DESIGN

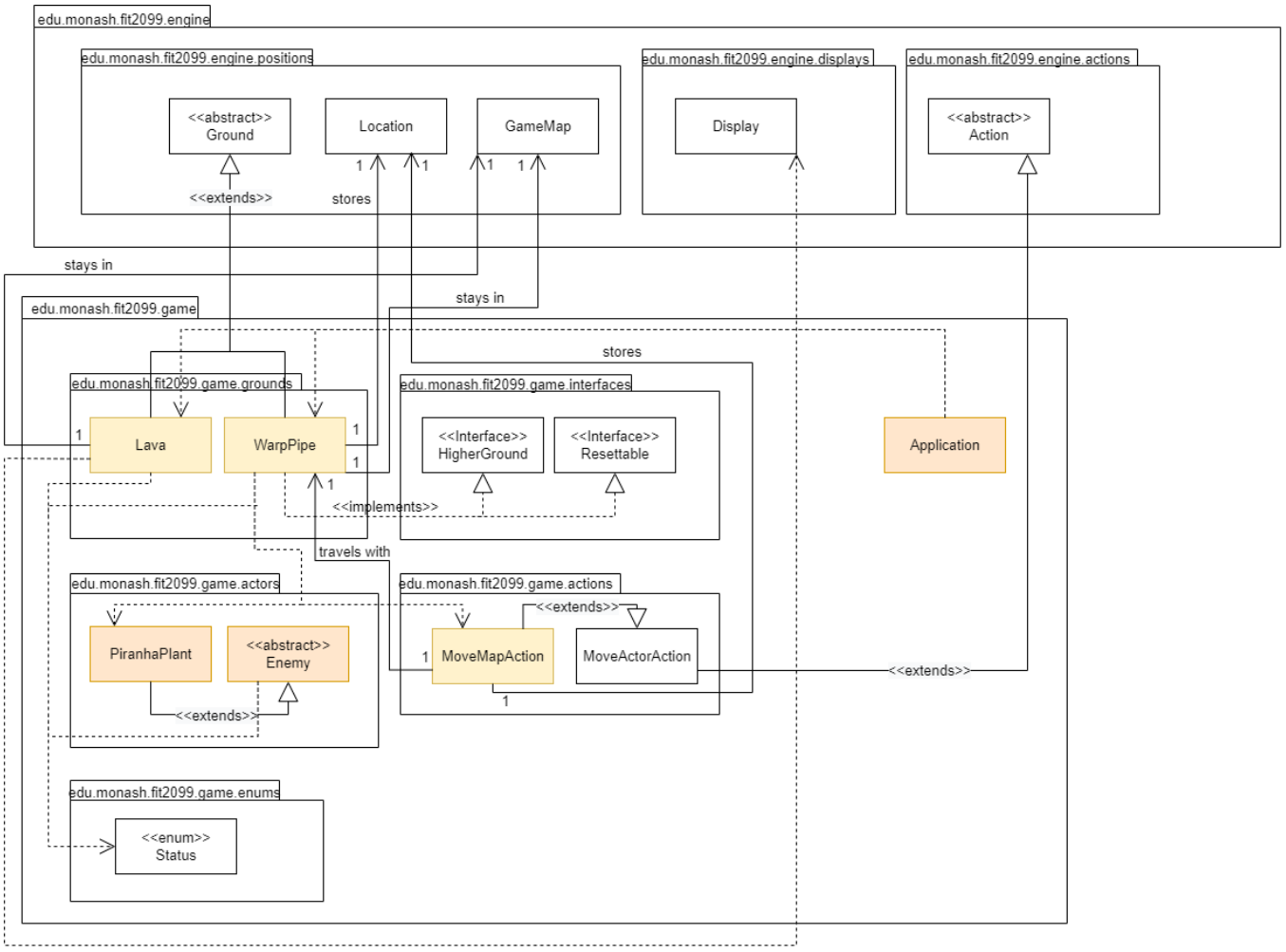
Note:

In the submission instructions in EdLesson, it also stated that we should have a separate text that outlines the overall responsibilities of the new classes (as shown in screenshot below). Therefore, we have added the overall responsibilities of the new classes or new/modified methods after the class and sequence diagrams.

Also, our generated index.html for javadoc is within the *docs* folder.

REQ1:

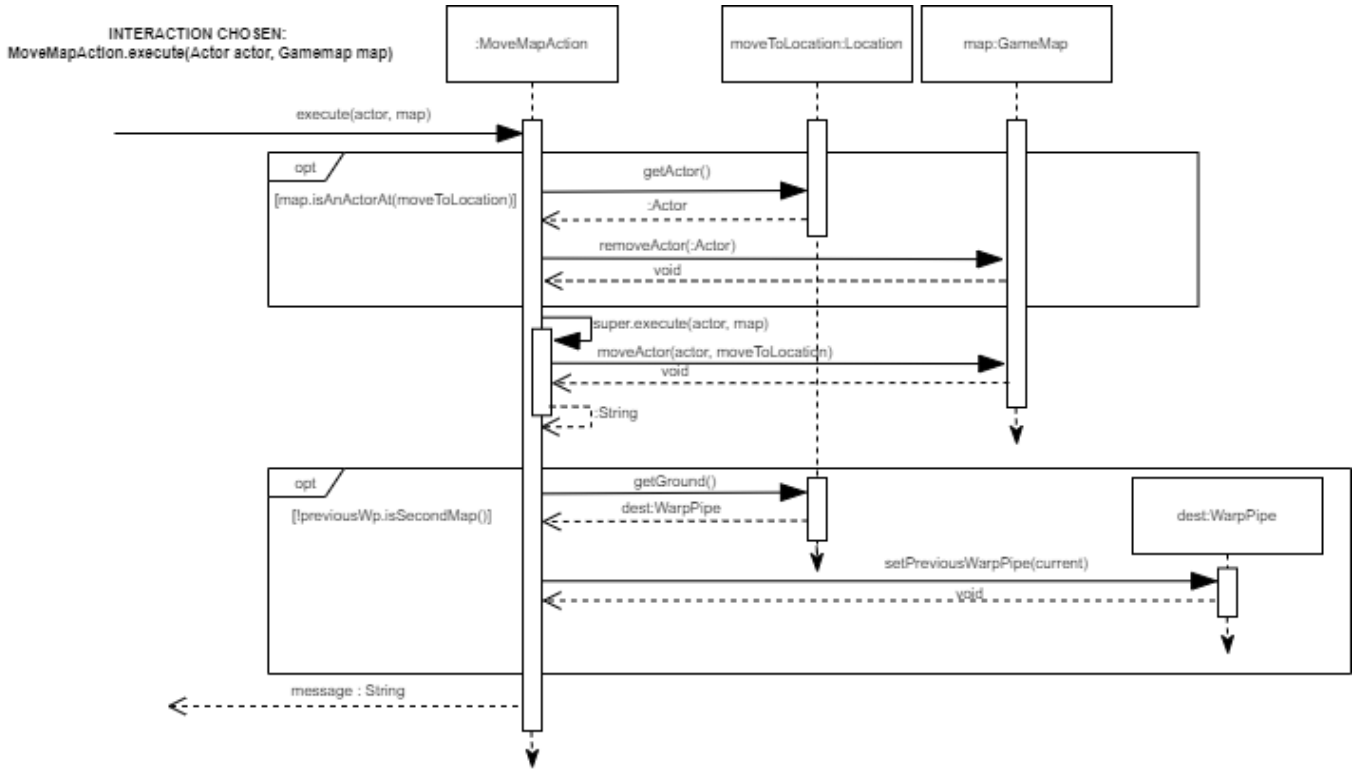
Class Diagram



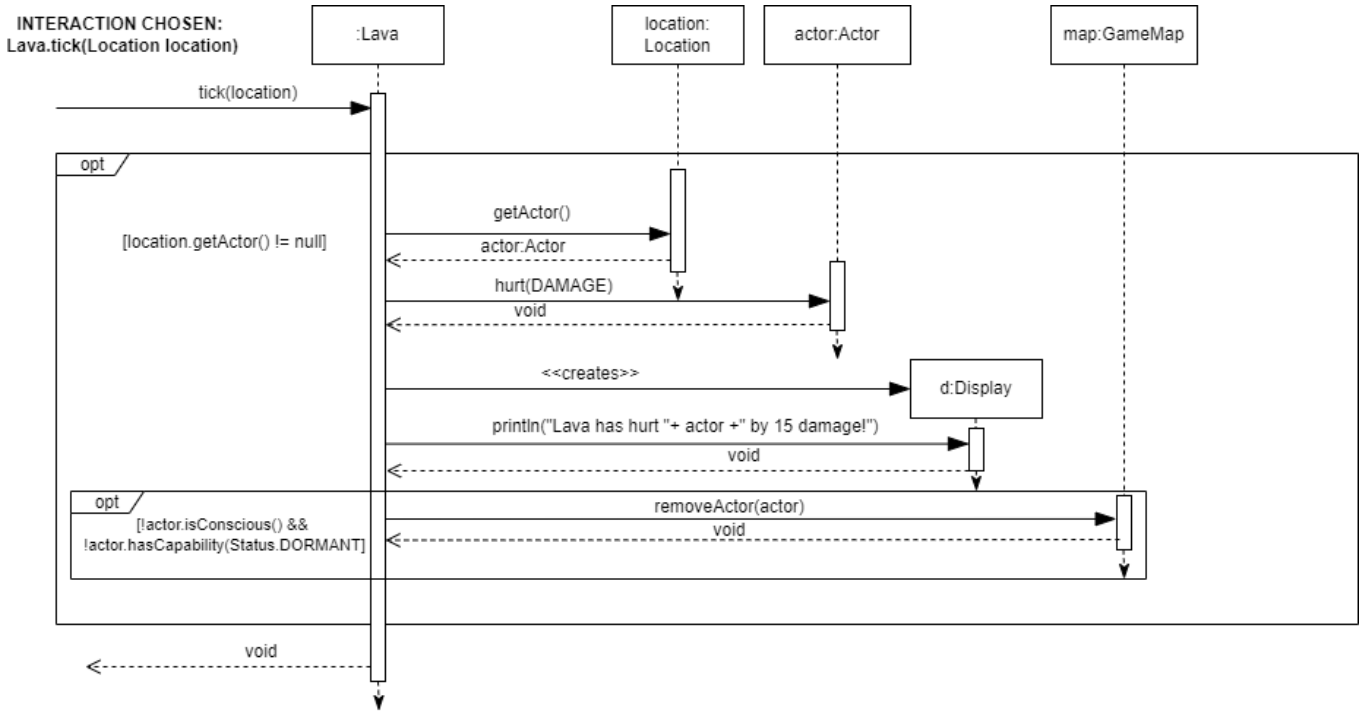
Yellow classes = classes added directly for this requirement
Orange classes = class added/modified related to the Yellow classes, that were either of: i. Already in base code, ii. Added from asgn2, iii. Added from another requirement in asgn3.

