Work Breakdown Agreement for FIT2099 Assignment 3

Team 33 - Lab 13:

7. Ong Di Sheng (31109667)
8. Mark Gabriel Sta. Ana Manlangit (29350387)
9. Kennedy Tan Sing Ye (31108121)

We thus agree to work on FIT2099 Assignment 3 as outlined below.

No	Task	Assigned	Reviewer	Deadline
1	**Class Diagram**			
1.1	REQ 1 - Added Lava, WarpPipe and TeleportAction classes	Mark Manlangit	Kennedy Tan & Di Sheng	22/5/2022
1.2	REQ 2 - Add Princess Peach, Bowser, Piranha Plant, Flying Koopa and RescueAction	Kennedy Tan	Mark Manlangit & Di Sheng	22/5/2022
1.3	REQ 3 (With optional challenges) - Added classes such as Fountain (abstract), Water (abstract), Bottle, RefillAction and ObtainBottleAction Creative mode 1 (blink + patrol) - Added classes such as Luigi, BlinkAction, BlinkingTower and PatrolBehaviour Creative mode 2 (Yoshi as adventure partner) - Added classes such as Yoshi, HealBehaviour and HealAction	Ong Di Sheng	Kennedy Tan & Mark Manlangit	22/5/2022
2	**Sequence Diagram**			
2.1	REQ 1 - Draw a TeleportAction sequence diagram	Mark Manlangit	Kennedy & Di Sheng	22/5/2022
2.2	REQ 2 - Draw a RescueAction sequence diagram	Kennedy Tan	Mark Manlangit & Di Sheng	22/5/2022
2.3	REQ 3 - Draw a ConsumeAction (account for Water), RefillAction and ObtainBottleAction sequence diagram	Ong Di Sheng	Kennedy & Mark Manlangit	22/5/2022
3	**Design Rationale**			

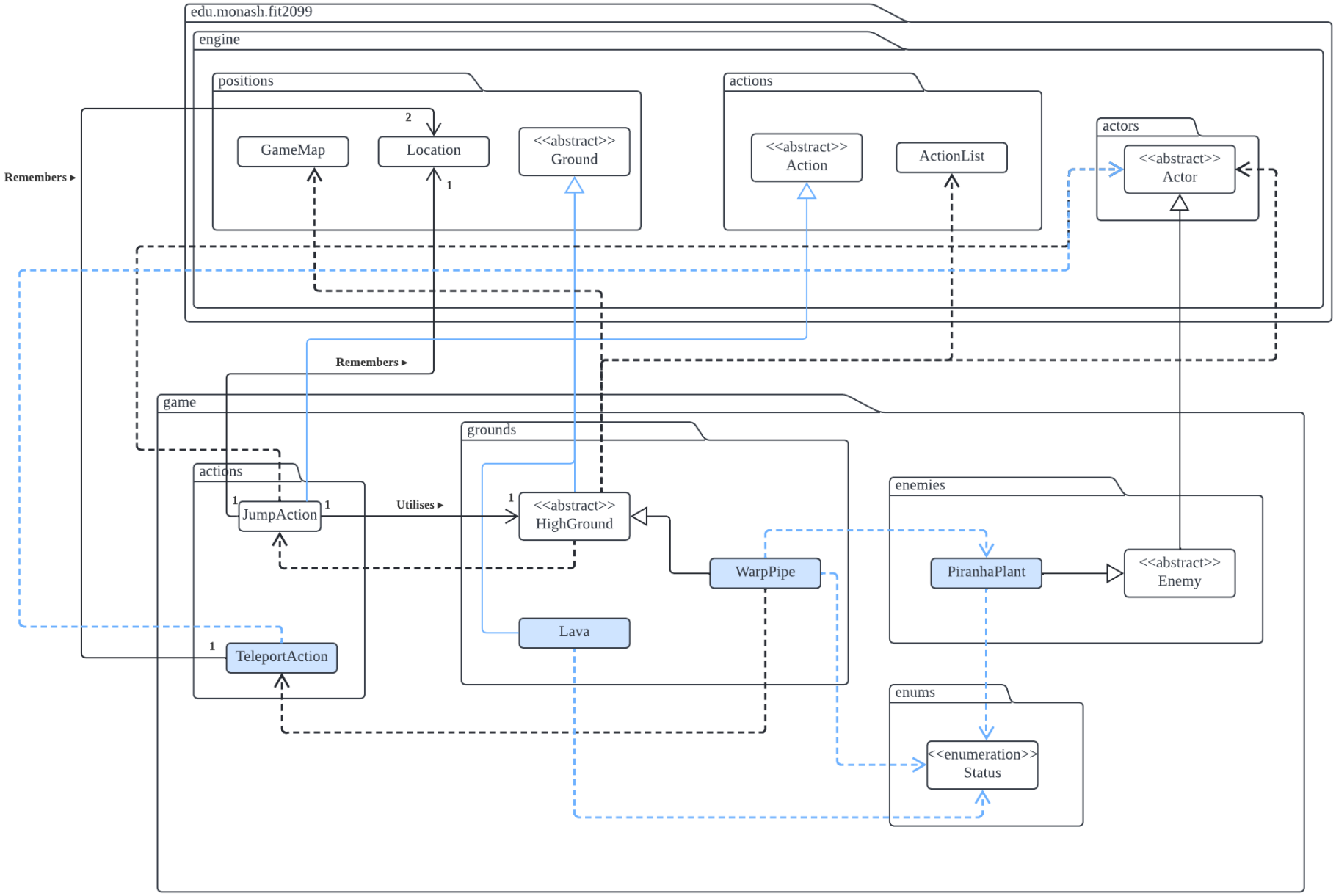
3.1	REQ 1 - Explain how SOLID principles were achieved in implementation of Lava, WarpPipe and TeleportAction classes	Mark Manlangit	Kennedy & Di Sheng	22/5/2022
3.2	REQ 2 - Explain usage of classes/interfaces involved in RescueAction and AttakAction using SOLID principles	Kennedy Tan	Di Sheng & Mark Manlangit	22/5/2022
3.3	REQ 3 - Explain the usage of classes/interfaces involved in Magical Fountain using SOLID principles Creative mode 1 (blink + patrol) - Explain the usage of BlinkAction + PatrolBehaviour using SOLID principles Creative mode 2 (Yoshi as adventure partner) - Explain the usage of Yoshi, HealBehaviour and HealAction using SOLID principles	Ong Di Sheng	Kennedy & Mark Manlangit	22/5/2022

