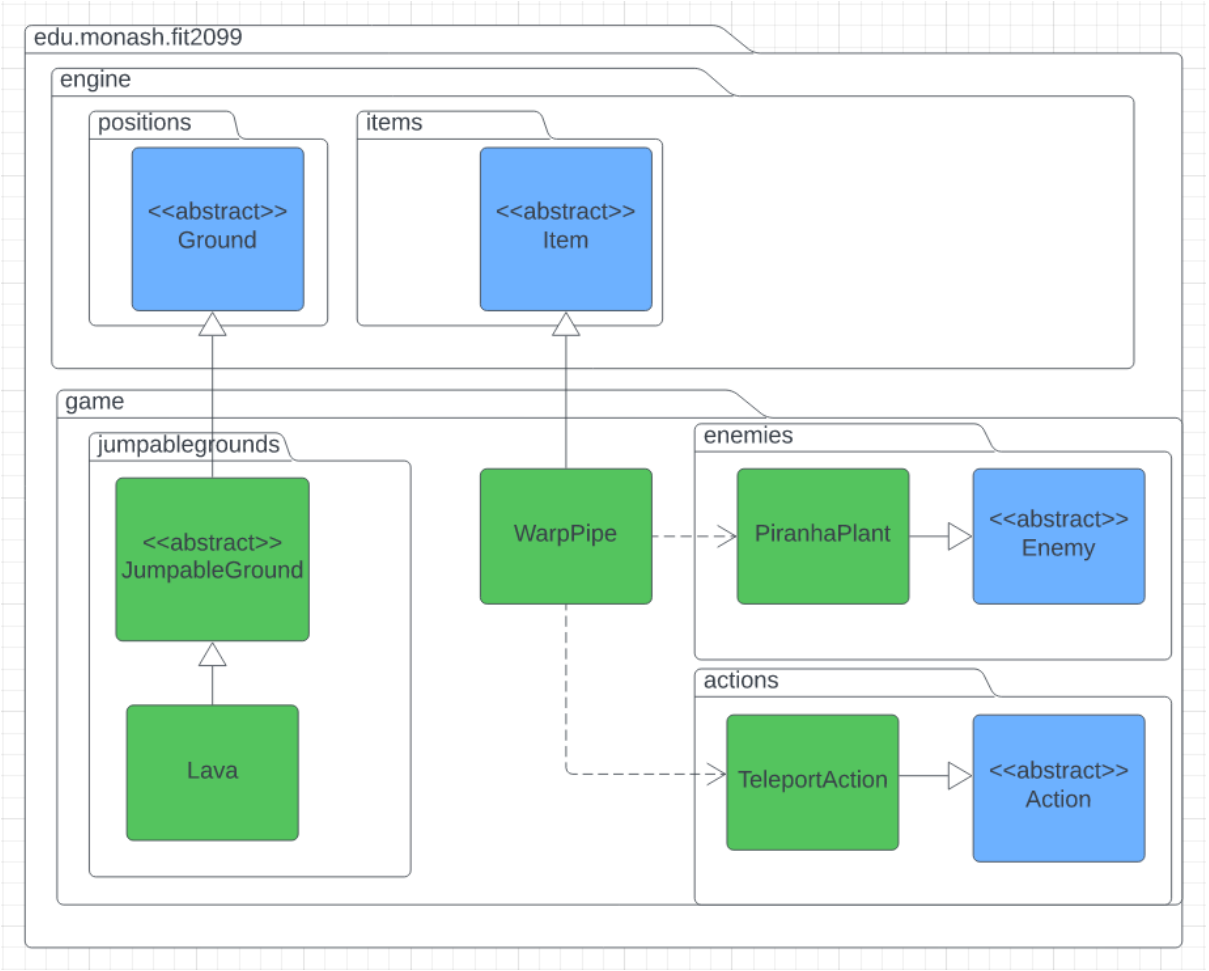
FIT2099 ASSIGNMENT 3

By Joel Atta, Jack Patton & Yepu Hou

Class diagram details:
Blue classes are classes that, as of starting that specific requirement, already existed in the project.
Conversely, green classes are classes that did not exist before starting that requirement and were added during it. Also, pre-existing classes that had their class header modified are green, such as when a non-abstract class that already existed is made abstract.
This means when a class is first created it will be in green, but if it is referenced in another UML diagram in a later requirement it will be in blue. Therefore, if you see a new class and don't understand its purpose, you can scroll up until you find the requirement in which that class is green, where it will be explained.

