Assignment 3 REQ2: More allies and enemies!

Overview

To achieve this feature, there will be eight new classes (i.e., GeneralKoopa, FlyingKoopa, PrincessPeach, Bowser, PiranhaPlant, VictoryAction, Key and FireAttackBehaviour) created in the extended system, and six existing classes (i.e., Koopa, Mature, HighGround, WarpPipe, Status and AttackAction) will be modified. As this design rationale is for REQ2, so all the implementation about Speakable and Monologue will be explain in details in design rationale of REQ5, this design rationale will mention “Speakable” but without explanation. The design rationale for each new or modified class is shown on the following pages.

1) PrincessPeach

Why I choose to do it that way:

According to Assignment 3 requirements, PrincessPeach is a class that extends from Actor class and implements Speakable. By extending Actor class, she cannot attack, follow or move around. In this class, only the actor with capability HAVE_KEY can interact with her in order to end this game. It follows SRP(single responsibility principle) as PrincessPeach have only one job which is interact with Mario and end this game. In addition, by adding many constant class attributes, it follows "avoid excessive use of literals" principles as I’m avoiding the use of magical numbers and literals. It makes our code clear and logic.

2) GeneralKoopa

Why I choose to do it that way:

In Assignment 3, I created a new abstract class called GeneralKoopa to be the parent class of all Koopas(normal Koopa and Flying Koopa) in this game, and GeneralKoopa is extends Enemy to obtain all the enemy features. By adding an abstract GeneralKoopa class, all the common behaviors of Koopas can be state in this class and its child class can inherit form it directly and save in a hashmap called behaviors. This is to avoid too many repeated code, this obeys the DRY design principle, which make our code a good maintenance.

Besides that, if more Koopas need to be implemented in this game, we don’t have to repeat so many codes. This follows the open close principle because when more Koopas added, you do not have to modified GeneralKoopa class but allow additional class to extends it, hence it allow extension.

Furthermore, by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I’m now avoiding the use of magical numbers and literals.

3) FlyingKoopa

Why I choose to do it that way:

According to Assignment 3 requirements, FlyingKoopa is a class that extends from GeneralKoopa class. By extending GeneralKoopa class, Flying Koopa can obtain all the behavior of GeneralKoopa directly without any implementation, this obeys the design principle DRY and make the code more logic and easy to read. Other than that, it adheres to the Liskov substitution principle, as all methods in the FlyingKoopa class retain the meaning of their parent class, GeneralKoopa. Besides that, I add a capability to it which is CAN_FLY, so FlyingKoopa class has its own responsibility which it performs the behavior of FlyingKoopa only, so it obeys SRP(single responsibility principle). And by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I’m now avoiding the use of magical numbers and literals.

4) Koopa

What changed in the design rationale between Assignment 2 and Assignment 3 and Why:
In Assignment 3, instead of let Koopa class extend Enemy class, I make Koopa class extends GeneralKoopa class. In Assignment 2, there was only one type of Koopa so extends Enemy class is enough. But at this stage, two types of Koopas is exist, so if these two types of Koopas still extends Enemy class, there will be too many repeated code appear, which disobey the design principle DRY(don’t repeat yourself).

Why I choose to do it that way:
By doing so, many default behaviors of Koopa can be override form its parent class(GeneralKoopa) directly without coding again. This obeys the design principle DRY and make the code more logic and easy to read. Besides that, Koopa class has its own responsibility which it performs the behavior of Koopa only, so it obeys SRP(single responsibility principle). It also adheres to the Liskov Substitution Principle, which ensures that the meaning of parental acts is preserved.

5) PiranhaPlant

Why I choose to do it that way:
According to Assignment 3 requirements, PiranhaPlaint is a class that extends from Enemy class and it is a plant which cannot move around, so I remove the wander behavior of it in the constructor. By extending Enemy class, the default behaviors of enemy can be override form its parent class (Enemy) directly without coding again. This obeys the design principle DRY and make the code more logic and easy to read. Besides that, PiranhaPlant class has its own responsibility which it performs the behavior of PiranhaPlant only, so it obeys SRP(single responsibility principle). In addition, by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I'm now avoiding the use of magical numbers and literals.

