FIT2099 Assignment 3 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 **design & coding implementation (Assignment 3)** into three part

Part 1 (Author: Samuel Tai Meng Yao)
REQ4 - Flowers (Structure mode)
REQ5 - Speaking (Structure mode)

Initial commit: 7/5/2022
Completion: 22/5/2022
Reviewer: George Tan Juan Sheng, Shun Yao Tee

Part 2 (Author Shun Yao Tee)
REQ1 - Lava zone
REQ3 - Magical fountains

Initial commit: 7/5/2022
Completion: 22/5/2022
Reviewer: George Tan Juan Sheng, Samuel Tai Meng Yao

Part 3 (Author George Tan Juan Sheng)
REQ2 - More allies and enemies
REQ4 - Flowers (Structure mode)

Initial commit: 7/5/2022
Completion: 22/5/2022
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

Disclaimer

All updates have been highlighted in yellow and for each requirement specification, the differences between assignment 2 and assignment 3 are documented below each requirement.

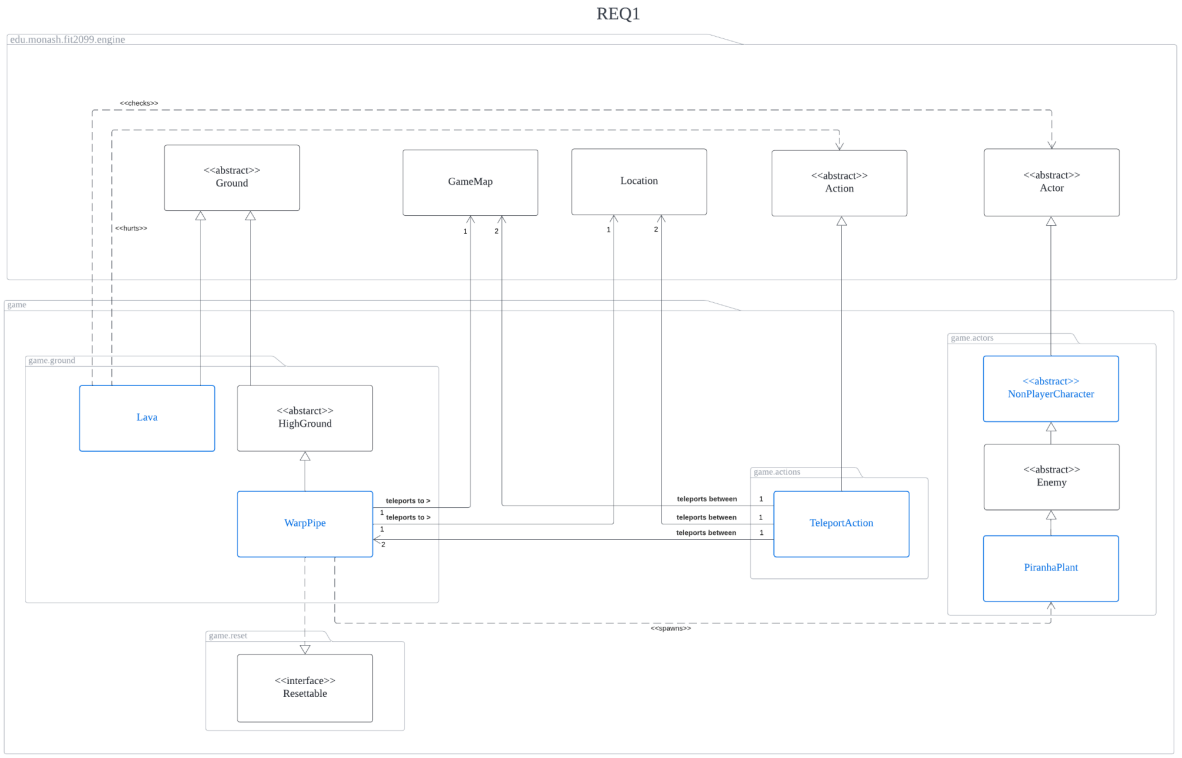
General Changes (Assignment 3)

In our previous implementation for Enemies' behaviours, we assumed all Enemies would have FollowBehaviour, AttackBehavior and WanderBehaviour. However, this is a mistake as we can see like **PiranhaPlant** only has AttackBehaviour, **Bowser** does not have WanderBehaviour etc. Thus, we made it such that each concrete subclass of **Enemy** class will define its own behaviours that need to be added. Although this may result in some repeated code throughout the concrete subclasses, this does not violate **SRP** as this repetition of code is needed because not all enemies have the same behaviours.

Design Documents (Assignment 3)

REQ1

Class Diagram:



Requirement:

- New map with lava ground
- Lava ground that inflicts 15 damage per turn to the player standing on it. Enemies cannot enter lava.
- Warp Pipe that would spawn a Piranha Plant in the second round. After Piranha Plant is dead, it allows teleportation from the first map to lava map and vice versa for the player.
- Resetting the game would make warp pipe spawn Piranha Plant again.
- Piranha Plant that cannot move and stays on a warp pipe.
- When an actor teleports from one warp pipe to another in another map, the actor should be able to teleport back from that warp pipe in the second map back to the

warp pipe he used. If the actor repeats the action again using another warp pipe, teleporting back from the second map will teleport the actor to the second warp pipe he used.

Design Rationale:

We decided to create a class called **Lava** which extends Ground class. There is one attribute called burnDamage in the class. We make a constructor which would initialise its display character and set burnDamage to 15. This follows **SRP** as by extending ground class, it is very easy to make a lava ground. In the tick method, we would hurt the actor standing on it. Instead of just putting the damage in the tick function, we make burnDamage as an attribute as it brings clearer meaning and it is easier to maintain.

Next, a **PiranhaPlant** class which extends the Enemy class is created. A **WarpPipe** class extending **HighGround** and implementing **Resettable** is made. Being a highground, warp pipe is set to have 100 jump success rate and 0 fall damage. If the actor is in invincible state, it will destroy warp pipe upon jumping onto it. There are 4 attributes in the WarpPipe class, including destinationMap, destinationLocation, destinationWarpPipe and boolean hasSpawned. In the requirement, we need to keep track of the warp pipe, location and map where the actor teleports from so when the actor teleports back from the second map to the first map, it can teleport to the previous warp pipe. In our design, we decided to record this information in the warp pipe using the 3 attributes destinationMap, destinationLocation and destinationWarpPipe. The hasSpawned attribute is used to track if the PiranhaPlant has been spawned as we can only spawn PiranhaPlant in the second round. During the second round and every time the game gets reseted, hasSpawned will be set to false and it will spawn again on warp pipe. Player needs to kill PiranhaPlant in order to stand on the warp pipe and teleport. In the allowableActions() method, it will first call `super.allowableActions()` from highGround, this follows **DRY** because we do not need to repeat the same code for all highGround. Next, it will check if a player is standing on it and add a new **teleportAction** to the second map, information of 3 destination details will be passed in. We make one 0-parameter constructor and a 3-parameter constructor which have 3 destination attributes for warp pipe class. This follows the principle that **classes should be responsible for their own properties.** The 3 destination information are closely related and change together, so they are placed together in the constructor and in passing

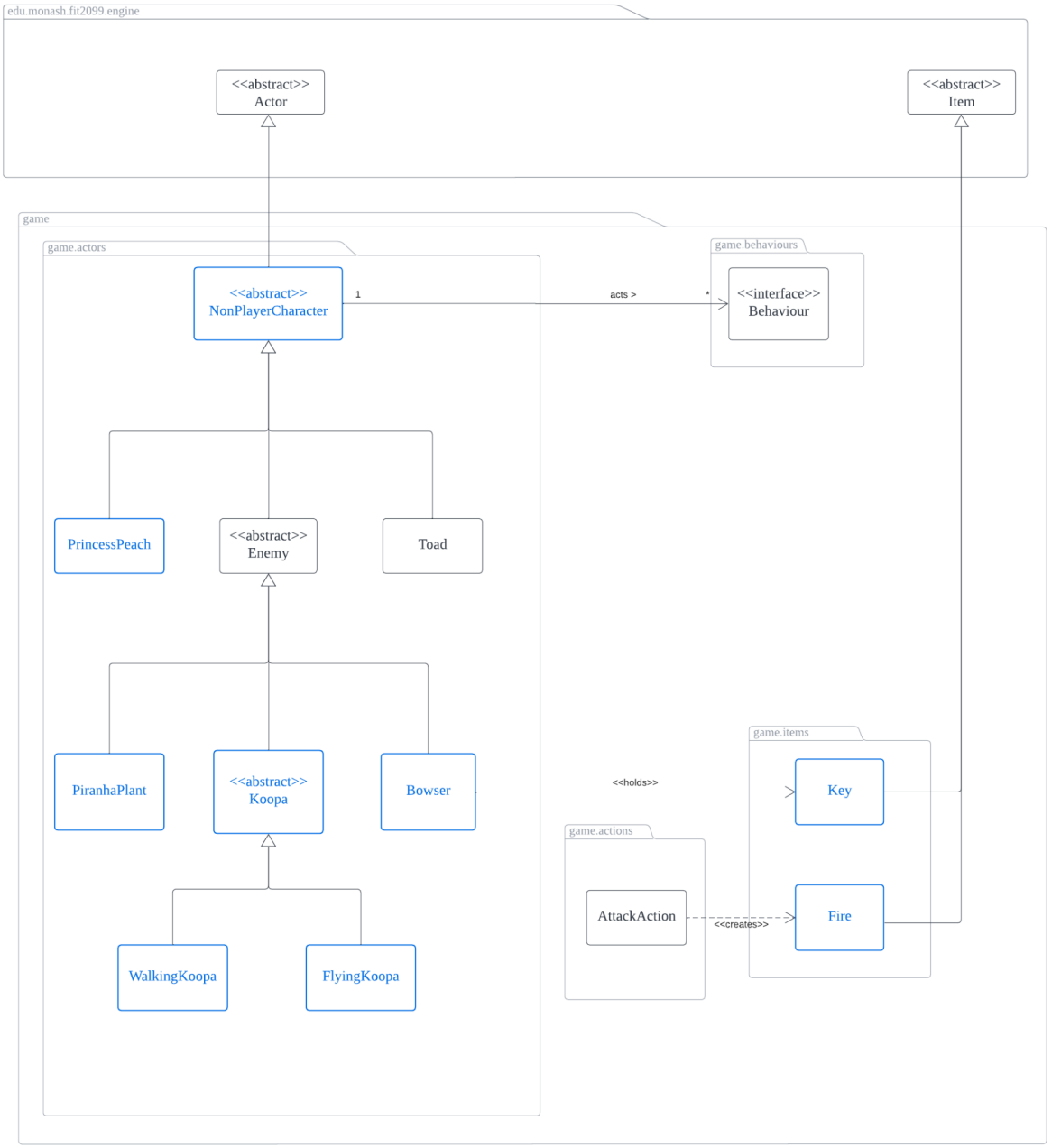
parameters of **teleportAction**. **WarpPipe** class is also responsible for spawning the warpPipe thus it has hasSpawned attribute to keep track of the spawning.

To enable teleportation, we added a **TeleportAction** class extending Action class. The extension follows **OCP** as adding a new action is very easy without affecting the code of other actions. There are 6 attributes in the class, destinationMap, destinationLocation, destinationWarpPipe, sourceLocation, sourceWarpPipe, sourceMap. We make a constructor which accepts and initialises these 3 attributes. In the `execute()` method, we will first check if the destination location has an actor. If there is, it will be PiranhaPlant in the current design and we will destroy it by removing it from the map. Then, we would move the actor to the destinationLocation, which is on a warp pipe. After that, we would update the destinationWarpPipe, destinationLocation and destinationWarpPipe attributes of the destination warpPipe to the ones the actor teleports from, which are the 3 source information. This design follows the principle that **classes should be responsible for their own properties.** The 3 attributes of the destination warp pipe are updated together using the 3 source information attributes. **SRP** applies here too. The TeleportAction class has only 1 feature, teleporting actors from a warpPipe in a map to another. No excess unnecessary features are added here.