Req 1: Lava Zone

Class Diagram:



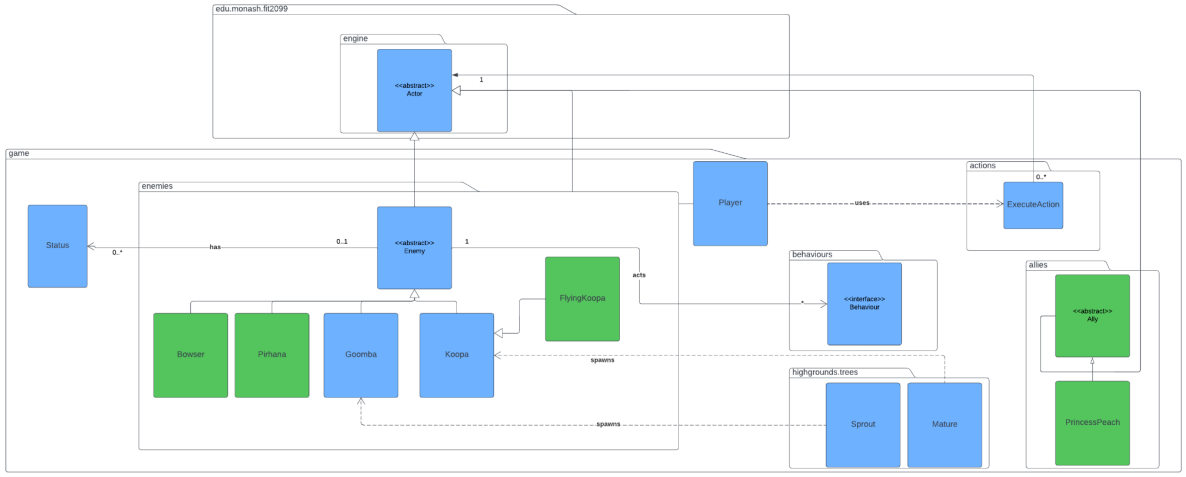
Design Rationale:

In similar roguelike games, the player is typically queried before they commit a dangerous action in order to ensure it wasn't a mis-inputted command, e.g. "Really jump into lava? Y/N". I didn't want the player to accidentally walk into lava, especially as we don't have an inspect command that can be used to describe what the ASCII character selected actually is, meaning a player might not know that L means lava is there and as such step onto it without meaning to. While this confirmation of stepping into lava would have taken more effort than it was worth for something outside the assignment specifications, I decided to instead make Lava a child of HighGround, meaning the player would have to deliberately select a jump action jumping into Lava and could not simply walk into it. However it didn't make much sense for something lower down to extend HighGround, so I renamed HighGround to JumpableGround. This allows for tiles that need to be jumped to that aren't necessarily above the player, such as jumping down a pit.

Initially, the WarpPipe class was a Ground object, specifically a JumpableGround. While the requirements mentioned using a JumpAction to get on top of the WarpPipe, they didn't mention a success chance or damage taken on a failed jump. This implied to me that Mario has a 100% chance of jumping onto the WarpPipe, so I instead decided to allow Mario to simply walk onto the pipe instead of jumping. This also allowed me to easily fix an issue where I couldn't create a move action targeting the 2nd map - both aspects were addressed by turning WarpPipe into a child of Item instead of JumpableGround.

Req 2: More Allies and Enemies!

