

FIT2099 Assignment 1 Final Submission

by Lab 5 Team 3

George Tan Juan Sheng (30884128)

Samuel Tai Meng Yao (32025068)

Shun Yao Tee (32193130)

Work Breakdown Agreement

Lab 5 Team 3:

1. George Tan Juan Sheng (30884128)
2. Samuel Tai Meng Yao (32025068)
3. Shun Yao Tee (32193130)

We will separate the **class diagram class and sequence diagram** into three part

Part 1 (Author: Samuel Tai Meng Yao)

REQ1 - Let it grow!

REQ6 - Monologue

REQ7 - Reset Game

Initial commit: 7/4/2022

Completion: 10/4/2022

Reviewer: George Tan Juan Sheng, Shun Yao Tee

Part 2 (Author Shun Yao Tee)

REQ2 - Jump Up, Super Star!

REQ5 - Trading

Initial commit: 7/4/2022

Completion: 10/4/2022

Reviewer: George Tan Juan Sheng, Samuel Tai Meng Yao

Part 3 (Responsible by George Tan Juan Sheng)

REQ3 - Enemies

REQ4 - Magical Items

Initial commit: 7/4/2022

Completion: 10/4/2022

Reviewer: Samuel Tai Meng Yao, Shun Yao Tee

Signed by (type "I accept this WBA):

I accept this WBA. - Samuel Tai Meng Yao

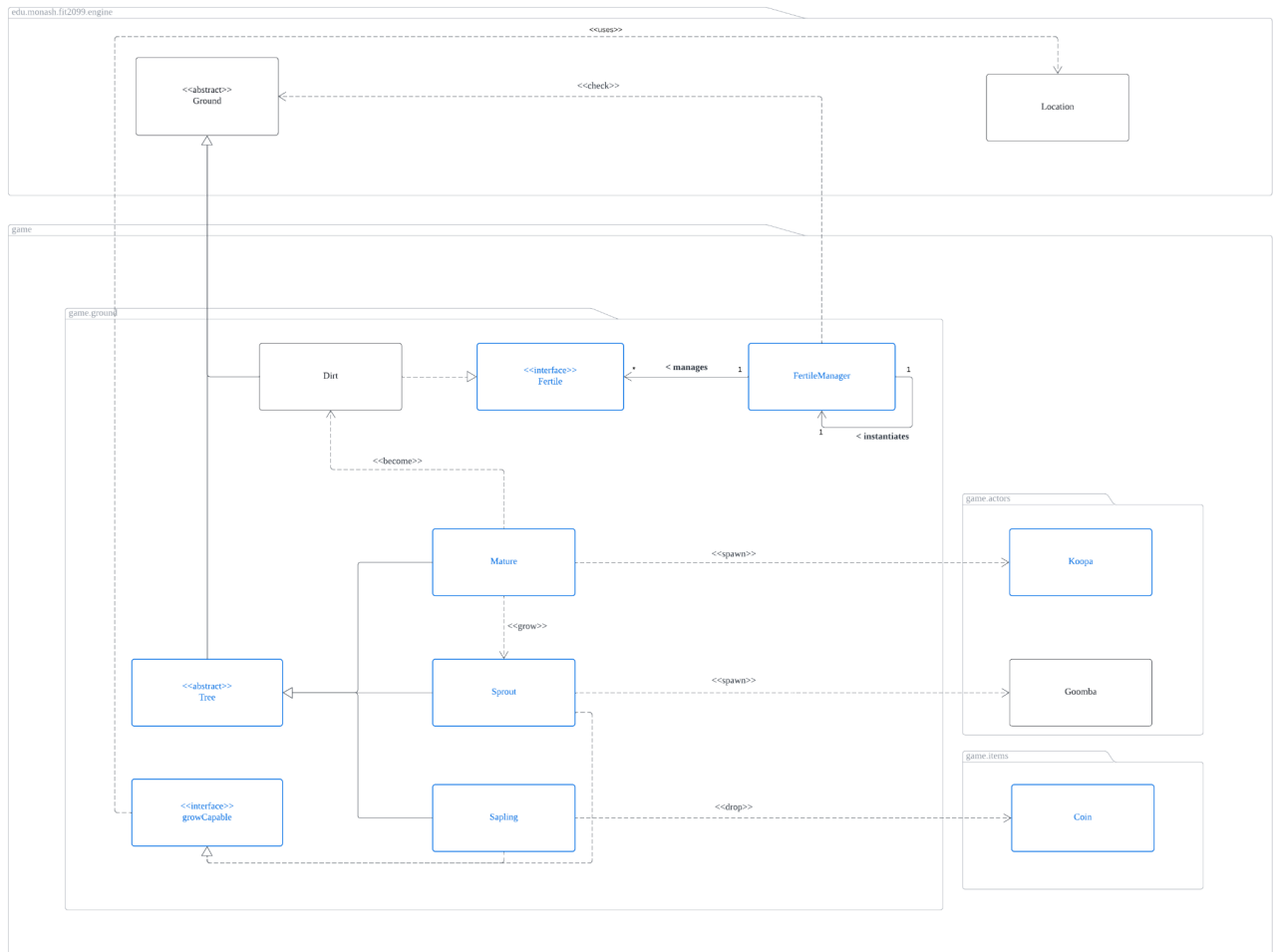
I accept this WBA. - George Tan Juan Sheng

I accept this WBA. - Tee Shun Yao

Design Documents

REQ1

Class diagram:



Requirement:

- Sprout & Sapling can grow into their next stage after they reach 10 turns
- Each stage of tree has their own unique spawning ability
 - Sprout: 10% chance spawn Goomba in every turn
 - Sapling: 10% chance to drop a coin (\$20) in every turn
 - Mature: 15% chance to spawn Koopa in every turn
 - Mature: Can grow a new sprout on surrounding fertile ground every 5 turns (if there is one)
 - Mature has a 20% chance to wither and die (become dirt) in every turn

Design Rationale:

In requirement 1, to represent the 3 stages of the tree, we have decided to change the **Tree** class into an abstract **Tree** class that is a subclass of **Ground**, we have also created 3 new classes which are **Sprout**, **Sapling** and **Mature** where all of them are the subclasses of **Tree** class.

The implementation of our design follows the **Open-Closed Principle (OCP)** because the **Tree** class is open for extension but closed for modification. Since there will be different spawning ability for each stage of the tree, there will also be different implementation of code. To tackle this, we would add an abstract method named `spawn()` in the **Tree** class so that every subclass can perform their own unique spawning ability. The design of abstract **Tree** class can ensure that whenever there is a new stage of tree which has another unique spawning ability, the **Tree** class would not be modified to add in new features.

To meet the requirement of unique spawning ability, we would override the `spawn()` method in all the subclasses to different code implementations since they will have different spawning ability. The design of `spawn()` method in each class is shown below:

spawn() in Sprout

1. Create a local variable named `goombaSpawnChance` and assign a value of 10 to it
2. Use `rand.nextInt(100) <= goombaSpawnChance` to check the 10% chance of spawning Goomba
3. Use `location.containsAnActor()` to check if there is an actor on the current location
4. If steps 2 & 3 returns true, then it would create an instance of Goomba and use `location.addActor(goomba)` to spawn the goomba on its position.

spawn() in Sapling

1. Create a local variable named `coinDropChance` and assign a value of 10 to it
2. Use `rand.nextInt(100) <= coinDropChance` to check the 10% chance of dropping coin
3. If statement 2 returns true, then it would create an instance of Coin (\$20) and use `location.addItem(coin)` to spawn the coin on its position.

spawn() in Mature

- **Spawn Koopa**

1. Create a local variable named `koopaSpawnChance` and assign a value of 15 to it
2. Use `rand.nextInt(100) <= goombaSpawnChance` to check the 15% chance of spawning Goomba
3. Use `location.containsAnActor()` to check if there is an actor on the current location
4. If steps 2 & 3 returns true, then it would create an instance of Koopa and use `location.addActor(koopa)` to spawn the koopa on its position.

- **Spawn sprout in surrounding fertile ground**

We would create a new interface called **Fertile** and a new class called **FertileManager** which aims to organise all the objects that implement the Fertile interface. For now, since the only class that is classified as fertile ground is **Dirt**, the **Fertile** interface is only implemented by the **Dirt** Class. In the constructor of every class that implements the **Fertile** interface, it will need to use the default `addToFertileManager()` method in the **Fertile** interface to add the object into ArrayList of Fertile in **FertileManager**.

The implementation of such design adheres to the **Dependency Inversion Principle (DIP)** as **FertileManager** does not depend on the low-level module such as **Dirt**. When new classes that implement Fertile are added in, we would not need to modify the **FertileManager** to keep track of all the objects that implement Fertile interface. This is because there is a layer of abstraction (**Fertile** interface) between all of

them and it can ensure that the high-level module and low-level module would not affect each other.

1. Use `(turn != 0 && (turn % 5 == 0))` to check every 5 turns requirement where `turn = super.getAge()`
2. Use for loop on `location.getExits()` to get every exits available at the current location
3. For every exit, we use `exit.getDestination()` to obtain the destination (location) of the exit.
4. For every destination, we use `destination.getGround()` to get the ground type of the current location.
5. For every ground, we check if the ground in that location is a fertile ground by using
`FertileManager.getInstance().getFertileGroundList().contains(ground)`
6. If the destination is a fertile ground, it will be added into a `ArrayList` of **Location** named `surroundFertileList`
7. After completing the for loop above, if there is any fertile ground around the mature, it will create a new object of **Sprout**, and use
`surroundFertileList(rand.nextInt(surroundFertileList.size()))`
to randomly choose one of the fertile ground (location) and use the `setGround()` method to set the ground type of that location to **Sprout**.

Besides, we have added some common attributes such as `age` in the `Tree` class to track the age of the tree so we know when the tree should grow into its next stage. This implementation complies with the **Don't Repeat Yourself Principle (DRY)** such that we do not need to keep declaring the same attribute in every subclass of **Tree**.

