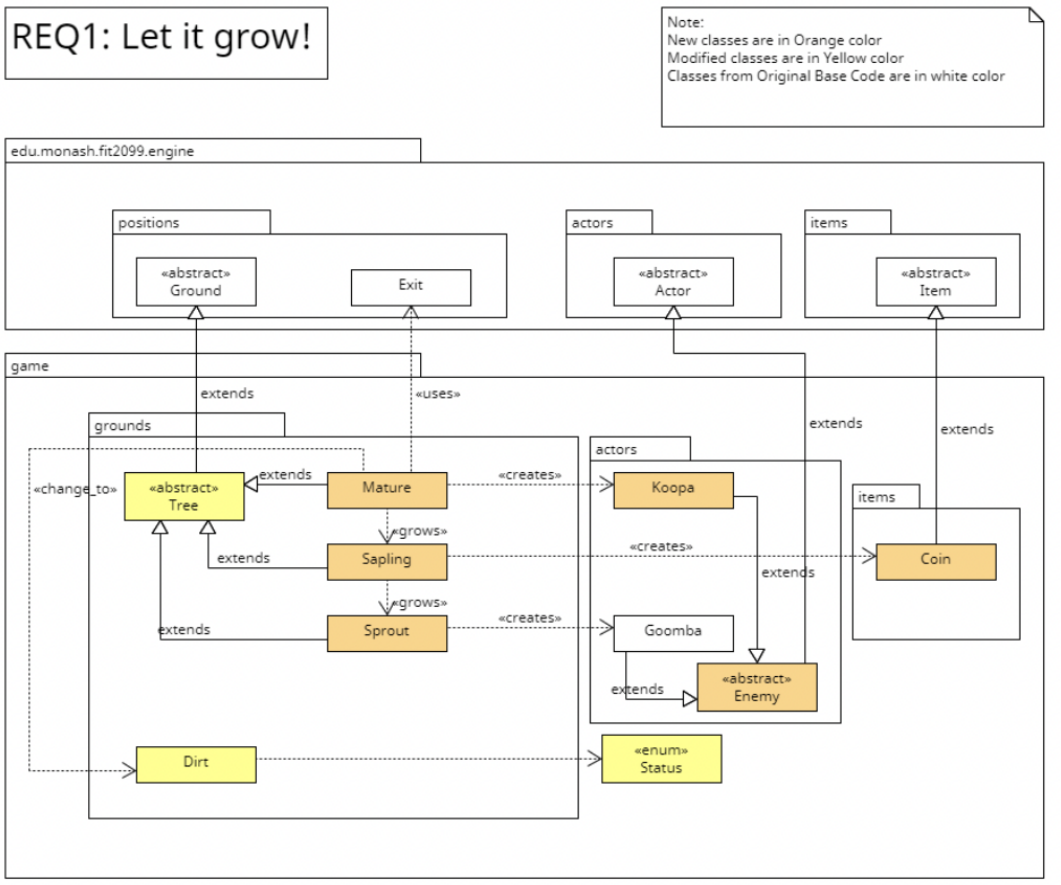
REQ1: Let it grow!

Overview:

To achieve this feature, there will be six new classes (i.e., Koopa, Mature, Sapling, Sprout, Enemy and Coin) created in the extended system, and three existing classes (i.e., Status, Tree, and Dirt) will be modified. The design rationale for each new or modified class is shown on the following pages. Please note that although Tree class extends HighGround (i.e., a new class), however, the purpose of creating HighGround is to achieve the feature in REQ2, therefore HighGround class will not be considered in the design rationale and UML diagrams for REQ1.



Overall responsibility for New and Modified classes

1) Tree: An abstract class that represents the tree in this Game

2) Sprout: A class that represents the sprout tree in the Game Map

3) Sapling: A class that represents the sapling tree in the Game Map

4) Mature: A class that represents the mature tree in the Game Map

5) Dirt: A class that represents the dirt in the Game Map

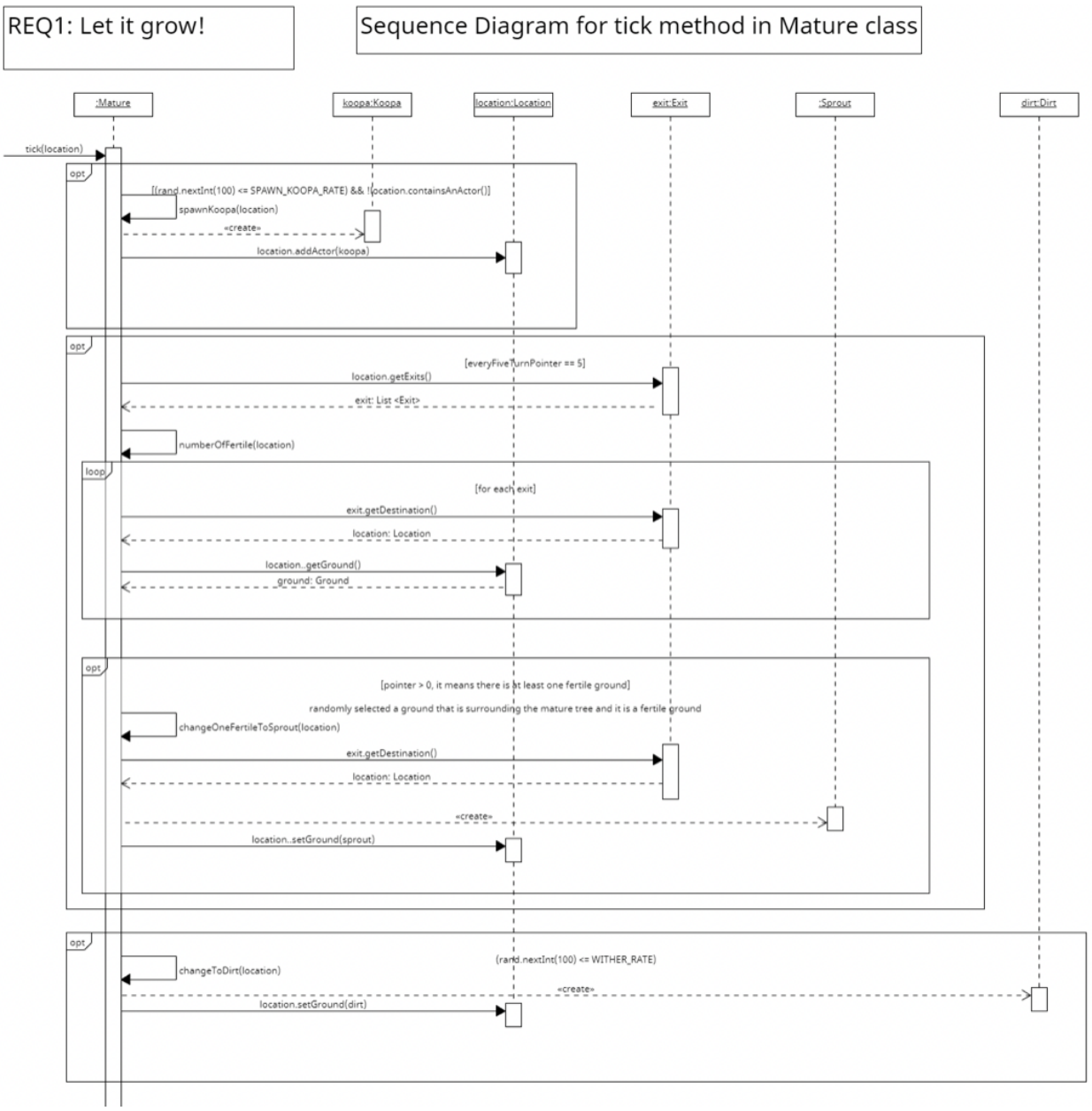
6) Coin: A class that represents the coin item in the Game Map

7) Status: This enum class is to give 'buff' or 'debuff'. It is also useful to give a 'state' to abilities or actions that can be attached-detached

8) Koopa: A class that represents one of the enemies in the Game, which is Koopa

9) Enemy: Enemy is an abstract class represents the enemies in the game.

class diagram for REQ 1



sequence diagram for req1

1) Tree class

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

In assignment 1, I did not create Mature, Sapling, Sprout class for each stage of the Tree, instead I only used the Tree class to handle all properties of each stage of a Tree. To keep track of the status of a Tree instance, I use the constants in enum class. As a result, there are too many if-else statements, repeated code, overused of addCapability, removeCapability and hasCapibility methods. This makes my code too complicated and less maintainable and readable. Therefore, in assignment 2, I had decided to make this Tree class to be an abstract class. The reason to make this class abstract is because we should not create any instance of Tree as Tree class is

only used to provide a common, implemented functionality for its subclasses. After that, I will create the Mature, Sapling, Sprout class to be the subclass of Tree class. This approach is better since each stage has a unique spawning ability and their properties are different (e.g., display character, success rate of a jump, fall damage etc), hence it is better to have different classes to handle this situation. Besides that, by doing so, whenever any of these subclasses have common code, I can avoid repeated code by putting that piece of code in their parent class (i.e., Tree class).

Why I chose to do it that way:

With this design, we are following the Open Closed Principle as when more stages of a Tree are introduced to the game, we can let that new class extend this class without modifying any existing code. Hence, we are allowing our class to be open for extension and close for modification. Besides that, we also create a class attribute to store the fixed value. Hence, we are adhering to the “avoid excessive use of literals” principle. Moreover, we are adhering to the DRY principle as all the subclasses of Tree class will not have any repeated code. In addition, we are following the Reduce Dependencies Principle as in REQ7, when reset action is called, we can remove the Tree instance (i.e., with 50% chance) by directly letting the Tree class implement Resettable. Since we do not need to let any of these subclasses implement Resettable, we are reducing the dependencies in our design. Besides that, inside the Tree class, I had added a helper method to convert the tree instance to dirt at that location. For this method, I had checked the pre-condition of the input parameter (i.e., currentLocation) such that it cannot be null, else it will throw a IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allow the developer to know where the problem is.

2) Sprout class

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

In assignment 1, instead of creating a Sprout class to indicate a tree in the map to become a sprout tree, I add the capability SPROUT to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class. Therefore, in assignment 2, I will create a Sprout instance if a tree in the map becomes a sprout tree at that location. Details to keep track of the age of a sprout is shown in the code.

Why I chose to do it that way:

By doing so, we are adhering to the Single Responsibility Principle as the method within Sprout class only shows the properties of a Sprout. Furthermore, for each of the properties of a Sprout instance (e.g., success rate of a jump, display character, the rate to spawn a goomba 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. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Sprout class still preserve the meaning from its parent class, Tree. In addition, in Sprout class, I had created several helper functions to perform the following operations such as change the sprout to sapling and spawn goomba at that location. For all the helper functions, I had to check their input to see if it is not null, else it will throw a IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allow the developer to know where the problem is.