Signed by (type “I accept this WBA”)

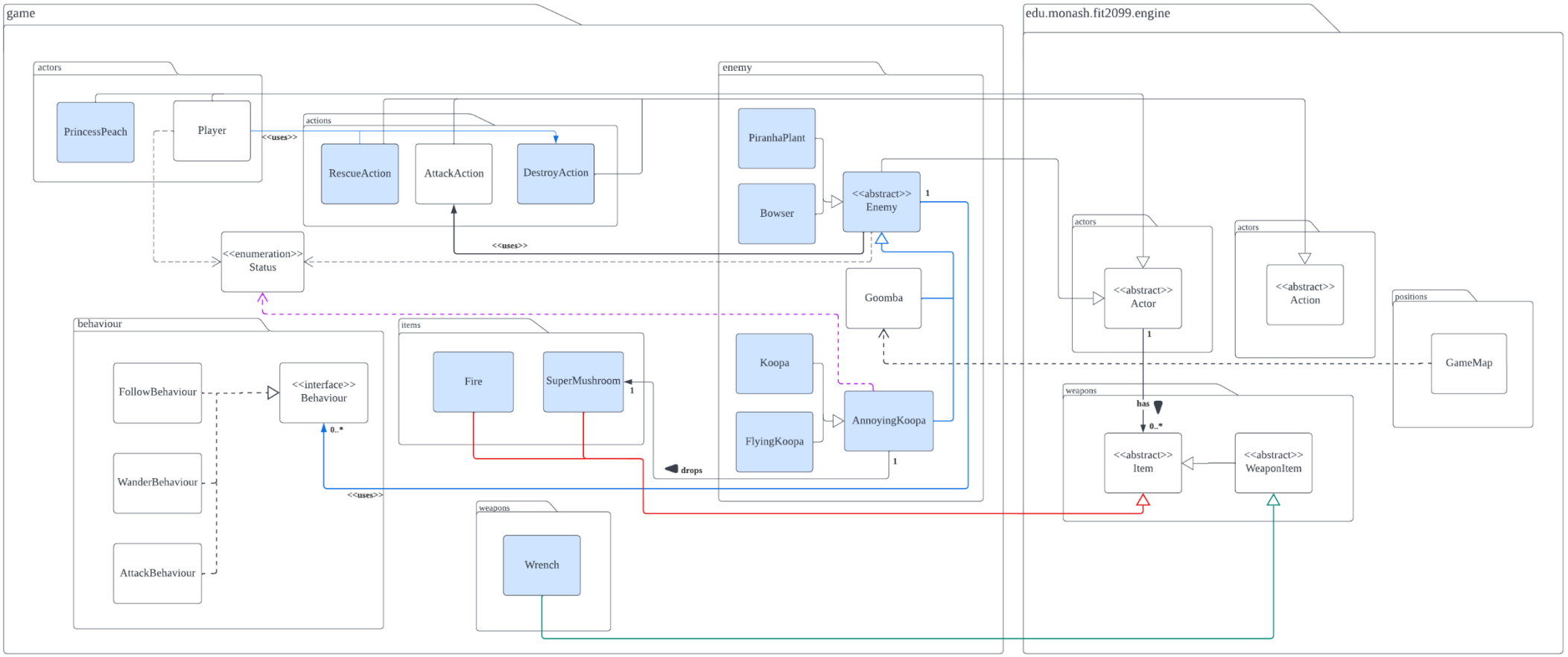
I, Ong Di Sheng accept this WBA
I, Kennedy Tan accept this WBA
I, Mark Manlangit accept this WBA