Moreover, we would create a new interface named **growCapable** to handle the growing of trees from one stage to another. This interface will be implemented by the **Sprout** and **Sapling** class only because they can grow into next stage such that

1. Sprout -> Sapling
2. Sapling -> Mature

In the `grow()` method from **growCapable**, we would override it in both classes that implemented it by creating a new instance of the class that will be the next stage of the tree, and pass it into the `location.setGround()` method to change the ground type of the current location. For instance, when the `grow()` method of **Sprout** is called, it will create a new Sapling object and pass it into the `setGround()` method which changes the ground type of the location from Sprout to Sapling that indicates the changing of stages of the tree.

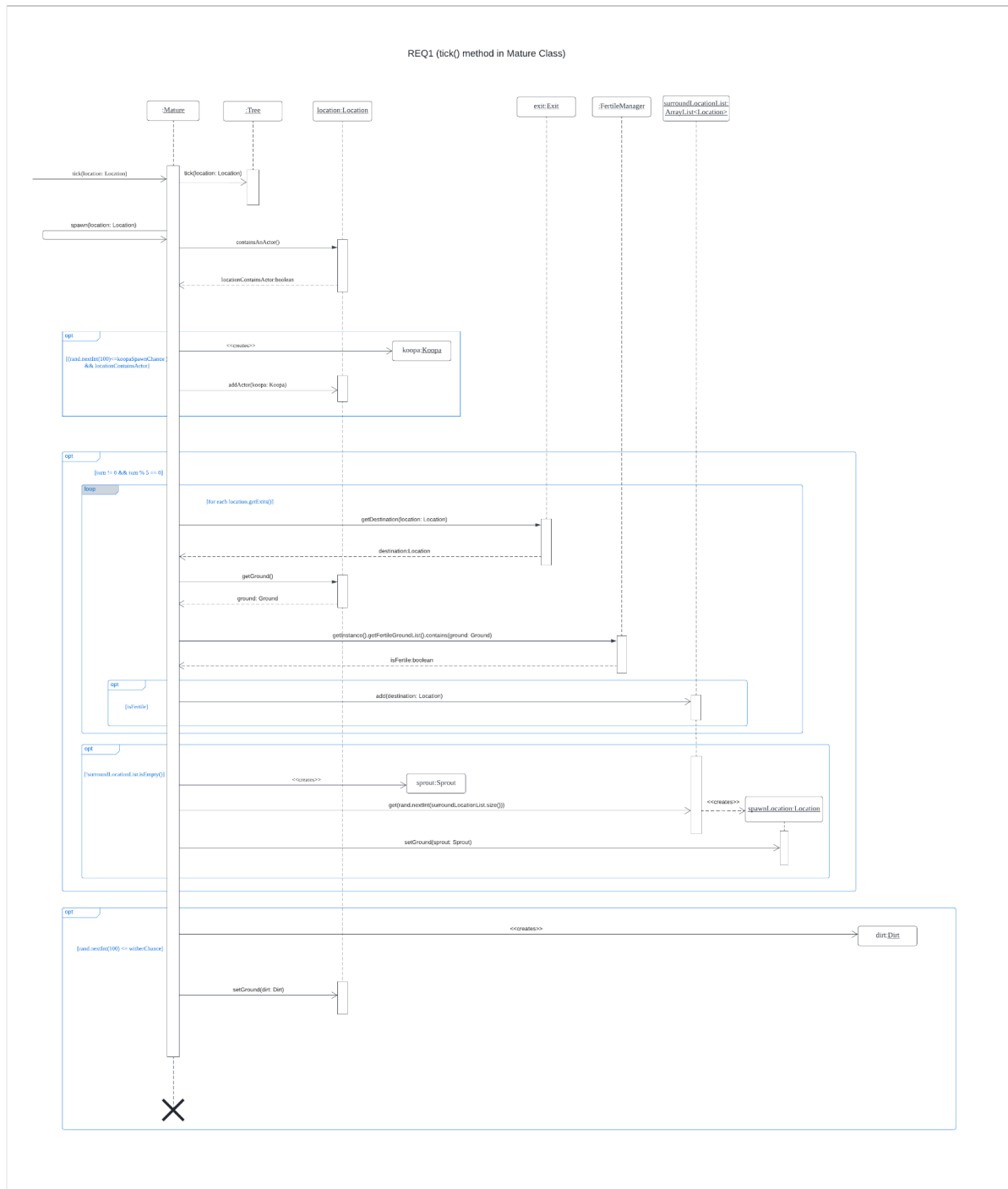
This design follows the **Liskov Substitution Principle (LSP)** as we are only implementing the **growCapable** interface in **Sprout** and **Seedling** class instead of making it as an abstract method in abstract **Tree** class. This is because sprout and sapling can grow into their next stage while mature cannot, if we put the `grow()` method as an abstract method in Tree class, it will break the LSP as there is a subclass that can't do what the base class can.

In addition, the design is in line with the **Interface Segregation Principle (ISP)** where interfaces should be smaller, and the client should not be forced to depend upon interfaces that they do not use. In our implementation, the **growCapable** interface only has one method which is `grow()` to indicate the growing of the tree to another stage. It is small enough and would not force the class that implements it to use something that they do not need to.

To implement the withering of Mature, we would use `(rand.nextInt(100) <= witherChance)` to check if it meets the 20% chance to be withered where the `witherChance` is a final attribute of Mature. If yes, then we would create a new Dirt object and use `location.setGround(dirt)` to change the ground type to Dirt.

Every method mentioned above such as `grow()` and `spawn()` will be implemented in the `tick()` method of their own class since the `tick()` method aims to let the location experience the passage of time, in other words, it is the turn of the game.

Sequence Diagram:



This sequence diagram will describe the operation of Mature in every turn whereby Mature would have 15% chance to spawn Koopa, grow a new sprout in one of its surrounding fertile squares randomly for every 5 turns and also have 20% chance to wither.

The starting point of the sequence diagram above is when the `tick()` method in the Mature class is being called. The `tick()` method will separate into 3 parts:

1. `super.tick()`

→ Firstly, it will call the `tick()` method from its super's class and perform any respective action without returning anything.

2. `spawn()` method

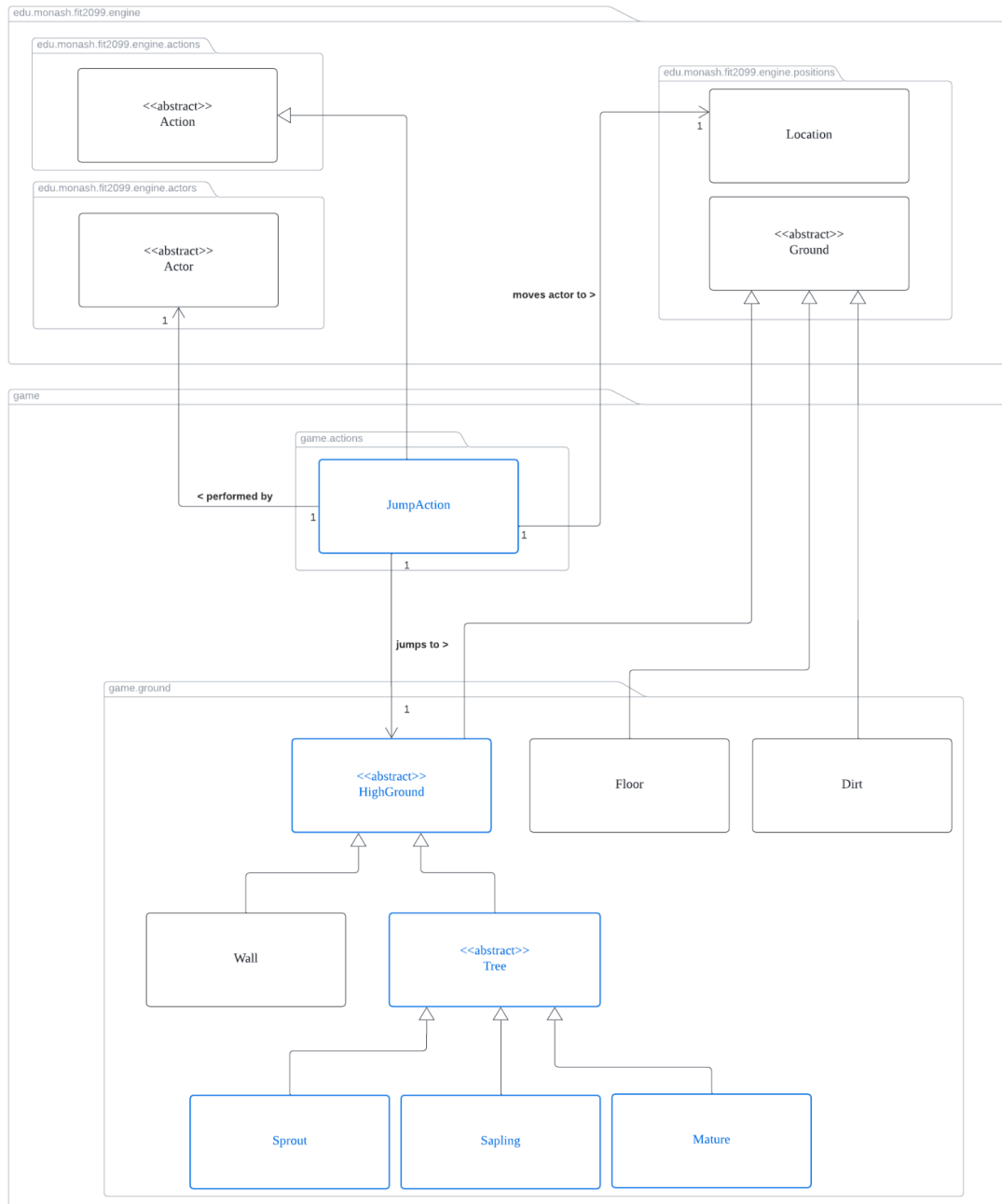
→ From then on, we will call the `spawn` method which is located in the Mature class itself. For more detailed information about the code in `spawn` method in the sequence diagram, please refer to the **spawn sprout in surrounding fertile ground** section in the design rationale of REQ 1.