Sequence Diagram

The interaction of MoveMapAction.execute() is chosen so that it can illustrate how the actor/player is being moved from one map to another - which is the main functionality of REQ1.



The interaction of Lava.tick() is chosen so that it can illustrate how the Lava can inflict damage on actor standing on it in each turn.



Design Rationale

1: WarpPipe and MoveMapAction

To be able to move between maps, the player needs to have an action option to “teleport” (move). Thus, we have MoveMapAction that extends the abstract Action class.
Since the player can only teleport through a WarpPipe, in our design the WarpPipe class will be the one in charge of providing the MoveMapAction to the player.

Alternatively, we can have the player in its play turn method to check if it is standing on top of a warp pipe, and if it is standing on a warp pipe, to add a new MoveActorAction to the actions ActionList. While this approach seems straightforward enough, however, this alternative would cause Player to have an extra dependency to WarpPipe, which leads to a violation of the Reduce Dependency Principle. Also, doing this would add a new responsibility for Player to handle, which is the responsibility of providing the MoveMapAction, thus this violates the Single Responsibility Principle as well.

Thus, with the current design, we will discard the above alternative since it violates the Reduce Dependency Principle and Single Responsibility Principle.
Also, with our current design, MoveMapAction extends engine class’s MoveActorAction class, and therefore, this enables MoveMapAction to work like MoveActorAction, and thus any changes made to the basic moving of an actor between locations, will be applied within MoveMapAction. Therefore, this adheres to the Don’t Repeat Yourself principle, because if MoveMapAction did not extend the MoveActorAction class and if the functionality of basic moving of an actor is changed, we would have to change the code and implementation in both MoveMapAction and MoveActorAction class, which means that there was indeed an repetition in code/functionality that could have been avoided, which thus violates the Don’t Repeat Yourself principle.

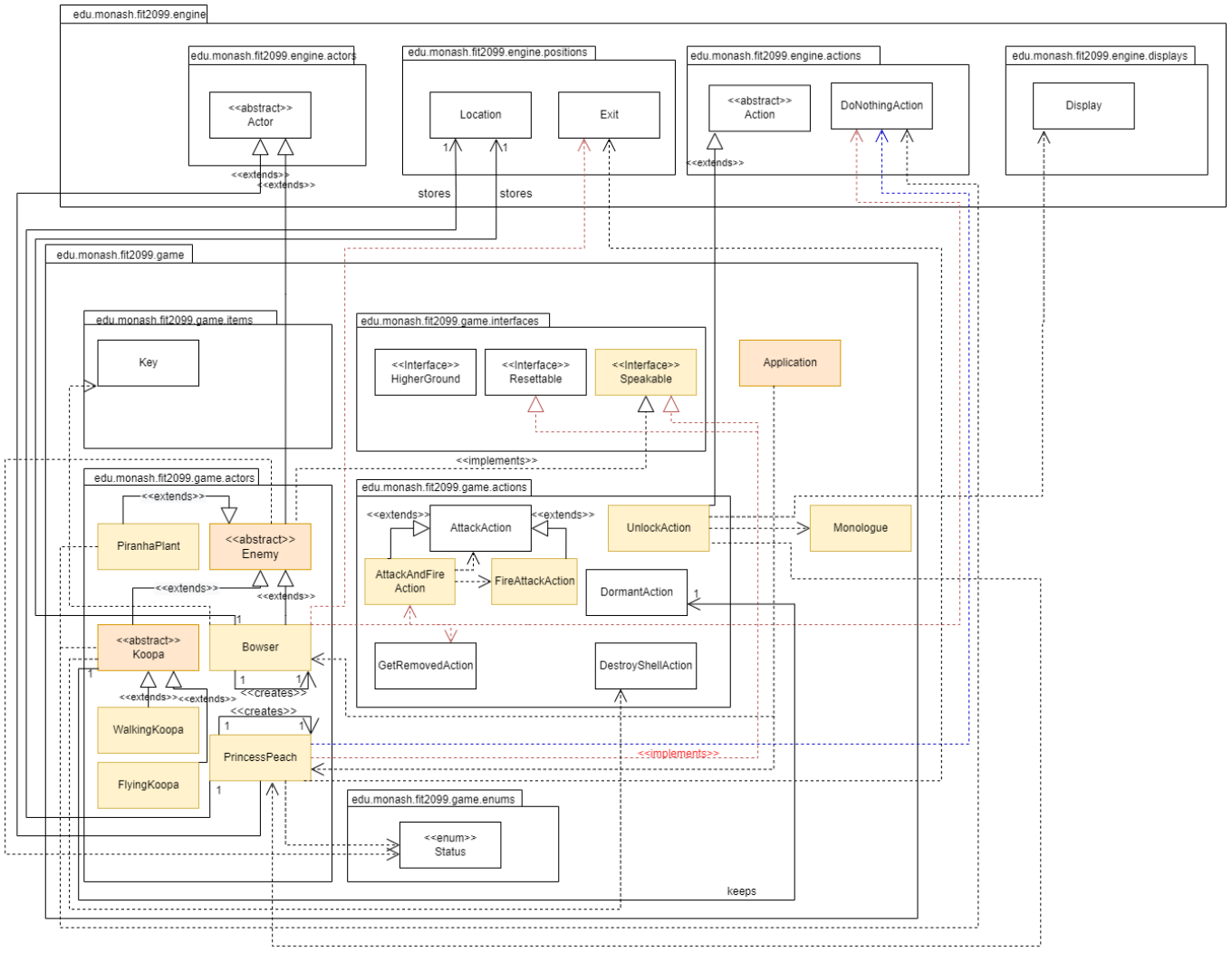
Therefore, based on the above rationale, our chosen design (Warp Pipe providing the action to player, and MoveMapAction extends MoveActorAction), adheres to the Reduce Dependency Principle, Single Responsibility Principle and Don’t Repeat Yourself principle.

2: Jumping onto WarpPipe

Note that in our implementation, we have made WarpPipe such that actors would need to jump onto it to be able to stand on it. Our design for this, is then as such: WarpPipe will implement the existing HigherGround interface, which allows actors that don’t have the *TALL* capability to be required to have a JumpAction in order to be able to move to the location of the WarpPipe and stand there.
An alternative to not implementing the HigherGround interface, is for the WarpPipe to add checks for if the actor has the TALL capability or INVINCIBLE capability, and for different capabilities/different combinations of capabilities the mode of moving onto the WarpPipe would be different and thus the WarpPipe would have to instantiate different movement actions - this would then increase the number of dependencies that WarpPipe due to these checking and instantiation of different actions in the WarpPipe class itself. This thus leads to a violation of the Reduce Dependency Principle. Also, these checkings as mentioned above, are already done in the HigherGround interface’s default method, and therefore, if WarpPipe did all the conditional checking in its own class, this would violate the Don’t Repeat Yourself principle.
Therefore, we discard the above alternative and go with our design of WarpPipe implementing the existing HigherGround interface. Thus, based on the above rationale, this chosen design adheres to the Reduce Dependency Principle and Don’t Repeat Yourself principle.

