

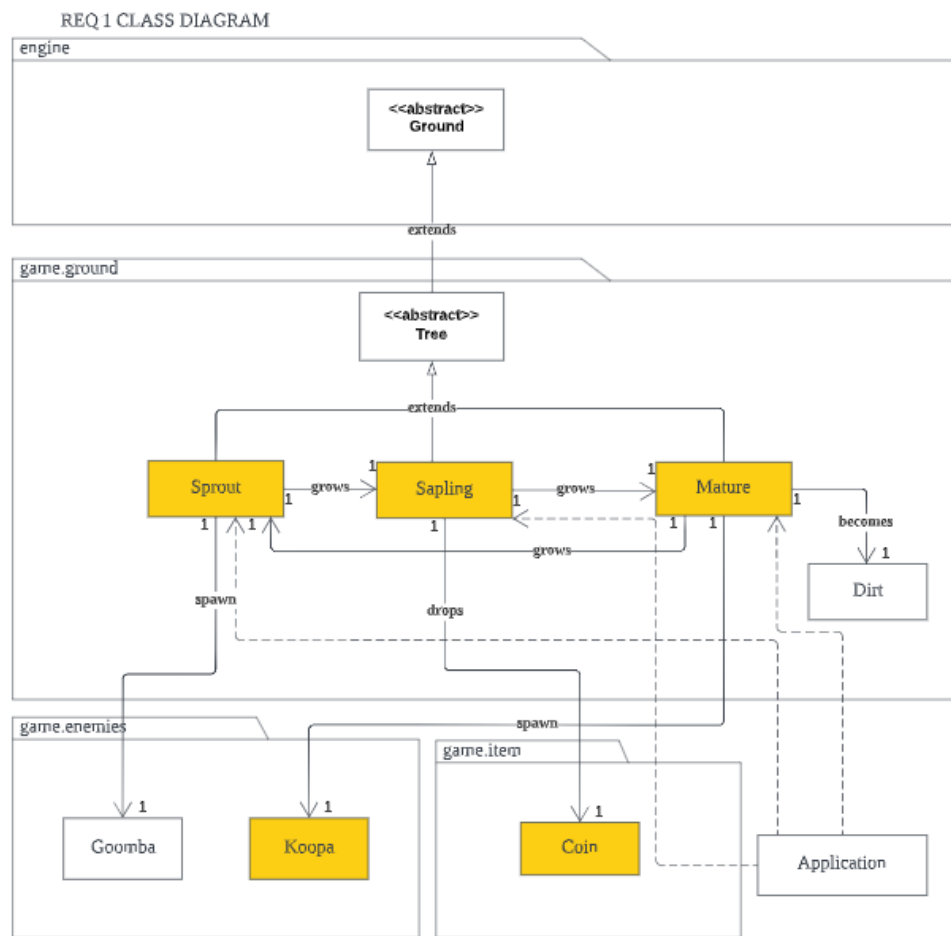
Design Rationale For Assignment 1

LAB 12 Team 4

Seng Wei Han (32229070)

Chua Jun Jie (32022425)

REQ1: Let it grow! 🌳



(The screenshot shown above is the UML designed for requirement 1)

Design Implementation

Tree Class:

- 1) The purpose of making the Tree class to be an abstract class is because we knew that we will not instantiate the Tree class in any of our code. Such implementation helped us to achieve abstraction.
- 2) From requirement 1 of the assignment, it is stated that a Tree has three stages, and each stage has a unique spawning ability. On the other hand, each spawning stage will also have additional characteristics, for instance each sprout has 10% chance to spawn Goomba. Therefore, instead of doing all the spawning in one Tree class which will cause the Tree class to be so called the God class, I have decided to split the classes into 3 small new classes namely Sprout, Sapling and Mature and make it inherit the Tree class. In each of these new subclasses, it will have their own responsibility so that this design will adhere to the Single Responsibility Principle (SRP). On the other hand, I have also planned to declare an abstract method called tick in Tree class so that any subclasses that inherits the Tree class will need to implement the tick method. By doing this way, we can prevent too many if-else statements in one Tree class so that it will not violate the second principle which is the Open-closed Principle.
- 3) Since sprout, sapling and mature are the 3 stages that a tree can experience, meaning to say the 3 classes will have the same characteristics for instance, having the same method implementation for the following method:

```
@Override  
public boolean canActorEnter(Actor actor)
```

