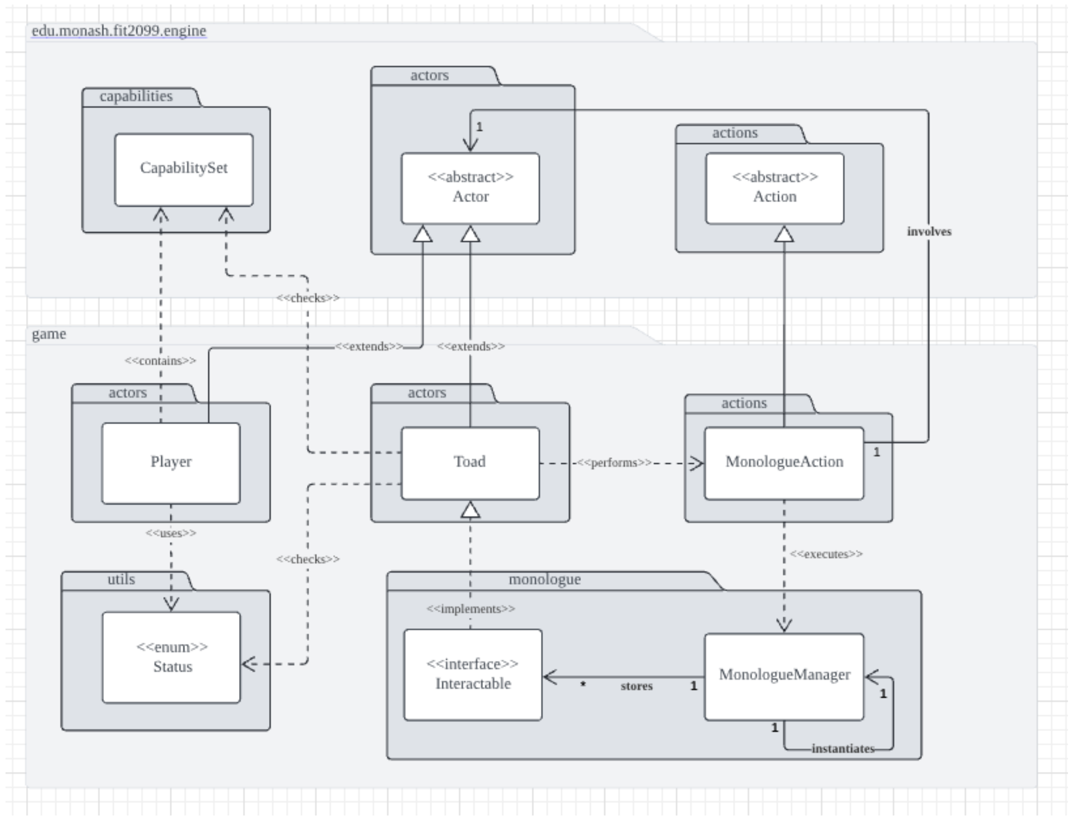
Requirement 6

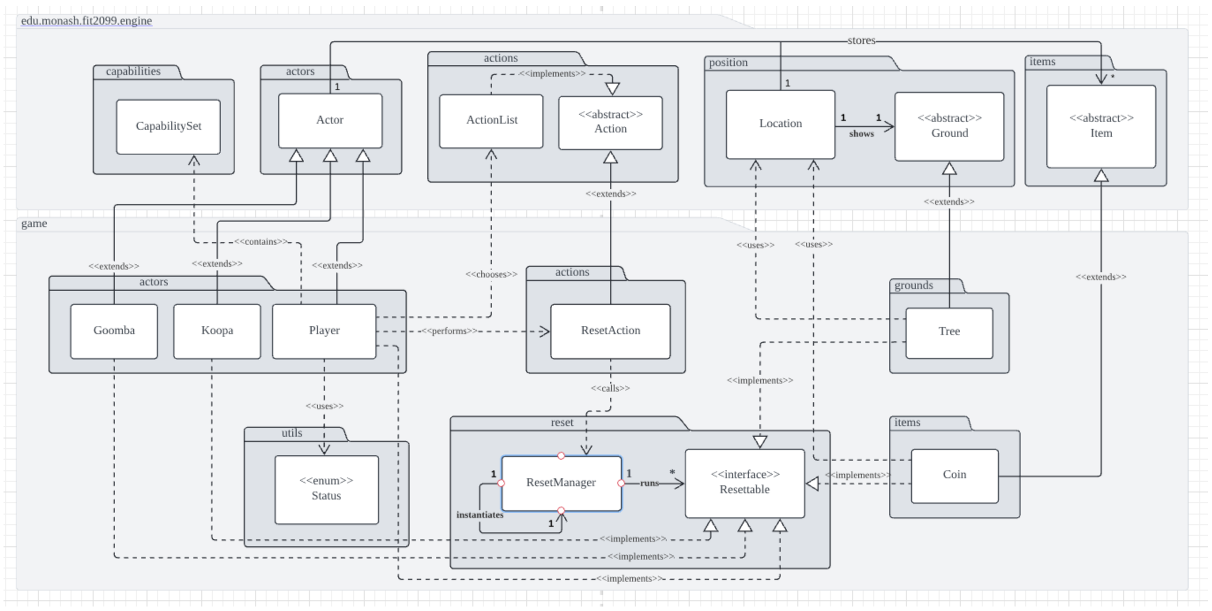


Roughly how it works:
For this requirement, we have created a MonologueAction class which extends from the abstract Action class to implement the monologue performed by the Toad to the Player.

Why use the Monologue Manager and Interactable interface?
An alternative approach is to let the Monologue Action handle the monologues for Toad. This is not a good approach as this breaks the Single Responsibility Principle (SRP) as one class is responsible for handling all the monologues for every Actor which implements it. Instead, the Interactable interface created allows any class which implements it to handle its own monologues. Then through the MonologueManager, we can access the methods from within classes which implement the Interactable interface. This design reduces a lot of dependencies and also makes the program much more scalable, as it is easier to add more objects which can Interact with the player without changing much of the existing code.

Why use Enums?
In this requirement, we have to check whether the player has certain capabilities in order to decide whether a monologue can be displayed or not. In our implementation we have decided to use Enums to check whether a player has certain capabilities. Another design is to just use String literals to define whether an object has a specific capability. This is not a good design as it can cause code smells to appear as String literals might cause confusion in what a particular capability means, and there could be many redundant String literals coded in the program, which is not ideal. Thus, Enums is a good way to solve this problem as it clearly defines what a capability is and it can be accessed easily throughout the whole game package without needing to change its definition.

Requirement 7



Design Rationale:

Roughly how it works:
For this requirement, we have decided to create a Reset Action function which extends from the Action abstract class. The reset action function will run the Reset Manager, which will iterate through a list of objects implementing the Resetabble interface, and call the resetInstance() for the objects. At the start of the game, the player is given a status to be able to use the Reset Action function. This status is then removed from the player after the Reset Action has been performed to only allow the player to use it once.

Why the use of Reset Manager and Resettable interface?
The alternative of this is to iterate through the whole GameMap to remove the Coin and convert the Trees to Dirt when Reset Action is called. This can be extremely inefficient especially if the game map is huge and many checks have to be made. Thus, the use of Reset Manager saves the hassle of iterating through the whole GameMap, by storing a list of Objects which are Resetabble, through implementing the Resetabble interface. In this way, each Object is responsible for the deletion/transformation of themselves, instead of relying on the Game Map class to do so.