FIT2099 Assignment 2
2022 Semester 1
Lab 15 Team 4

Team members:

1) Chan Jia Zheng (32223315)
2) Caelan (30524105)
3) Justin Chuah (32022433)

Rationale

REQ 1

New map:
The new map would be hardcoded in Application, like the first map. This new map will have Lava tiles and will introduce Warp Pipes to both maps, as a means of transportation between the two maps.

New Ground class, Lava:
Lava will simply extend from ground as it behaves almost like a normal Dirt tile would. The main difference is the overriding of the canActorEnter method, where it returns if the actor has HOSTILE_TO_ENEMY capability. We are using this capability as currently only Mario has it, and only Mario is permitted to walk on Lava tiles.

In case of future additions where a new character is introduced where they can also walk on lava, a new status can be implemented and simple refactoring can be done. However, for our current requirements, reusing the HOSTILE_TO_ENEMY capability is sufficient.

The tick function would also be overridden to check if there is an actor on the current Lava tile, and it should deduct 15 hp from the actor if they are.

New Ground class, WarpPipe:
WarpPipe will implement the Jumpable interface, as it is considered a "high ground", since Mario needs to be ON the WarpPipe to warp. The warp action was initially planned to be handled by WarpPipes, but there is no way to create a new WarpPipe and store the instance of GameMap without hard coding the locations, which would cause issues if the map design were to be changed.

New interface, Warpable:
The solution was to make the actors Warpable, instead of ground objects that allow warping. Each Warpable should store a HashMap of the GameMaps available, and their last used pipe's Location. This method however, assumes there will always be a pipe in the top left corner, at (0,0), as that is the default value the previousWarpPoint hashmap is initialized with.

New Actions class, WarpAction:
The WarpAction class will provide the player the option to warp themselves when they are on a warp pipe. The WarpAction class will call the methods in Warpable to actually move the player character.

REQ 2

New Actor class, PrincessPeach:
PrincessPeach is an actor who is only able to return DoNothingAction() until she is freed after obtaining the key from the actor Bowser which will be covered below. She will be freed by the player if the player has obtained the key by returning an action called SavePrincessAction(), which will be explained later.

New Item Class, Key, New Status HAS_KEY:
This Key class is an item that has capability HAS_KEY will then return a SavePrincessAction() when the player comes into range of the actor PrincessPeach. The Key item can be obtained from defeating Bowser.

New Action class, SavePrincessAction:
This action is returned as a menu option in the console when the player is in range of the PrincessPeach actor, and has the Key item in his inventory. This action would then remove the player from the map, print a victory message and end the game.

New Actor class, Bowser:
Bowser is an actor that extends the Enemy abstract class. Reason as to why Bowser extends from Enemy is because it functions similarly to the other actors who extend from it, except that it does have the capability to wander around. So in the constructor of this class, the key that stores the WanderBehaviour in the specified class's behaviour hashmap has been removed. Hence, so as to not repeat a majority of the code unnecessarily, which would then violate the DRY principle, Bowser has extended the Enemy abstract class.

A key item has also been initialised into its inventory that drops when it is defeated by the player, hence the dependency relationship with the Key class. The key can then be brought to the PrincessPeach actor so that the SavePrincessAction() can be returned as one of the menu options to end the game.

Whenever the player comes into contact with this actor class, the actor will then be able to attack and follow the player.

Bowser has also been initialised with a FINAL_BOSS status, which will act as an indicator to allow his AttackAction() to spawn fire if it lands on the player.

New Actor Class, PiranhaPlant:
Similarly to the Bowser class, PiranhaPlant also extends the enemy for the same reasons, which is to not violate the DRY principle. This class only consists of AttackBehaviour() as its lists of behaviours as its only possible moveset is that it can only Attack the player when it comes into its range, since the PiranhaPlant does not move from its location ever. So the list of behaviours will only contain the AttackBehaviour, whenever the otherActor is the player, it will then return an attack action, otherwise it will only return DoNothingAction().

New Abstract Class, Koopa:
Since a Flying Koopa will be introduced later, and that the only difference from a regular GroundKoopa is that it has a higher HP value, and has the ability to just fly over high ground. To adhere to the Liskov Substitution Principle, this abstract class has been created. The original Koopa in assignment 2 has been refactored to GroundKoopa instead.

New Actor Class, GroundKoopa and FlyingKoopa:
These two new classes have been created and extend from the abstract class Koopa for reasons stated above. Except that the FlyingKoopa now has a FLYING status to indicate that it is able to go over tall objects.

