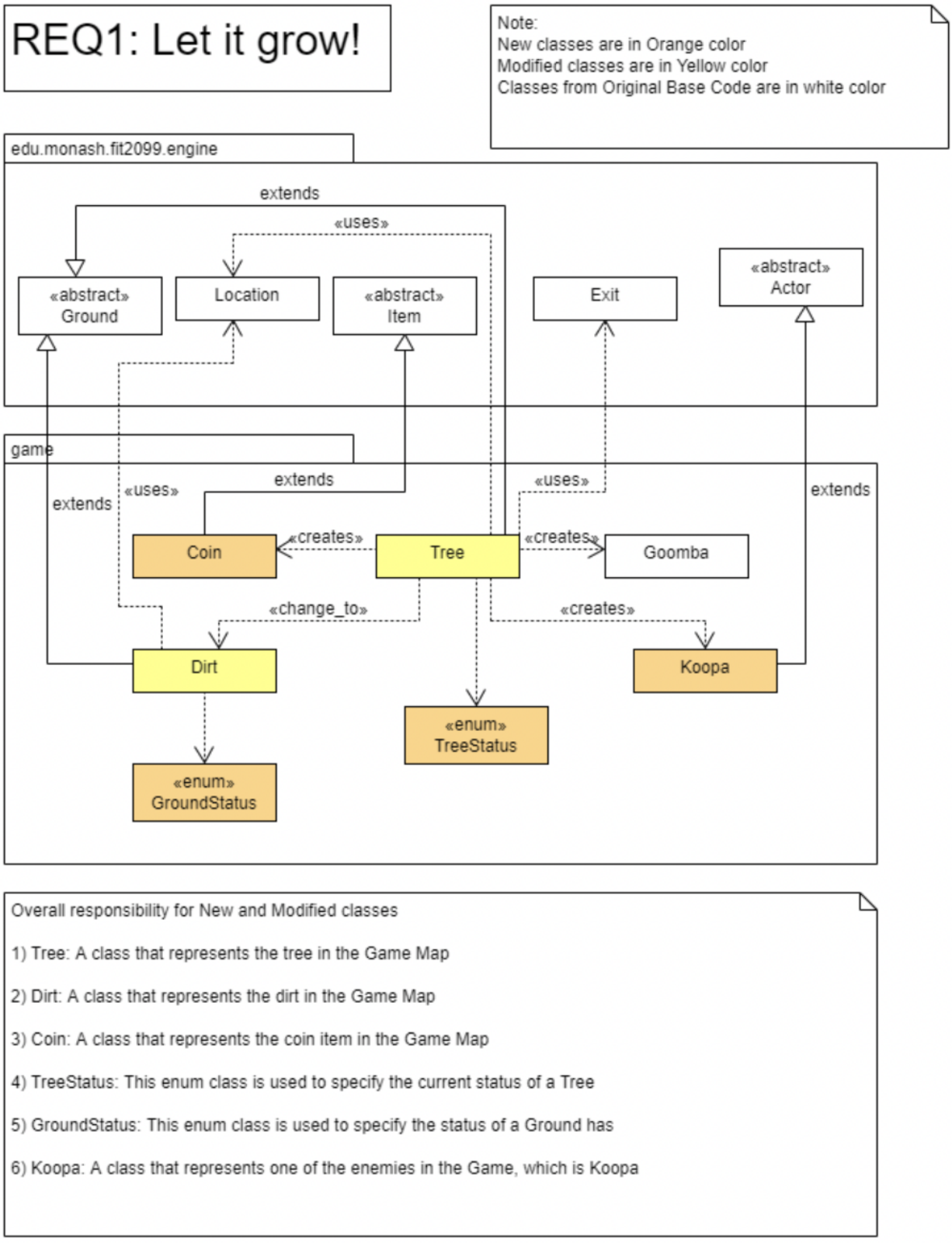
Design Rationale

REQ1: Let it grow!

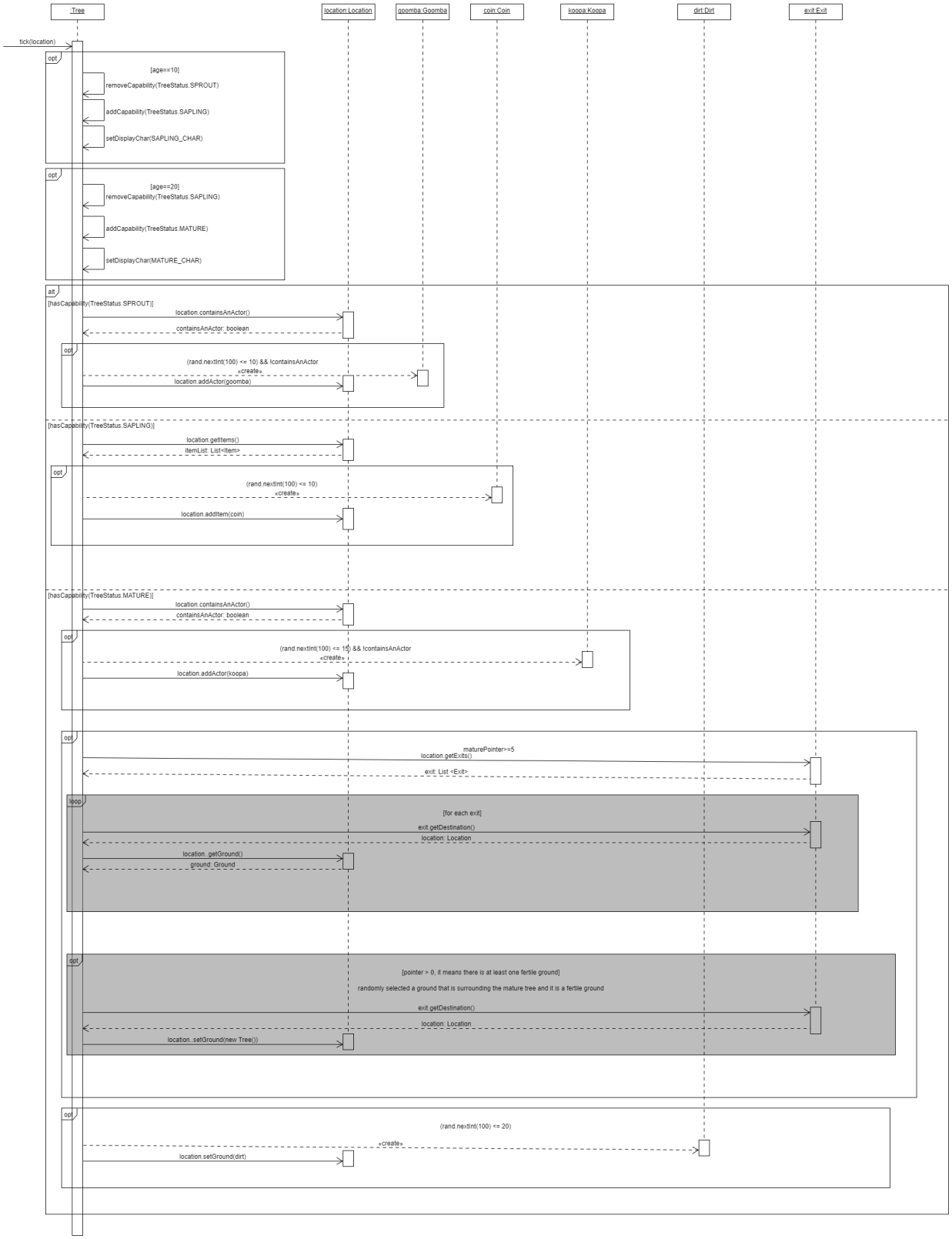
Overview:

To achieve this feature, there will be four new classes (i.e., Koopa, TreeStatus, GroundStatus and Coin) created in the extended system, and two existing classes (i.e.,Tree and Dirt) will be modified. The design rationale for each new or modified class is shown on the following pages. However, the explanation of the design rationale for Tree class will be the last among all six classes. The reason is that the implementation of Tree uses all other classes, hence I would need to explain the design rationale for other classes first so that the marker can understand the implementation of Tree better.



Class diagram for REQ1

REQ1: Let it grow!



Sequence diagram for tick method in Tree class

TreeStatus class

Note: the design rationale for the TreeStatus class in REQ1 will be the same as the one in REQ2 as they both refer to the same class with the same functionality.

TreeStatus is a new enumeration class with three constants that represents the three stages of a Tree, which are SPROUT, SAPLING and MATURE.

Description of constant:

- A Tree instance has a status of SPROUT if its age is below 10.
- A Tree instance has a status of SAPLING if its age reaches 10.
- A Tree instance has a status of MATURE if its age reaches 20.

Why I chose to do it that way:

Since there are three stages for each Tree instance, hence I had decided to create a new enumeration to keep track of the stages for each Tree. By doing so, we are simplifying our code as within the Tree class, we do have to use attributes to keep track of it. This also makes our code more readable. Another reason is that the stages of a Tree are constant, hence there is no point of using any class attributes to keep track of the stages when we can use enumeration class to do so.

Advantages:

The purpose of creating this new enumeration class is that we can avoid excessive use of literal within the Tree class to check the tree's status. This implementation also provides us the flexibility for future development as we can simply add new tree status when there is any in the future. Besides that, we also adhere to the Single Responsibility Principle as this enumeration class is only responsible for storing the three possible stages of a Tree instance.

Disadvantages:

In future, if there are more stages for a Tree instance, it might confuse when debugging.

GroundStatus class

GroundStatus is a new enumeration class with only one constant that represents the status of ground, which is IS_FERTILE.

Description of constant:

- A ground has a status of IS_FERTILE if it is considered as a fertile ground.

Why I chose to do it that way:

According to the assignment specification for this requirement, it says that if a Tree instance is mature, then for every 5 turns, it can grow a new sprout in one of the surrounding fertile grounds and the only fertile ground currently is dirt. However, it means that we need to figure out a way to check if the surrounding ground is a fertile ground, otherwise based on the game logic, we should not spawn the sprout if it is not a fertile ground. Hence, I had decided to create this new class with IS_FERTILE constant. Hence, whenever we initialise a fertile ground, we can add this constant to its capability to indicate that it is a fertile ground. Thus, if we want to grow a sprout on a particular ground, we can determine if it is a fertile ground by checking whether it has the capability IS_FERTILE.

Initially I was thinking of creating an interface (e.g., FertileGround) and using the static factory method (e.g., FertileManager) to store all the fertile ground in the array list. However, I realized that this would overcomplicate our code when we need to randomly grow the sprout and increase the time complexity as we need to search through the array list to find a specific fertile ground instance. Therefore, using this enumeration class would definitely simplify my code and make it more efficient.

Advantages:

In this case, we are using constant to specify that a Ground is a fertile ground. By doing so, we can avoid excessive use of literals as we do not have to create any local attributes within Dirt class for example to indicate that all Dirt instances are considered as fertile ground. This implementation also provides us the flexibility for future development. For instance, if there are more types of fertile ground that extends Ground class introduced to the game, we could simply add this constant as the capability of it. We also adhere to the Single Responsibility Principle as this enumeration class is only responsible to store the possible statuses that a Ground can have.

Disadvantages:

N/A

Koopa class

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

Coin class

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

5) Dirt class

Dirt is a class that represents the dirt in the Game Map. It is a class that extends from the Ground. In this class, I had only modified the constructor of the Dirt class.

Description:

- In the constructor of the Dirt class, it will call the parent's constructor by inputting it's display character. After that, it will add the capability of IS_FERTILE to itself to indicate that Dirt is a fertile ground.

Why I chose to do it that way:

Based on the specification for this REQ, only fertile ground is possible to grow sprout. Therefore, by adding this capability, it will be very useful for us to determine whether a ground is a fertile ground and perform specific tasks to it.

Advantages:

With this design, we are adhering to the Single Responsibility Principle as the method within Dirt class only shows the properties of a Dirt and what an actor can do to the Dirt. Moreover, we also fulfil the Liskov Substitution Principle as all the methods in Dirt class still preserve the meaning from its parent class, Ground.

Disadvantages:

N/A

Tree class

Note: the design rationale for Tree class in REQ1 will be the almost same as the one in REQ2 as they both referring to the same class, but the design rationale will focus on different aspect (i.e., REQ1 focus on what will happen when a Tree is in a certain stage, whereas REQ2 is focus on Player can jump to a Tree), although there will still be some repetitive content.

Tree is a class that represents the tree in the Game Map. It is a class that extends from the Ground. According to the assignment specification, a Tree has three stages, which are sprout, sapling and mature, hence there will be a significant amount of class attributes in this class.

private int age = 0;

private static final char *SPROUT_CHAR* = '+';

private static final char *SAPLING_CHAR* = 't';

private static final char *MATURE_CHAR* = 'T';

The meaning of the constants above are known by looking at the name of it.

In addition, Tree class has overridden 1 method which are tick and create two new methods, which are isSapling and isMature.

Description of methods:

- isSapling will return True if age reaches 10.
- isMature will return True if age reaches 20.
- tick method is called once per turn. For each turn, it will increment the age attribute by 1. After that, it will check if the age of this Tree instance reaches 10, if yes, it will remove the capability of SPROUT (these constants were introduced in TreeStatus class) and add the capability of SAPLING to indicate that it is now a sapling tree. Similarly, if the age of this Tree instance reaches 20, then it will remove the capability of SAPLING and add the capability of MATURE to indicate that it is now a mature tree. After updating the status of this Tree instance according to its age, it

will now check the current status of this Tree and perform different tasks. If this Tree has a status of SPROUT, then it might have a 10% chance to create a Goomba object on that location only if there is no actor standing on that Tree in every turn. If this Tree has a status of SAPLING, it will have a 10% chance to create a Coin ($20) instance on that location at every turn. If the tree has a status of MATURE, then it will have 15% chance to create a Koopa object on that location only if there is no actor standing on that Tree in every turn, have 20% to wither and die in every turn (i.e., become Dirt) and grow s new sprout in one of the surrounding fertile grounds randomly for every 5 turns. If there is no available fertile ground, then it will stop spawning sprouts. In order to get the surrounding fertile ground, we would need to use the exits of that particular location. So, for each exit, we will get that ground instance and check if it has the capability of IS_FERTILE. After that, it will grow a sprout among all available fertile ground.

