Assignment 2

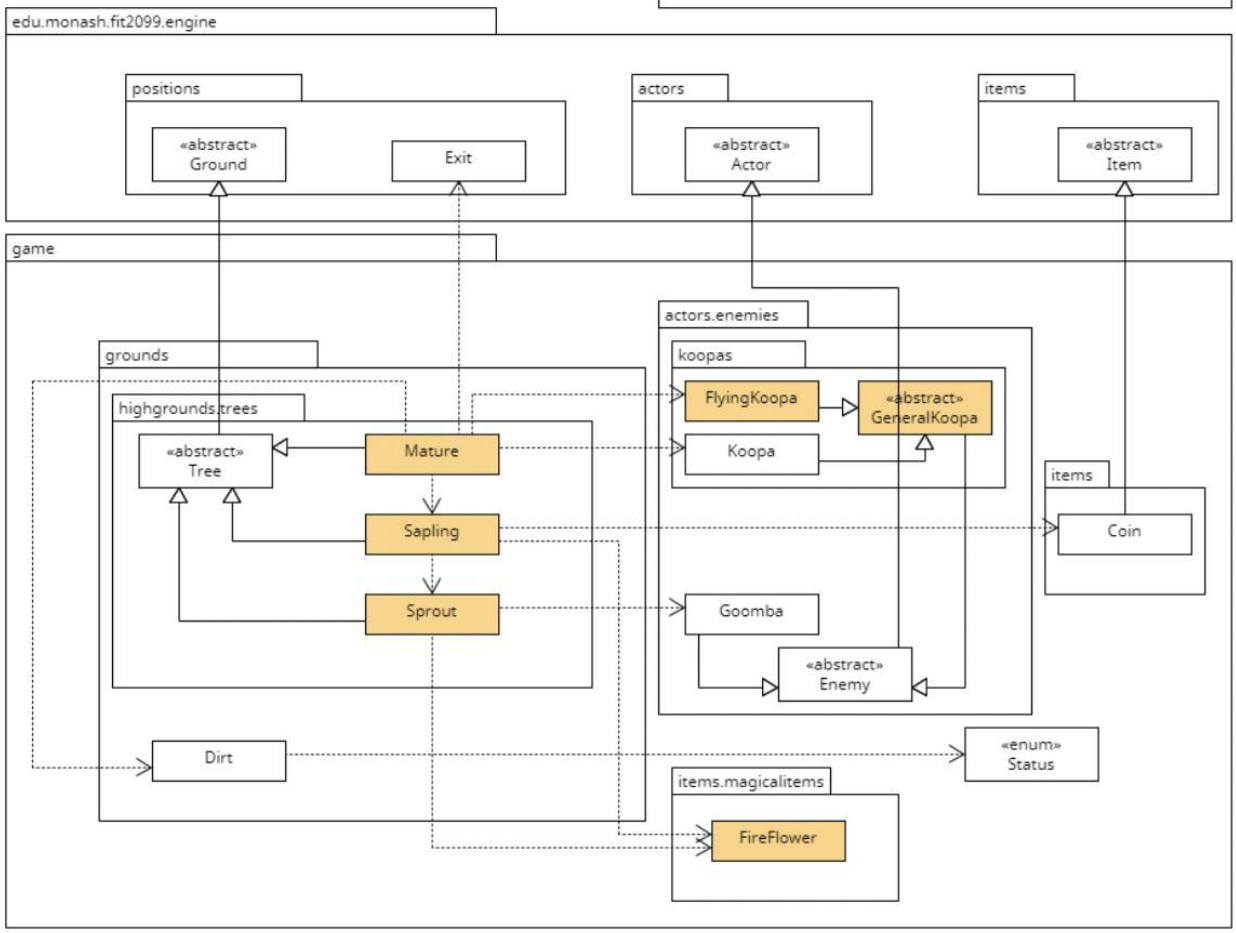
REQ1: Let it grow!

Assignment 2 REQ1: Let it grow!

Note:
New classes are in Orange color

Modified classes are in Yellow color

Classes from Original Base Code OR classes created in Assignment 2 but without any modifications are in white color



Overall responsibility for New and Modified classes

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

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

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

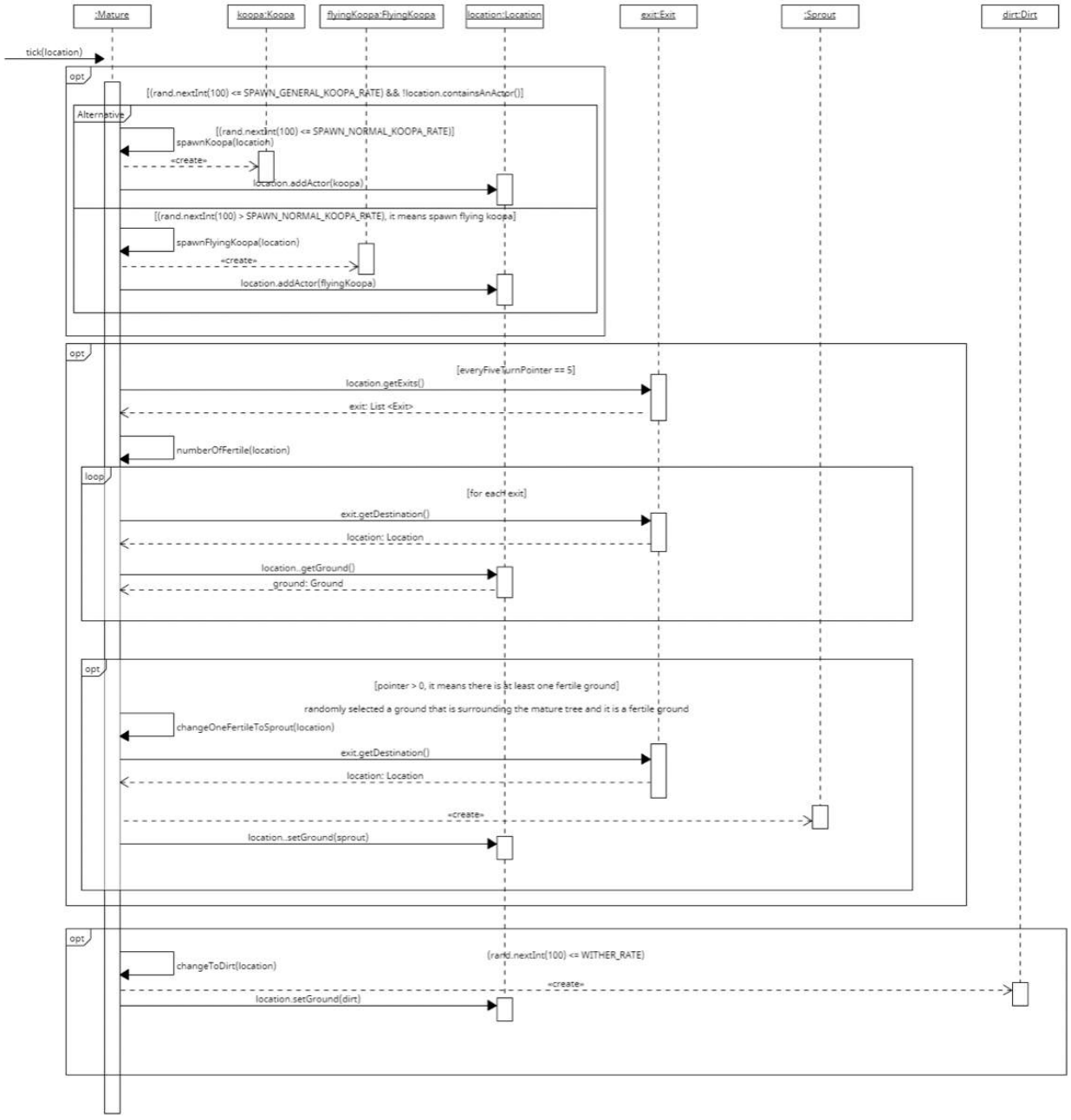
4) FlyingKoopa: A class that represents one of the enemies in the Game, which is FlyingKoopa

5) GeneralKoopa: It is an abstract class that functions as a base class for subclasses (e.g. FlyingKoopa, Koopa).
However, it cannot be represented on the game map as it is not concrete.

6) FireFlower: A class that represents the fire flower in the game map. After it is consumed by the actor, it will allow actor to perform the fire attack action

Assignment 2 REQ1: Let it grow!

Sequence Diagram for tick method in Mature class



Overview:

Due to the reason that mature tree can now spawn flying koopa (Assignment 3 REQ2) and sprout and sapling have 50% chance to spawn fire flowers (Assignment 3 REQ4) when it reached the growth age, I had updated the class diagram, sequence diagram

and the design rationale. However, since only minor changes are made, therefore the most part of the design rationale for this REQ will be no change. In other words, only Mature, Sapling and Sprout class will be modified due to the changes made. Please note that although Tree class extends HighGround (i.e., a new class created in Assignment 2), however, the purpose of creating HighGround is to achieve the feature in Assignment 2 REQ2, therefore HighGround class will not be considered in the design rationale and UML diagrams for REQ1.

1) Tree class

No changes are made between Assignment 2 and Assignment 3.

2) Sprout class

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

The only change that I made between Assignment 2 and Assignment 3 is when the Sprout instance' age reaches 10, before we change the type of ground at the location to be Sapling, we will have a 50% chance to spawn a fire flower on that location. This could be done by using an if statement. However, the rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

3) Sapling class

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

The only change that I made between Assignment 2 and Assignment 3 is when the Sapling instance's age reaches 10, before we change the type of ground at the location to be Mature, we will have a 50% chance to spawn a fire flower on that location. This could be done by using an if statement. However, the rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details

4) Mature class

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

The only change that I made between Assignment 2 and Assignment 3 is that for each turn, the mature tree will have 15% chance to spawn either Koopa or FlyingKoopa on that location. However, the rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

5) Enemy class

No changes are made between Assignment 2 and Assignment 3.

6) GeneralKoopa class

It is an abstract class that functions as a base class for subclasses (e.g., FlyingKoopa, Koopa). However, since GeneralKoopa class will not be implemented in this REQ (i.e., it's implementation will only be considered in Assignement 2 REQ3), therefore the design rationale for GeneralKoopa class will not be available in this section. **Hence, please go to the Assignement 2 REQ3 section in the following pages to see the design rationale for GeneralKoopa class.**

7) Koopa class

No changes are made between Assignment 2 and Assignment 3.

8) FlyingKoopa class

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

9) Coin class

No changes are made between Assignment 2 and Assignment 3.

10) Dirt class

No changes are made between Assignment 2 and Assignment 3.

11) Status class

No changes are made between Assignment 2 and Assignment 3.

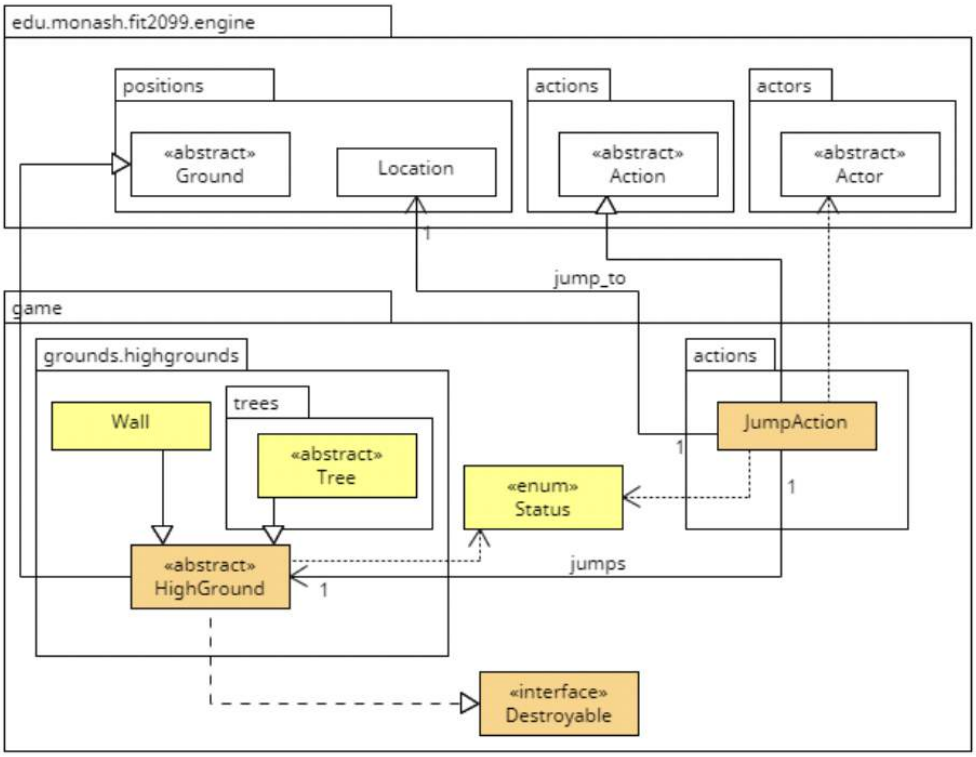
12) FireFlower class

FireFlower class is a class that represents the fire flower in the game map. After it is consumed by the actor, it will allow actor to perform the fire attack action. However, since FireFlower class will not be implemented in this REQ (i.e., it's implementation will only be considered in Assignment 3 REQ4) and in this REQ, it only creates the FireFlower instance without considering any features of a FireFlower instance, therefore the design rationale for FireFlower class will not be available in this section. **Hence, please go to the Assignment 3 REQ4 section in the following pages to see the design rationale for FireFlower class.**

REQ2: Jump Up, Super Star!