3) Sapling class

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

Like sprout class, in assignment 1, instead of creating a Sapling class to indicate a tree in the map to become a sapling tree, I add the capability SAPLING to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class. Therefore, in assignment 2, I will create a

Sapling instance if a tree in the map becomes a sapling tree at that location. Details to keep track of the age of a sapling is shown in the code.

Why I chose to do it that way:

Similarly, we are adhering to the Single Responsibility Principle as the method within Sapling class only shows the properties of a Sapling. Furthermore, for each of the properties of a Sapling instance (e.g., success rate of a jump, display character, the rate to spawn a coin 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. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Sapling class still preserve the meaning from its parent class, Tree. Besides that, like Sprout class, I had checked the input parameter for each helper function (e.g., spawn a coin and change to mature tree), if any of them is null, then else it will throw a IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allow the developer to know where the problem is.

4) Mature class

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

Similar to other subclasses of Tree class, in assignment 1, instead of creating a Mature class to indicate a tree in the map to become a mature tree, I add the capability MATURE to a tree instance. However, I realized this kind of design is a bad practice, the reason had been stated in the first paragraph of Tree class. Therefore, in assignment 2, I will create a Mature instance if a tree in the map becomes a Mature tree at that location.

Why I chose to do it that way:

Likewise, we are adhering to the Single Responsibility Principle as the method within Mature class only shows the properties of a Mature. Furthermore, for each of the properties of a Mature instance (e.g., success rate of a jump, display character, the rate of wither in every turn 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. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the Mature class still preserve the meaning from its parent class, Tree. Additionally, like Sprout class, I had checked the input parameter for each helper function (e.g., spawn koopa and change a fertile ground to dirt and so on), if any of them is null, then else it will throw a 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.

5) Enemy class

Enemy class is an abstract class that represents the enemies in this game. However, since Enemy class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ3) and in this REQ, it only creates the instance of Enemy’s subclasses without considering any features, therefore the design rationale for Enemy class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Enemy class.**

6) Koopa class

Koopa class is a class that represents one of the enemies in this game, which is Koopa. However, since Koopa class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ3) and in this REQ, it only creates the Koopa instance without considering any features of a Koopa instance, therefore the design rationale for Koopa class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Koopa class.**

7) Coin class

Coin class is a class that represents the coin item in this game. Similar to Koopa class, Coin class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ5) and in this REQ, it only creates the Coin instance without considering any features of a Coin instance, therefore the design rationale for Coin class will not be available in this section. **Hence, please go to the REQ5 section in the following pages to see the design**

rationale for Coin class.

8) Dirt class

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

No changes are made between Assignment 1 and Assignment 2.

9) Status class

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

In the previous assignment, I created a class GroundStatus to store the constant IS_FERTILE. However, I realized the GroundStatus class is redundant as I could simply put IS_FERTILE in the status class. Hence, by doing so, I am able to avoid creating purposeless classes.

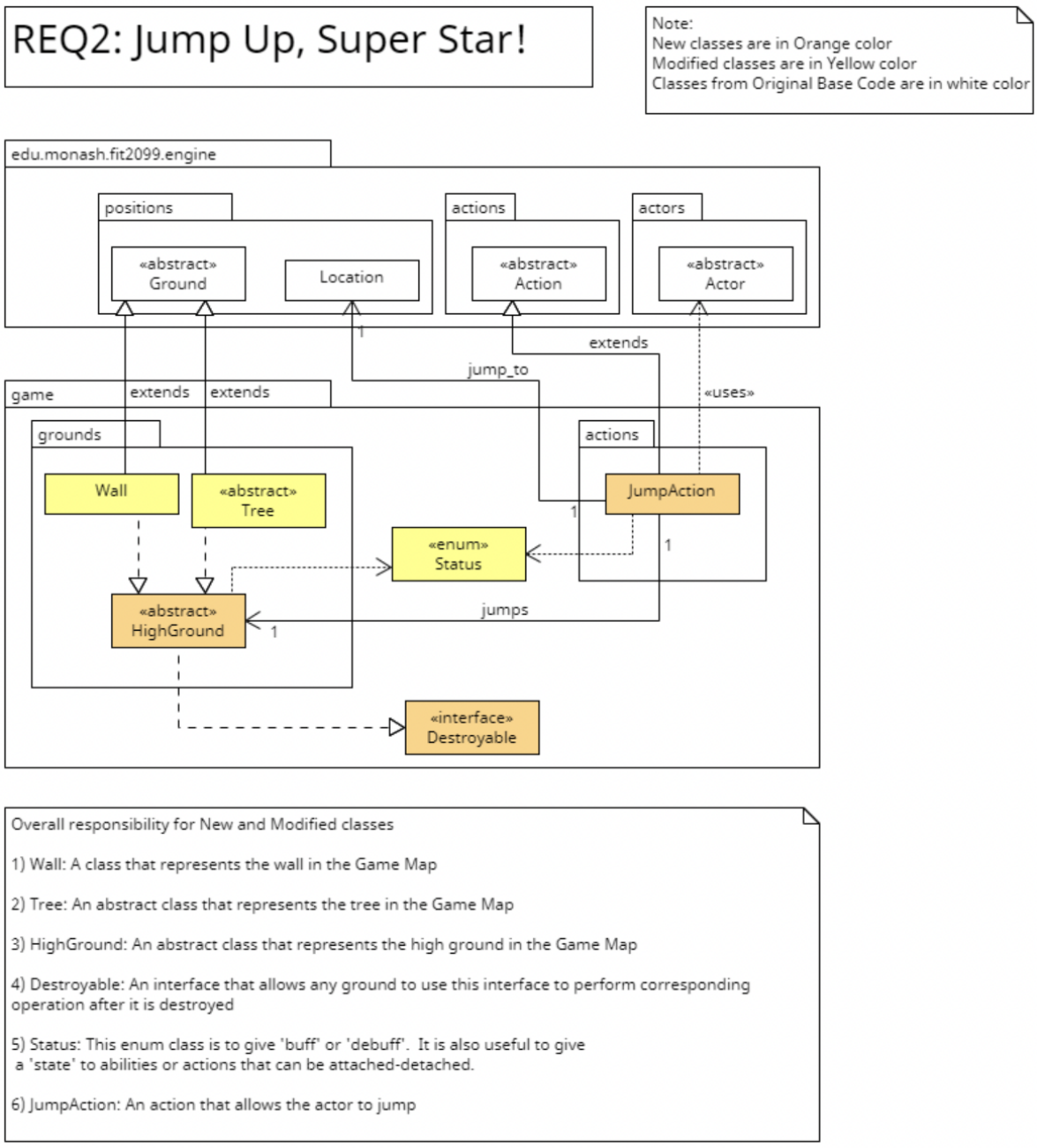
Why I chose to do it that way:

According to the assignment specification for this requirement, it says that for each Mature instance, for every 5 turns, it can grow a new sprout in one of the surrounding fertile grounds and the only fertile ground currently is dirt. However, it means that we need to figure out a way to check if the surrounding ground is a fertile ground, otherwise based on the game logic, we should not spawn the sprout if it is not a fertile ground. Hence, I had decided to add this IS_FERTILE constant in Status class. Hence, whenever we initialise a fertile ground, we can add this constant to its capability to indicate that it is a fertile ground. Thus, if we want to grow a sprout on a particular ground, we can determine if it is a fertile ground by checking whether it has the capability IS_FERTILE. In this case, we are using constant to specify that a Ground is a fertile ground. By doing so, we can avoid excessive use of literals as we do not have to create any local attributes within Dirt class for example to indicate that all Dirt instances are considered as fertile ground. This implementation also provides us the flexibility for future development. For instance, if there are more types of fertile ground that extends Ground class introduced to the game, we could simply add this constant as the capability of it. We also adhere to the Single Responsibility Principle as this enumeration class is only responsible to store the constant that will be used during our implementation.

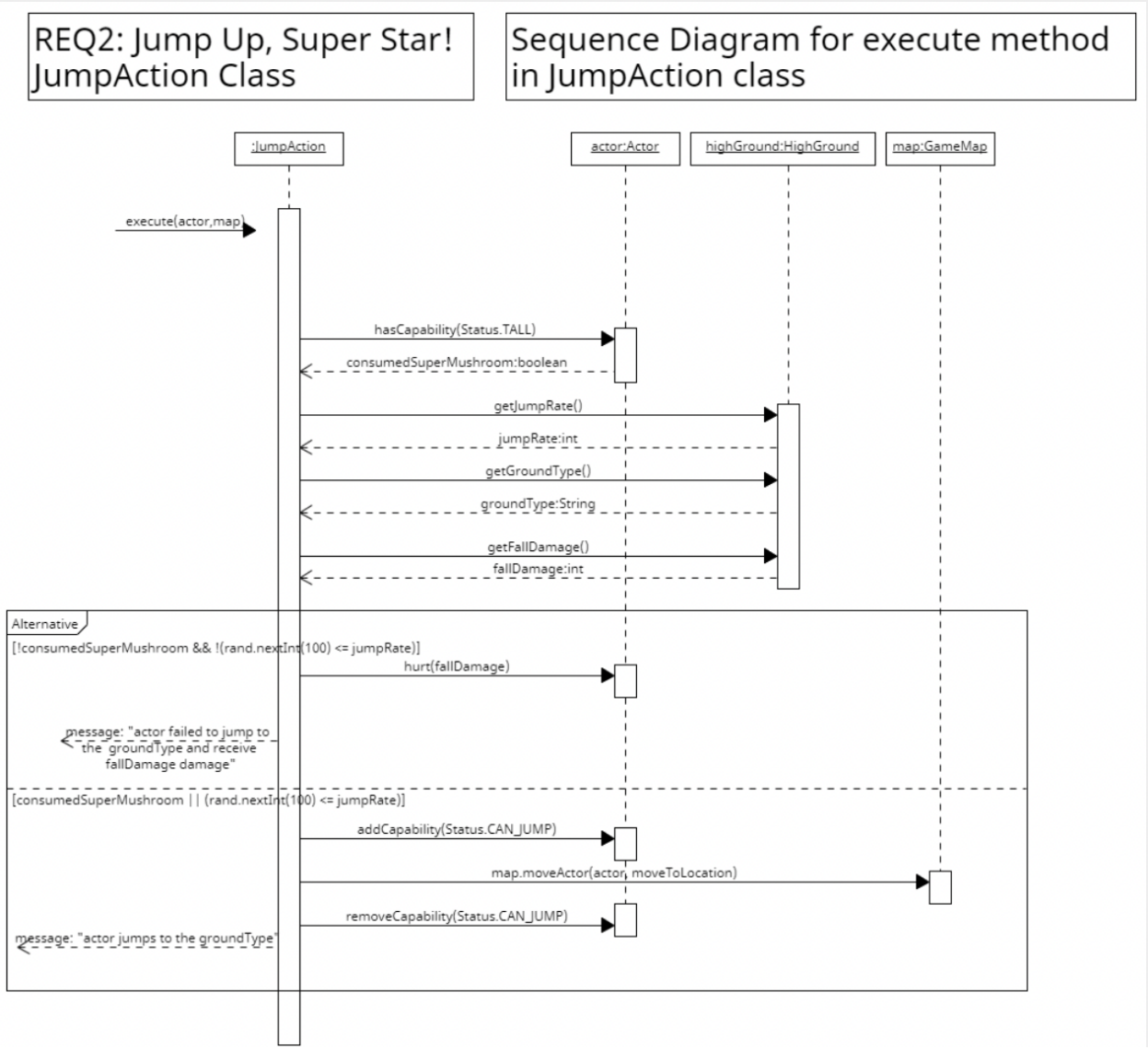
REQ2: Jump Up, Super Star!