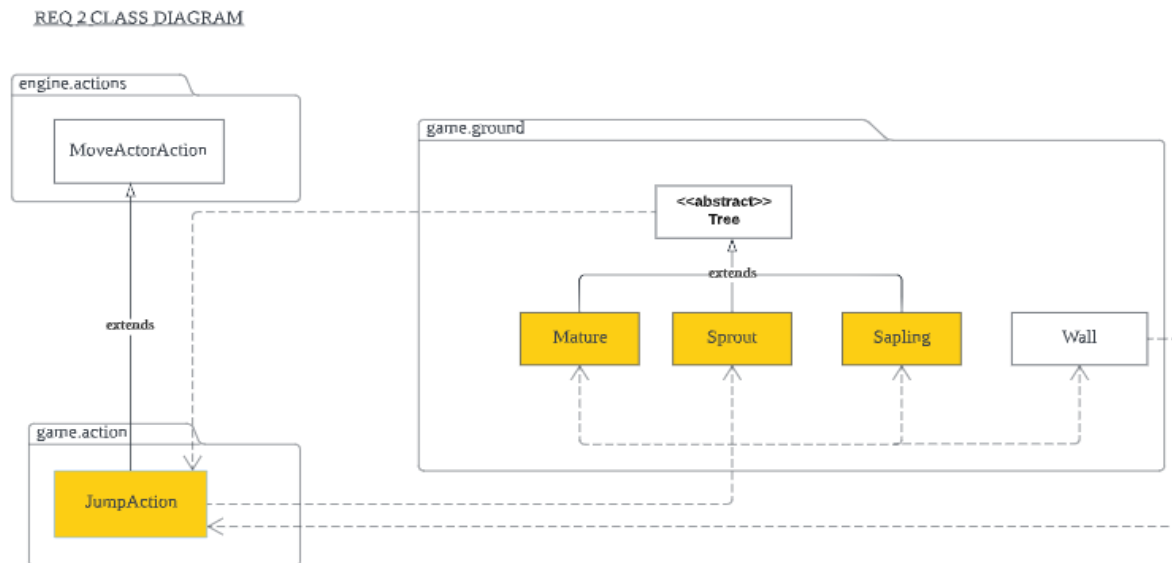
```
@Override  
public ActionList allowableActions(Actor actor, Location location, String direction)
```

Thus, it's a better idea to implement those method in the Tree class and allow the other 3 new classes (Sprout, Sapling and Mature) to inherit the method from their parent class. By using this implementation, we can ensure that DRY(Don't Repeat Yourself) Principle will not be violated.

Relationship :

- 1) The dependency relationship between Application and the three subclasses of Tree is due to the fact that for each new ground that we created, we will need to add into the FancyGroundFactory constructor.
- 2) The association relationship between Sprout and Goomba , Sprout and Sapling is due to the fact that each sprout can spawn goomba, so we will need to have Goomba as an attribute in the Sprout class and the same goes for Sprout and Sapling.
- 3) The association relationship between Sapling and Coin , Sapling and Mature is because the requirement states that a sapling can have a chance to drop a coin and a sapling will grow into a mature tree, therefore we will need to have those attributes in the sapling class.
Since coins are needed, we will need to create a new class for coins.

REQ2: Jump Up, Super Star! 🌟



(The screenshot shown above is the UML designed for requirement 2)

Design implementation

Jump Action :

- 1) Since requirement 2 is about jumping features, then we definitely need to have a `JumpAction` class implementation. This `JumpAction` class will inherit (extends) the `moveActorAction`. The ground inside the UML class diagram is basically the ground that does not allow player to enter without using the jump action . Therefore, for each ground that player walked through, we will need to do a checking inside the execute method .The checking needs to be done due to the fact that certain ground does have certain success rate which means if the player failed to jump to that particular ground, it will get inflicted a certain amount of damage by that ground. We need to set an allowable action to the ground that allows the player to use the jumping action. Since `Sapling`, `Sprout` and `Mature` are inheriting the `Tree` class, we can just set the allowable actions in `Tree` class and also for `Wall` class. This design implementation does satisfy the requirement for the Single Responsibility Principle (SRP) since this class is only responsible for the jumping action of a player . As for the case of the `SuperMushroom`, we can

