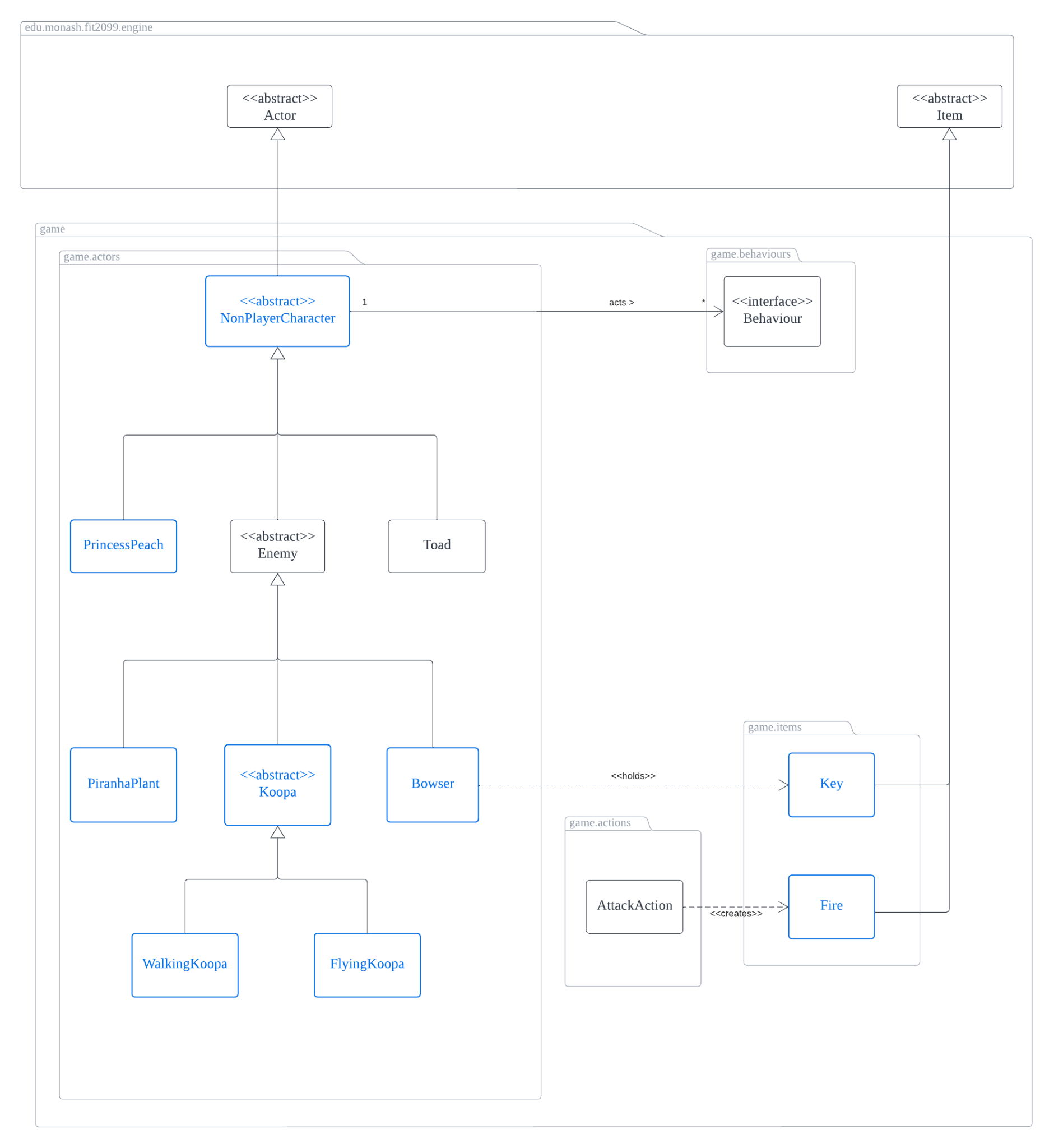
**REQ2**

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**.