REQ 3

New Item class, Bottle:
A new Bottle class has been created to store Water objects which will be covered below. Extending the item class made the most sense as it is required to enter the players inventory, through interaction with Toad via GetBottleFromToadAction which will be covered below since Idecided on doing the optional challenge. The bottle will have the capability to give the holder the Status HAS_BOTTLE, which will indicate whether the player currently wields the bottle. An array list has been created that holds Water objects since it was required for it to function similarly to a stack as per the requirements. The Bottle class was also treated as a singleton class, hence the private attribute constructor and the static method getInstance() in the class. The main reason for the consideration of turning the Bottle class into a singleton class is that because in the entire game, there can only be one existence of this Bottle item, which will require the Player to interact with Toad as explained above, once obtained, it can never be dropped from the inventory.

Changes to actor Toad, and New Action class, GetBottleFromToadAction:
Toad now returns an action that is available to the player in its AllowableActions if the player does not have the HAS_BOTTLE status. The action returned is GetBottleFromToadAction, which will then place a Bottle into the player's inventory, allowing him to be able to fill Water objects from the Fountains which will be covered below. Reason for this new Action is to adhere to Single-Responsibility principles.

New abstract class, Fountain and its child classes, HealthFountain and PowerFountain:
The fountain class is an abstract class that extends from the Ground in the engine code. Reason for the creation of this abstract class is that in the optional requirements, fountains have limited amounts of water and require time to be refilled again if water gets emptied(represented by adding a temporary status IS_DEPLETED that indicates its empty for 5 turns), so to not violate DRY principles and Dependency Inversion Principle, abstract class has been used.
When a player stands on a Fountain , the FillBottleAction() will be returned to the player if the fountain is not depleted and the player has a bottle in its inventory.
When the player uses said action, the fountain will then be presented with a temporary status, WAS_COLLECTED, which will then reduce the available amount of water remaining at the fountain and remove the status right after.
The HealthFountain and PowerFountain both have HEALING and POWERING statuses on the ground respectively. These two statuses would allow the actor who stands on it to know the effects of said fountain.

New abstract class, Water and its child classes, HealthWater and PowerWater:
The water class is an abstract class that has properties similar to their respective sources, the fountains. The water class has been created for SRP purposes, mainly to represent the effects of said water after consumption to the actors. HealthWater only heals the actor who consumed it so that is direct.

However, since the PowerWater provides permanent buffs to the actor who drinks it, a buffable interface has been created for it, mainly to adhere to the Dependency Inversion Principle which will be covered below.
The Power Water provides temporary 1 turn status,POWER_UP, to the actors who consumed it so that the effects can be handled accordingly.

New action class, FillBottleAction:
This action class uses a location class and an integer representing the remaining amount of water at the fountain as its parameters in its constructor. The integer is mainly to be used to return console messages neatly as per requirements. The location is used to know what kind of water to be filled in the Bottle. Bottle is filled with water by knowing what kind of fountain the actor is currently standing on.
New Behaviour class, DrinkBehaviour:
This behaviour like other behaviours implements the behaviour interface and was created to return DrinkWaterAction if the actor that is not the player is standing on a fountain that is not depleted. This behaviour has been balanced so that it will never be allowed to occur for the actor that is not the player twice in a row.

New Action class, DrinkWaterAction:
If the player currently has a bottle and that it is non-empty, the player will then drink the latest water that was added into the bottle, by getting the instance of the Bottle. After drinking, the water that was drunk will first have its effects applied onto the player and will then be removed from the bottle. Once the bottle has been emptied, this action will no longer be available as the list of choices in the menu description.
If an actor that isn't the player drinks from the fountain, a DRANK_FORM status will be added to the respective fountain so that the remaining amounts of water can be reduced accordingly.

New Interface, Buffable, Changes to Player and Enemy:
Player class and actors who extend from Enemy abstract class now have private attributes powerBuffCounter to keep track of the effects of PowerWater on said actors.

This interface has been created so that it can keep track of actors who have been buffed, at the current stage only after the actor has drank the water from the Power Fountain. The main reason for the creation of this interface is to adhere to the Dependency Inversion Principle. The interface is there to keep track of the number of times an actor has drank the powe water by getting the counter found as part of the attributes in said actor classes. Since the buffs only affect the base damage of the actors, the getIntrinsicWeapon abstract method has been overridden, returning the respective base damage of actors with addition of the number of times the actor has been buffed* 15, which is also known as the getAttackIncrease method created at the interface.