REQ2: Jump Up, Super Star!

Note:
New classes are in Orange color
Modified classes are in Yellow color
Classes from Original Base Code are in white color



Overall responsibility for New and Modified classes

1) Wall: A class that represents the wall in the Game Map

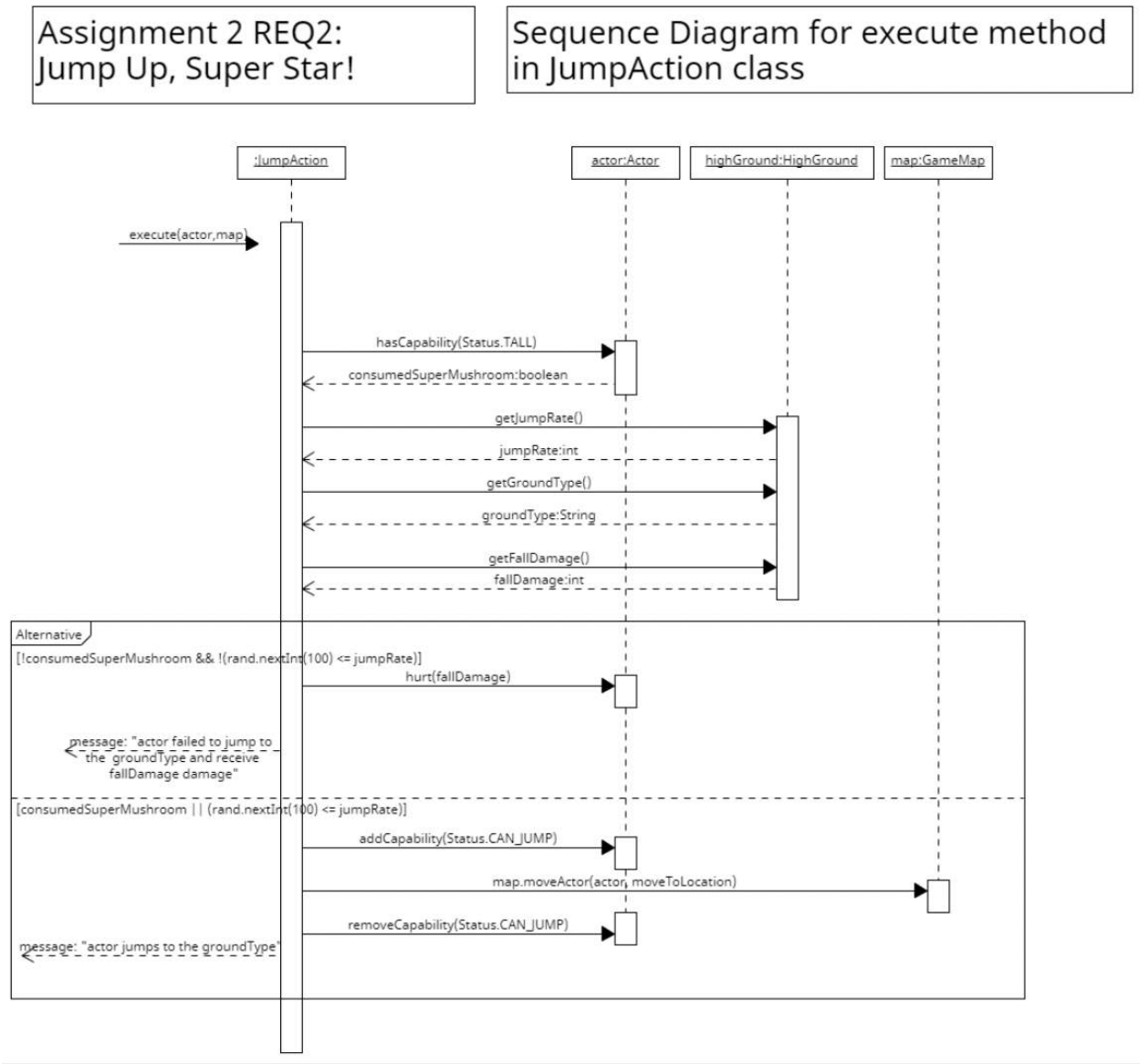
2) Tree: It is an abstract class that functions as a base class for subclasses (i.e., Mature, Sapling and Sprout). However, it cannot be represented on the game map as it is not concrete.

3) HighGround: It is an abstract class that functions as a base class for subclasses (i.e.,Tree, Wall and so on). However, it cannot be represented on the game map as it is not concrete.

4) Destroyable: An interface that allows any ground to use this interface to perform corresponding operation after it is destroyed

5) 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.

6) JumpAction: An action that allows the actor to jump



Overview:

In assignment 2, I made the mistake such that I used the wrong arrowheads in the class diagram. The reason is that when I updated my class diagram from Assignment 1 to Assignment 2, I forgot to change the arrows for several classes although I had changed my implementation between two Assignments. Therefore, I decided to update my class diagram. However, unlike the class diagram for this REQ, my sequence diagram and design rationale for this REQ are consistent with my implementation in Assignment 2. In other words, I will only update the class diagram for this REQ. Thus, even though I had included the sequence diagram and the design rationale here, they are not being

modified at all (except that I had removed my mistake made in the design rationale in Assignment 2). The reason to do so is to let the marker check for consistency easier.

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, 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 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:

Why I chose to do it that way:

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.

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. 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 enum

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. Therefore, please refer to the design rationale of this class in assignment 1.

6) Destroyable interface

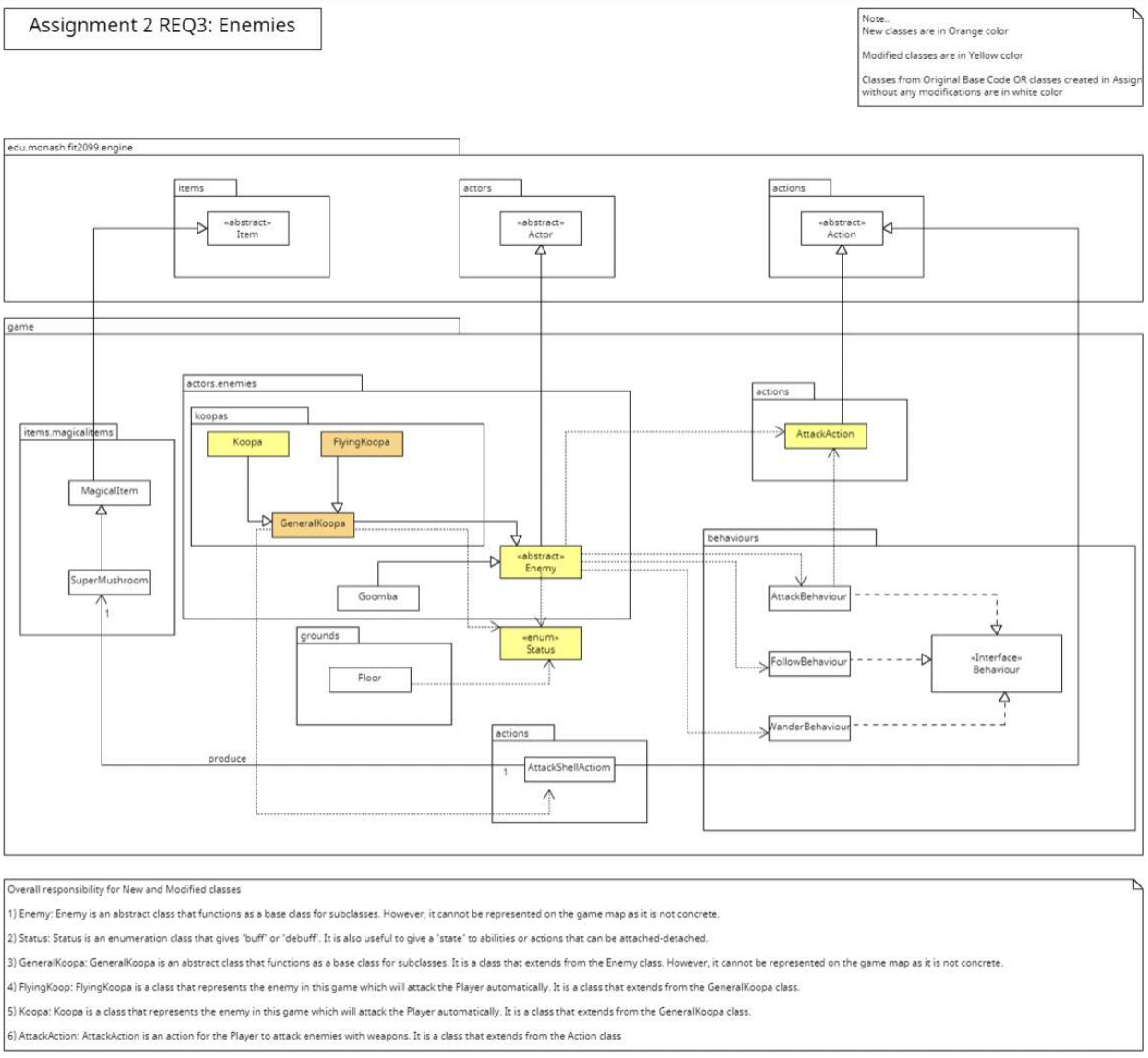
What changed in the design between Assignment 1 and Assignment 2 and Why:

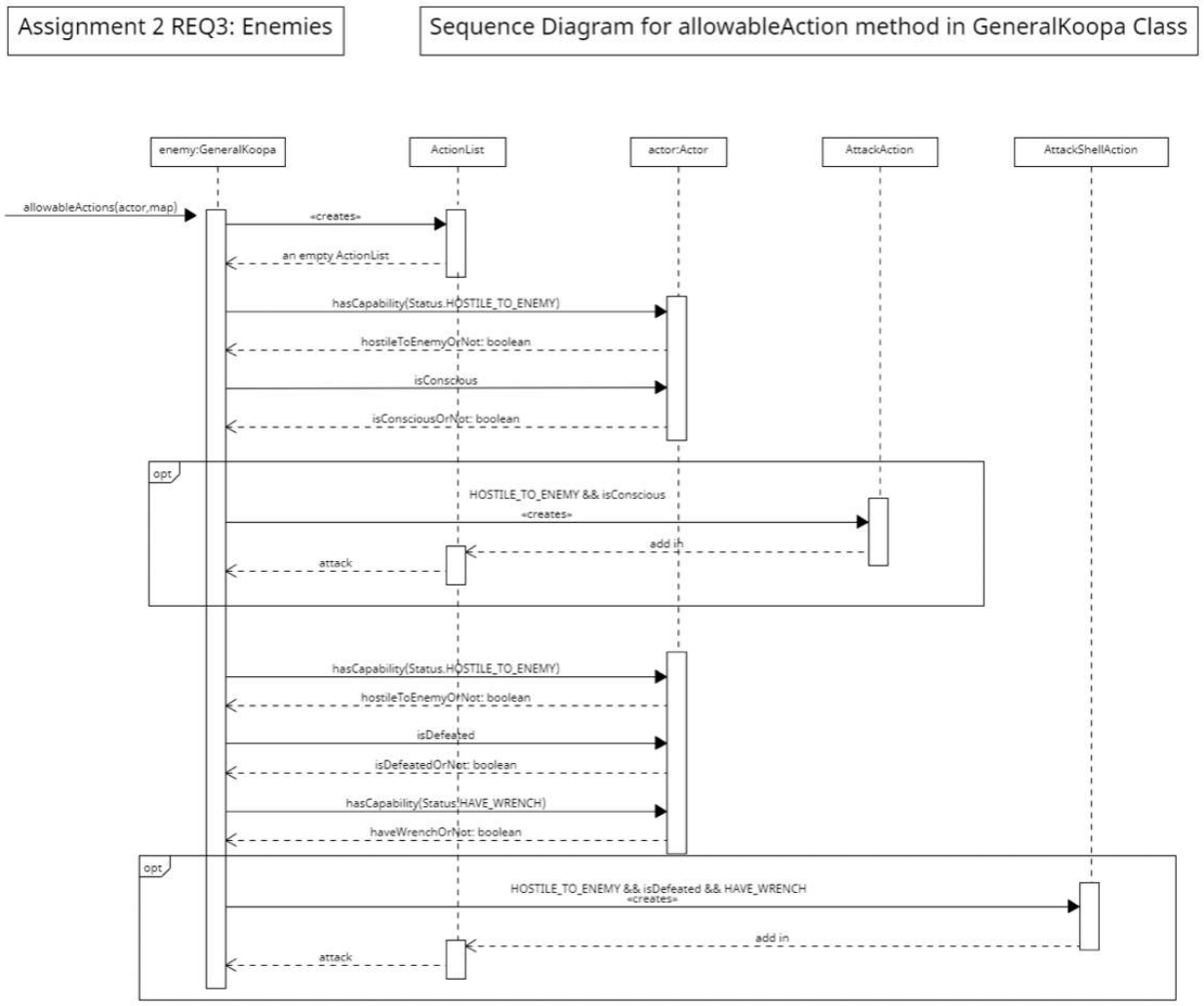
According to the assignment specification, when the player is invincible, it will destroy the high ground (i.e., convert it to dirt) and drop 5 coins. However, in assignment 1, we did not create this class and we planned to do these operations inside Wall and Tree, which is a bad design practice. Thus, 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, 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. 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 the developer to know where the problem is.