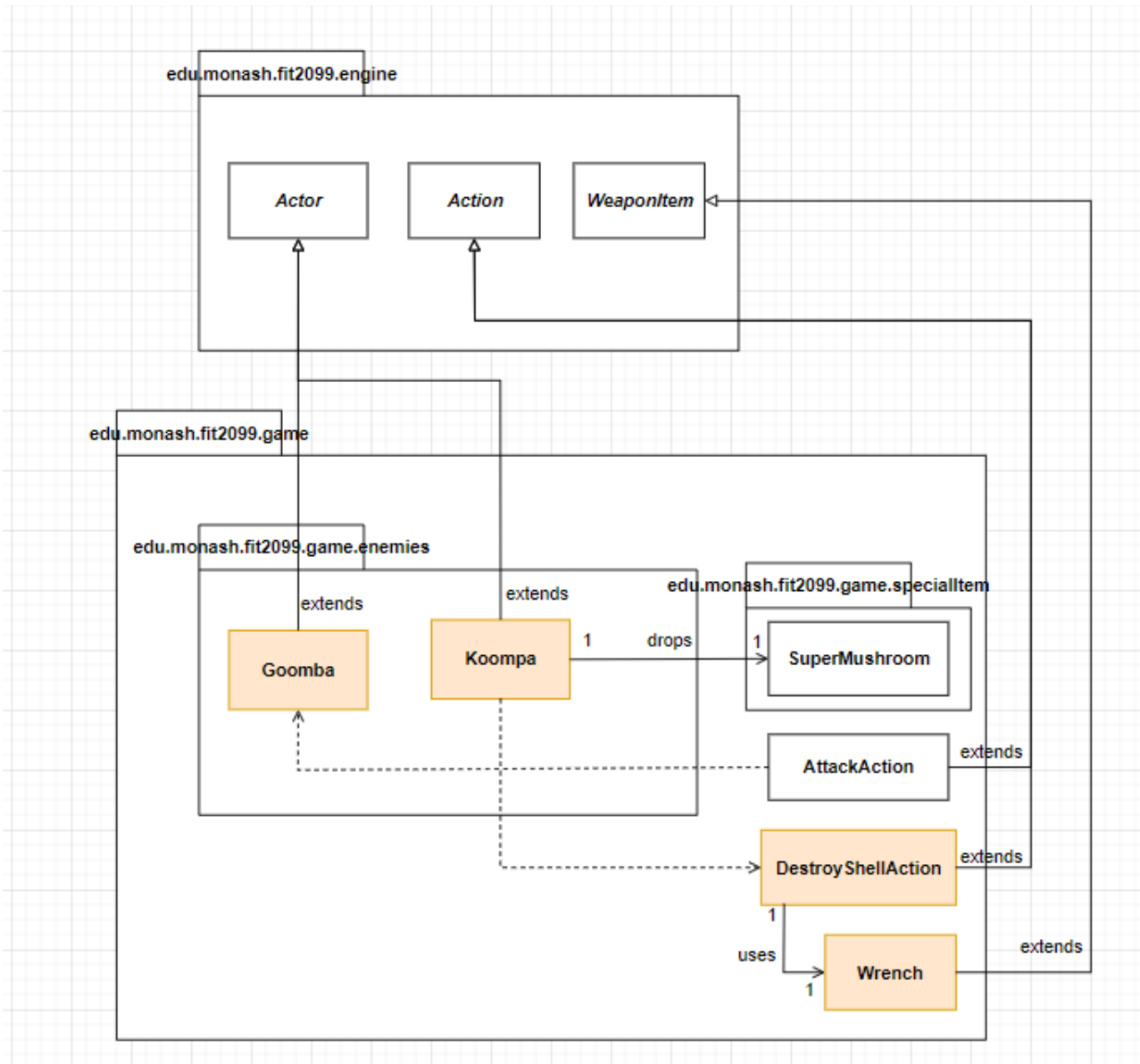
just check whether a player has the status when it consumes super mushroom, if it does then the player will have a 100% success rate of jumping through those grounds.