Why I chose to do it that way:

As you can see from above, we know that for each status of a Tree instance, we will need to perform different tasks, hence by using the constant in TreeStatus, it will help us to keep track of the status of trees efficiently. In previous pages, I had mentioned that the IS_FERTILE will help us to determine whether a ground is a fertile ground and hence this constant will be used when we need to randomly grow a sprout on its surrounding ground when the tree is mature. With this constant, we do not need to use “instanceOf” to check what type of ground it is. This is actually a good design practice as when more and more types of fertile ground are introduced to the game, we do not need to use many if-else statements in our implementation.

Advantages:

With the above design, we are following the Open Closed Principle as when more types of fertile ground are introduced, we could just determine whether a ground is a fertile ground by looking at its capability instead of using “instanceOf”, hence we can extend our game with modifying any existing code. We also adhere to the Single Responsibility Principle as all the methods and attributes above tell the properties of a Tree without modifying any code in its parent class. Besides that, we also create several class attributes to store the fixed value. Hence, we are adhering to the “avoid excessive use of literals” principle.

Disadvantages:

Since all the requirements and logic need to be performed within the tick method, the method will be too long and hence less readable and maintainable. It might cause confusion to debug when an error occurs.

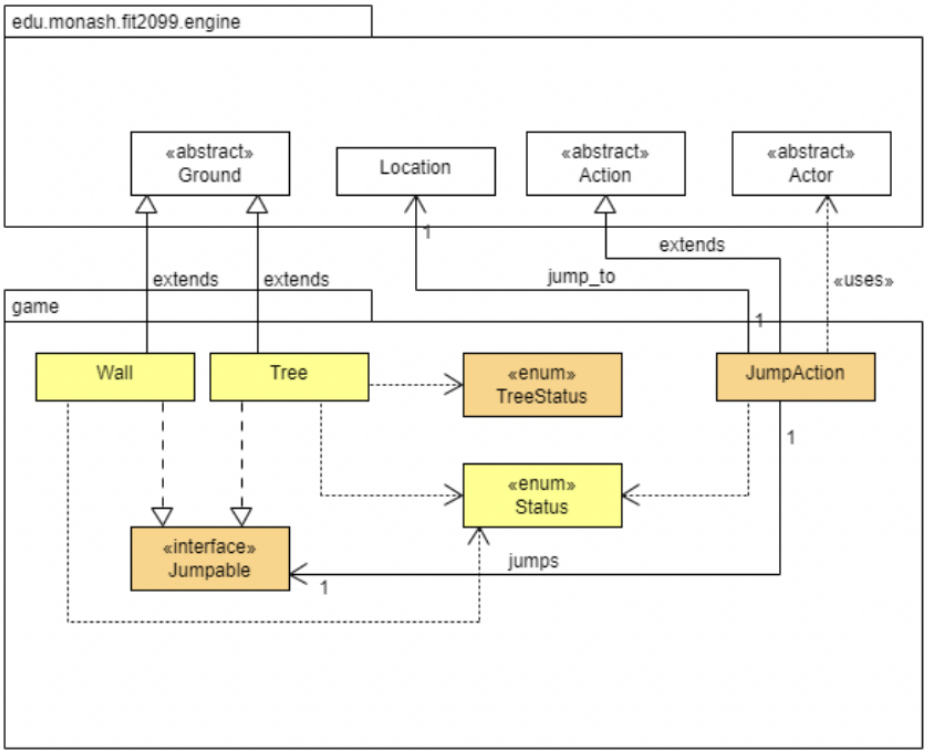
REQ2: Jump Up, Super Star!

Overview:

To achieve this feature, there will be three new classes (i.e., JumpAction, Jumpable, and TreeStatus) created in the extended system, and three existing classes (i.e., Wall, Tree, and Status) will be modified. The design rationale for each new or modified class is shown on the following pages. However, the explanation of the design rationale for JumpAction class will be the last among all six classes. The reason is that the implementation of JumpAction uses all other classes, hence I would need to explain the design rationale for other classes first so that the marker can understand the implementation of JumpAction better.

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: A class that represents the tree in the Game Map

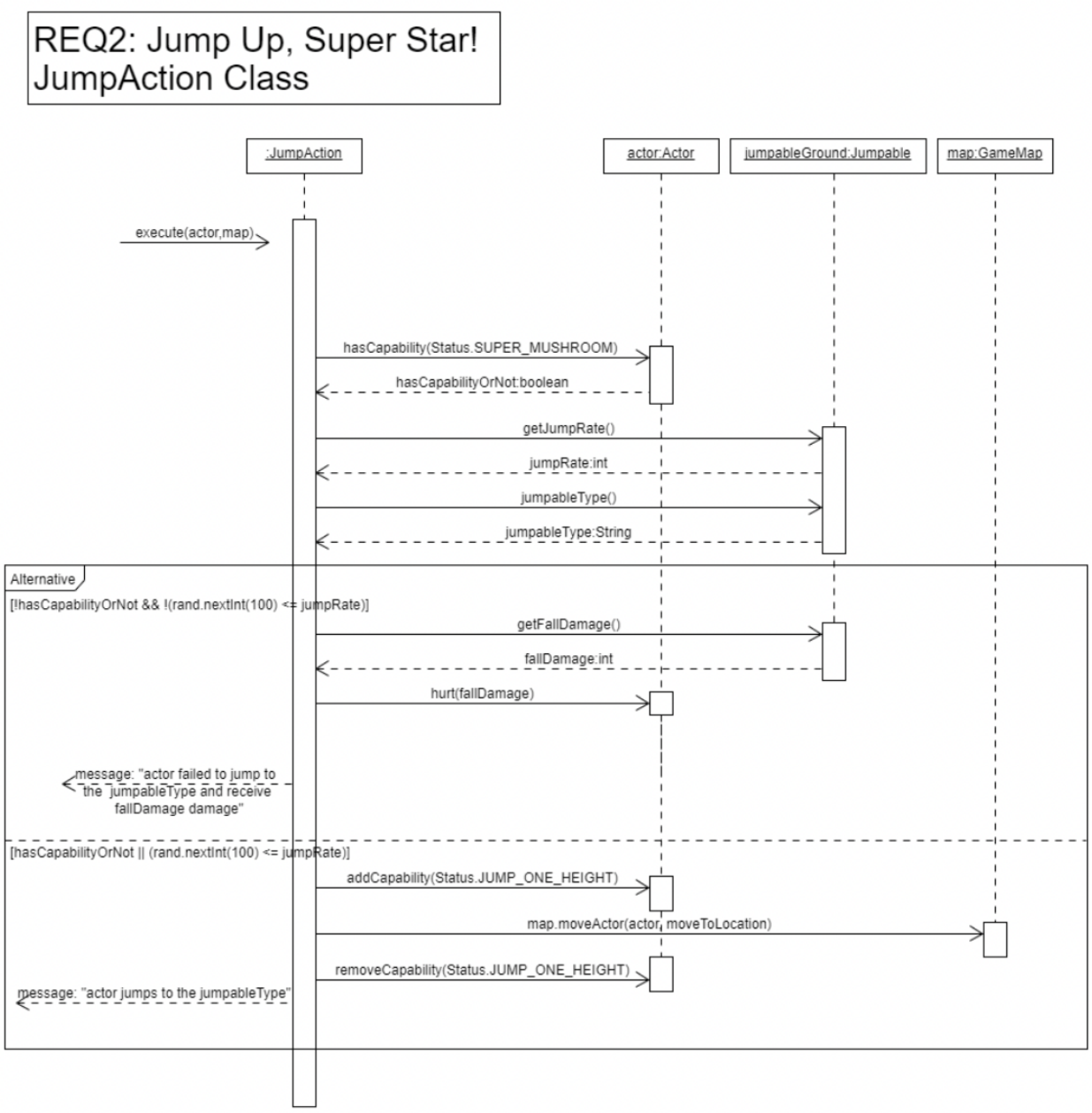
3) Jumpable: Interface for classes that are jumpable by the Player

4) TreeStatus: This enum class is specific to the current status of a Tree

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

Class diagram for REQ2



Sequence diagram for execute method in JumpAction class

Jumpable class

Jumpable is an interface for classes that are jumpable by the Player. It is a new interface class that contains three methods, which are getJumpRate, getJumpableType and getFallDamage.

Description of methods:

- getJumpRate will return an integer that indicates the success rate for an actor to jump to that ground.
- getJumpableType will return a string that indicates the type of that ground.
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that ground.

Why I chose to do it that way:

Currently based on the requirement of this feature, the Wall class and Tree class will implement this interface. This indicates that Wall and Tree on the map are jumpable for the Player. The reason for us to create an interface instead of an abstract class is that multiple inheritances are not allowed as Wall and Tree already extends from the Ground. 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 implements Jumpable.

Advantages:

By doing so, we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the three methods above are just a query. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both.

Moreover, our implementation also fulfils the principle of Dependency Inversion Principle. It is because instead of having JumpAction class to have an attribute of each jumpable ground (i.e., Wall or Tree) (we will discuss this more in JumpAction section), we can have an attribute of type Jumpable. By doing so, the concrete class JumpAction will not directly depends on the Wall and Tree class.

Disadvantages:

N/A

Status class

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. It is an existing enumeration

class with some additional modification. In this class, I had added two statuses, which are JUMP_ONE_HEIGHT and SUPER_MUSHROOM.

Description of constant:

- JUMP_ONE_HEIGHT indicates that the Player have the capability to jump to wall or tree. However, according to the assignment specification, it says that the height and depth might change in future, hence I had decided to name this status in a more specific way to avoid confusion in future.
- SUPER_MUSHROOM indicates that the Player had consumed a super mushroom.

Why I chose to do it that way:

Instead of creating class attributes or literals to indicate whether a particular actor has a capability to jump or consume super mushrooms, it is better to make use of the enumeration class to make our code easier to read. Besides that, I also can utilise the engine code provided to check the actor's capability to simplify our code.

Advantages:

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.

Disadvantages:

As the game develops, more and more capability will be added to this status class. If there are too many statuses within this class, it might cause confusion when debugging.

Wall class

Wall is a class that represents the wall in the Game Map. It is a class that extends from the Ground and implements Jumpable. There are three class attributes in the Wall class, which are JUMP_RATE, FALL_DAMAGE, and JUMP_TYPE.

Description of class attributes:

- JUMP_RATE is a private static final integer class attribute with a value of 80 that indicates the success rate to jump to a wall is 80%.
- FALL_DAMAGE is a private static final integer class attribute with a value of 20 that indicates the fall damage after a failed jump is 20.
- JUMP_TYPE is a private static final string class attribute that is assigned to “Wall” to indicate the type of this jumpable ground is a wall.

The purpose of creating these class attributes is to adhere the “avoid excessive use of literals” principle. Hence, I had declared these attributes as constant.

In addition, Wall class has overrides 6 methods, which are canActorEnter, blockThrownObjects, getJumpRate, getJumpableType, getFallDamage and allowableActions.

Description of methods:

- canActorEnter will return true if and only if the actor has the capability of JUMP_ONE_HEIGHT
- blockThrownObjects will always return true to indicates that wall can block thrown objects in the game
- getJumpRate will return an integer that indicates the success rate for an actor to jump to that wall (i.e., JUMP_RATE).
- getJumpableType will return a string that indicates the type of ground (i.e., FALL_DAMAGE).
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that wall (i.e., JUMP_TYPE).
- allowableActions will return a JumpAction instance if the actor is currently not on that wall, else return an empty ActionList. By doing so, we can make sure the actor will not be able to jump to that wall if the actor is currently already on that wall.

Why I chose to do it that way:

Not all types of ground in this game are jumpable, hence I had decided to create an interface class specifically for those grounds that are jumpable. Hence, in this case I had made the Wall class to implement the Jumpable class. By doing so, we do not have to do any modification to the code within Ground class and this kind of implementation does provide the flexibility for us to extend our game easily. (i.e., we may create more

types of ground that are jumpable for actors in future) By implementing Jumpable class, we also make sure Wall class will implement the three methods that are mandatory for a jumpable ground.

Advantages:

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., Ground). 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 Jumpable interface.

Disadvantages:

This kind of implementation does not provide us the flexibility to extend our Wall class. What I meant is that in future, a Wall instance might have several statuses (e.g., broken) and lead to different fall damage and success rate of a jump. When this happens, we will have many if-else statements for the getters of these attributes. This might be a problem as the number of statuses keep growing and are hard to maintain and debug.

TreeStatus class

Note: the design rationale for the TreeStatus class in REQ2 will be the same as the one in REQ1 as they both refer to the same class with the same functionality.

TreeStatus is a new enumeration class with three constants that represents the three stages of a Tree, which are SPROUT, SAPLING and MATURE.

Description of constant:

- A Tree instance has a status of SPROUT if its age is below 10.
- A Tree instance has a status of SAPLING if its age reaches 10.
- A Tree instance has a status of MATURE if its age reaches 20.

Why I chose to do it that way:

Since there are three stages for each Tree instance, hence I had decided to create a new enumeration to keep track of the stages for each Tree. By doing so, we are simplifying our code as within the Tree class, we do have to use attributes to keep track of it. This also makes our code more readable. Another reason is that the stages of a Tree are constant, hence there is no point of using any class attributes to keep track of the stages when we can use enumeration class to do so.

Advantages:

The purpose of creating this new enumeration class is that we can avoid excessive use of literal within the Tree class to check the tree's status. This implementation also provides us the flexibility for future development as we can simply add new tree status when there is any in the future.

Disadvantages:

Similar to the Status class, if a Tree will have many statuses in the future, it might confuse when debugging.

Tree class

Note: the design rationale for Tree class in REQ1 will be the almost same as the one in REQ2 as they both referring to the same class, but the design rationale will focus on different aspect (i.e., REQ1 focus on what will happen when a Tree is in a certain stage, whereas REQ2 is focus on Player can jump to a Tree), although there will still be some repetitive content.

Tree is a class that represents the tree in the Game Map. It is a class that extends from the Ground and implements Jumpable. According to the assignment specification, a Tree has three stages and the success rate and fall damage is different for each stage, hence there will be a significant amount of class attributes in this class.