To create a new map with lava ground, we make a lavaGroundFactory in application class. We then make new maps filled with a lot of lava grounds. Next, we initialise a few warp pipes in the first map in random locations using the 3-parameters constructor in warp pipe class. In this case, all warp pipes in the first map will allow teleportation to the only warp pipe in the top left corner of the lava map. After the first teleportation, the warp pipe in lava map will record the information the actor teleports from. Thus, the system works well. Instead of initialising the 3 destination attributes in warp pipe class which requires a static method and static variables for the initial values, we decided that it is not good design. So we do the work to initialise the warp pipe and the 3 attributes in application class. This follows the principle of **avoiding excessive use of literals.**

REQ2

Class Diagram:



Requirement:

- Princess Peach does not do anything until Mario has acquired a key to interact with her in order to win the game

- Bowser does not wander around, only stands still/follow/fire attack Mario. Bowser drops a key once defeated
    - Key allows Mario to interact with Princess Peach
- Piranha Plant that spawns on top of a Warp Pipe that attacks Mario
- Flying Koopa which is similar to our Walking Koopa, but flies around and is able to fly past high grounds as well

Design Rationale:

From an Object Oriented Approach, we decided to create an abstract **NonPlayerCharacter** (NPC) class that extends **Actor** class, where all NPCs will have behaviors. By defining such an NPC class, our program becomes very extensible as we have more and more NPCs in our game. This is highly possible as any actors excluding the **Player** would be an NPC. Furthermore, such an NPC class also allows us to conveniently implement REQ5 as all speakable actors will have a **SpeakBehaviour**. Do refer to REQ5 for a more detailed explanation of how **SpeakBehaviours** works.

Moving on to the requirements, **Princess Peach** can be easily implemented by extending the **NonPlayerCharacter** class. **Princess Peach** would also execute a DoNothingAction on each turn and its `allowableActions` would return a VictoryAction when **Mario** has the **Key** and stands beside her. The VictoryAction basically removes **Mario** from the map and prints a victory message on the display. The reason why VictoryAction removes **Mario** is so that the game would stop and a game-over message would be shown. As VictoryAction solely handles the scenario of **Mario** winning the game, this follows **SRP (Single Responsibility Principle)**.

Bowser is implemented by extending **Enemy** so that its `allowableActions` would return an AttackAction that allows **Mario** to attack it. **Bowser** would always execute a DoNothingAction in its playTurn if **Mario** has not entered its attack range (stand beside Bowser). Once **Mario** stands beside **Bowser**, **Bowser** would be added a FollowBehaviour and AttackBehaviour targeted towards **Mario**. Then, **Bowser** would attack **Mario** whenever possible, if not **Bowser** would keep following **Mario**. A **Key** item is also added to **Bowser's** inventory so that when **Bowser** dies the **Key** will be dropped for **Mario** to pick it up. The **Key** class extends **Item** class and adds a `Status.VICTORY` capability to itself, thus when **Mario** has this **Key**, **Mario** also has the `Status.VICTORY` capability which allows **Mario** to interact with **Princess Peach**. Upon instantiation,

Bowser is also given the `Status.FIRE_ATTACK` capability so that it is able to perform fire attacks. **Fire** attack would be covered in detail in Requirement 4. Since all enemies implement Resettable interface, **Bowser** is also resettable. When the player resets the game, **Bowser's** resetInstance would provide **Bowser** the `Status.RESET` capability. During **Bowser's** `playTurn`, we would check if **Bowser** has the `Status.RESET` capability, if yes, means we need to reset **Bowser**, where **Bowser** would be healed to maximum hp and be placed in its original position. **Bowser's** behavior is also cleared so that it will just stand on its spot until **Mario** approaches **Bowser** again. Finally, **Bowser's** `Status.RESET` capability would also be removed.

Piranha Plant is implemented by extending **Enemy** as well so that **Mario** could attack it. Similar to **Bowser**, **Piranha Plant** always executes a **DoNothingAction** and when **Mario** stands next to **Piranha Plant**, **Piranha Plant** is added an AttackBehaviour targeted towards **Mario**, so **Piranha Plant** attacks **Mario** whenever **Mario** is beside **Piranha Plant**. When **Mario** is not in range for **Piranha Plant** to attack it, **Piranha Plant** continues to execute DoNothingAction and does not move. Since **Piranha Plant** is also resettable, in **Piranha Plant's** `playTurn` we would check if **Piranha Plant** has the `Status.RESET` capability, if yes, **Piranha Plant's** max HP would be increased by 50 and it would be fully healed. Finally, `Status.RESET` capability would be removed from it.

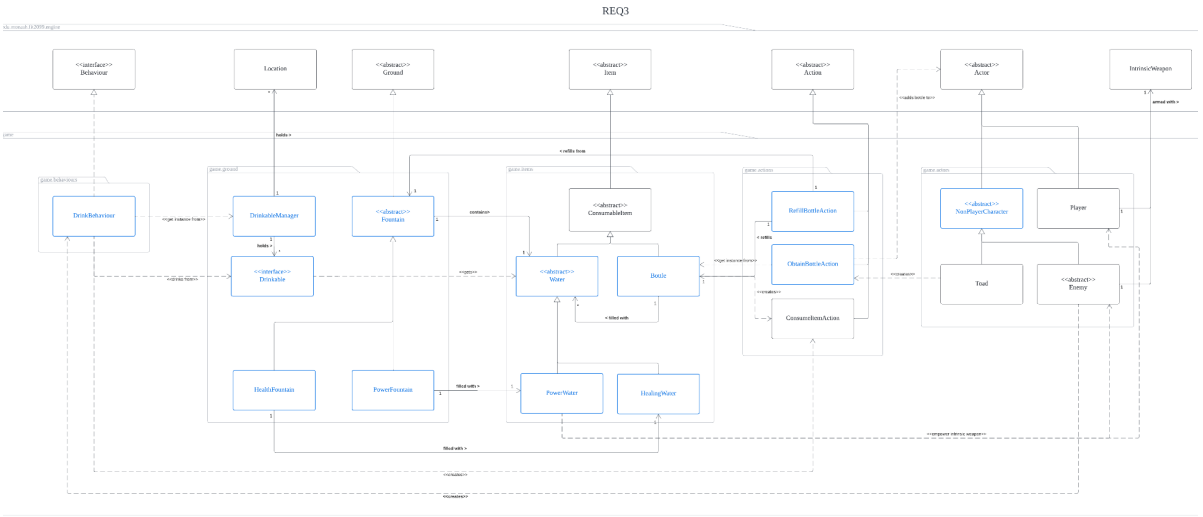
For implementing **FlyingKoopa**, taking into consideration of adhering to **LSP (Liskov Substitution Principle)**, we have decided to create an abstract **Koopa** class which is extended by a concrete **WalkingKoopa** class and a concrete **FlyingKoopa** class. The abstract **Koopa** class also extends **Enemy** so that all the **Koopas** can be attacked by **Mario** as an AttackAction would be returned in its `allowableActions`. The abstract **Koopa** class would contain all the common features of **Koopa** such as being able to enter dormant state and have its shell broke by **Wrench**, wander around, follow and attack **Mario** etc. Since **WalkingKoopa** would behave similarly to our base **Koopa** class, **WalkingKoopa** would just access **Koopa's** methods (for allowableActions, playTurn etc) by using super. **FlyingKoopa** also behaves very similar to the **Koopa** base class, but it is given a capability of `Status.FLY`. By having such an implementation, both **WalkingKoopa** and **FlyingKoopa** could be replaced with **Koopa** as they both hold the similar traits as the base abstract **Koopa**, just that **WalkingKoopa** walks and **FlyingKoopa** flies. While we are not expecting the base abstract **Koopa**

class to behave in a certain manner for its movements, replacing **WalkingKoopa** or **FlyingKoopa** with **Koopa** would not result in an unexpected behaviour, hence we adhere to **LSP.** By having this abstract **Koopa** base class, we also adhere to **OCP** as it is extensible if we do more types of **Koopas** in the future, such as **SwimmingKoopa** etc.

For letting **FlyingKoopa** to be able to fly around, we utilize the `Status.FLY` capability being added to **FlyingKoopas** and we added a line of checking in **HighGround's** canActorEnter to always return true if actors have the capability `Status.FLY`. Despite us modifying our current system for this feature, it is only **2 lines of code added** and if we were not to do it this way, it might require implementations of many other abstract classes and changes, which might lead to an **Overengineering Odor**, hence we decided to keep it simple since its only 2 lines of code added. Furthermore, although we modified the current code for **HighGround** for this new feature, it did not break any of the old features, hence it is arguable that this implementation does not really violate **OCP**.

REQ3

Class Diagram:



* for clearer image refers to file in docs docs/A3/REQ3/UML_REQ3

Requirement:

- Bottle that can contain unlimited magical water. Mario can drink water in a bottle to gain a unique effect.
- Health Fountain that contains healing water which heals drinker by 50 hp
- Power Fountain that contains power water that increase drinker’s intrinsic weapon damage by 15
- Optional: Enemies standing on fountains can drink water and gain effect. Find a balance in behaviour priorities.
- Mario doesn’t have a bottle at the start of the game, he needs to obtain it from Toad.
- Fountain has limited water capacity. Refilling a bottle will use up 1 water slot, drinking water will consume 5 water slots. Once the fountain is exhausted, it will replenish in 5 turns.

Design Rationale:

In requirement 3, we make a **Bottle** class extending **ConsumableItem**. By extending it, **OCP** applies here as we don't have to repeat code for consumableItem. Initially we are not making bottles as consumable as we think it will be more reasonable to consume water and not bottle. Due to the sample output of menu description in requirement 3 which shows "mario consumes bottle[water]" and the fact that bottle is consumable is still acceptable, we extend the bottle as ConsumableItem. There are 2 attributes in Bottle, waterStack which stores a stack of water and static Bottle instance. We make a private constructor and public `getInstance()` method which returns the bottle instance. The reason behind this implementation is because accessing Bottle in Actor's inventory will require casting to call the method of Bottle that Item does not have access to. This applies **DIP** because high level modules (Item) in inventory do not need to depend on low level modules (Bottle). When obtaining Bottle from inventory we do not need to check if Item instanceOf Bottle, we access the method in Bottle. Bottle would depend on the abstraction on ConsumableItem and Item now, Item does not need to depend on the details of Bottle. Inside Bottle, we would have `getWater()`, `fillWater()`, `consumeWater()`, `consumeItem()` and `allowableActions()` which add consumeItemAction for Bottle. Talking about Bottle, we can relate it to **ObtainBottleAction** and **RefillBottleAction**. In ObtainBottleAction, it will add the Bottle instance to the actor's inventory. This applies **SRP** as this class has only 1 responsibility which is to add the bottle into inventory. This class also follows **LSP**. In the actor.addItemToInventory(x) method, x is expected to be an Item object. According to LSP, it can accept subclass objects of Item class, which means adding a Bottle instance into inventory is acceptable. The ObtainBottleAction will be added in **Toad** class when a player which has no bottle in inventory is beside Toad. In RefillBottleAction, there are 2 attributes, fountain and bottle. It follows **SRP** too because it only does 2 things in execute() method, decreasing water in fountain and adding water into bottle.

Following our design we have a **Water** class which extends **ConsumebleItem**. We also have **PowerWater** and **HealingWater** extending Water class. **OCP** is reflected in the design here. By using inheritance and abstraction, it is very easy to add new water without changing the implementation of other water or items. Water class override constructor from ConsumableItem. Healing Water and PowerWater inherit constructors and override `consumeItem()` method from ConsumableItem. In healing water

implementation, consumeItem() would heal player by 50hp. For PowerWater, we check if actor has the capability of `Status.HOSTILE_TO_ENEMY`. If yes, we would cast to player `((Player) actor)`, else we cast to enemy `((Enemy) actor)` and then call `empowerIntrinsicWeapon()` method. In player and enemy classes, we would make an **intrinsicWeapon** attribute and an `empowerIntrinsicWeapon()` method which increase damage by 15. We have our only use case of casting in this method under the limitation that engine code is unchanged. This is because we need to keep track of number of times the player or enemy consume powerWater or keep track of how strong the intrinsic weapon has grown into. So we need to have an attribute in player and enemy classes. However, the actor class has no access to intrinsicWeapon attribute and `empowerIntrinsicWeapon()` method, so casting is needed here. It is possible to prevent casting here by making an ableToEmpower interface and a manager. But this would just add all the player and enemy into the manager and we think that the effort is not worth to put some many extra work on adding all player and enemy to the manager and maintaining them to prevent small use case of casting in the current stage.

