulfil the Liskov Substitution Principle as all the methods in the Wall class still preserve the meaning from its parent class, HighGround.

3) Tree class

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

Similar to the Wall class, instead of implementing the Jumpable interface (which is already removed in Assignment 2) as it does not reduce the repeated code efficiently, I created a new abstract class called HighGround and let this class extend it. The reason is because I realized that all high ground classes (i.e., currently only Wall and Tree) share some part of the code. To avoid any repeated code, I had decided to let Tree class extend HighGround. More details regarding Tree class and its subclasses can be viewed in the design rationale of Tree class in REQ1.

Why I chose to do it that way:

With the above design, we are following the Single Responsibility Principle as all the methods and attributes above tell the properties of a Tree and what an actor can do to the Wall without modifying the implementation of its parent class (i.e., HighGround). We also obey the principle of Command-Query Separation Principle as none of the methods above do both command and query. Besides that, we are also adhering to the Reduce Dependencies principle as JumpAction will not have to depend on the Tree and Wall since we make use of the HighGround abstract class. Besides that, since I am storing all the constants as private class attributes, consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Furthermore, we also fulfil the Liskov Substitution Principle as all the methods in the Tree class still preserve the meaning from its parent class, HighGround. Besides that, inside the Tree class, I had added a helper method to convert the tree instance to dirt at that location. For this method, I had checked the pre-condition of the input parameter (i.e., currentLocation) such that it cannot be null, else it will throw a IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allow the developer to know where the problem is.

4) Status 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 to rename the constant JUMP_ONE_HEIGHT to CAN_JUMP and delete SUPER_MUSHROOM. The reason to rename JUMP_ONE_HEIGHT is because I think CAN_JUMP is a more appropriate name. In addition, the reason to delete SUPER_MUSHROOM is because I realized the constant TALL does the exact same function with it, hence this new constant seems to be redundant. Other than that, there are no more changes made.

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.

5) JumpAction class

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

The only changes made between Assignment 1 and Assignment 2 is that instead of storing a type of Jumpable as the class attribute, we are now storing a type of HighGround private attribute in JumpAction. Besides that, since the name of the constant (i.e., stated in Status class), we must change the name accordingly in the execution method. Other than these two minor changes, the rest of the part is the same as the explanation stated in Assignment 1. Therefore, please refer to the design rationale of this class in assignment 1.

6) Destroyable class

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

According to the assignment specification, when the player is invincible, it will destroy the high ground (i.e., convert it to dirt) and drop 5 coins. However, in assignment 1, we did not create this class and we planned to do these operations inside Wall and Tree, which is a bad design practice. Thus, in assignment 2, I had decided to create a destroyable interface class which has two methods. (i.e., convert the current location to dirt and create a coin instance with a value of 5 at the location. After that, I will let HighGround implement the Destroyable interface.

Why I chose to do it that way:

It is possible that more types of ground (i.e., other than high ground) can be destroyed in the future, therefore in order to allow extension, we make this as an interface class, so that other classes can implement it in the future. This is also the reason why I did not put the two operations (i.e., convert to dirt and drop coins) directly in HighGround class. By doing so, we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are just a command. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Besides that, we are adhering to the Interface Segregation Principle as it only implements the methods that it cares about. Furthermore, for the two default methods in this interface, I had checked their input parameter to ensure that the location given is not null, and the coin value is 5. The reason to check if the coin value is 5 is that currently based on the assignment specification, when a high ground is destroyed, the value of the coin must be 5. However, if the value of the coin drops when the high ground is destroyed can be different (i.e., not only

5), then we can remove this exception. By doing so, we are adhering to the Fail Fast Principle as we immediately stop the game when the code detects something wrong, and this helps the developer to know where the problem is.