Overview:

To achieve this feature, there will be three new classes (i.e., JumpAction, HighGround, and Destroyable) 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.



class diagram for req2



sequence diagram for req2

1) HighGround class

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

In assignment 1, I let those high-level grounds implement the Jumpable interface (the details of Jumpable interface can be viewed in the pdf of Assignment 1). However, I realized this is not a good design practice as using an interface does not reduce the repeated code of high-level grounds efficiently. Therefore, I had decided to create this abstract class called HighGround which extends Ground. In addition, I will let Tree and Wall class to extend this class as they are the current high ground in the game. By doing so, they can share the same code such as canActorEnter, getJumpRate and so on.

Why I chose to do it that way:

Currently based on the requirement of this feature, the Wall class and Tree class will be the extension of this class. This indicates that Wall and Tree on the map are jumpable for the Player as they are high ground. The reason for us to create an abstract class is that I realized Wall and Tree have a lot of repeated code and they both extend from the Ground. Hence, I decided to let HighGround extend Ground and let Wall and Tree class extend HighGround. 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 extend HighGround. By doing so, we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods in the class are either a query or command but not both. Hence, we are following the Command-Query Separation Principle. Moreover, our implementation also fulfils the Reduce Dependencies Principle. It is because instead of having a JumpAction class to have an attribute of each high ground (i.e., Wall or Tree) (we will discuss this more in the JumpAction section), we can have an attribute of type HighGround. By doing so, the concrete class JumpAction will not directly depend on the Wall and Tree class. Additionally, we also fulfil the Liskov Substitution Principle as all the methods in the HighGround class still preserve the meaning from its parent class, Ground. Additionally, within the allowableAction method in the HighGround class, we pass in the instance of HighGround to the JumpAction class, which is known as constructor injection. The benefit of using dependency injection is to ensure the reusability of code and ease of refactoring.

2) Wall class

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

Instead of implementing the Jumpable interface (which is already removed in Assignment 2) as it does not reduce the repeated code efficiently, I created a new abstract class called HighGround and let this class extend it. The reason is because I realized that all high ground classes (i.e., currently only Wall and Tree) share some part of the code. To avoid any repeated code, I had decided to let Wall class extend HighGround. Therefore, in Wall class, it only overrides one method which is blocksThrownObjects, as Wall can block thrown objects.

Why I chose to do it that way:

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., HighGround). 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 HighGround abstract class. Besides that, since I am storing all the constants as private class attributes, consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Furthermore, we also fulfil the Liskov Substitution Principle as all the methods in the Wall class still preserve the meaning from its parent class, HighGround.

3) Tree class

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

Similar to the Wall class, instead of implementing the Jumpable interface (which is already removed in Assignment 2) as it does not reduce the repeated code efficiently, I created a new abstract class called HighGround and let this class extend it. The reason is because I realized that all high ground classes (i.e., currently only Wall and Tree) share some part of the code. To avoid any repeated code, I had decided to let Tree class extend HighGround. More details regarding Tree class and its subclasses can be viewed in the design rationale

of Tree class in REQ1.

Why I chose to do it that way:

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 Wall without modifying the implementation of its parent class (i.e., HighGround). 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 HighGround abstract class. Besides that, since I am storing all the constants as private class attributes, consequently, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Furthermore, we also fulfil the Liskov Substitution Principle as all the methods in the Tree class still preserve the meaning from its parent class, HighGround. Besides that, inside the Tree class, I had added a helper method to convert the tree instance to dirt at that location. For this method, I had checked the pre-condition of the input parameter (i.e., currentLocation) such that it cannot be null, else it will throw a IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allow the developer to know where the problem is.

4) Status class

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

The only changes I made between Assignment 1 and Assignment 2 is to rename the constant JUMP_ONE_HEIGHT to CAN_JUMP and delete SUPER_MUSHROOM. The reason to rename JUMP_ONE_HEIGHT is because I think CAN_JUMP is a more appropriate name. In addition, the reason to delete SUPER_MUSHROOM is because I realized the constant TALL does the exact same function with it, hence this new constant seems to be redundant. Other than that, there are no more changes made.

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

5) JumpAction class

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

The only changes made between Assignment 1 and Assignment 2 is that instead of storing a type of Jumpable as the class attribute, we are now storing a type of HighGround private attribute in JumpAction. Besides that, since the name of the constant (i.e., stated in Status class), we must change the name accordingly in the execution method. Other than these two minor changes, the rest of the part is the same as the explanation stated in Assignment 1.

6) Destroyable class

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

Therefore, please refer to the design rationale of this class in assignment 1. According to the assignment specification, when the player is invincible, it will destroy the high ground (i.e., convert it to dirt) and drop 5 coins. However, in assignment 1, we did not create this class and we planned to do these operations inside Wall and Tree, which is a bad design practice. Thus, in assignment 2, I had decided to create a destroyable interface class which has two methods. (i.e., convert the current location to dirt and create a coin instance with a value of 5 at the location. After that, I will let HighGround implement the Destroyable interface.

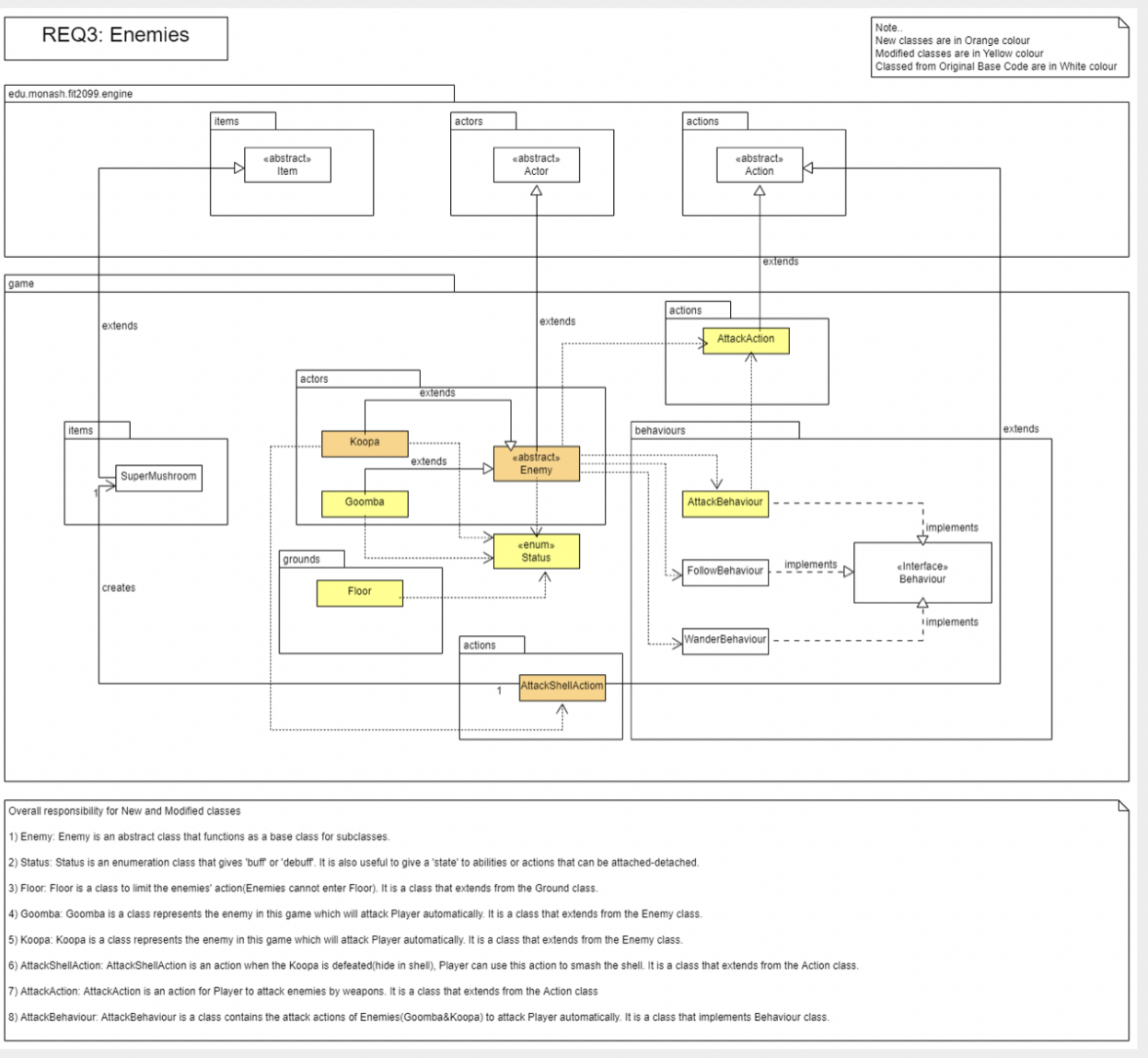
Why I chose to do it that way:

It is possible that more types of ground (i.e., other than high ground) can be destroyed in the future, therefore in order to allow extension, we make this as an interface class, so that other classes can implement it in the future. This is also the reason why I did not put the two operations (i.e., convert to dirt and drop coins) directly in HighGround class. By doing so, we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are just a command. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Besides that, we are adhering to the Interface Segregation Principle as it only implements the methods that it cares about. Furthermore, for the two default methods in this interface, I had checked their input parameter to ensure that the location given is not null, and the coin value is 5. The reason to check if the coin value is 5 is that currently based on the assignment specification, when a high ground is destroyed, the value of the coin must be 5. However, if the value of the coin drops when the high ground is destroyed can be different (i.e., not only 5), then we can remove this exception. By doing so, we are adhering to the Fail Fast Principle as we immediately stop the game when the code detects something wrong, and this helps developers to know where the problem is.