REQ3: Enemies





Overview:

Due to the design changes for achieving Assignment 3 features, there will be some classes created or modified in Assignment 2's design. (i.e., GeneralKoopa and FlyingKoopa created in the extended system, Koopa and Status will be modified due to the new design.) Besides, GeneralKoopa and FlyingKoopa is part of Assignment 3 REQ2, so detailed explanation will be state in Assignment 3 design rationale. The design rationale for each new or modified class is shown on the following pages.

1) GeneralKoopa

Why I choose to do it that way:

In Assignment 3, I created a new abstract class called GeneralKoopa to be the parent class of all Koopas(normal Koopa and Flying Koopa) in this game, and GeneralKoopa is extends Enemy to obtain all the enemy features. By adding an abstract GeneralKoopa class, all the common behaviors of Koopas can be state in this class and its child class can inherit form it directly and save in a hashmap called behaviors. This is to avoid too many repeated code, this obeys the DRY design principle, which make our code a good maintenance. Besides that, if more Koopas need to be implemented in this game, we don't have to repeat so many codes. This follows the open close principle because when more Koopas added, you do not have to modified GeneralKoopa class but allow additional class to extends it, hence it allow extension.

2) FlyingKoopa

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

In Assignment 3, I created a new class called FlyingKoopa because it is a requirement stated in Asssignment 3 REQ2, so the detailed explanation will be written in Assignment 3 design rationale.

3) Enemy

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

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

4) Goomba

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

There is no changes in this class of REQ3 between Assignment 2 and Assignment 3.

5) Koopa

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

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

Why I choose to do it that way:

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

6) Status Enum

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

For this enumeration class, I create a new capability for FlyingKoopa which is CAN_FLY. Since it is part of Assignment 3 REQ2 requirement, I will explain this in details in Assignment 3 design rationale.

7) Floor

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

There is no changes in this class of REQ3 between Assignment 2 and Assignment 3.

8) AttackAction

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

There is no changes in this class of REQ3 between Assignment 2 and Assignment 3.

9) AttackBehavior

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

There is no changes in this class of REQ3 between Assignment 2 and Assignment 3.

10) AttackShellAction

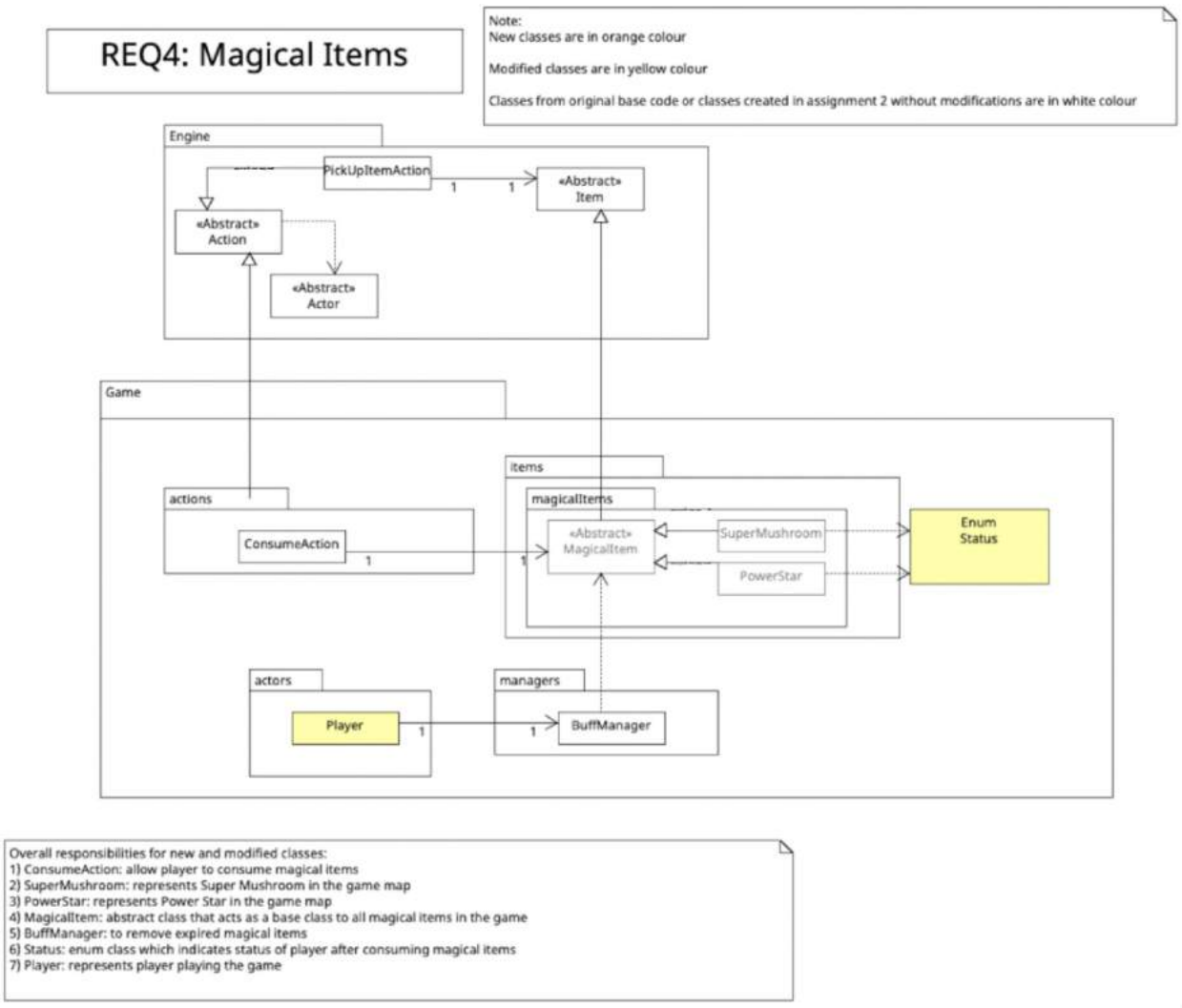
What changed in the design rationale between Assignment 2 and Assignment 3 and Why:

There is no changes in this class of REQ3 between Assignment 2 and Assignment 3.

REQ 4: Magical Items

Overview

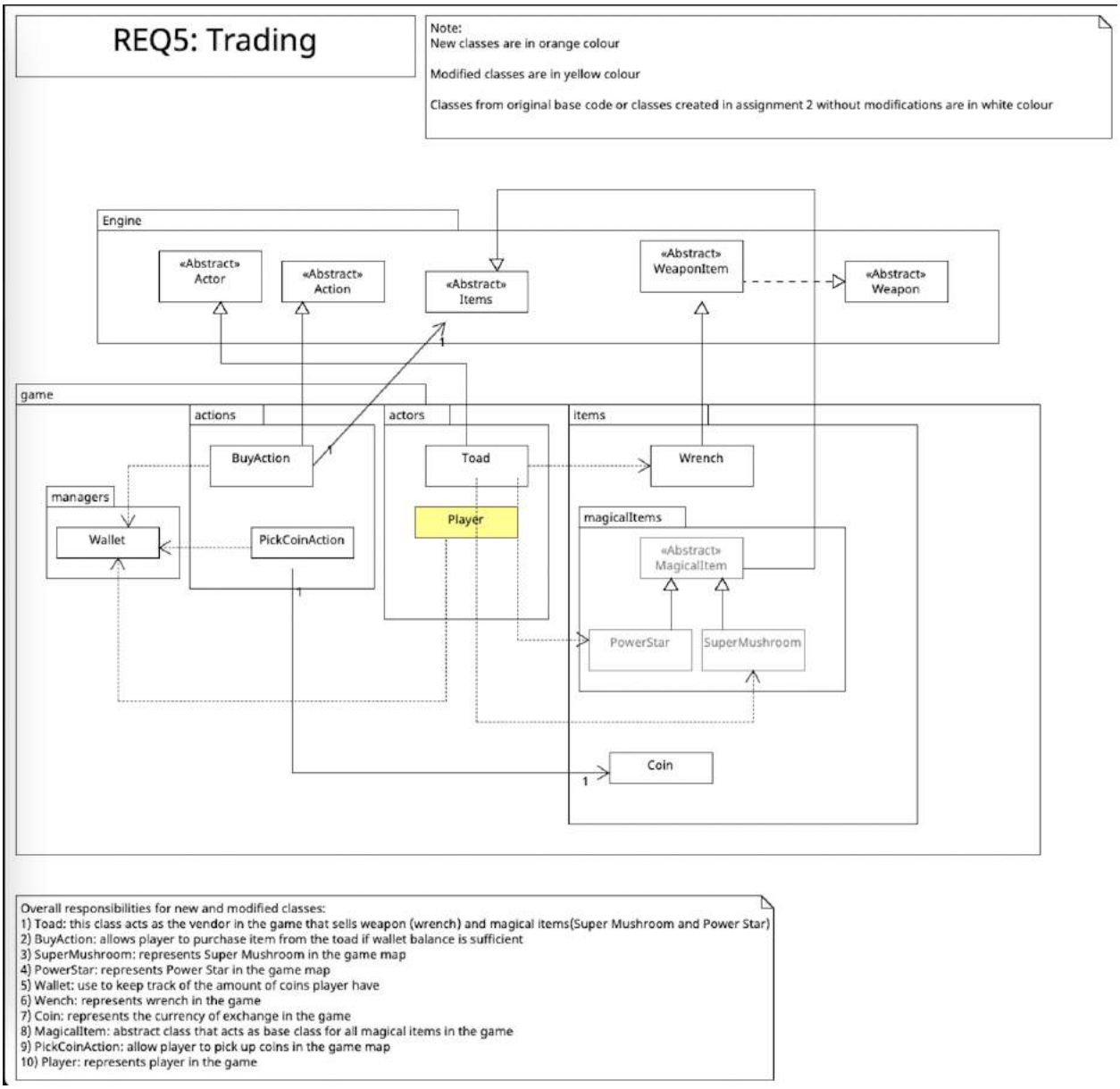
There are no changes to classes in involved in this REQ. Only arrangement of packages to provide easier searching and location of classes and interfaces. Hence only the class diagram is modified.



REQ 5: Trading

Overview

Similar to REQ 4, there are no changes to classes in involved in this REQ. Only arrangement of packages to provide easier searching and location of classes and interfaces. Hence only the class diagram is modified.



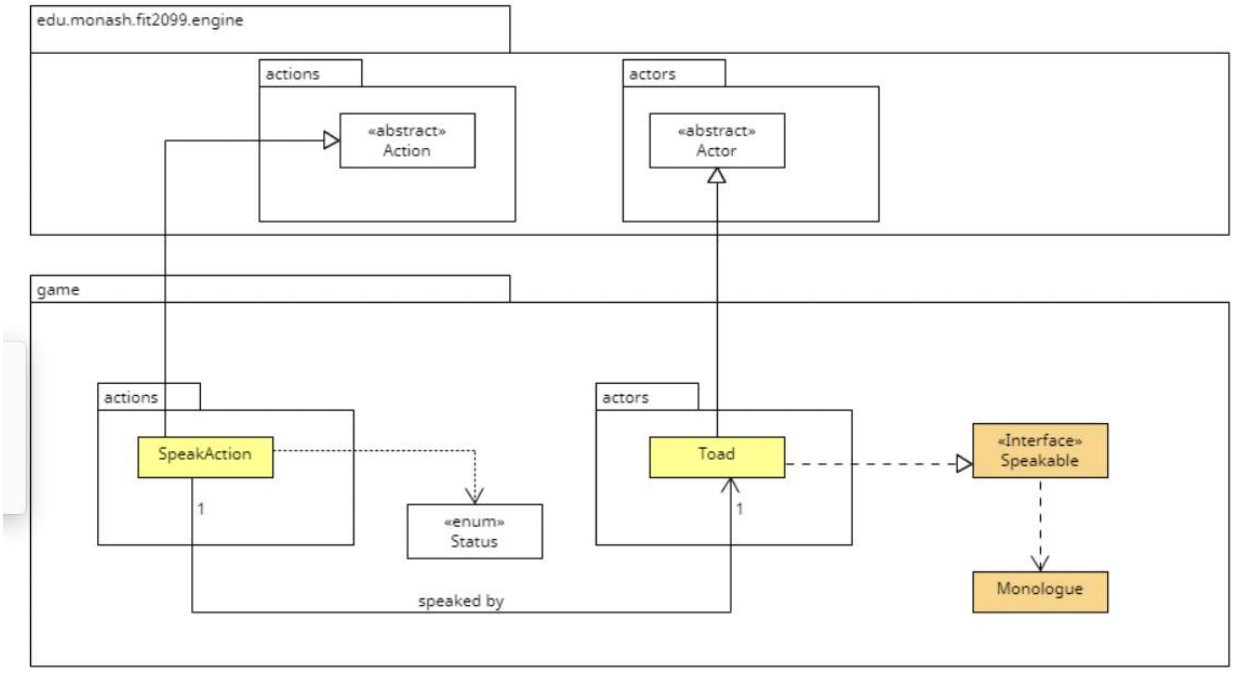
REQ6: Monologue

Assignment 2 REQ6: Monologue

Note..
New classes are in Orange color

Modified classes are in Yellow color

Classes from Original Base Code OR classes created in Assignment 2 but without any modifications are in white color