REQ 4

New Consumable class, FireFlower:
Implementation for the FireFlower class is rather simple. It will extend from our previously implemented Consumables class, and grant the consumer the SHOOTING_FIRE status, which will allow other methods to grant the actor the ability to spawn fire with its attacks, for a certain number of turns.

Refactoring Player's playTurn:
With the possibility of having 2 timed power ups simultaneously, we needed a way to track the power ups better, since we need to keep the possibility of more timed power ups being implemented in the future.

New attribute, ConcurrentHashMap<Status, int> timedStatusHashMap:
Whenever a status is added, its corresponding duration is added to the timedStatusHashMap. Everytime Player's playTurn is run, it goes through the timedStatusHashMap and subtracts the duration by 1. It then prints the remaining duration of the status in a message or removes the status when the duration hits 0.

We also use a ConcurrentHashMap instead of a normal HashMap because a normal HashMap does not allow modifications to its contents while being iterated.

New Item class, Fire:
It made the most sense to implement the Fire spawned from Fire attacks as an item. This is because there can be multiple Fires in one location, so it would not be feasible to implement it as a ground class. The fire class is also initialized to not be portable, as it shouldnt be able to be picked up.

The tick function for Fire checks for actors sharing the same location as it, and deducts hp from said actors, then it reduces the current duration by 1 and removes itself when the duration hits 0.

New Action class, FireAttackAction:
This class is created to represent a new action that is possible for the player when it comes under the effects of the Fire Flower. Since the Fire Flower allows the player to be able to set fire to the target’s ground, it just gets the location of the target on the map, and creates a Fire on said location. If the player happens to also be under the effect of the PowerStar(), after creating the Fire on said location, the target will then be killed immediately. Hence, Single-Responsibility Principle has been adhered to rather than just adding more conditional checks in AttackAction().