All the class attributes in Tree class:

private int age = 0;

private String type;

private static final char *SPROUT_CHAR* = '+';

private static final char *SAPLING_CHAR* = 't';

private static final char *MATURE_CHAR* = 'T';

private static final int *SPROUT_JUMP_RATE* = 90;

private static final int *SAPLING_JUMP_RATE* = 80;

private static final int *MATURE_JUMP_RATE* = 70;

private static final int *SPROUT_FALL_DAMAGE* = 10;

private static final int *SAPLING_FALL_DAMAGE* = 20;

private static final int *MATURE_FALL_DAMAGE* = 30;

private static final String *SPROUT_TYPE* = "Sprout";

private static final String *SAPLING_TYPE* = "Sapling";

private static final String *MATURE_TYPE* = "Mature";

The meaning of the constants above are known by looking at the name of it.

In addition, Tree class has overridden 6 methods which are canActorEnter, tick, getJumpRate, getJumpableType, getFallDamage and allowableActions and create two new methods, which are isSapling and isMature.

Description of methods:

- canActorEnter will return true if and only if the actor has the capability of JUMP_ONE_HEIGHT
- tick method will increment the age of Tree instance by 1 after each turn. Besides that, it will also check if the Tree instance reaches the age of 10 or 20 by using the methods isSapling and isMature. If yes, it will set the tree status, display character and type accordingly.
- getJumpRate will return an integer that indicates the success rate for an actor to jump to that tree according to the tree stages.
- getJumpableType will return a string that indicates the type of ground according to the tree stages.
- getFallDamage will return an integer that indicates the damage the actor will receive after a failed jump to that tree according to the tree stages.

- allowableActions will return a JumpAction instance if the actor is currently not on that tree, else return an empty ActionList. By doing so, we can make sure the actor will not be able to jump to this wall if the actor is currently already on that wall.
- isSapling will return true if the age of the Tree instance reached 10
- isMature will return true if the age of the Tree instance reached 20

Why I chose to do it that way:

In the Tree class, I had created quite a number of class attributes. The purpose of creating these class attributes is to adhere to the “avoid excessive use of literals” principle, so that I can use these attributes instead of literals within the methods and hence make our code more readable. Besides that, since Tree is jumpable for actors, hence I let the Tree class implement Jumpable. By doing so, we can achieve our objective without modifying any existing code in the Ground as not all types of ground are jumpable. This makes our class structure more organised and understandable.

Advantages:

Like Wall class, 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 Tree without modifying the implementation of its parent class (i.e., Ground). 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 Jumpable interface.

Disadvantages:

Since the fall damage, success rate of jump and display character for each stage of tree is different, we will need a lot of class attributes as you have seen above. If there is a new stage introduced, we will need even more attributes. This makes our code less maintainable and cause confusion. In addition, we will have many if-else statements for the getters of these attributes. This might be a problem as the number of statuses keep growing and are hard to maintain and debug.

JumpAction class

JumpAction is a class that allows the actor to jump to the jumpable ground. It is a new class that extends Action class. In this class, there are four class attributes, which are moveToLocation, direction, rand and jumpableGround.

Description of class attributes:

- moveToLocation is a protected attribute with a type of Location. It indicates the destination of the actor.
- Direction is a protected attribute with a type of string. It indicates the direction of the actor's destination.
- rand is a private constant attribute with a type of Random, it will be used in the execute method.
- jumpableGround is a private attribute with a type of Jumpable.

The JumpAction class overrides the execute and menuDescription method from its parent class.

Description of methods:

- For the execute method, there will be an if-else statement. If the actor does not consume super mushroom (i.e., check the actor's capability) and the random number generated is lower than the success rate of that jumpable ground, the piece of code within the if statement will execute. Within the if statement, the actor will receive the fall damage (i.e., check the fall damage for that jumpable ground using getFallDamage method), and a message that indicates the actor has failed to jump to that jumpable ground will be printed out. If the piece of code within the else statement has executed (i.e., the actor has successfully jumped to the jumpable ground), it will add the JUMP_ONE_HEIGHT capability to the actor. By doing so, we can ensure that the actor can jump to that jumpable ground as the canActorEnter will return true for that jumpable ground. After that, it will move the actor to that location using the moveActor method. In addition, it will remove the capability of JUMP_ONE_HEIGHT from that actor so that we can ensure that the actor will need to "jump" again if the actor wants to jump to any jumpable ground. Eventually, a message that indicates that the actor has successfully jumped to the jumpable ground will be printed out.
- menuDescription will return a string that will appear on the command list in the console. This string included the type of the jumpable ground, the coordinate of that

jumpable ground and the direction.

Why I chose to do it that way:

Since JumpAction is an action and we need the functions in the Action class, and thus we let JumpAction class extend Action class. However, since we need to ensure that when JumpAction is called, the actor will have a successful/failed jump, hence we will need to override the execute method to ensure our game logic works properly. The description of the execute method is mentioned above. Besides that, I had also override the menuDescription method, the reason to do so is to let the user have a better understanding on what will happen when he/she lets the actor perform a JumpAction. In addition, in order to let the execute method know what the fall damage is, type of ground and success rate of a jump for that particular jumpable ground, I had included that particular jumpable ground to be one of the input parameters for the constructor. By doing so, we can get all the necessary information that we need and use it in the execute and menuDescription method. Thus, this simplifies our code.

Advantages:

With the design above, we are adhering to the Single Responsibility Principle as the JumpAction class only focuses on the action that will be executed when the actor jumps. Besides that, we also follow the Open Closed Principle. The reason is because based on our current implementation, when there is more jumpable ground created in future, we do not have to modify any code in JumpAction. In addition, we also fulfil the Liskov Substitution Principle as JumpAction preserves the meaning of execute and menuDescription method behaviours from Action class.

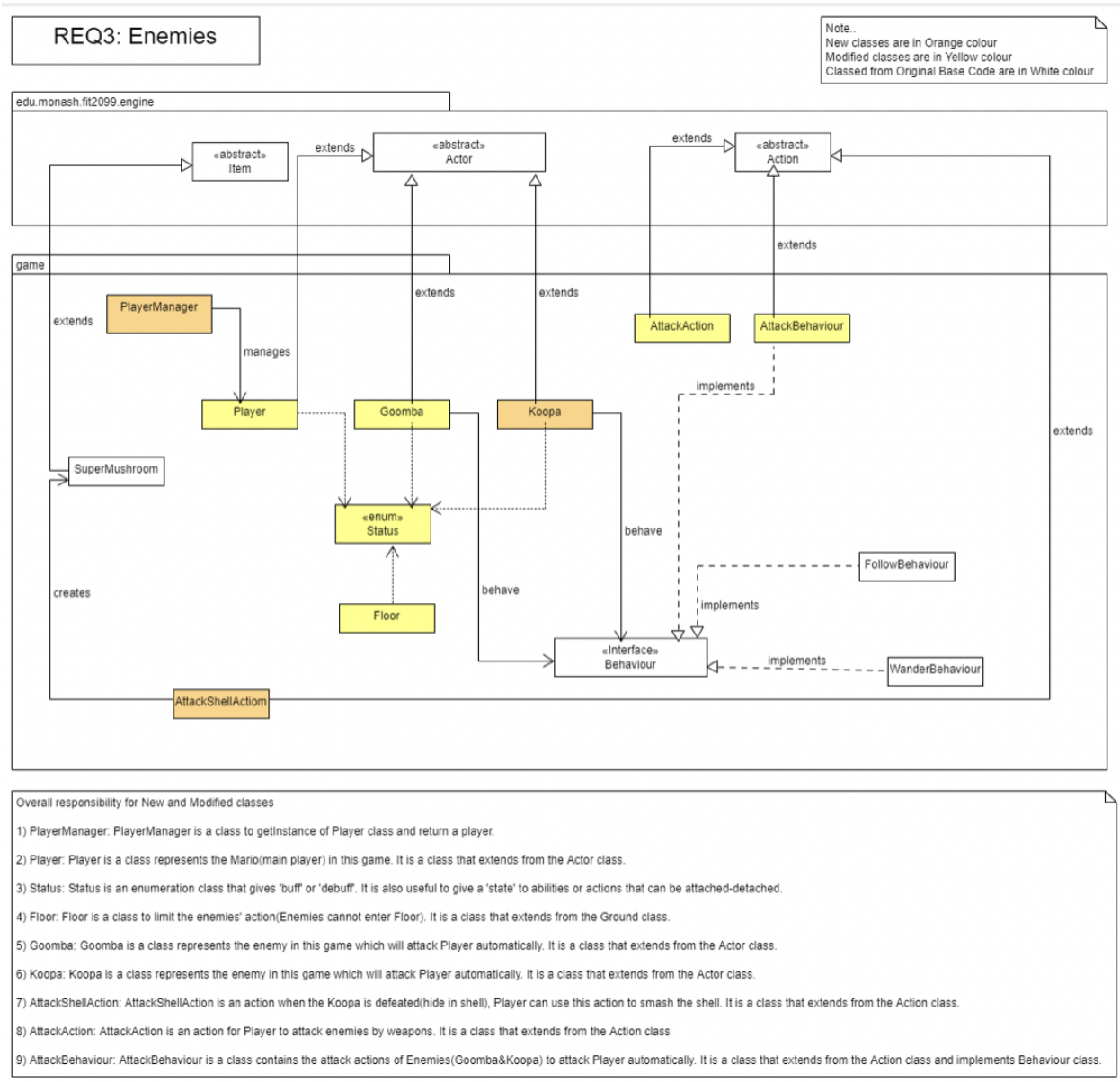
Disadvantages:

N/A

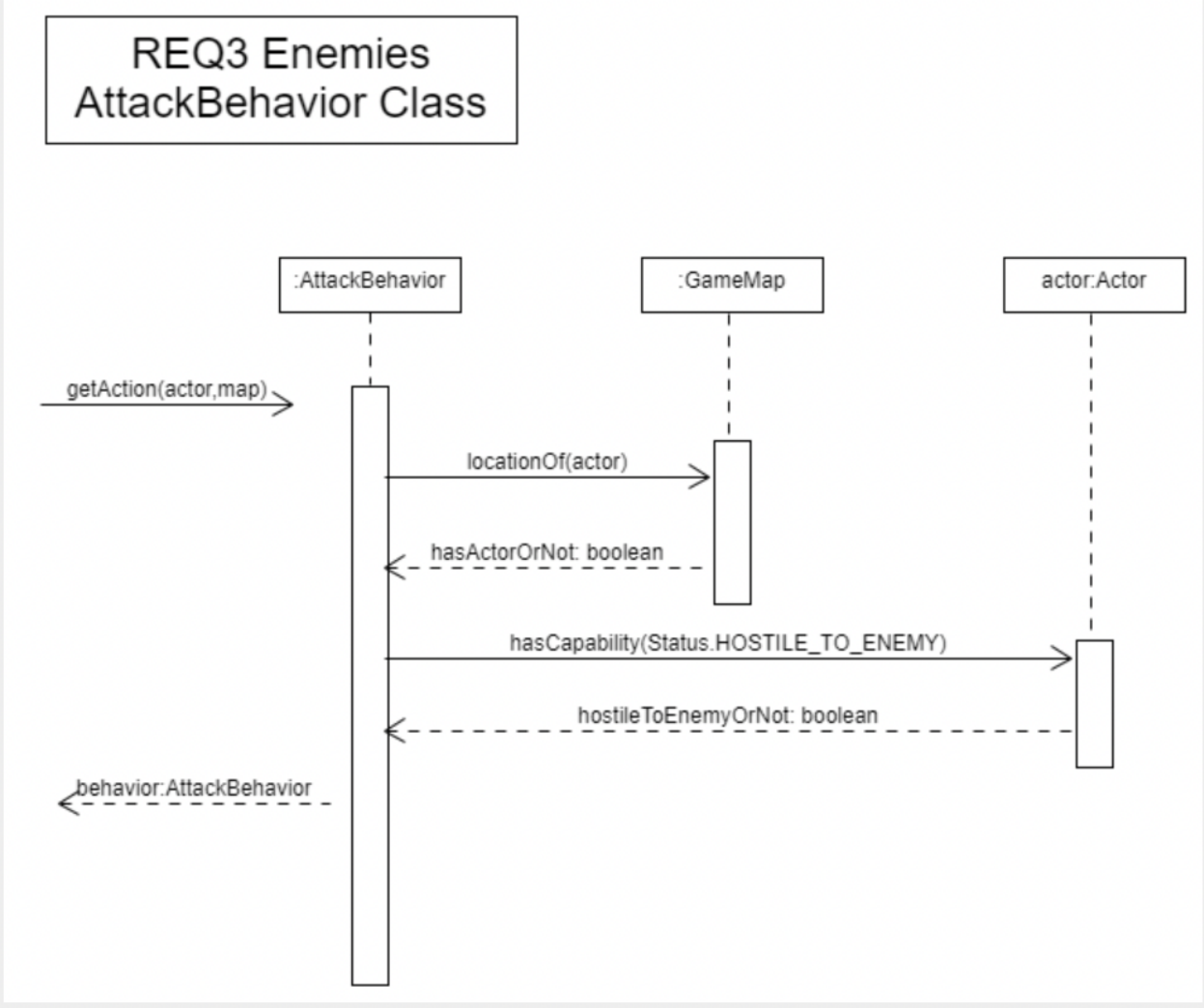
REQ3: Enemies

Overview:

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



Class diagram for REQ3



Sequence diagram for getAction method in AttackBehaviour class

1) Player class

Player class is a class that represents the main player(Mario) in this game. It is a class that extends from the Actor class. According to the assignment requirements, player can move around on the game map, speak to Toad(a friendly NPC) and fight with enemies with weapons.

For this class, I modified it by adding PlayerManager in its constructor to append a player in PlayerManager’s instance, to make sure we can access the actor player easily in each round of the game. Because we need to point out which actor is to be attacked/followed in the enemies' behavior, it’s hard for us if we use Player class only, so we add PlayerManager to get a player for us to determine which player is going to be the target.

2) PlayerManager class

PlayerManager is a class used to return an actor player in the game for us easy to use it in the enemies' behavior, so the enemies know who to attack and who to follow. In this class I create a player instance, by appending the player into its instance, we can get the player whenever we need it by PlayerManager.getInstance().returnPlayer().

Why I choose to do it that way:

By adding PlayerManager to get a player for us to determine which player is going to be the target because we need to point out which actor is to be attacked/followed in the enemies behavior, we couldn't do it with just the Player class.