Overall responsibility for New and Modified classes

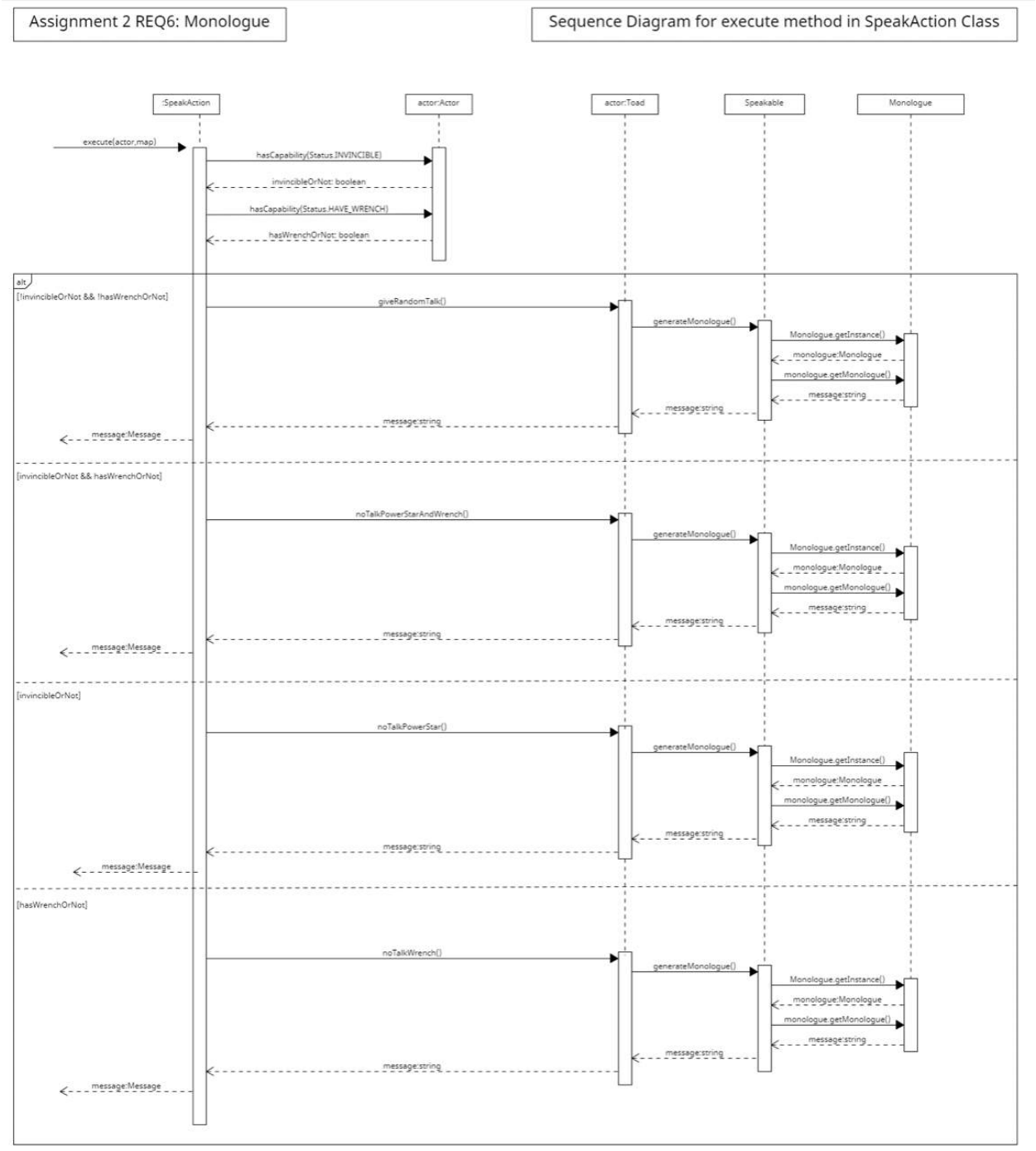
1) SpeakAction: SpeakAction class is a class that allows the actor to have a conversation with toad(friendly NPC) to get some useful information.

2) Toad: Toad is a class that represents the NPC Toad in the game map. It is a class that extends from the Actor class.

3) Status: Status is an enumeration class that gives 'buff' or 'debuff'. It is also useful to give a 'state' to abilities or actions that can be attached-detached.

4) Speakable: Speakable is an interface that contains 2 methods for the actors to speak.

5) Monologue: Monologue is a class contains all the monologue of actors.



Overview:

Due to the design changes for achieving Assignment 3 features, there will be some classes created or modified in Assignment 2's design. (i.e., Monologue and Speakable created in the extended system, Toad and SpeakAction will be modified due to the new design.) Besides, Monologue and Speakable is part of Assignment 3 REQ5, so detailed

explanation will be state in Assignment 3 design rationale. The design rationale for each new or modified class is shown on the following pages.

1) SpeakAction

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

There is no changes in this class of REQ6 between Assignment 2 and Assignment 3.

2) Toad

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

In Assignment 2, I added all the sentence belongs to Toad into the arraylist: toadTalk, and generate the monologue by using getReplyString() to modify which sentence to be talk depends on different capabilities the player had and different index to be used. But In Assignment 3, the design is changed, all of the actors can talk every even turn automatically, so the way we get corresponding monologue is change to use Speakable interface and Monologue class instance.

Why I choose to do it that way:

To reduce the repetitive code in those actor’s class, I created an interface Speakable and Monologue class to store all the monologues of this game. I let toad implements Speakable, so it can generate monologues by using the default method stated in Speakable and get the correct monologues by using boundary index. By doing so, we don’t have to repeat all the methods in every actor’s class, this obeys DRY principle.

3) Monologue

Why I choose to do it that way:

By create the monologue class, I can store all the monologues belongs to Toad inside the arraylist of this class. For Toad, it has two ways of speaking, one is when Mario interact with it and another one is generated automatically every evern turn, so by creating an instance of this class, we can get the instance of the monologue class inside the Speakable interface in order to use the boundary index to generate the correct

monologues of Toad. For the case of speaking under interaction, we generate the monologue depends on the capability of Mario. For the case of speaking automatically, we generate the monologues randomly from the four sentences. By doing so, we follow the SRP(single responsibility principle) as this class is focus on generate monologues of actors.

4) Speakable

Why I choose to do it that way:

By create a interface called Speakable, toad will implement this interface in order to use the default methods inside to get the correct monologues and speak automatically. By using this interface, it makes our code less repetitive as we can use its default method in any class which implement this interface. It follows the DRY(don’t repeat yourself) principle.

5) Application

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

There is no changes in this class of REQ6 between Assignment 2 and Assignment 3.

6) Status Enum

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

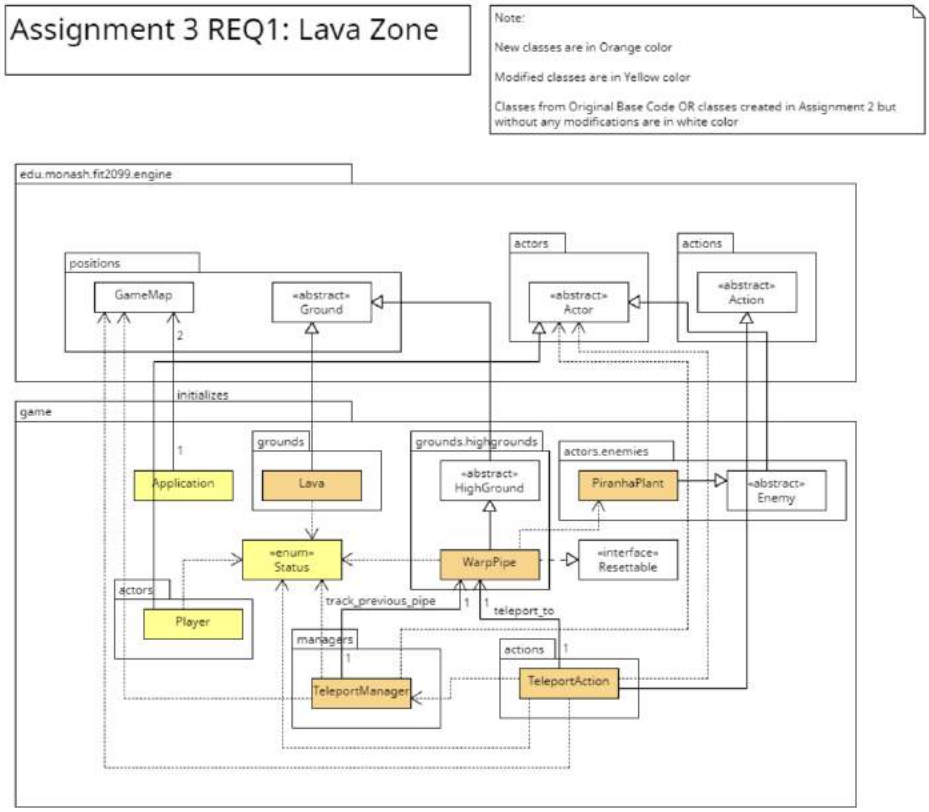
There is no changes in this class of REQ6 between Assignment 2 and Assignment 3.

Assignment 3

REQ1: Lava Zone

Overview:

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



Overall responsibility for New and Modified classes

1) Application: The main class for the Mario World game. The main method of this class is used to initialize the game map and add necessary instances to the game map.

2) Player: Class representing the Player (i.e., the mario in the game map) This is the actor that will be controlled by the user using the console.

3) Lava: A class that represents the lava in the Game Map.

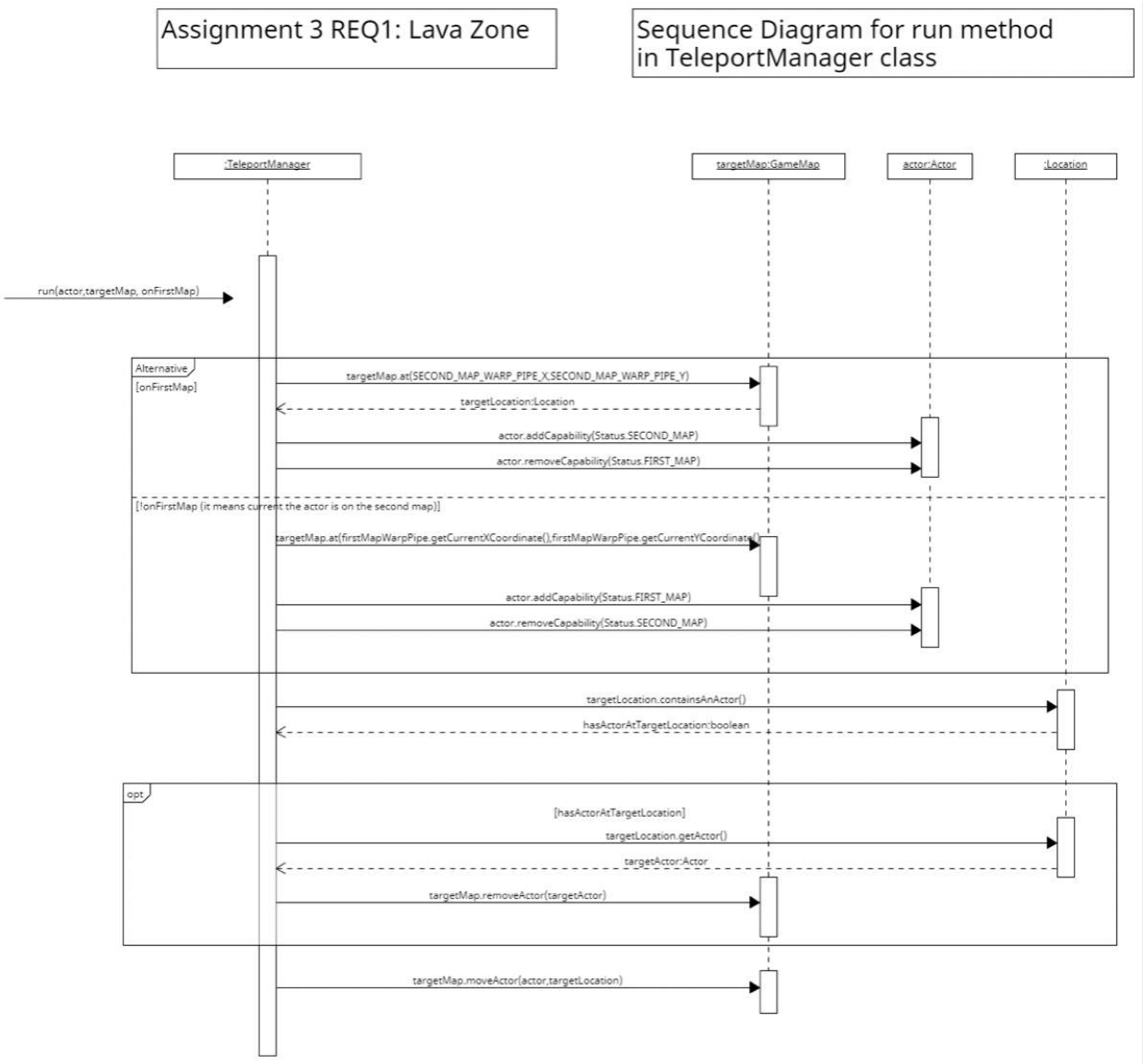
4) Status: This enum class is to give 'buff' or 'debuff'. It is also useful to give a 'state' to abilities or actions that can be attached-detached.

5) TeleportManager: A global Singleton manager that is responsible to perform teleportation after teleport action is executed.

6) WarpPipe: A class that represents the warp pipe in the Game Map.

7) TeleportAction: An action that allows the actor to teleport to the desired location.