Relationship :

- 1) The dependency relationship between Tree and JumpAction, Wall and JumpAction is because we need to add a new instance of the JumpAction to the actionList inside the allowableActions method.
- 2) The dependency relationship between JumpAction class and the 4 other ground is due to the checking process inside the execute method where we need to check whether the ground in the player exit surrounding is one of that four ground , if it is, then it will trigger the jump action in the console to allow the player to jump through that ground.

REQ3: Enemies

UML Diagram



Overview

REQ3 is all about the enemies in this game. There are two enemies in the game, Goomba and Koopa. Each of them has their individual game mechanics that will be implemented in this requirement.

Goomba (g)

- Starts with 20HP.
- Deal 10HP upon every successful attack, has 50% chance to register a successful attack.
-
-

Has 10% chance to despawn (removed) from the map in every turn.

Koomba (K)

- Starts with 100HP.
- Deals 30HP on every successful hit, has 50% chance to land a successful hit.
- When defeated, Koomba will turn into Dormant state (D), and it stays on the ground and does nothing.
- Wrench must be used to destroy the shell (when Koomba is in Dormant state). Wrench deals 50 damage and 80% hit rate.
- Destroying the shell drops Super Mushroom.

Once engaged in combat mode (either player attacks the enemy or the other way around), the enemy will follow the player until it is defeated.

Implementation

Goomba

The Goomba class is given already, thus we don't need to create another class just for the character. So, at first, Goomba's basic stats will be added to the class as its global attributes

```
public class Goomba extends Actor {  
    /*  
    * CONSTANT 10% chance to be removed from the map  
    * sets HP attributes  
    * sets HP per damage, and hit percentage  
    */  
}
```

For every turn, Goomba has 10% chance to be removed from the map. Thus, we can implement a method to remove Goomba if Goomba hits the 10% chance.

```
public void removeGoomba () {
    /* if goomba hits the CONSTANT 10% chance created
    * goomba is removed from the map.
    */
}
```

Koomba

Since there is no given Koomba class before, we shall create its own dedicated class. Similar to Goomba, Koomba's basic stats will be declared as a global attribute.

```
public class Koomba extends Actor {
    /*
    * sets HP attributes
    * sets HP per damage, and hit percentage
    */
}
```

There is another similar check that is needed to perform if the player manages to defeat the Koomba. If the player defeats Koomba, Koomba will turn into Dormant state. We can put Dormant state in the Status class, and we implement the Status into the AttackAction as well.

```
public enum Status {
    HOSTILE_TO_ENEMY, // use this status to be considered hostile towards enemy (e.g., to be attacked by enemy)
    TALL, // use this status to tell that current instance has "grown".
    DORMANT // use this status to tell Koomba is in Dormant state
}
```

Since Koomba has a special state when it is defeated, a special class dedicated to deal with destroying the shell of Koomba.

```
public class DestroyShellAction extends Action {
    /* code goes here
    */
}
```

Justification of Implementations

Goomba — Advantages

The implementation method written above uses several design principles, mainly Open-closed Principle (OCP) and Single Responsibility Principle (SRP).

Goomba has its own dedicated class, and it is a subclass of Actor superclass. This is essentially OCP as whatever changes Goomba class has, it will not affect other classes, especially Actor superclass.

SRP is used when Goomba's attack action is placed inside AttackAction class. This is because AttackAction has one and only one purpose/responsibility only: to create and register the attack action made by one actor to a target.

Goomba — Disadvantages

NA

Koomba — Advantages

Similar to Goomba's justification written above, Koomba also uses OCP and SRP while implementing its methods.

Since Koomba has its own special state when defeated (but not killed), we put the status inside the Status enum class so that we can avoid using excessive literals.