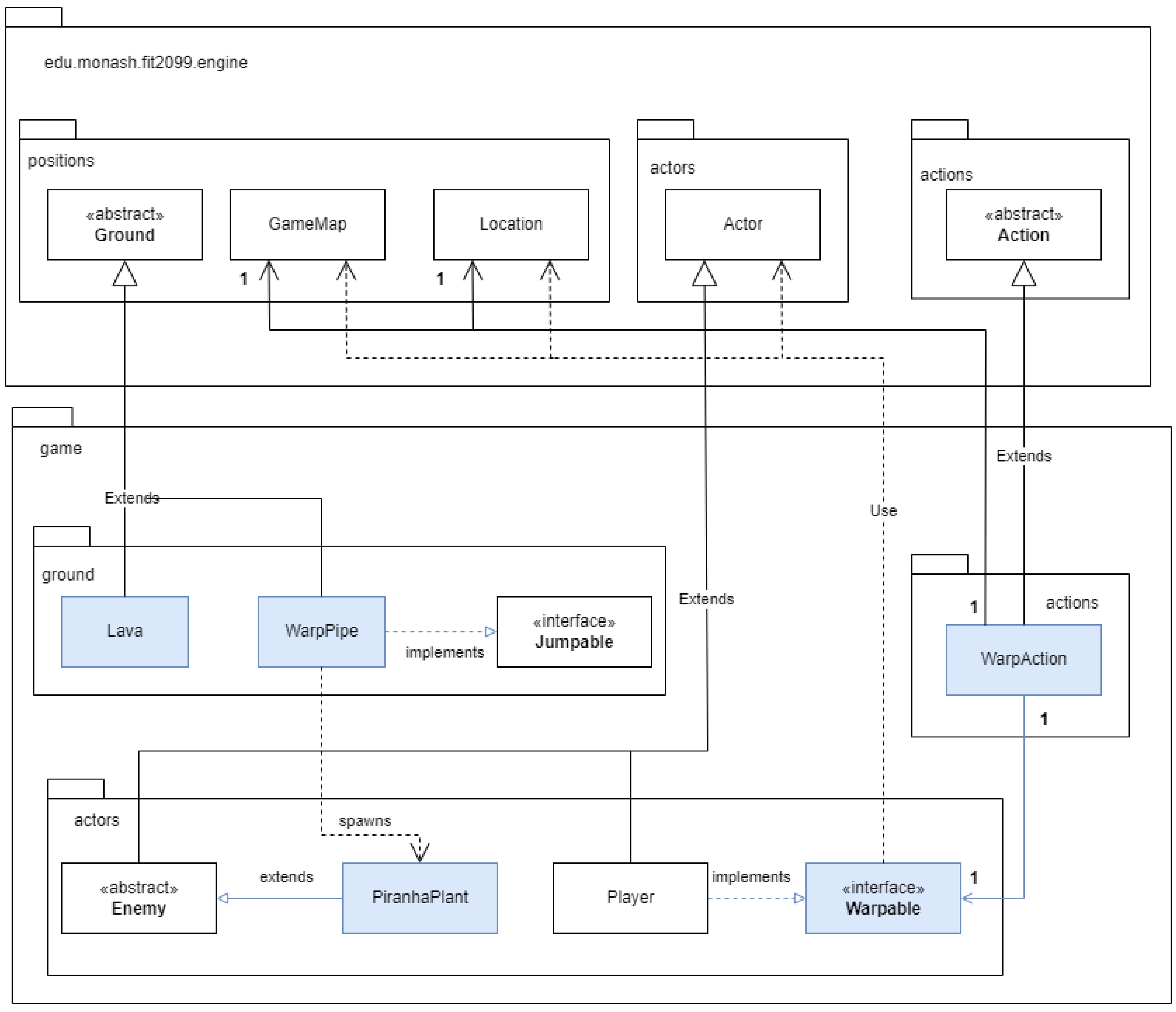
REQ 5

Newclass, Monologue:
Since the actors that can talk shouldn’t extend from a concrete class to reduce coupling, I created the Class Monologue. The Monologue class will handle all the logic for the monologues (Single Responsibility Principle).Every actor that can talk should have a class attribute monologue of type Monologue. Also, the actors should have a static final string[] attribute called sentences. I use it to create the Monologue object for the actors in