REQ2:

Class Diagram

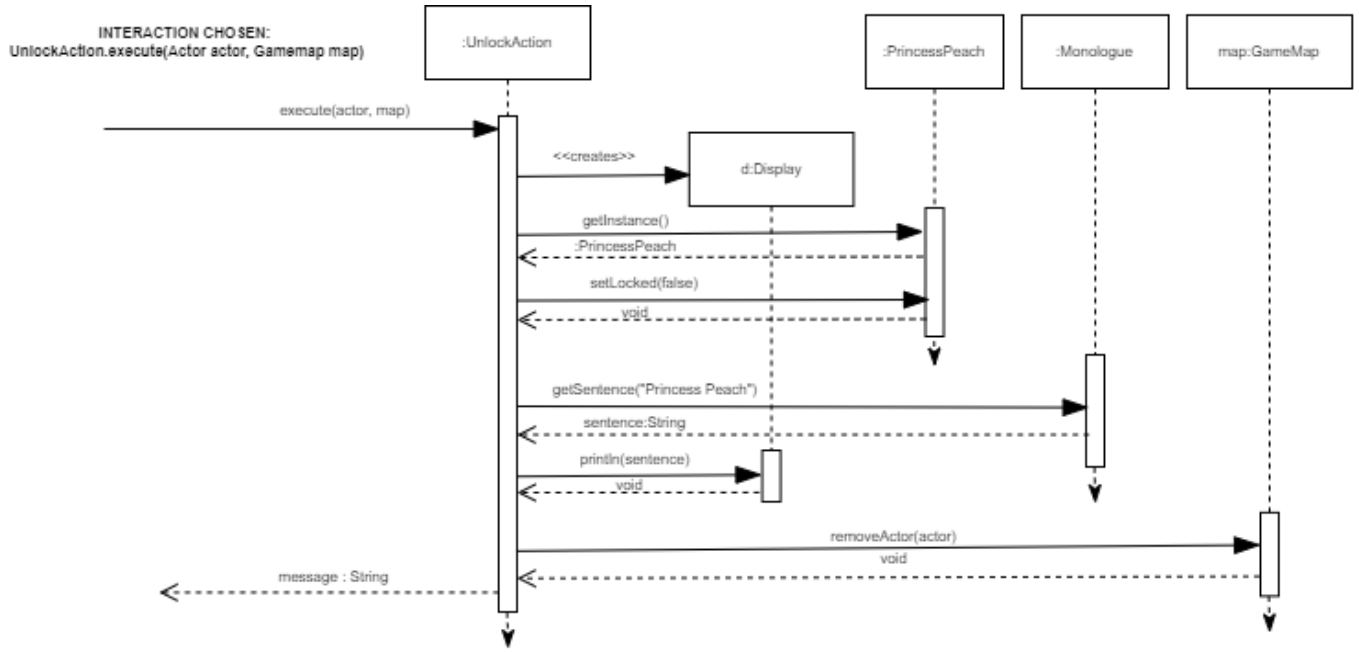


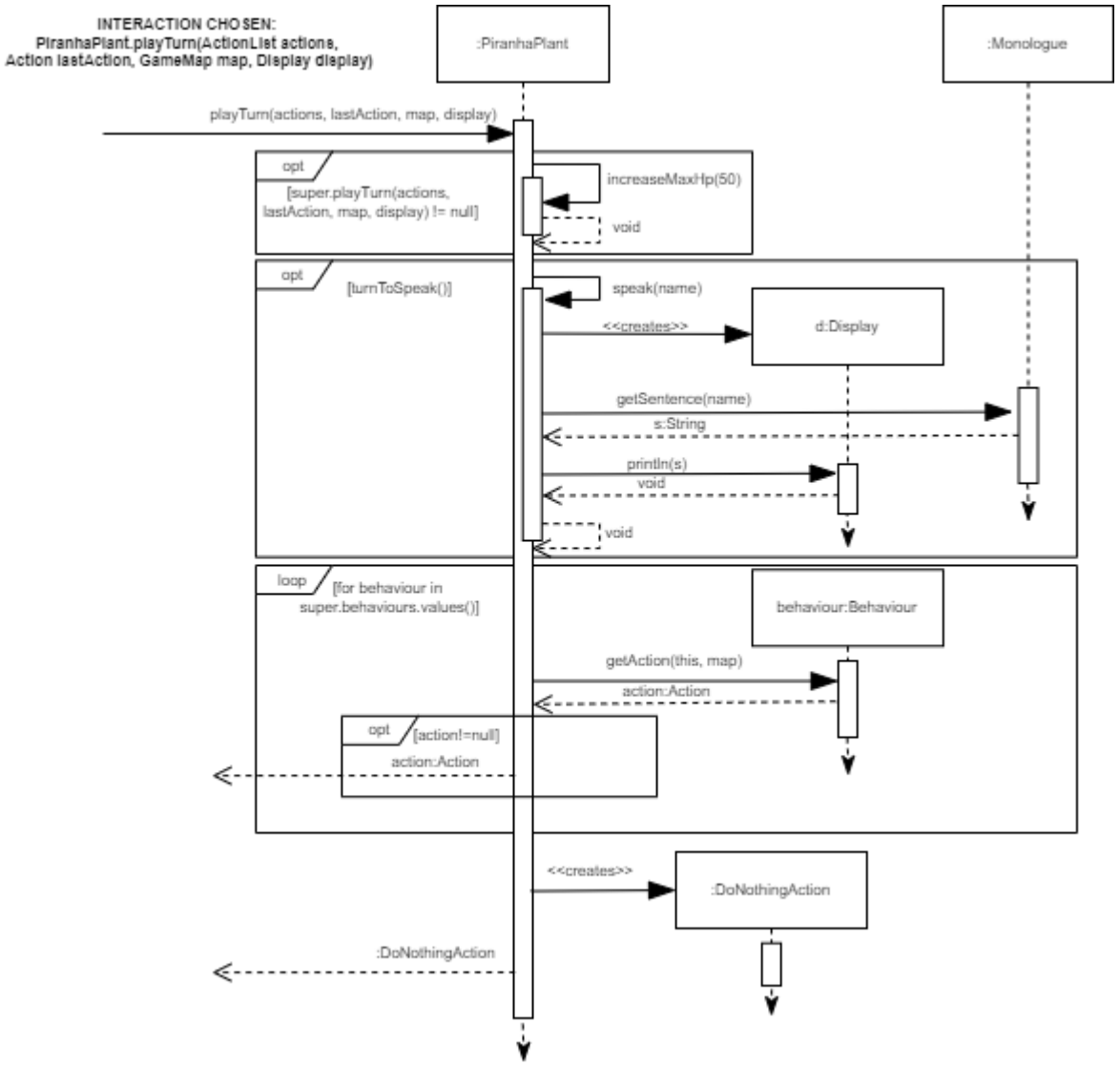
Note: In the above diagram, some lines are coloured and some not, this is for the sole purpose of making it clear where the lines lead to - there is no other significant purpose of the colours of the lines related to the design.

Sequence Diagram

Note: That the AttackAndFireAction.execute() is not shown here **because**

1. It is mostly the combination of AttackAction and FireAttackAction.
2. AttackAction's functionality is known since assignment 1, thus is not as important for analysis.
3. FireAttackAction, which is new and more important, is shown in REQ4's section.





Design Rationale

1: Key, Princess Peach and UnlockAction

To be able to check if PrincessPeach is locked by Bowser or not, we have a boolean static variable in PrincessPeach to check if she is allowed to leave. Also, to allow the player to have an action to unlock princess peach, there needs to be an instance of UnlockAction supplied to it when the player retrieves the key. Therefore, UnlockAction is only instantiated within the Key item class in its getPickUpItemAction method.

The first alternative we had initially, was to have a checking within Player class's playturn method: to see if it has a key in its inventory, then instantiate an UnlockAction class to be added to the list of actions it can do.

However, this approach would cause the Player to have an added responsibility of adding the UnlockAction, as well as a responsibility of seeing if the Key is in the inventory, which violates Single Responsibility Principle. Doing this also adds dependencies to Key and UnlockAction and therefore this approach also violates the Reduce Dependency Principle. Thus, instead of having to check if there is a key within the player class to add the action, our implementation improves on the above problems by

adding the unlockaction to the key that is picked up, and based on engine code functionality, the actions in the item's actions list are the actions that the actor can do **if** the item is in that actor's inventory. Therefore, we discard the first alternative we had, and thus our design as justified will adhere to the Single Responsibility and Reduce Dependency Principles.

2: Bowser, AttackAndFireActor
Our design of Bowser is such that Bowser extends the Enemy class so that it can have the shared functionality of the Enemy types. This is in line with Don't Repeat Yourself as we will be reducing the repetition of functionality and code implementation.

Also, one interesting thing about Bowser is it's method of attack. First it attacks the player with an attackaction, and only if it is successful will a fire attack action happen.

How we have done this is by:
Creating another Action class called AttackAndFireAction, that extends AttackAction class, where this Action combined the functionality of both the AttackAction and FireAttackAction. In its execute method, first it calls on the super.execute() function to perform the normal attackaction, and if there was a successfully attack and if the target is still conscious, a fireattack is created, and will then be executed.

How bowser uses this is by having another method called getAttackAction() in its own class, where this method is responsible for checking if the player is in its surroundings. If the player is indeed its surroundings, this getAttackAction returns an instance of an AttackAndFireAction. For bowser, this getAttackAction() method call happens in its playTurn method, and if an instance of AttackAndFireAction is returned, the playturn method returns this action for it to be executed,

Another alternative our approach is this:
In Bowser, have an attackAndFollowActor method that is designed to be specifically to be called within the playturn method of Bowser. While this is similar to the above approach, the difference is that attackAndFollowActor method within Bowser would be the one to instantiate an AttackAction, execute that action, and then if the attack was successful, then only return a separate fireAttackAction to its playturn method.
However, this alternative approach violates the Single Responsibility Principle since having the attackAndFollowActor method means executing the attack within the Bowser class. Therefore, we discard this alternative and chose to go with the design of having a new AttackAndFireAction class that extends the AttackAction class, and that combines the functionality of the inherited AttackAction, as well as the FireAttackAction class, all the adhering to the Single Responsibility Principle.

3: Piranha Plant
The functionality related to PiranhaPlant, and how they are implemented in our design, is mainly this:

Make PiranhaPlant an enemy, this is done because PrinanhaPlant will attack mario similar to all other enemies just with a different attack damage and way of attack, and therefore, we can use the shared functionality of the abstract Enemy class for this. Doing this thus leads to our design adhering to the Don't Repeat Yourself Principle.

4: Flying Koopa and Walking Koopa

implementing flying koopa

Our initial code implemented Koopa as an extension of an Enemy base abstract class. Originally we planned to simply extend that Koopa class and to use all its behaviours and methods accordingly since they would essentially remain the same except for health and the ability to fly. As we began coding we found that it may not have been the best design choice
since it would likely have violated two important SOLID principles. Those being LSP (Liskov's Substitution Principle) as well as DIP (Dependency Inversion Principle).

