REQ2: Jump Up, Super Star!

Overview:

To achieve this feature, there will be three new classes (i.e., JumpAction, Jumpable, and TreeStatus) 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. However, the explanation of the design rationale for JumpAction class will be the last among all six classes. The reason is that the implementation of JumpAction 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 JumpAction better.

1) Jumpable class

Jumpable is an interface for classes that are jumpable by the Player. It is a new interface class that contains three methods, which are getJumpRate, getJumpableType and getFallDamage.

Description of methods:

- getJumpRate will return an integer that indicates the success rate for an actor to jump to that ground.
- getJumpableType will return a string that indicates the type of that ground.
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that ground.

Why I chose to do it that way:

Currently based on the requirement of this feature, the Wall class and Tree class will implement this interface. This indicates that Wall and Tree on the map are jumpable for the Player. The reason for us to create an interface instead of an abstract class is that multiple inheritances are not allowed as Wall and Tree already extends from the Ground. 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 implements Jumpable.

Advantages:

By doing so, we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the three methods above are just 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, our implementation also fulfils the principle of Dependency Inversion Principle. It is because instead of having JumpAction class to have an attribute of each jumpable ground (i.e., Wall or Tree) (we will discuss this more in JumpAction section), we can have an attribute of type Jumpable. By doing so, the concrete class JumpAction will not directly depends on the Wall and Tree class.

Disadvantages:

N/A

2) Status class

Status is an enumeration class that gives 'buff' or 'debuff'. It is also useful to give a 'state' to abilities or actions that can be attached-detached. It is an existing enumeration class with some additional modification. In this class, I had added two statuses, which are JUMP_ONE_HEIGHT and SUPER_MUSHROOM.

Description of constant:

- JUMP_ONE_HEIGHT indicates that the Player have the capability to jump to wall or tree. However, according to the assignment specification, it says that the height and depth might change in future, hence I had decided to name this status in a more specific way to avoid confusion in future.
- SUPER_MUSHROOM indicates that the Player had consumed a super mushroom.

Why I chose to do it that way:

Instead of creating class attributes or literals to indicate whether a particular actor has a capability to jump or consume super mushrooms, it is better to make use of the enumeration class to make our code easier to read. Besides that, I also can utilise the engine code provided to check the actor's capability to simplify our code.

Advantages:

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.

Disadvantages:

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

3) Wall class

Wall is a class that represents the wall in the Game Map. It is a class that extends from the Ground and implements Jumpable. There are three class attributes in the Wall class, which are JUMP_RATE, FALL_DAMAGE, and JUMP_TYPE.

Description of class attributes:

- JUMP_RATE is a private static final integer class attribute with a value of 80 that indicates the success rate to jump to a wall is 80%.
- FALL_DAMAGE is a private static final integer class attribute with a value of 20 that indicates the fall damage after a failed jump is 20.
- JUMP_TYPE is a private static final string class attribute that is assigned to “Wall” to indicate the type of this jumpable ground is a wall.

The purpose of creating these class attributes is to adhere the “avoid excessive use of literals” principle. Hence, I had declared these attributes as constant.

In addition, Wall class has overrides 6 methods, which are canActorEnter, blockThrownObjects, getJumpRate, getJumpableType, getFallDamage and allowableActions.

Description of methods:

- canActorEnter will return true if and only if the actor has the capability of JUMP_ONE_HEIGHT
- blockThrownObjects will always return true to indicates that wall can block thrown objects in the game
- getJumpRate will return an integer that indicates the success rate for an actor to jump to that wall (i.e., JUMP_RATE).
- getJumpableType will return a string that indicates the type of ground (i.e., FALL_DAMAGE).
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that wall (i.e., JUMP_TYPE).
- allowableActions will return a JumpAction instance if the actor is currently not on that wall, else return an empty ActionList. By doing so, we can make sure the actor will not be able to jump to that wall if the actor is currently already on that wall.

Why I chose to do it that way:

Not all types of ground in this game are jumpable, hence I had decided to create an interface class specifically for those grounds that are jumpable. Hence, in this case I had made the Wall class to implement the Jumpable class. By doing so, we do not have to do any modification to the code within Ground class and this kind of implementation does provide the flexibility for us to extend our game easily. (i.e., we may create more types of ground that are jumpable for actors in future) By implementing Jumpable class, we also make sure Wall class will implement the three methods that are mandatory for a jumpable ground.

Advantages:

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., Ground). 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 Jumpable interface.

Disadvantages:

This kind of implementation does not provide us the flexibility to extend our Wall class. What I meant is that in future, a Wall instance might have several statuses (e.g., broken) and lead to different fall damage and success rate of a jump. When this happens, we will have many if-else statements for the getters of these attributes. This might be a problem as the number of statuses keep growing and are hard to maintain and debug.

4) TreeStatus class

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.

Disadvantages:

Similar to the Status class, if a Tree will have many statuses in the future, it might confuse when debugging.

5) Tree

Tree is a class that represents the tree in the Game Map. It is a class that extends from the Ground and implements Jumpable. According to the assignment specification, a Tree has three stages and the success rate and fall damage is different for each stage, hence there will be a significant amount of class attributes in this class.