Koomba — Disadvantages

NA

-
- The diagram illustrates the class hierarchy and relationships for the Super Mario Bros. game engine and game components. It is organized into several packages:
- edu.monash.fit2099.engine**: Contains the base classes `Actor`, `Item`, `Ground`, and `Action`.
 - edu.monash.fit2099.game**: Contains the `Player` class, which extends `Actor`. It also contains a package `edu.monash.fit2099.game.consume`.
 - edu.monash.fit2099.game.consume**: Contains the `ConsumeSuperMushroom` and `ConsumePowerStar` classes, which extend `ConsumeItem` (which in turn extends `Item`).
 - edu.monash.fit2099.game.specialItem**: Contains the `MagicalMushroom` and `PowerStar` classes, which extend `Item`.
 - edu.monash.fit2099.game.world**: Contains the `Dirt` class, which extends `Ground`. It also contains a package `edu.monash.fit2099.game.world.action`.
 - edu.monash.fit2099.game.world.action**: Contains the `JumpAction`, `AttackAction`, and `Coin` classes, which extend `Action`.
- Key relationships and associations include:
- `Player` (1) consumes (1) `MagicalMushroom`.
 - `Player` (1) consumes (1..*) `PowerStar`.
 - `ConsumeSuperMushroom` (1) consumes (1) `MagicalMushroom`.
 - `ConsumePowerStar` (1) consumes (1) `PowerStar`.
 - `Dirt` (1) consumes (1) `PowerStar`.
 - `Dirt` (1) consumes (1) `JumpAction`.
 - `Dirt` (1) consumes (1) `AttackAction`.
 - `Dirt` (1) consumes (1) `Coin`.

The player's display character turns to uppercase ($m \rightarrow M$).

- The player can jump freely with no fall damage and 100% jump success rate.

The effect lasts indefinitely till the player receives any damage (from the enemy). Once the effect wears off, the player's display character turns back to lowercase ($M \rightarrow m$).

Power Star (*)

When a player consumes Power Star:

- The player does not need to "jump" to higher ground. Instead, the higher ground will be destroyed and turn into dirt.
- Destroyed ground drops a Coin (5\$).
- The player receives immunity and does not take any damage from enemies.
-
- Under the effect of Power Star, a successful attack from the player will always kill the enemy.

Power Star lasts for 10 turns. If within the 10 turns it does not get picked up by a player, it will disappear from the game. If Power Star is picked up by the player, it will last for 10 turns in the player's inventory, before disappearing from the player's inventory. When the player consumes Power Star, the effects

written above last for 10 turns only.

Implementation

Super Mushroom

In order to implement Super Mushroom as a Magical Item into the game, a class called SuperMushroom is created, and it is a subclass of the Item abstract class, as shown below.

```
public class SuperMushroom extends Item {
    /* creates a super constructor since its a subclass
    * the implementation continues below
    */
}
```

Since the SuperMushroom effect lasts for an indefinite time until the player receives damage from enemies, we can utilize the given enum class Status and use it to set the status of the player.

```
public enum Status {
    HOSTILE_TO_ENEMY, // use this status to be considered hostile towards enemy (e.g., to be attacked by enemy)
    TALL, // use this status to tell that current instance has "grown".
    EFFECT_SUPER_MUSHROOM // example, use this status when the player consumes super mushroom
}
```

We can create a ConsumeItem abstract class, and it is a subclass of Action abstract superclass. Then we can create a subclass of ConsumeItem named ConsumeSuperMushroom class.

```
public abstract class ConsumeItem extends Action {
    //abstract class for other items to be consumed
}
```

```
public class ConsumeSuperMushroom extends ConsumeItem {
    /* a specific class to run the consume super mushroom action
    * when run, increase player HP by 50
    * set status EFFECT_SUPER_MUSHROOM
    * set player display character uppercase
    */
}
```

The effect last until the player receives damage from the enemies, therefore, we can implement it inside the given AttackAction class.

```
public class AttackAction extends Action {

    /*
    * ... given code in the class
    */

    @Override
    public String execute(Actor actor, GameMap map) {

        /*
        * given method code ...
        */

        /* check if target has EFFECT_SUPER_MUSHROOM
        * if yes, remove the target's status,
        * reset the target's display character to lowerCase
        */
    }
}
```

```

    */

    /*
    * ... code continues below
    */
}

/*
* ... given code in the class
*/
}

```

Not only that, if the effect hasn't worn off, the player can jump to any place with 100% success rate and no fall damage will be taken. We can also implement this in the JumpAction class.

```

public class JumpAction extends Action {
    /*
    * some code...
    */

    /* a method to check if the actor has EFFECT_SUPER_MUSHROOM
    * if yes, the actor jumps with 100% success rate
    * and does not take any fall damage
    */
}

```

Power Star

In order to implement Power Star class, we would do exactly the same as to how we implemented Super Mushroom.

```

public class PowerStar extends Item {
}