Class Diagrams (Blocks in colours are new Classes/Interfaces/Enumerations created)

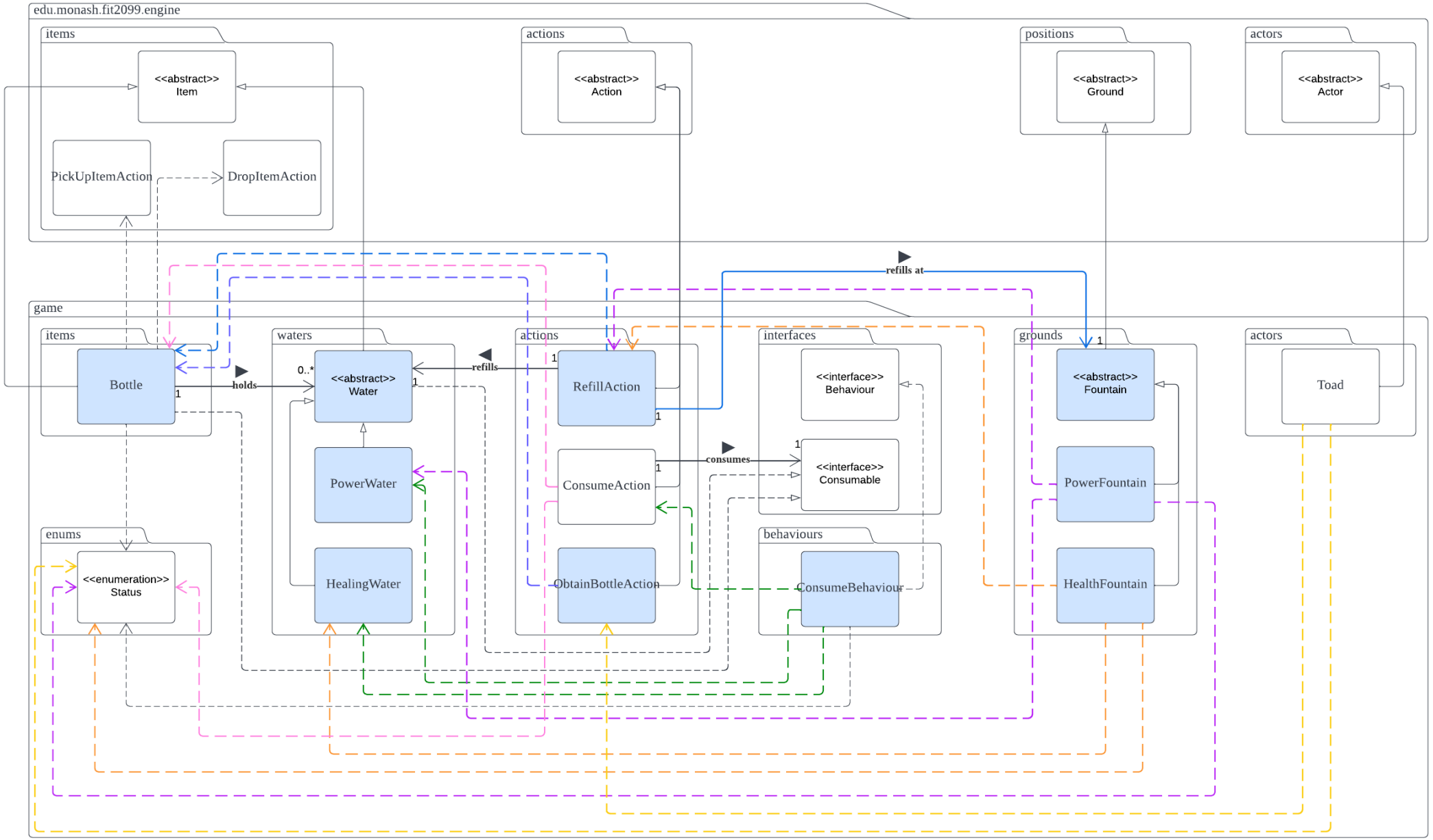
REQ1: Lava Zone



REQ2: More allies and enemies!