All the class attributes in Tree class:

```
private int age = 0;
private String type;
private static final char SPROUT_CHAR = '+';
private static final char SAPLING_CHAR = 't';
private static final char MATURE_CHAR = 'T';
private static final int SPROUT_JUMP_RATE = 90;
private static final int SAPLING_JUMP_RATE = 80;
private static final int MATURE_JUMP_RATE = 70;
private static final int SPROUT_FALL_DAMAGE = 10;
private static final int SAPLING_FALL_DAMAGE = 20;
private static final int MATURE_FALL_DAMAGE = 30;
private static final String SPROUT_TYPE = "Sprout";
private static final String SAPLING_TYPE = "Sapling";
private static final String MATURE_TYPE = "Mature";
```

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

In addition, Tree class has overridden 6 methods which are canActorEnter, tick, getJumpRate, getJumpableType, getFallDamage and allowableActions and create two new methods, which are isSapling and isMature.

Description of methods:

- canActorEnter will return true if and only if the actor has the capability of JUMP_ONE_HEIGHT
- tick method will increment the age of Tree instance by 1 after each turn. Besides that, it will also check if the Tree instance reaches the age of 10 or 20 by using the methods isSapling and isMature. If yes, it will set the tree status, display character and type accordingly.
- getJumpRate will return an integer that indicates the success rate for an actor to jump to that tree according to the tree stages.
- getJumpableType will return a string that indicates the type of ground according to the tree stages.
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that tree according to the tree stages.
- allowableActions will return a JumpAction instance if the actor is currently not on that tree, else return an empty ActionList. By doing so, we can make sure the actor will not be able to jump to this wall if the actor is currently already on that wall.
- isSapling will return true if the age of the Tree instance reached 10
- isMature will return true if the age of the Tree instance reached 20

Why I chose to do it that way:

In the Tree class, I had created quite a number of class attributes. The purpose of creating these class attributes is to adhere to the “avoid excessive use of literals” principle, so that I can use these attributes instead of literals within the methods and hence make our code more readable. Besides that, since Tree is jumpable for actors, hence I let the Tree class implement Jumpable. By doing so, we can achieve our objective without modifying any existing code in the Ground as not all types of ground are jumpable. This makes our class structure more organised and understandable.

Advantages:

Like Wall class, 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 Tree without modifying the implementation of its parent class (i.e., Ground). 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 Jumpable interface.

Disadvantages:

Since the fall damage, success rate of jump and display character for each stage of tree is different, we will need a lot of class attributes as you have seen above. If there is a new stage introduced, we will need even more attributes. This makes our code less maintainable and cause confusion. In addition, we will have many if-else statements for the getters of these attributes. This might be a problem as the number of statuses keep growing and are hard to maintain and debug.

6) JumpAction

JumpAction is a class that allows the actor to jump to the jumpable ground. It is a new class that extends Action class. In this class, there are four class attributes, which are moveToLocation, direction, rand and jumpableGround.

Description of class attributes:

- moveToLocation is a protected attribute with a type of Location. It indicates the destination of the actor.
- Direction is a protected attribute with a type of string. It indicates the direction of the actor's destination.
- rand is a private constant attribute with a type of Random, it will be used in the execute method.
- jumpableGround is a private attribute with a type of Jumpable.

The JumpAction class overrides the execute and menuDescription method from its parent class.

Description of methods:

- For the execute method, there will be an if-else statement. If the actor does not consume super mushroom (i.e., check the actor's capability) and the random number generated is lower than the success rate of that jumpable ground, the piece of code within the if statement will execute. Within the if statement, the actor will receive the fall damage (i.e., check the fall damage for that jumpable ground using getFallDamage method), and a message that indicates the actor has failed to jump to that jumpable ground will be printed out. If the piece of code within the else statement has executed (i.e., the actor has successfully jumped to the jumpable ground), it will add the JUMP_ONE_HEIGHT capability to the actor. By doing so, we can ensure that the actor can jump to that jumpable ground as the canActorEnter will return true for that jumpable ground. After that, it will move the actor to that location using the moveActor method. In addition, it will remove the capability of JUMP_ONE_HEIGHT from that actor so that we can ensure that the actor will need to "jump" again if the actor wants to jump to any jumpable ground. Eventually, a message that indicates that the actor has successfully jumped to the jumpable ground will be printed out.
- menuDescription will return a string that will appear on the command list in the console. This string included the type of the jumpable ground, the coordinate of that jumpable ground and the direction.

Why I chose to do it that way:

Since JumpAction is an action and we need the functions in the Action class, and thus we let JumpAction class extend Action class. However, since we need to ensure that when JumpAction is called, the actor will have a successful/failed jump, hence we will need to override the execute method to ensure our game logic works properly. The description of the execute method is mentioned above. Besides that, I had also override the menuDescription method, the reason to do so is to let the user have a better understanding on what will happen when he/she lets the actor perform a JumpAction. In addition, in order to let the execute method know what the fall damage is, type of ground and success rate of a jump for that particular jumpable ground, I had included that particular jumpable ground to be one of the input parameters for the constructor. By doing so, we can get all the necessary information that we need and use it in the execute and menuDescription method. Thus, this simplifies our code.

Advantages:

With the design above, we are adhering to the Single Responsibility Principle as the JumpAction class only focuses on the action that will be executed when the actor jumps. Besides that, we also follow the Open Closed Principle. The reason is because based on our current implementation, when there is more jumpable ground created in future, we do not have to modify any code in JumpAction. In addition, we also fulfil the Liskov Substitution Principle as JumpAction preserves the meaning of execute and menuDescription method behaviours from Action class.

Disadvantages:

N/A