3. 20% chance of wither

→ In this part, it will check whether it meets the 20% chance to wither. If yes, it will create a **Dirt** instance and pass it as the method argument into `location.setGround(dirt)` to change it to dirt which indicates that the mature has withered.

REQ2

Class Diagram:



Requirement:

A Super Mario game will not be complete without a **"jump"** feature. When the actor is standing next to the high ground, the actor can jump onto it. Going up requires a jump, but going down doesn't require it (the actor can walk normally). Each jump has a certain success rate. If the jump is unsuccessful, the actor will fall and get hurt. The success rate and fall damage are determined by its destination ground as listed below:

- Wall: 80% success rate, 20 fall damage
- Tree:
 - Sprout(+): 90% success rate, 10 fall damage
 - Sapling(τ): 80% success rate, 20 fall damage
 - Mature(⊔): 70% success rate, 30 fall damage

Design Rationale:

In requirement 2, we are required to create the feature "jump" which allows the actor to jump to the place the actors cannot move into. We would create a **JumpAction** class, a **HighGround** abstract class and modify **Tree**, **Mature**, **Sapling**, **Sprout** and **Wall** classes.

First, the **HighGround** abstract class is created because there are many grounds which the actors cannot move into. Each highGround has their own value of jumpSuccessRate and fallDamage. In our implementation, highGround will extend ground. Two attributes jumpSuccessRate and fallDamage will be declared in highGround class, as well as their setters and getters. These values are to be defined in its subclass as different grounds have different values. In the class, we override the canActorEnter method in Ground class and make the method return false as all highGround are not available for actors to move in, actors should jump onto it. Also in the highGround, we would implement a getHighGroundName which would return a string value of the name of the ground which would be used in displaying the result of jumpAction. Next, we have allowableAction overridden from Ground class. This allowableActions method would add a jumpAction to ActionList if the actor is a player. Parameters passed to the method include actor, direction, location and highGround. This implementation is following **Open-Closed Principle (OCP)** as by having a highGround abstract class, it is very easy to extend another type of highGround like rooftop, without changing the existing

code. By using abstraction, we reduce the amount of effort to add new highGround in the future. The new type of highGround will have the same attributes jumpSuccessRate and fallDamage along with their setters and getter and allowableAction methods. This implementation also follows **Single Responsibility Principle (SRP)** as highGround only contains the attributes and methods which are needed. It does not implement any other new things which are not necessary for a highGround. This implementation follows the **Don't Repeat Yourself Principle (DRY)** because we do not have to repeat creating a setter and getter for attributes in different types of highGround.

We are going to discuss Tree, Mature, Sapling, Sprout and Wall in this paragraph. Tree is an abstract class extending HighGround. Mature, Sapling and Sprout are extending Tree, which is implemented in requirement 1. Wall is a class that extends HighGround. Tree class is left unchanged except extending the class. Meanwhile, sapling, mature and sprout override the method from HighGround. Two methods setFallDamage and setJumpSuccessRate are called during the initialization of sapling, mature and sprout via constructor. The values set are following the requirement, which are: 1) sprout – FallDamage:10, SuccessRate:90 2) sapling – FallDamage:20, SuccessRate:80 3) mature – FallDamage:30, SuccessRate:70. They also override the getGroundName methods which would return “sprout”, “sapling” and “mature” respectively. Similar implementation goes to wall class which directly extends HighGround class.

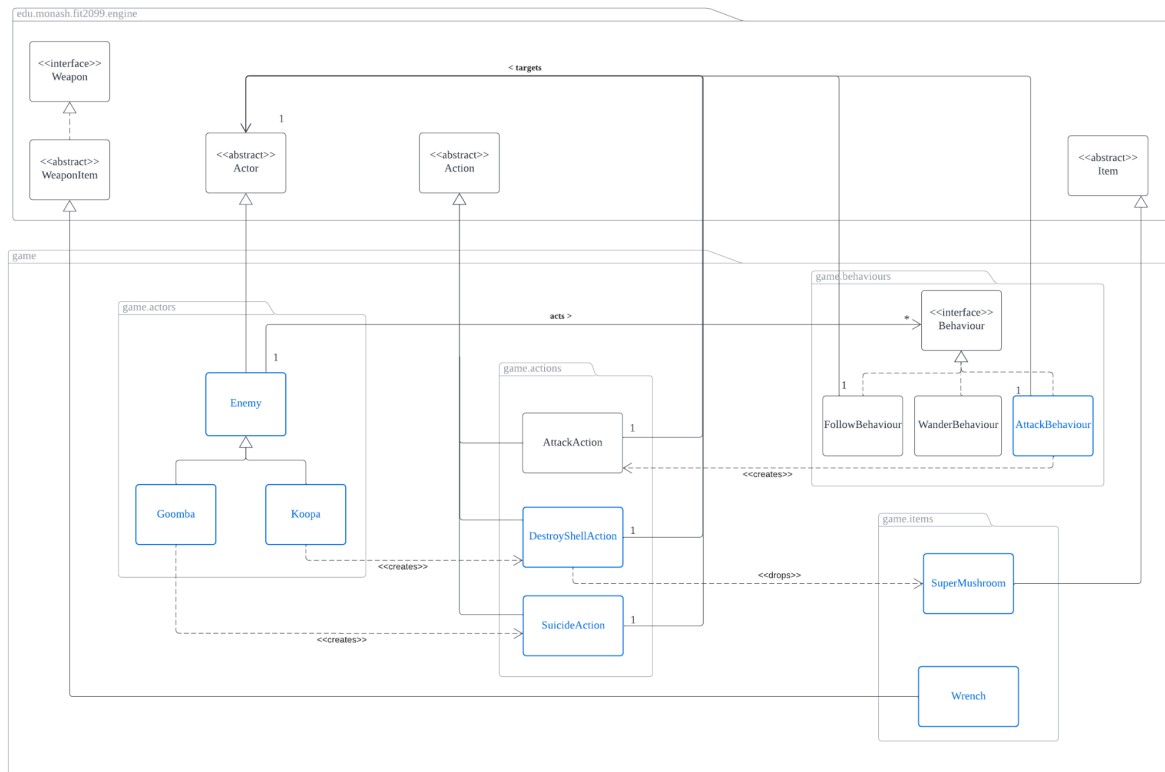
Wall will override the method to set fallDamage:20, SuccessRate:80, and return “wall” string. We are choosing this implementation as the reason mentioned above. New and existing highGround can be added and managed easily because they all inherit the same class.

In this section, we are looking at the JumpAction implementation. JumpAction is extending Action class. In jumpAction, it has 5 instance variables: 1) actor 2) direction 3) destination 4) highGround 5) random. We first create a constructor which takes 4 parameters and initialises them, including actor, direction, destination and highGround. Next we would override the execute method from the Action class. In this method, we would first get the jumpSuccessRate and fallDamage using getJumpSuccessRate and getFallDamage methods of highGround. Next, using the random which is previously initialised, we generate a number below 100 and see if the number is bigger than the range of jumpSuccessRate. If it is bigger, it means the jump action fails. We would

reduce actor hit points by fallDamage value using the hurt method in Actor class and print a message saying the actor fails the jump and the damage received. We then check if the player is still alive using isConscious method in Actor. Message of "actor is dead" would be printed if the player dies. If the player succeeded in the jump, we would move the actor to that location using the moveActor method in GameMap class and print a message saying that the jump is successful. The other method we override is menuDescription which returns a string that "actor jumps to <<direction>>". This is used to let the users know what options they should choose in the menu if they want to jump to a certain direction.

REQ3

Class diagram:



Requirement:

- Both enemies, Goomba and Koopa would be able to automatically attack players and follow players once engaged in a fight
- Unconscious enemy will be removed from map (except Koopa that is in dormant state) and enemies can't enter Floor
- Goomba has a 10% chance to suicide and be removed from map in each turn
- Koopa will go into dormant state once it becomes unconscious and will do nothing until its shell gets destroyed
 - Koopa's shell can only be destroyed when player has wrench
 - Destroying Koopa's shell would drop a Super Mushroom

Design Rationale:

In Requirement 3, there are a few things we need to implement. The first thing that we need to implement is to create our two enemies, which are **Goomba** and **Koopa**, and we have to make these enemies be able to have behaviours such that they will automatically attack and follow **Player** once the enemy is engaged in a fight. Since we already have FollowBehaviour implemented for us, we would only need to implement the AttackBehaviour class. Upon creating an AttackBehaviour, we would need to set the target and direction as ultimately, these two attributes would be used as we override `getAction` in AttackBehaviour, where `getAction` would check if both enemy and player exists in the map, and also check the distance between the player and enemy and see if its in range, if all these conditions are fulfilled, `getAction` would return an AttackAction that is targeted towards player with the respective direction.