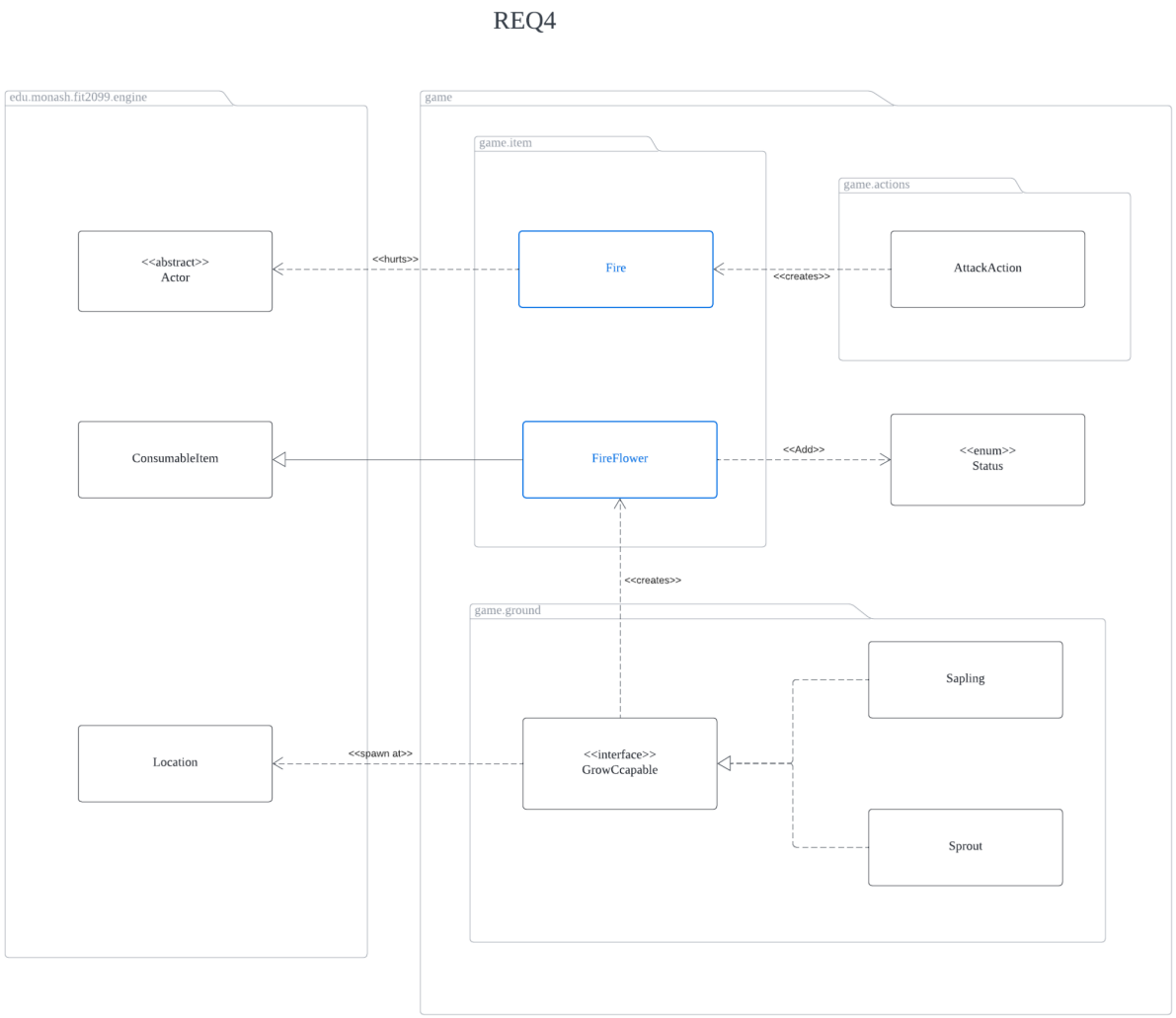
Moving on, we have **Fountain** class extending ground, followed by **HealthFountain** and **PowerFountain** extending the Fountain. This follows **OCP** as adding a new type of fountain is easy and will not affect codes for other fountain or any other parts. This also implements **DRY** principle because we have 4 attributes and 9 methods in fountain class. They will be inherited or overridden by HealthFountain and PowerFountain. This reduces repetition of code by a lot. Inside fountain class, we have water, waterAmount, capacity and turnsToReplenish attributes. During initialization in the constructor we will have waterAmount set to 10, capacity set to 10 and turnsToReplenish set to 5. In the tick() method overridden, we would check if the waterAmount is 0. If it is, we would start the counter of turnsToReplenish by decrementing the counter. Once turnsToReplenish reaches 0, we set the turns back to 5 and call `replenishFountain()` method which sets waterAmount to maximum capacity. In the allowableActions, we check if the current location contains an player, if the player has a bottle in inventory and if the waterAmount of fountain > 0. If all conditions are passed, we would call refillWater() and add a refillBottleAction(fountain, bottle). RefillWater() is a method to add a new instance of water to fountain as long as waterAmount > 0. It is an abstract method in fountain. It is overridden in PowerFountain to return new instance of PowerWater and overridden in HealthFountain to return a new instance of HealingWater.

We also have a drink() method which will call refillWater() and decrease waterAmount of fountain by 5, which would be used when enemy drinks water directly on fountain.

Next, we have **DrinkBehvaviour** which would be executed by enemy only. The drinkBehaviour will be added to enemy during initialisation of enemy by calling constructor of DrinkBehaviour class. For all behaviours, there are priorities and the priorities list including drinkBehaviour is as speak > attack > follow > drink > wander. In drinkBehaviour getAction(actor, gameMap) method, our team faces issue as our original implementation is to check if the location is an instanceOf fountain, then cast the location to get the fountain and check if the fountain has enough water. If yes, we would call drink() method in fountain class and add consumeItemAction of the water. To prevent this, we added **Drinkable** interface and **DrinkableManager** class. Fountain would implements Drinkable interface. Drinkable interface has 3 methods, getWaterAmount(), getWater() and drink() where all of them have no implementations and need to be overridden. This follows **OCP** as adding new drinkable Ground is easy due to the abstraction. This also applies **ISP** as the interface is kept as small as possible to have only the necessary features of a drinkable ground. In the application driver class, we would initialize one powerFountain and one HealthFountain and add it into DrinkableManager. Inside DrinkableManager we have a static instance of manager. We also have a hashMap which records location of drinkable ground as key and drinkable ground as value. With these implementations, instead of using instanceOf to check if the location is fountain in drinkBehaviour, we would call `DrinkableManager.getInstance().getDrinkableGroundCollection().containsKey(map.locationOf(actor))` to check the location. Then get the drinkableGround from the hashMap in drinkableManager. Check if the water is enough using drinkable.getWaterAmount() and use drink() method to refill and decrease water in drinkable ground. This implementation follows **DIP** as high level modules (DrinkableManager) does not depends on low level modules (PowerFountain and HealthFountain). Drinkable ground does not need to depends on details of fountain, fountain would depends on Drinkable ground via abstraction.

REQ4

Class Diagram:



Requirement:

- 50% chance to spawn/grow Fire Flower during the growing stage of the tree (Sprout -> Sapling and Sapling -> Mature)
- Mario can consume the fire flower and gain Fire Attack
- Multiple Fire flowers in one location
- Actor can consume the Fire Flower and use fire attack on the enemy
- After consuming Fire flowers, attacking will drop a fire at the target's ground
- "Fire attack" effect will last for 20 turns
- Fire will stay on the ground for 3 turns

- Fire deals 20 damage per turn

Design Rationale:

In order to fulfil part 1 of REQ4, we have created a new class called **FireFlower.** FireFlower class is a subclass of **ConsumableItem**. Due to the earlier implementation of **ConsumableItem** (making it as an abstract class), now we can easily implement the FireFlower class as a new item that can be consumed. Firstly, we override the consumeItem() method in **FireFlower** class to implement what should be done when the actor consumes the fire flower. Below is the code implementation and explanation,

1. Firstly, we will get the current location of the actor by using `Location currentLocation = map.locationOf(actor)`
2. Similar to the implementation of power star and super mushroom, we will remove the item after we consume the item by using `currentLocation.removeItem(this)`
3. Similar to the implementation of power star, we will add the FIRE_ATTACK capability to the **FireFlower** object by using, store it into the inventory of the player and remove the consumeAction of the **FireFlower** object so that it would not provide the consume option in the menu by using,
   ```
   this.addCapability(Status.FIRE_ATTACK)
   actor.addItemToInventory(this)
   this.removeAction(consumeAction)
   ```

We have added a new attribute called turnsLeft in the **FireFlower** class. This attribute is responsible for keeping track of the number of turns of “fire attack” effect left in the player after he/she consumed it. To do that, we overridden the tick() method of the **FireFlower** so that it will decrement the turnsLeft for every turn until it is equal to 0, and we will remove the **FireFlower** object from the player’s inventory (the effect has over).

The design above follows the **Single Responsibility Principle (SRP)** such that the responsibility of keeping track of the number of turns of fire attack effect will be given to the FireFlower object itself instead of the actor who consumed it. The FireFlower class will just need to focus on its single role, provide the actor that has consumed it the fire attack effect and remvoe it when the effect has over.

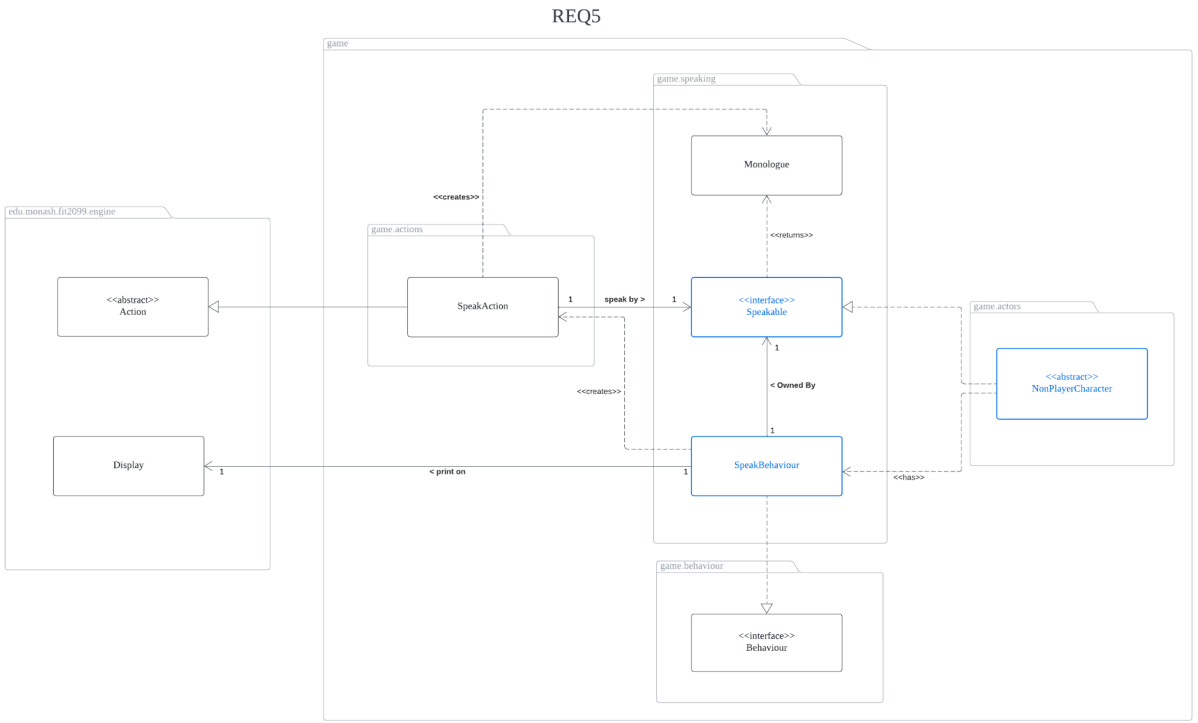
The spawning of fire flower during the growing stage of the tree is done in the **GrowCapable** interface, **Sprout** and **Sapling** class. Firstly, to fulfil the **DRY** principle, we have introduced the new default method called `spawnFireFlower()` method in the **GrowCapable** interface. This is very useful since it is the same code that will be used in both of the growing stage of the tree. In this method, we will check the random probability of the fire flower spawn, if it meets the probability, it will create a new **FireFlower** object and add it to the currentLocation of the tree. This method will be called in the overriden `grow()` method in the **GrowCapable** objects, which are **Sapling** and **Sprout** class for now. All the necessary checking and spawning will be done in the `spawnFireFlower()` method itself and the **Sprout** and **Sapling** objects just need to call it.

In order to implement **Fire Attack**, those actors that are able to perform Fire attack would be given a `Status.FIRE_ATTACK` capability. **Bowser** would be given this capability upon instantiation and other actors that have consumed **Fire Flower** would also be given this capability. In **AttackAction**, if the player has `Status.FIRE_ATTACK` capability, the `menuDescription` would be added a “with fire” message to indicate that the player would be performing a **Fire Attack**. During the execution of **AttackAction**, if the actor does not miss its target and its has the `Status.FIRE_ATTACK` capability, a **Fire** would be spawned at the location of the target. This **Fire** object extends **Item** and it is set to be not portable. Fire also has a counter attribute which is initialised to 3. This counter attribute would be useful for us to track if **Fire** should be removed from the map as **Fire** should only last for 3 turns. This also follows the **SRP** as **Fire** is responsible for tracking its own turns before it fades.

