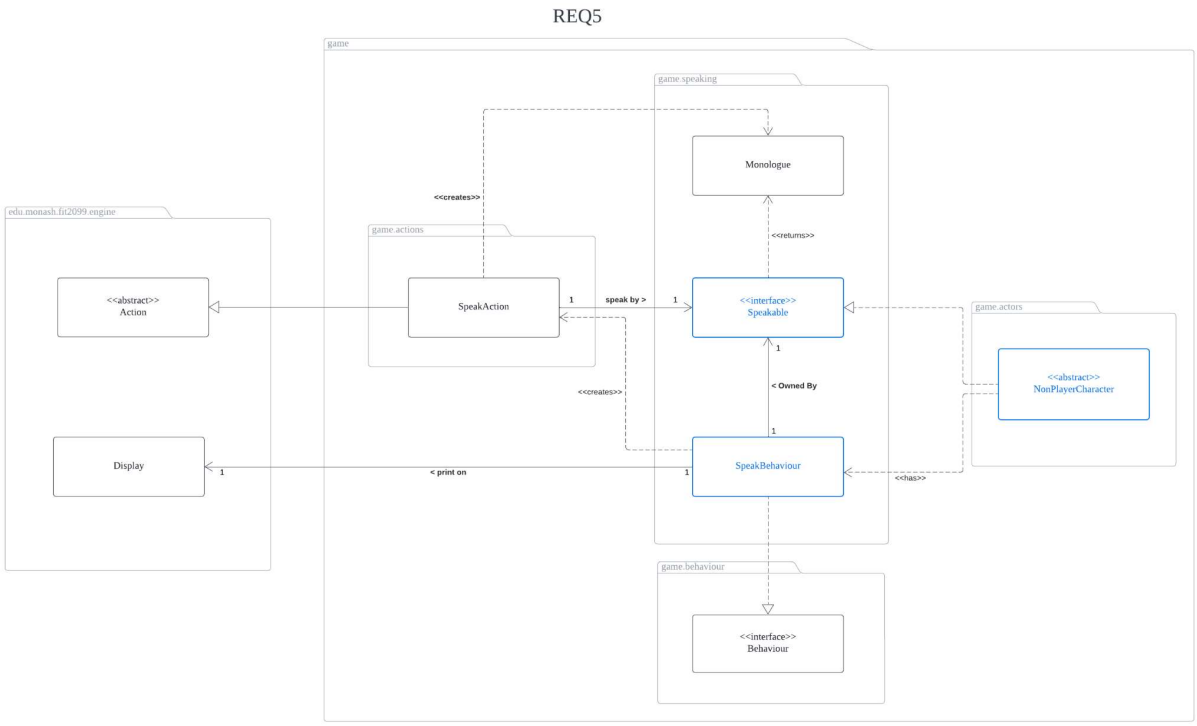
REQ5

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,

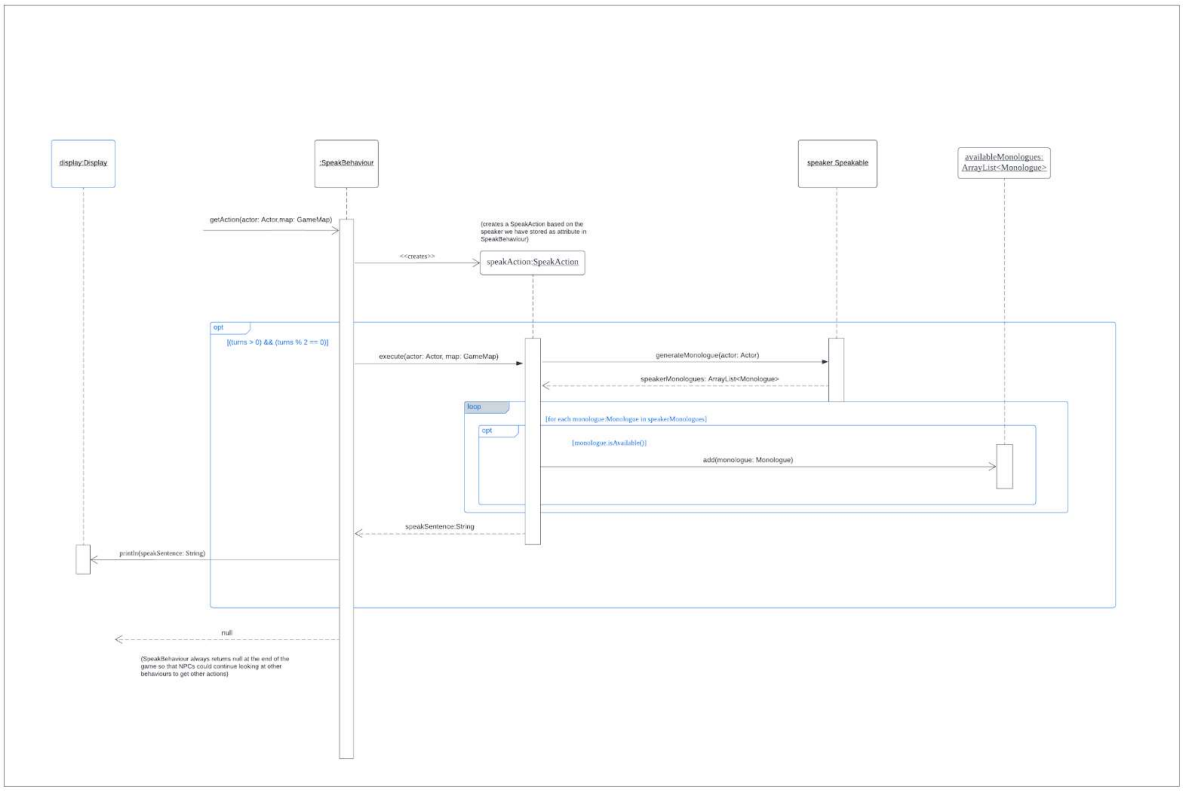
1. We will create a new **SpeakAction** object which passed in the speaker to initialise by using `SpeakAction speakAction = new SpeakAction(speaker);`
2. Then, we will increment the turns by 1. This has to be done before the checking of the even turns to ensure that the speaker can speak in the second 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.