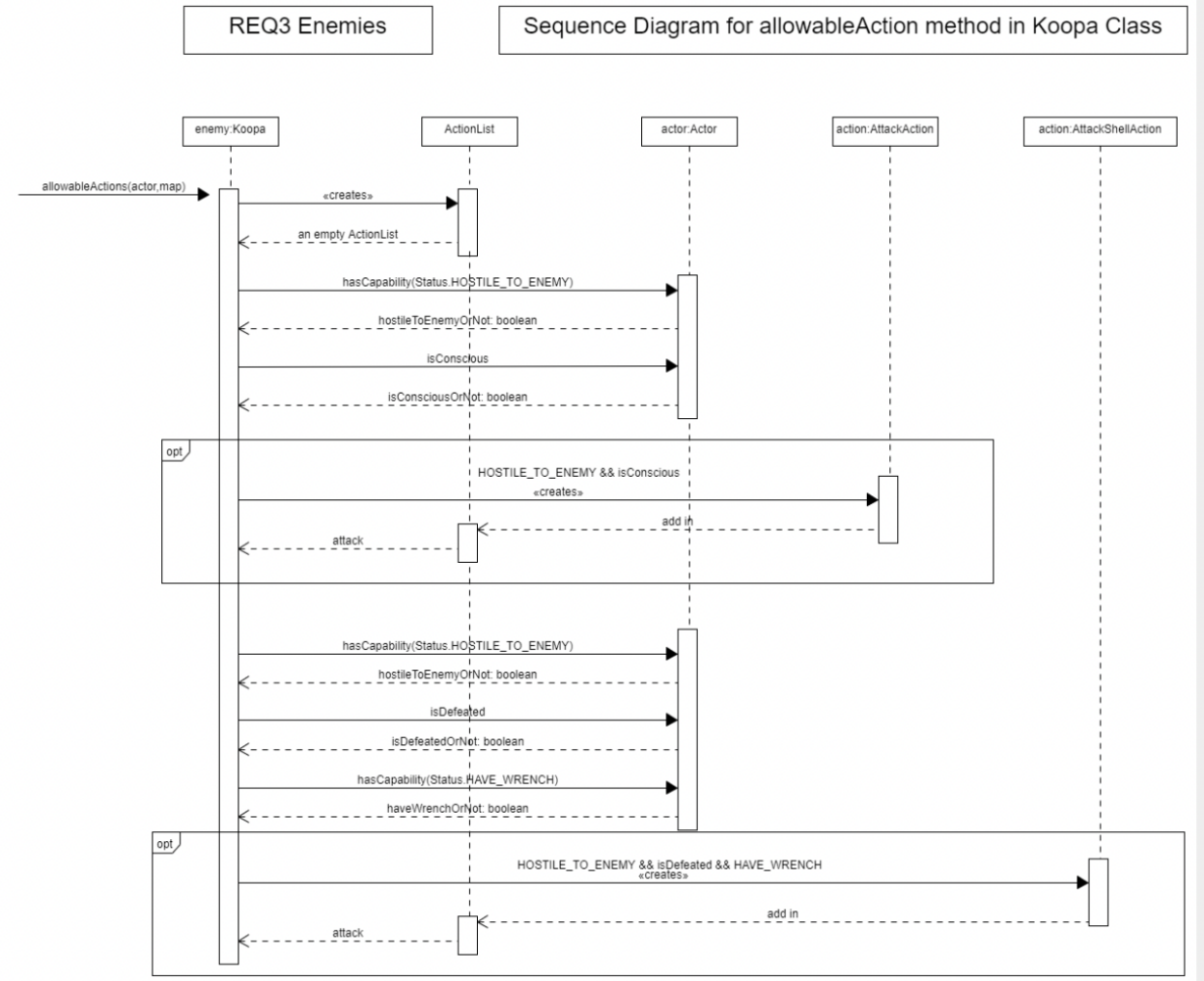
REQ3: Enemies

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.



class diagram for req3



sequence diagram for req3

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 {
```

```
/**
     * A constant integer that will be used as a key in the hash map to indicate which
     * behaviour is the first priority
     */
    protected static final int FIRST_PRIORITY = 1; // key of hashmap

    /**
     * A constant integer that will be used as a key in the hash map to indicate which
     * behaviour is the second priority
     */
    protected static final int SECOND_PRIORITY = 2; // key of hashmap

    /**
     * A constant integer that will be used as a key in the hash map to indicate which
     * behaviour is the third priority
     */
    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

```
/**
 * 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.
 *
 * @author Huang GuoYueYang
 */
public class Goomba extends Enemy {
```

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).

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

```
/**
 * 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 {
```

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).

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.

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.

```
@Override
    public boolean canActorEnter(Actor actor) {
        // Player can enter floor
        if (actor.hasCapability(Status.HOSTILE_TO_ENEMY)){
            canEnter = true;
        }
        // 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'.

Why I choose to do it that way:

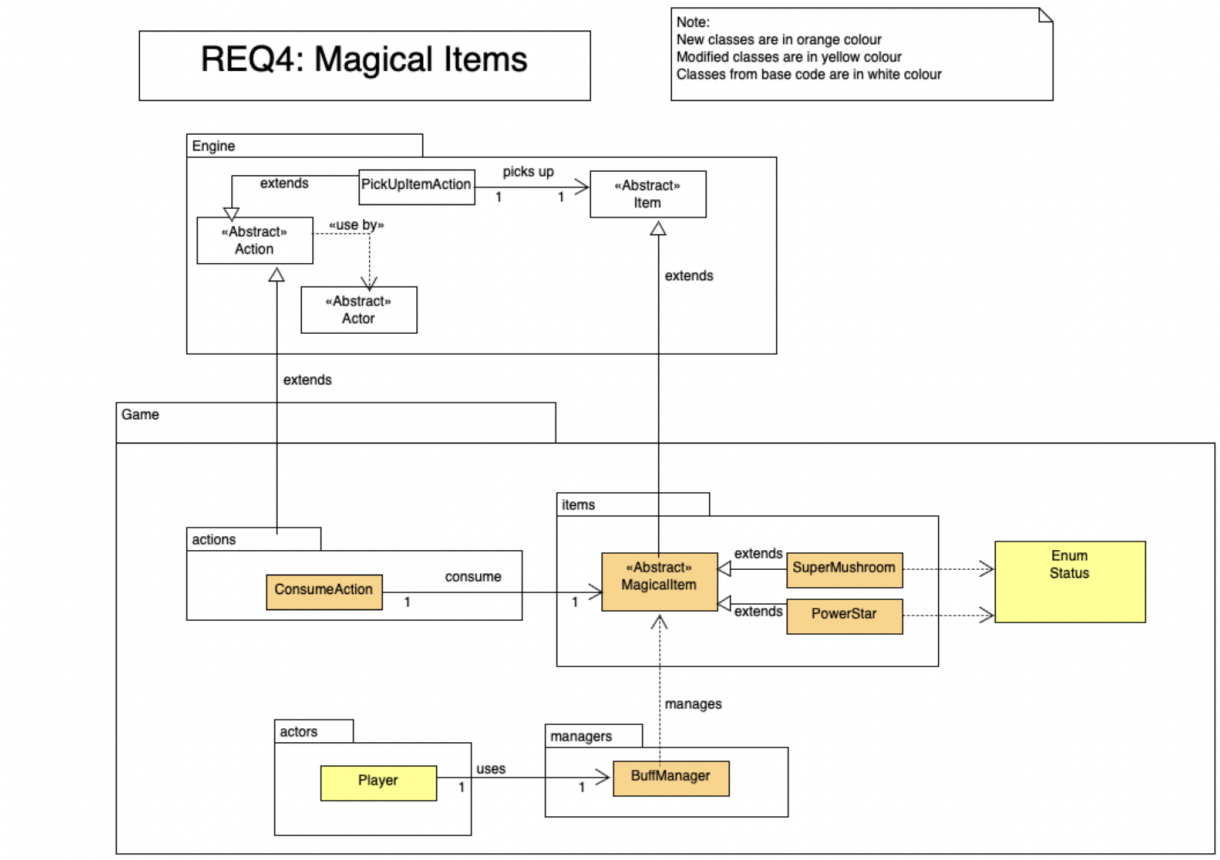
Instead of creating two attack method/behavior separately for Koopa and Goomba, I will just use getAction method in this class to return the AttackAction to attack player automatically. Also, it obeys the SPR code since it has its own single responsibility. By doing so, there will be much lesser repeating 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:

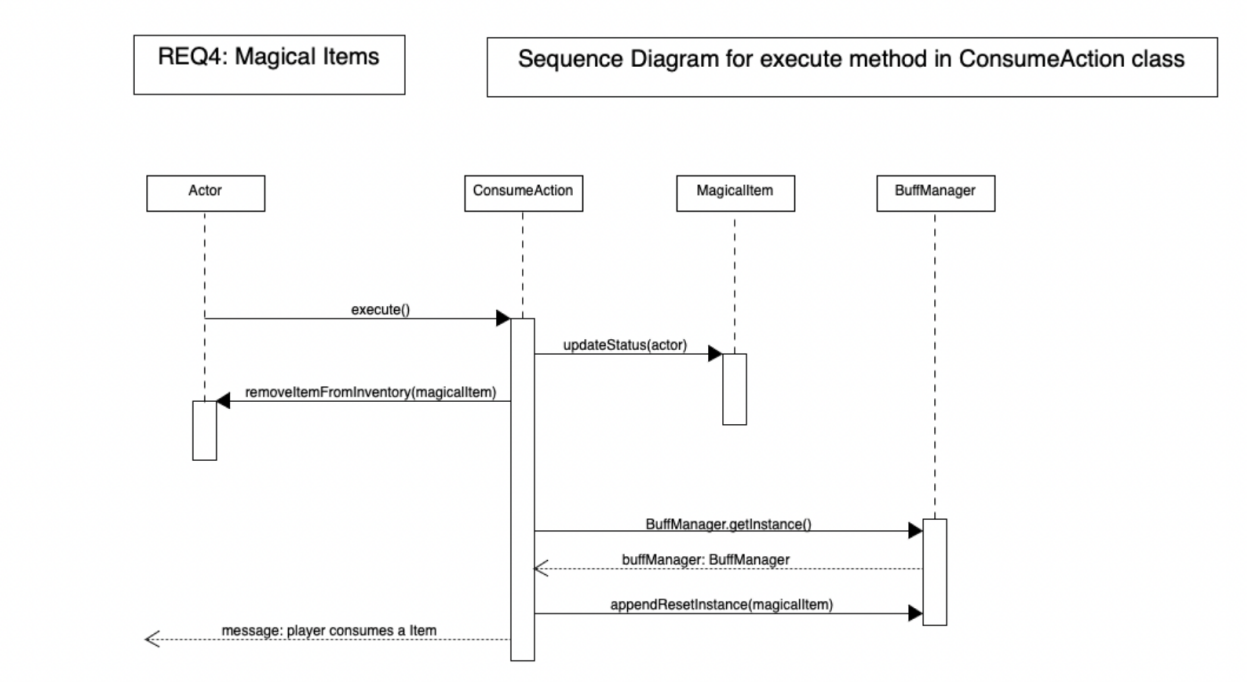
There is no changes in this class of REQ3 between Assignment 1 and Assignment 2.

REQ4: Magical Items



Overall responsibilities for new and modified classes:
1) ConsumeAction: allow player to consume magical items
2) SuperMushroom: represents Super Mushroom in the game map
3) PowerStar: represents Power Star in the game map
4) MagicalItem: abstract class to represent all magical items involve in the game
5) BuffManager: to remove expired magical items
6) Status: enum class which indicates status of player after consuming magical items
7) Player: represents player playing the game

class diagram for req4



sequence diagram for req4

1) ConsumeAction class

What changes in the design between assignment 1 and assignment 2:

- There is association between this class with the MagicalItem class because it has an attribute of MagicalItem in its constructor
- execute method in assignment 2 is modified to update status of actor depending on magical item consumed and add the item into actor's inventory by obtaining the magicalItem returned from menuDescription method instead of using if-else statement (in assignment 1) to check which magicalItem is consumed