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



1) Application class

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

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

Why I chose to do it that way:

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

2) Status enum

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

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

Why I chose to do it that way:

By declaring the two GameMap instances as the private class attribute and only allowing public getters to access them, we are ensuring the encapsulation. Hence, we are protecting our GameMap objects from unwanted access by other classes. In addition, we can see that the two new methods are just a query. Hence, we are following the Command-Query Separation Principle as the new methods are either a command or query but not both. With this design, we will not use any more literal to specify whether a Player/Actor has a particular capability in another class as we can make use of these constants. Hence, we are adhering to the "avoid excessive use of literals" principle. Besides that, if there are more actors who can perform teleportation, we could just add TELEPORT to its capability set. This kind of implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

3) Player class

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

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

4) Lava class

Why I chose to do it that way:

According to the assignment specification, there is a new type of ground called lava that will inflict 15 damage per turn when the player steps on them. Hence, to achieve this feature, I created a new class called Lava. Lava is a class that extends Ground. Inside its tick method, it would first get the actor that is standing on that location using location.getActor(). If an actor is standing at that location (i.e., getActor() does not return null), then it would hurt the actor by 15 damage. Besides that, if the actor is dead because of the damage, it will be directly removed from the map. Since the enemy is not allowed to enter the lava ground, inside the canActorEnter method, I will return true if and only if the actor does not have IS_ENEMY (i.e, created in Assignment 2) capability. With the design above, we are adhering to the Single Responsibility Principle as the Lava
class only focuses on what will happen when the actor standing in that location and the method within the Lava class only shows the properties of a Lava. Furthermore, for each of the properties of a Lava instance (e.g., display character and damage per turn), I store them as private constant class attributes. Consequently, we are following the "avoid excessive use of literals" principle as it makes our code more readable.

5) WarpPipe class

Why I chose to do it that way:

In assignment 3, there is a new type of ground called WarpPipe such that when the Player stands on it, it can perform teleportation to its desired location. To achieve this feature, I created a new class called WarpPipe. Since the Player must jump to the WarpPipe to perform teleportation, I realized that it has a lot of common code with the HighGround class,

therefore I let WarpPipe extend HighGround. By doing so, we are reducing our repeated code and, hence, following the DRY principle. According to the FAQ of Assignment 3, I realized that Warp Pipe should not be destroyable when the Player is invincible and standing on it, hence I would need to override the tick method in HighGround (since HighGround is meant to be destroyable in Assignment 2 and its tick method does this operation). After that, only if the player is standing on it and no enemy (i.e., PiranhaPlant) is standing on it, the TeleportAction will be available to the user, and it will pass itself (i.e., the current WarpPipe instance) to the constructor of the TeleportAction (please refer to the design rationale of TeleportAction for more details). Otherwise, if the player is not standing on it (i.e., besides it) then JumpAction will be available (i.e., by calling the HighGround allowableAction method).

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

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

6) PiranhaPlant class

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

7) TeleportManager class

Why I chose to do it that way:

TeleportManager is a global singleton manager responsible to teleport the actor to the desired location. It will only be called if the actor executes TeleportAction. Please refer to TeleportAction class for more details. The reason to have this TeleportManager class is that we need a way to keep track of the previous WarpPipe and store the coordinates of the warp pipe in the second map. This cannot be done in the TeleportAction class as it will violate the Single Responsibility Principle since TeleportAction should not be responsible for storing this information. Hence, TeleportManager is needed. According to the assignment specification, assuming there are two warp pipes called A and B in the first map, it says that if the Player teleported to the second map using pipe A, then when the Player teleported back to the first map using the only pipe in the second map, the Player must be teleported to the previous pipe, which is pipe A, not B. Hence, to keep track of the previous warp pipe, I created a private class attribute to store the previous warp pipe instance (i.e., the warp pipe used in the first map), by doing so, when the actor needs to teleport back to the first map, we can teleport to that location using the X and Y coordinates of the previous WarpPipe instance (due to the two getters mentioned in WarpPipe class section). The previous warp pipe only will be stored (i.e., using the setter) if the actor wanted to teleport from the first map to the second map. The main method in the TeleportManager class is the run method. It takes in three input parameters, which are the actor that wants to teleport, the map that the actor wants to teleport to, and a Boolean that indicates whether the actor is currently on the first map or not. After that, it will perform the teleportation (i.e., move the actor to the desired location) corresponding to the value of the parameters, please refer to the code and design diagrams for more details.

With this design, we are following the Command-query principle as none of our methods in this class are both command and query. Besides that, we also store the coordinate of the warp pipe in the second map as the private constant class attributes. Thus, we are following the “avoid excessive use of literals” principle as it makes our code more readable. Additionally, I had declared the class attribute (i.e., WarpPipe instance that indicates the previous warp pipe) as private and only the public setter can modify it, hence ensuring encapsulation. Moreover, for the run method and setFirstWarpPipe method, I had checked the pre-condition of the input

parameter (i.e., actor and targetMap for run method, and firstMapWarpPipe for setFirstMapWarpPipe) such that it cannot be null, else it will throw an IllegalArgumentException. Thus, our design aligns with the Fail Fast Principle as it fails the system immediately when the code detects something is wrong and allows the developer to know where the problem is.

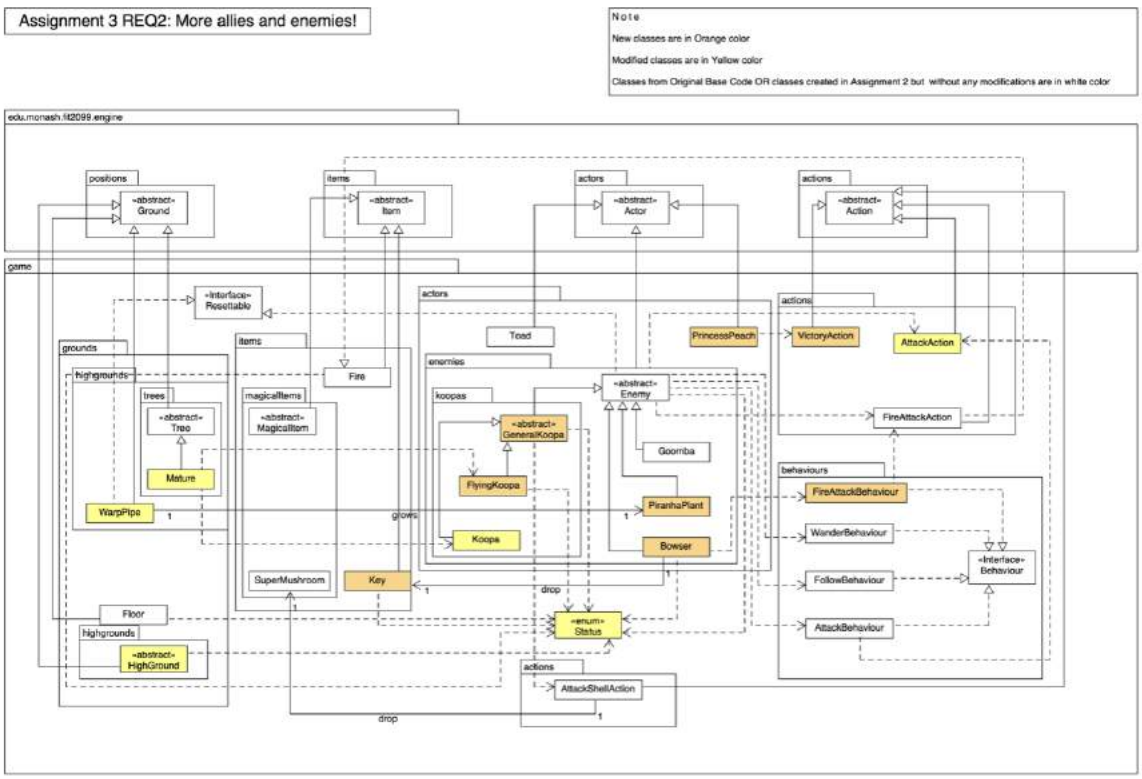
8) TeleportAction class

Why I chose to do it that way:

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

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

REQ2: More allies and enemies!



Overall responsibility for New and Modified classes

1) WarpPipe: WarpPipe is the place for Player to teleport to second map. It is a class that extends from the Ground class and implements Resettable.

2) HighGround: HighGround is an abstarct class as a base class for subclasses(i.e, Wall and WarpPipe). However, it cannot be represented on the game map as it is not concrete.

3) Mature: Mature tree will spawn Koopa or Flying Koopa under spawn rate. It is a class that extends from the Tree class.

4) Key: Key is dropped by Bowser after Player defeat Bowser. It is a class that extends from the Item class.

5) GeneralKoopa: GeneralKoopa is an abstract class that functions as a base class for subclasses.However, it cannot be represented on the game map as it is not concrete.

6) FlyingKoop: FlyingKoopa is a class represents the enemy in this game which will attack Player automatically. It is a class that extends from the GeneralKoopa class.

7) Koopa: Koopa is a class represents the enemy in this game which will attack Player automatically. It is a class that extends from the GeneralKoopa class.

8) PiranhaPlant: PiranhaPlant is an enemy who stand at warppipe and attack Player when player come close. It is a class that extends from the Enemy class.

9) Bowser: Bowser is an enemy who attack Player by using fires. It is a class that extends from the Enemy class.

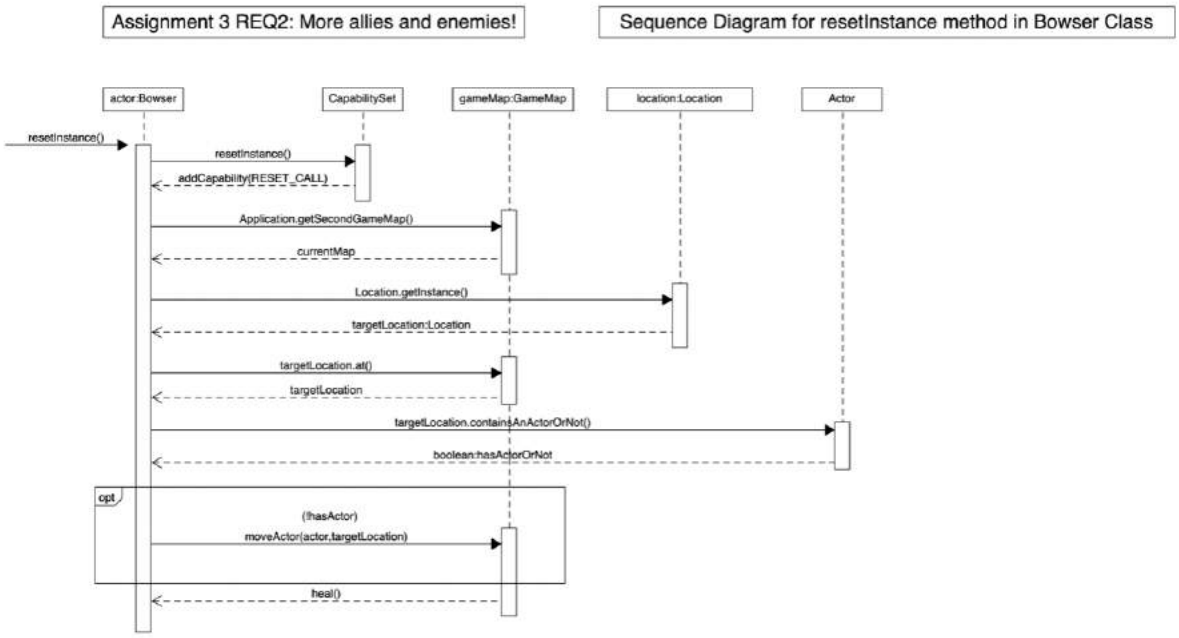
10) Status: Status is an enumeration class that gives 'buff' or 'debuff'. It is also useful to give a 'state' to abilities or actions that can be attached-detached.

11) PrincessPeach: PrincessPeach is an actor who caught by Bowser and wait for Mario. It is a class that extends from the Actor class.

12) VictoryAction: VictoryAction is an action for Player to interact with PrincessPeach to end this game. It is a class that extends from the Action class

13) AttackAction: AttackAction is an action for Player to attack enemies by weapons. It is a class that extends from the Action class

14) FireAttackBehaviour: FireAttackBehaviour is a class contains the fire attack actions of Enemies(Bowser) to attack Player automatically. It is a class that implements Behaviour class.



Overview

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

1) PrincessPeach

Why I choose to do it that way:

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

2) GeneralKoopa

Why I choose to do it that way:

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

Besides that, if more Koopas need to be implemented in this game, we don’t have to repeat so many codes. This follows the open close principle because when more Koopas added, you do not have to modified GeneralKoopa class but allow additional class to extends it, hence it allow extension. Furthermore, by adding the constant class attributes, it follow "avoid excessive use of literals" principles as I’m now avoiding the use of magical numbers and literals.

3) FlyingKoopa

Why I choose to do it that way:

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

4) Koopa

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

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

Why I choose to do it that way:

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

5) PiranhaPlant

Why I choose to do it that way:

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

6) Bowser

Why I choose to do it that way:

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

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

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

7) Key

Why I choose to do it that way:

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

8) Status Enum

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

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

Why I choose to do it that way:

These status are useful in my implementation as, I can utilize the engine code provided to check the actor's capability to simplify our code. The HAVE_KEY status is used to indicate whether this actor pick up the key or not. This status is initialized in Key class, by having this status, the actor can interact with PrincessPeach in order to end this game.