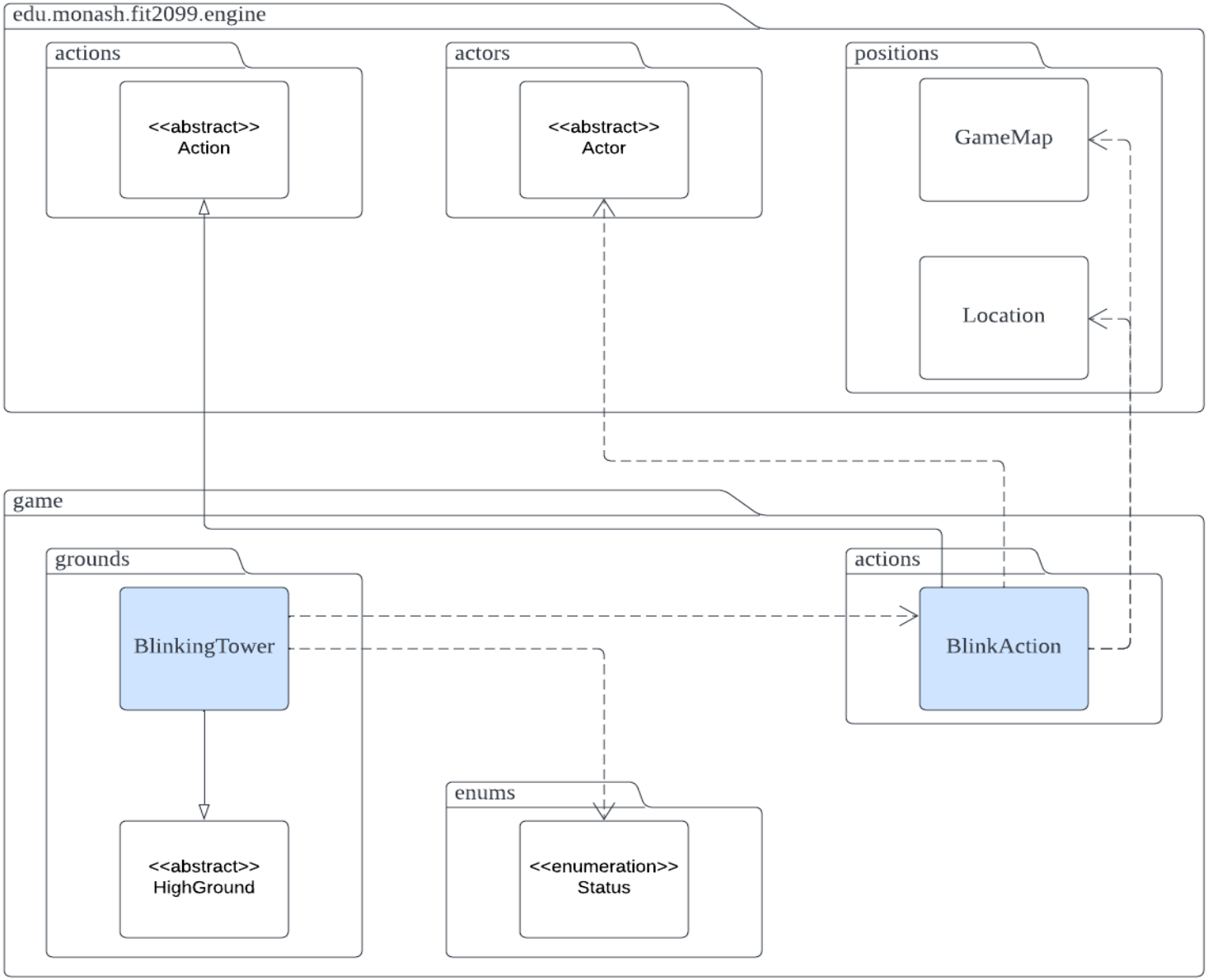


REQ3: Magical Fountain (With