In **Fire’s** `tick` method, any actor that is on this **Fire’s** location would be hurt with 20 damage. We can do this by just getting the actor at **Fire’s** location, and if the result is not None, means we have an actor standing on the Fire, who would be damaged by 20 HP. At the end of the `tick` method, the counter attribute would also be decremented by 1. When the counter attribute reaches 0, the **Fire** would be removed from the map. If the **Fire** burns an actor to death, all of the droppable items in the actor’s would be dropped and the actor would be removed from the map. If a **Koopa** that has not entered dormant state is burnt to death, the **Koopa** would be put into dormant state. A **Koopa** that has entered dormant state would also get burned however it would still not destroyed until a **Player** destroys it with a **Wrench**.

REQ5

Class Diagram:



Requirement:

- Every listed character can speak at every even turn of the character
- Randomly picked one of the sentences
- Princess Peach, Toad, Bowser, Goombas, Koopas, Flying Koopa & Piranha Plant can speak

Design Rationale:

In requirement 5, since the self speaking of each character would run concurrently with the action that should be done by the character at that turn, therefore, we cannot implement it using the existing SpeakAction and implement it as usual (as an action). To address this problem, since we would assume that it will be the speak behaviour of the Non Player Character, we have created a new **SpeakBehaviour** class to solve this problem.

We have created a new **NonPlayerCharacter** abstract class that extends the **Actor** abstract class. The **NonPlayerCharacter** abstract class is extend by all the "NPC", such as **Enemy** class (which then extends by all the enemy), **Princess Peach** and **Toad**. The **NonPlayerCharacter** class provides an attribute named behaviours which is the HashMap that stores the possible behaviours that could be done by the NPC. The behaviours HashMap is the same with the one we implemented in the **Enemy** class in Assignemnt 2 so that the enemy can automatically do its action. The behaviours hashmap is needed for all the NPC due to all of them has the speak behaviour which is not the case in the assignment 2 where Toad does not has its own behaviour. The design follows the **Open-closed Principle (OCP)** since it can be extended in the future such that more NPC that can speak is added in. It also fulfil the **Don't Repeat Yourself (DRY)** since behaviours is the common attribute among all the instance of **NonPlayerCharacter** class and all of them can speak.

We will add the **SpeakBehaviour** into the behaviours HashMap of the **NonPlayerCharacter** objects in their `playTurn()` method. We will first check whether the object has this behaviour or not, if yes, then we will just continue the normal routine of playTurn, else we will add the **SpeakBehaviour** into the behaviours HashMap of the object using `addBehaviour(7, new SpeakBehaviour(this, display))`. The SpeakBehaviour has the highest priority among all the other behaviours due to the fact that it is compulsory for the method to be executed if it is the even turn of the object and it might be terminated if it has the lower priority than other behaviours. The reason why we did not add the speak behaviour during the initialization of the object is due to the fact that we need the display parameter in the SpeakBehaviour initialization which is only available in the `playTurn()` method.

We have created 3 attributes for the **SpeakBehaviour** class, which are

1. (int) turns: to keep track of the turns of the character to determine when to speak
2. (Speakable) speaker: the speaker that suppose to speak
3. (Display) display: manage the I/O of the system

We have created a new 2 parameter constructor for the SpeakBehaviour class such that it takes in the parameters and initialize the speaker and display attribute. While the turns attribute will be initialised to 0. This implementation fulfil the **SRP** because the speak behaviour need to keep track of the turns to speak instead of the **NonPlayerCharacter**

object itself. This proved that the **SpeakBehaviour** class only responsible for handling the self speaking feature, and it has reduced that responsibility of NonPlayerCharacter object to keep track of the number of turns it has spawned.

In order to let the **NonPlayerCharacter** object to speak without affecting the other possible behaviours that can be done by the object, we have made a slightly different code implementation in the `getAction()` method of the SpeakBehaviour. The SpeakBehaviour will always returns null so that it would not be affect the other possible behaviour. The code implementation is done as below,

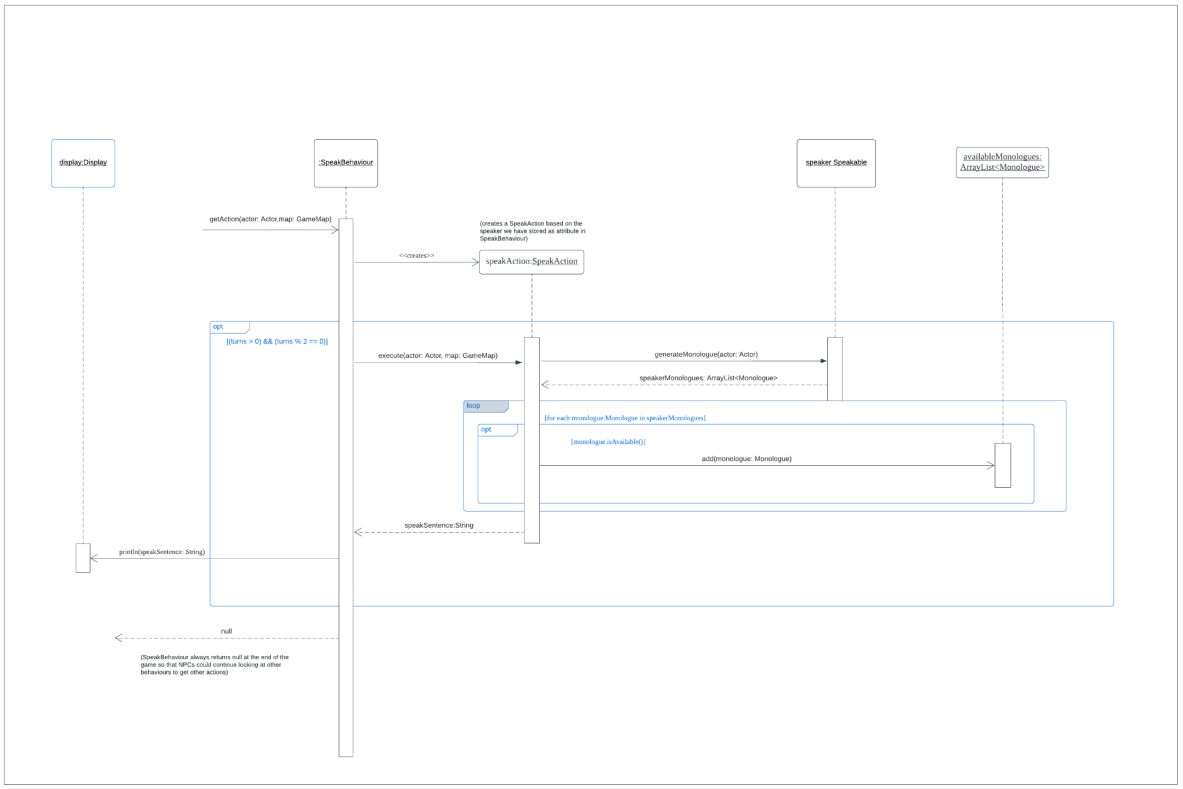
1. We will create a new **SpeakAction** object which passed in the speaker to initialise by using `SpeakAction speakAction = new SpeakAction(speaker);`
2. Then, we will increment the turns by 1. This has to be done before the checking of the even turns to ensure that the speaker can speak in the second turn after it has been created. Otherwise, it will be speak at the third turn after it spawns due to the turns attribute is initialized to 0 in the beginning.
3. Lastly, we will check whether it is a even turn using `if (turns > 0 && turns % 2 == 0)`. If true, then we will called the `execute()` method of **SpeakAction** and print the chosen sentence return from the method using `display.println(speakAction.execute(actor, map))`.

All the subclasses of **NonPlayerCharacter** should override the `generateMonologue()` method from the **Speakable** interface. In that method, the same thing will be done as the REQ6 of assignment 2 without the layer of checking since they can only have the self-speaking feature for now. For more detailed implementation of SpeakAction and the overriding of the `generateMonologue()` method, please refer to the updated REQ6 of assignment 2 above.

In conclusion, this REQ5 is basically the same thing as the updated REQ6 of assignment 2. The only difference is that the talk feature before needs to be triggered by the player and the current self-talking feature will be executed as a behaviour of NPC that will be triggered on every even turns, and that is the reason we utilise the code on REQ6 of assignment 2 and added in some new one to complete the current feature.

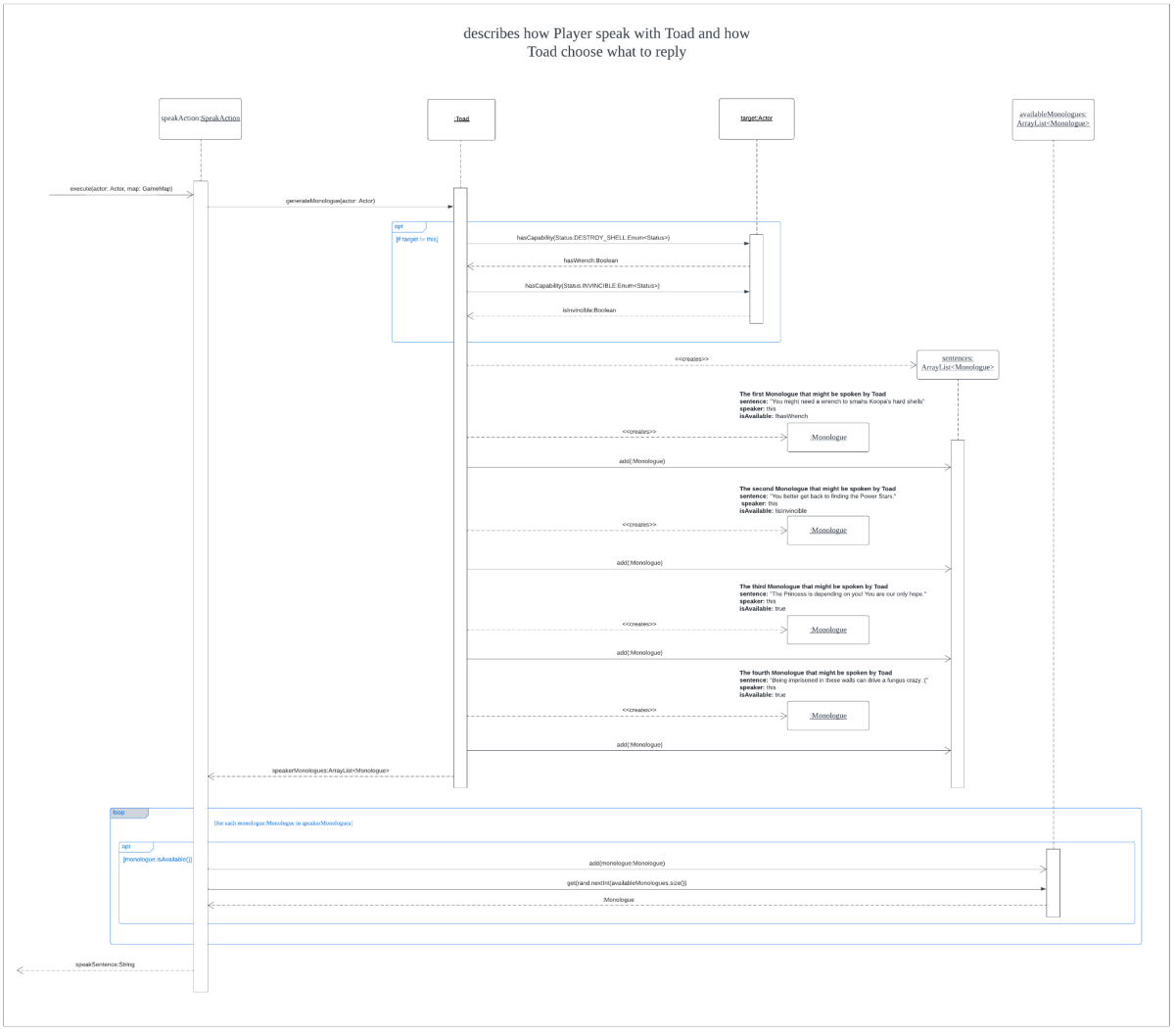
Sequence Diagram:

Sequence Diagram 1



This sequence diagram describes how each Speakable actors are able to speak in every even turn with the implementation of SpeakBehaviour, SpeakAction and Monologue. Each Speakable actors would be able to generate a list of Monologues with its respective availability. Then in SpeakBehaviour, a randomly selected available Monologue’s sentence would be displayed in console.