Although at this point in time Koopa and FlyingKoopa both have mostly the same uses for all methods, it would be extremely unreasonable for us to assume that FlyingKoopa may not have new features or different uses of the current methods in future. I believe taking this into account and preparing for the future is a display of good design.

Additionally, Koopa and FlyingKoopa make use of different capabilities to traverse obstacles on the map, Koopas will use the MUST_JUMP status, whereas FlyingKoopa's use the FLYING status. This difference shows us that the FlyingKoopa object does not behave similarly to the object (Koopa) it is extending, which is in direct violation of LSP. For example, if another class depended on checking whether the Koopa object had FLYING or MUST_JUMP, different results would be given between the parent class and subclass.

Besides this, extending Koopa would have also violated the Dependency Inversion Principle as we would have been extending a concrete class; the Dependency Inversion Principle states that "A concrete class should not depend on another concrete class. Instead, it should depend on abstractions."

Our initial design choice would have placed our code in direct violation with that statement.

Because this design violated two important SOLID principles, we decided to look into other ways to code FlyingKoopa with good design in mind.

A major problem we ran into when not using the aforementioned design was trying to uphold the DRY (Do not Repeat Yourself) principle. This is because the main functionality of Dormancy in Koopas comes from code in the playTurn() method and their own capabilities.

The next option we had was trying to implement an interface or a system that allowed us to keep all basic functionality of dormancy in a separate interface.
This proved to be very difficult because even with normal and default methods in the interface there was no way to give them the required capabilities without having to repeat these methods in the constructors of Koopa and FlyingKoopa. We were also faced with the difficult task of having to not repeat similar code in the playTurn of each class to carry out any Dormancy-related actions.

Because this design was seemingly too difficult and troublesome we opted to look into other ways once again.

FINAL CHOICE
Our final and chosen option was to refactor the old Koopa class into an <<abstract>>Koopa (base) class that would keep the basic general functionality of all current and future Koopa types.
We found that this change allowed us to have better future extensibility with our code since it would make it much easier to perform maintenance or re-use. For example, there are many more types of Koopas in Mario games such as MechaKoopas or MagiKoopas that can still be added, the addition of this base class will allow us to add them easily as well as follow the OCP principle strictly when doing so since we can just add new Koopa types without making any changes to old code. All methods in the Koopa<<abstract>> class can also be overridden if different functionality is required so that nothing in the base class changes.

Besides that, this option also allows us to follow SRP since each class will be solely responsible for specific functionalities, for example, the Koopa class only ensures that basic functionalities and behaviours for all Koopas are present such us attacking, following, wandering and dormancy whilst each extension of Koopa<<abstract>> will be responsible for initialising the correct health, damage and display character as well as any other newer required functionality.

So for FlyingKoopa, we extended the refactored Koopa<<abstract>> class and gave it its required name, display character, health points and capabilities in the constructor. For the basic Koopa we created a new WalkingKoopa class that once again used the constructor to initialise required naming, display character, health points and capabilities.

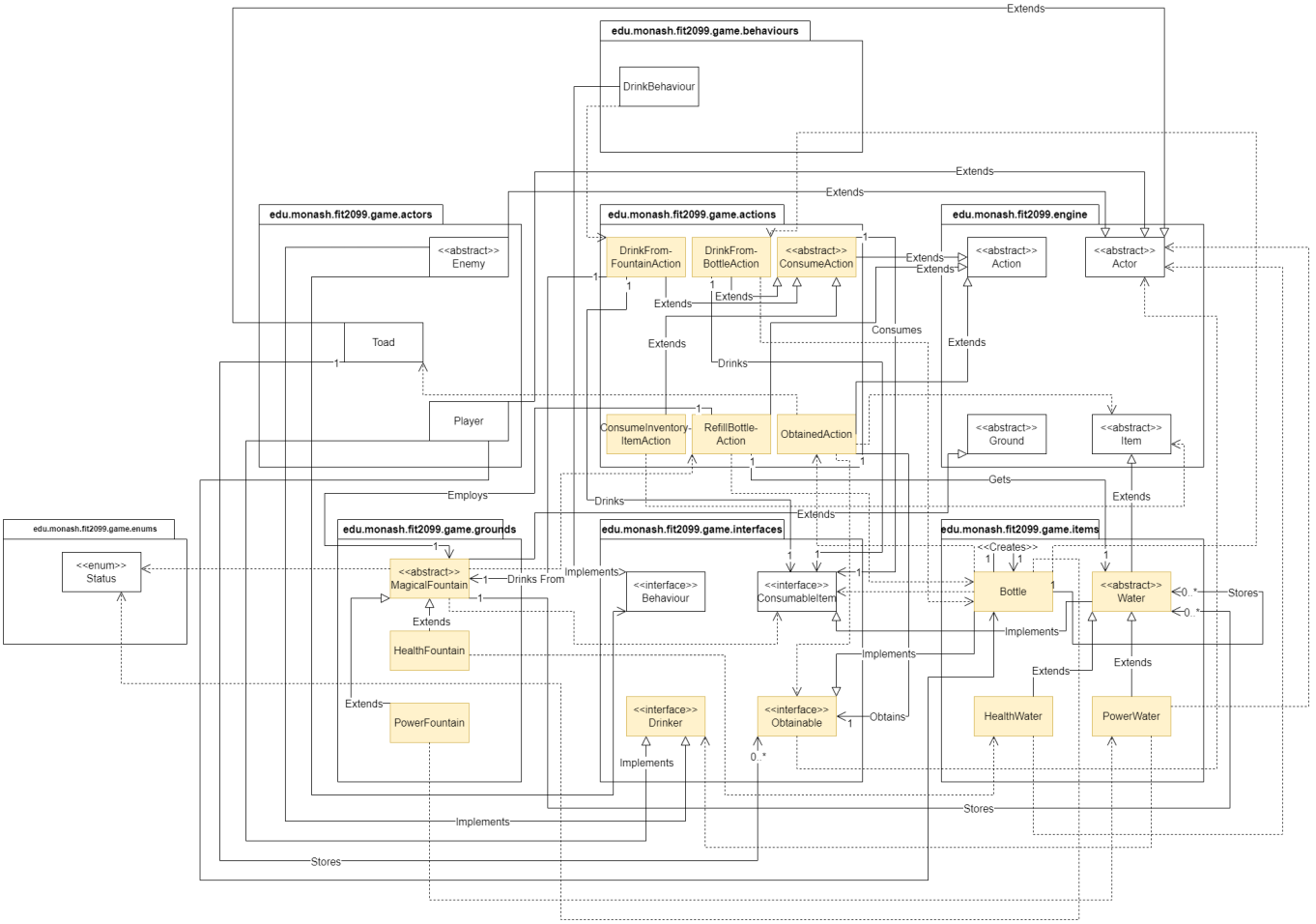
Since this implementation allowed us to follow OCP and SRP whilst avoiding the violation of DIP and LSP we deemed it to be the best option.

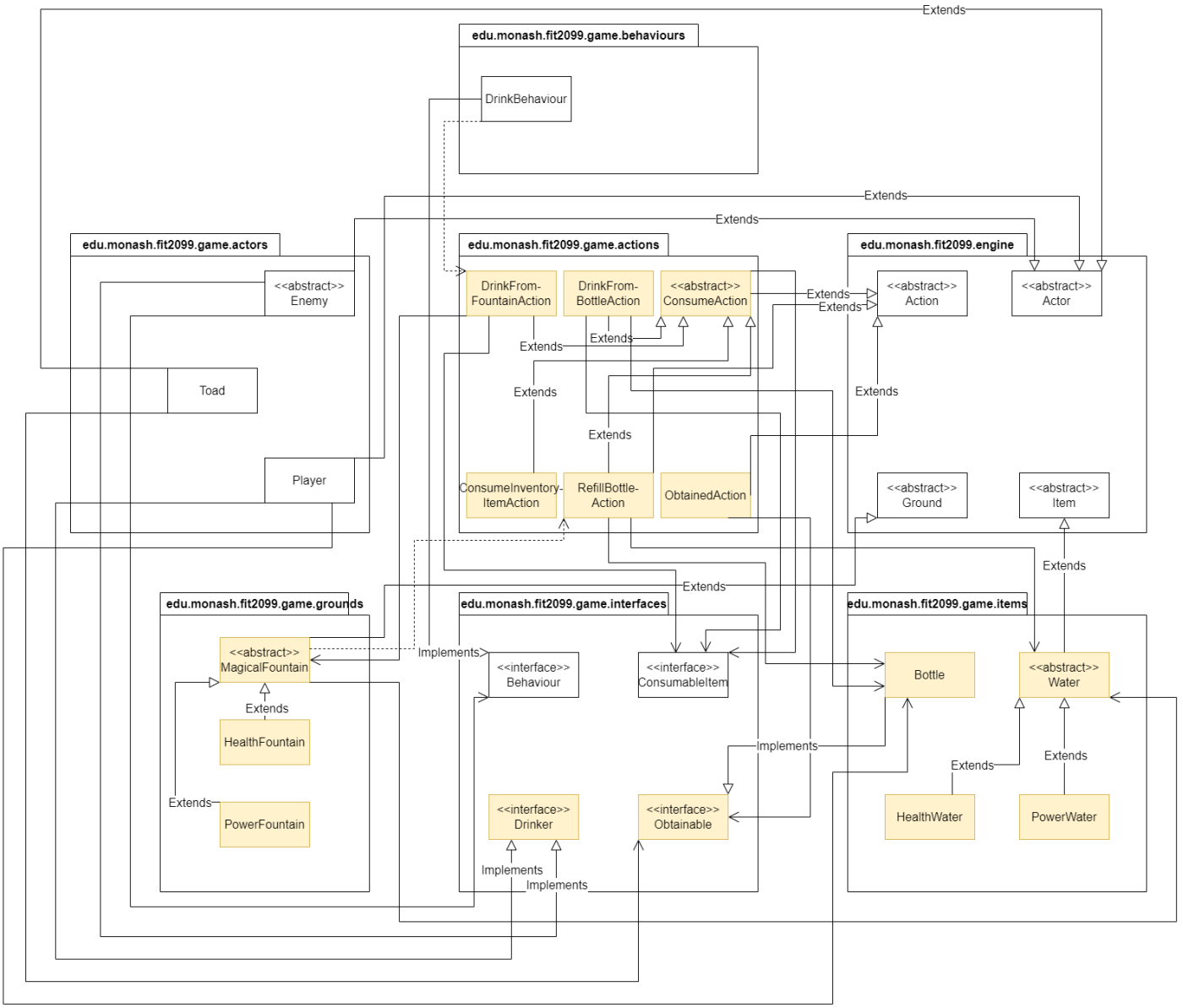
implementing flying itself
The implementation of flying in our code was very simple and straightforward. We just had to add the FLYING status to our FlyingKoopa class and add minor code to our HigherGround interface to check if Actors had the status "FLYING". We felt this was a good choice as there was very little code to change and we were making use of our interface for its original intended purpose.

