REQ3: Enemies

Overview:
To achieve this feature, there will be three new classes (i.e., Koopa, Enemy and AttackShellAction) created in the extended system, and five existing classes (i.e., Goomba, Status, Floor, AttackAction and AttackBehavior) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) Enemy

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
In Assignment 2, I created a new abstract class called Enemy to be the parent class of other enemies(Goomba and Koopa). In Assignment 1, there is no Enemy class created. Because of that, there are a lot of repeated code in Goomba class and Koopa class. It is not a good practice since it does not follow the design principle DRY(don't repeat yourself). To get rid of this, an abstract class is needed. Besides that, I add constant class attributes, so less magical numbers and literals is used.

```
/**
 * Enemy is an abstract class represents the enemies in the game. It is a class that extends from the Actor.
 * There are two types of enemies in this game, which is Goomba and Koopa.
 */
public abstract class Enemy extends Actor implements Resettable {
```

```
/**
 * KEY of the hashmap which also indicates the order of 3 behaviors.
 */
protected static final int FIRST_PRIORITY = 1; // key of hashmap
protected static final int SECOND_PRIORITY = 2; // key of hashmap
protected static final int THIRD_PRIORITY = 3; // key of hashmap
```

Why I choose to do it that way:
Enemy is an abstract class represents the enemies in the game. As we know, abstract class provides the default behaviour for sub classes so that all child classes should have performed the same functionality. By adding an abstract enemy class, all the common behaviors of enemies 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 enemies 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 enemy added, you do not have to modified enemy class but allow additional class to extends it, hence it allow extension. 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.

Apart from that, it adheres to the Liskov substitution principle, as all methods in the Enemy class retain the meaning of their parent class, Actor.

2) Goomba

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
In Assignment 2, instead of let Goomba class extends Actor class, I make Goomba class extends Enemy class. In Assignment 1 I make too many repeated code with Koopa, which disobey the design principle DRY(don't repeat yourself).

```
/**
 * Goomba class is a class represents the enemies in this game. It is a class that extends from the Enemy class.
 * Goomba can move around in the game map but cannot enter floor.
 * Once goomba is engaged in a fight (the Player attacks the enemy or the enemy attacks player --
 * when the player stands in the enemy's surroundings), it will follow the Player.
 * It causes 10 damages to player with 50% hit rate.
 * To make sure the map is clean and not too overcrowded, goomba will has a 10% chance to suicide each round of this game.
 */
public class Goomba extends Enemy {
```

Why I choose to do it that way:
By doing so, many 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, Goomba class has its own responsibility which it performs the behavior of goomba only, so it obeys SRP(single responsibility principle). In addition, it follows Liskov Substitution Principle too, which means the meaning of parent's behaviors is maintained, 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.

3) Koopa

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
Same with Goomba class, instead of let Koopa class extends Actor class, I make Koopa class extends Enemy class. In Assignment 1 I make too many repeated code with Goomba, which disobey the design principle DRY(don't repeat yourself).

```
/**
 * Koopa class is a class represents the enemies in this game. It is a class that extends from the Enemy class.
 * Koopa can move around in the game map but cannot enter floor.
 * Once koopa is engaged in a fight (the Player attacks the enemy or the enemy attacks player
 * -- when the player stands in the enemy's surroundings), it will follow the Player.
 * It causes 30 damages to player with 50% hit rate and koopa has the same behaviors with goomba.
 * When koopa is not conscious(means it is defeated), it will hide inside its shell, and its
 * character will change from 'K' to 'D'.
 * Player cannot attack it anymore, and all the behaviors will removed from koopa(attack/follow/wander).
 */
public class Koopa extends Enemy {
```

Why I choose to do it that way:
By doing so, many 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, Koopa class has its own responsibility which it performs the behavior of koopa only, so it obeys SPR(single responsibility principle). It also adheres to the Liskov Substitution Principle, which ensures that the meaning of parental acts is preserved, 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) Status Enum

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

For this enumeration class, instead of states two capabilities(IM_GOOMBA and IM_KOOPA), I make it become IS_ENEMY. Since both of Goomba and Koopa are enemies, it is not necessary to put two capabilities for them since they almostly doing the same thing, have the same behavior.

```
    IS_ENEMY, // use this status to determine all enemies
```

Why I choose to do it that way:

Instead of creating class attributes to indicate whether this actor is enemy or player, it is better to make use of the enumeration class to make our code easier to read. Besides that, I also can utilize the engine code provided to check the actor's capability to simplify our code. Besides that, if there are more enemies in the future, it is impossible for me to state the capability of themselves one by one, it will become a big trouble.

Other than that, IS_ENEMY is useful in my implementation, I add it to Enemy's capability, so whatever enemies which extends Enemy class will have this capability to indicate they are enemies. By doing so, when I state the capability(i.e. Enemies cannot enter floor), instead of writing actor.hasCapability (IM_GOOMBA) &&actor.hasCapability (IM_KOOPA), I can code actor.hasCapability(IS_ENEMY) straight away. It obeys the design principle DRY(don't repeat yourself).

5) Floor

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

In Assignment 2, I change the capability inside the code. Instead of using IM_GOOMBA && IM_KOOPA, IS_ENEMY is more clear and easy to code. By using IM_GOOMBA && IM_KOOPA will brings me many trouble.(i.e. if there are more enemies in the future implementation, very hard for me to code one by one) Besides that, it doesn't follow the design principle.

```
// Enemy cannot enter floor
else if (actor.hasCapability(Status.IS_ENEMY)){
    canEnter = false;
}
return canEnter;
```

Why I choose to do it that way:
By doing so, my code will be much more shorter and readable than before. Also, it obeys the SPR code since it has its own single responsibility. I don't need to code all the capability of each enemy one by one, it makes my code more logic and easy to debug because if too many capabilities here, it will be very easy to miss one of them or typo.

6) AttackAction

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
There is no changes in this class of REQ3 between Assignment 1 and Assignment 2.

7) AttackBehavior

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
In Assignment 1, AttackAction class extends Action class and implements Behavior, in assignment 2, I change it to implement Behavior only. Since there is not necessary for me to print the menuDescription and put the different actions of Goomba and Koopa in execute here, I can remove the 'extends Action'.

```
public class AttackBehaviour implements Behaviour {
```

Why I choose to do it that way:
Instead of creating two attack method/behavior separately for Koopa and Goomba, I will just use getAction method in this class to return the AttackAction to attack player automatically. Also, it obeys the SPR code since it has its own single responsibility. By doing so, there will be much lesser repeating appear since the attack action of Goomba and Koopa is similar. It obeys the deaign principle DRY.

8) AttackShellAction

What changed in the design rationale between Assignment 1 and Assignment 2 and Why:
There is no changes in this class of REQ3 between Assignment 1 and Assignment 2.