```
@Override
    public String execute(Actor actor, GameMap map) {
        magicalItem.updateStatus(actor);
        actor.removeItemFromInventory(magicalItem);
        BuffManager.getInstance().appendResetInstance(magicalItem);
        return actor + " consumed the " + magicalItem;
    }
```

Why i choose to do it that way:

I decided to make that change because,

the use of if-else statements as planned in assignment 1 will be a breach of Open Closed Principle because if a new item is added a new conditional statement needs to be added to the method, making the method not close for modification. As the game develops further, It might cause confusion during methods implementation as more and more if-else statements will be needed in the execute method. Liskov Substitution Principle(LCP) will also be breached in this case as the method will not work if a new instance of magical item subclass is pass in unless a new conditional statement is added, but that will not be a good programming practice.

By applying changes in assignment 2,

the OCP can be implemented as the method will remain the same regardless of what kind of and how many items are involved in the game. Besides, LCP can be implemented too as an instance of MagicalItem subclass can always be pass in and this ease the process of further development when more magical items are involve.

2) MagicalItem class

What changes in the design between assignment 1 and assignment 2:

This class does not exits in assignment 1. In assignment 2, some important methods in this class are:

- updateStatus method to update status of the actor upon consumption
- tick method to keep track of the number of turns player makes while having the magical item
- setIsExpired to set isExpired to true when player loses the magical item effect

Why i choose to do it that way:

In assignment 2, this is an abstract class created to be the parent class of PowerStar and SuperMushroom class. Since Power Star and Super Mushrooms are magical items, they both share some same functions and hence an abstract MagicalItem class can be created to reduce duplicated codes as they can access methods from MagicalItem class by calling super. In assignment 1, without this abstract class, there are some repeated codes in the PowerStar and SuperMushroom class, leading to the breach of the Don't Repeat Yourself (DRY) principle.

The Don't Repeat Yourself Principle can be applied using approach in assignment 2 as common used methods don't have to be copy pasted in every magical item subclasses and this improve code readability and reduces possibilities of making errors.

3) PowerStar class

What changes in the design between assignment 1 and assignment 2:

- PowerStar class is extended from MagicalItem class instead of Item class
- a tick attribute is added into the PowerStar constructor
- There are two tick methods added into this class. One to keep track of the number of turns player makes when player is under the power star effect. INVINCIBLE capability will be remove from player after 10 turns and the power star will be remove from player's inventory, while the other one is to keep track of the number of turns it is left to be available in player's inventory or in the game map. The power star will be remove from the map once it reaches 10 turns.
- currentStatus method is added into this class to add INVINCIBLE capability to player while player haven't reach 10 turns while having the power star effect.

Why i choose to do it that way:

As explained in the MagicalItem class section, this class is extended from MagicalItem class instead of Item class in order to reduce duplicated code and hence obey the DRY principle. By doing so, default methods can be inherited from the its parent class (MagicalItem).

By adding tick method to the Power Star class, the SRP can be implemented since the power star effect is only available to actor 10 turns after consuming it. By having a tick method, each power star in the inventory, ground, and

consumed power star will have their own “timer”. This is important to deal with overlapping power star effect. For example, if actor consumes a power star, play 5 turns, and then consume another power star, the actor will suppose to have 15 turns with the power star effect.

Furthermore, exception is added in order to implement the Fail Fast principle, so that when the power star is added to an invalid location, the program will throw an error and stop running right after instead of continue to run until power star is to be spawn. By doing this, this eases debugging process as it will be easier to identify the cause of error.

4) SuperMushroom class

What changes in the design between assignment 1 and assignment 2:

- SuperMushroom class is extended from MagicalItem class instead of Item class
- exception is added into the currentStatus method

```
@Override
    public void currentStatus(Location location) {

        if (location == null){
            throw new IllegalArgumentException("The input parameter (i.e., location) cannot be null");
        }
```

Why i choose to do it that way:

As explained in the MagicalItem class section, this class is extended from MagicalItem class instead of Item class in order to reduce duplicated code and hence obey the DRY principle. By doing so, default methods can be inherited from the its parent class (MagicalItem).

Similar to as in the PowerStar class, exception is added in order to implement the Fail Fast principle.

5) BuffManager class

What changes in the design between assignment 1 and assignment 2:

This class does not exist in assignment 1

- run method to transverse through the list of magical items. This method is execute in every play turn of the players.
- appendResetInstance method to add resettable instance to the list
- exception is added into the appendResetInstance method

```
public void appendResetInstance(MagicalItem magicalItem){

        if (magicalItem == null){
            throw new IllegalArgumentException("The input parameter (i.e., magicalItem) cannot be null");
        }

        magicalItemList.add(magicalItem);
    }
}
```

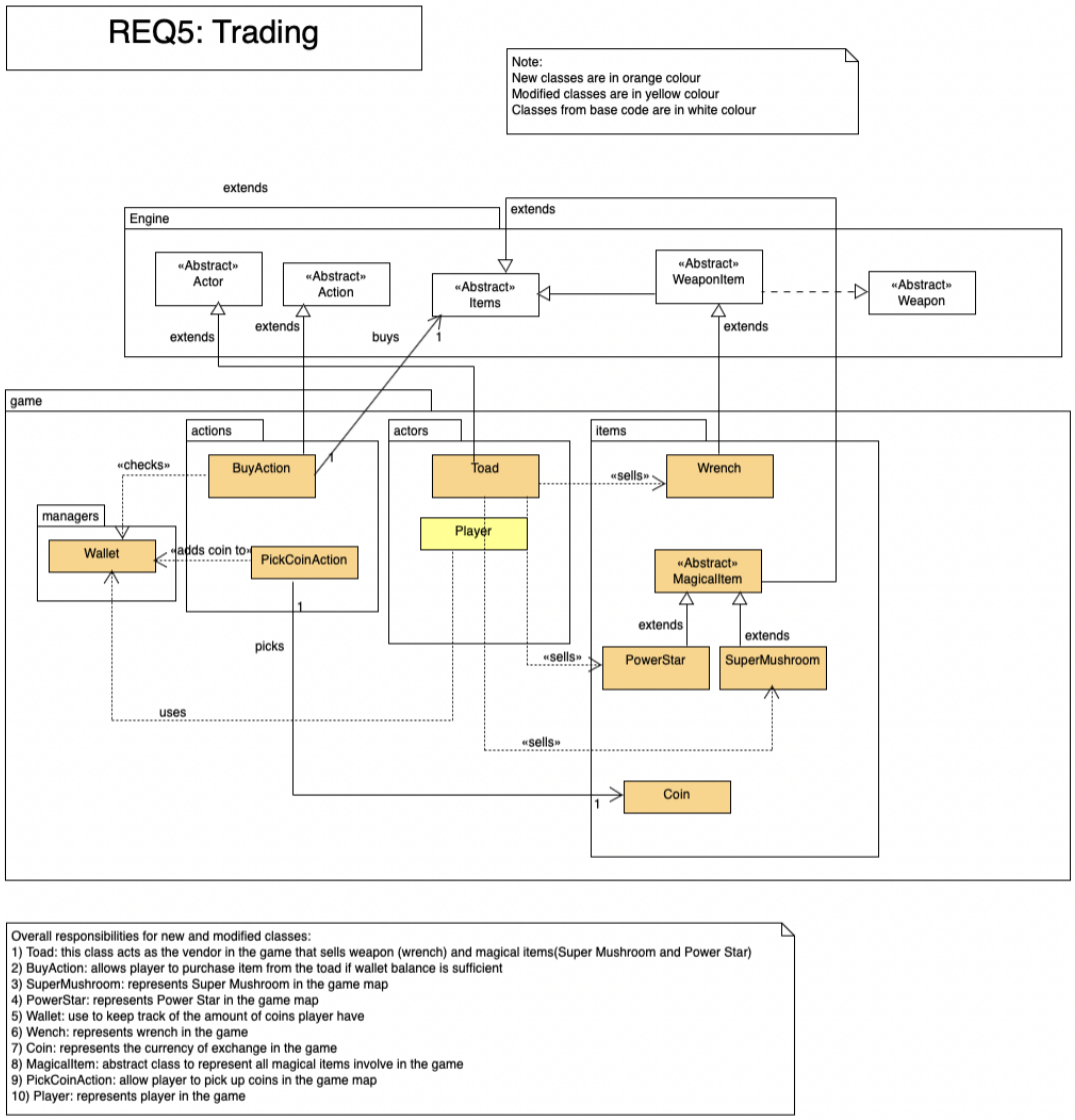
Why i choose to do it that way:

This class is added in assignment 2 to loop through the list of magical items and remove expired magical items from the list. By having a static factory getInstance method, the constructor of the BuffManager class can be call without creating a new instance of it. For example `BuffManager. getInstance() .run ( map.locationOf ( this )) ;`

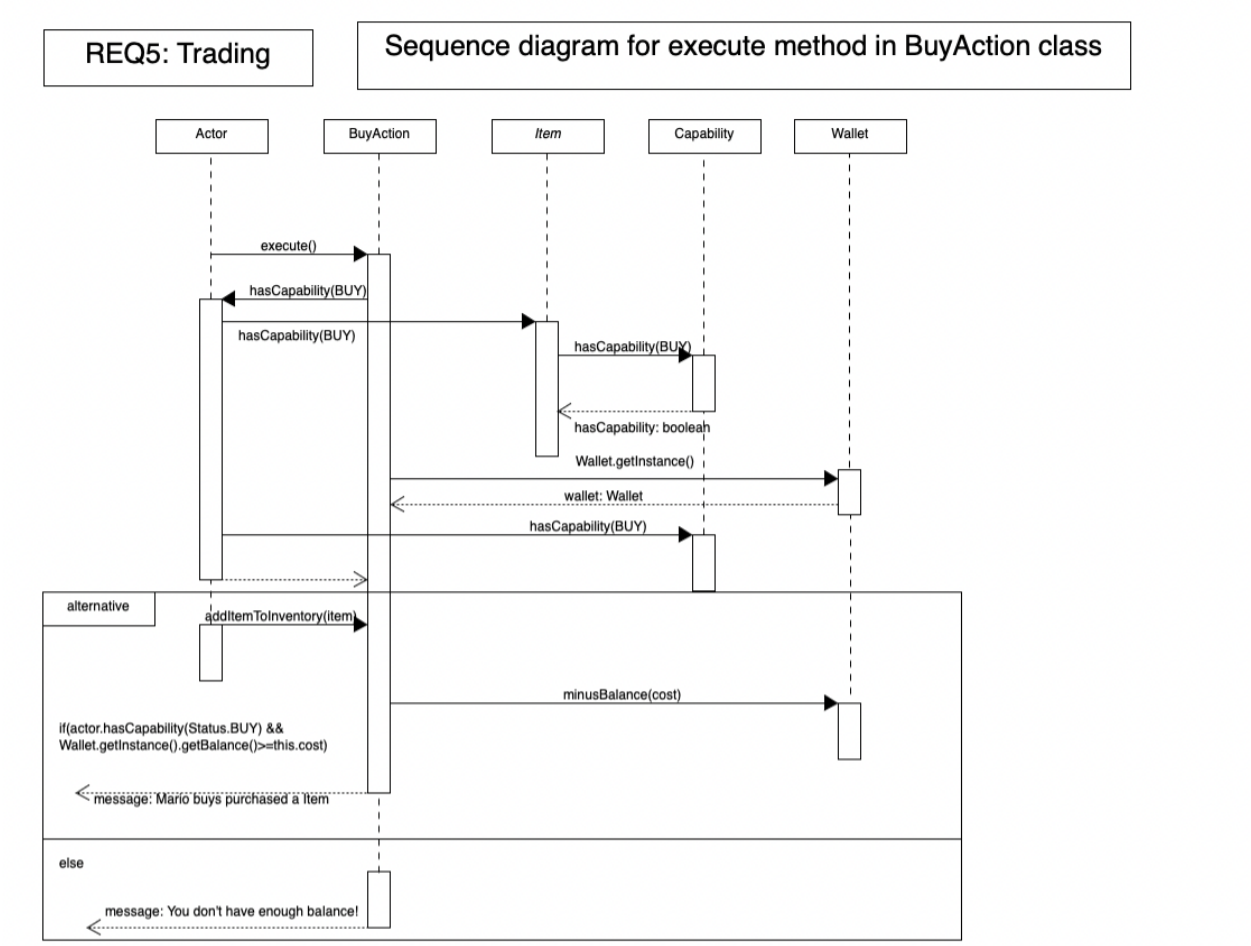
This therefore implements encapsulation as the run() method can be call without getting internal information of the class, hence hiding implementation.

Furthermore, exception is added into the appendResetInstance method in order to implement the Fail Fast principle, so that when a null magical item is added into the list, the program will throw an error and stop running right after instead of continue to run until player consumes the magical item. By doing this, debugging process will be easier as it will be easier to identify the cause of error.

REQ5: Trading



class diagram for req5



sequence diagram for req5

1) PickCoinAction class

What changed in the design between assignment 1 and assignment 2:

- execute method is a method that is override from its parent class (Item class) to add the value of coins collected by player into the player's wallet balance if player picks up the coin, and then display the amount of coin picked up by player in the menu description.