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

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

9) Mature

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

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

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

Why I choose to do it that way:

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

10) WarpPipe

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

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

Why I choose to do it that way:

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

11) HighGround

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

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

Why I choose to do it that way:

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

12) VictoryAction

Why I choose to do it that way:

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

13) AttackAction

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

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

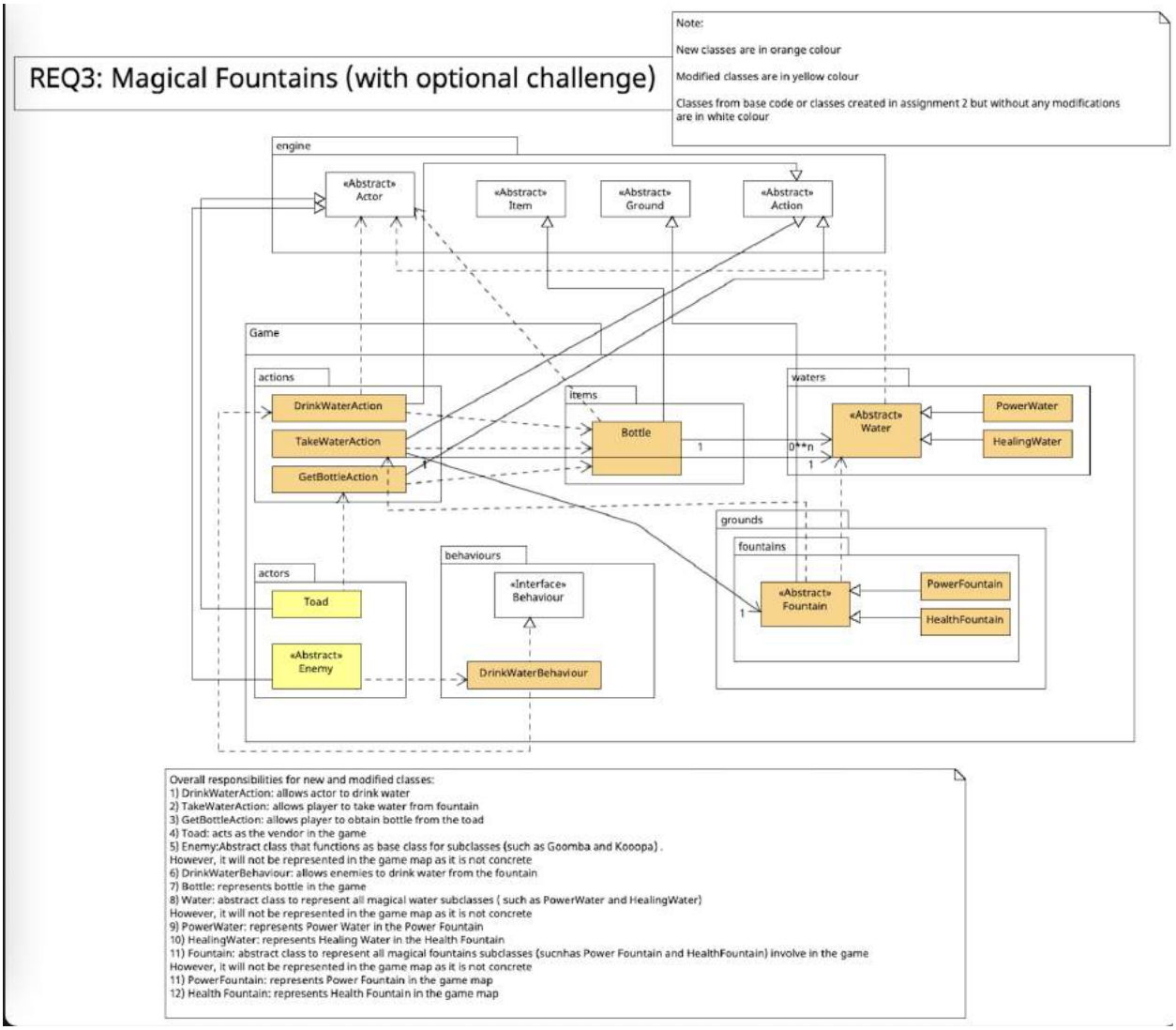
Why I choose to do it that way:

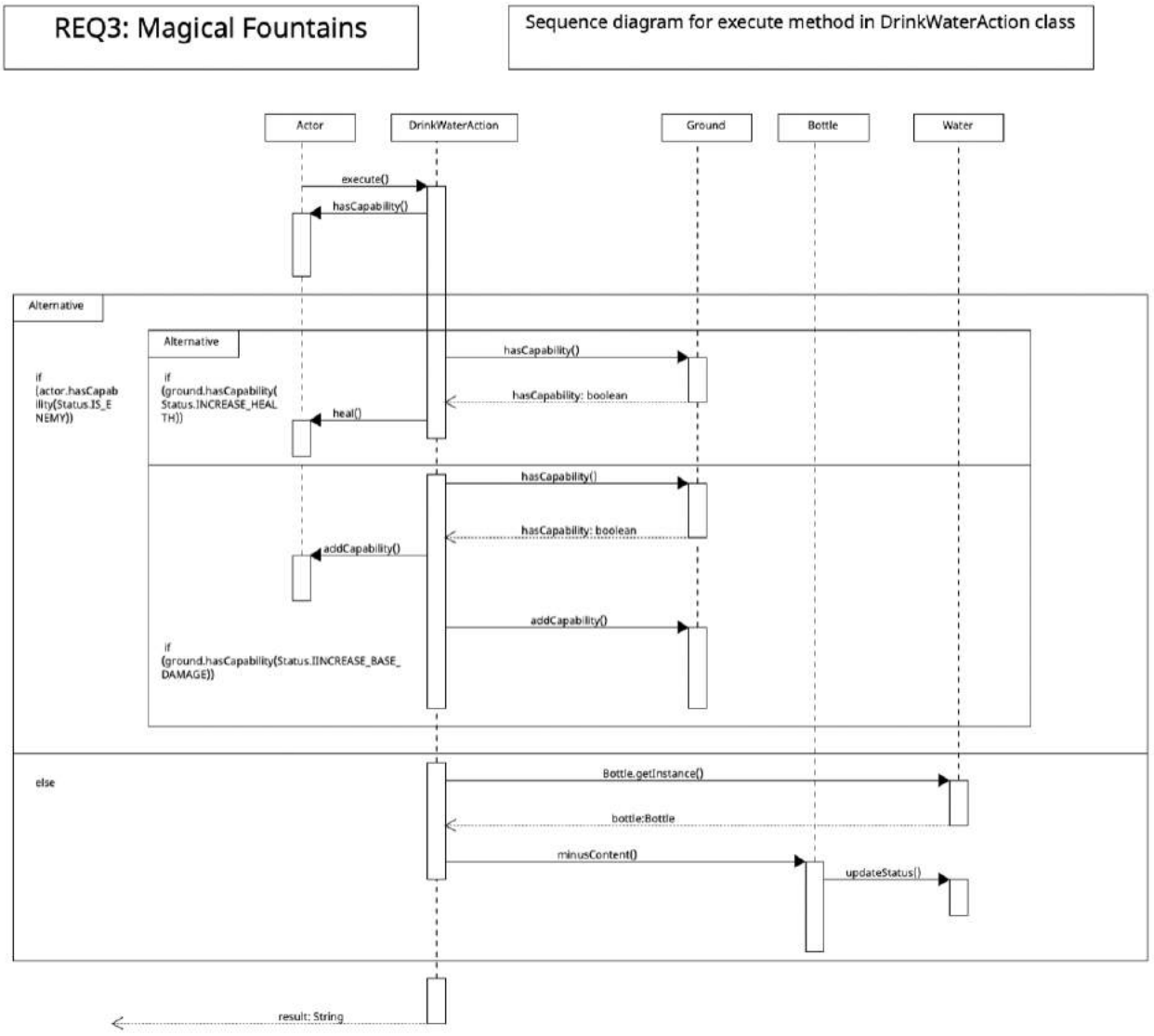
By doing so, we can make sure that when the actor who has capability to drop a key is dead, it will not be removed from the map straight away without drop the key, which make sure the game will run properly under this condition.

14) FireAttackBehaviour
Why I choose to do it that way:

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

REQ 3: Magical fountains (with optional challenge)





1) DrinkWaterAction class

This class which extends Action class is to enable player to drink water from the bottle and enemy to drink water from fountain. The water drank will minus one water slot from player's bottle.

Description of method:

- execute method is a method overridden from its parent class. It will check if the actor is enemy. if actor is an enemy, the actor can gain its effect straightaway from the fountain. Else, water will be added into bottle (if actor is a player). A description will then be display in the console showing what water is drank of which actor.

```
@Override
    public String execute(Actor actor, GameMap map) {
        String result= "";
        if (actor.hasCapability(Status.IS_ENEMY)){
            if (ground.hasCapability(Status.INCREASE_HEALTH)) {
                actor.heal(HEALED_HP);
                result = actor + " drank Healing water";
            } else if (ground.hasCapability(Status.INCREASE_BASE_DAMAGE)) {
                actor.addCapability(Status.POWER_WATER);
                result = actor + " drank Power water";
```

```
            }
            ground.addCapability(Status.DRANK_BY_ENEMY);

        } else {
            result = actor + " drank " + Bottle.getInstance().getLast();
            Bottle.getInstance().minusContent(actor);
        }
        return result;
    }
```

why i choose to do it that way:

By doing this, we are adhering to the Single Responsibility Principle(SRP) as this class is only in charge of allowing actor to drink water from fountains .

2) TakeWaterAction class

This class which extends Action class is to enable player to obtain water from the fountain. The water obtained will add one water slot into player's bottle.

Description of method:

- the execute method is to add one slot of water into player's bottle when player choose to take water from the fountain by obtaining the water returned from the menuDescription method.

```
@Override
    public String execute(Actor actor, GameMap map) {
        if (actor.hasCapability(Status.HAS_BOTTLE)) {
            Bottle.getInstance().addContent(water);
            fountain.minusContent();
        }
        return menuDescription(actor);
    }
```

Why i choose to do it that way:

By doing this, we are adhering to SRP as this class is only in charge of allowing player to refill water from fountains.

3) GetBottleAction

This class which extends Action class is to enable player to obtain bottle from the toad. The bottle obtained will be added into player's inventory

Description of method:

- execute method is to add bottle into player's inventory after player obtain bottle from the toad

```
public String execute(Actor actor, GameMap map) {
        actor.addItemToInventory(bottle);
        return menuDescription(actor);
    }
```

why i choose to do it that way:

By doing this, we are adhering to SRP as this class is only in charge of allowing player to obtain bottle from the toad.

4) Bottle class

This class represents bottle in the game. This bottle can contain unlimited magical waters. This bottle will be a permanent item that cannot be dropped or picked up. This bottle can be filled with water, and Mario can drink/consume the water inside the bottle. Each water will give a unique effect depending on its original source (fountains). Mario doesn't have a bottle in his inventory at the start of the game. Instead, Mario will obtain it from Toad.

Description of method:

- addContent method to add one slot of water into bottle
- minusContent method to minus one slot of water from bottle

```
public void addContent(Water water) {
        this.content.add(water);
    }

public void minusContent(Actor actor) {
        if (actor == null){
            throw new IllegalArgumentException("The input parameter (i.e., actor) cannot be null");
        }

        Water drankWater=content.get(content.size()-1);
        drankWater.updateStatus(actor);
        this.content.remove(drankWater);
    }
```

Why i choose to do it that way:

Since there can only be one bottle in the game, static factory method is used in Bottle class so that the constructor can be called directly instead of creating a new instance of it. By doing this, encapsulation can be use as behaviour of the Bottle class can be specify via parameters without needing to know the internal implementation of the Bottle class. This improves code readability. For example, content of the bottle can be obtained by `Bottle.getInstance.getContent()`.

Furthermore, exception is added into the minusContent method order to implement the Fail Fast principle, so that when the actor is null, the program will throw an error and stop running right after instead of continue to run until water is to be consume from the bottle. By doing this, this eases debugging process as it will be easier to identify the cause of error.

5) Water class

This is an abstract class that represents water in the fountain. Each water from a fountain provides different effects that will help Mario in his journey. Water class is an abstract class created to be the parent class of HealingWater and PowerWater classes.

Why i choose to do it that way:

This class is made to be an abstract class because this class is not use to create objects to represent anything in the game and this class is not instantiated throughout the code. Instead, it is to be inherited by HealingWater and PowerWater classes.

Since HealingWater and PowerWater are waters, they both share some same methods and hence a Water class is created to reduce duplicated codes as they can access method from Water class by calling super. By doing this, we are adhering to the Don't Repeat Yourself (DRY) principle.

Open Closed Principle (OCP) can also be implemented as we are able to add new functionalities to the water without changing existing codes in the Water class. For example, if they exist a water which have exceptionally more functions, that water class can extend this Water class, implementing all methods from Water class, and override or create additional methods in it for additional functionalities.

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

6) PowerWater class

This class represents power water that can be obtained from power fountain in the game. When the water is consumed, it increases the drinker's base
/intrinsic attack damage by 15.

Description of method:

- updateStatus method to add POWER_WATER capability to drinker.

```
public void updateStatus(Actor actor) {
        super.updateStatus(actor);
        actor.addCapability(Status.POWER_WATER);
    }
```

Why i choose to do it that way:

As explained in the Water class section, this class is extended from Water class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Water class.

7) HealingWater class

This class represents healing water that can be obtained from health fountain in the game. Drinking this water will increase the drinker's hit points/healing by 50 hit points.

Description of method:

- updateStatus method to heal drinker by 50 hp upon consumption

```
@Override
    public void updateStatus(Actor actor) {
        super.updateStatus(actor);
        actor.heal(HEALED_HP); // HEALED_HP = 50
    }
}
```

Why i choose to do it that way:

As explained in the Water class section, this class is extended from Water class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Water class.

