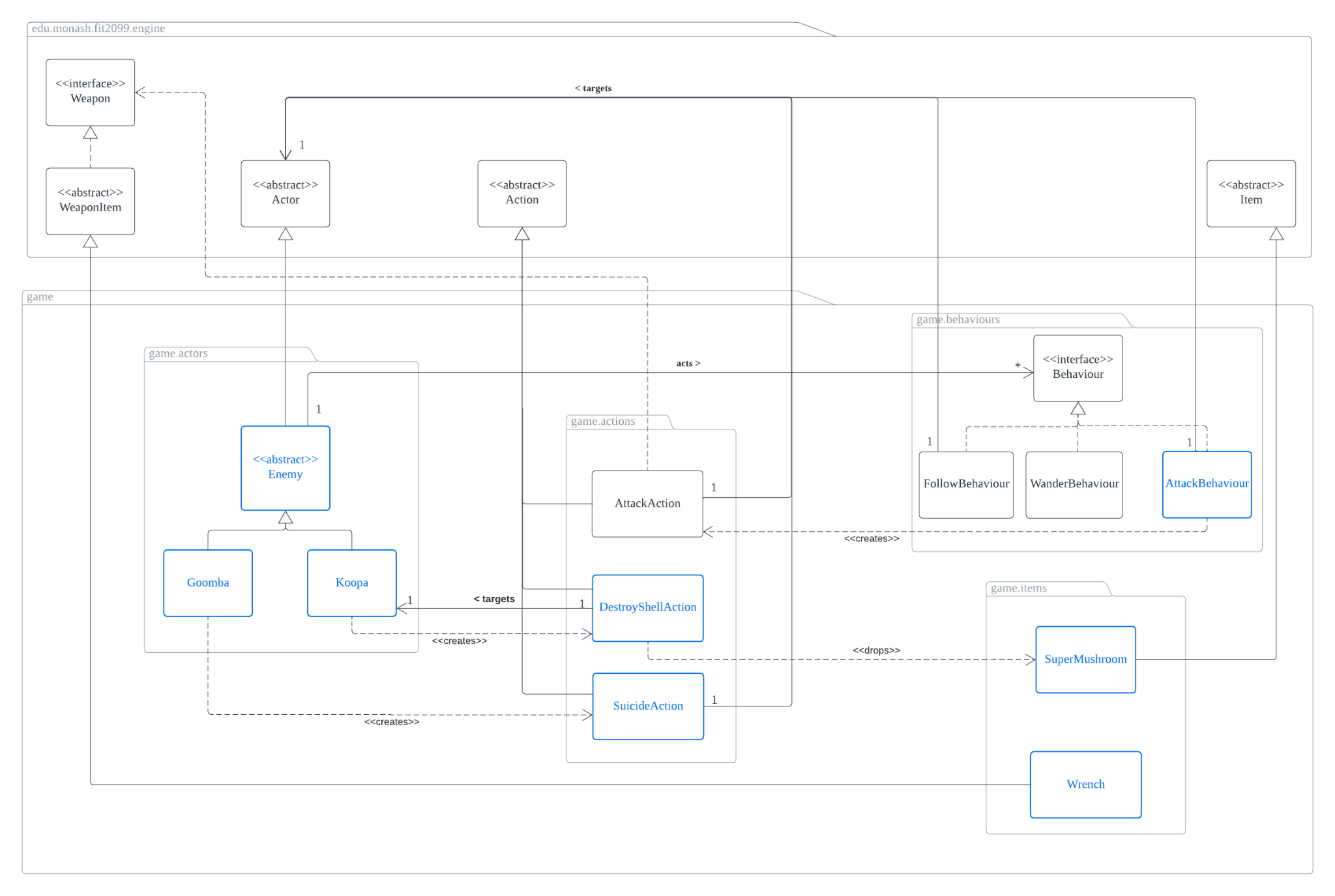
**REQ3**

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.