their constructor. In the future if we want to modify the sentences the actor should say, we can directly modify the static variable sentences.

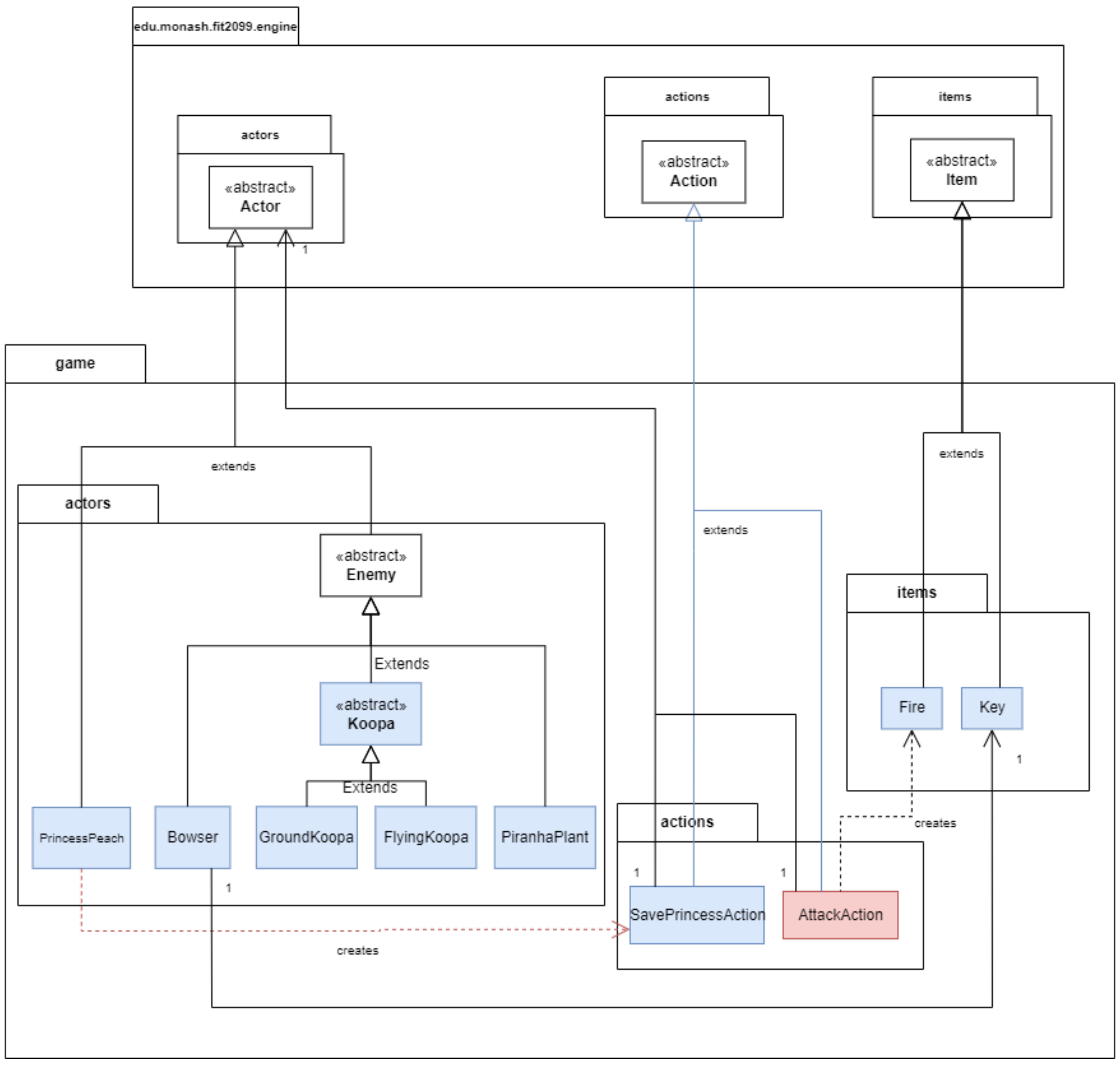
As Flying Koopa has an additional sentence, I added a method called addSencente() in Monologue. As such, we can customly add sentences to the Monologue object. In the future if we want to customise the sentences that an object can say, we can add methods to add or remove sentences from the Monologue class attribute sentences.