Advantage:

By using this class, it does not require creating a new object each time they're invoked. That will be nice for us since we don’t need to create a player each time when we need to attack/follow. It makes our code clear and logical.

Disadvantage:

N/A

3) Goomba class

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

In addition, enemies should have some behaviors: attackbehavior, followbehavior and wanderbehavior. By adding all the behaviors into the behavior hashmap, we can make sure that the goomba will attack the player automatically when a player is in the enemy's surroundings. When goomba is no longer able to attack/follow the player(i.e player enters the floor which goomba is not allowed to), goomba will be wandering around again.

Goomba class has two methods which are overridden from its parent class Actor: allowableActions and playTurn.

- allowableActions() is a method used to make goomba can be attacked by Player.
- playTurn() is a method used to figure out what to do next.

Why I choose to do it that way:

For goomba class, we involve an if loop in the method. Because goomba will have a 10% chance to suicide each round of this game, so by adding the if loop, a random number which less than 10 will be picked from 100 numbers, means a 10% chance the goomba will be removed from the map.

Advantage:

By doing this, the game map will not be overcrowded and our game map will be clean.

Disadvantage:

It will be a possibility that all the goomba been removed from the map, so at that time the game map will not have enemies anymore.

4) Koopa class

Koopa class is a class that represents the enemies in this game. It is a class that extends from the Actor class. According to the assignment requirements, Koopa can move around in the game map but cannot enter the floor. Once Koopa is engaged in a fight (the Player attacks the enemy or the enemy attacks player -- when the player stands in the enemy's surroundings), it will follow the Player. It causes 30 damages to player with 50% hit rate and Koopa has the same behaviors as goomba.

But when Koopa died, it will not be removed from the map directly. When Koopa is not conscious(which means it is defeated), it will hide inside its shell, and its character will change from ’K’ to ‘D’. Player cannot attack it anymore, and all the behaviors will be removed from Koopa(attack/follow/wander).

Goomba class has two methods which are overridden from its parent class Actor: allowableActions and playTurn.

- allowableActions() is a method used to make Koopa can be attacked by Player. In this method, we have an if & else if loop inside. Not the same as goomba, when Koopa is not conscious(which means it is defeated), it will hide inside its shell, at this time we cannot attack it with a normal weapon anymore. When it hides

in its shell, only the wrench can break it. So when the Koopa is still conscious(not defeated), we can attack it as usual until it is defeated and hide inside a shell(‘K’ -> ‘D’), at this time we need to use a wrench to smash its shell.

- playTurn() is a method used to figure out what to do next. In this method, we add an if loop inside to detect whether Koopa is defeated. In each round of the game, if Koopa is defeated, the character of it will change to ‘D’, and all the behaviors of Koopa will be removed by removing the key in hashmap behaviors.

Why I choose to do it that way:

For the Koopa class, I created a class attribute isDefeated and initialized it as false at the beginning. When the Koopa is not conscious, I assigned true to the isDefeated, which means Koopa hides in its shell already, so we can easily define the condition after Koopa is defeated.

Advantage:

By doing this, once the Koopa is defeated, we can detect it, change its character and remove all the behaviors of it which is responsive and efficient.

Disadvantage:

N/A

5) Status class

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. It is an existing enumeration class with some additional modifications. In this class, I added two

statuses, which are IM_GOOMBA and IM_KOOPA.

Description of constant:

- IM_GOOMBA indicates that the Actor goomba. By adding this capability to Goomba, we are able to add specific attack behavior to it.
- IM_KOOPA indicates that the Actor koopa. By adding this capability to Koopa, we are able to add specific attack behavior to it.

Why I choose to do it that way:

Instead of creating class attributes to indicate whether this actor is goomba or Koopa, it is better to make use of the enumeration class to make our code easier to read. Besides that, I also can utilize the engine code provided to check the actor's capability to simplify our code.

Advantage:

With this design, we will not use any more literal to specify whether this actor is goomba or Koopa 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.

Disadvantage:

As the game develops, more and more capabilities will be added to this status class. If there are too many statuses within this class, it might cause confusion when debugging.

6) Floor class

Floor class is the class used to represent the floor inside a building. It extends from Ground class. In this class, we override canActorEnter() method from its parent class. By modified this method to return a boolean, we add an if - else if loop inside. For the actor has the capability HOSTILE_TO_ENEMY(means player), he can enter the floor. For the enemies who have the capability IM_GOOMBA or IM_KOOPA, they cannot enter the floor which makes sure that the player has somewhere safe to stay.

7) AttackShellAction class

AttackShellAction is a class which used to smash Koopa's shell. It extends from its parent class Action. When the Koopa is defeated and the player has the capability HAVE_WRENCH, the player can use only this action to smash Koopa's shell in order to get a super mushroom.

AttackShellAction class has two methods which are overridden from its parent class Action: execute() and menuDescription.

- execute() is the method to perform the action (smash the shell). This method get the current location of the target first, then remove it from the map. After that

create a super mushroom there and return a string: "Koopa is gone, pick up the Super Mushroom!"

- menuDescription is the method that returns a descriptive string in the menu, which gives the player a choice to smash the shell.

Why I choose to do it that way:

Because when the Koopa is defeated, player cannot attack it as usual. Only when player is standing beside it and with a wrench, so I create an attackshell action for attacking the shell only to make sure that our code is easy to understand and clear.

Advantage:

With this design, we will not use any more literal to specify whether this actor is goomba or Koopa 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 the implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

Disadvantage:

As the game develops, more and more capabilities will be added to this status class. If there are too many statuses within this class, it might cause confusion when debugging.

8) AttackAction class

AttackAction is a class used by player to attack the enemies, it extends from its parent class Action class. In this class, player can attack enemies with weapons with a corresponding hit rate. If the target(enemies) is no longer conscious, all the item of that target will drop and it will be removed from the map(except Koopa). I modified this class by adding a new if loop in it, if the target is not Koopa, then remove it from the map. In the end, there is a string will be printed to show how much you hurt the enemies.

9) AttackBehaviour class

AttackBehavior is the class that developed and use to attack the player automatically. This class extends from its parent class Action and implements Behavior class. By adding this behavior into hashmap behavior, the enemy will attack the player automatically if player is beside it.

Why I choose to do it that way:

Instead of creating two attack methods/behavior separately for Koopa and Goomba, we put two actions(kick and punch) together in one AttackBehavior class. By doing so, there will be less repeating code.

AttackBehavior class has three methods which is override from its parent class Actoion: execute(), getAction() and menuDescription().

- execute() is the method to perform the action(kick or punch). In this method an if - else if loop will be use to determine whether the actor is Goomba(kick) or Koopa(punch).
- getAction() is the method used to attack the player automatically. When the player is in the enemies' surroundings, will return this attackBehavior to attack player.
- menuDescription() is the method that returns a descriptive string in the menu, but for this class, enemies will attack the player automatically, so an empty string will be returned.

Advantage:

With this design, we will not use any more classes to specify whether this actor is goomba or Koopa as we can make use of these capabilities. 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 actions when needed.

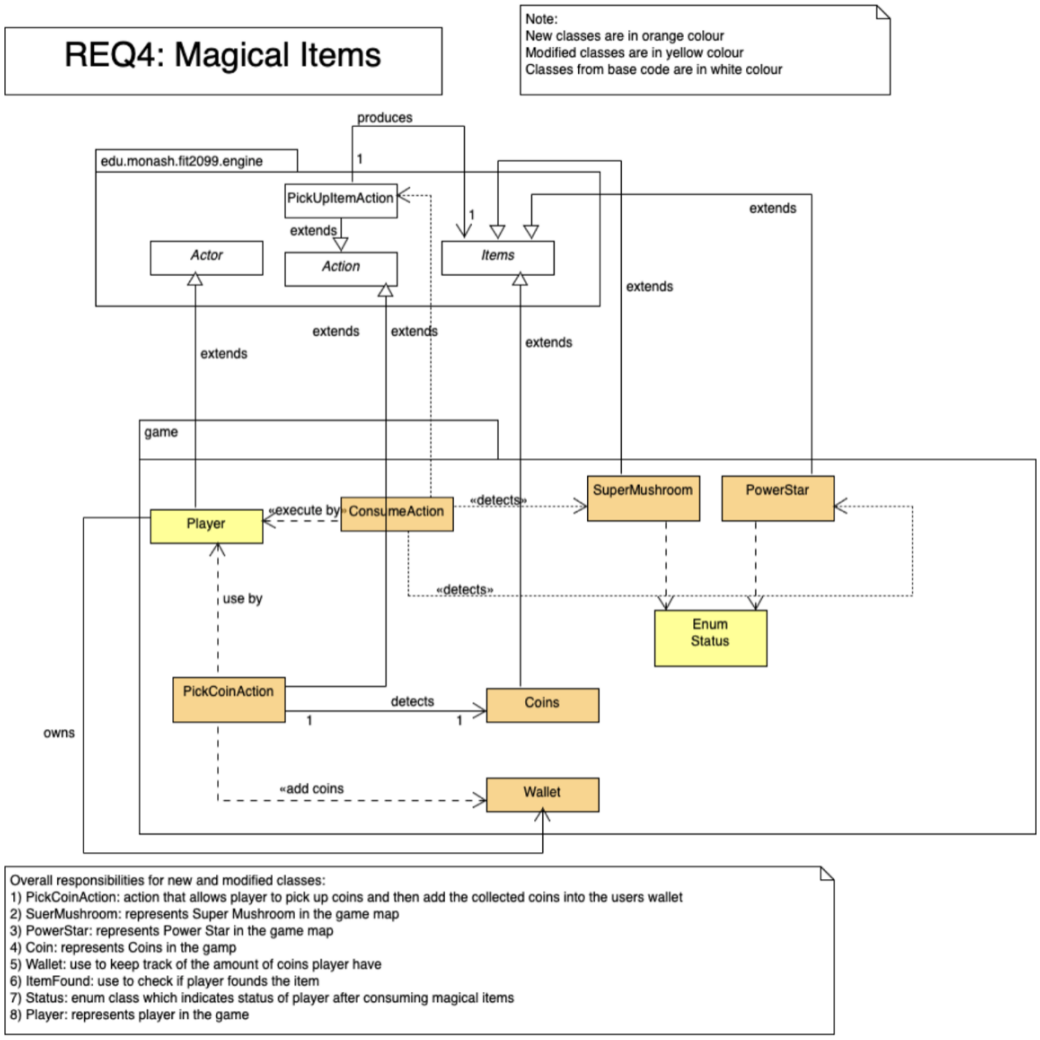
Disadvantage:

N/A

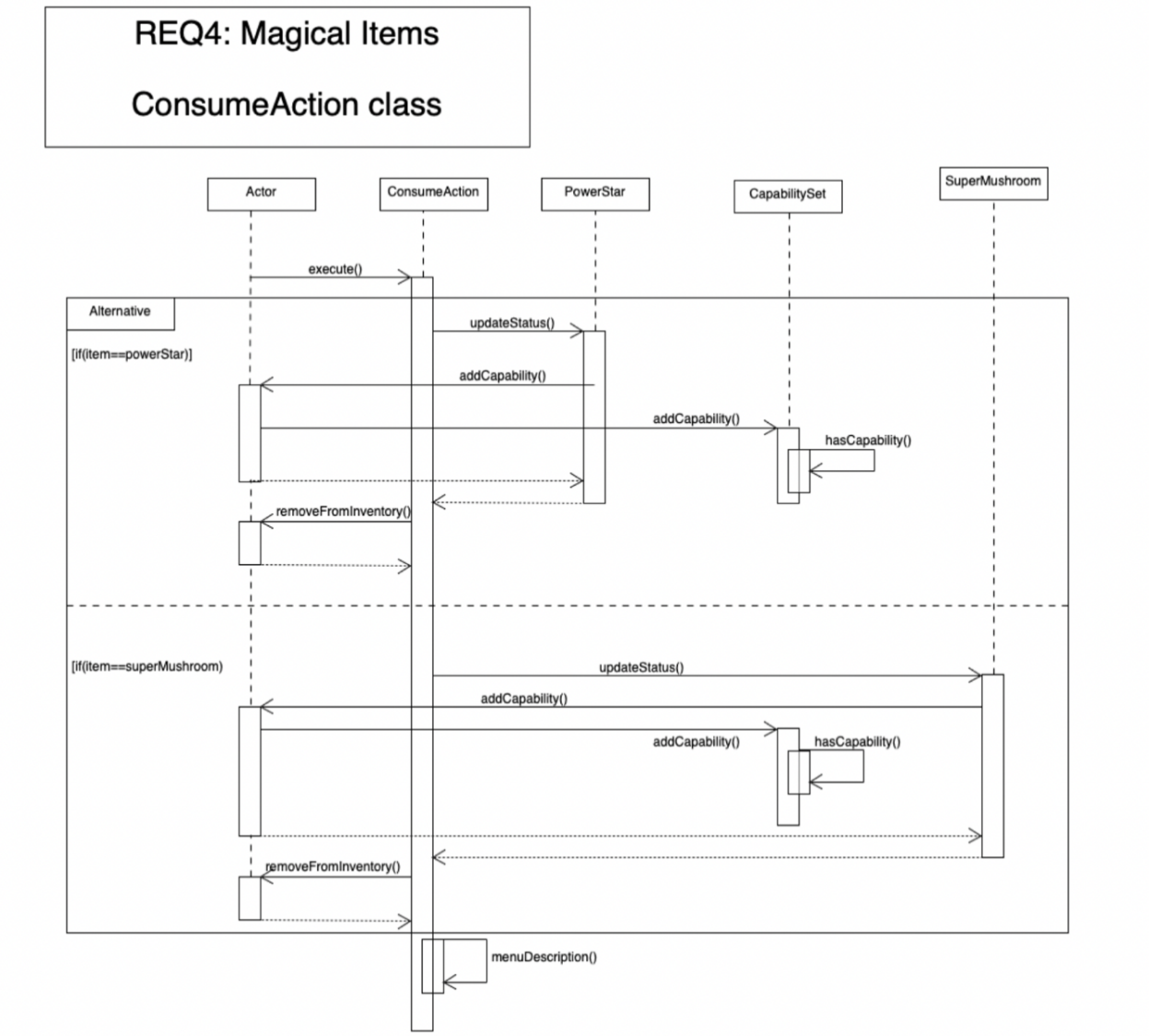
REQ4: Magical Items

Overview:

To achieve this feature, there will be five new classes (SuperMushroom, PowerStar, ConsumeAction, Wallet,Coin and PickCoinAction) in the extended system and two modified existing classes (Player and Status). The design rationale for each new or modified classes will be shown below.