Sequence Diagram 2

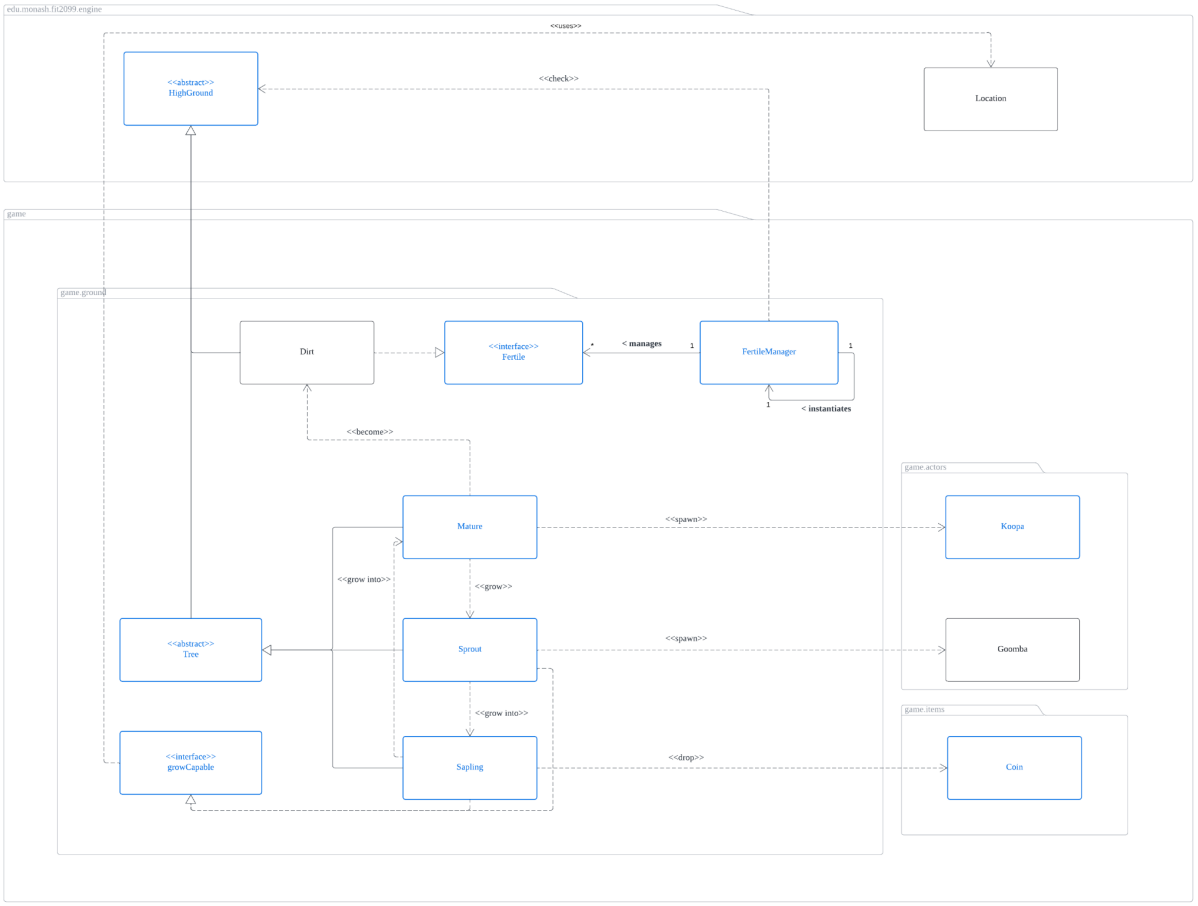


The sequence diagram above describes how Player speaks with Toad and how Toad chooses what to reply. This is actually one of the sequence diagram from REQ6 of assignment 2, but since we changed the whole implementation of the REQ6 of assignment 2 so that it can be used together with REQ5 of assignment 3, we decided to create this sequence diagram to illustrate the changing in code implementation of REQ6 will makes it more extensible in the future, and can reuses the code in REQ5 of assignment 3.

Design Documents (Assignment 1 & 2)

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 **HighGround**, 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 **Sapling** 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.

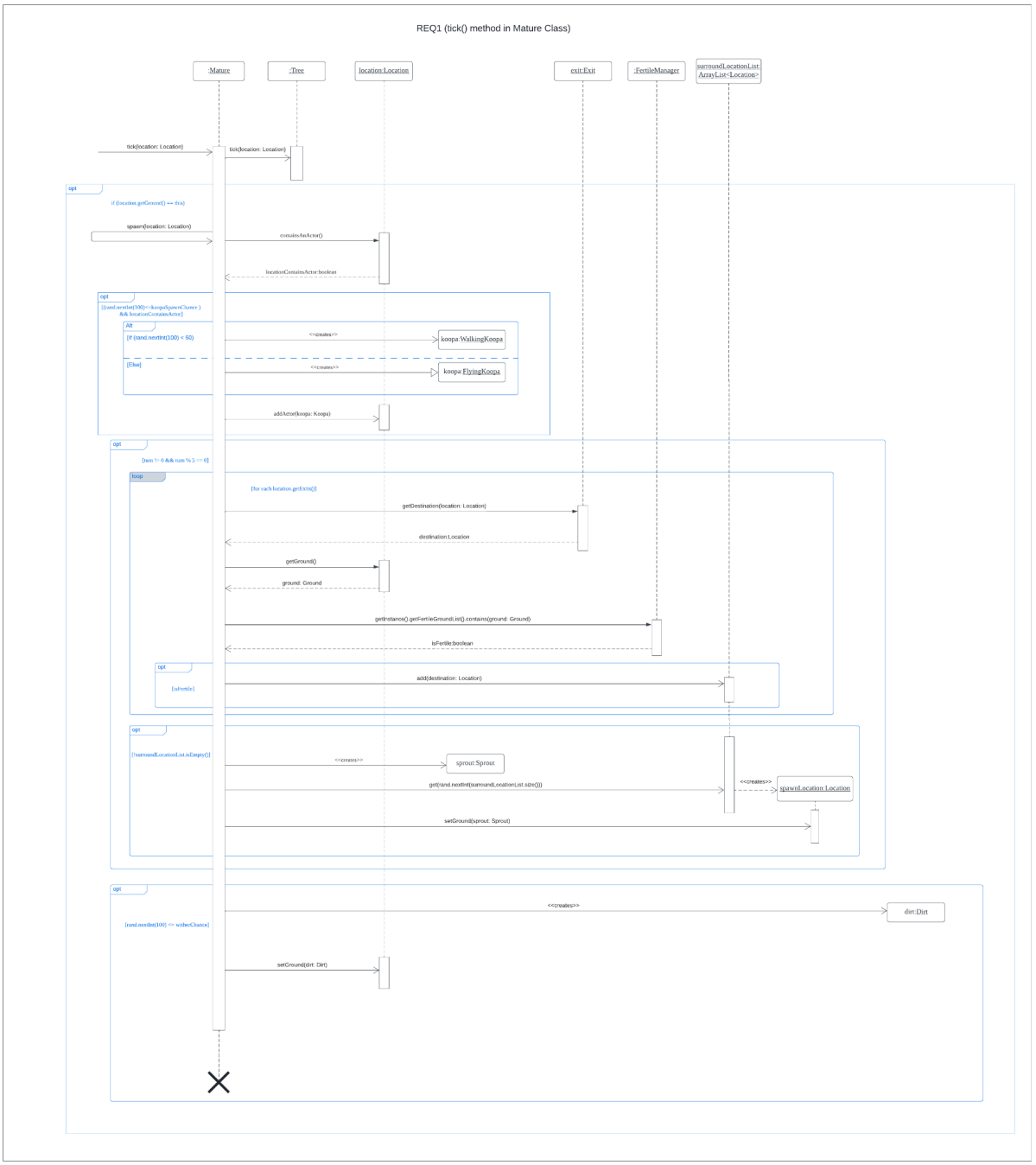
To implement the withering of Mature, we would use `(rand.nextInt(100)<= witherChance)` to check if it meets the 20% chance to be withered where the witherChance is a final attribute of Mature. If yes, then we would create a new Dirt object and use `location.setGround(dirt)` to change the ground type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

*Updates that have been done for REQ1 (class diagram & design rationale)

I. Class diagram:
    A. Tree extends from HighGround now
II. Design rationale:
    A. Changed Tree extends from Ground to Tree extends from HighGround

Sequence Diagram:



This sequence diagram will describe the operation of Mature in every turn whereby Mature would have a 15% chance to spawn Koopa, with 50% chance to spawn either Walking or Flying Koopa, grow a new sprout in one of its surrounding fertile squares randomly for every 5 turns and also have a 20% chance to wither.

The starting point of the sequence diagram above is when the `tick()` method in the Mature class is 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 check whether the current location ground type is equal to Mature. This layer of checking is needed because after the reset action has been done, the tick() method of Mature will still be ongoing although the type of ground of the current location has changed to Dirt. Without this checking, this might leads to a problem where the Koopa is spawned above a ground where there is no more Mature on it.
   ➔ If true, we will call the spawn method which is located in the Mature class itself. For more detailed information about the code in the 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.

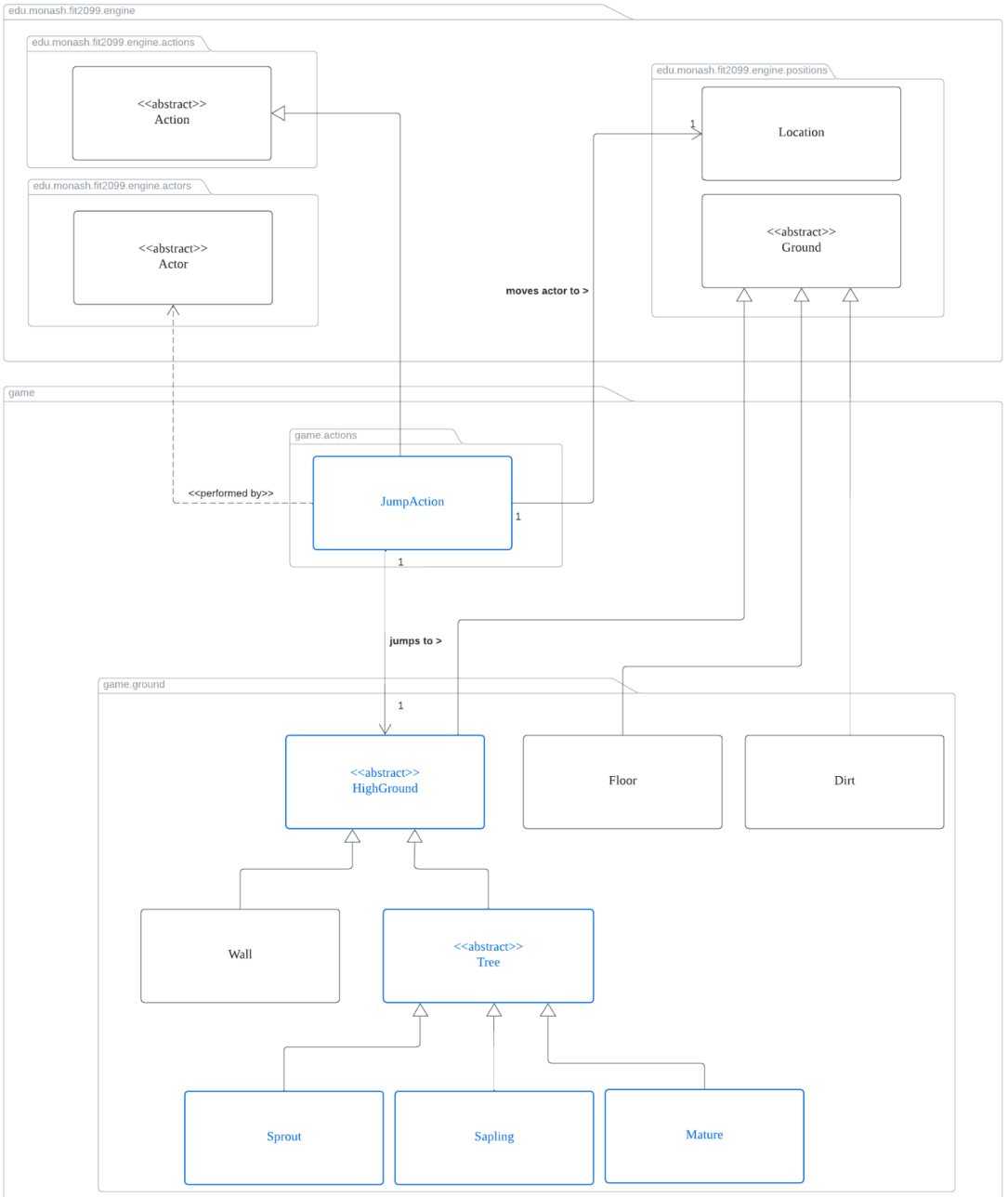
*Updates that have been done for REQ1 (sequence diagram & explanation)

I. Added in checking for 50% chance to spawn either WalkingKoopa or FlyingKoopa while spawning Koopa

REQ2

Class Diagram:

REQ2



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(`t`): 80% success rate, 20 fall damage
    - Mature(`T`): 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 fallDamge. 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.