The purpose of HigherGround was to make it so we did not have to update allowableActions() in each type of HigherGround whenever adding new capabilities as well as adhering to OCP when adding new HigherGround types. Another benefit of this is that any bugs in the code can be easily traced back to the interface and maintenance can be easily done. Because of these reasons, we decided to go with this option.

REQ3:

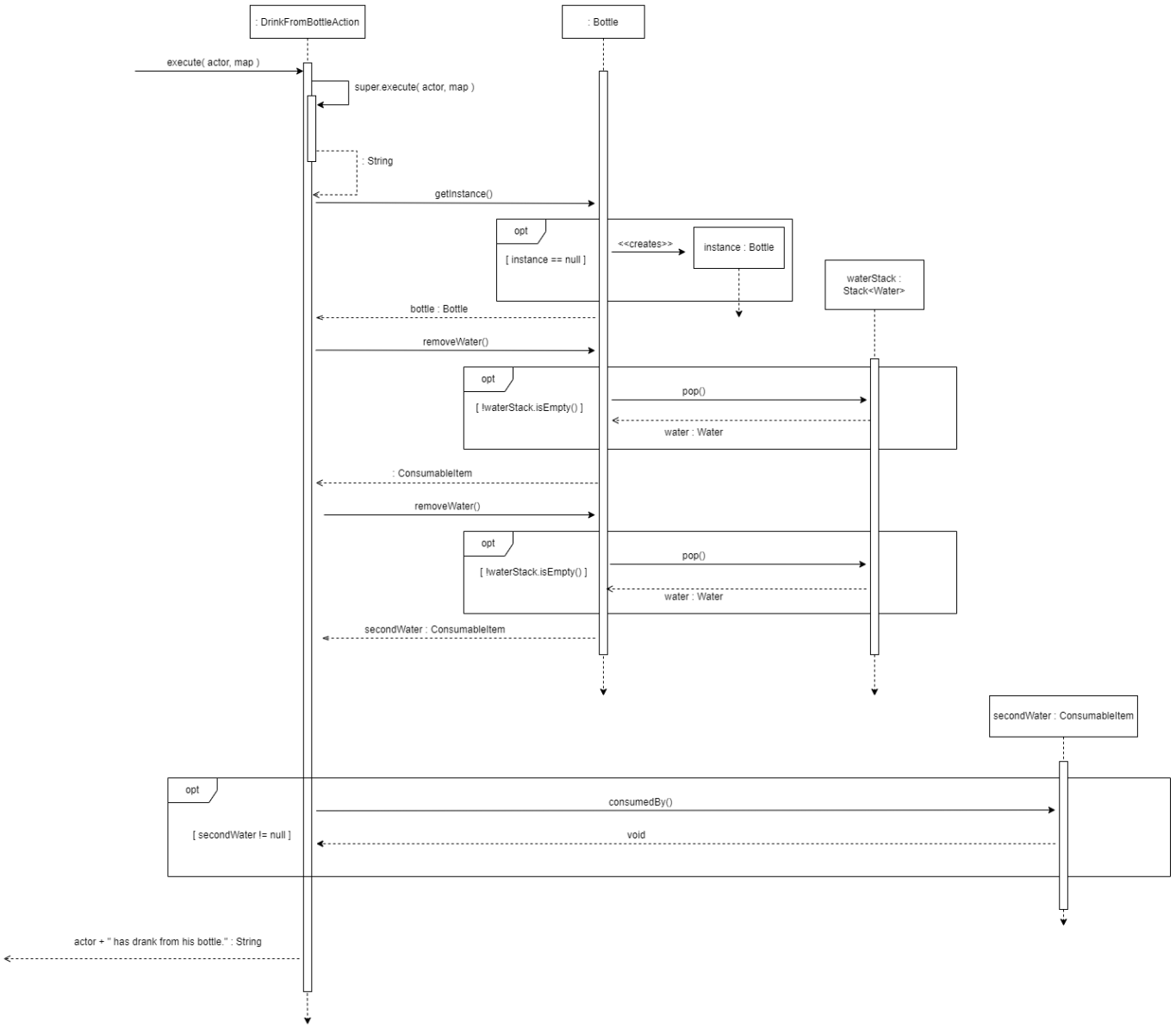
Class Diagram





Sequence Diagram

Interaction chosen :
DrinKFromBottleAction.execute(Actor actor, GameMap map)



Design Rationale

Note: In our design and implementation, we have made it such that for drinking/consuming water either from Bottle or Fountain, the maximum number of water slots per "drink" action or "sip" is *2 water slots.*

REQ 3 Magical Fountain

implementing the fountains

For the implementation of fountains we opted to create an extension of Ground, <<abstract>>base class MagicalFountain which provides most of the basic functionality required of all fountains. This includes an ArrayList as an attribute for storage of water types and methods both abstract and non-abstract to manipulate and make use of said ArrayList.

The two concrete classes HealthFountain and PowerFountain then extend this class and override any methods needed for specific functionality, namely, the refillFountain() method to add its specific water back into the ArrayList.

The justification for this design choice is very much similar to that stated in the paragraph under "FINAL CHOICE", see : REQ2 *implementing flying koopa*.

We are able to maintain good extensibility through the use of the <<abstract>>MagicalFountain class. For example adding a new RandomFountain that gives random waters that refills based on our Utils number generator or a MaxHealthFountain or JumpingFountain. Besides this changes to the general functionality of fountains can be easily made through the base class.

We also adhere to the OCP with this base class when adding new fountains since no changes will be made to the base class. This option also follows SRP as each class will be responsible for specific functionalities, the base class ensuring all basic behaviours are present and all subclasses just adding any other specific functionality required.

The use of a base class also avoids violating the Dependency Inversion Principle since we do not extend any concrete classes in the implementation, however we are in violation of the LSP principle since each subclass will likely not behave similarly to the parent class. We felt this violation justifiable since we avoid having to repeat our code by doing so, thus adhering to the DRY principle.

Since this design choice adheres to 4 of the design principles (DRY, SRP, OCP, DID), we deemed it the best option available.

implementing bottle

For the implementation of Bottle we decided to create a singleton class that implements a new Obtainable interface and extends Item with a Stack attribute to store the waters acquired by the player during the game.

Similarly to previous designs we created the Obtainable interface to reduce the dependencies between the Bottle class and the ObtainedAction. Another purpose of the Obtainable interface is to adhere to OCP when adding any new Obtainable items in future. The interface also has an obtainedBy() method to ensure all obtainable items added in future have to implement that method. This method also makes the item itself responsible for giving the player the HAS_BOTTLE status. We believe this follows the SRP as it makes sense for the Item itself to be responsible for providing the player with required statuses but not

for removing or adding itself to the player's inventory, therefore, the ObtainedAction will have the responsibility of when to call this method and to remove the bottle from toad and give it to the player. Without having to remove the item from Toad and give it to the Player, ObtainedAction would be pretty much functionless as well, which is in direct violation of the SRP. Toad is given a new ArrayList to store all obtainables and various methods to manipulate the ArrayList, so that any addition of Obtainables in the future will adhere to the OCP as well.

Our Bottle was also made a singleton class to adhere to the specifications given on Ed where it states that "There is only one bottle at a time.".

implementing new consume actions
Originally we planned to use consume action in conjunction with our ConsumableItem interface to allow it to be solely responsible for the consumption of all types of consumable items. This would've been done by having methods in the ConsumableItem interface to identify the type of item or whether the item belonged in the inventory or bottle. However, we opted out of this as we felt that we were violating the Interface Segregation Principle by creating methods like isInBottle() for items in the inventory; since inventory items do not really care for that method. Besides this, by using one ConsumeAction class to consume all items we were essentially creating a god class which is a very poor design choice.

Because of this, we opted to change our original ConsumeAction class into an <<abstract>>base class and create new actions to consume items in the inventory and bottle respectively. Another class, DrinkFromFountainAction was created for enemies to use to more strictly follow SRP, we felt like having separate classes for players and enemies made sense to not give the actions too much overbearing responsibility. The benefits of having the ConsumeAction base class greatly mirror that of having the KoopaBase<<abstract>> class and the MagicalFountain<<abstract>> class. Under “FINAL CHOICE”, see : REQ2 *implementing flying koopa*. See : REQ3 *implementing the fountains*.