Class Diagram:



Design Rationale:

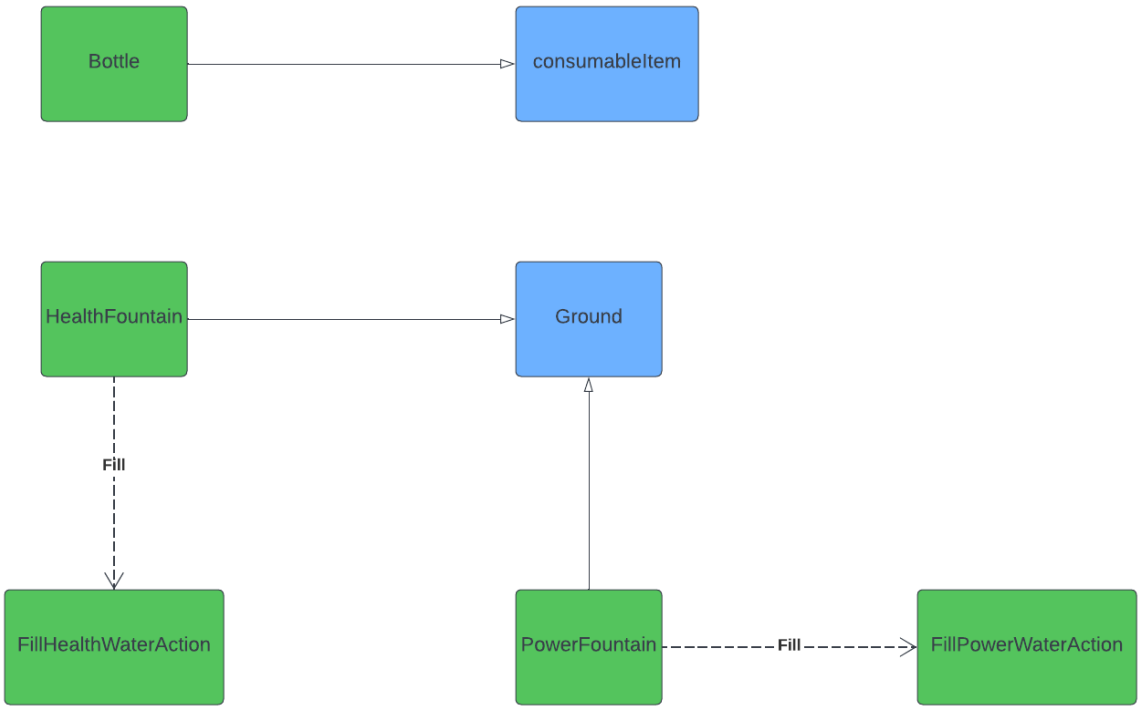
PrincessPeach	Mario's damsel in distress! Her release ends the game.
Bowser	Mario's archnemesis! He spews fire when he attacks and he must be defeated to obtain the key which is used to release Princess Peach and end the game.
PiranhaPlant	A plant that guards the warp pipes.
FlyingKoopa	A variation of Koopa that can fly over walls and trees and that also has higher health.
Ally	An abstract class which Mario's allies

	extend. In this assignment, Princess Peach extends from this class. Sets behaviours for allied NPCs
Key	Key dropped by Bowser and used to release PrincessPeach
Fire	Used by Bowser and FireFlower to inflict ‘burn’ on actors

