Assignment 3 REQ1: Lava Zone

Overview:

To achieve this feature, there will be five new classes (i.e., Lava, WarpPipe, PiranhaPlant, TeleportAction, and TeleportManager) created in the extended system, and three existing classes (i.e., Application, Player, and Status) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) Application class

What changed in the design between Assignment 2 and Assignment 3 and Why:

One of the changes I made between Assignment 2 and Assignment 3 is that I had created a new game map that follows the requirement (i.e., randomly place some Lava ground and one warp pipe at the top left corner). Besides that, I had placed some warp pipes on the first map too. Moreover, I had made the two GameMap instances be the private static class attribute of the Application class and created two public static methods (i.e., getters) that can be used to access these two GameMap instances. The reason to do so is that these two public static methods are mandatory to teleport the player to the desired location. Please refer to the design rationale of TeleportAction for more details regarding how the two public static methods are used.

Why I chose to do it that way:

By declaring the two GameMap instances as the private class attribute and only allowing public getters to access them, we are ensuring the encapsulation. Hence, we are protecting our GameMap objects from unwanted access by other classes. In addition, we can see that the two new methods are just a query. Hence, we are following the Command-Query Separation Principle as the new methods are either a command or query but not both.

2) Status enum

What changed in the design between Assignment 2 and Assignment 3 and Why:

Due to the requirement in Assignment 3, I had created three new constants, which are TELEPORT, FIRST_MAP, and SECOND_MAP. The purpose of creating TELEPORT is to identify which actor can use the warp pipe to perform teleportation. Currently, in this game, the only actor that can perform teleportation is Player, whereas other actors like all subclasses of Enemy (e.g., Koopa) or Toad are not allowed to do so, hence TELEPORT will be added to Player using its constructor. FIRST_MAP and SECOND_MAP are used to identify which map currently the Player is on, these two constants are very useful when the TeleportAction is executed (Please refer to the design rationale for TeleportAction for more details) as we need to know in advance which map is the target map after performing teleport action. For instance, if the Player currently is on the first map and it performed a teleport action, then we know that the target map would be the second map, and vice versa.

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. Besides that, if there are more actors who can perform teleportation, we could just add TELEPORT to its capability set. This

kind of implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

3) Player class

What changed in the design between Assignment 2 and Assignment 3 and Why:

The only changes that I made between Assignment 2 and Assignment 3 are that I added TELEPORT and FIRST_MAP to the Player's capability set and make sure TELEPORT, FIRST_MAP and SECOND_MAP will not be removed from the Player's capability set when the reset action is called, the reason is that these three constants should not be removed (i.e., even reset action is called, Player still can perform teleportation, and FIRST_MAP and SECOND_MAP only tell which map the Player is currently on, hence should not be removed).

4) Lava class

Why I chose to do it that way:

According to the assignment specification, there is a new type of ground called lava that will inflict 15 damage per turn when the player steps on them. Hence, to achieve this feature, I created a new class called Lava. Lava is a class that extends Ground. Inside its tick method, it would first get the actor that is standing on that location using location.getActor(). If an actor is standing at that location (i.e., getActor() does not return null), then it would hurt the actor by 15 damage. Besides that, if the actor is dead because of the damage, it will be directly removed from the map. Since the enemy is not allowed to enter the lava ground, inside the canActorEnter method, I will return true if and only if the actor does not have IS_ENEMY (i.e, created in Assignment 2) capability.

With the design above, we are adhering to the Single Responsibility Principle as the Lava class only focuses on what will happen when the actor standing in that location and the method within the Lava class only shows the properties of a Lava. Furthermore, for each of the properties of a Lava instance (e.g., display character and damage per turn), I store them as private constant class attributes. Consequently, we are following the "avoid excessive use of literals" principle as it makes our code more readable.

5) WarpPipe class

Why I chose to do it that way:

In assignment 3, there is a new type of ground called WarpPipe such that when the Player stands on it, it can perform teleportation to its desired location. To achieve this feature, I created a new class called WarpPipe. Since the Player must jump to the WarpPipe to perform teleportation, I realized that it has a lot of common code with the HighGround class, therefore I let WarpPipe extend HighGround. By doing so, we are reducing our repeated code and, hence, following the DRY principle.

According to the FAQ of Assignment 3, I realized that Warp Pipe should not be destroyable when the Player is invincible and standing on it, hence I would need to override the tick

method in HighGround (since HighGround is meant to be destroyable in Assignment 2 and its tick method does this operation).

After that, only if the player is standing on it and no enemy (i.e., PiranhaPlant) is standing on it, the TeleportAction will be available to the user, and it will pass itself (i.e., the current WarpPipe instance) to the constructor of the TeleportAction (please refer to the design rationale of TeleportAction for more details). Otherwise, if the player is not standing on it (i.e., besides it) then JumpAction will be available (i.e., by calling the HighGround allowableAction method).