```
// assignment 2
@Override
public String execute(Actor actor, GameMap map) {
        map.locationOf(actor).removeItem(coin);
        Wallet.getInstance().addBalance(coin.getValue());
        System.out.println("Wallet balance: " + Wallet.getInstance().getBalance());
        return menuDescription(actor);
    }
```

Why i choose to do it that way:

Changes were applied in assignment 2 because I realised there is no need to obtain the hashCode of the actor and coin's location. I also decided to write those lines of code in the override execute method because the method will be invoked after actor enters the hotkey, hence it is not needed to create a separated method to add value of coins into wallet balance.

Using the new approach, the Single Responsibility principle can be implemented as this class will only be used when used to add value of the picked up coin into the actor's wallet balance.

2) Toad class

What changes in the design between assignment 1 and assignment 2:

- tradeItem method is removed
- playTurn method is a method that is override from the parent class (Actor class). This is to add new instance of items into the saleItem list in every play turn of the toad.
- allowableAction method is another method that is override from the parent class (Actor class). This method is to add buyAction to the toad's customer (player) so that they can purchase item from the toad based on the hotkey keyed in.

Why i choose to do it that way:

tradeItem method is removed because that process was moved to be carried out by the BuyAction class. Having the tradeItem method in assignment 1 will cause the Toad class to breach SRP as the Toad should only act as an "item provider" in this requirement.

3) BuyAction class

What changes in the design between assignment 1 and assignment 2:

- execute method checks if actor's (which has BUY capability) wallet has enough money, if yes, add the item into actor's inventory and subtract the item price from wallet, and then return a string indicating item bought by player. Else, inform player that balance is insufficient.

```
/**
     * Add the item to the actor's inventory and subtract the cost from actor's wallet balance
     *  display "You don't have enough coins!" is actor's balance is insufficient
     * @see Action#execute(Actor, GameMap)
     * @param actor The actor performing the action.
     * @param map The map the actor is on.
     * @return a suitable description to display in the UI
     */
    @Override
    public String execute(Actor actor, GameMap map) {
        String output="";
        if(actor.hasCapability(Status.BUY) &&
                 Wallet.getInstance().getBalance()>=this.cost){
            actor.addItemToInventory(item);
            Wallet.getInstance().minusBalance(cost);
            output+= actor + " purchased a " + this.item;
        }
        else{
            output+="You don't have enough coins!";
        }
```

```
        return output;
    }
```

Why I choose to do it that way:

In assignment 2, the execute method which is override from the action class is use to process the actor's buying action instead of using a separate method as designed in assignment 1 because the execute method will be invoke once the actor enters the hotkey corresponding to the BuyAction class. In the execute method, the actor needs to be check if it has the BUY capability to ensure that it is a player.

Furthermore, by having Item in the BuyAction constructor, the LCP can be implemented as as an instance of Item subclass can always be pass into the execute method. This approach is important because the Toad not only sells magical item but weapon such as wrench too (or even other types of item in the future). Hence the execute method can take in any type of item as long as the class is a subclass of Item class.

4) Wallet class

What changes in the design between assignment 1 and assignment 2:

- Wallet class is made to have static factory methods
- receiveCoin method is removed
- getBalance method to return the current amount of coins in player's wallet
- minusBalance method to subtract item cost from player's wallet balance
- addBalance method to add value of coin into player's wallet balance
- exception is added into the addBalance and minusBalance methods