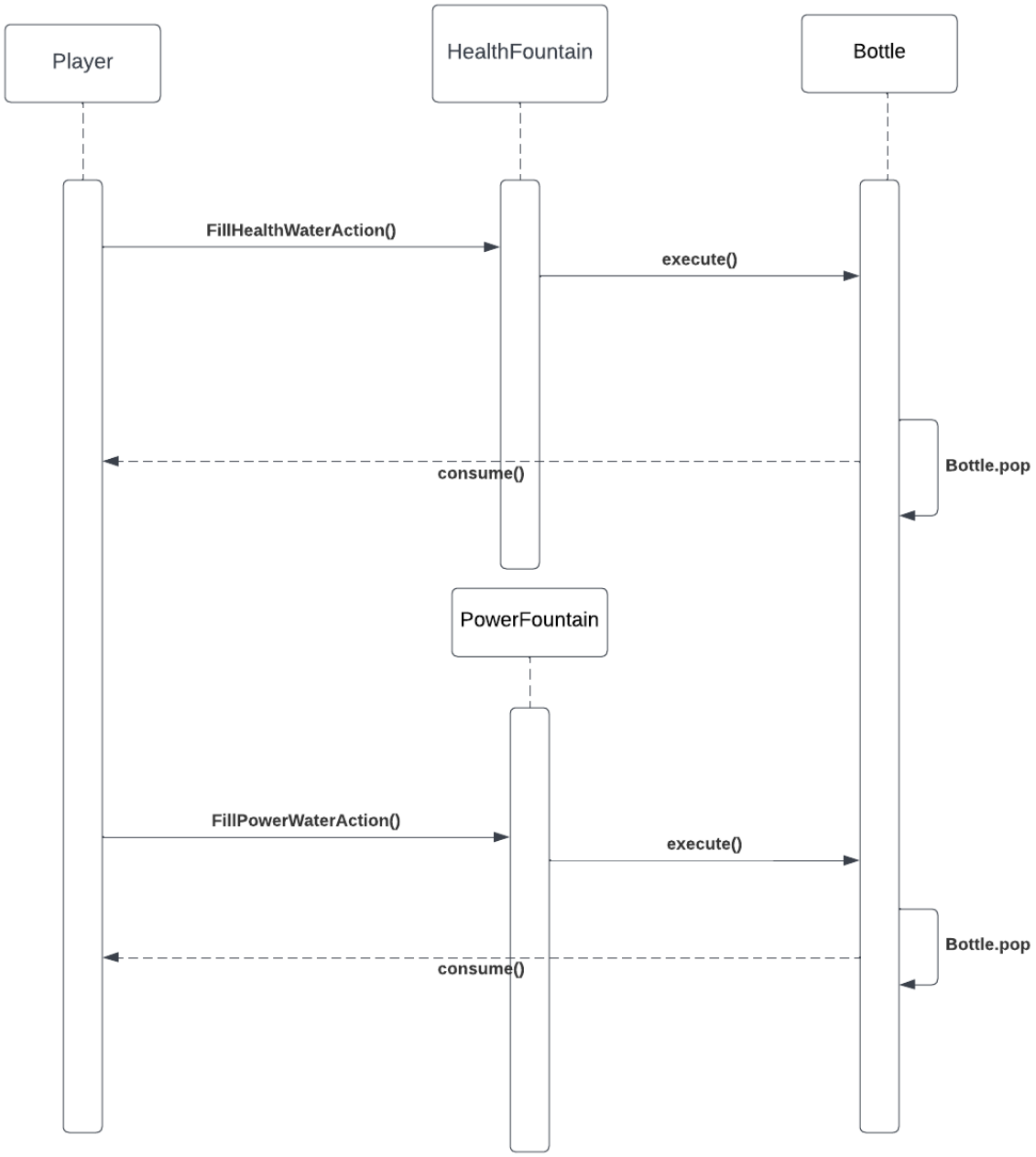
The design of this requirement builds heavily off of the work done in the previous two assignments, most notably making use of the Enemy abstract class when adding in the three new enemies: Bowser, Flying Koopa and Piranha Plant, as was intended from the beginning. In a similar vein, since we now had to implement an Ally (Princess Peach) I chose to create an ‘Ally’ abstract class for allied NPCs. As there is only Princess Peach in this one it didn’t matter whether or not I did this, however if we were to continue this assignment into the future in theory, having this Ally class so that allied NPCs can perform actions (based on behaviours) automatically is extremely beneficial, and it helps that they would be differentiated from regular actors and/or enemies. In line with the SRP, this Ally class has a very specific responsibility as is paramount to good program design. The other element crucial to this task was the creation of a Flying Koopa. At its core this is still a Koopa and thus it should be treated like one with additional capabilities, and so it is extended from Koopa. Whilst this can be seen as poor design since it extends from a concrete class, since it is only a single instance/type of Koopa it makes the most sense to do it this way, and in addition, the child class does not make enough changes to violate the Open-Closed Principle. There was also a focus on not violating LSP, since certain actions are linked to a Koopa, FlyingKoopas should not break the program when the same actions are performed upon them, and this is true for the way the program has been set up. Note that the new Enemy types also make use of the Resettable interface and its corresponding functions.