Also, from this we can actually notice how both enemies Goomba and Koopa will have this same behaviour, thus we can create an **Enemy** abstract class that extends Actor, and this abstract Enemy class would be extended by both Goomba and Koopa. Thus, any actors that extend the Enemy class would be able to automatically attack and follow the player once the enemies are engaged in a fight. In terms of how we plan to implement the Enemy abstract class, upon instantiation of the Enemy object, the Enemy would be added a WanderBehaviour as all enemies would wander around. We would also override the `allowableActions()` method such that when a Player is getting the allowable actions that can be done on the enemy, not only an AttackAction that is targeted to the enemy would be returned, but that enemy would also be added a FollowBehaviour and AttackBehaviour that will both target towards the player. This is because whenever a Player is in range to get the allowable actions that can be done on Enemy (Enemy is at one of the locations of Player's exits), Enemy would also definitely be in range to be able to attack Player and then follow Player, hence this is why this logic would work. In addition to that, when adding FollowBehaviour and AttackBehaviour, we would add AttackBehaviour with a higher priority (which is a lower Integer as according to the codes, lower Integer corresponds to higher priority since we look at the behaviours according to its key in ascending order) compared to FollowBehaviour, and FollowBehaviour would have a priority higher than WanderBehaviour. Assuming these three behaviours have been added, during `playTurn`, Enemy would always look at AttackBehaviour first to get the action, if the player exists and is in range to attack, Enemy would get the AttackAction, if player does not exist or enemy is not in range to

attack, Enemy would get action from FollowBehaviour and follow the Player instead, and if the Enemy reaches a point where it is not able to follow Player anymore (reach places that Enemy can't enter or moving anywhere would make the distance between Enemy and Player even further), Enemy would get action from WanderBehaviour and just wander around.

By having such an Enemy class, all enemies would automatically be able to wander around as well as automatically attack and follow players once engaged in a fight, this follows the **DRY principle** as if we do not have such an Enemy class, each Goomba, Koopa or other enemies in the future would all have to add a WanderBehaviour upon instantiation and also override allowable actions to add FollowBehaviour and AttackBehaviour when Player is in range (just like how we did for Enemy class), essentially we would just be repeating the same code. Hence, by having such Enemy class we would avoid repeating the same code again and again. For this implementation, I chose to do it this way because I initially wanted to override and make an allowable actions method in player, where this method would add the FollowBehaviour and AttackBehaviour to the enemies when enemies are trying to get the allowable actions that can be done on player, but I realised how this could be troublesome as it would involved adding new capabilities like HOSTILE_TO_PLAYER to check if its an enemy, hence I feel like adding the behaviours directly to enemy when player is getting allowable actions that can be done on enemy is much easier as the logic is indeed the same same, just that no new Enum attributes for Status has to be created and we would not have to override the allowableActions method in player.

For the implementation of enemies cannot enter **Floor**, this can be done by overriding the canActorEnter method in Floor and just check if the actor has the capability `Status.HOSTILE_TO_ENEMY`, if the actor does not have this capability, return false. This is because if the actor does not have this capability, it must be either Enemy or Toad, and since Toad can't move, setting canActorEnter to return false when the actor is enemy or toad, would only really affect Enemy, hence this implementation would work.

Moving on to the details of **Goomba** class, as mentioned early Goomba would extend Enemy and its constructor would call super (Enemy's constructor) with the correct attributes of Goomba as its name, 'g' as its display character and 20 as its hitpoints. We would also override the `getIntrinsicWeapon()` to return a new

IntrinsicWeapon that kicks with 10 damage, and since by default intrinsic weapons would have a hit rate of 50%, we would not need to do any other modifications. For Goomba's suicide feature, we decided to create a new action for this, which is called

SuicideAction. For each Goomba's turn, we could just use `(rand.nextInt(100) <= 10)` to calculate the 10% chance, if we do generate an Integer lesser or equal to 10, Goomba would return a SuicideAction which will be executed. This SuicideAction basically would just check if Goomba is still in the map and if it is, Goomba will be removed from the map and prints out a message saying Goomba has suicided. The reason why we decided to create a new action for suiciding is because we wanted to follow the **Single Responsibility Principle (SRP)** as we wanted a single action to only handle one single scenario.

Next, for **Koopa**, similar to Goomba, Koopa would extend Enemy and its constructor would call super with the correct attributes of Koopa as its name, 'K' as its display character and 100 as its hitpoints. We would also override the `getIntrinsicWeapon()` to return a new IntrinsicWeapon that punches with 30 damage, and since by default intrinsic weapons would have a hit rate of 50%, we would not need to do any other modifications. Upon instantiation, Koopa would also add a capability called `Status.NOT_DORMANT`, which essentially is used to check if Koopa has gone to dormant state or not. Whenever Koopa is damaged by player via AttackAction, in AttackAction we would check if Koopa is unconscious and has capability `Status.NOT_DORMANT`, if yes, we would remove Koopa's capability `Status.NOT_DORMANT` and add a new capability `Status.DORMANT` for Koopa. This means that Koopa is defeated and will enter to dormant state. Since the display character for Koopa has to change to D when Koopa is in dormant state, we would need to also override the method `getDisplayChar()` in Koopa to return 'D' if Koopa has the capability `Status.DORMANT`, else return `super.displayChar()` which would return 'K'. Besides, since Koopa also has to stay on the ground and not do anything when in dormant state, we also need to check if Koopa has capability `Status.DORMANT` in `playTurn()` method, if yes, we would return a new `DoNothingAction()` so that Koopa doesn't do anything and just stay where it is at. This works because we would check if Koopa is in dormant state before actually looking at its behaviours to decide what action it would take, and if it is in dormant state, Koopa would just return a new `DoNothingAction` without looking at its behaviours.

Moving on to the actions that can be done on Koopa, if Koopa has the capability `Status.NOT_DORMANT`, Koopa's `allowableActions()` method would just return the list of allowable actions that can be done on Enemy (which is basically just calling `super.allowableActions()`), which essentially allows player to attack Koopa and allows Koopa to attack and follow player. But if Koopa has the capability `Status.DORMANT`, we would need to check if the player has a wrench to destroy koopa's shell. This can be done by creating a Wrench class that extends `WeaponItem` and ultimately, Wrench would add a capability `Status.DESTROY_SHELL`, this means that whenever a player has wrench in its inventory, the player would have the capability `Status.DESTROY_SHELL`. If Koopa is in dormant state and the actor has capability `Status.HOSTILE_TO_ENEMY` and `Status.DESTROY_SHELL`, Koopa's `allowableActions()` method would return an `ActionList` consisting of only one action, which is `DestroyShellAction`. `DestroyShellAction` basically removes Koopa from the map and spawns a new `SuperMushroom` at its location. In this sense, we would get the expected output as players would not get the option to attack Koopa if it is in dormant state because of Koopa is in dormant state, a destroy shell option would also only appear if the player has a wrench in inventory (either by picking up/buying wrench). If the player has no wrench, players can't do anything when Koopa is in dormant state, and when Koopa is not in dormant state, players would have options to attack Koopa just like for any other enemies. Again, we specifically created this `DestroyShellAction` as we wanted to define an action that solely allows players to destroy Koopa's shell, this follows the **SRP**. We could also implement the destroy shell in `attackAction` but this would violate **SRP**.

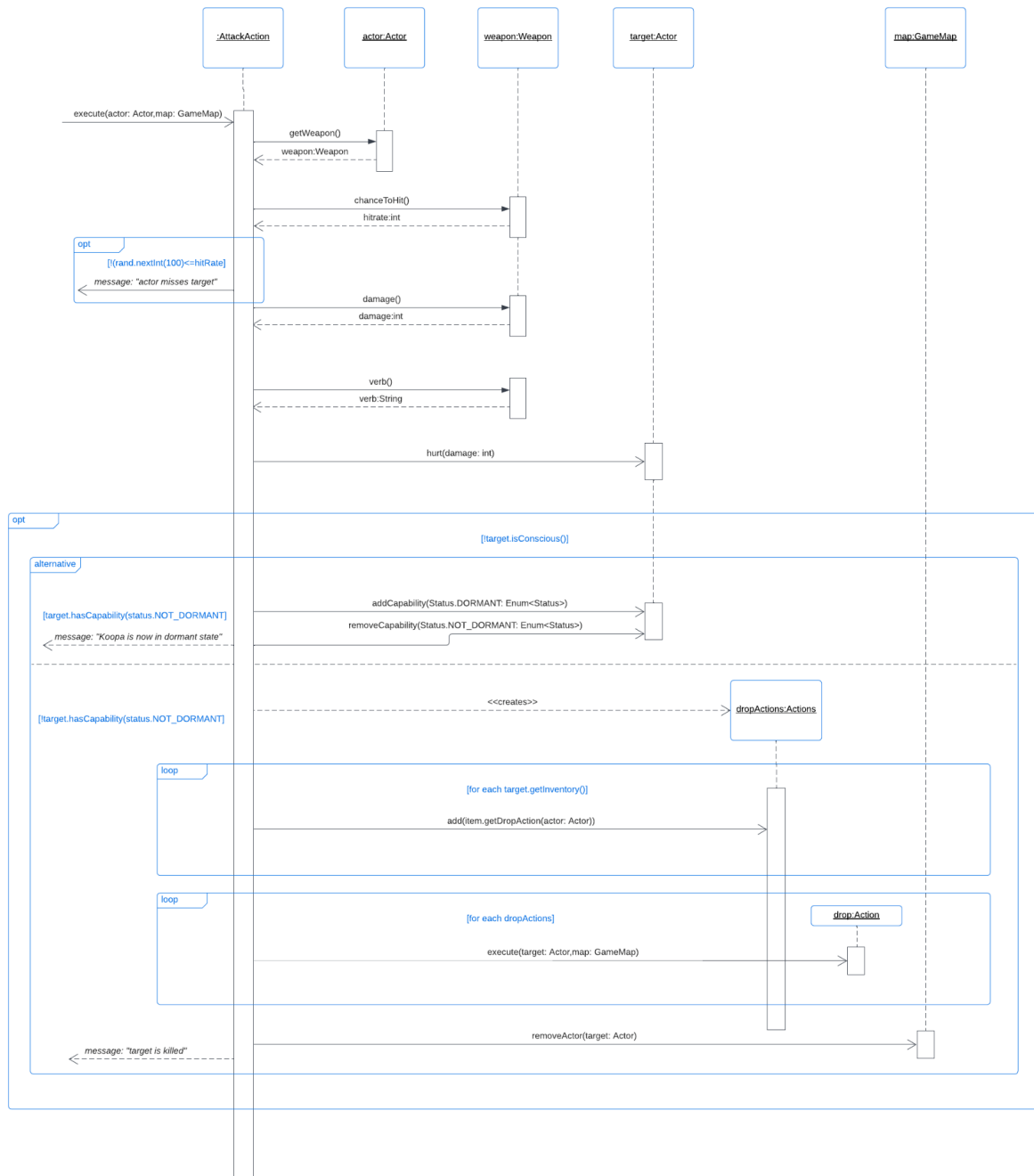
However, one may argue why do we need both `Status.DORMANT` and `Status.NOT_DORMANT`, wouldn't an actor that does not have capability of one of the status would technically mean otherwise? Well, if we only had `Status.DORMANT`, a bug would happen in `AttackAction` as normally we would check if the target is unconscious, we would then check if it has capability `Status.NOT_DORMANT`, if yes (this means target is a Koopa), we would remove capability `Status.NOT_DORMANT` and add capability `Status.DORMANT` to signify that koopa enters dormant state. But if we do not have `Status.NOT_DORMANT`, upon knowing that target is unconscious, we would need to check if target does not have capability `Status.DORMANT`, but there may be cases

where the target is not Koopa and then it would definitely not have capability `Status.DORMANT`, and that target would then be added capability `Status.DORMANT`, which is not what we want as we only want Koopa to enter dormant state when its unconscious. Hence, this was why I decided to have both `Status.NOT_DORMANT` and `Status.DORMANT`.