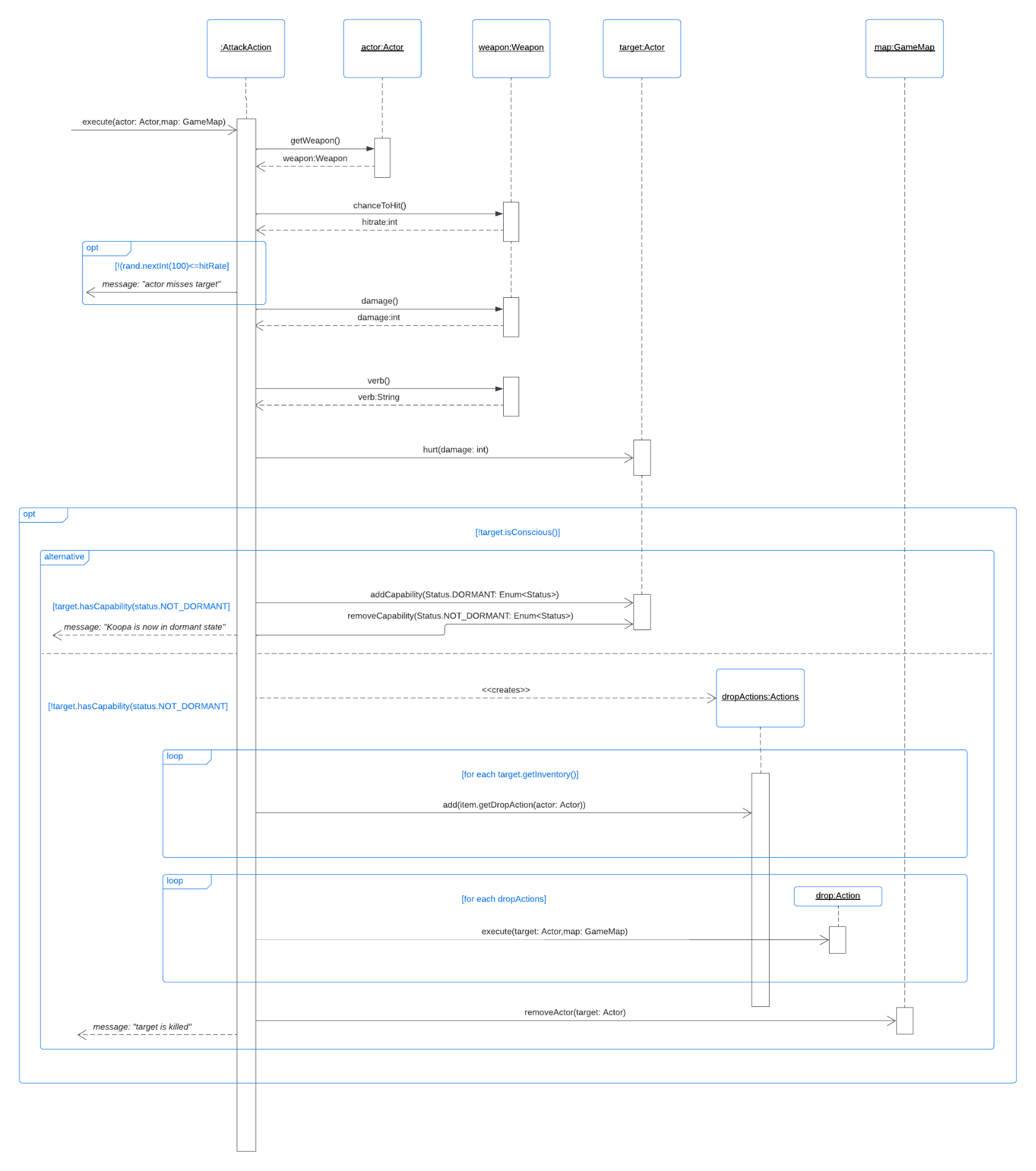
All in all, these would be all the changes made to the existing classes as well as newly created classes in order to implement all of the requirements needed for REQ3. To sum it all up for each classes’ responsibilities, Goomba and Koopa would extend a newly created abstract class Enemy and any Enemy 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)

1. Class Diagram
2. Removed dependency relationship between AttackBehaviour and WanderBehaviour.
3. Design Rationale
4. 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.