Req 3: Magical Fountain

Class Diagram:



Sequence Diagram:



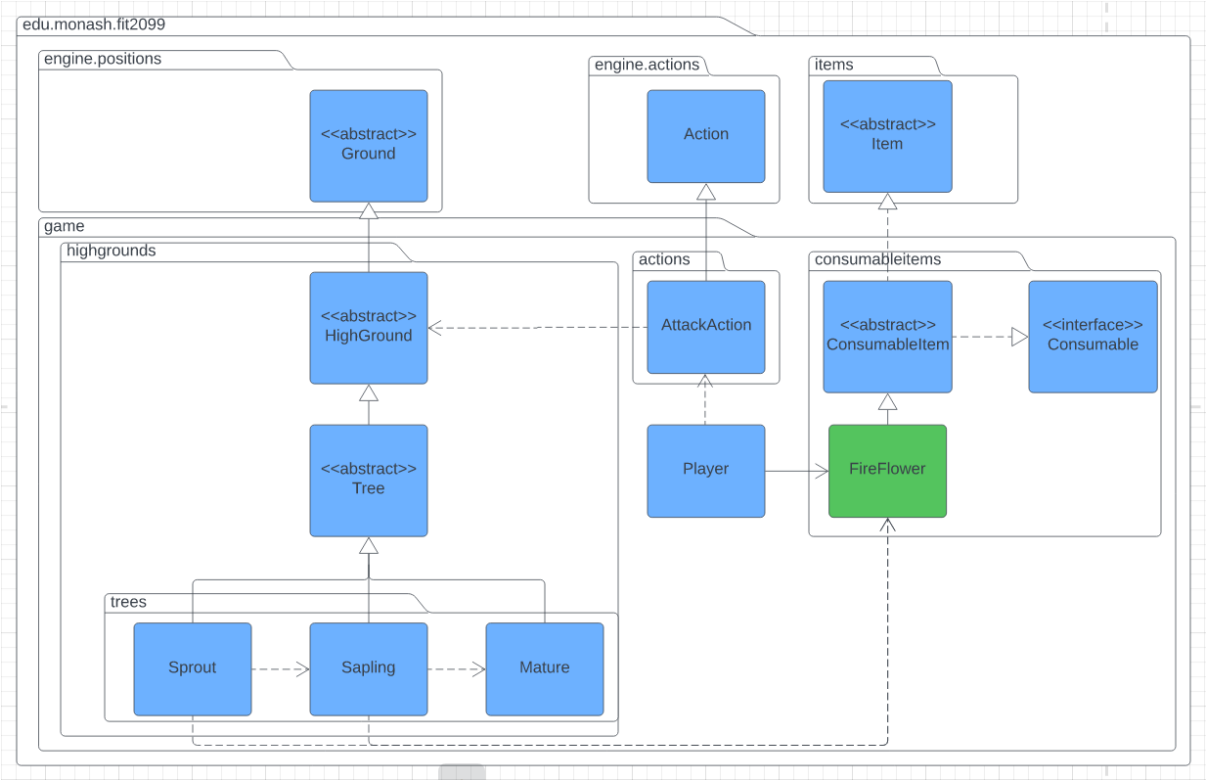
Design Rationale:

The design for requirement three is having two fountains which are health fountain and power fountain. This satisfies the singularity purpose. A bottle class will be initialised as a stack so that every time we fill up the water from different fountains the water will be pushed to that bottle as a string. The bottle is extending from the consumable item this means that in the consume action it will have an if-else statement which will check whether the next element in the bottle is either power water or health water, and will give specific power according to the water that the player has consumed. There will also be two different types of fill water actions one to fill the health water and one to fill the power water. Every time this is called it will push either health water or power water inside the bottle.

HealthFountain	A fountain which extends from the ground that has a fill health water action attached to it.
PowerFountain	A fountain which extends from the ground that has a fill power water action attached to it.
Bottle	A bottle extends to the consumable items which are implemented as a stack.
FillHealthWaterAction	When the Mario goes near to health fountain he will have an action to fill up different types of water and it will be pushed in to a bottle like a stack.
FillPowerWaterAction	When the Mario goes near to power fountain he will have an action to fill up different types of water and it will be pushed in to a bottle like a stack.