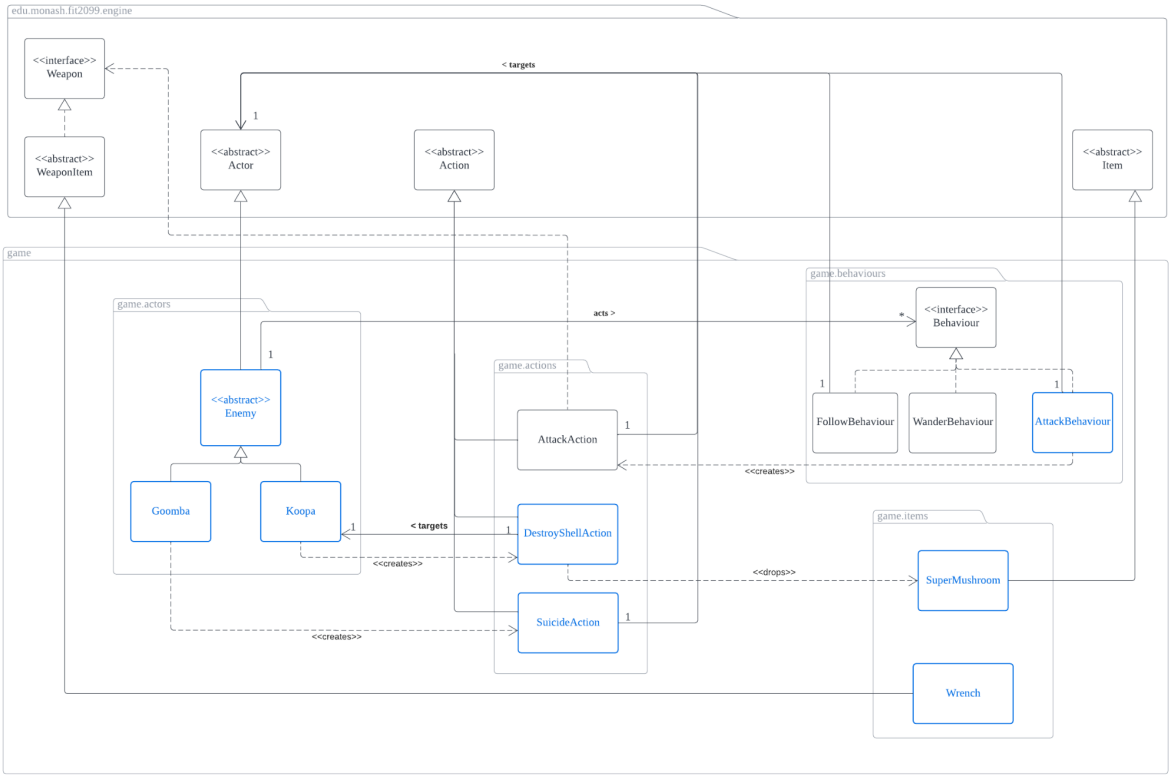
We are going to discuss Tree, Mature, Sapling, Sprout and Wall in this paragraph. Tree is an abstract class extending HighGround. Mature, Sapling and Sprout are extending Tree, which is implemented in requirement 1. Wall is a class that extends HighGround. Tree class is left unchanged except extending the class. Meanwhile, sapling, mature and sprout override the method from HighGround. Two methods setFallDamge 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 fallDamge: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 assignment 1, jumpAction has 5 instance variables: 1) actor 2) direction 3) destination 4)highGround 5) random. In assignment 2, change made is to remove the actor attribute because it is not required for the action. The change has changed the relationship between jumpAction and actor from association to dependency. Actor is only needed in the execute method as a parameter to indicate which actor is jumping. This will make the two classes to be more loosely coupled. This follows the principle that **classes should be responsible for their own properties to reduce dependency** on each other. 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. Change made here is to check if the player has the status of SUPER due to the effect of super mushroom. If it is true, we will set jumpSuccessRate to 100. 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 exits 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. In the overridden playTurn() method, the **Enemy** would check if a **Player** is next to it (by checking through all its exits to see if there is an actor, if yes, check if the actor has `Status.HOSTILE_TO_ENEMY` capability, if yes, we found a Player), if yes, a FollowBehaviour and AttackBehaviour would be added to the **Enemy's** behaviours hashMap. This is because whenever a **Player** is next to the **Enemy**, the **Enemy** would also definitely be in range to attack and then follow **Player**, hence this is why this logic would work.

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, if the player exists and is in range to attack, then Enemy would execute the AttackAction onto the player. In this AttackBehaviour, if **Enemy** is not in range to attack **Player**, AttackBehaviour's `getAction` would return null so that **Enemy**

could look at FollowBehaviour to follow the **Player**. If **Player** is not able to move to a position that allows it to be closer to **Player** compared to where its located now, FollowBehaviour's `getAction` would return null so that **Enemy** would look at WanderBehaviour's `getAction` to wander around. This logic works as whenever an **Enemy** has both the AttackBehaviour and FollowBehaviour, it means the **Enemy** is in range to attack **Player** or the **Enemy** is already engaged in a fight. Thus, this would make sure that **Enemy** doesn't follow **Player** when **Enemy** has not been engaged in a fight with Player.

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 `playTurn` 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, we chose to do it this way because we 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. 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 `playTurn`. The reason for adding it only in `playTurn` 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.

*Updates that have been done for REQ3 (class diagram & design rationale)

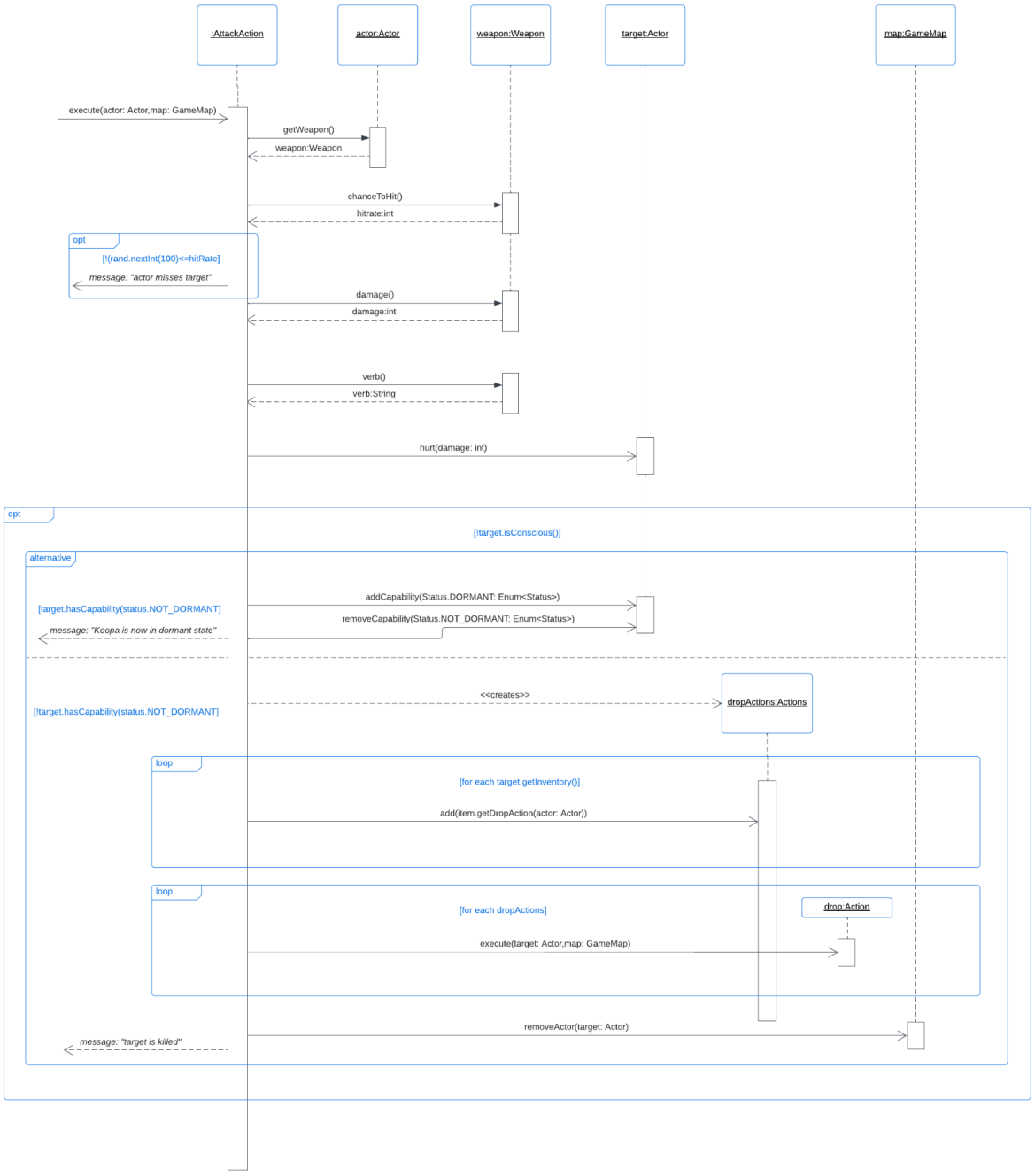
I. Class Diagram
   A. Removed dependency relationship between AttackBehaviour and WanderBehaviour.

II. Design Rationale
   A. Behaviours are now given in the NPCs playTurn instead of allowableActions. This is because we noticed a timing issue when behaviours are added in allowableActions, such that if it is the first time Mario is next to an Enemy, Enemy will not be able to attack Mario if Mario walks away from Enemy. Imagine in Turn 1 if Mario is one block apart from an Enemy, when Mario walks next to Enemy, Enemy does not have the

behaviours so it does not attack Mario. In Turn 2, when player is choosing which actions to perform, only the Enemy would be given the behaviours since Enemy's allowableAction would be executed when the actions that the player can perform is being processed. Assuming if player chose to walk away from Enemy resulting in player being one block apart from Enemy again, Enemy would try to attack Player, however Player is no longer in range for Enemy to attack Player, hence this results in a scenario where Enemy does not attack Player if it is the first time Player is next to Enemy. When the behaviours are given in Enemy's playTurn, the Enemy would check if player is next to it, if yes, the behaviours would be added directly, thus this allows Enemy to be able to directly get the action from its behaviours to attack Enemy.