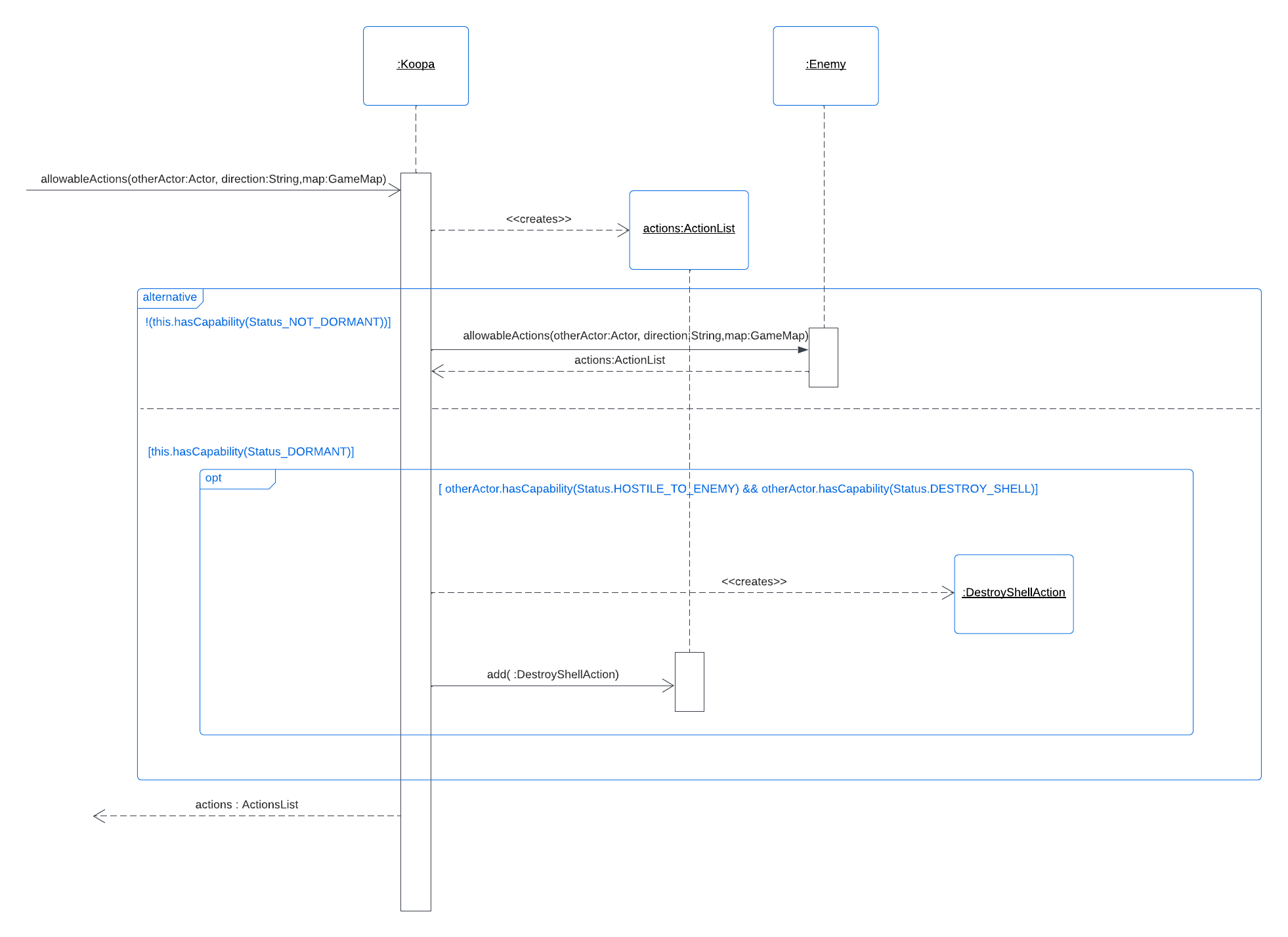
1. This part of code is basically the code that will be executed if there is no special condition.

* 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.