All in all, these would be all the changes made to the existing classes as well as newly created classes in order to implement all of the requirements needed for REQ3. To sum it all up for each classes' responsibilities, Goomba and Koopa would extend a newly created abstract class `Enemy` and any `Enemy` object would be able to automatically attack and follow players once engaged in fight as we add the behaviours to `Enemy` in `Enemy's allowableActions`. The reason for adding it only in allowable actions is because this signifies that the behaviours are only added when the player is in range with the enemies. Goombas would also have a 10% chance of executing `SuicideAction` which removes it from the map. Koopas would enter dormant state once they are unconscious and they would require players to have a wrench in order to access `DestroyShellAction` which can be executed to remove Koopas from map and create a `SuperMushroom` at that Koopa's location.

Sequence Diagram:

Sequence Diagram 1

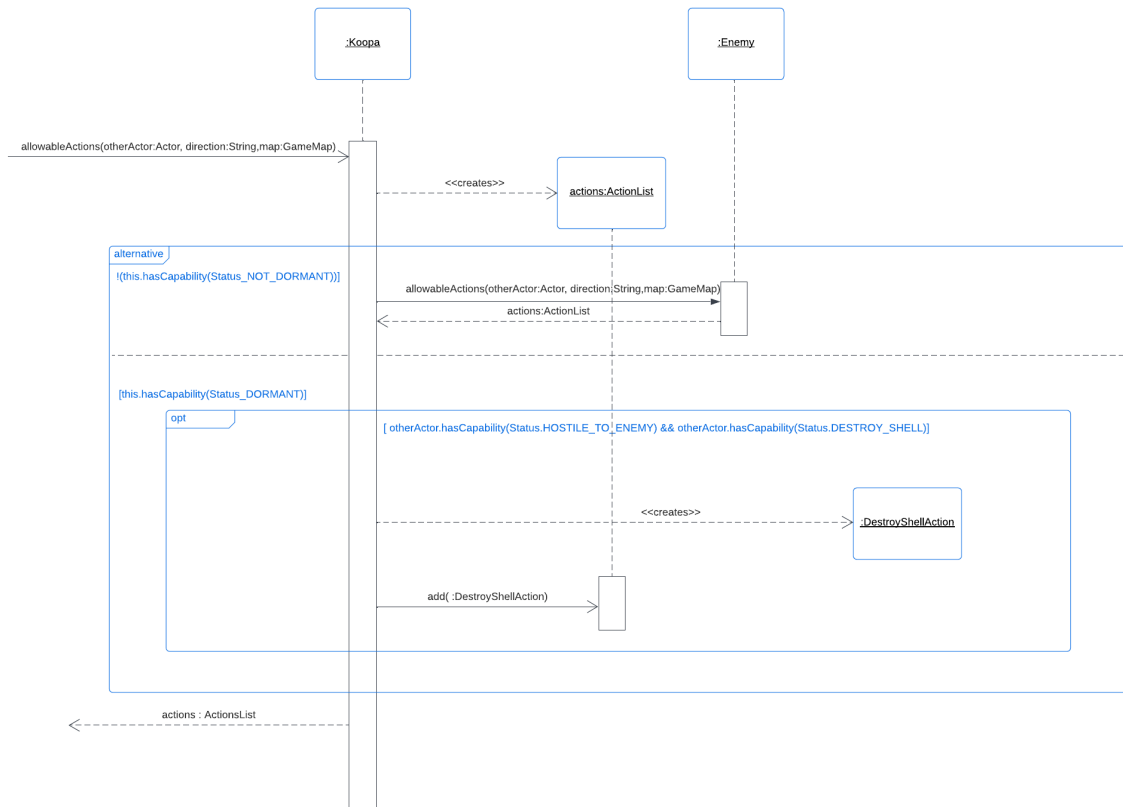


This sequence diagram's objective is to show the operation where how Koopa will be given the capability `Status.DORMANT` if Koopa is unconscious upon being attacked. The starting point of the sequence diagram above is when the `execute()` method in the **AttackAction** class is being called. Basically, the `execute()` method will separate into 2 parts:

1. Check the chance of successfully hitting
 - It will check whether it meets the hitting chance of the weapon held by the actor.

- If yes, it will proceed to the next part of the code
 - If not, it will just return a String message and terminate this method.
2. This part of code is basically the code that will be executed if there is no special condition.
- Retrieve the damage of the weapon
 - Hurt the target with the damage retrieved above
 - Check whether the target is conscious or unconscious
 - Conscious: proceed to the next part of code
 - Unconscious:
 1. Check if the target has capability of `NOT_DORMANT` (check whether the target is koopa)
 2. If yes, then change the status of koopa into `DORMANT`
 3. If not, drop all the items on the target on the current location that the target is on and remove the target from the map (target is killed)
 - Return the String message and terminate the method

Sequence Diagram 2:

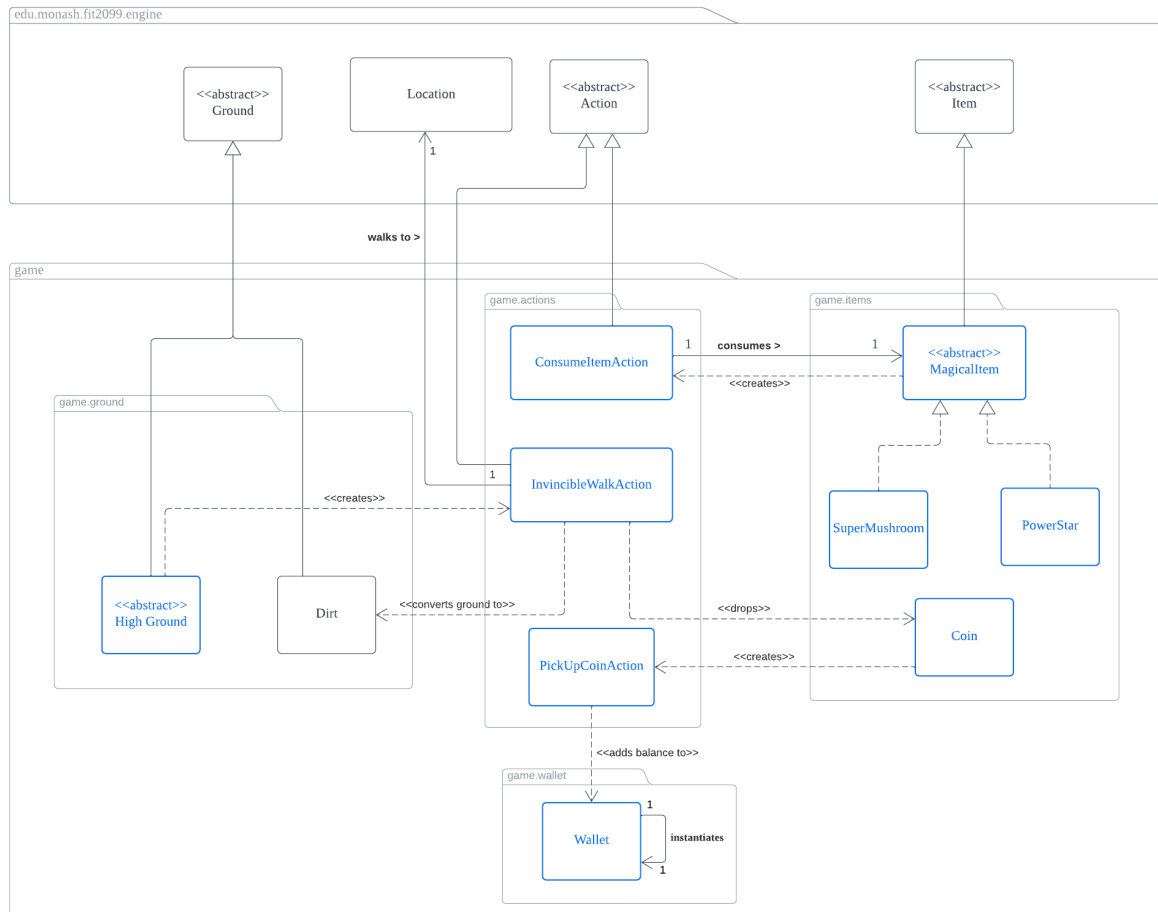


This sequence diagram would demonstrate the operation of the `allowableActions` of Koopa whereby Koopa would either allow actors to attack it as usual or allow actors to be able to destroy its shell or allow actors to not be able to do anything to it.