8) Fountain class

This class represents fountain in the game. Mario can refill the bottle when he stands on the fountain. A fountain produces an endless amount of water. Each water from a fountain provides different effects that will help Mario in his journey. Fountains have limited amount of
water(10 slots of water capacity).

Description of method:

- allowable action method to allow player to refill water from the fountain if the fountain is not empty.

```
@Override
    public ActionList allowableActions(Actor actor, Location location, String direction) {
        if (location.containsAnActor() && actor.hasCapability(Status.HAS_BOTTLE) && !this.isEmpty()) {
            return new ActionList(new TakeWaterAction(this.getWater(),this));
        }
        return new ActionList();
    }
```

- tick method to keep track of number of turns in the game and minus one slot of water whenever water in it is drank my enemies or refilled by players. When content of the fountain runs out, the “turnWhenWaterRunOut” will be use to store that particular current turn, and content will be set to -1 to indicate that the fountain is empty. Content of the fountain will then be reset to 10 five turns after “turnWhenWaterRunOut”.

```
@Override
    public void tick(Location location) {
        this.currentTurn++;

        if (this.hasCapability(Status.DRANK_BY_ENEMY)){
```

```
            this.minusContent();
            this.removeCapability(Status.DRANK_BY_ENEMY);
        }

        if (this.content == 0){
            this.addCapability(Status.IS_EMPTY);
            this.turnWhenWaterRunOut =this.currentTurn;
            this.content=EMPTY_INDICATOR;
        }

        if (this.content==EMPTY_INDICATOR && (this.currentTurn - this.turnWhenWaterRunOut ==REPLENISH_TURN)){
            this.turnWhenWaterRunOut =INITIAL_TURN;
            this.setContent(MAX_CONTENT);
            this.removeCapability(Status.IS_EMPTY);
        }

    }
```

Why i choose to do it that way:

Similar to Water class, this class is made to be an abstract class because this class is not use to create objects to represent anything in the game and this class is not instantiated throughout the code. Instead, it is to be inherited by HealthFountain and PowerFountain classes.

Since HealthFountain and PowerFountain are fountains, they both share some same methods and hence this Fountain class is created to reduce duplicated codes as they can access method (such as allowableActions, tick, setContent, minusContent etc methods) from Fountain class by calling super. By doing this, we are adhering to the Don't Repeat Yourself (DRY) principle.

Open Closed Principle (OCP) can also be implemented as we are able to add new functionalities to the fountain without changing existing codes in this class. For example, if they exist a fountain which have exceptionally more functions and characteristic, that class can extend this Fountain class, implement methods from Fountain class, and override or create additional methods in it for additional functionalities.

9) HealthFountain class

This class represents health fountain ('H') in the game. This fountain contains HealingWater which players can refill it into the bottle while enemies can consume it straight away from the fountain.

Why i choose to do it that way:

As explained in the Fountain class section, this class is extended from Fountain class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Fountain class.

10) PowerFountain class

This class represents power fountain ('A') in the game. This fountain contains Power Water which players can refill it into the bottle while enemies can consume it straight away from the fountain.

Why i choose to do it that way:

Similar to HealthFountain, this class is extended from Fountain class in order to reduce duplicated code and hence adhere to the DRY principle. By doing so, default methods can be inherited from Fountain class.

11) DrinkWaterBehaviour class

This class is to enable enemies to drink water from magical fountains. it implements the Behaviour class.

Description of method:

- getAction method checks if enemy is located on a fountain. If yes, allow enemy to drink water from the fountain by returning the DrinkWaterAction.

```
@Override
public Action getAction(Actor actor, GameMap map) {
```

```
Ground currentGround = map.locationOf(actor).getGround();
    if(currentGround.hasCapability(Status.IS_FOUNTAIN)){
return new DrinkWaterAction(currentGround);
}
return null;
}
```

Why i choose to do it that way:

By doing this, we are adhering to the Single Responsibility Principle(SRP) as this class is only in charge of allowing enemy to drink water from fountains, by returning a DrinkWaterAction to enemies if the ground they are located on is a fountain.

12) Toad class

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

```
if(! otherActor.hasCapability(Status.HAS_BOTTLE)) {
            actions.add(new GetBottleAction());
        }
```

in Toad class's allowable action, a GetBottleAction will be added into otherActor's action list. This is to enable player to obtain bottle from the toad.

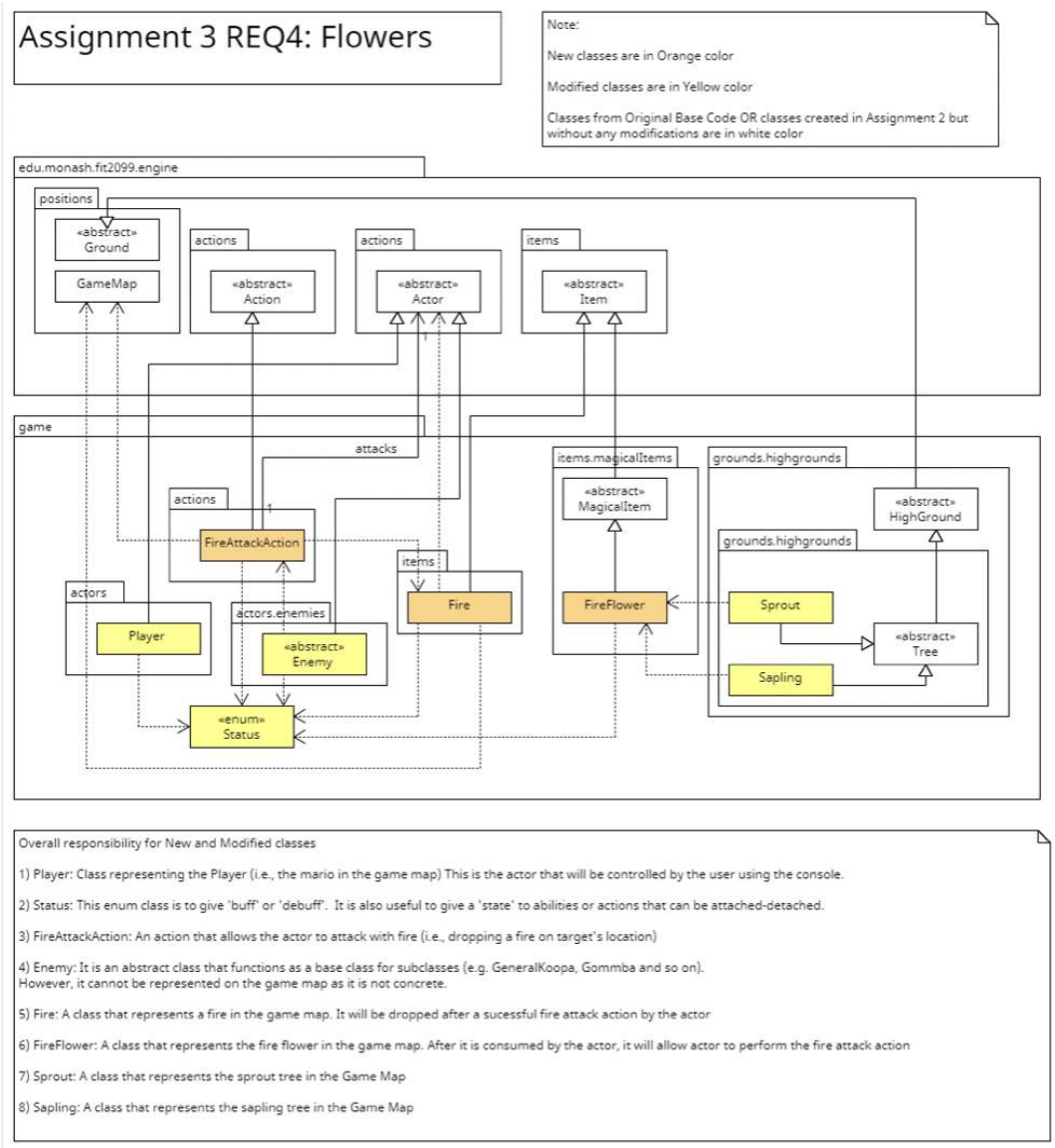
13) Enemy class

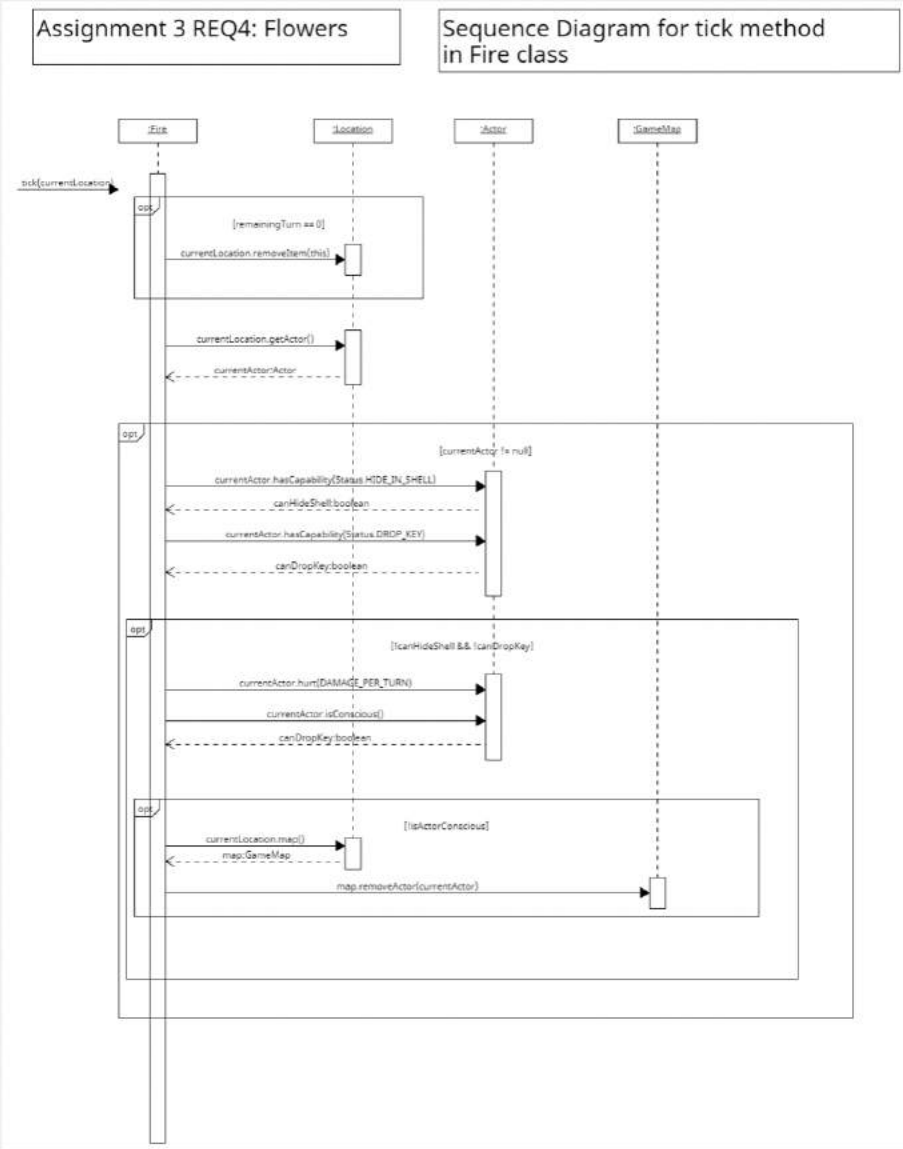
What changed in the design rationale between Assignment 2 and Assignment 3 and Why:

```
if (this.hasCapability(Status.POWER_WATER)) {
            this.damage+=EXTRA_DAMAGE;
            this.removeCapability(Status.POWER_WATER);
        }
```

The section of code above is added into Enemy class's playTurn method to check if Enemy drank the power water. If so then add EXTRA_DAMAGE, which is 15 to enemy's base damage.

REQ4: Flowers


Assignment 3 REQ4: Flowers
Note:
New classes are in Orange color
Modified classes are in Yellow color
Classes from Original Base Code OR classes created in Assignment 2 but without any modifications are in white color
edu.monash.fit2099.engine
positions
«abstract» Ground
GameMap
actions
«abstract» Action
actions
«abstract» Actor
items
«abstract» Item
game
attacks
actions
FireAttackAction
items
Fire
actors
Player
actors.enemies
«abstract» Enemy
«enum» Status
items.magicalItems
«abstract» MagicalItem
FireFlower
grounds.highgrounds
«abstract» HighGround
grounds.highgrounds
Sprout
«abstract» Tree
Sapling
Overall responsibility for New and Modified classes
1) Player: Class representing the Player (i.e., the mario in the game map) This is the actor that will be controlled by the user using the console.
2) 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.
3) FireAttackAction: An action that allows the actor to attack with fire (i.e., dropping a fire on target's location)
4) Enemy: It is an abstract class that functions as a base class for subclasses (e.g. GeneralKoopa, Gommba and so on).
However, it cannot be represented on the game map as it is not concrete.
5) Fire: A class that represents a fire in the game map. It will be dropped after a sucessful fire attack action by the actor
6) FireFlower: A class that represents the fire flower in the game map. After it is consumed by the actor, it will allow actor to perform the fire attack action
7) Sprout: A class that represents the sprout tree in the Game Map
8) Sapling: A class that represents the sapling tree in the Game Map



Overview:

To achieve this feature, there will be three new classes (i.e., Fire, FireFlower, and FireAttackAction) created in the extended system, and five existing classes (i.e., Player, Enemy, Status, Sprout, and Sapling) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) Fire class

Why I chose to do it that way:

Fire is a new class that extends the Item class. According to the assignment specification, it says that the fire will stay on the ground for three turns. Hence, I created a pointer and initialized it as 3. For each tick method being called, the pointer will be minus 1, and the fire will be removed once the pointer is equal to 0. Besides that, it will deal 20 damage per turn to the actor currently standing in the same location with the fire. After the actor gets the damage, if the actor is not conscious, then we will remove it. However, we could not directly remove those unconscious actors that can hide in the shell or drop a key, so that the next play turn method of those actors can do the operation (i.e., hide in the shell or drop a key).