```
if (balance <= 0){
            throw new IllegalArgumentException("The balance cannot be 0 or negative");
        }
```

Why i choose to do it that way:

Static factory methods are use in Wallet class so that the constructor of the Wallet class can be call directly instead of creating a new instance of Wallet class. By doing this, encapsulation can be use as the behaviour of the Wallet class can be specify via parameters without knowing the Wallet class hierarchy. Also, static factory methods eliminates the need to create a new instance of Wallet for every player instantiated. Code readability can also be improved in this case. For example, the wallet balance can be obtained by `Wallet.getInstance().getBalance()`.

Besides that, exception is added into the addBalance and minusBalance method to detect the addition and subtraction of wallet balance that involve negative values. By doing this, the Fail Fast principle can be implemented as an error will be thrown and program will stop running right after a negative value were to be subtracted or added into the wallet balance to ensure the correctness of the value of player's wallet balance. This makes debugging process easier as it will be easier to track cause of error.

5) Wrench class

What changes in the design between assignment 1 and assignment 2:

- HAVE_WRENCH capability is added to the Wrench in the constructor of the Wrench class instead of creating an additional updateStatus method as in assignment 1
- Wrench class extends WeaponItem class instead of Item class

Why i choose to do it that way:

Based on the Actor class given in the engine, the hasCapability method will iterate through the actor's inventory and returns true if the item has the required capability, and if the item has that capability, the capability will be added to the actor too. Which means, actor will have capabilities that items in the inventory have. Hence there is no need to create an additional method as it may be redundant. By using this approach, the SRP can be implemented as the Wrench class will only be responsible to store information of itself and add capability to actors having it. It also reduces lines of code and therefore improve readability.

Furthermore, Wrench was decided to be extended from WeaponItem class because wrench is a weapon and instance variables from the WeaponItem constructor is needed for Wrench to contain additional instance such as damage, hit rate and verb. By extending from WeaponItem, Wrench class will be able to access those instance variable just by calling super, hence reducing lines of repeated code. Extending Wrench class from Item class as did in assignment 1 was an incorrect approach as additional instance variables need to be declare and it may cause confusion in the BuyAction class.

6) Coin class

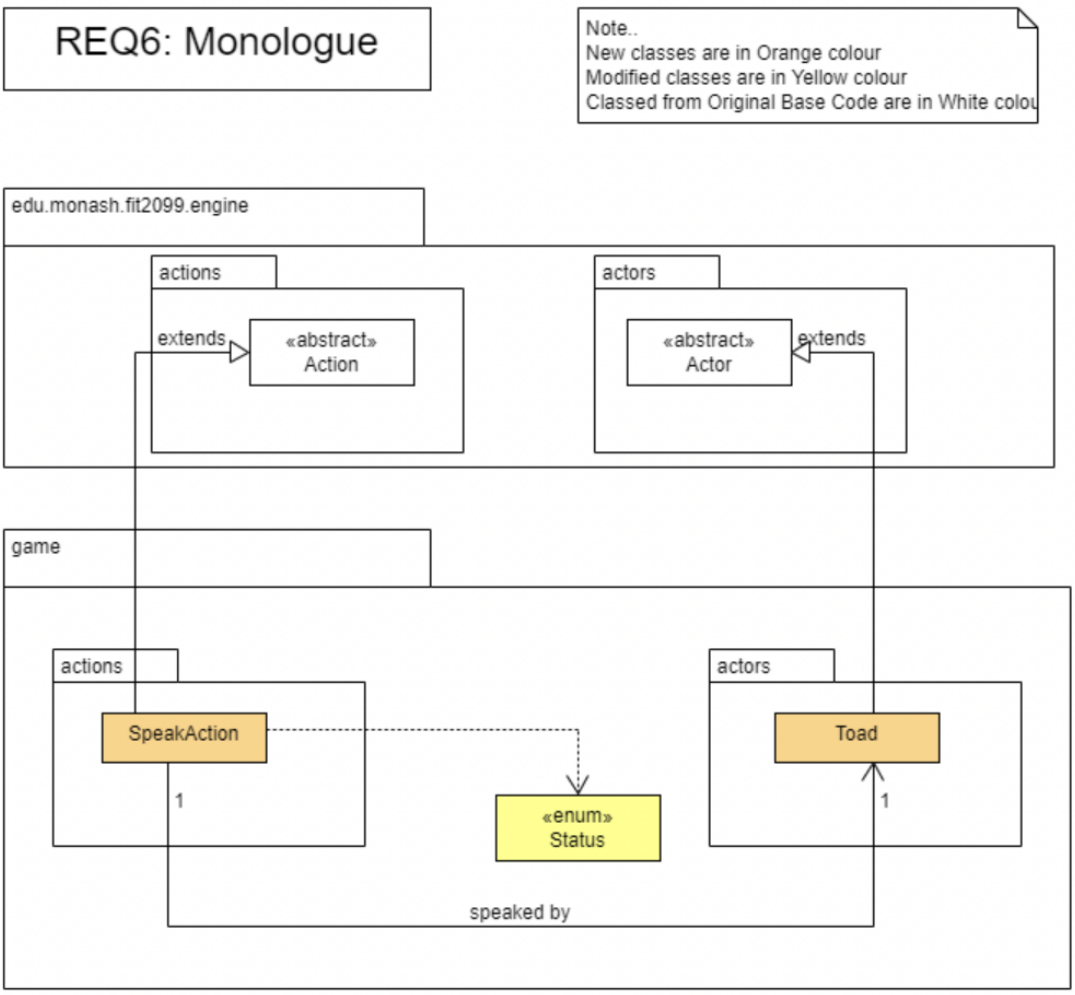
What changes in the design between assignment 1 and assignment 2:

- no changes

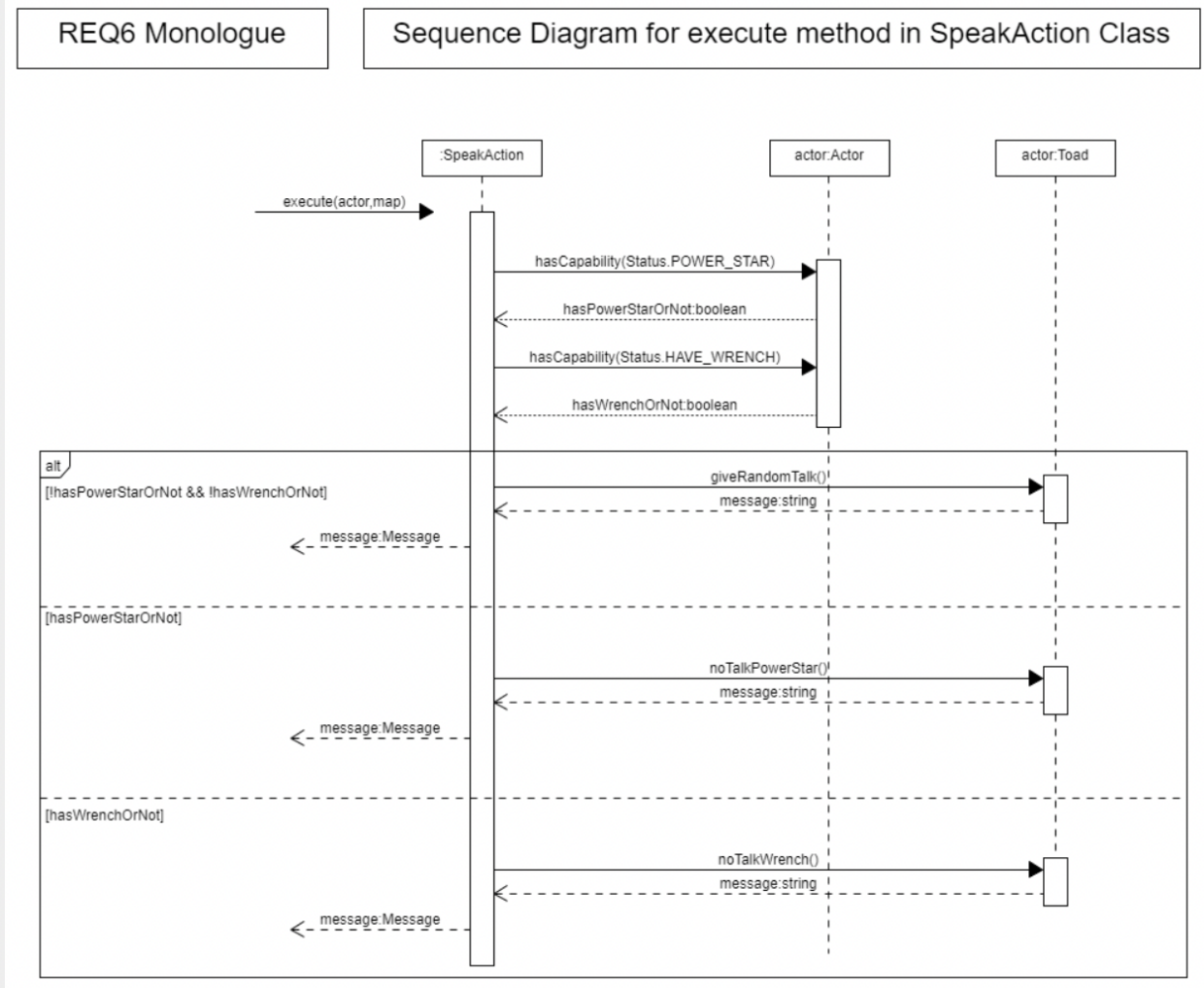
REQ6: Monologue

Overview:

To achieve this feature, there will be two new classes (i.e., SpeakAction and Toad) created in the extended system, and two existing classes (i.e., Application and Status) will be modified. The design rationale for each new or modified class is shown on the following pages.



class diagram for req6



sequence diagram for req6

1) SpeakAction

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

There is no changes in this class of REQ6 between Assignment 1 and Assignment 2.

2) Toad

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

In Assignment 2, I add one method getReplyString() to modify which sentence to be talk depends on different capabilities the player had. In Assignment one, there are three methods which have the similar code structure, which are: giveRandomTalk(), noTalkPowerStar() and noTalkWrench(). The only difference between them is the index of the sentence to be remove. It does not obey the design principle DRY, so I add one getReplyString() method for this class, to make it more logical and readable.

Why I choose to do it that way:

To reduce the repetitive code in those method, I add this method which has an index called removeIndex (represent the index will be remove) as its input parameter and return an currentIndex(after fulfil some conditions). In this method, if the removeIndex equals to null, then the currentIndex will be any random number within the size of the toadTalk Array. Else we put the currentIndex as any random number within the size of the toadTalk Array. If the currentIndex equals to the removeIndex, the currentIndex will jump over that removeIndex, so the corresponding sentence will not be print. This method helps to get the correct sentences, so when I want to print the corresponding sentences, I just call this method and input the index of the sentence which I want to remove from the arraylist. It obeys the design principle and make the code logical.

```
/**
     * Give any sentences from the arraylist
     */
    public String giveRandomTalk() {
        return getReplyString(null);
    }

    /**
     * Give any sentences from the arraylist except the sentence about the power star
     */
    public String noTalkPowerStar() {
        return getReplyString(POWER_STAR_INDEX);
    }

    /**
     * Give any sentences from the arraylist except the sentence about the wrench
     */
    public String noTalkWrench() {
        return getReplyString(WRENCH_INDEX);
    }
```

In addition, I added exceptions for this method to make sure that the removeIndex is valid. If an index which is out of the range, an ArrayIndexOutOfBoundsException will raise to notify the user the index is wrong. By doing this, I follow the Fail Fast principle that make our code more elaborate.