Class diagram for REQ4



Sequence diagram for execute method in ConsumeAction class

1) Player class

Player is a class that represents the Player in the Game Map, is extends Actor class. It is an existing class given in the base code. In this class, a Wallet parameter is added into the Player class constructor. Hence making the Player class associated with the Wallet class.

Description of method:

- getWallet method is use to access the wallet attribute in the Player class constructor whenever the wallet balance needs to be access.

Why I choose to do it that way:

I decided to add a wallet parameter to the Player's constructor because a player can only have one wallet. By adding a wallet attribute into the Player class, this will ensure that wallet belongs to the particular player.

Advantage:

Open Close Principle can also be implemented as this class extends the Actor class, by adding functionality to the Actor class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Actor class. This enables the Actor class to support new functionalities as well as being added new methods easily. For example, Player class extends the Actor class functionality by having an extra wallet system feature. This class will then be further extended in other REQs, such as in REQ7, where the number of times player resets the game needs to be check to ensure player can only reset the game once.

Disadvantage:

N/A

2) ConsumeAction class

This class is use to update player's status and remove magical items from players inventory whenever player consumes the item. There is dependency between this class with the PowerStar and SuperMushroom class because we need to check which item is consume by players and update their status depending on item consumed.

Description of method:

- execute method is a method overridden from the parent class, Action class. It is use to update the player's status, adding capability to the player when player consumes the magical item. The item will then be removed from player's inventory. A description will then be display in the console showing which item is consumed by player (actor) using the menuDescription method.

```
@Override
    public String execute(Actor actor, GameMap map) {
        if(item==powerStar){
            powerStar.updateStatus(actor);
            actor.removeItemFromInventory(item);
        }
        if(item==superMushroom){
```

```
            superMushroom.updateStatus(actor);
            actor.removeItemFromInventory(item);
        }
        return menuDescription(actor);
    }

    @Override
    public String menuDescription(Actor actor) {
        return actor + " consumes a " + item;
    }
```

Why i choose to do it that way:

Since ConsumeAction is an action, functions in the action class are required (displaying actions carried out by player's into the console) and hence need to extend the action class. The execute and menuDescription from the Action class are implemented and overridden as shown in the code block above. This class is created to deal with magical items collected by players because items picked up by players will be added into the player's inventory first instead of carrying out their respective effect immediately. Hence this class is needed to update player's status after player choose to consume the items.

Advantage:

Using this design, the Single Responsibility principle can be implemented as this class will only be used when used to update player's status upon consuming magical items, hence higher cohesion.

Disadvantage:

As the game develops further, more item will be available for players to pick up and gain different capabilities. It might cause confusion during methods implementation as more and more if-else statements will be needed in the execute method.

3) PickCoinAction class

This class extends the Action class to allow player to pick up coins and then add the collected coins into the users wallet. This class has association relationship with the Coin class as there will be a coin attribute initialised in the PickCoinAction constructor so that it knows which coin (of what value) is picked.

```
private Coin coin;
    public PickCoinAction(Coin coin) {
        this.coin = coin;
    }
```

Description of method:

- addToWallet method adds the amount of coins collected by player into the player's wallet if player reaches location that contains coins. The coin will then be removed from the map after player collects it

```
public void addToWallet(Player player, GameMap map,Coin coin) {

        int itemLocation= coin.hashCode();
        int actorLocation= player.hashCode();

        if(itemLocation==actorLocation){
            map.locationOf(player).removeItem(coin);
            int current=player.getWallet().getBalance();
            player.getWallet().setBalance(current+ coin.getValue());
        }

    }
```

Why i choose to do it that way:

Since PickCoinAction is an action, functions in the action class are required and hence need to extend the action class. This class is created for players to pick up coins instead of using the PickUpItemAction in the engine package because the PickUpItemAction in the engine package stores item picked by player into the inventory. However, this is not the case for coins. Hence I created this class to add values of coin into the player's wallet instead of into player's inventory.

Advantage:

Using this design, the Single Responsibility principle can be implemented as this class will only be used when used to pick up coins (which this class will then be used to increase the player's wallet balance) hence higher cohesion. Open Close Principle can also be implemented as this class extends the Action class, by adding functionality to the Action class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Action class. This enables the Action class to support new functionalities as well as being added new methods easily. For example, the menuDescription method from the Action class can be override to display the amount of coin player received.

```
public String menuDescription(Player player) {
        return "Player receives " + coin.getValue() + " coins.";
    }
```

Disadvantage:

N/A

4) PowerStar class

This class is a subclass of items. Players that consume it will be healed by 200 hit points (hp) and become invincible. The invincible effect replaces fading duration (aka, fading turn's ticker stops), and it lasts for another 10 turns. It fades and disappears from the game within 10 turns. Sets the player’s status such that player does not need to jump to higher level ground, add 5 coins into player’s wallet for every destroyed ground, make player immune to damage and enable player to attack enemy successfully

Description of method:

- updateStatus method that add capabilities(effects of PowerStar) to the player. This method will be call when player consumes the Power Star

```
public void updateStatus(Player player){
        player.addCapability(Status.POWER_STAR_BUFF);
    }
```

Description of attributes:

- HEALED_HIT_POINTS is a public static integer attribute with value of 200 that indicates the hit points players can get healed by after consuming the Power Star

Why i choose to do it that way:

Since Power Star is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the effect it can bring to players and its ability to be traded for coins

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns

and as little overlapping as possible. Which in this class, the class is only responsible for storing information about the Power Star. Excessive use of literals was also prevent by declaring HEALED_HIT_POINTS as private static attribute. This prevents confusion during coding process. Furthermore, if the value of HEALED_HIT_POINTS needs to be change, changes only need to be done at one place, which is at the line where that attribute is declared instead of going through entire code and changing the value all of the “200” . This minimise possibilities of producing errors too. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Item class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, the PowerClass can use the addCapability method from the Item class to add capability to players.

Disadvantage:

Since player’s status will be updated, two if-statements will be used in the execute() method and the SuperMushroom class. This reduces the efficiency of the game as time will be spent on checking conditions of whether player is under the effect of Power Star (immunity effect where all enemies’ attack damage are 0) when player is being attacked by enemies. Furthermore, the Power Star provides player with the ability to reach higher level ground without jumping. Hence whenever player performs a jump action, an if statement may be needed to check if player is under Power Star effect too.

5) SuperMushroom class

This class is a subclass of items. The effect will last until it receives any damage (e.g., hit by the enemy). Once the effect wears off, the display character returns to normal (lowercase), but the maximum HP stays.

Description of attribute:

EXTRA_HP is a public static integer attribute with value of 50 that indicates the max hit points players can get after consuming the Super Mushroom.

Description of method:

updateStatus method that add capabilities to player such that:

- the display character evolves to the uppercase letter (e.g., from m to M).

- it can jump freely with a 100% success rate and no fall damage.
- increase max HP by 50

```
public void updateStatus(Player player){
        //display character evolves to the uppercase letter
        player.addCapability(Status.TALL);

        player.addCapability(Status.SUPER_MUSHROOM);
    }
```

Why i choose to do it that way:

Since Super Mushroom is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the effect it can bring to players.

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns and as little overlapping as possible. Which in this class, the class is only responsible for storing information about the Super Mushroom. Excessive use of literals was also prevent by declaring EXTRA_HP as private static attribute. This prevents confusion during coding process. Furthermore, if the value of EXTRA_HP needs to be change, changes only need to be done at one place, which is at the line where that attribute is declared instead of going through entire code and changing the value all of the “50” . This minimise possibilities of producing errors too. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Item class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, the SuperMushroom class can use the addCapability method from the Item class to add capability to players.

Disadvantage:

Since player’s status will be updated, an if statement may be used in the AttackAction class whenever the player is attacked by enemies to check whether player is under the Super Mushroom effect. This may reduce the efficiency of the game in terms of run

time, time complexity and resources as the if-statement will always run to check if player has capability provided by the Super Mushroom.

6) Coin class

This class is a subclass of Items. It is currency the player uses to trade for magical items. Coins will spawn randomly from the Sapling (t). The collected coins can be traded with Toad for Wrench, Power Star or Super Mushroom. A coin has an integer value (e.g., $5, $10, $20, or even $9001).

Description of method:

- getValue and setValue method is to return the value of coin when needed to add coin into the player's wallet and to set the value of coin upon spawning by the sapling

Why i choose to do it that way:

Since Coin is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the name, display character, portability and value.

A 'value' parameter is also added into the constructor as coins spawned will contains different values hence we need a setValue to set the value of the coin.

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns and as little overlapping as possible. In this class, the class is only responsible for storing and returning information about the Coin. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Actor class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, super() can be called set the characteristic of the coin instead of recreating constructors, getters and setters. Extra characteristic (value) can be added to the Coin too.

```
public Coin(String name, char displayChar, boolean portable,int value) {
        super("Coin", '$', false);
       setValue(value);
    }

public void setValue(int value) {
        this.value = value;
    }
public int getValue(){
        return this.value;
    }
```

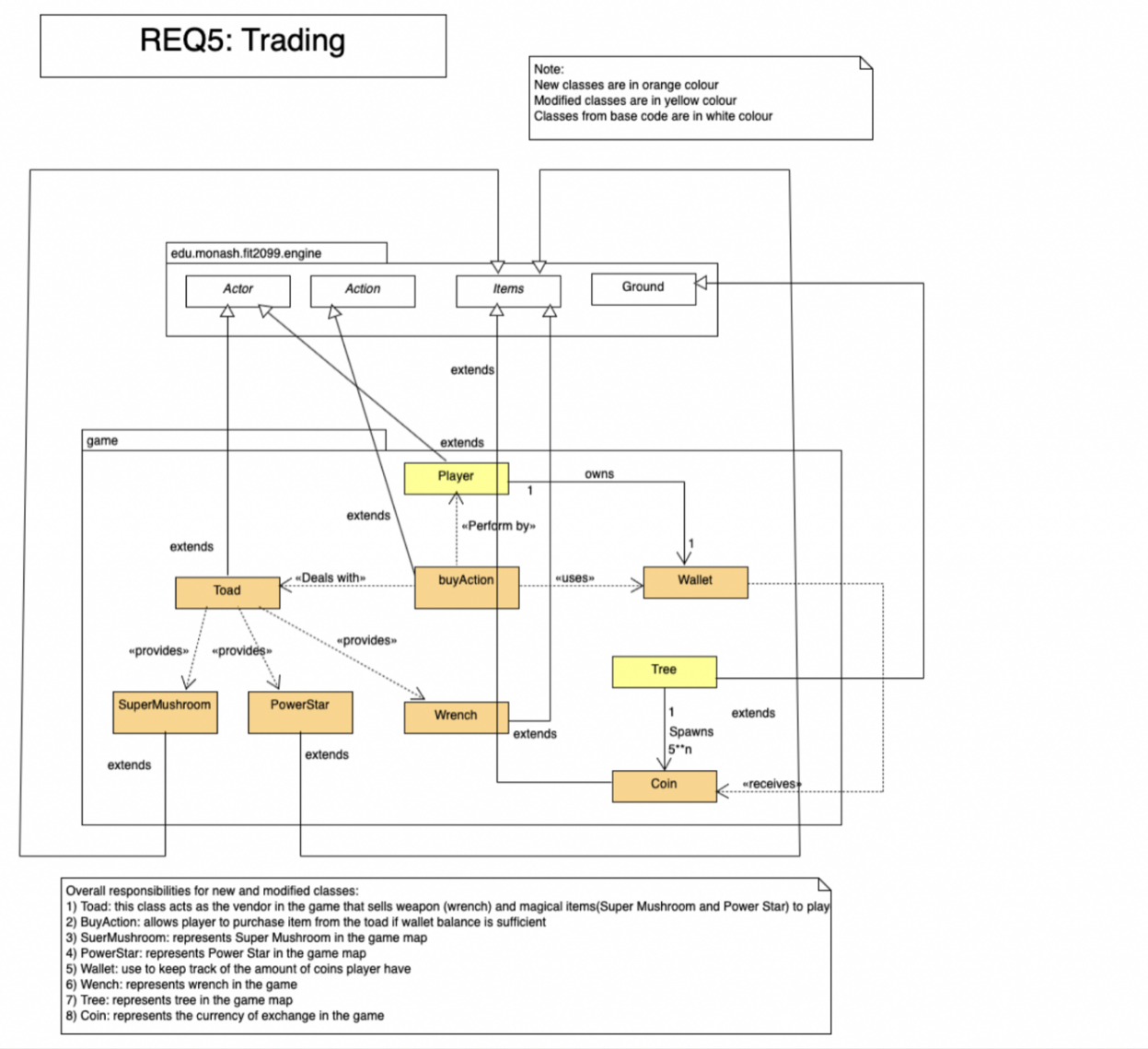
Disadvantage:

N/A

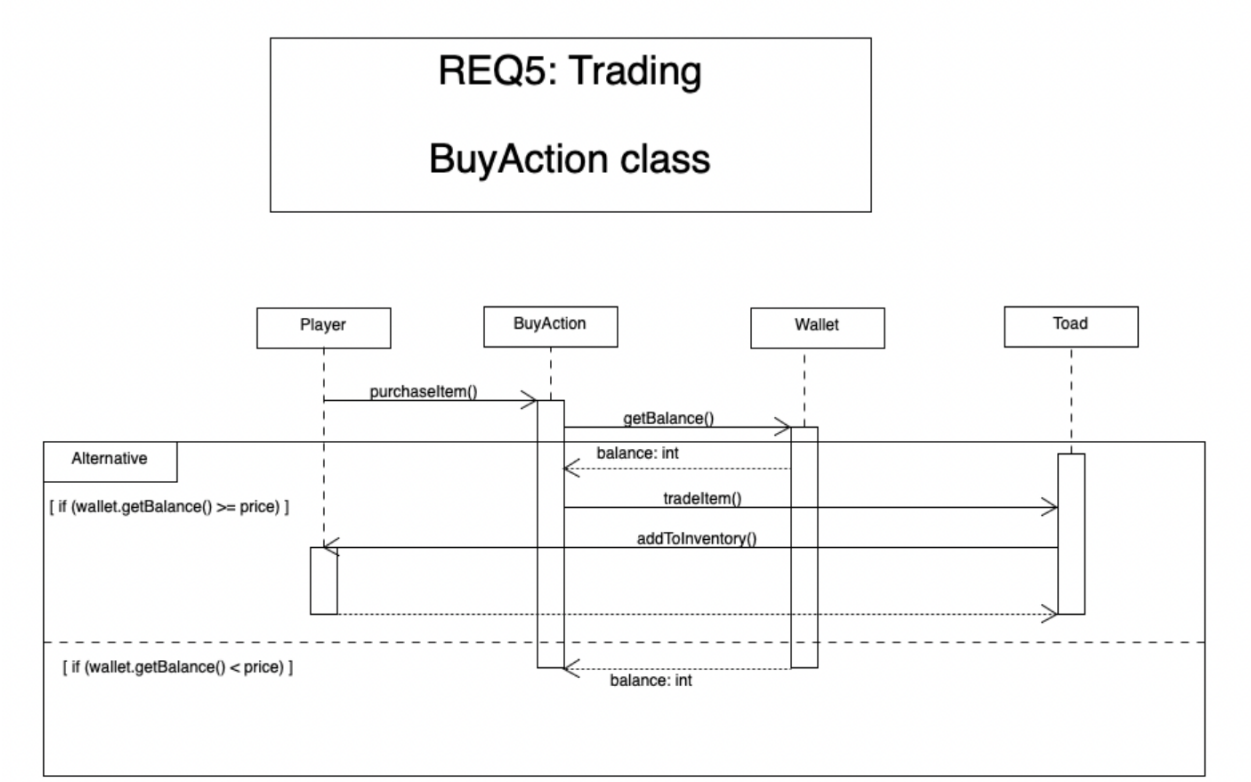
REQ5: Trading

Overview:

To achieve this feature, there will be seven new classes (SuperMushroom, PowerStar, BuyAction, Wallet, Coins, Toad and Wrench) in the extended system and two modified existing classes (Player and Tree). The design rationale for each new or modified classes will be shown below.