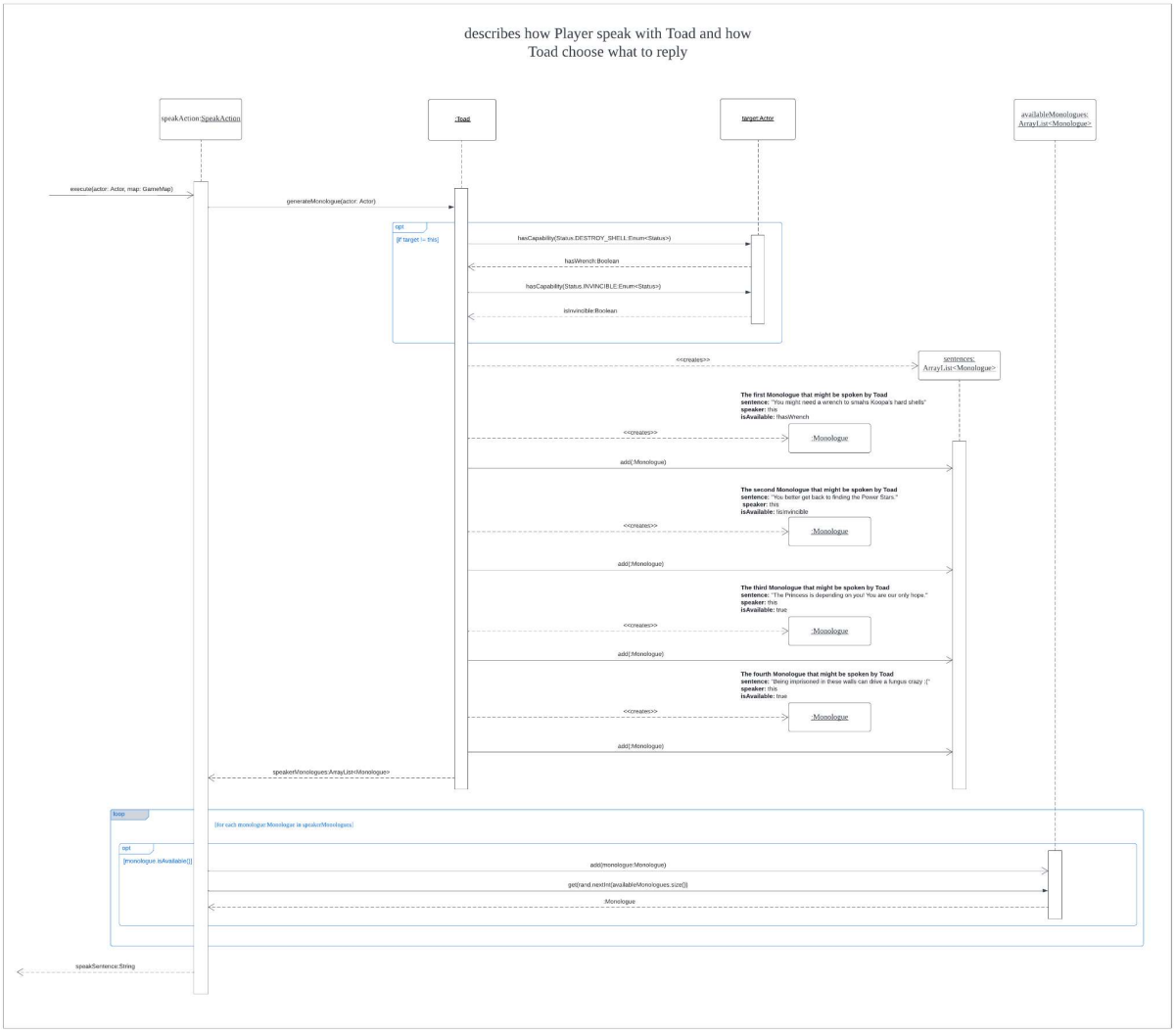
Sequence Diagram:

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.