Examples being, adhering to OCP when adding new consuming-related actions and avoiding violating the Dependency Inversion Principle. Each new subclass overrides the execute method to remove the items from their respective locations as required(inventory, fountain for enemies or bottle), which illustrates how the OCP can be adhered to when adding new consuming-related actions.

Since this design choice is similar to previous ones in regards to the adhering of design principles, we have deemed it appropriate for our game code.

OPTIONAL CHALLENGES :

implementing limited water in fountains & refilling
For the implementation of limited water and refilling, we decided to create a final variable in the base class MagicalFountain to ensure that the amount of water added is never greater than the set number. The fountains themselves have an ArrayList to store water objects to ensure that the RefillBottleAction or DrinkFromFountain action is not passed onto any actors when there is no water. The fountains are filled by using a for loop that adds the Water objects up till the given attribute in the constructor at first and then has methods to do so later on, we felt that it made sense for the fountain to have the responsibility

to fill itself up as it was a "magical" fountain with no given sources of water so we did not feel like we were violating SRP by doing so.

The <<abstract>>MagicalFountain class also has an abstract fillFountain() method for all subclasses to override and use similar for-loops to fill the fountains with the specified water, this does not violate the DRY principle because the types of water added are different for each fountain. Another way to reduce repeating code further would be to create a RefillFountainAction that takes as input Water type to refill the fountains and use it in the base class but doing so would also create dependencies between the classes and be quite a hassle to code. Because of this we opted to use fillFountain() instead. This method is called in the tick() method in the MagicalFountain<<abstract>> class which checks if the fountain is empty and increments the counter attribute to ensure 5 turns have passed before refilling the fountain. This is done in the abstract class to adhere to the DRY principle.

implementing enemies being able to drink

For the implementation of enemies being able to drink, we created a new DrinkBehaviour class which behaves similarly to FollowBehaviour and AttackBehaviour.
As there were no issues design-wise from our previous implementation of FollowBehaviour and AttackBehaviour, we decided to follow suit for enemies drinking.
This method of implementing Drinking in enemies also validates and demonstrates the benefits of our design choices from Assignment 2 as we stated that changing Enemy Interface into an Abstract Class would allow us to "add more functionality and behaviours to all enemies, thus greatly improving the future extensibility of our code, for example, adding behaviours/actions for enemies being able to jump". We only had to add the behaviour into the HashMap in the constructor of the Enemy Abstract Class and create the Behaviour itself for all enemies to be able to drink.

The new DrinkBehaviour class implements the given Behaviour interface in the engine code and overrides its getAction() method. This implementation of the behaviour follows OCP as we do not change any of the previous code in the parent class with this addition. We also do not violate the Dependency Inversion Principle as we do not extend any concrete classes.

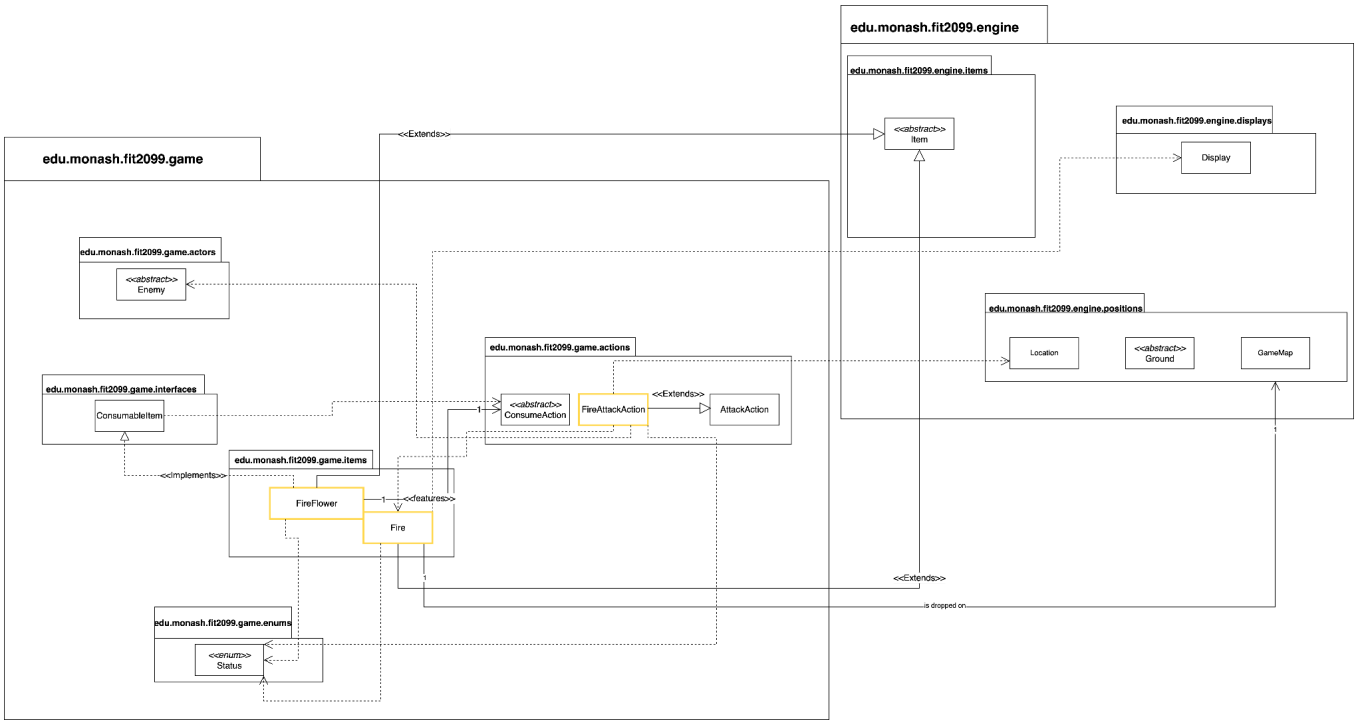
This behaviour will be given to enemies by putting it into the Enemy<<abstract>> classes HashMap attribute so that all enemies can drink, this in turn also allows us to adhere strictly to the OCP when adding new enemies as all of them will be able to drink. We believe this implementation also follows SRP strictly as well, as the new behaviour class is only responsible for giving the Enemy object action priority (ie what action to carry out first) and providing it with access to the action to carry out without being responsible for anything involved in the action itself.

Because the code adheres to all the stated design principles and we were able to re-use our code from the previous assignment with purpose and intention it was designed for, we have deemed this to be appropriate for our game code.

REQ4:

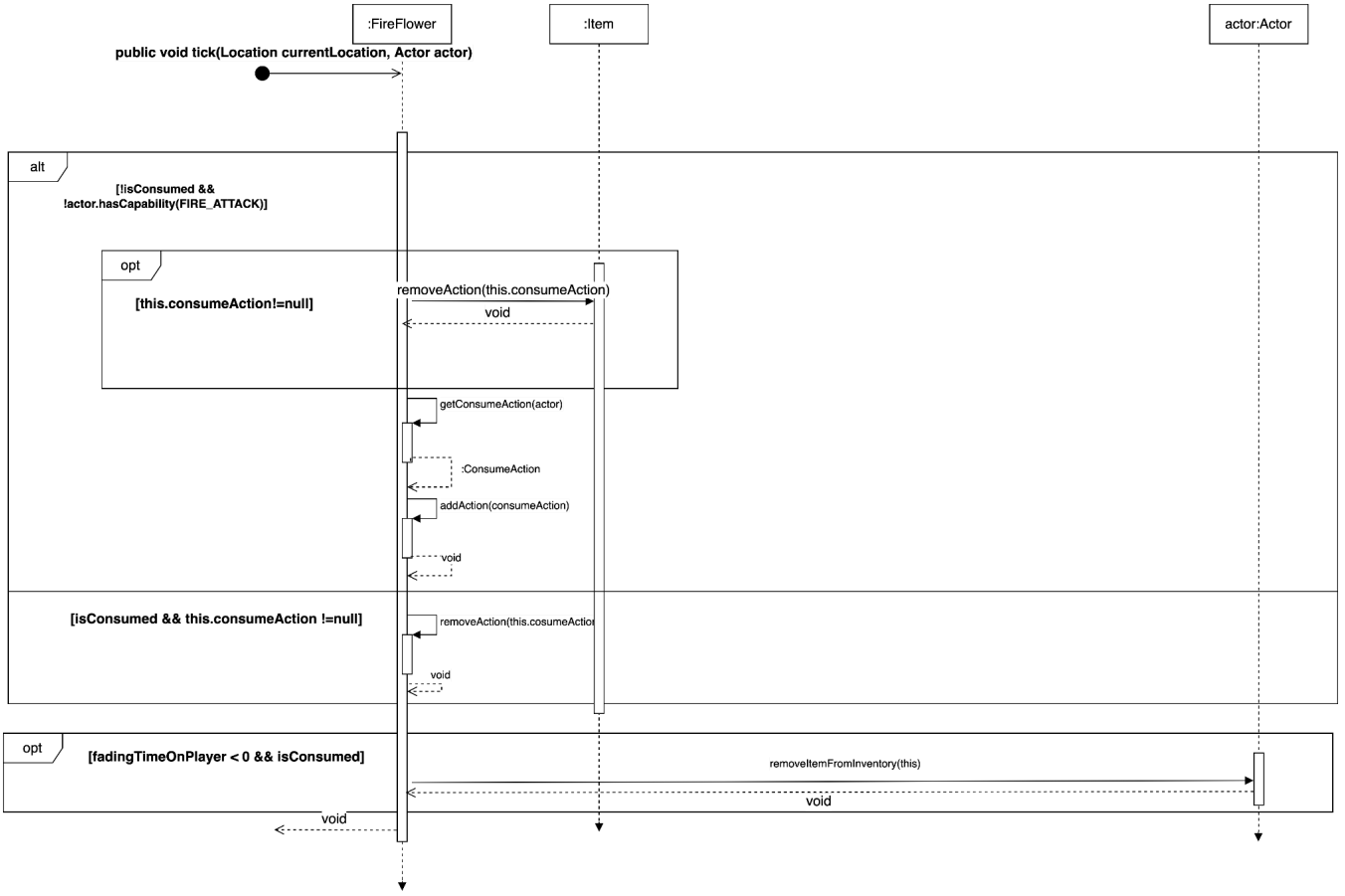
Class Diagram

Class diagram for FireFlower,Fire and FireAttackAction

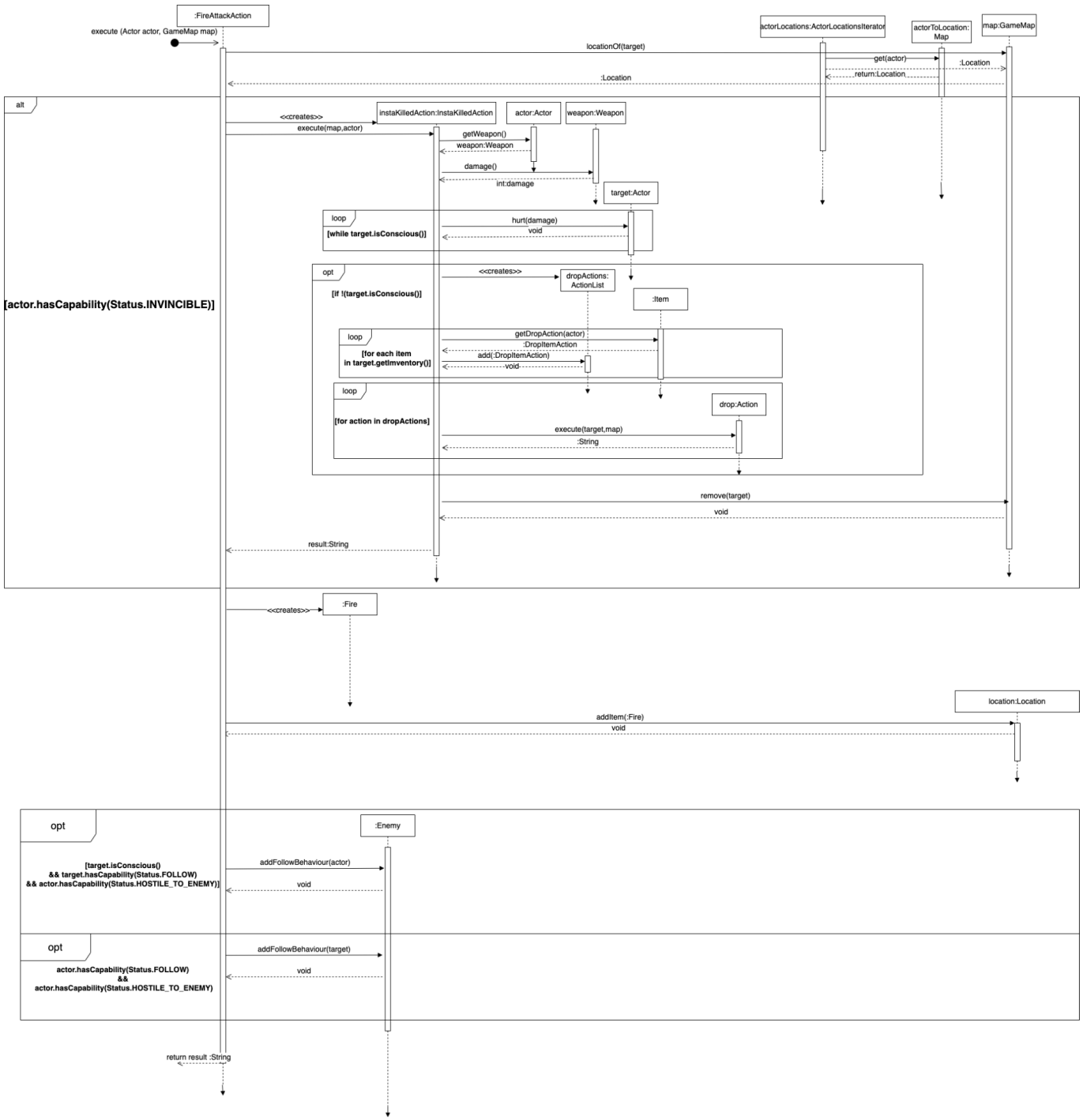


Sequence Diagram

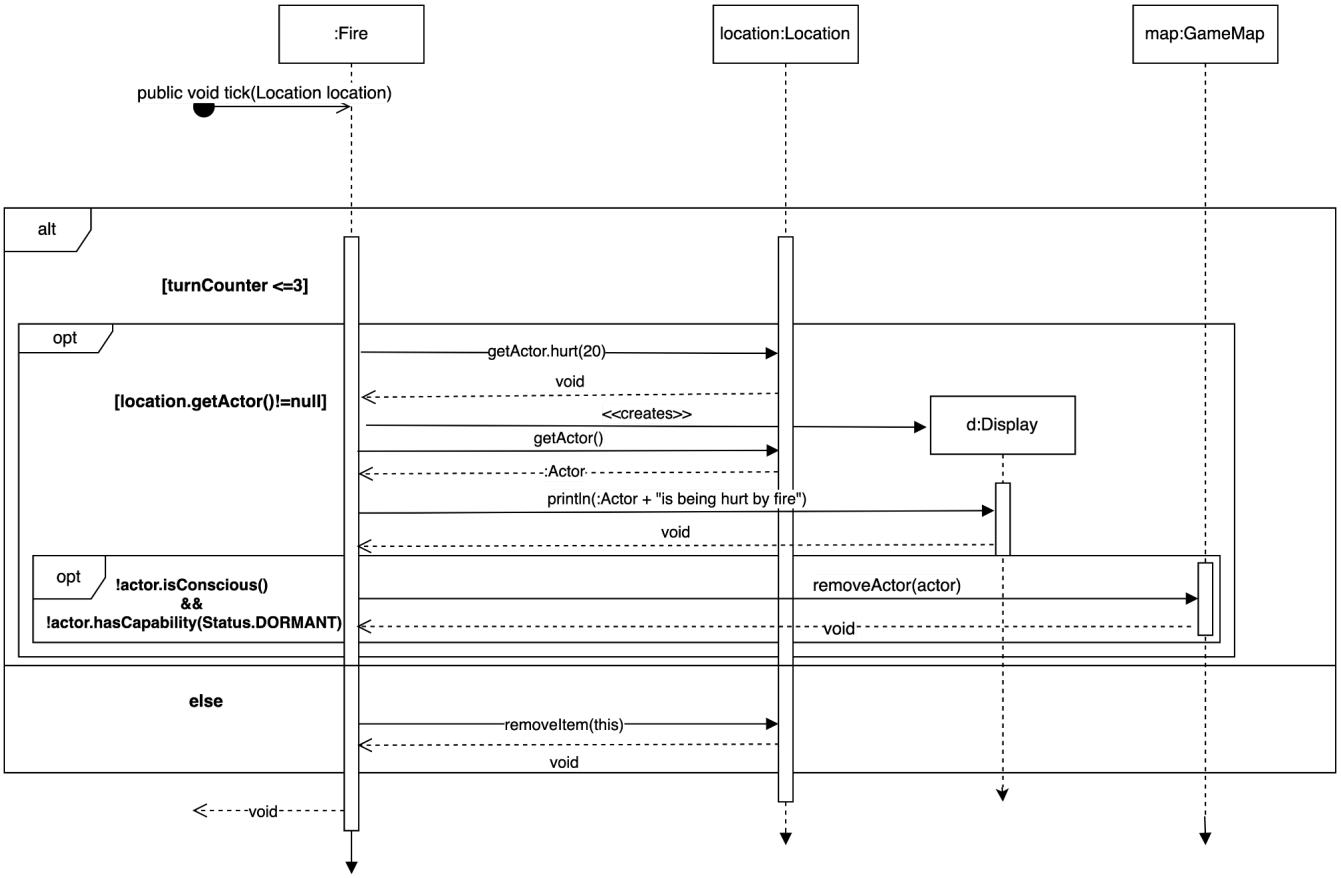
The tick method () in FireFlower class was chosen for the interaction



execute() method was chosen for the interaction in the FireAttackAction class :



The tick() method was chosen for this interaction in the Fire class:



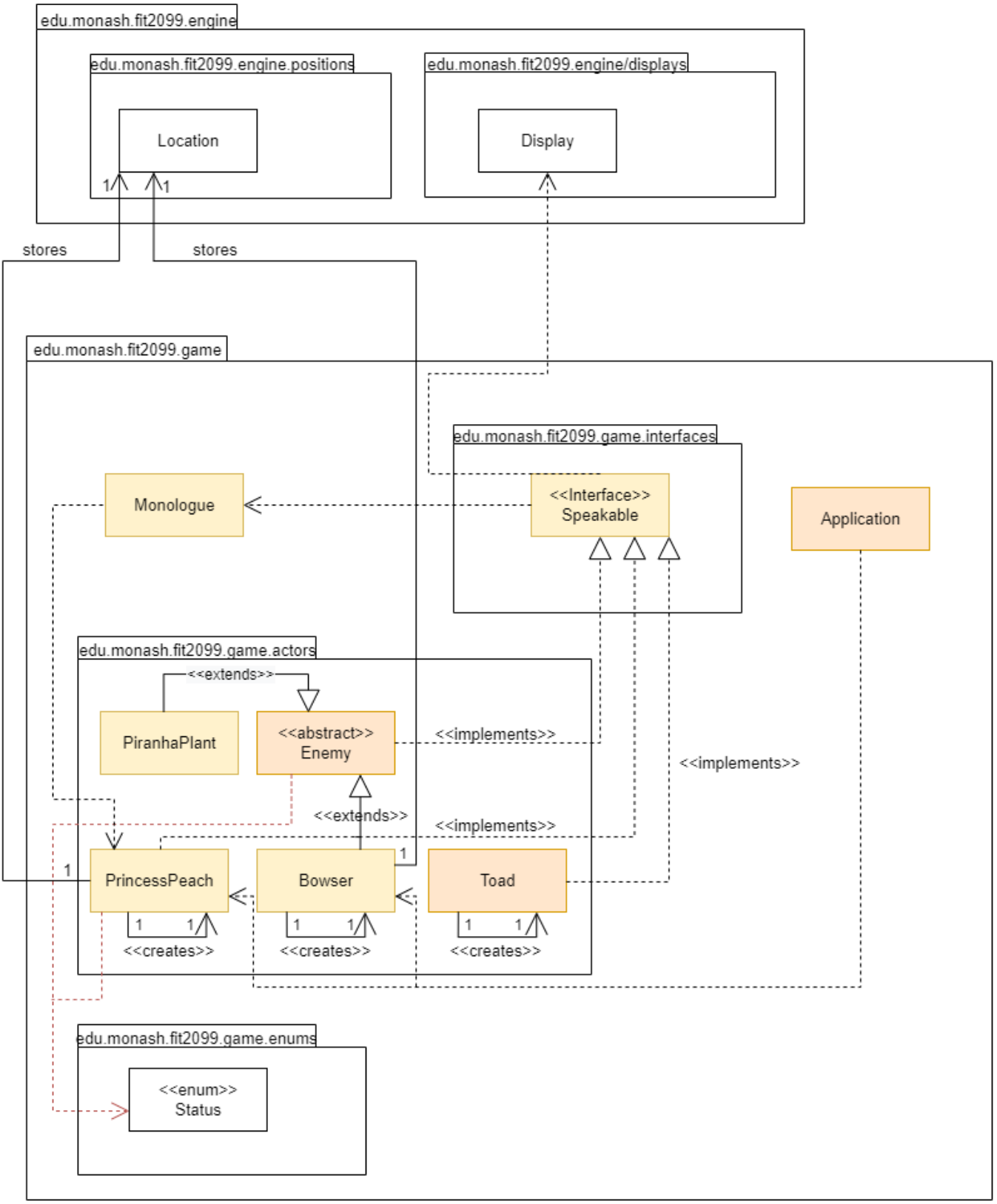
Design Rationale

For the implementation of REQ4, the functionality was segregated into three classes. Right off the bat, we made sure that our design in this case perfectly aligned with the Single Responsibility Principle. Each class is responsible for exactly what its name suggests. Firstly, the FireFlower class is created and made into a ConsumableItem (by implementing the interface) and all the methods required for Mario to pick it up and use it, is done within this class. The FireAttackAction class was created to enable Mario to use the effect that activates when Mario consumes the FireFlower.Every fire attack drops fire on the ground of the enemy's location, so to instantiate a 'Fire' object we've created a separate Fire class to accommodate the implementation. The damage dealt by the fire is handled by this class itself.

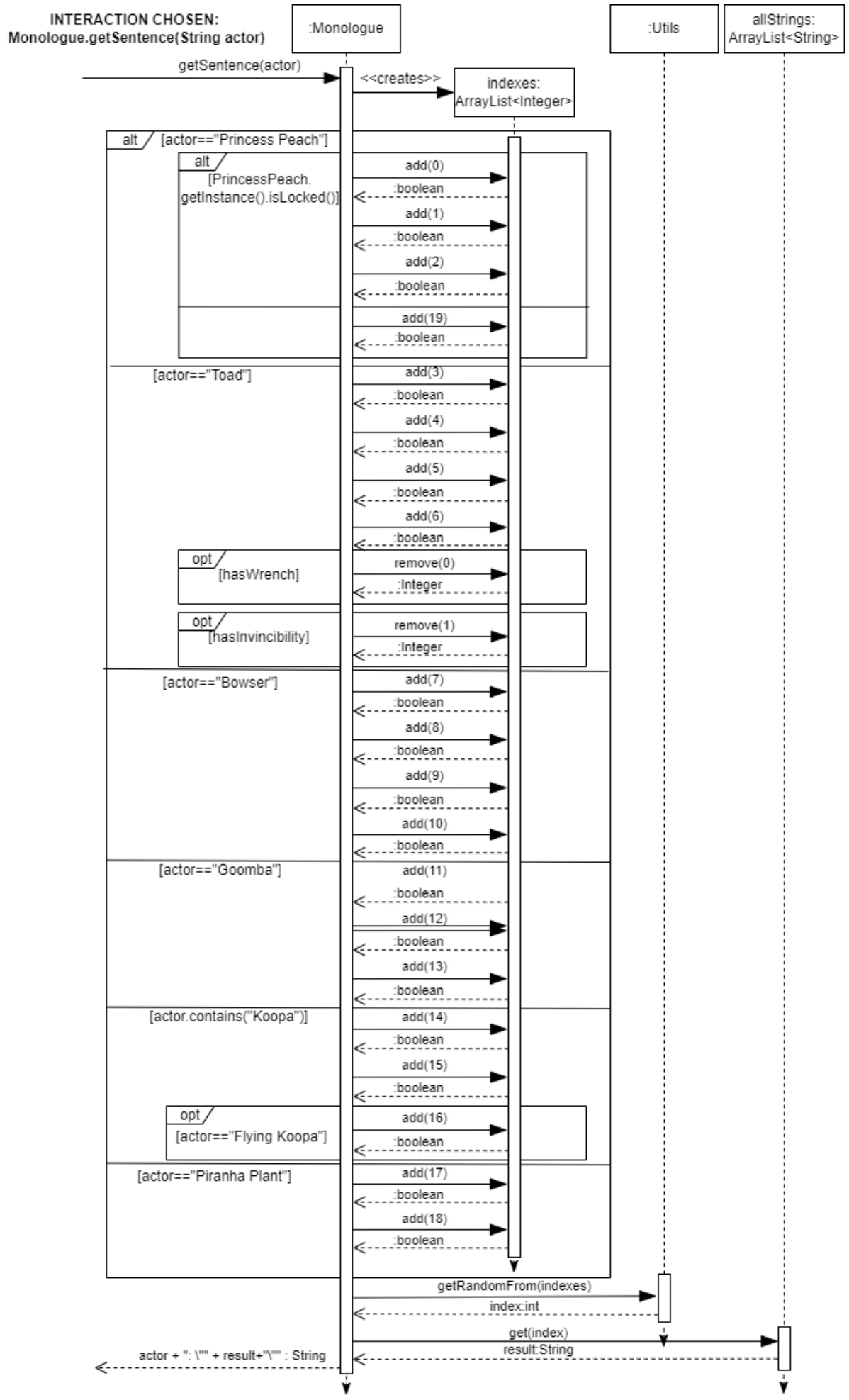
Additionally,Bowser also uses the fire attack action. A possible implementation would be to have a fire attack method for Bowser and Player separately. Doing that would violate DRY, i.e same functionality would have to be coded in different classes. Another alternative way,is having Bowser have the capability of FIRE_ATTACK and in the AttackAction class, make it check if the actor hasCapability of FIRE_ATTACK; if it has this capability then the functionality would take place accordingly. But this violates SRP by creating a God class as AttackAction already has to check for many functionalities for a single attack, by the player already. Thus, we'd be adding more fucntionality it shouldn't be responsible for.

REQ5:

Class Diagram



Sequence Diagram



Design Rationale

1: Monologue and Speakable

How we represent speaking is this:

Have a Monologue concrete class that stores in an arraylist the list of all available string sentences that can be spoken by the speakable actors. This monologue class will have a getSentence() function that returns the specific sentence that is spoken by the specific actor in a turn when it is called, based on checkings within the getSentence() method itself to determine which sentence can be allowed to be said.

Then, all actors who can speak, implement a new Speakable interface. In this interface there are two methods: 1. turnToSpeak() that must be implemented by the Speakable actors, 2. speak(), a default interface method that is responsible for getting the sentence from Monologue class and displaying it.

Thus, in each of the speakable actor's playturn method, there will be a checking of whether it is its turn to speak using the turnToSpeak() method, and if it returns true, then the Speakable default method of speak() is called, which as mentioned retrieved the String sentence and displays it.

An alternative would be this:
In every speakable actor's playturn method, first check if it is its turn to speak with a turnToSpeak() method defined in its class. Then, if it is its turn to speak, get the String from Monologue with getSentence(), then print it. The downside of this approach is that it would cause repetition in code as getting the string from Monologue then printing it is repeated in every speakable actor. Thus this would violate the Don't Repeat Yourself principle. Also, doing this would introduce a dependency between every speakable actor and the Monologue class, which increases the amount of dependency and thus would violate the Reduce Dependency principle.
Therefore, in our current design, only the Speakable interface has a dependency with the Monologue class, and thus our design adheres to the Reduce Dependency principle. Also, by having every speakable actor implement the Speakable interface, this allows the repetition of code to be reduced and therefore aligns with the Don't Repeat Yourself principle.

Therefore, the above rationale is our justification as to why we implemented the Monologue, and Speakable interfaced in this way in our current design.