Sequence Diagram:

Sequence Diagram 1

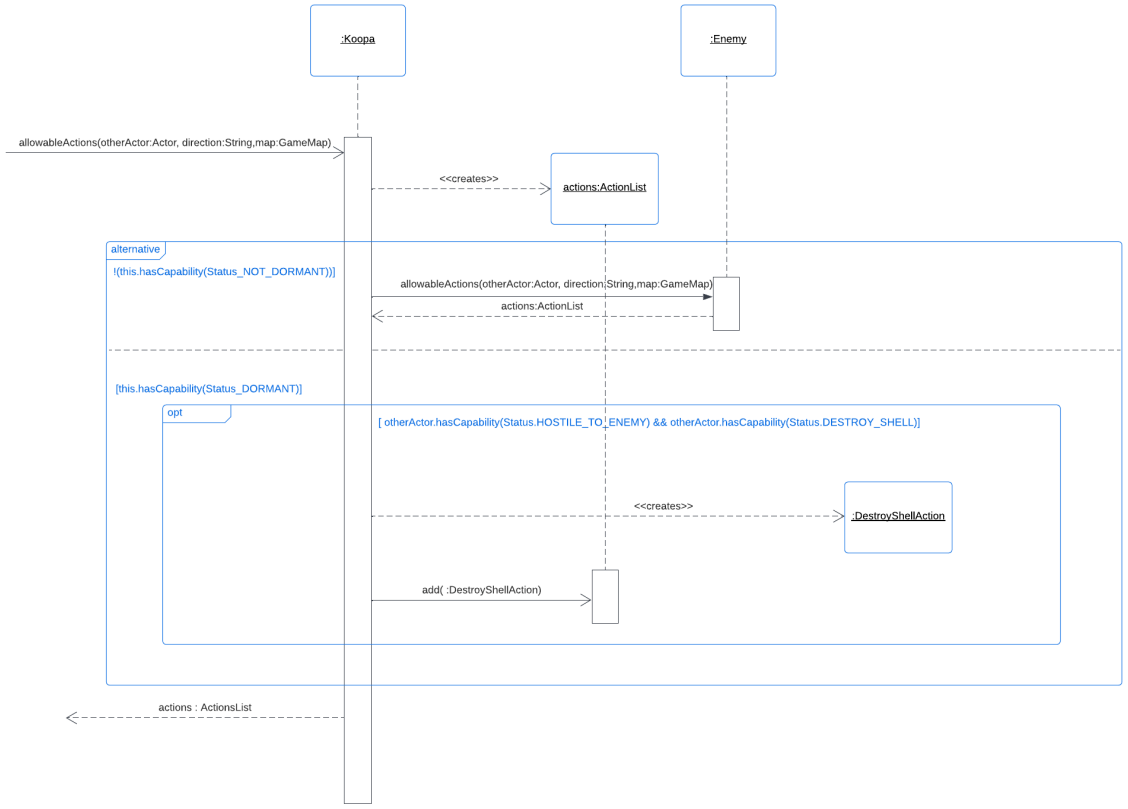


This sequence diagram's objective is to show the operation where Koopa will be given the capability Status.DORMANT if Koopa is unconscious upon being attacked. The starting point of the sequence diagram above is when the `execute()` method in the **AttackAction** class is being called. Basically, the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

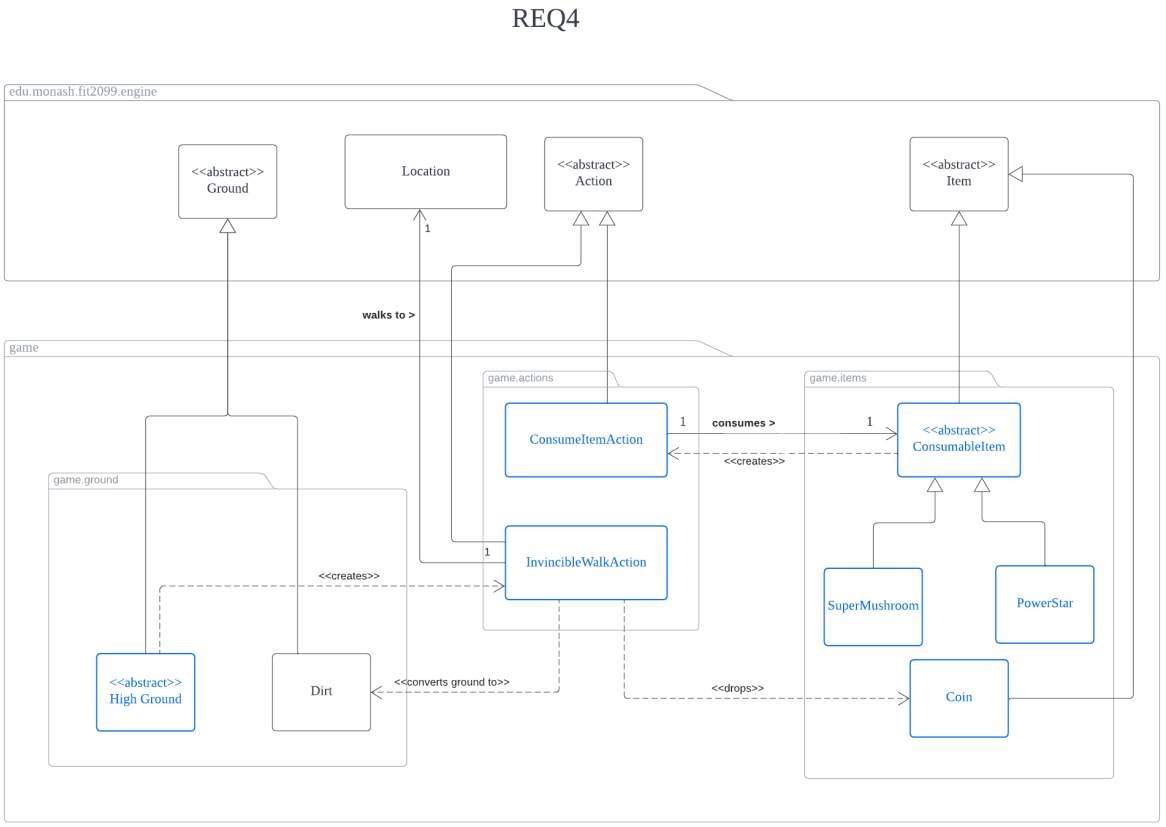


This sequence diagram would demonstrate the operation of the allowableActions of Koopa whereby Koopa would either allow actors to attack it as usual or allow actors to be able to destroy its shell or allow actors to not be able to do anything to it.

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 **ConsumableItem class** 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 ConsumableItem 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 holding the PowerStar after consuming it. In other words, we implemented it in the way that after consuming Power Star, it would remain in the actor's inventory but it doesn't give the ConsumeItemAction anymore, and PowerStar would be added the INVINCIBLE capability. By having this capability Status.INVINCIBLE, 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 ConsumableItem? 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. For SuperMushroom, regardless if it is consumed when it is on the ground/in the actor's inventory, it would be removed from ground/inventory upon consumption. However for Power Star, if it is consumed when it is on the ground, Power Star would be added to the actor's inventory as Power Star grants the actor the INVINCIBLE effect upon having it in inventory. If Power Star is consumed when it is in actor's inventory, it would remain in the actor's inventory. In both scenarios, Power Star would automatically remove itself when the invincible effect wears off. 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 ConsumableItem. 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.

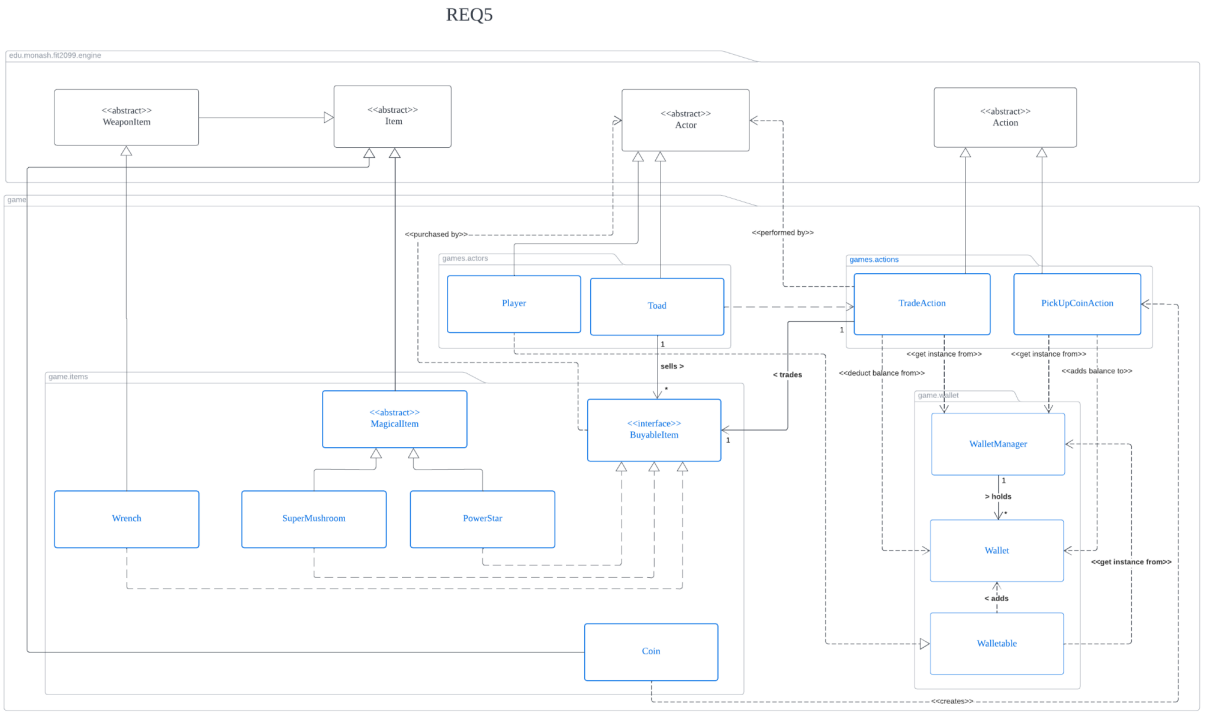
Last but not least, since PowerStar will fade and it would also have limited invincible effect, we could calculate these turns in PowerStar's `tick` method. As mentioned earlier, consumable items have two scenarios, whether it can be on the ground or in the inventory of the player, hence we would need to override both tick method's (one for when item on ground and another for when item in inventory). PowerStart 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 Power Star, the turns left counter would be resetted such that the turns left for the invincible effect to wear off is 10. If the turns left counter ever reaches 0, the Power Star would immediately be removed regardless if it has been consumed, depending if it is on the ground or in the actor's inventory. Since we have decided to let PowerStar have the responsibility of keeping track of the turns left instead of letting Player handle it, this follows the **SRP** as the invincible effect came from Power Star. Furthermore, if Player were to handle it, Player would have too many responsibilities and this violates **SRP**. In addition, by leaving Power Star in the actor's inventory and letting Power Star handle the tracking of turns also reduces dependency between Power Star and Player.

Besides, inside the PowerStar method, we also would override the toString method so that it returns "`Power Start (<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 the 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:



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 (`t`) 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 (`O`) 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 two interfaces are created or modified here, which are **TradeAction**, **PickUpCoinAction** extending Action class, **Wallet, WalletManager, Walletable interface, 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. In assignment 2, we added a purchasedBy(actor) method implemented by all the buyableItem to addItemToInventory(). The reason for creating this method is because we want to addItemToInventory(this) in the item class instead of calling addItemToInventory(buyableItem) which would be explained below. 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 is small and it only returns buyableItemPrice and puts the item into inventory when purchase is done. 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 assignment 1, tradeAction has 3 variables: actor, map, and buyableItem. In assignment 2, we decide to remove actor and map attributes as they are not required. To run a tradeAction, actor and map just have to be inserted as parameters in the execute() method. This follows the principle that **classes should be responsible for their own properties to reduce dependency** of different classes. 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. Previously, we added the item to inventory in the execute() method. However, a problem arises when we cannot add buyableItem to inventory because addItemToInventory() method only accepts Item class and not BuyableItem. It can be solve by upcasting (Item)this.buyableItem but it is not a good design. So we assign the work of adding items to inventory to the buyableItem itself. We would call the purchasedBy(actor) method of the item to add the item into the inventory of the actor for current implementation. Then, the execute() method prints 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.

For the trading process, another key implementation we had is when a Power Star is created inside Toad's inventory, the Power Star's turns left is set to 11 instead of 10, this is because we wanted the game to work in a way such that when a newly bought Power Star is passed from Toad to Player, the Power Star would have 10 turns left to consume in the Player's inventory, which is logically correct according to the specifications. If we had set the turns left to 10, after the Power Star is bought and added into Player's inventory, the Power Star would have only 9 turns left to consume, which we thought is not logical and a bad gaming experience.

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 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. We also wanted to make our game extensible and be able to accommodate for multiple players/actors, hence not only we have a **Wallet** class, we also have a **Walletable** interface that would be implemented by generally actors that can have a wallet, and also a **WalletManager** to handle all actor's wallets. Whenever a walletable actor is created, a wallet for the actor is created and registered to the wallet manager's HashMap with the actor as key, and the wallet as the value. This works because whenever we want to get the wallet of an actor, we could just access the WalletManager instance and use that actor as the key to get that wallet, and we could add or deduct balance from the wallet. This follows the **OCP** as our system is open for extension as we can handle other classes that might have a wallet in the future, thus we are not limited to just having wallet for a player,

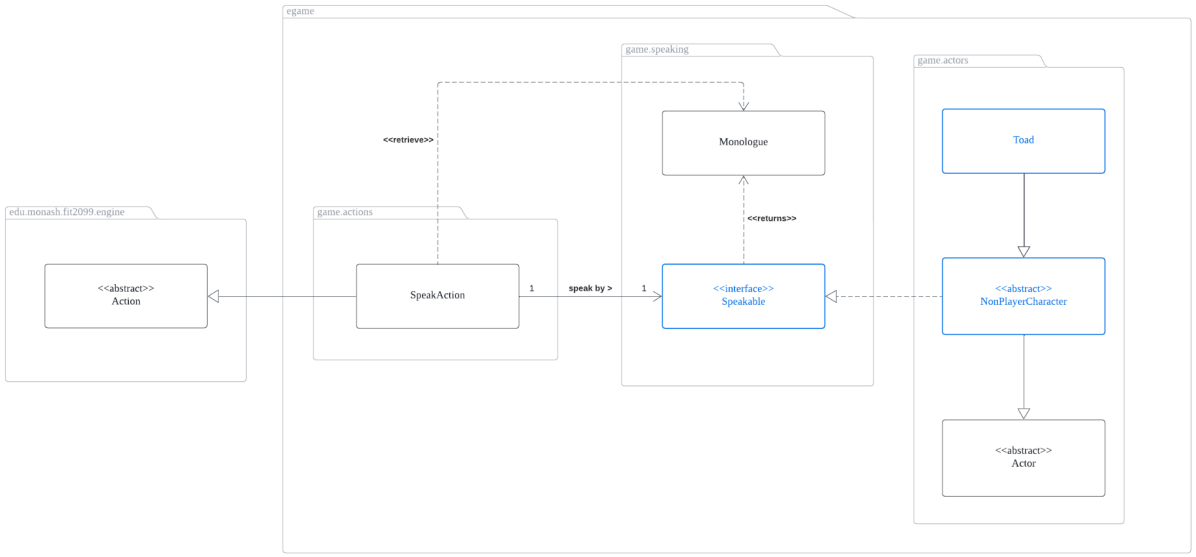
As we already have our wallet system, the key question is how would we pick up Coins that are on the ground and add the credit into our wallet balance? We did this by implementing another new action called **PickUpCoinAction**. A coin would add **PickUpCoinAction** into its allowableActions list and 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.