```

Likewise, when a Power Star is consumed, a status will be granted to the player. Unlike the status for Super Mushroom, the status for Power Mushroom must be displayed in the console.

```

public enum Status {
    HOSTILE_TO_ENEMY, // use this status to be considered hostile towards enemy (e.g., to be attacked by enemy)
    TALL, // use this status to tell that current instance has "grown".
    EFFECT_SUPER_MUSHROOM, // example, use this status when the player consumes super mushroom
    IMMUNITY //example, use this status when the player consumes power star
}

```

A similar consume class will also be implemented.

```

public class ConsumePowerStar extends ConsumeItem {
}

```

We can utilize the special status effect to grants the subsequent special effects to the player. For example, under the special effect, each successful attack made by the player is an instant kill to the enemies.

```
public class AttackAction extends Action {

    /*
     * ... given code in the class
     */

    @Override
    public String execute(Actor actor, GameMap map) {

        /*
         * given method code ...
         */

        /* check if actor has IMMUNITY status, if yes
         * actor damage to target = a very large number,
         * deal damage to the target
         * else, do normal damage target
         */

        /*
         * checks if target has IMMUNITY status
         * if yes (that means the target is the actor), actor takes 0 damage
         * else, continue Super Mushroom code above
         */

        /*
         * ... code continues below
         */
    }

    /*
     * ... given code in the class
     */
}
```

Furthermore, we can use this status effect to make the jump effect as well. For example:

```
public class JumpAction extends Action {

    /*
     * some code..
     */

    /* special check for IMMUNITY status
     * if yes, actor does not need to jump, move normally to the tile
     * sets the ground to dirt, then drop the coin on the ground
     */
}
```

While high ground will be destroyed and convert to dirt, and drops a coin.

```
public class Coin extends Item {  
    /*  
    * some code...  
    */  
  
    /* special method for IMMUNITY status  
    * drops the coin on the ground  
    */  
}
```

Justification of Implementations

Super Mushroom — Advantages

The implementation method written above uses several design principles, mainly Open-closed Principle (OCP) and Single Responsibility Principle (SRP).

OCP is used when the SuperMushroom class is created and it is a subclass of the abstract class Item. This way, SuperMushroom will have its dedicated class, thus its own changes will not affect other classes.

Not only that, ConsumeSuperMushroom is a subclass of the ConsumeAction. This also follows the idea of OCP while implementing.

SRP is used when the player attacks any enemies after consuming a Super Mushroom. Since AttackAction class is given, it shall only be used as one single action only. We create an if-statement to check if the player has the effect status, if yes, the player's attack action is modified.

This if-statement is not and shouldn't be placed in the SuperMushroom class because it's an AttackAction, therefore it should be grouped and placed within the AttackAction class.

The special effect when consuming a Super Mushroom is placed inside the given enum special class. This prevents excessive use of literals, which will make the code extra messy.

Super Mushroom — Disadvantages

By putting an if-statement inside the AttackAction class, this means that every time when a non-actor (Goomba/Koopa) attacks the player, the if-statement will always run and check if the player is under Super Mushroom effects. This is inefficient and resources are wasted to run this check.

Power Star — Advantages

Similar to the advantages of the implementation method of Super Mushroom, OCP and SRP design principles are also used when implementing Power Star item. OCP is used when a dedicated

PowerStar class is created, and also ConsumePowerStar subclass is created, and SRP is also used when the attack action is modified and placed inside AttackAction class, with similar reasoning written

above. The special effect of Power Star is also created with the same reason written on the Justification of implementation of Super Mushroom.

Power Star — Disadvantages

Similar to Super Mushroom's disadvantage, two if-statements are placed within the execute() method, one is for dealing damage, another one is within the SuperMushroom check . This is very inefficient as every time an attack action runs, it will spend time checking these conditions.

Furthermore, if the player has *IMMUNITY* status, whenever the player performs a jump action, the jump action will be overwritten to normal move action, and then it will call the special method in Location class that sets the ground to dirt, and then Location special method will call Coin class special method to drop a coin on the ground. This is similar to a "spaghetti code", and it is an inefficient way to code out the effects. If there is a change in the method, it will affect the subsequent methods and cause a chain of errors.