Class diagram for REQ5



Sequence diagram for purchaseItem method in BuyAction class

1) Tree class

The tree class represents tree in the game. Sapling (t) will randomly spawn coins(with different integer values) to be collected by players. Tree class will not be implemented in this REQ (its implementation will be considered and focus on in REQ1. Hence the design rationale for this class can be found in the REQ1 section.

2) Toad class

This class acts as the vendor in the game. Toad class sells weapons (wrench) and magical items (Super Mushroom and Power Star) to players by taking in coins. It is also used to upgrade player's attributes. Toad class extends actor as it is an actor and it is associated with BuyAction as players are allowed to purchase items from it. There is dependency between Toad class and the PowerStar, SuperMushroom and Wrench class because we need to add a new instance of those items into the player's inventory.

Description of attributes:

- SUPER_MUSHROOM_PRICE is a public static integer attribute with value of 400 that indicates the amount of coin to be deducted from player’s wallet when player purchases the Super Mushroom
- POWER_STAR_PRICE is a public static integer attribute with value of 600 that indicates the amount of coin to be deducted from player’s wallet when player purchases the Power Star
- WRENCH_PRICE is a public static integer attribute with value of 200 that indicates the amount of coin to be deducted from player’s wallet when player purchases the Wrench

```
    private static int SUPER_MUSHROOM_PRICE=400;
    private static int POWER_STAR_PRICE=600;
    private static int WRENCH_PRICE=200;
```

Description of method:

- tradeItem method that subtract a certain amount of coins from player’s wallet and provide items to player depending on item purchased by players. If item is a Power Star, subtract 600 coins from player’s wallet; if it is a Super Mushroom, subtract 400 coins player’s wallet; if it is a Wrench, subtract 200 from player’s wallet. Wallet balance will be set to the subtracted amount and purchased item will then be added to player’s inventory.

```
public void tradeItem(Wallet wallet,Item item, int price,Actor actor){
        int currentBalance=wallet.getBalance();
        currentBalance-=price;
        wallet.setBalance(currentBalance);
        actor.addItemToInventory(item);
    }
```

Why i choose to do it that way:

Since Toad is an actor, functions in the Actor class are required and hence need to extend the Actor class. I created this class in order to enable player to use coins to trade for weapons and magical items.

Advantages:

Excessive use of literals was also prevent by declaring SUPER_MUSHROOM_PRICE, POWER_STAR_PRICE and WRENCH_PRICE as private static attribute. This prevents confusion during coding process. Furthermore, if the value of HEALED_HIT_POINTS needs to be change, changes only need to be done at one place, which is at the line where that attribute is declared instead of going through entire code and changing their values . This minimise possibilities of producing errors too. Open Close Principle can also be implemented as this class extends the Actor class, by adding functionality to the Actor class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Actor class. This enables the Actor class to support new functionalities as well as being added new methods easily. For example, super() can be called set the characteristic of the toad instead of recreating constructors, getters and setters

```
public Toad(String name, char displayChar, int hitPoints) {
        super(name, displayChar, 0);
    }
```

Disadvantages:

N/A

3) BuyAction class

BuyAction is an action that allows the player to use coin to purchase items from the toad. It is trigger by the Toad class when players use buy action to purchase items from the toad. This class has dependency relationship with the Toad class as a new instance of Toad will be created each time this action is carried out.

Description of method:

- purchaseItem method checks if actor’s wallet has enough money, if yes, enable player to trade with the toad using the tradeItem method . Else, inform player that balance is insufficient

```
public void purchaseItem(int price,Item item,Actor actor){
        if (wallet.getBalance()>=price){
            toad.tradeItem(wallet,item, price,actor);
        }
```

```
        else{
            System.out.println("insufficient balance");
        }
    }
```

Why I choose to do it that way:

Since BuyAction is an action, functions in the Action class are required and hence need to extend the Action class. I created this class in order to enable player to purchase items from the toad.

Advantage:

This class is created using the Single Responsibility Principle where it does not need to take extra responsibility. In case of need to change responsibility, all pieces needed will be there. This makes the system easier to maintain and extend. Therefore, this class is only responsible for allowing player to trade with the Toad if wallet balance is sufficient. This is because in the future there may be more items available for player's to purchase. The Liskov Substitution Principle can also be fulfil as the initial meaning of the purchaseItem method behaviour from the Action class.

Disadvantage:

N/A

4) Wallet class

This class is used to store and keep track of the amount of coins players have. The Player class is associated with this class because there is a wallet attribute in the Player class constructor to ensure a wallet belongs to a particular player.

Description of method:

- getBalance and setBalance method returns the current amount of coins in player's wallet and set the new wallet balance

```
public int getBalance() {
        return balance;
    }
public void setBalance(int balance) {
        this.balance = balance;
    }
```

- receiveCoin method to add coin value into the wallet’s balance

```
public void receiveCoin(Coin coin){
        int currentBalance=getBalance();
        int newBalance= currentBalance + coin.getValue();
        setBalance(newBalance);
    }
```

Why i choose to do it that way:

I created this class to keep track of the amount of coins players have and implement a getWalletBalance method to check if players balance is sufficient to purchase magical items and to increase player’s wallet balance whenever player receives coin.

Advantage:

This class is created using the Single Responsibility Principle where it does not need to take extra responsibility. In case of need to change responsibility, all pieces needed will be there. This makes the system easier to maintain and extend. Therefore, this class is only responsible to constantly being update about player’s coin amount and return the amount of coins players have if needed

Disadvantage:

N/A

5) Wrench class

This class is a subclass of items. It represents the weapon players use in the game to defeat enemies, such as the Koopa’s shell. It has 80% hit rate and 50 damage.

Description of method:

- updateStatus method that add capabilities (ability to attack) to the player. This method will be call when player gets a wrench.

```
public void updateStatus(Player player){
        player.addCapability(Status.HAVE_WRENCH);
    }
```

Why i choose to do it that way:

Since Wrench is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the hit rate and damage

Advantage:

Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Item class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, the Wrench can use the addCapability method from the Item class to add capability (HAVE_WRENCH) to players.

Disadvantage:

N/A

6) Coin class

This class is a subclass of Items. It is currency the player uses to trade for magical items. Coins will spawn randomly from the Sapling (t). The collected coins can be traded with Toad for Wrench, Power Star or Super Mushroom. A coin has an integer value (e.g., $5, $10, $20, or even $9001).

Description of method:

- getValue and setValue method is to return the value of coin when needed to add coin into the player’s wallet and to set the value of coin upon spawning by the sapling

Why i choose to do it that way:

Since Coin is an item, functions in the Item class are required and hence need to extend the Item class. I created this class in order to store information about this item, such as the name, display character, portability and value.

A ‘value’ parameter is also added into the constructor as coins spawned will contains different values hence we need a setValue to set the value of the coin.

Advantage:

This class is created using the Separation of Concern principle where the program is separated into sections with its own responsibilities, by having well- defined concerns and as little overlapping as possible. In this class, the class is only responsible for storing and returning information about the Coin. Open Close Principle can also be implemented as this class extends the Item class, by adding functionality to the Actor class without modifying its already available functionalities, in a way that does not change the way we use existing code in the Item class. This enables the Item class to support new functionalities as well as being added new methods easily. For example, super() can be called set the characteristic of the coin instead of recreating constructors, getters and setters. Extra characteristic (value) can be added to the Coin too.

```
public Coin(String name, char displayChar, boolean portable,int value) {
        super("Coin", '$', false);
       setValue(value);
    }

public void setValue(int value) {
        this.value = value;
    }
public int getValue(){
        return this.value;
    }
```

Disadvantage:

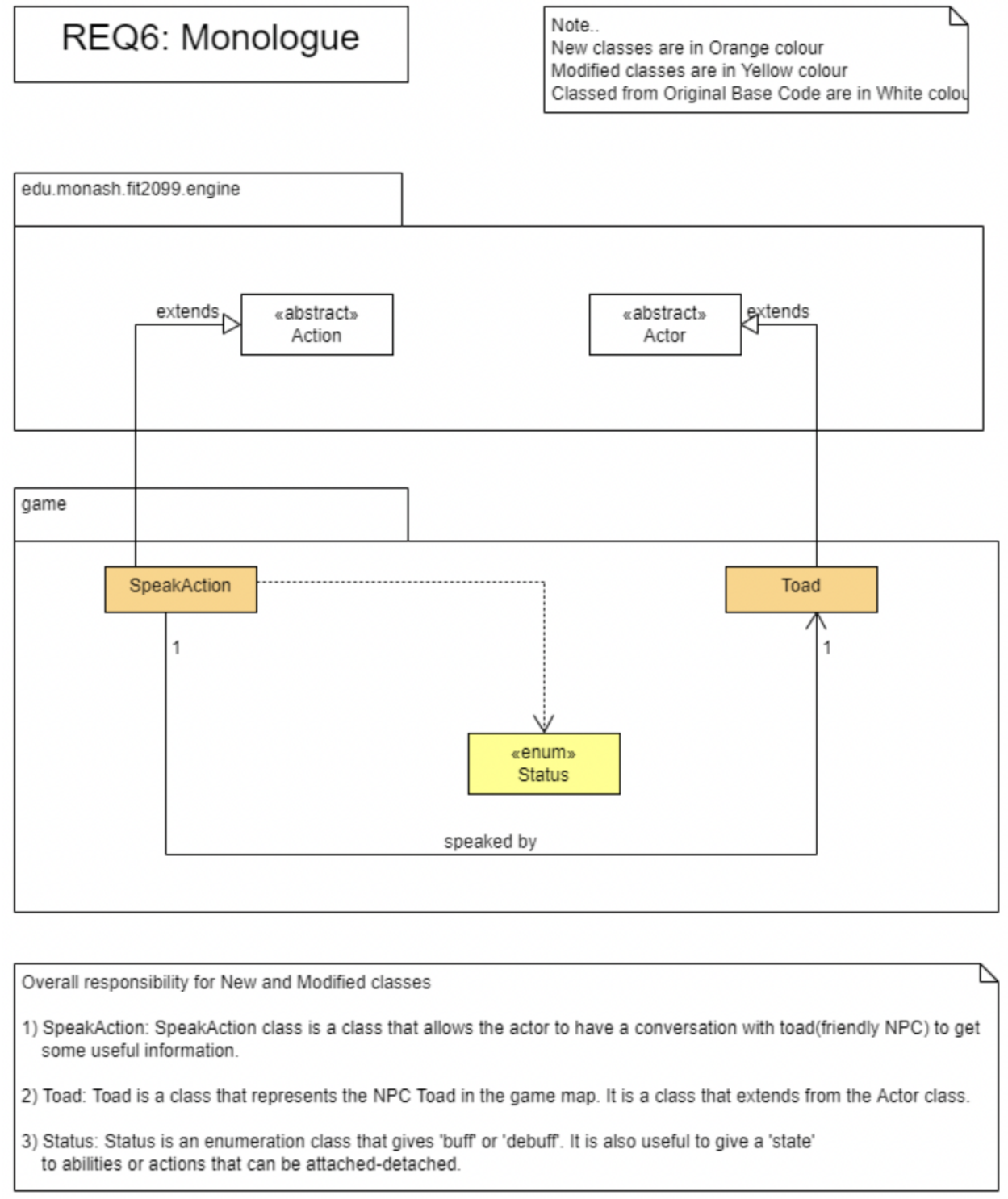
N/A

REQ6: Monologue

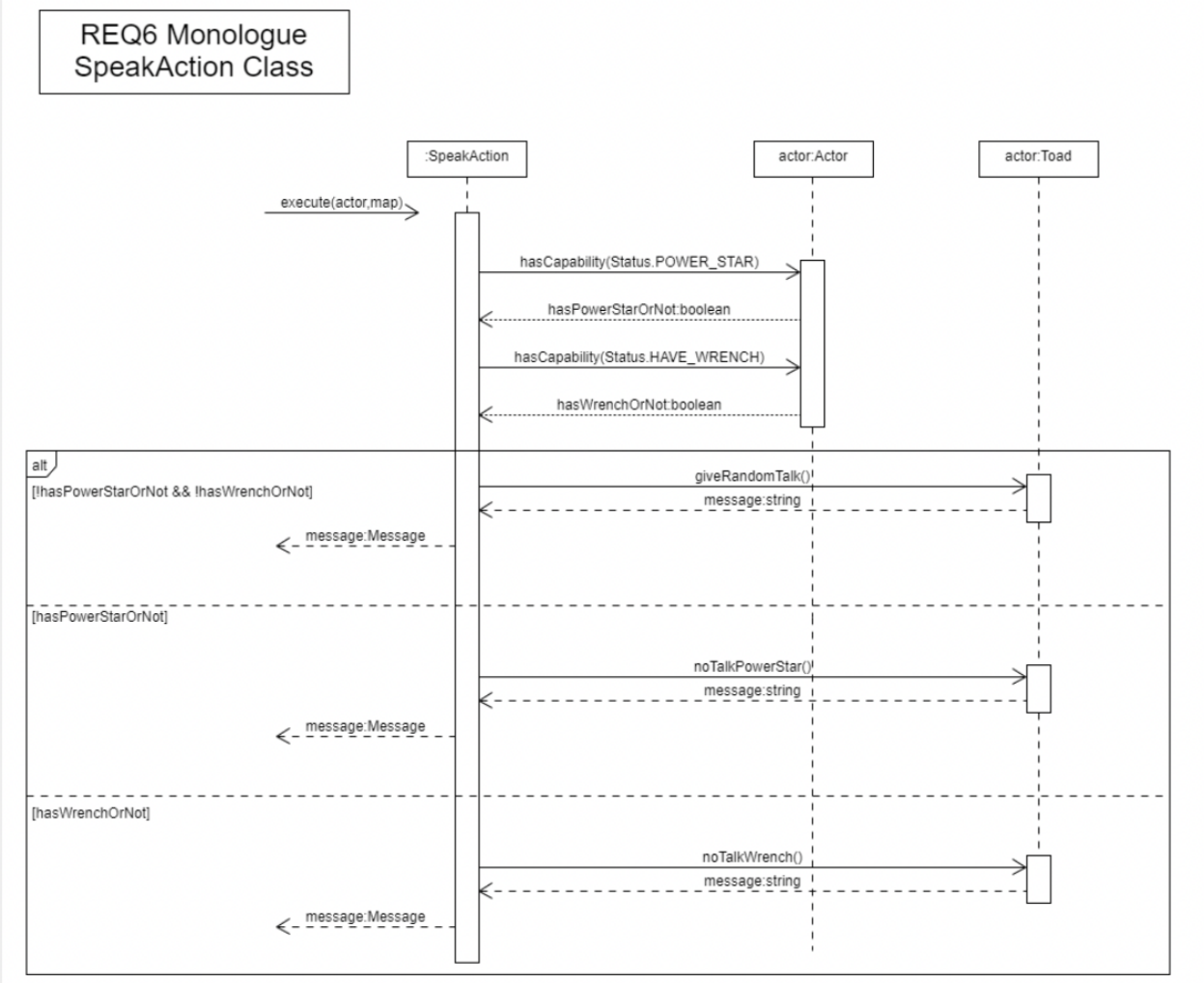
Overview:

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

on the following pages.



Class diagram for REQ6



Sequence diagram for execute method in SpeakAction class

1) SpeakAction Class

SpeakAction class is a class that allows the actor to have a conversation with a toad(friendly

NPC) to get some useful information. It is a new class that extends the Action class. In this class, there is 1 attribute which is the toad object.

Description of class attribute:

- toad is an instance of Toad class. It will be used to have conversations with actor.

The SpeakAction class overrides the execute and menuDescription method from its parent class.

Description of methods:

- For the execute method, there are 4 sentences maybe print out:

1. "You might need a wrench to smash Koopa's hard shells.",

2. "You better get back to finding the Power Stars.",

3. "The Princess is depending on you! You are our only hope.",

4. "Being imprisoned in these walls can drive a fungus crazy :("

There will be an if-else statement. When the actor is close to the toad:

If the actor didn't consume power star AND have a wrench (i.e., check the actor' s capability), the toad will speak one sentence from that four sentences at a time randomly.

Else if the actor consumes the power star, the toad will only speak one sentence from sentences(1,3,4) at a time randomly.

Else if the actor buys a wrench, the toad will only speak one sentence from sentences(2,3,4) at a time randomly.

- menuDescription will return a string that will appear on the command list in the console. This string states that the actor speaks to the toad.

Why I choose to do it that way:

Since SpeakAction is an action and we need the functions in the Action class, and thus we let SpeakAction class extend Action class. However, since we need to ensure that when SpeakAction is called, the actor will have a conversation with toad, hence we will need to override the execute method to ensure our game logic works properly. Besides that, I had also overridden the menuDescription method, the reason to do so is to let the user have a better understanding on what will happen when he/she lets the actor performs a SpeakAction to toad.

Advantage:

According to the design above, we are adhering to the Single Responsibility Principle as the SpeakAction class only focuses on the action that will be executed when the

actor speaks to the toad. Besides that, we fulfill the Liskov Substitution Principle as SpeakAction preserves the meaning of executing and menuDescription method behaviors from Action class.

Disadvantage:

N/A

2) Toad Class

Toad is a class that represents the NPC Toad in the game map. It is a class that extends from the Actor class. According to the assignment specification, toad is a NPC is friendly (0 hits point) and it will have a conversation with the actor when the actor is beside it.

For this class, I created an ArrayList to store the 4 sentences of a toad. By using the indexation I can display the correct sentence according to the actor's capabilities.

In addition, the Toad class has overridden 2 methods from its parent class which are playTurn() and allowableActions(). Besides that, I created 3 new methods to display the conversation between toad and actor which are giveRandomTalk(), noTalkPowerStar() and noTalkWrench().

- playTurn() states the behavior of the Toad. If the toad stays on its own position(didn't move around), it returns a string: toad does nothing
- allowableActions will return a SepakAction instance if the actor is currently one step beside the Toad else return an empty ActionList. By doing so, we can make sure the actor will not be able to speak to the toad if the actor is currently far away from the toad.
- giveRandomTalk will return a sentence randomly from the four sentences.
- noTalkPowerStar will return a sentence randomly from the three sentences.

("You better get back to finding the Power Stars." will be removed from the

ArrayList since the actor has power star buff already)

- noTalkWrench will return a sentence randomly from the three sentences.

("You might need a wrench to smash Koopa's hard shells." will be removed

from the ArrayList since the actor has wrench already)

Why I choose to do it that way:

In this Toad class I create an ArrayList for the four sentences, rather than an array I think ArrayList is easier to modify and get the exact output I want. By doing so, I can edit

the ArrayList content for different methods, and will not have so much redundant code.

Advantage:

By creating an ArrayList for all the sentences of toad instead of creating an array, I can

add/remove the sentence from the ArrayList easily since the size of ArrayList is not fixed. According to the assignment requirement, when the actor consumes the power star (have power star buff), toad will not remind the actor again to get a power star, so I just remove the sentence regarding to power star from the ArrayList by using

ArrayList.remove(). The same goes to wrench when an actor has wrench, toad will not as the actor to get a wrench, so I just remove the sentence regarding wrench. It fulfills

the “dry”(don’t repeat yourself).

Disadvantage:

N/A

3) Application Class

The application class is the driver class of the whole game, it contains the default game

map. In this class, I need to initialize the new actor Toad into the game map since he

stands in the middle of the map(surrounded by brick Walls) and never moves around.

Why I choose to do it that way:

In this class, I just initialize the instance of Toad to create a new toad object and make it appears on the game map when the game is start. I can also specify its position when I

add it to players.

Advantage:

By initializing the toad here, it will be easy for me to edit its position and displayChar. I can modify it anytime I want.

Disadvantage:

N/A

4) Status Class

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. It is an existing enumeration class with some additional modifications. In this class, I added two

statuses, which are POWER_STAR_BUFF and HAVE_WRENCH.

Description of constant:

- POWER_STAR_BUFF indicates that the actor has consumed a power star,

so the toad doesn’t need to tell the actor to find a power star.

- HAVE_WRENCH indicates that the actor has bought a wrench, so the toad

don’t need to tell the actor to get a wrench.

Why I choose to do it that way:

Instead of creating class attributes to indicate whether a particular actor has a wrench or consume power star, it is better to make use of the enumeration class to make our code easier to read. Besides that, I also can utilize the engine code provided to check the actor's capability to simplify our code

Advantage:

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.

Disadvantage:

As the game develops, more and more capabilities will be added to this status class. If there are too many statuses within this class, it might cause confusion when debugging.

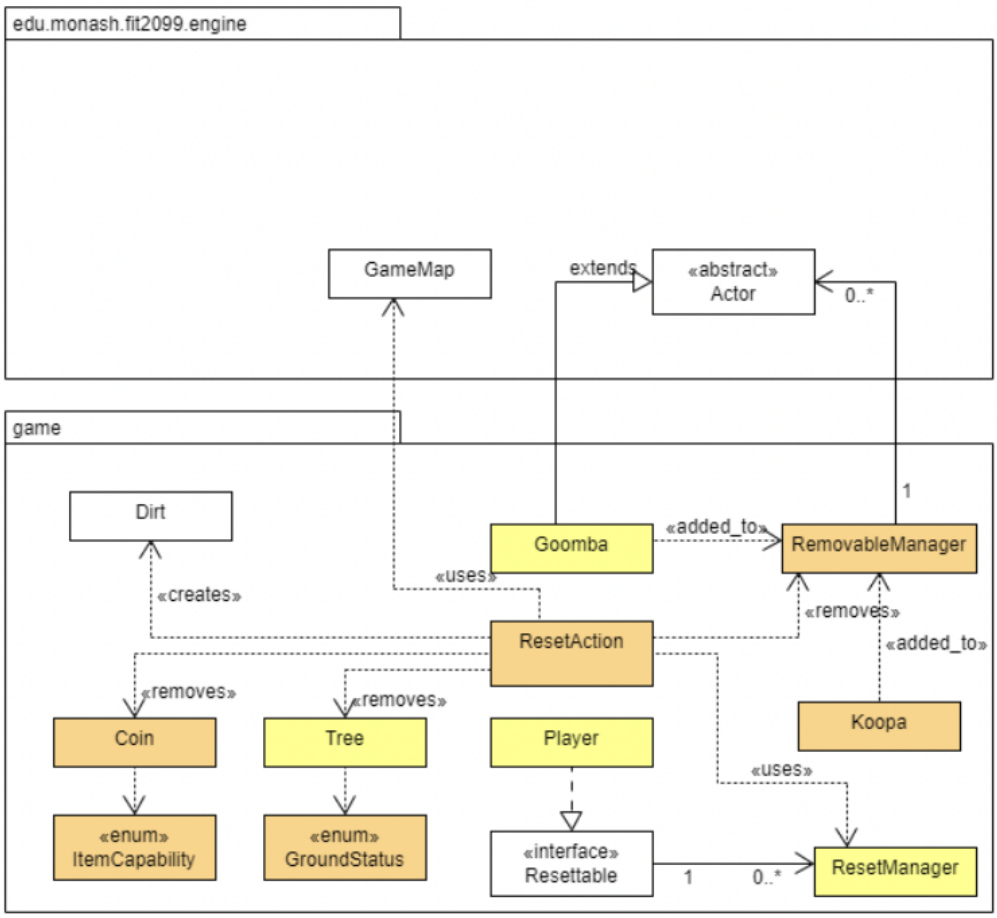
REQ7: Reset Game

Overview:

To achieve this feature, there will be six new classes (i.e., Koopa, RemovableManager, ResetAction, ItemCapability, GroundStatus and Coin) created in the extended system, and four existing classes (i.e., Goomba, Tree, ResetManager and Player) will be modified. The design rationale for each new or modified class is shown on the following pages. However, the explanation of the design rationale for ResetAction class will be the last among all ten classes. The reason is that the implementation of ResetAction uses all other classes, hence I would need to explain the design rationale for other classes first so that the marker can understand the implementation of ResetAction better.

REQ7: Reset Game

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

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

3) RemovableManager: A class that stores all the Actors that will be remove after reset option is selected

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

5) Tree: A class that represents the tree in the Game Map

6) ItemCapbility: This enum class is used to specify the capabilityof an Item has

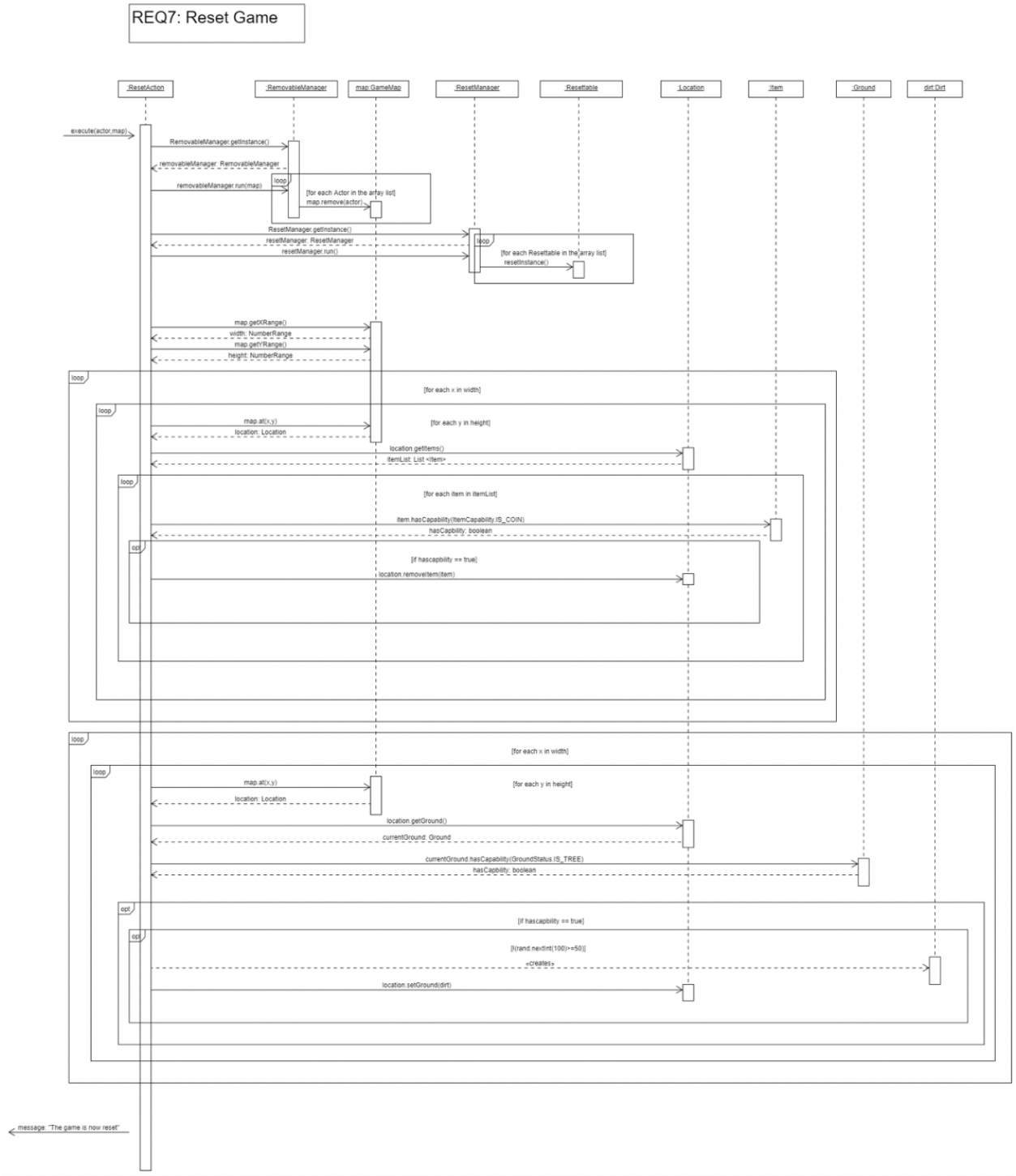
7) GroundStatus: This enum class is used to specify the status of a Ground has

8) Player: A class that represents the Player in the Game Map

9) ResetManager: A global Singleton manager that does soft-reset on the instances

10) ResetAction: A reset action that allow user to reset the game

Class diagram for REQ7



Sequence diagram for execute method in ResetAction class

1) ItemCapability class

ItemCapability is a new enumeration class with only one constant that represents whether the item on a particular location is a Coin or not, which is IS_COIN.