Req 4: Flowers

Class Diagram:



NOTE: Fire also is extended from Item, but is not consumable nor is it portable

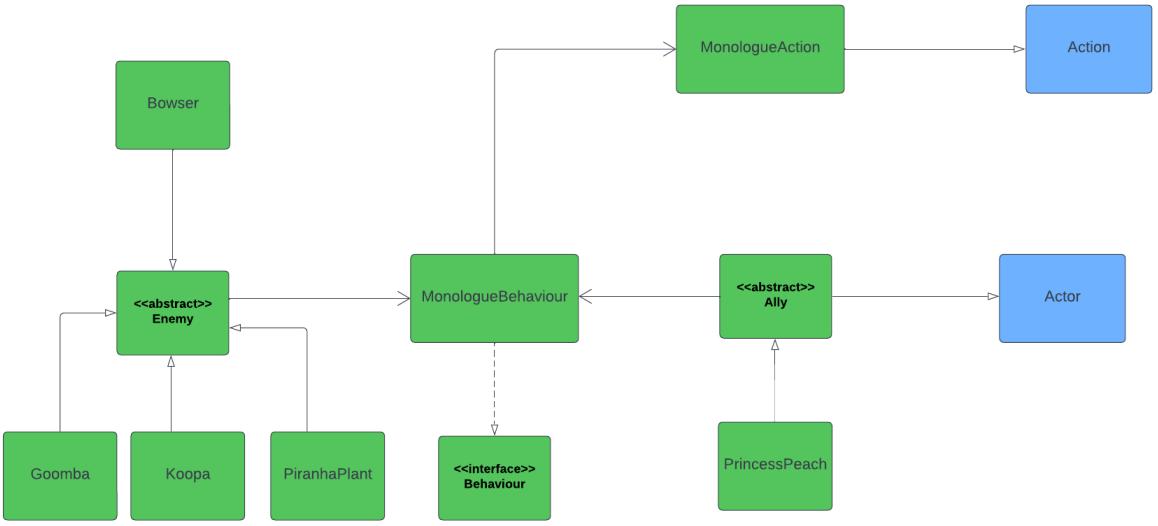
Design Rationale:

The fire flower requirement was simple to implement, as we could simply drop the flower spawning logic from Req2 (Bowser) into the pre-existing tree stage growth logic. Fire attack logic was also fairly simple, as we could check if the player had the FIREPOWERED Status when the attack action was executed, as is done with the HasCapability function on several occasions throughout the project, and simply spawn a fire if it was true. The Fire class is a child of item; while it could have been made a ground this would have caused a few issues, such as needing to replace the tile with the original ground once the fire is extinguished, or enabling actors to pass over otherwise impassable terrain if it was replaced by a fire. These problems are all fixed by simply making fire a (non portable) item. The Tick function of the fire will check if there's an actor standing on top of it and apply some damage along with a message about burning if there's someone on top of it. The only potential negative with this method is that multiple fire instances can be active on the same location at once, but perhaps this just represents the fire being larger and more damaging than the others after Mario has concentrated multiple fire attacks there (specifications don't state that there can only ever be one fire on a single tile, so I left it in). Additionally, fire doesn't last very long (only 3 turns) so it won't stack up to an absurd

degree and start instantly killing things. Also, we've been referring to the FireFlower as a child of Item for convenience sake, but really it's a child of ConsumableItem, which is just an abstract class that extends Item and implements the Consumable interface, which contains a Consume method for deleting an item and applying any effects it may have. This is how Mario obtains that FIREPOWERED status in the first place.

Req 5: Speaking

Class Diagram:



Design Rationale:

MonologueBehaviour	Used to retrieve the MonologueAction for NPCs, both Enemy and Ally. This action is retrieved every alternating turn.

Similar to all the other requirements in this assignment, this requirement heavily made use of many of the pre-existing classes to ensure good design is maintained. In this case, the behaviour interface was used since we want AI to speak automatically and so like all of the other behaviours, the MonologueBehaviour is used to retrieve the MonologueAction at the appropriate time and execute it. Since we want the MonologueAction to occur every second turn, we need to separately execute the MonologueBehaviour.getAction() every turn so that the AI in question will say their dialogue and still perform another action, as opposed to the monologue being their only Action. We decided to put the actual dialogue selection process within the MonologueAction as opposed to making several children of

MonologueAction for each and every creature that will have dialogue. To do this would be wildly inefficient and would violate the Open-Closed principle as it would result in modifying the parent class of MonologueAction to reflect each and every possible option. It is much easier and more efficient to use if-else statements within the action, which takes the actor as a parameter, to determine what it should say. This approach allowed for MonologueBehaviour and MonologueAction to interact cleanly.

NOTE: All classes in diagram should be blue except for MonologueBehaviour