1. Create a new `ActionList` called `actions` as this would be the object we would be returning.
2. We would then check if Koopa is in dormant state or not, if not, it means actors are able to attack it as usual, just like any other enemies, hence we would assign actions to `Enemy's allowableActions`.
3. If Koopa is in dormant state, we would need to check if the `otherActor` is player (has capability `Status.HOSTILE_TO_ENEMY`) and if the player has a wrench (has capability `Status.DESTROY_SHELL` as if a player has a wrench it would add this capability to player). If yes, a `DestroyShellAction` would be created and this action will be added into the `ActionList actions`.
4. In the end, the `ActionList actions` would be returned. In some cases, the `ActionList actions` might be null when Koopa is in dormant state and `otherActor` is not play or `otherActor` does not have a wrench, this follows our requirements as actor would not be able to do anything to a Koopa that is in dormant state if the actor does not have a wrench.

REQ4

Class Diagram:



Requirement:

- Both magical items, Super Mushroom and Power Star would buff actor once consumed
- Super Mushroom increases actor's max HP by 50, mario's display character turns to M and actor can jump freely with 100% success rate and receive no fall damage
 - Super Mushroom buff wears off if actor gets damaged but max HP stays
- Power Star grants actors an invincible effect where it heals the actor by 200 HP and allows the actor to not need to jump to high grounds and can walk normally. Power Star also grants actor immunity and actors are also able to instantly kill enemies.

- High grounds being stepped on would be converted to Dirt and a coin with \$5 would be dropped
- PowerStar would fade after 10 turns if not consumed, and if consumed, Power Star's invincible effect would only last for 10 turns.

Design Rationale:

For Requirement 4, we would be required to create and implement the only two **Magical Items** we would have so far, which are **Super Mushroom** and **Power Star**, thus, everything covered in this requirement would be related allowing players to consume these magical items as well as buffing the players according to which magical item consumed and also the after effects after consuming the magical items. Since both Super Mushroom and Power Star are able to be consumed by the player to have some buffs to the player, we created an abstract class **MagicalItem** that extends **Item** class and this abstract class would have an abstract method called `consumeItem`, which defines how the actor would be buffed upon consuming the magical item.

SuperMushroom and **PowerStar** class would both extend **MagicalItem** and need to override `consumeItem` to add in how the actor would be buffed.

We would now first talk about how we would implement the buffs for both SuperMushroom and PowerStar. Upon consumption for **SuperMushroom**, the actor's maximum hit points would be increased by 50 and the actor would add a capability `Status.SUPER`. By having this capability `Status.SUPER`, we can make the display character of mario to turn from `m` to `M`. This can be easily done in player's `getDisplayChar()` where we just check if the player has capability `Status.SUPER`, if yes, return an uppercase version of player's initial display char(which is `m`), if not, just return the initial display char. In addition to having this capability, we can also implement it so that the player can jump freely onto **HighGround** objects with a 100% success rate and no fall damage, where inside our defined **JumpAction** in REQ2, it would check if actor has capability `Status.SUPER`, if yes, we would just directly move the actor the respective location and not hurt the actor, so that the actor will always jump successfully and receive no fall damage when actor has capability `Status.SUPER`. Other than that, we also need to make sure that mario loses the Super Mushroom buff if it ever gets

damaged by an enemy, hence we would need to check in `AttackAction` where if target has capability `Status.SUPER`, remove that capability so mario loses the buff and gets converted back to its original display char. By having these implementations, the buffs of Super Mushroom for actors would be done as per the requirements. Also, I have noticed that the base code has `Status.TALL` which tells that the current instance has grown, and this `Status.TALL` is used in the player's class when getting its display character. I have decided to change `Status.TALL` to `Status.SUPER` as I feel like it's more meaningful as we would only have an uppercase letter M for the player's display character once we consumed a Super Mushroom.

For **PowerStar**, the actor who has consumed it would be healed by 200 hit points and be added a capability `Status.INVINCIBLE`. By having this capability `Status.INVINCIBLE`, actors we can allow actors to not need to jump to higher level grounds anymore and just walk normally, while any higher level ground that is stepped on would be converted to dirt and an a coin of \$5 value would be spawned at that ground's location. We could of course add if and else statements in `JumpAction` to move the actor as well as set the ground as dirt and create a new coin if the actor has capability `Status.INVINCIBLE`, however, we took consideration regarding the **Single Responsibility Principle (SRP)** and we feel like if we implement it this way, `JumpAction` would be having more than one responsibility, which is essentially allowing actors to be able to jump on **High Ground** objects and also walk over High Ground objects, thus we feel like this implementation would violate SRP. Hence, we decided to create a new action for actors to walk over High Ground objects, which is called **InvincibleWalkAction**. In High Ground, we would override `allowableActions` and we would check if the actor has capability `Status.INVINCIBLE`, if yes, it would return the `InvincibleWalkAction`, if not, it would return the `JumpAction`. Under `InvincibleWalkAction`'s `execute` method, we would call our existing `MoveActorAction` method to move the actor to the chosen location. We would also set the ground of that location and set it to a new `Dirt` object. A new coin object of value \$5 would also be created and added to that location. The reason why we called our existing `MoveActorAction` method is we wanted to follow the **DRY principle**, where we don't repeat the code of moving the actor from one place to another, as it is already defined in the `MoveActorAction` and there is no point writing it twice.

Moving on, enemies would also deal no damage to the invincible actor (grants immunity) and the actor would be able to instantly kill any enemies if the actor doesn't miss (including Koopas that are in dormant state). In terms of the immunity buff upon consuming Power Star, we would check if the target has capability `Status.INVINCIBLE`, if yes, the damage dealt would be set to 0. For being able to instantly kill enemies if the actor does not miss, we would just need to check if the actor misses in the `AttackAction`, if it does not miss and the actor has capability `Status.INVINCIBLE`, the target is removed immediately from the map. We would also check if the target has capability `Status.DORMANT` or `Status.NOT_DORMANT`, this is to check if the instantly killed actor is a Koopa or not, if yes we would also spawn a Super Mushroom at that target's location. We would also override in Koopa's `allowableAction` to return `super.allowableActions` when the actor has capability `Status.INVINCIBLE`. Hence, by having all these implementations, the buffs of PowerStar upon consumption would be done as per the requirements.

Moving on, we would talk about how we would implement the action to consume these magical items. We have two scenarios for the two magical items, the first scenario is when the magical items are on the ground, and we consume it when we stand on top of them, the second scenario is when we purchase magical items from Toad and the items are added into the mario's inventory. However, regardless of which scenario we are facing, we could create an action called **ConsumeItemAction** which basically allows us to consume any magical items. Remember the `consumeItem` method that was required to be implemented by every subclass of `MagicalItem`? This `consumeItem` method would be used in `ConsumeItemAction`'s `execute` method whereby we would call the `consumeItem` method of the magical item onto the actor to provide the buffs to the actor. After that, we would then check whether the magical item is on the ground that the actor is positioned at or is it in the actor's inventory, if it is the former, the magical item would be removed from the ground, if it is the latter, the magical item would then be removed from the player's inventory. The reason why we implemented this `ConsumeItemAction` in this way is because if in the future we have many more magical items, we could still use this `ConsumeItemAction` to consume those magical items as those newly added magical items would just need to implement the `consumeItem` method which defines how they would buff/debuff the player once consumed. This follows the **Open-Closed Principle (OCP)** where we are not modifying our current system and we are just extending our system by adding in new magical items in the

future. This also relates to the **Dependency Inversion Principle (DIP)** as our system would not depend on the concrete classes of magical items such as Super Mushroom and Power Star, instead it depends on abstraction where it just depends on the abstract class `MagicalItem`. In a way, I would see that OCP and DIP are related as when our system doesn't depend on low-level modules and depends on abstraction, such systems that depend on abstraction are generally very easy to extend and would not modify the current system.

Last but not least, since `PowerStar` will fade and it would also have limited invincible effect, we could calculate these turns in `PowerStar`'s `tick` method. As mentioned earlier, magical items have two scenarios, whether it can be on the ground or in the inventory of the player, hence we would need to override both `tick` methods (one for when item on ground and another for when item in inventory). `PowerStar` would by default have 10 turns left to signify the turns left before it fades, each tick would then decrease then turns left by 1. Since this implementation only uses one turns left counter, if the actor ever consumes the magical item, the turns left counter would be reset back to 10 to signify that the 10 turns left is for the invincible effect to wear off. If the turns left counter ever reaches 0, we check if the magical item has been consumed or not, if it has not, then we can just remove the magical item (either remove from ground or remove from the actor's inventory). If the turns left counter reaches 0 and the magical item has been consumed, this means that the invincible effect has worn off and we would just need to remove the capability `Status.INVINCIBLE` from the actor. The reason why we do not need to remove the magical item either from ground or actor's inventory if the player has consumed the magical item is because when the magical item has been consumed, the magical item would already be removed from either the ground or the actor's inventory.