In addition, I had created two private integer class attributes that store the X and Y coordinates of the WarpPipe location and two getters to access these two class attributes. The reason to do so is to help TeleportManager get the coordinates of the previous WarpPipe. Please refer to the design rationale for the TeleportManager class for more details. (i.e., how the previous WarpPipe that the Player used to teleport from is kept tracked by TeleportManager)

With the design above, we are adhering to the Single Responsibility Principle as the method within WarpPipe class only shows the properties of a WarpPipe. Furthermore, for each of the properties of a WarpPipe instance (e.g., display character, a success rate of a jump, and so on), I store them as private constant class attributes. Consequently, we are following the "avoid excessive use of literals" principle as it makes our code more readable. In addition, we can see that all the methods are either a command or query but not both. Hence, we are following the Command-Query Separation Principle.

6) PiranhaPlant class

PiranhaPlant class is a class that represents one of the enemies in this game, which is Piranha Plant. However, since PiranhaPlant class will not be implemented in this REQ (i.e., it's implementation will only be considered in Assignment 3 REQ2) and in this REQ, it only creates the PiranhaPlant instance without considering any features of a PiranhaPlant instance, therefore the design rationale for PiranhaPlant class will not be available in this section. **Hence, please go to the Assignment 3 REQ2 section in the following pages to see the design rationale for PiranhaPlant class.**

7) TeleportManager class

Why I chose to do it that way:

TeleportManager is a global singleton manager responsible to teleport the actor to the desired location. It will only be called if the actor executes TeleportAction. Please refer to TeleportAction class for more details. The reason to have this TeleportManager class is that we need a way to keep track of the previous WarpPipe and store the coordinates of the warp pipe in the second map. This cannot be done in the TeleportAction class as it will violate the Single Responsibility Principle since TeleportAction should not be responsible for storing this information. Hence, TeleportManager is needed.

According to the assignment specification, assuming there are two warp pipes called A and B in the first map, it says that if the Player teleported to the second map using pipe A, then when the Player teleported back to the first map using the only pipe in the second map, the Player must be teleported to the previous pipe, which is pipe A, not B. Hence, to keep track of the previous warp pipe, I created a private class attribute to store the previous warp pipe instance (i.e., the warp pipe used in the first map), by doing so, when the actor needs to

teleport back to the first map, we can teleport to that location using the X and Y coordinates of the previous WarpPipe instance (due to the two getters mentioned in WarpPipe class section). The previous warp pipe only will be stored (i.e., using the setter) if the actor wanted to teleport from the first map to the second map.

The main method in the TeleportManager class is the run method. It takes in three input parameters, which are the actor that wants to teleport, the map that the actor wants to teleport to, and a Boolean that indicates whether the actor is currently on the first map or not. After that, it will perform the teleportation (i.e., move the actor to the desired location) corresponding to the value of the parameters, please refer to the code and design diagrams for more details.

With this design, we are following the Command-query principle as none of our methods in this class are both command and query. Besides that, we also store the coordinate of the warp pipe in the second map as the private constant class attributes. Thus, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Additionally, I had declared the class attribute (i.e., WarpPipe instance that indicates the previous warp pipe) as private and only the public setter can modify it, hence ensuring encapsulation. Moreover, for the run method and setFirstWarpPipe method, I had checked the pre-condition of the input parameter (i.e., actor and targetMap for run method, and firstMapWarpPipe for setFirstMapWarpPipe) such that it cannot be null, else it will throw an IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allows the developer to know where the problem is.

8) TeleportAction class

Why I chose to do it that way:

TeleportAction class is a class that allows an actor to perform teleportation. The input parameter of its constructor is the instance of the WarpPipe that the actor is currently standing on. When the actor executes the TeleportAction, the execute method will first check if the Player is currently on the first map or not (i.e., using the constant). If yes, it will store the current WarpPipe instance (i.e., the one that passed using the constructor) to TeleportManager. The reason to do so is that we need to keep track of the previous warp pipe so that when the actor teleports back to the first map, it will be standing on this WarpPipe instance. After that, we will get the target map (i.e., the GameMap instance) using the two static methods created in the Application class. Lastly, we will call the run method in TeleportManager and pass the actor, target map, and a Boolean indicating whether the actor is now on the first map. Hence, the teleportation is now performed successfully by calling that run method. Eventually, we just return a string to indicate to the user that we had performed the teleport action. Please refer to the code in TeleportAction class for more details.

With the design above, we are adhering to the Single Responsibility Principle as the TeleportAction class only focuses on what will happen when the user selects teleport action. Additionally, within the allowableAction method in the WarpPipe class, we pass in the instance of WarpPipe to the TeleportAction class, which is known as constructor injection. The benefit of using dependency injection is to ensure the reusability of code and ease of refactoring.