There was also another issue that we faced along the way, which is, as we overridden the toString method for PowerStar to also include the turns left, this results

that when player is near Toad to purchase items, the turns left would appear in the console, for example “Mario buys Power Star ($600) (10 turns left)”, which is very weird. Hence we added another checking in PowerStar such that when PowerStar’s counter has 11 turns left, it would not return the turns left along with its name. This works because the PowerStar created in Toad has 11 turns left (as explained earlier).

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.

In order to handle which sentence is possible for the speaker to speak, we have created a new interface called **Speakable** interface. The interface will be implemented by the **NonPlayerCharacter** abstract class which is the subclass of Toad since all the NPC can speak right now. For the current requirement, Toad needs to override the generateMonologue() method that are the abstract method in the **Speakable** interface. In this method, we will create a local variable of ArrayList<Monologue> and add the new instance of **Monologue** into the ArrayList<Monologue> such that each **Monologue** objects represents a speakable sentence. For instance, there are 4 possible monologue that can be spoken by Toad, thus we will add 4 newly created and initialized **Monologue** objects into the ArrayList<Monologue>.

To elaborate more on filtration of sentences according to the scenario, we will first explain on the **Monologue** class. As mentioned before, each **Monologue** object will represents a sentence, hence, the Monologue class will have 3 attributes,

1. (String) sentence: the sentence that can be spoken by the speker
2. (Speakable) speaker: the speaker that speak the sentence
3. (boolean) isAvailable: is the sentence available for the speaker to speak

In Toad's generateMonologue(Actor actor) method, we can done the checking by,

1. Initialized two boolean variables, hasWrench and isInvincible to false
2. Since all the actor can self-speaking according to assignment 3 REQ5, we will check if the target is not equal to itself.
3. If false, we will proceed to the Monologue adding parts mentioned above since there is no special condition that the speaker would not speak any of the sentences. And all the Monologue objects isAvailable attribute will be initialized to true
4. if true, means the speaker is talking to another actor. Then we will need to check whether the target (player in this case) holding wrench or is invincible and saved the result to hasWrench and isInvincible attributes.
5. Then, we will add initialized new Monologue objects and add them to the ArrayList<Monologue>

```
sentences.add(new Monologue("You might need a wrench to smash Koopa's hard shells.", this, !hasWrench));
sentences.add(new Monologue("You better get back to finding the Power Stars.", this, !isInvincible));
sentences.add(new Monologue("The Princess is depending on you! You are our only hope.", this, true));
sentences.add(new Monologue("Being imprisoned in these walls can drive a fungus crazy :(", this, true));
```

The isAvailable attribute in Monologue object will be used in filtering the sentences in SpeakAction `execute()` method.

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 new SpeakAction(this). **SpeakAction** is a subclass of Action which we created to handle the conversation, including the conversation between the player and the Toad. SpeakAction class has an attribute speaker which is data type of Speakable which indicates which actor is the one that is responsible to generate the monologues and speak. The implementation of this SpeakAction class follows the **Single Responsibility Principle (SRP)** as this class will only have one responsibility, which is to handle the retrieval of possible Monologue from the speaker which is Toad in this case and randomly choose the Monologue and returns the sentence as a String. The **SpeakAction** class follows the **SRP** since it is just a class that retrieve the possible monologues and randomly chooses one of them to return. Basically, just like the name of SpeakAction class, it just handles what to speak and speak out the monologue chosen.

We would override the `execute()` method in the **SpeakAction** class to create a new ArrayList of data type Monologue - availableMonologues. We will get the possible monologue by calling the generateMonologue() method of the speaker and check whether it is available, if true, then we will add it to the availableMonologues. After that, we will just randomly choose the sentence by using `availableMonologues.get(rand.nextInt(availableMonologues.size()))` and return it.

The design above that involved Speakable interface, SpeakAction class, Monologue class follows the **open-closed principle (OCP)** since it is open for extension and closed for modification. This is because the new actor that can speak will just need to

implement the **Speakable** interface and override the necessary method. The design of Speakable interface also fulfil the **Interface Segregation Principle (ISP)** since the interface is small enough and only implemeted by the classes that can speak. Hence, the system is more extensible and the classes that implement this interface can fully substitute the interface.

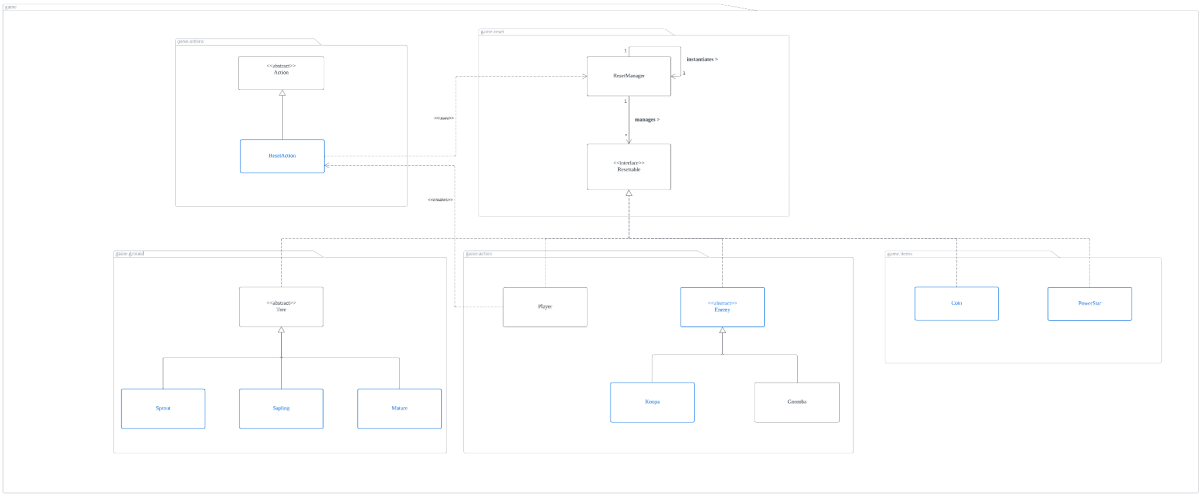
*Updates that have been done for REQ6 (class diagram & design rationale)

I. Class diagram:
   A. Added in a new **Speakable** interface
   B. Change the code implementation of the **Monologue** class and **SpeakAction** class
   C. Added in **NonPlayerCharacter** abstract class that extends by Toad

II. Design Rationale:
   A. Reimplement the handling of conversation between Toad and Player
      1. Older version:
         a) Hard coded the **Monologue** class to store and do the filtering of possible sentences between **Toad** and **Player**
         b) **SpeakAction** only acts as a middle man and call the **Monologue** class to do the necessary actions which violates the design smell of middle man.
      2. Newer version:
         a) Each instance of **Monologue** class now represents a single monologue sentence
         b) **SpeakAction** will in charge of choosing one of the sentences from the possible sentences that can be spoken and return it
         c) `generateMonologue()` method in the **Speakable** interface will be override by all the classes that implement it.
         d) `generateMonologue()` method will be used to filter and select possible sentences that can be spoken by the speaker

REQ7

Class Diagram:



Requirement:

- Add in an option for the player to reset the game
- Game can only be reset 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, **PowerStar** 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 use `addCapability()` method to add the `Status.RESET` to all the **resettable** objects except **Player**. The actual resetting is done in the tick or playTurn method for all **resettable** objects except **Player.** The explanation is broken 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 happens in the `tick()` method as below:

1. When the `tick()` method in subclasses of Tree is called, they will call the `super.tick()`.
2. In the `tick()` method, we would check whether the instance of Tree has the capability of RESET by using `this.hasCapability(Status.RESET)`. If yes, it will proceed to step 3, else it will continue the normal `tick()` routine.
3. It will check whether the **Tree** object meet the 50% chance of dying by using `if ((rand.nextInt(100) <= treeDieChance))`
4. If true, a new instance of **Dirt** will be created and set the ground to dirt (the Tree object has died). If false, it will continue with the next step.
5. It will remove every **Tree** object's capability of RESET by using `this.removeCapability(Status.RESET)` so that they will not be reset anymore.

We also added another layer of checking in the `tick()` method of Sprout, Sapling and Mature class to check whether the ground type of the current location is equal to the respective class. This implementation is needed to avoid the instance of these classes

executing the unique spawning ability after resetting since the `tick()` method will still continue to run for the current turn even after it has been removed from the current location

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.

The actual execution of reset will happens in the `playTurn()` method for both of the classes:

1. In their `playTrun()` method, it will firstly check if the instance of the class (**Koopa** or **Goomba**) has the capability of RESET or not.
2. If true, it will return a **SuicideAction** and it will be removed from the map when the action get executed.
3. If false, it will continue to perform the normal `playTurn()` routine.

Reset player status

We would override the `resetInstance()` method in the **Player** class and **PowerStar** class.

The actual implementation of resetting will be implemented in the `resetInstancce()` method for **Player** and in the `tick()` method for **PowerStar**.

Player Class

1. When `resetInstance()` method is called, it will first reset player status by removing their capabilities from super mushroom using the following code
   a. `this.removeCapability(Status.SUPER)`

`PowerStar` Class

1. When the `resetInstance()` method is called, it will check whether the PowerStar instance has the capability of INVINCIBLE.
2. If true, it will add the RESET capability to it
3. If false, it will just continue its normal operation

4. During the tick() method of the PowerStar instance, it will removed the PowerStar objects from the player's inventory if it has the RESET capability by using the following code,
    a. If (this.hasCapability(Status.RESET)) - for checking purpose
    b. consumedBy.removeItemFromInventory(this) - removing it from the inventory means resetting the player's status.

Heal player to maximum

In the same resetInstance() method in the Player, we will add a few codes to reset the player's health.

1. It will heal the player to its maximum health by using `this.heal(this.getMaxHp())`
2. Lastly, we will remove the RESET capability of the player by using this.removeCapability(Status.RESET) so that it will not be reset in the following turns again.

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 happens in the `tick()` method,

1. Firstly, it will check if the **Coin** instance has the RESET capability by using `this.hasCapability(Status.RESET)`
2. If true, it will remove the coin from its current location by using `currentLocation.removeItem(this).`
3. If false, it will continue with normal execution sequence.

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.

The actual execution of ResetAction will be as the following,

1. In the `execute()` method of ResetAction, it will trigger the resetting of all **Resettable** objects by using `ResetManager.getInstance().run()` which loops through all the instance of **Resettable** and triggers their `resetInstance()` method.
2. The player will be instantly reset once the resetInstance() method has been called and a RESET capability will be added into the other **Resettable** objects and execute the actual resetting in their `tick()` or `playTurn()` method.
3. After that we will use ResetManager.hasbeenReset() to set the attribute “ableToReset” to false so that the game cannot be reset again.
4. Lastly, return a String message to indicate the game has been reset.

We have also added a new static boolean attribute called **ableToReset** for **ResetManager** class that keeps track of the reset status of the whole game, i.e., whether it has been reset before or not. ableToReset will be true as default and it will be changed to false through a method called `hasBeenReset()` after the **ResetAction** has been executed once.

To provide the reset action as one of the allowable actions to the player, we will check if we can reset the game through `ResetManager.getAbleToReset()`. If it returns true, it will add ResetAction as one of the allowable actions, else it will continue to continue with normal checking.

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.