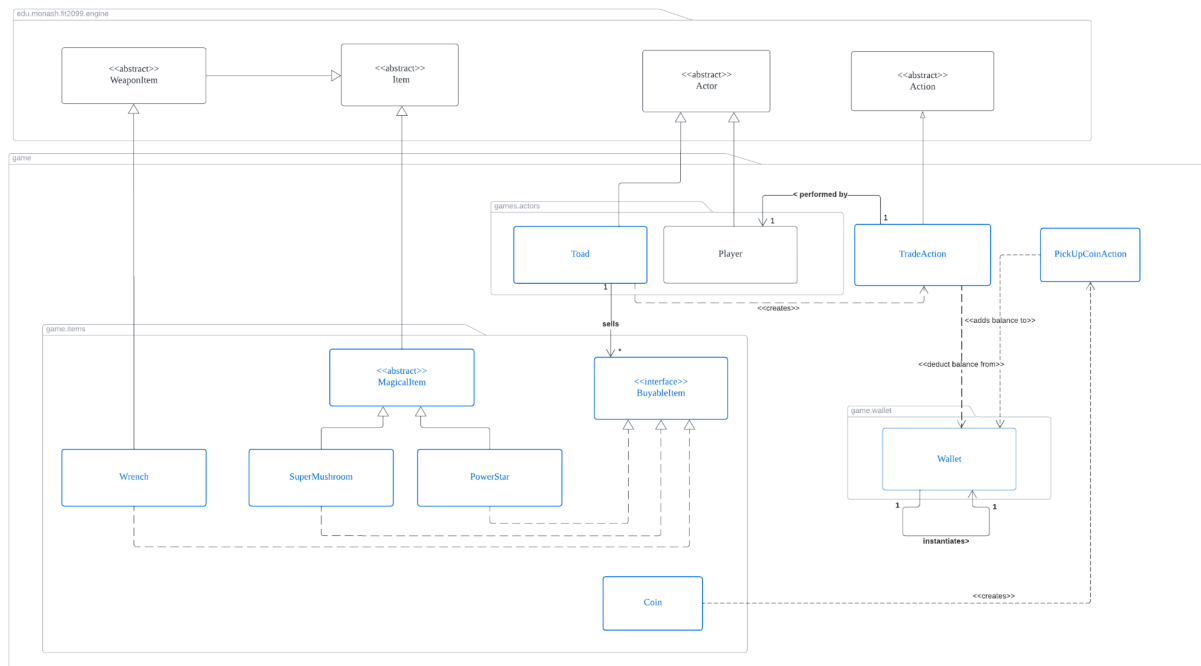
Besides, inside the `PowerStar` method, we also would override the `toString` method so that it returns `"Power Star (<turnsleft> turns left)"`. The reason for doing this is so that it would be very convenient to get `Power Star` along with the turns left so that it could be easily used in `ConsumeItemAction`'s `menuDescription` method as when we print `magicalItem`, we would automatically get `Power Star` along with how many turns left to consume it if the magical item is a `Power Star`, and ultimately this would give us the right sentence in the menu for user to know how many turns left is there for the user to consume power star. All in all, by implementing all these classes

along with its definitions, we would be able to fulfil the requirements for Requirement 4 and the system would work.

REQ5

Class Diagram:

REQ5



Requirement:

The coin(\$) is the physical currency that an actor/buyer collects. A coin has an integer value (e.g., \$5, \$10, \$20, or even \$9001). Coins will spawn randomly from the Sapling (τ) or from destroyed high grounds. Collecting these coins will increase the buyer's wallet (credit/money). Using this money, buyers (i.e., a player) can buy the following items from Toad:

1. Wrench: \$200
2. Super Mushroom: \$400
3. Power Star: \$600

Toad (○) is a friendly actor, and he stands in the middle of the map (surrounded by brick Walls).

Design Rationale:

In requirement 5, we are required to implement the functionality of trading. When the player is beside Toad, players are able to buy Wrench, superMushroom or PowerStar. Few classes and one interface are created or modified here, which are

TradeAction, **PickUpCoinAction** extending Action class, **Wallet** and **Toad**, **Coin**, **PowerStar**, **SuperMushroom**, **Wrench** classes, and **BuyableItem** interface.

We would first discuss BuyableItem, PowerStar, SuperMushroom and Wrench. BuyableItem is an interface for all the items which the actor can buy from toad. In this interface there is only one abstract method getBuyableItemPrice which would be overridden by the buyable items which have a price. PowerStar, SuperMushroom extending MagicalItem and Wrench extending WeaponItem are implementing BuyableItem. They will override the getBuyableItemPrice methods to return their respective price value, which are 1) PowerStar \$600 2) SuperMushroom \$400 3) Wrench \$200. This implementation follows **Interface Segregation Principle (ISP)** because the interface has only 1 method in it which returns buyableItemPrice. The interface is kept as small as possible and has very specific tasks so buyableItems are not forced to implement other methods they do not require. It also obeys **OCP** as it is very easy to add an item which is buyable without modifying codes in other classes.

We then move to the discussion of TradeAction class. In TradeAction class, we have 3 variables: actor, map, and buyableItem. We create a constructor to initialise these 3 variables so toad can add tradeAction. We also implement a menuDescription to let users know what items they are buying. An execute method is created to create a tradeAction for actors to buy BuyableItem. This method starts with getting the balance in the wallet using the getInstance and getBalance method in Wallet. Then, we obtain the price of buyableItem using the getBuyableItemPrice method. If the balance is higher than the price, trade is successful. We would call getInstance and deductBalance methods in Wallet to reduce the balance and add the item into actor's inventory using the method addItemToInventory in Actor class, then print a message saying the actor buys the item. If the balance is not enough, we print a message indicating balance is not enough. This implementation follows **Don't Repeat Yourself Principle (DRY)** as in the buyableItem there is a method for retrieving price. By using this interface, we do not need to get the price of Wrench, PowerStar and SuperMushroom using 3 different getBuyableItemPrice methods, only 1 is needed.

Moving on to toad, it is a class extending Actor. It has an instance variable which is an arraylist of BuyableItem called buyableItemList. This list is used to store a list of all buyable items. We build a constructor which would call Actor (super) constructor. After

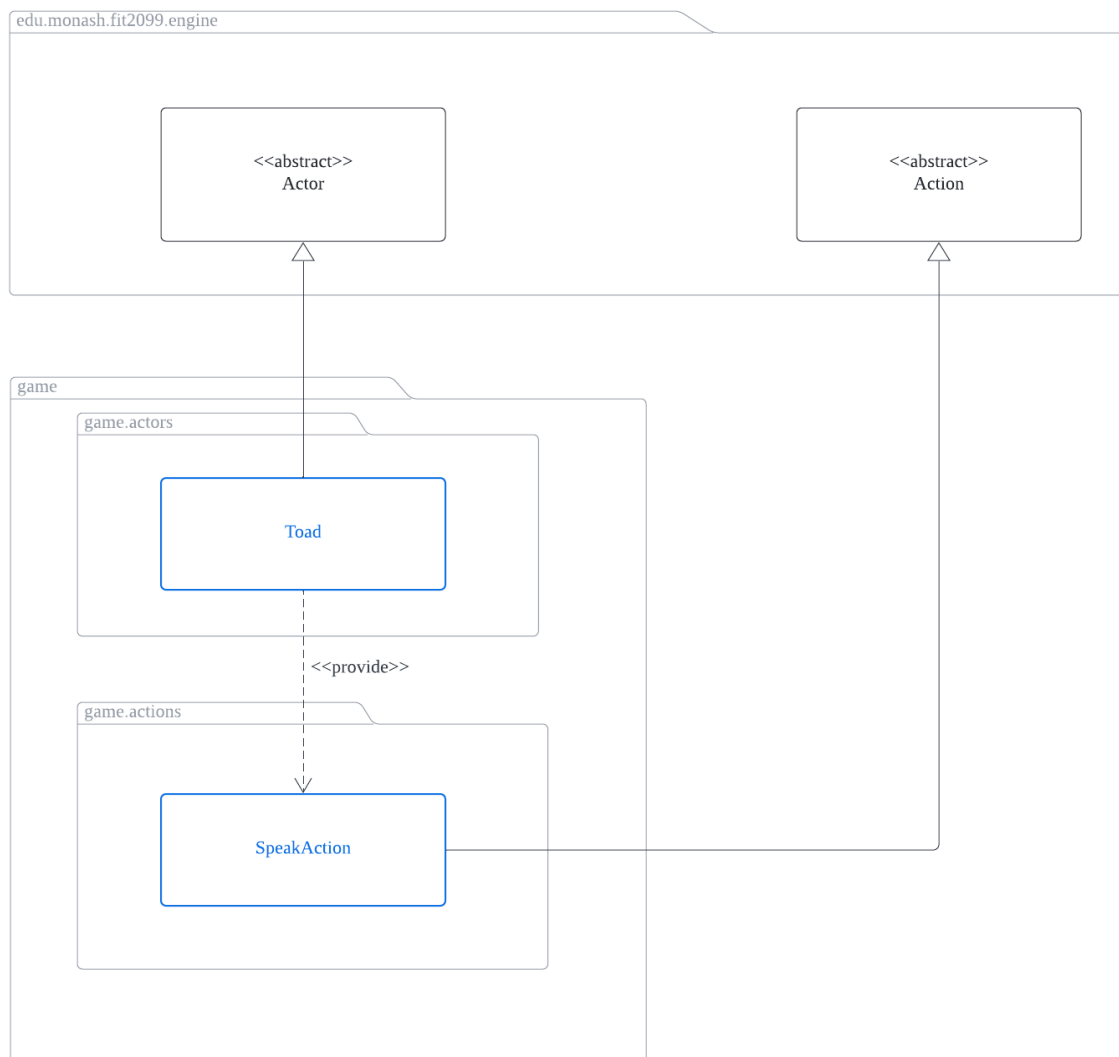
that we initialise our `buyableItemList` by calling the `refillBuyableItemList` method. `RefillBuyableItemList` is a method used to create a new arraylist containing `PowerStar`, `SuperMushroom` and `Wrench` and assign it to `buyableItemList`. This method is created because after `TradeAction`, we would add the item in the list to the inventory of actor. We then need to refresh the list using the method to prevent the situation of different players holding the same item instance. We also override the `allowableActions` from `Actions`. In this scenario, if a player is near toad, toad will add a `speakAction` to `actionList`. Then, it will `refillBuyableItem` list to create new instances of `buyableItems`, next loop through the `buyableItemList` to add `TradeAction` of every `buyableItem` to the actor and return the `ActionList`. We would then override the abstract method from `Action` and perform `DoNothingAction` in it because it is toad. It follows the principle of **Classes should be responsible for their own properties** because we are putting `refillBuyableItemList` right before adding `TradeAction` rather than anywhere else because they are related to each other.