```
if (removeIndex != null && (removeIndex<0 || removeIndex>=toadTalk.size())){
            throw new ArrayIndexOutOfBoundsException("Incorrect Index to avoid a sentence from toadTalk");
        }
```

3) Application

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

There is no changes in this class of REQ6 between Assignment 1 and Assignment 2.

4) Status Enum

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

In Assignment 2, I change the capability name after player consume the power star, instead of using POWER_STAR_BUFF, I will use INVINCIBLE. POWER_STAR_BUFF is not so readable as INVINSIBLE.

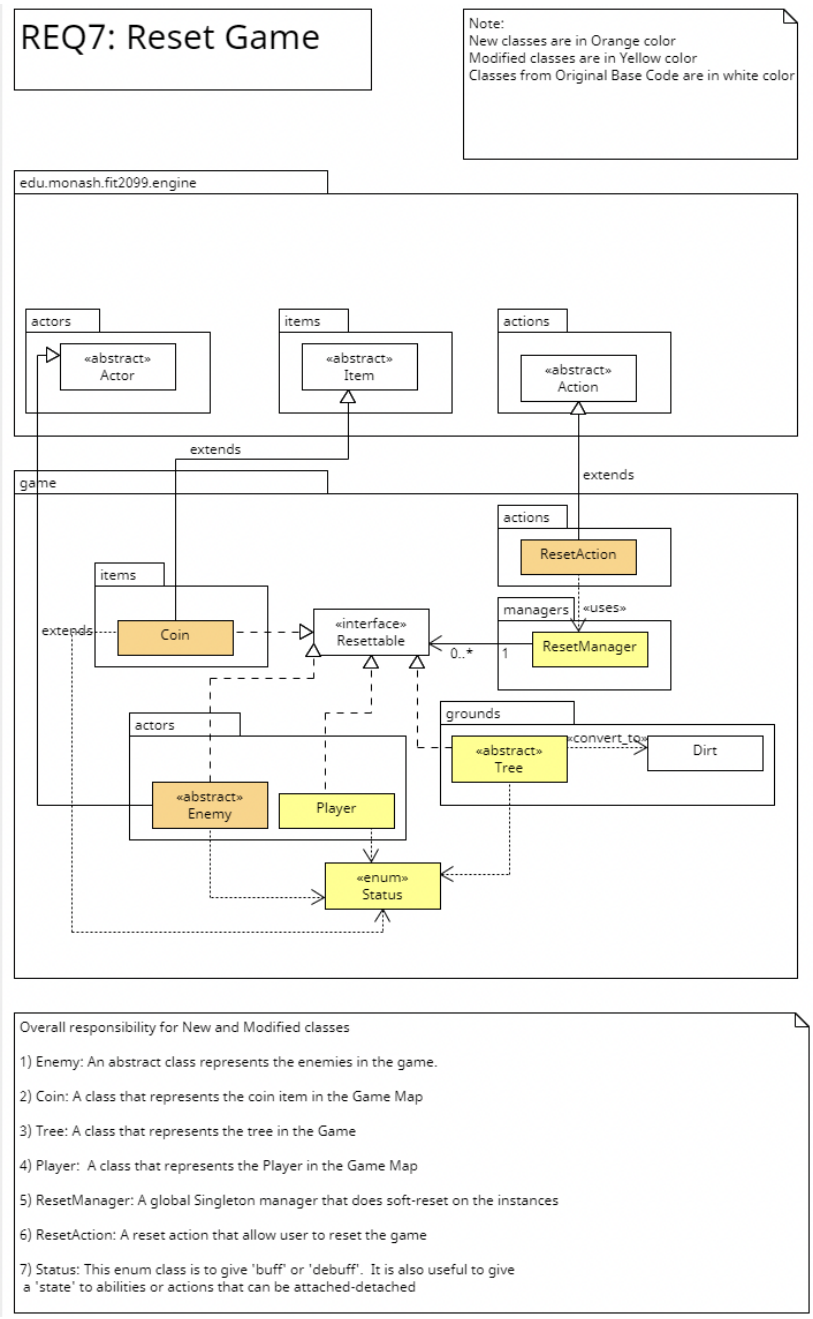
Why I choose to do it that way:

In the assignment specification, it mentions that after player consume the power star, he will become invincible, so I think INVINCIBLE will be more suitable for this capability. It helps me to give capability to player after player consume a power star. By using this enum class, it provides us more flexibility for future development as we can simply add additional capability when needed.

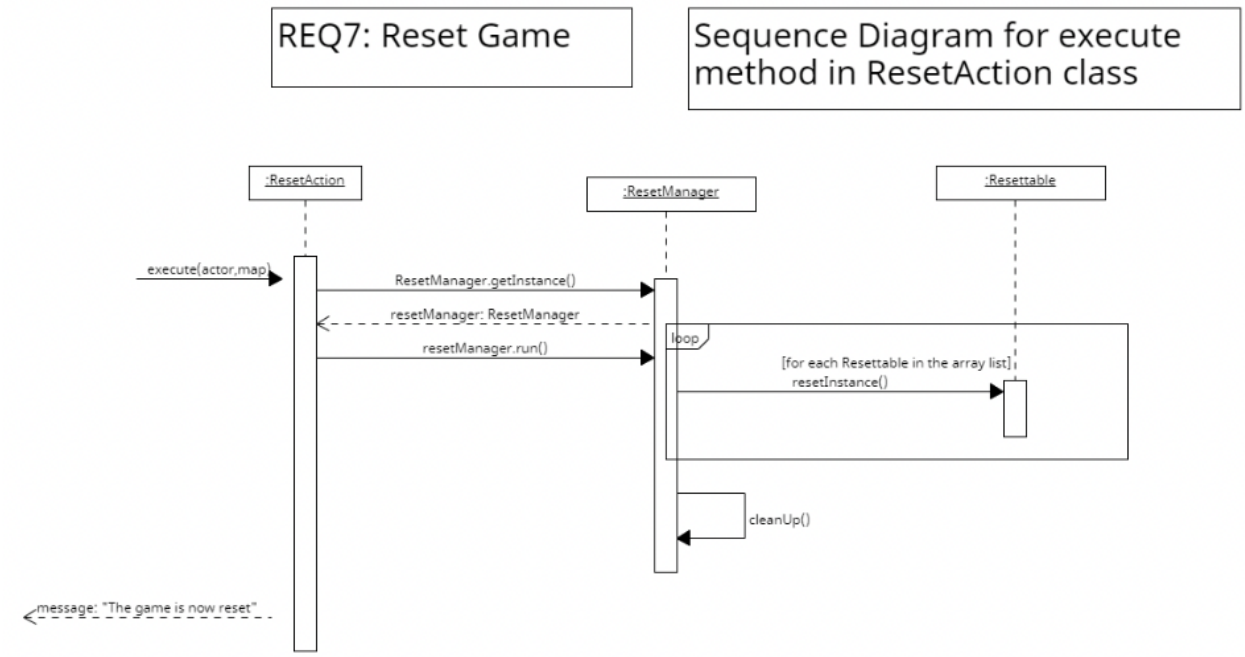
REQ7: Reset Game

Overview:

To achieve this feature, there will be three new classes (i.e., Enemy, ResetAction, and Coin) created in the extended system, and four existing classes (i.e., Status, Tree, ResetManager and Player) will be modified. The design rationale for each new or modified class is shown on the following pages.



class diagram for req7



sequence diagram for req7

1) Status class

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

In assignment 1, I created a lot of constants in different new classes such as ItemCapability and GroundStatus to perform the corresponding operation to Coin and Tree when reset is called. I realized that this is a bad design practice as it overcomplicated my code and I could do all this by just adding one constant to all the instances of these classes. Therefore, in assignment 2, I created a constant called RESET_CALLED. This constant will be added to the instances that their class implements Resettable and resetInstance is called. Therefore, this constant will help me to perform corresponding operations to those instances in either playTurn or tick method.

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

2) Enemy class

Enemy class is an abstract class that represents the enemies in this game. However, since Enemy class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ3) and in this REQ, it only let Enemy class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, when playTurn is called, it will remove that enemy instance from the map. Thus, the design rationale for Enemy class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Enemy class.**

3) Tree class

Tree class is a class that represents the tree in this game. However, since Tree class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ1 and REQ2) and in this REQ, it only let Tree class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, if it has this capability, the this tree instance will have 50% chance to convert to dirt, without considering any features of a Tree instance, therefore the design rationale for Tree class will not be available in this section. **Hence, please go to the REQ1 and REQ2 section in the following pages to see the design rationale for Tree class.**

4) Coin class

Coin class is a class that represents the coin item in this game. However, since Coin class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ5) and in this REQ, it only let Coin class to implements Resettable and inside the resetInstance method, it will add RESET_CALLED to its capability set, and in the next turn, if it has this capability, then this item will be remove from the map, without considering any features of a Coin instance, therefore the design rationale for Coin class will not be available in this section. The usage of IS_COIN is explained in the design rationale of ItemCapability class. **Hence, please go to the REQ5 section in the following pages to see the design rationale for Coin class.**

5) Player class

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

The only changes I made between Assignment 1 and Assignment 2 is that in assignment 1, I not remove HOSTILE_TO_ENEMY when resetInstance is called, but in assignment 2, I will not remove the following three constant, which are HOSTILE_TO_ENEMY, BUY and SPEAK. The reason is because even if the game is reset, the player should still be able to speak to toad, buy from toad and be hostile to the enemy. Rest of the part remains the same as Assignment 1. Thus, please refer to the design rationale for this class in Assignment 1 for more details.

6) ResetManager class

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

No changes are made between Assignment 1 and Assignment 2.

7) ResetAction class

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

In assignment 1, I create a RemoveManager class to store all enemies that will be removed when reset is called. In addition, I use two nested for loops in the execute method of ResetAction class to remove coins and convert trees to dirt by looping through every single location in the map. However, I realized this is a very bad design practice as it violates Single Responsibility Principle. After that, I realized I can do all these operations by letting those classes implement Resettable and call its resetInstance method using ResetAction. This makes our code more readable, maintainable, and allow for extension.

Why I chose to do it that way:

With the design above, we are adhering to the Single Responsibility Principle as the ResetAction class only focuses on what will happen when the user selects option 'r'. Besides that, we also follow the Open Closed Principle. The reason is because based on our current implementation, when there is more Resettable instance or removeable instance created in future, we do not have to modify any code in ResetAction as we could just simply call the run method in ResetManager class and RemoveableManager class. In addition, we also fulfil the Liskov Substitution

Principle as ResetAction preserves the meaning of execute and menuDescription method behaviours from Action class.

```
WBA FIT2099 - Assignment 2

Team Details:

Guoyueyang Huang ghua0010@student.monash.edu
Jia Chen Kuah jkua0008@student.monash.edu
Fluoryynx Lim flim0012@student.monash.edu


Tasks:

REQ1: Let it grow!

    Responsible person(s): Kuah Jia Chen

    Implementation: DONE
    -  Date to be completed: 21-Apr-2022
        - Debug: Kuah Jia Chen DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29-Apr-2022

    Review:
    - (1) Guoyueyang Huang DONE
    - (2) Lim Fluoryynx DONE


REQ2: Jump Up, Super star

    Responsible person(s): Kuah Jia Chen

    Implementation: DONE
    -  Date to be completed: 21-Apr-2022
        -  Debug: Kuah Jia Chen DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29-Apr-2022

    Review:
    - (1) Guoyueyang Huang DONE
    - (2) Lim Fluoryynx DONE


REQ3: Enemies

    Responsible person(s): Guoyueyang Huang

    Implementation: DONE
    -  Date to be completed: 21-Apr-2022
        -  Debug: Guoyueyang Huang DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29-Apr-2022

    Review:
```

```
    - (1) Kuah Jia Chen DONE
    - (2) Lim Fluoryynx DONE


REQ4: Magical Items

    Responsible person(s): Lim Fluoryynx

    Implementation: DONE
    -  Date to complete: 21/4/2022
        -   Debug: Lim Fluoryynx DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29/4/2022

    Review:
    - (1) Guoyueyang Huang DONE
    - (2) Kuah Jia Chen DONE


REQ5: Trading

    Responsible person(s): Lim Fluoryynx

    Implementation: DONE
    -  Date to complete: 21/4/2022
        -  Debug: Lim Fluoryynx DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29/4/2022

    Review:
    - (1) Guoyueyang Huang DONE
    - (2) Kuah Jia Chen DONE


REQ6: Monologue

    Responsible person(s): Guoyueyang Huang

    Implementation: DONE
    -  Date to be completed: 21-Apr-2022
        -  Debug: Guoyueyang Huang DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
        -  Date to be completed: 29-Apr-2022

    Review:
    - (1) Kuah Jia Chen DONE
    - (2) Lim Fluoryynx DONE


REQ7: Reset Game

    Responsible person(s): All

    Implementation:
    - Kuah Jia Chen DONE
    - Date to be completed: 21-Apr-2022
        -  Debug: -  Guoyueyang Huang DONE
                  -  Lim Fluoryynx DONE

    -  Update Class Diagram: DONE
    -  Update Interaction Diagram: DONE
    -  Update Design Rationale: DONE
```

```
        -  Date to be completed: 29-Apr-2022

    Review:
    - (1) Guoyueyang Huang DONE
    - (2) Kuah Jia Chen DONE
    - (3) Lim Fluoryynx DONE

Kuah Jia Chen 32286988: I accept this WBA
Lim Fluoryynx 32023774: I accepted this WBA
HuangGuoYueYang 32022891: I accepted this WBA
```