6) Bowser

Why I choose to do it that way:
According to Assignment 3 requirements, Bowser is a class that extends from Enemy class, and Enemy class is and implements Resettable, so Bowser can use resetInstance also. It is a enemy which stand still until Mario come close, so I remove the wander behavior of it in the constructor. Other than that, when Bowser is defeated(not conscious anymore), it will drop a key for Mario to save the PrincessPeach, so I add the capability DROP_KEY for it in the constructor also.

Refer to the requirements, when the game is reset, Bowser should go back to its original position and heal it to the max and if there is an actor stand on Bowser's original position, Bowser should move to one of the 8 exits, so I set the Bowser's location by initialize the coordinates of it in the class attributes, and when the game is reset, Bowser will be move back to that location and heal to the max. If there is an actor stand on Bowser's original position, I decide let Bowser do not move in this case. Because if 8 exits of that position is occupied by other actor or item, a bug may occur when I reset the game, so to prevent that condition happen, I will not let Bowser go back to its own position when game is reset if the position is occupied by other actors.

By extending Enemy class, the default behaviors of enemy can be override form its parent class (Enemy) directly without coding again. This obeys the design principle DRY and make the code more logic and easy to read. Besides that, Bowser class has its own responsibility which it performs the behavior of Bowser only, so it obeys SRP(single responsibility principle). In addition, by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I'm now avoiding the use of magical numbers and literals.

7) Key

Why I choose to do it that way:

According to Assignment 3 requirements, Key is a class that extends from Item class and it is an item that drop by Bowser when Bowser is not conscious anymore, so by picking up the key, the actor will have capability HAVE_KEY in order to interact with PrincessPeach. In this class, I add constant class attributes, it follow "avoid excessive use of literals" principles as I'm now avoiding the use of magical numbers and literals. Besides that, Key class has its own responsibility which it performs the behavior of Key only, so it obeys SRP(single responsibility principle).

8) Status Enum

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

In Assignment 3, due to the new design of this game, I have added some new status in order to help this game function properly which are HAVE_KEY, DROP_KEY and CAN_FLY.

```
HAVE_KEY, // use this status to indicate that player has key or not
DROP_KEY, // use this status to indicate that the actor is Bowser or not
CAN_FLY, // use this status to indicate tha the actor is Flying Koopa or not
```

Why I choose to do it that way:

These status are useful in my implementation as, I can utilize the engine code provided to check the actor's capability to simplify our code.

The HAVE_KEY status is used to indicate whether this actor pick up the key or not. This status is initialized in Key class, by having this status, the actor can interact with PrincessPeach in order to end this game.

The DROP_KEY status is used to indicate whether an actor has the capability to drop a key, because after Bowser is killed, it will drop a key at that position, so if the Bowser is killed by invincible actor(invincible: remove the target straight away after it killed), we should use a if loop to check that, if the target has the capability to drop a key, then after it dead, we will add a key to that position.

The CAN_FLY status is used to indicate whether an actor has the capability to fly. In the game setting, FlyingKoopa can walk (fly) over the trees and walls(any highgrounds) when it wanders around, so by adding this capability to it, we can easily modify the canActorEnter() method in HighGround class in order to make sure that every actor who has the capability to fly can enter any highgrounds when it wander around. Besides that, this allow for extension as when more actors in the future has the same capability (i.e., can drop a key or can fly), we could reuse these constant.