With this design, we are following the Single Responsibility Principle as the method within the Fire class only shows the properties of a Fire. Additionally, for each of the properties of a Fire instance (e.g., damage per turn, display character, 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.

2) FireFlower class

Why I chose to do it that way:

According to the assignment specification, after the actor consumes the fire flower, it can perform a fire attack action for 20 turns. However, I realized that this magical item is very similar to the power star, as it also gives the actor a particular buff for a certain number of turns. Hence, I will do the same thing I did to PowerStar class, which is to let FireFlower class extend MagicalItem and override the currentStatus() and updateStatus() to do the corresponding operation. By doing so, there is no other code needed to be modified to keep track of the effect of the fire flower (e.g., remaining turn, when will it be expired, and so on) as the MagicalItem class and BuffManager class created in Assignment 2 already does the job. In other words, I just need to override those two methods based on the description and effect of fire flowers to make the implementation work. Besides that, we will add the FIRE_ATTACK constant to the actor that consumed it. Please refer to the Status class section for more details regarding this constant.

With the above design, we are following the DRY principle as I did not create a class that has repeated code that is the same as those magical items (e.g., super mushroom and power star) since I just extended the MagicalItem class and use the common code. Besides that, we are following the Single Responsibility Principle as the method within FireFlower class only focuses on the buff that can be given to the consumer (i.e., player). Moreover, for each of the properties of a FireFlower instance (e.g., name, display character, 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.

3) Status enum

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

In Assignment 3, I created a new constant called FIRE_ATTACK. This constant will be added to the actor once it consumes the fire flower. As a result, fire attack action will be available to the actor.

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.

4) FireAttackAction class

Why I chose to do it that way:

FireAttackAction class allows the actor to perform a fire attack action (i.e., dropping fire on the target’s location). It is a new class that extends Action. It will be available to the player once it consumes the fire flower. Once the player executes the fire attack action, a fire will be dropped at the target’s location. However, if the player is currently invincible, it will directly remove the target. Besides that, if the target can drop a key, then a key will spawn at that location. Eventually, a string with a suitable description will be printed out to notify the user that a fire attack action has been performed.
With the above design, we are adhering to the Single Responsibility Principle as FireAttackAction class only focusing on what should happen when a fire attack action is performed.

5) Player class

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

The only change that I made between Assignment 2 and Assignment 3 is that I added a new if statement in playTurn such that it will display “Mario can use FIRE ATTACK!” when the player has the capability of FIRE_ATTACK. The rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

6) Enemy class

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

The only change that I made between Assignment 2 and Assignment 3 is that I added a new if statement in allowableAction such that it will add a FireAttackAction instance to the actions list if the when the player has the capability of FIRE_ATTACK (i.e., allow the player to perform fire attack action on the enemy). The rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

7) Sprout class

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

The only change that I made between Assignment 2 and Assignment 3 is when the Sprout instance' age reaches 10, before we change the type of ground at the location to be Sapling, we will have a 50% chance to spawn a fire flower on that location. This could be done by using an if statement. However, the rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

8) Sapling class

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

The only change that I made between Assignment 2 and Assignment 3 is when the Sapling instance's age reaches 10, before we change the type of ground at the location to be Mature, we will have a 50% chance to spawn a fire flower on that location. This could be done by using an if statement. However, the rest of the part remains the same as previous assignments. Thus, please refer to the design rationale for this class in previous assignments for more details.

REQ5: Speaking

Overview

To achieve this feature, there will be seven new classes (i.e., GeneralKoopa, FlyingKoopa, PrincessPeach, Bowser, PiranhaPlant, Monologue and Speakable) created in the extended system, and four existing classes (i.e., Enemy, Toad, Koopa and Goomba) will be modified. The design rationale for each new or modified class is shown on the following pages.

1) Monologue

Why I choose to do it that way:

According to the requirement states in Assignment 3 REQ5, I create a new class called Monologue to store all the monologues of actors. In this class, an arraylist is created to store all the monologues and a getInstance() method is there for us to get the instance of this class. Other than that, the way I make it to get the monologue from that arraylist is using the boundary index. In the getMonologue() method, we input two parameters: monologueIndexLowerBound and monologueIndexUpperBound. The integer currentIndex will be calculated using the boundary index, then the corresponding string will be return by arraylist.get(currentIndex).

```
public String getMonologue(int monologueIndexLowerBound, int monologueIndexUpperBound) {
        if (monologueIndexLowerBound < 0 || monologueIndexUpperBound >= allMonologue.size()){
            throw new IllegalArgumentException("The value of upper bound and lower bound must be a valid index to retrieve string from allM
        }

        int currentIndex = monologueIndexLowerBound +
                rand.nextInt((monologueIndexUpperBound - monologueIndexLowerBound) + 1);
        return allMonologue.get(currentIndex);
    }
```

By doing this in this class, we don’t need to repeat these methods in every actor class who is able to speak, so this obeys DRY(don’t repeat yourself) principle. And if there is any new monologues added, we can add it into the arraylist directly and get it by index without adding monologue to each actor can speak, this make our code more tidy and logic. Besides that, Monologue class has its own responsibility which it performs the behavior of Monologue only, so it obeys SRP(single responsibility principle). In addition, I added exceptions for this method to make sure that the boundary index is valid. If monologueIndexLowerBound is less than 0 or monologueIndexUpperBound is greater than the size of the arraylist, an IllegalArgumentException 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.

2) Speakable

Why I choose to do it that way:

Speakable is an interface which contains two default methods: generateMonologue() and timeToSpeak(). In generateMonologue() method, we input the boundary index, it will return the monologue by get the instance of Monologue class and get the monologue from it by using the boundary index.

```
default String generateMonologue(int monologueIndexLowerBound, int monologueIndexUpperBound){
        return Monologue.getInstance().getMonologue(monologueIndexLowerBound,monologueIndexUpperBound);
    }
```

As metionded in the Assignment REQ, each listed character will talk at every "even" or second turn (alternating). It means, they don't talk all the time. In timeToSpeak() method, we input the currentTurn counter, and divide the turn by 2, if the remainder is 0, that means the current turn is an even turn, which is the time to speak.

```
default boolean timeToSpeak(int currentTurn){
        return currentTurn % 2 == 0;
    }
```

By doing so, we don’t need to repeat these methods in every actor class who is able to speak, so this obeys DRY(don’t repeat yourself) principle. Besides that, Speakable class has its own responsibility which it performs the behavior of Speakable only, so it obeys SRP(single responsibility principle). In addition, we followed Open and Close Principle(OCP) also as when there is more actor that can speak at even turn, we just need to let that actor class to implement speakable interface without modifying speakable interface code.

3) Enemy

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

According to the requirements state in Assignment 3 REQ5, all actors will speak every even turn automatically, so I make Enemy class implements Speakable, which an interface contains two default methods, one for generate monologues and another one is the turn counter. In the playTurn() method of this class, the currentTurn counter will increase by 1 in each turn of the game. Once the turn is even, a random monologue of particular actor will be generate and display. In the constructor of Enemy, I added the monologueIndexLowerBound and monologueIndexUpperBound, so all the class extends Enemy can input the boundary index directly and generate the monologues.

Why I choose to do it that way:

To reduce the repetitive code in those actor's class, I created an interface Speakable and Monologue class to store all the monologues of this game. By implementing Speakable, we can use the default method inside directly without repeat the code in every actor who can speak, also every actor who extends Enemy can use those two methods directly without implement Speakable again, this obeys DRY(don't repeat yourself) principle. Besides that, Enemy class has its own responsibility which it performs the behavior of Enemy only, so it obeys SRP(single responsibility principle).

4) Toad

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

In Assignment 2, I added all the sentence belongs to Toad into the arraylist: toadTalk, and generate the monologue by using getReplyString() to modify which sentence to be talk depends on different capabilities the player had and different index to be used. But In Assignment 3, the design is changed, all of the actors can talk every even turn automatically, so the way we get corresponding monologue is change to use Speakable interface and Monologue class instance.

Why I choose to do it that way:

To reduce the repetitive code in those actor's class, I created an interface Speakable and Monologue class to store all the monologues of this game. I let toad implements Speakable, so it can generate monologues by using the default method stated in Speakable and get the correct monologues by using boundary index. By doing so, we don't have to repeat all the methods in every actor's class, this obeys DRY principle.

5) PrincessPeach

Why I choose to do it that way:

By implementing Speakable, PrincessPeach class is able to use the default method stated in the interface Speakable. The speakable class will generate corresponding monologue by using boundary index. All the monologues are store in an arraylist in Monologue class, so we can get its instance and using the boundary index to retrieve the monologue we want. PrincessPeach class has its own responsibility which it performs the behavior of PrincessPeach only, so it obeys SRP(single responsibility principle).

6) Goomba

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

In Goomba class, I add the boundary index in its constructor, by doing so I can generate the monologues of Goomba directly since it is a class extends Enemy.

Why I choose to do it that way:

By doing so, the monologues of Goomba can be generate directly, if there is any new monologues added, we can modify the boundary index directly. Besides that, Goomba class has its own responsibility which it performs the behavior of Goomba only, so it obeys SRP(single responsibility principle).

7) GeneralKoopa

Why I choose to do it that way:

Because GeneralKoopa is a class that extends Enemy, so in the constructor of GeneralKoopa, the monologueIndexLowerBound and monologueIndexUpperBound is added, so all the class extends GeneralKoopa can input the boundary index directly and generate the monologues.

8) FlyingKoopa

Why I choose to do it that way:

In FlyingKoopa class, I add the boundary index in its constructor, by doing so I can generate the monologues of FlyingKoopa directly since it is a class extends GeneralKoopa. By doing so, the monologues of FlyingKoopa can be generate directly, if there is any new monologues added, we can modify the boundary index directly. Besides that, FlyingKoopa class has its own responsibility which it performs the behavior of FlyingKoopa only, so it obeys SRP(single responsibility principle).

9) PiranhaPlant

Why I choose to do it that way:

In PiranhaPlant class, I add the boundary index in its constructor, by doing so I can generate the monologues of PiranhaPlant directly since it is a class extends Enemy. By doing so, the monologues of PiranhaPlant can be generate directly, if there is any new monologues added, we can modify the boundary index directly. In the playTurn() method, I increase the currentTurn counter by 1 each turn of the game, when there is an even turn, the corresponding monologue of this class will display.
Besides that, PiranhaPlant class has its own responsibility which it performs the behavior of PiranhaPlant only, so it obeys SRP(single responsibility principle).

10) Koopa

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

In Koopa class, I add the boundary index in its constructor, by doing so I can generate the monologues of Koopa directly since it is a class extends GeneralKoopa.

Why I choose to do it that way:

By doing so, the monologues of Koopa can be generate directly, if there is any new monologues added, we can modify the boundary index directly. Besides that, Koopa class has its own responsibility which it performs the behavior of Koopa only, so it obeys SRP(single responsibility principle).

11) Bowser

Why I choose to do it that way:

In Bowser class, I add the boundary index in its constructor, by doing so I can generate the monologues of Bowser directly since it is a class extends Enemy. By doing so, the monologues of Bowser can be generate directly, if there is any new monologues added, we can modify the boundary index directly. In the playTurn() method, I increase the currentTurn counter by 1 each trun of the game, when there is an even turn, the corresponding monologue of this class will display. Besides that, Bowser class has its own responsibility which it performs the behavior of Bowser only, so it obeys SRP(single responsibility principle).

```
WBA FIT2099 - Assignment 3

Team Details:

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


Tasks:

REQ1: Lava Zone

    Responsible person(s): Kuah Jia Chen

    Implementation: DONE
        -  Date to be completed: 15-May-2022
```

```
    - Class Diagram: DONE
    - Interaction Diagram: DONE
    - Design Rationale: DONE
        -  Date to be completed: 21-May-2022

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


REQ2: More allies and enemies!

    Responsible person(s): Guoyueyang Huang

    Implementation: DONE
        -  Date to be completed: 15-May-2022

    - Class Diagram: DONE
    - Interaction Diagram: DONE
    - Design Rationale: DONE
        -  Date to be completed: 21-May-2022

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


REQ3: Magical fountain + Challenge task

    Responsible person(s): Lim Fluoryynx

    Implementation: DONE
        -  Date to be completed: 15-May-2022

    - Class Diagram: DONE
    - Interaction Diagram: DONE
    - Design Rationale: DONE
        -  Date to be completed: 21-May-2022

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


REQ4: Flowers (Structured Mode)

    Responsible person(s): Kuah Jia Chen

    Implementation: DONE
        - Date to complete: 15-May-2022

    - Class Diagram: DONE
    - Interaction Diagram: DONE
    - Design Rationale: DONE
        - Date to be completed: 21-May-2022

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


REQ5: Speaking (Structured Mode)

    Responsible person(s): Guoyueyang Huang

    Implementation: DONE
        - Date to complete: 15-May-2022

    - Class Diagram: DONE
    - Interaction Diagram: DONE
    - Design Rationale: DONE
        - Date to be completed: 21-May-2022

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

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