In order for the buyer (essentially our player as we are assuming we only have one player) to be able to trade with the toad, we would need to have a wallet system that can track the total balance of the buyer's money. Thus, we created a **Wallet** class where the instance of the **Wallet** that stores the balance would be publicly accessible via a public static method, which we would be able to add balances to and also deduct balances from. And since we are assuming we only have one buyer in this game, we would only need to keep track of one wallet, and this is why our implementation would work as getting the wallet instance would return us the one and only wallet in the game.

As we already have our wallet system, the key question is how would we pick up Coins that are on the ground and add the credit into our wallet balance? We did this by implementing another new action called **PickUpCoinAction**. A coin would add **PickUpCoinAction** into its `allowableActions` list and whenever an actor executes this **PickUpCoinAction**, we would get the wallet instance and add balance to the wallet according to the coin's value, then the coin would be removed from the map. The reason for creating a new **PickUpCoinAction** that extends **Action** class is because we wanted to follow **SRP** where it would handle situations where we pick up coins and interact with our wallet.

REQ6

Class Diagram:



Requirement:

- The player can speak with Toad if Toad is in its range
- Toad must not speak certain sentences in certain scenarios.
- Toad must randomly choose one of the sentences from all the available sentences

Design Rationale:

Firstly, to start this requirement, we have created a **Toad** class which extends Actor abstract class as Toad is classified as an actor. We initialise the instance of Toad with 3 input parameters, which is its name (Toad), its displayChar (0), and its hitPoints which theoretically can be any number that are greater than 0 since any of the actor in the map will not be able to hit the Toad. We would also override the `allowableActions()` method such that when the **Player** is getting the allowable actions from the Toad, it will return a **SpeakAction** action as one of the allowable actions that can be performed between the Toad and the player.

Besides, to ensure that no other actor can hurt the Toad, we will not return **AttackAction** as one of the allowable actions that can be done on Toad. Basically, the allowable actions that can be done on the Toad by the player are just the **SpeakAction** and the **TradeAction**, and any other actor would not have any allowable action that can be done on the Toad.

Moreover, since Toad is a NPC that does nothing in every round, we will override the `allowableActions()` method in Toad class and return new **DoNothingAction()** so that Toad does nothing in every round.

Next, as we mentioned earlier, one of the allowable actions will be returned to the player when he/she is in range of the Toad will be **SpeakAction()**. **SpeakAction** is a subclass of **Action** which we created to handle the conversation between the player and the Toad. The implementation of this **SpeakAction** class follows the **Single Responsibility Principle (SRP)** as this class will only have one responsibility, which is to handle what should the Toad reply to the player once the player talks with it. While all the methods in the class contain all functionality to support the responsibility.

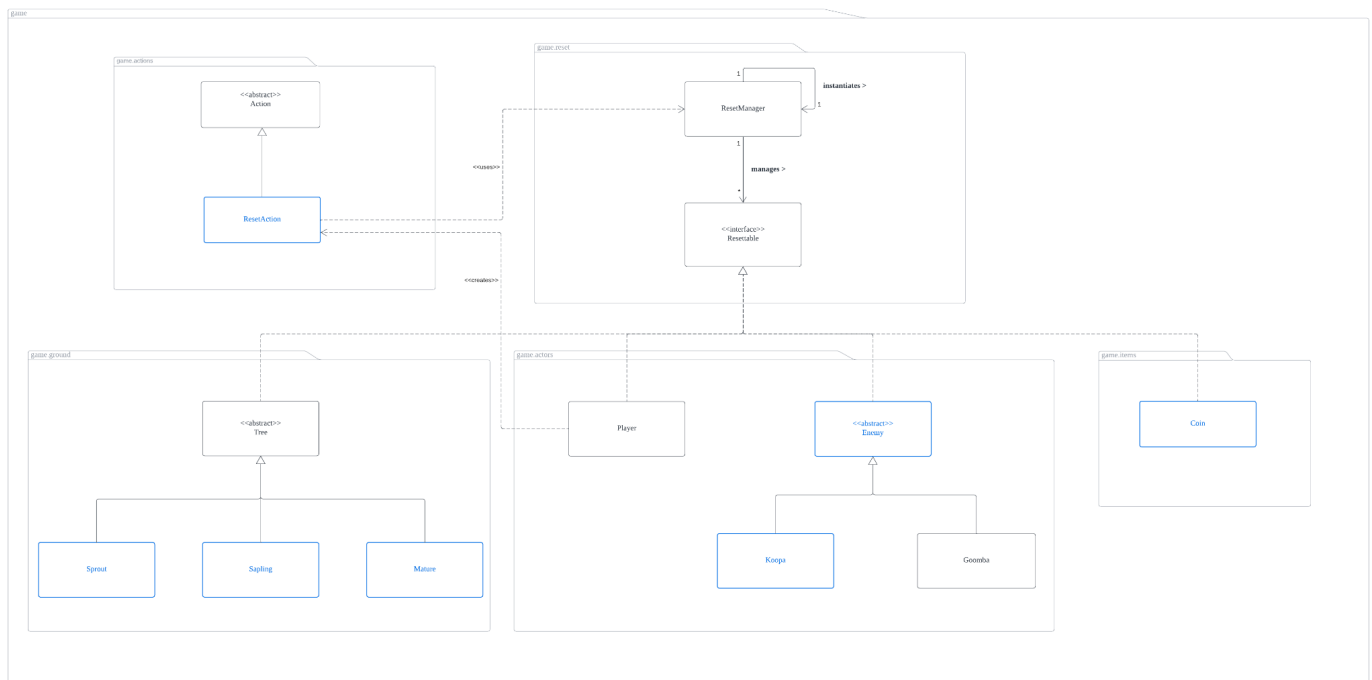
We would override the `execute()` method in the **SpeakAction** class to choose which sentences the Toad should reply to the player. We would use `actor.hasCapability(Status.DESTROY_SHELL)` to check if the player is holding a Wrench, if yes, then the first sentence would not be one of the option sentence for the Toad to choose from, if not, then the first sentence will be added into an **ArrayList** of **String** named **speakableSentences**. Also, we would use `actor.hasCapability(Status.INVINCIBLE)` to check if the player's power star effect is active, if yes, the second sentences would not be added as one of the options, if not, it will be added into the **speakableSentences**. After that, the third and fourth

sentences will always be added into the `speakableSentences` since they do not need to meet any conditions.

Subsequently, we would use the method `speakableSentences.get(rand.nextInt(speakableSentences.size()))` to choose one of the sentences from the `speakableSentences` `ArrayList` and return it as the reply sentence from the Toad. The working principle of the line of code above is that `rand.nextInt(speakableSentences.size())` generate any random number between 0 (inclusive) and the size of the `speakableSentences` (exclusive), the result integer will be passed into the `get()` method in `speakableSentences` to choose the respective sentence at that particular index.

REQ7

Class Diagram:



Requirement:

- Add in an option for the player to reset the game
- Player can only reset it once
- Trees have a 50% chance to be converted back to Dirt
- All enemies are killed
- Reset player Status
- Heal player to maximum
- Remove all coins from the ground

Design Rationale:

In requirement 7, we would utilise the existing **Resettable** interface to reset all the things that stated in the requirement. To elaborate more on this, we would implement the **Resettable** interface in the abstract **Tree** class, abstract **Enemy** class, **Player** class and the **Coin** class. We choose to implement the interface in the abstract **Tree** and **Enemy** class because all the subclasses should be reset if the reset option is chosen. Besides, in the constructor of each concrete class that implements the interface, we would use the method `this.registerInstance()` to register them to the **ArrayList** of type **Resettable** in the **ResetManager** class to simplify the reset process. The use of

ArrayList of Resettable follows the **Dependency Inversion Principle (DIP)** since the High-level modules (**ResetManager** class) does not depend on low-level modules such as **Player** class, **Coin** class and more. When we add in more classes that need to be reset, we do not need to modify the **ResetManager** class but just implement the **Resettable** interface in that class and the method in its constructor, and it will automatically be part of the reset cycle.

Moreover, in each of the classes that implement the **Resettable** interface, all of them will need to override the `resetInstance()` method and do what should be done according to the reset requirement specified. The explanation is break down to different sections:

Reset for all the Trees

We would override the `resetInstance()` method in the **Tree** abstract class, and would not override it again in the subclasses since all of them will have the same code implementation for this method, which has a 50% chance to be converted back to Dirt. This is an implementation that follows the **Don't Repeat Yourself Principle (DRY)**. The actual execution of reset will happen in the `tick()` method.

All enemies are killed

We would override the `resetInstance()` method in the **Enemy** abstract class. If the method is called, it will add the capability of RESET to the instance of the Koopa and Goomba. And the actual execution of reset will happen in the `playTurn()` method of both of the classes.

Reset player status & heal player to maximum

We would override the `resetInstance()` method in the **Player** class. To reset the player's status to the original state, we would tackle this problem by remembering the initial value of status of the player and reassign the initial value to the player attribute and reset all the status to default value if the method is called.

Remove all coins on the ground

We would override the `resetInstance()` method in the **Coin** class. If the method is called, it will add the capability of RESET to the instance of the **Coin** class. The actual execution of reset will happen in the `tick()` method where all the coins on the ground will be removed.

Furthermore, we would create a **ResetAction** class that is the subclass of **Action** which aims to provide the player a reset option in the menu. The implementation of **ResetAction** class obeys the **SRP** as this class has only one responsibility, which is to reset the whole game. In order to provide the player, the reset action as one of the allowable actions, we would create a static attribute for the player called `ableToReset` which checks the status of reset, in other words, whether it is able to reset or not. Besides, we would add in a checking mechanism in the `playTurn()` method of the player to check if this reset action is still one of the available options. Once the reset action has been executed, we would change the `ableToReset` attribute to false so that it will not be an allowable action anymore. Hence, the requirement of resetting the game once has been fulfilled.