9) Mature

What changed in the design rationale between Assignment 2 and Assignment 3 and Why:
In Assignment 3 requirements, the mature tree has a 50:50 chance of spawning either normal Koopa or Flying Koopa (after a successful 15% spawn rate), so to achieve this feature, I add a new method in this class which is spawnFlyingKoopa(). Besides that, in the tick method, after the 15% of spawn a Koopa, there is 50% to spawn a Koopa or a Flying Koopa, so a if else statement is added to spawn Koopa and Flying Koopa at that location.

```
if ((rand.nextInt( bound: 100) <= SPAWN_GENERAL_KOOPA_RATE) && !location.containsAnActor()){
    if ((rand.nextInt( bound: 100) <= SPAWN_NORMAL_KOOPA_RATE)) {
        spawnKoopa(location);
    }else{
        spawnFlyingKoopa(location);
    }
}
```

Why I choose to do it that way:
By doing so, we can make the mature tree spawn Flying Koopa successfully under correct spawn rate, and by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I'm now avoiding the use of magical numbers and literals.

10) WarpPipe

What changed in the design rationale between Assignment 2 and Assignment 3 and Why:
WarpPipe is a class extends from Ground class and implements Resettable. According to Assignment 3 requirements, the PiranhaPlant is grows on the warppipe, each warppipe will have one PiranhaPlant correspondingly. In the tick method, I check whether this game is reset by checking the capability of this warppipe, if this game is reset and the warppipe contains an enemy(only PiranhaPlant), the PiranhaPlant there will increase its max hit point. And if the warppipe there don't have any actor, I will add a new PiranhaPlant there.

Why I choose to do it that way:
By doing so, when the game is reset, if there is no actor stands on the warppipe, I will add a new PiranhaPlant there, if there is an actor which is enemy(only can be PiranhaPlant), I will increase its max hit point by 50. Besides that, Key class has its own responsibility which it performs the behavior of Key only, so it obeys SRP(single responsibility principle), and by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I'm now avoiding the use of magical numbers and literals.

11) HighGround

What changed in the design rationale between Assignment 2 and Assignment 3 and Why:
In Assignment 3, a new enemy which is FlyingKoopa can walk (fly) over the trees and walls(any highgrounds) when it wanders around, so in the canActorEnter() method, I add another capability which is CAN_FLY to indicate that every actor who has the capability to fly can enter any highgrounds when it wander around.

Why I choose to do it that way:
By modifying the canActorEnter() method in HighGround class in order to make sure that every actor who has the capability to fly can enter any highgrounds when it wander around.

12) VictoryAction

Why I choose to do it that way:
According to Assignment 3 requirements, VictoryAction is a class extends from Action class. When VictoryAction is execute, it will print a descriptive message in the menu to ask the player whether want to end this game or not. If the player choose to end this game, the actor will be remove and the game will be ended with a ending message. Besides that, VictoryAction class has its own responsibility which it performs the action of VictoryAction only, so it obeys SRP(single responsibility principle).

13) AttackAction

What changed in the design rationale between Assignment 2 and Assignment 3 and Why:
To achieve the feature of Assignment 3, I add a nested if loop inside the execute() method in the AttackAction class. When player has capability INVINCIBLE, that means the actor will kill the enemy instantly(remove it from the game map directly), but for the actor who has capability to drop a key(i.e, Bowser), we cannot remove it from map straight away, when the actor who has capability to drop a key is dead, it should drop a key, so in this nested if loop, we check that if the actor is invincible and if the target has capability DROP_KEY, the game map will add a Key at the position of the actor who has capability to drop a key, then only remove the actor from game map.

Why I choose to do it that way:
By doing so, we can make sure that when the actor who has capability to drop a key is dead, it will not be removed from the map straight away without drop the key, which make sure the game will run properly under this condition.

14) FireAttackBehaviour

Why I choose to do it that way:

FireAttackBehavior is a class which implements Behavior and used by Bowser. When player come close to Bowser, it will attack player automatically by using FireAttackBehavior. I will just use getAction method in this class to return the FireAttackAction to attack player automatically. Also, it obeys the SPR code since it has its own single responsibility.