Description of constant:

- An item has a capability of IS_COIN if it is an instance of Coin.

Why I chose to do it that way:

For this requirement, when the reset action is selected, I must remove all the coins on the ground, hence I would need a way to identify whether the item is a Coin or not. My initial thought is to use “instanceOf” to determine whether an item is a Coin instance, however, I realized this is not a good design practice, therefore I had decided to create this enumeration class instead. Thus, for each Coin instance, IS_COIN capability will be added to them.

Advantages:

With this enumeration class, we can avoid excessive use of literals as we do not need any local parameter to keep track of whether an item is a coin instance. Besides that, this class is created to adhere to the Separation of Concerns principle as the enumeration class only focuses on a single aspect, that is the possible capability of an item could have. This kind of implementation also provides us more flexibility for future development as we can simply add additional capability when needed.

Disadvantages:

As the game develops, more and more capability will be added to this ItemCapability class. If there are too many constants within this class, it might cause confusion when debugging.

2) Coin class

Coin class is a class that represents the coin item in this game. However, since Coin class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ5) and in this REQ, it only add the IS_COIN capability using addCapability() method within Coin’s constructor without considering any features of a Coin instance, therefore the design rationale for Coin class will not be available in this section. The usage of IS_COIN is explained in the design rationale of ItemCapability

class. **Hence, please go to the REQ4 and REQ5 section in the following pages to see the design rationale for Coin class.**

3) GroundStatus class

Note: the design rationale for GroundStatus class in REQ7 will be the almost same as the one in REQ1 as they both referring to the same class, but the design rationale will focus on different aspect (i.e., REQ1 focus on determining whether a ground is a fertile ground whereas REQ7 focus on determining whether a ground is a tree), hence there will still be some repetitive content.

GroundStatus is a new enumeration class with only one constant that represents the status of ground, which is IS_TREE.

Description of constant:

- A ground has a status of IS_COIN if it is a Tree instance.

Why I chose to do it that way:

According to the assignment specification for this requirement, it says that when the user selects the reset option, Tree has a 50% chance to be converted back to Dirt. It means that in the ResetAction class, when I loop through the whole map (will discuss more in the ResetAction section), I would need to know what types of ground is at a particular location. Therefore, by adding this capability to each Tree instance, it helps to determine whether the ground on a particular location is a Tree, so that I can perform the corresponding task (i.e., 50% chance to be converted back to Dirt).

Advantages:

Using constant to specify that whether a ground is a Tree helps us to adhere to the “avoid excessive use of literals” principle as we do not have to create any literals to keep track of the status of a ground. This implementation also provides us the flexibility for future development. For instance, if we decided to remove more types of ground or convert one type of ground to another when reset action is selected in future, we can simply add more constant in this class to monitor the type of a particular ground. We also adhere to the Single Responsibility Principle as this enumeration class is only responsible to store the possible statuses that a Ground can have.

Disadvantages:

Similar to ItemCapability class, as the game develops, more and more status will be added to this Ground status class. If there are too many constants within this class, it might cause confusion when debugging.

4) Tree class

Tree class is a class that represents the tree in this game. However, since Tree class will not be implemented in this REQ (i.e., it's implementation will only be considered in REQ1 and REQ2) and in this REQ, it only add the IS_Tree capability using addCapability() method within Tree's constructor without considering any features of a Tree instance, therefore the design rationale for Tree class will not be available in this section. The usage of IS_TREE is explained in the design rationale of the GroundStatus class. **Hence, please go to the REQ1 and REQ2 section in the following pages to see the design rationale for Tree class.**

5) RemovableManager class

RemovableManager class is a global singleton manager that stores all the actors that will be removed after the reset action is selected. In this case, we are using the static factory method. Within this class, it has two class attributes, one constructor and three methods.

Description of attribute:

- removeableActorList is a private static array list that stores Actor instance that will be remove after the reset action is selected.
- instance is a private static RemovableManager that represents the unique instance of this class.

Description of constructor:

- the constructor of this class will initialize the removeableActorList to an empty array list.

Description of method:

- getInstance is a method that will return the only instance of this class if the instance is not null, otherwise it will create a new instance of this class and return it.
- appendRemovableActor will append the input parameter which is an Actor instance to removeableActorList.

- run takes an input parameter of a GameMap instance. After that, it will loop through the removeableActorList and call the removeActor(removeableActor) on the map instance to remove each Actor in the array list

Why I chose to do it that way:

By using this static factory method, it helps me to monitor all the Actors that are ready to be removed. This method is very convenient as I only have to call the run() method to remove everything efficiently. Besides that, with this design, if we had decided to remove more Actor instances when reset action is selected by the user, we could simply add that instance to this class, hence we do not have to modify the existing code to achieve this objective. Thus, according to the specification of this requirement, all Koopa and Goomba instances will be added to the array list in this class.

Advantages:

With the above design, we make our class open for extension (i.e., Actor that are removeable can be added to this class) but closed for modification (i.e., we do not need to modify RemoveableManager class), so we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are either a command or a query. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Moreover, we also follow the Single Responsibility Principle as this class is only responsible for storing the removeable Actor instance and performing remove action on it.

Disadvantages:

We store all the removeable actor instance in a single array list (i.e., both Goombas and Koopas are storing in a single array list), hence there is no way for us to identify the actual instance of the Actor (i.e., no way to identify whether the Actor is Goomba or Koopa). Thus, if we need to perform specific operations to each type of enemy in future implementation, this might be a serious problem.

6) Koopa class

Koopa class is a class that represents one of the enemies in this game, which is Koopa. However, since Koopa class will not be implemented in this REQ (i.e., it’s implementation will only be considered in REQ3) and in this REQ, it only adds the

Koopa instance to the RemoveableManager class by using the appendRemovableActor method within Koopa's constructor without considering any features of a Koopa instance, therefore the design rationale for Koopa class will not be available in this section. **Hence, please go to the REQ3 section in the following pages to see the design rationale for Koopa class.**

7) Goomba class

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

8) ResetManager class

ResetManager is a global Singleton manager that does soft reset on the instances. It is an existing class given in the base code. In this class, I had only modified the run method and the appendResetInstance method.

Description of method:

- when the run method is called, it will loop through the array list that stores the instances of Resettable and for each resettable instance, it will call the resetInstance() method.
- appendResetInstance is a method that appends the input parameter with the type of Resettable to the array list in this class.

Why I chose to do it that way:

According to the specification of this requirement, I would need to heal the player to maximum and reset the player status. Hence, in order to do this, I had decided to let Player class implement Resettable and it will then add the instance of Player to ResetManager by using appendResetInstance method. Besides that, inside the Player

class, I will also implement the resetInstance() method (will discuss more in the Player section) to heal the player to maximum and reset the player status. Thus, when the run method is called, it will call the resetInstance() of the Player and perform the corresponding operations. With this design, it provides us the flexibility for future development. For instance, if there are more actors that need to reset their status once reset button is pressed in future, we could simply let the class of that actor implement Resettable and add it to ResetManager. Simply calling the run () method in this class will automatically reset all the actors.

Advantages:

With the above design, we make our class open for extension (i.e., Actor that are resettable can be added to this class) but closed for modification (i.e., we do not need to modify ResetManager class), so we are adhering to the Open-closed Principle (OCP). In addition, we can see that all the methods above are either a command or a query. Hence, we are following the Command-Query Separation Principle as all the methods are either a command or query but not both. Moreover, we also follow the Single Responsibility Principle as this class is only responsible for storing the resettable Actor instance and performing resetInstance action on it. Besides that, it also allows each Resettable instance to have their own resetInstance method, which provides much flexibility for future implementation.

Disadvantages:

Similar to RemoveableManager class, we store all the resettable actor instance in a single array list (i.e., currently only Player is in the array list), hence there is no way for us to identify the actual instance of the Actor (i.e., no way to identify whether the Actor is Player or something else). Thus, if we need to perform specific operations to each type of resettable in future implementation, this might be a serious problem.

9) Player class

Player is a class that represents the Player in the Game Map, extends Actor class, and implements Resettable. It is an existing class given in the base code. In this class, I had created a new class attribute and only modified the constructor of the Player class, playTurn method and added a new method called resetInstance.

Description of attribute:

- oneReset is a Boolean which is initialized as false. After the user performs the reset action then it will then set the true. (Will discuss more in ResetAction section)

Description of constructor:

- For the constructor of the Player class, the only modification that I had done is to add the Player instance to the ResetManager by using this line of code, this.registerInstance()

Description of method:

- playTurn is a method to figure out what to do next for the Player. In this method, the only modification I have done is to use an if statement to check whether oneReset has been set to true. If oneReset is true, it means the user had performed the reset action hence reset action should not be available to the user anymore, otherwise, oneReset will be false and allow the user to perform the reset action on the menu.
- resetInstance is a method that must be implemented since the Player class now implements Resettable. In this method, it will use a for loop to loop through all the status of this Player instance and remove every single status except HOSTILE_TO_ENEMY. After that, it will heal the player to maximum by using this line of code, this.heal(this.getMaxHp()). Eventually, it will set oneReset to true.

Why I chose to do it that way:

For this requirement, I had to remove all the status of the Player except and heal the Player to maximum. Initially, I was thought to do these operations in the ResetAction class. However, I realized that this is not an efficient way. The reason is because what if there is more Player/Actor exists in the Game and needs to be reset after the reset button is pressed? If I reset each Player/Actor in the ResetAction class, the code will be very long and not readable and maintainable. Therefore, by using the ResetManager and Resettable interface, it helps me to reset all the Player/Actor that are required to reset by just calling a single method, which is run () in ResetManager. This is an efficient way and makes our code readable and maintainable and easy to extend.

Advantages:

With the above design, we are adhering to the Single Responsibility Principle as the method within Player class only shows the properties of a Player and what a Player can do. Moreover, we also fulfil the Liskov Substitution Principle as all the methods in Player class still preserve the meaning from its parent class, which is Actor. Other than that, we

also created a class attribute called oneReset to keep track whether the reset action has been used, this is to avoid excessive use of literals principle.

Disadvantages:

N/A

10) ResetAction class

ResetAction is a class that will reset the game when the user selects the reset option on the menu. It is a new class that extends Action class. In this class, it overrides the execute menuDescription and hotkey method from its parent class.

Description of method:

- For the execute method, it will only be called when the user selects the reset option on the menu. In this method, firstly, it will get the instance of the RemoveableManager and call the run method. In the previous section, we know that RemoveableManager stores all the Actors (i.e., currently Koopa and Goomba) that should be removed once reset option is selected, therefore, by using the run method, we can remove all the enemies from the map. After that, it will get the instance of the ResetManager and call the run method. In the previous section, we also know that ResetManager currently stores the Player instance. Hence, when we call the run method in ResetManager, it will reset the status of the Player and heal the Player to maximum. Next, in order to remove all the Coin instances from the map, we will loop through every single location in the map using the map.at(x,y) method by using two for loops where x and y are the coordinates for each location. Thus, for each location, we will check the itemList for that location, if there is a Coin instance (by checking whether the item has the capability of IS_COIN), then we will remove it (i.e., using removeItem()). In addition, based on the description of this REQ, each Tree instance will have 50% to convert back to Dirt. Therefore, similarly, we will use two for loops to loop through every single location on the map and get the ground instance of that location using getGround() method. After that, I will set that particular ground to a Dirt by using the setGround() method with 50% chance. Eventually, a string “The game is now reset” will be printed out. In conclusion, by doing all the operations above, we had successfully fulfilled the requirement for this feature.
- menuDescription will return a string that will appear on the command list in the console. The String that will be returned is “Reset the game”.

- hotkey returns a key used in the menu to trigger this reset action. In this case, the hotkey is ‘r’.

Why I chose to do it that way:

Since ResetAction is an action and we need the functions in the Action class, and thus we let ResetAction class extend Action class. However, since we need to ensure that when ResetAction is called, the corresponding operations will be performed (i.e., all the scenario listed in the assignment specification), hence we will need to override the execute method, menuDescription and hotkey to ensure our game logic works properly. The description and the method to perform the logic for the execute method is mentioned above. Besides that, the reason for me to override the menuDescription method is to let the user have a better understanding on what will happen if he/she pressed ‘r’ on the console.

Advantages:

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

Disadvantages:

N/A