As all the Enemy should talk, I put the attribute monologue in the abstract class Enemy. We can see that all actors except Player have a monologue, the designer can consider adding this attribute to the class Actor to reduce the repetition of code in its children class (DRY principle). All the actors who have a monologue will show their monologue within their playTurn() method, using monologue.show().

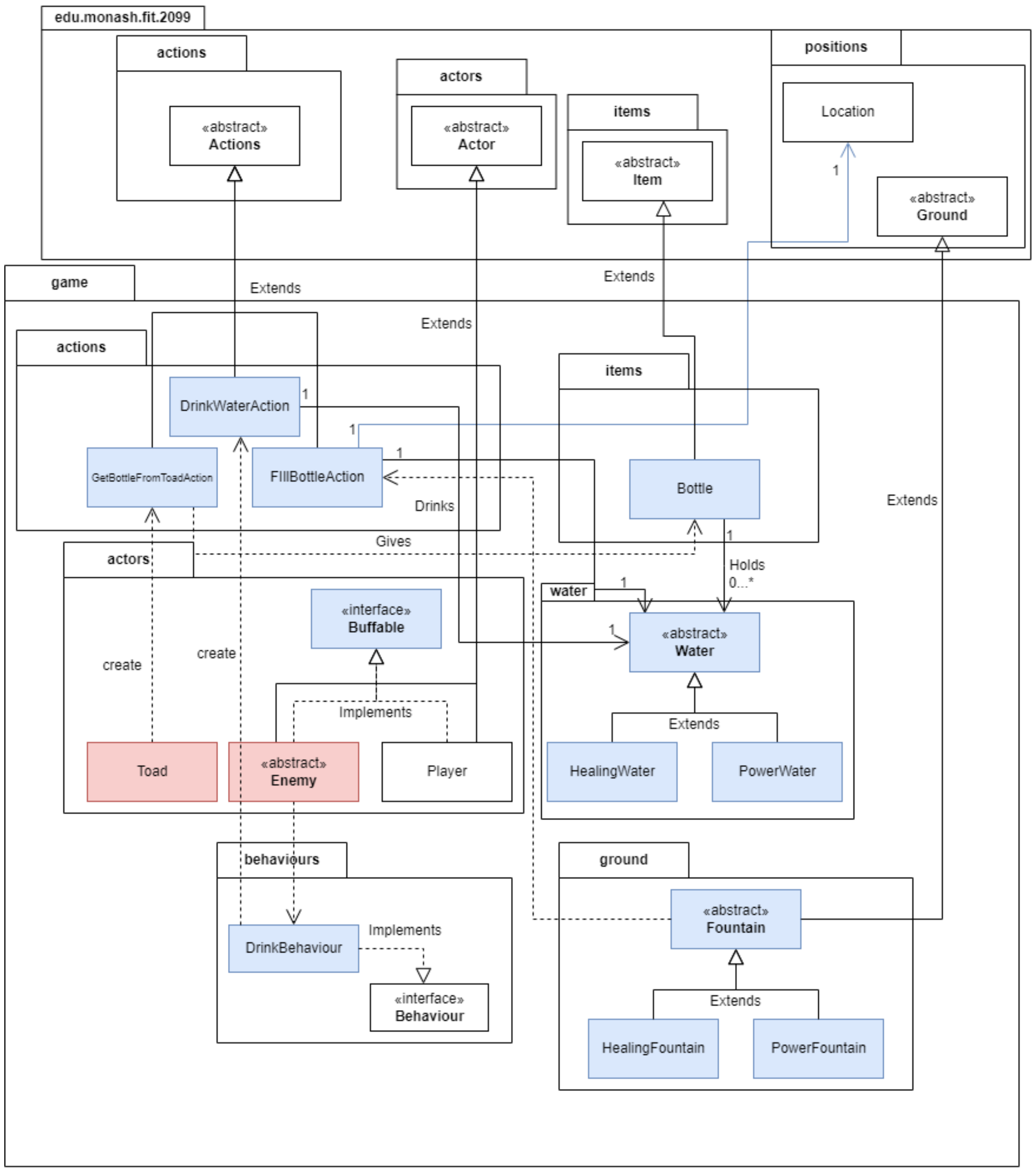
UML Class Diagram Requirement 1



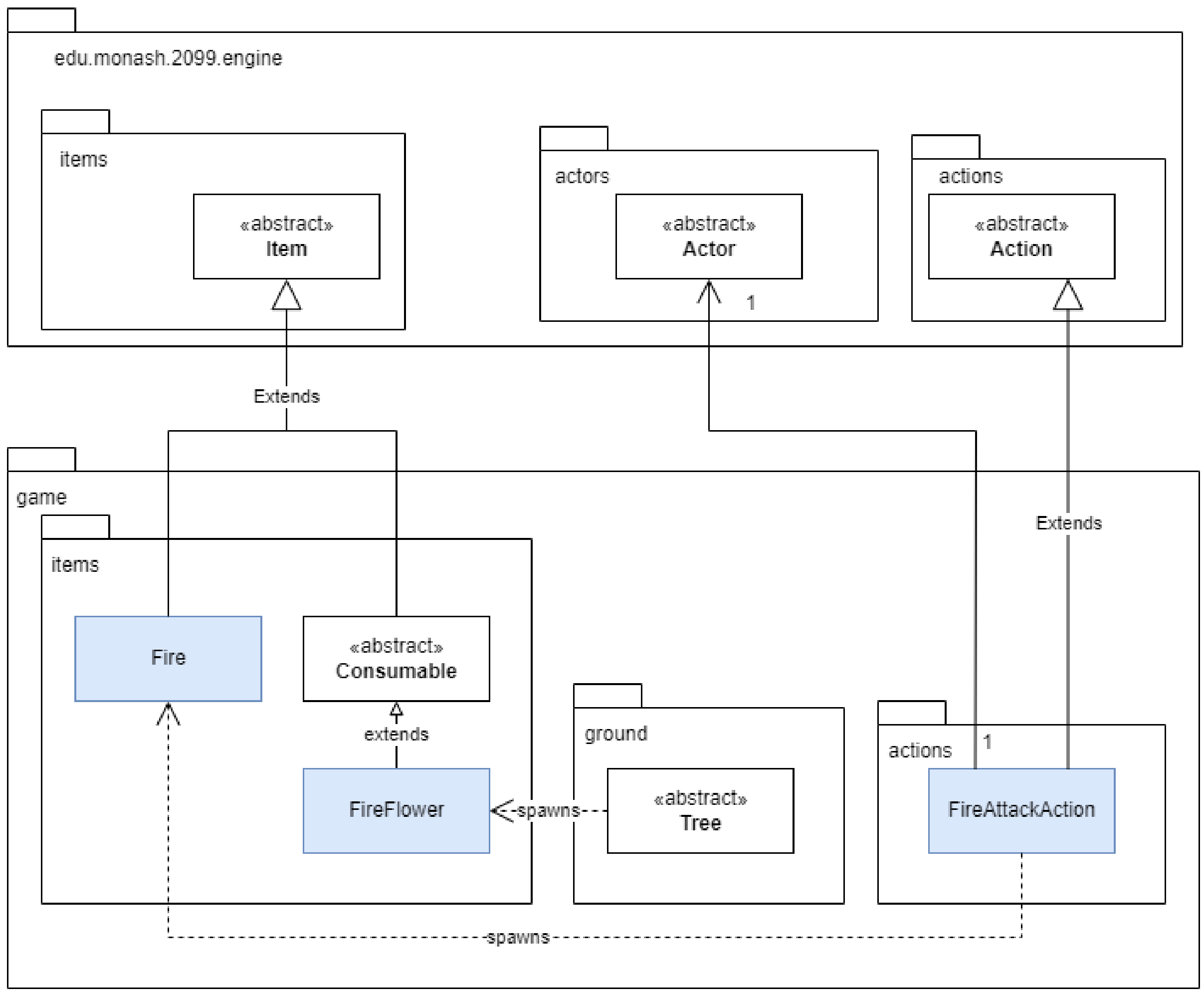
UML Class Diagram Requirement 2



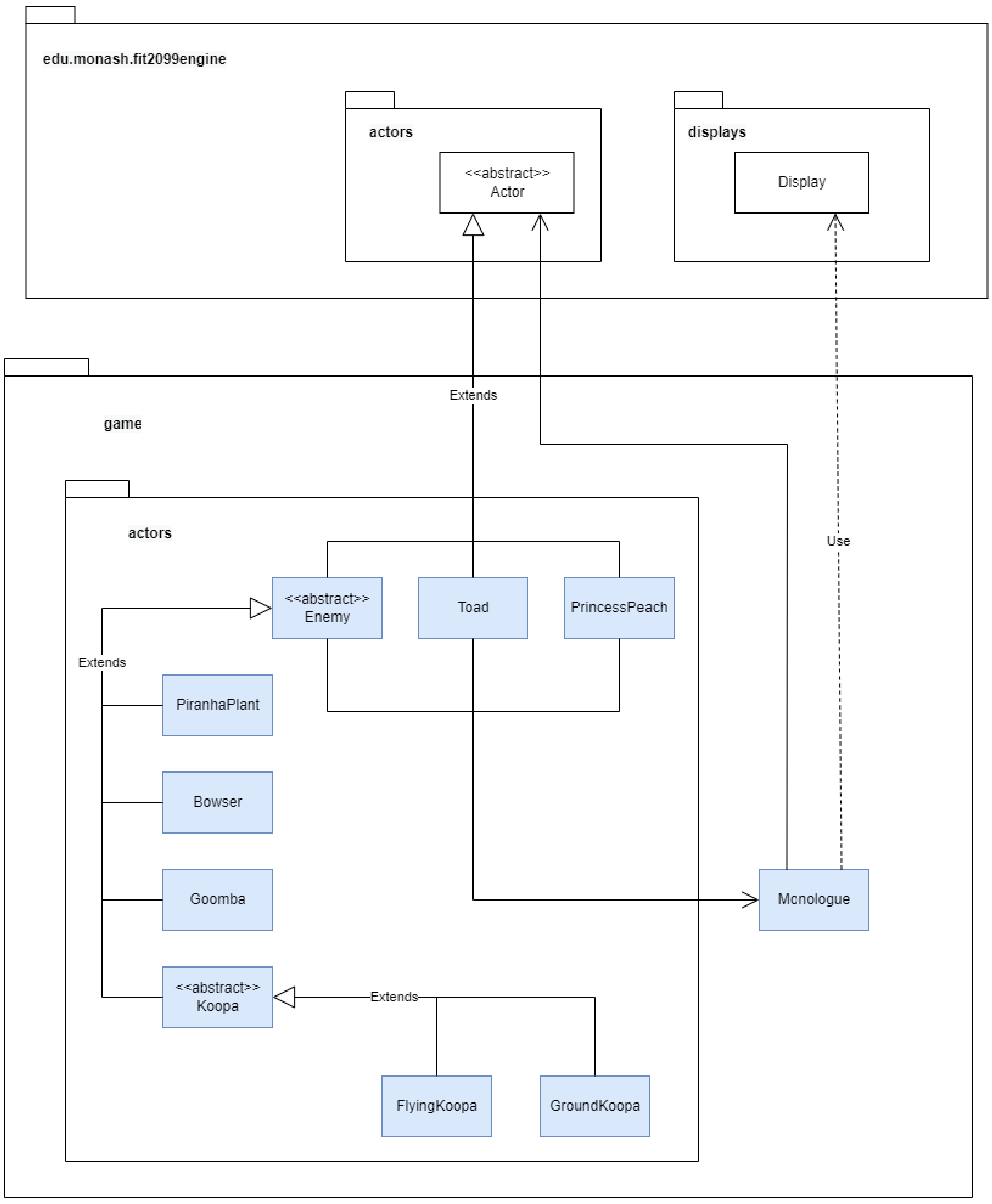
UML Class Diagram Requirement 3



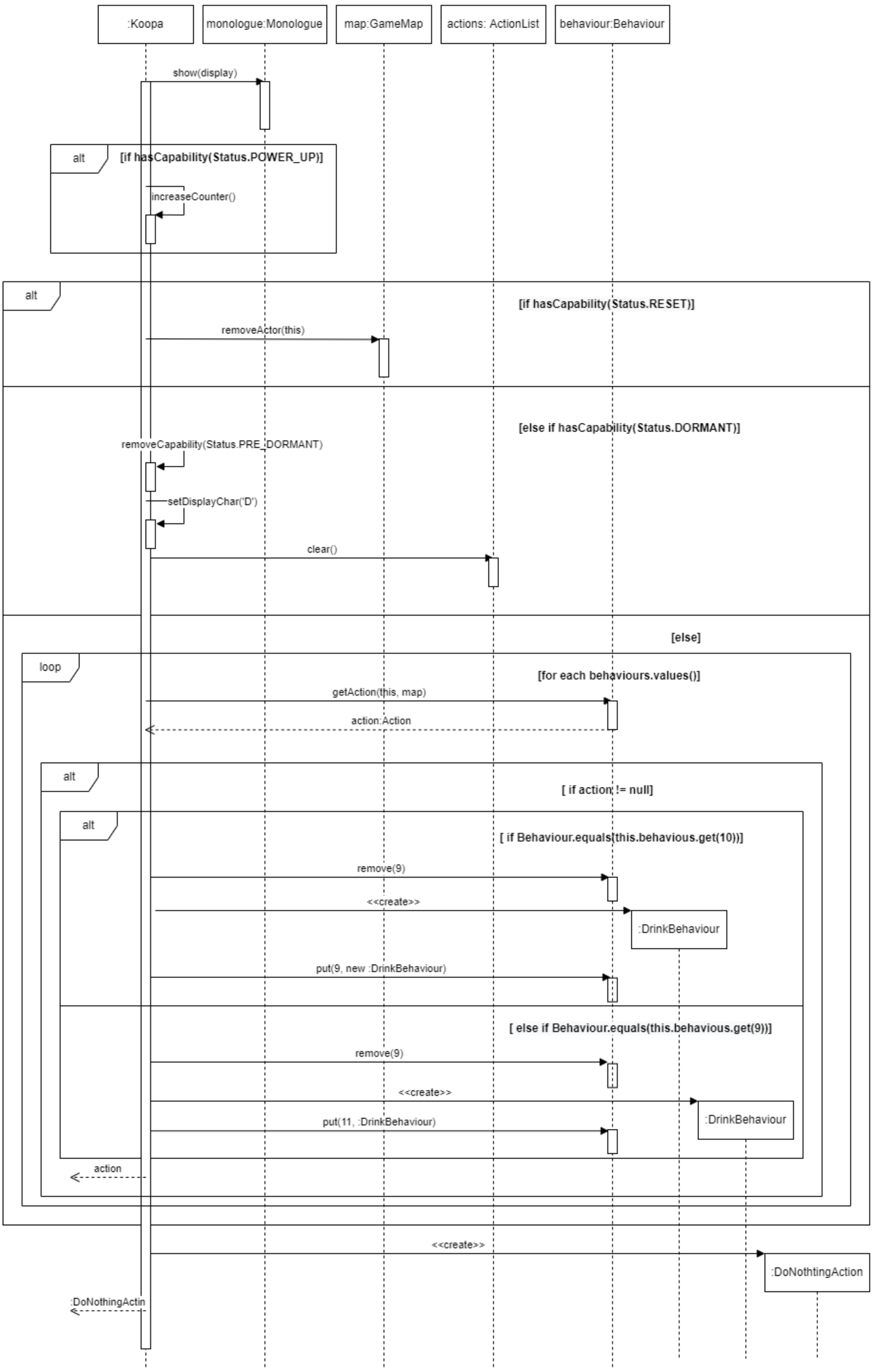
UML Class Diagram Requirement 4



UML Class Diagram Requirement 5



UML Sequence Diagram 1
Method playTurn() in abstract class Koopa



Link to the google drive
https://drive.google.com/drive/folders/15GM9REcm9RnQOhpAVWkoOM-GyStR13u1?usp=sharing

We hereby agree to work on FIT2099 Assignment 2 according to the following breakdown:

We will work on the requirements most related to the requirements done in Assignment 2, along with their corresponding UML diagrams.

- Caelan - Requirements 1 & 4
- Justin - Requirements 2 & 3
- Jia Zheng - Requirements 5, all reset functionality, interaction diagram

Quality control done by Jia Zheng

Any bugs or errors detected during testing are dealt with as a group.

I accept and agree to this WBA - Caelan Kao (30524105)
I accept and agree to this WBA - Chan Jia Zheng (32223315)
I accept and agree to this WBA - Justin Chuah (32022433)