Optional Challenges)

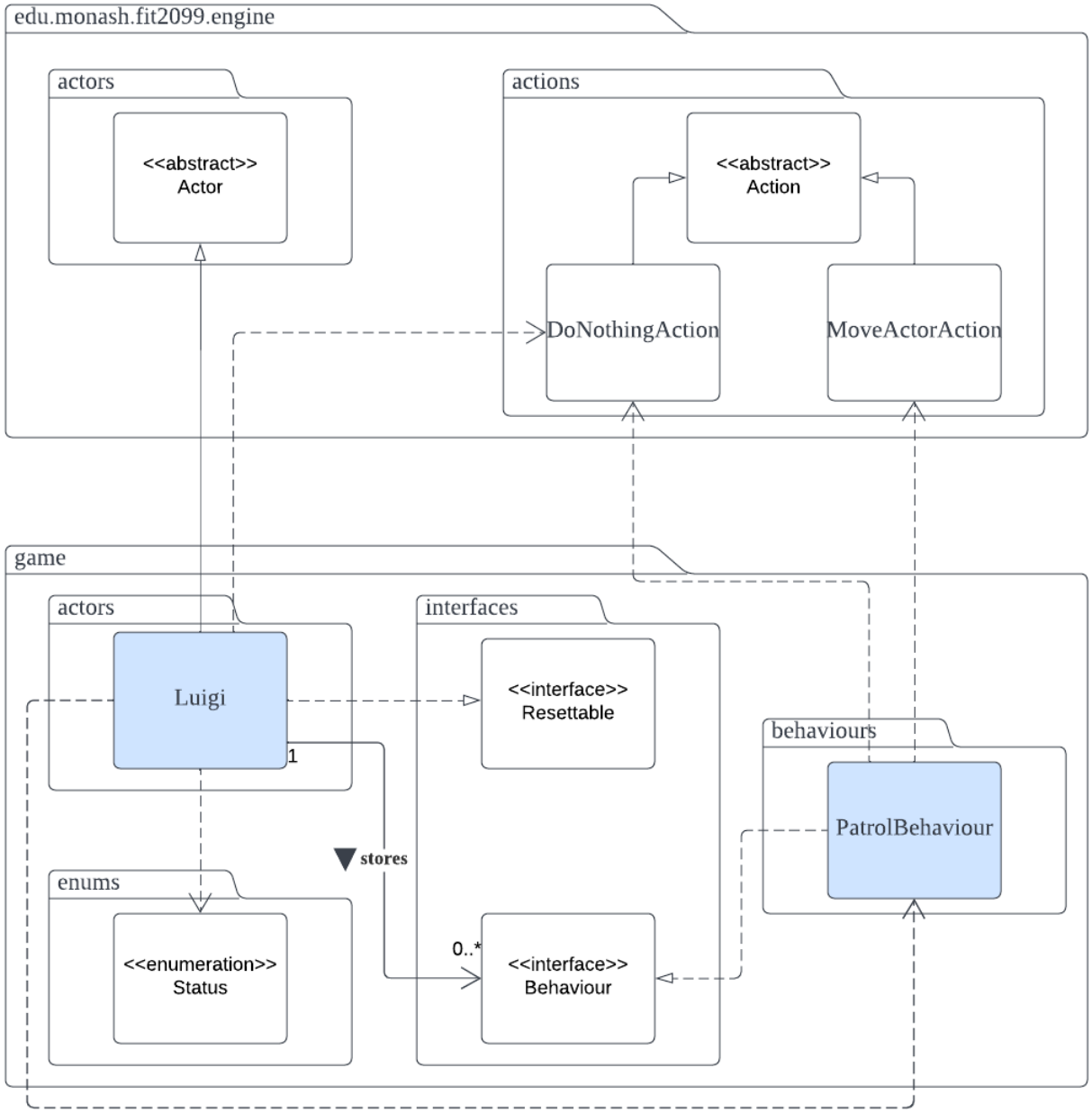


REQ 4:

Title: Blink action

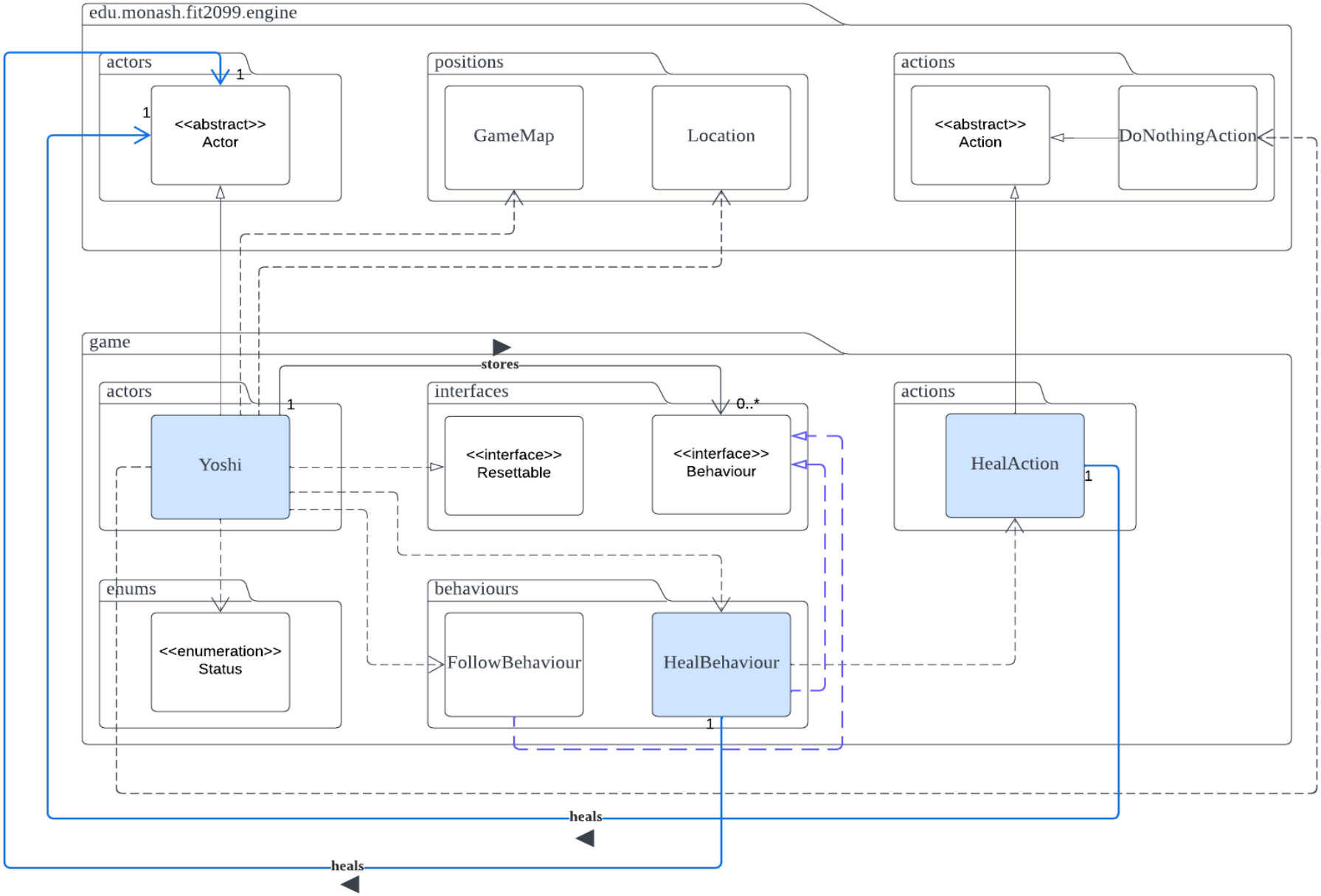


Title: Patrol behaviour



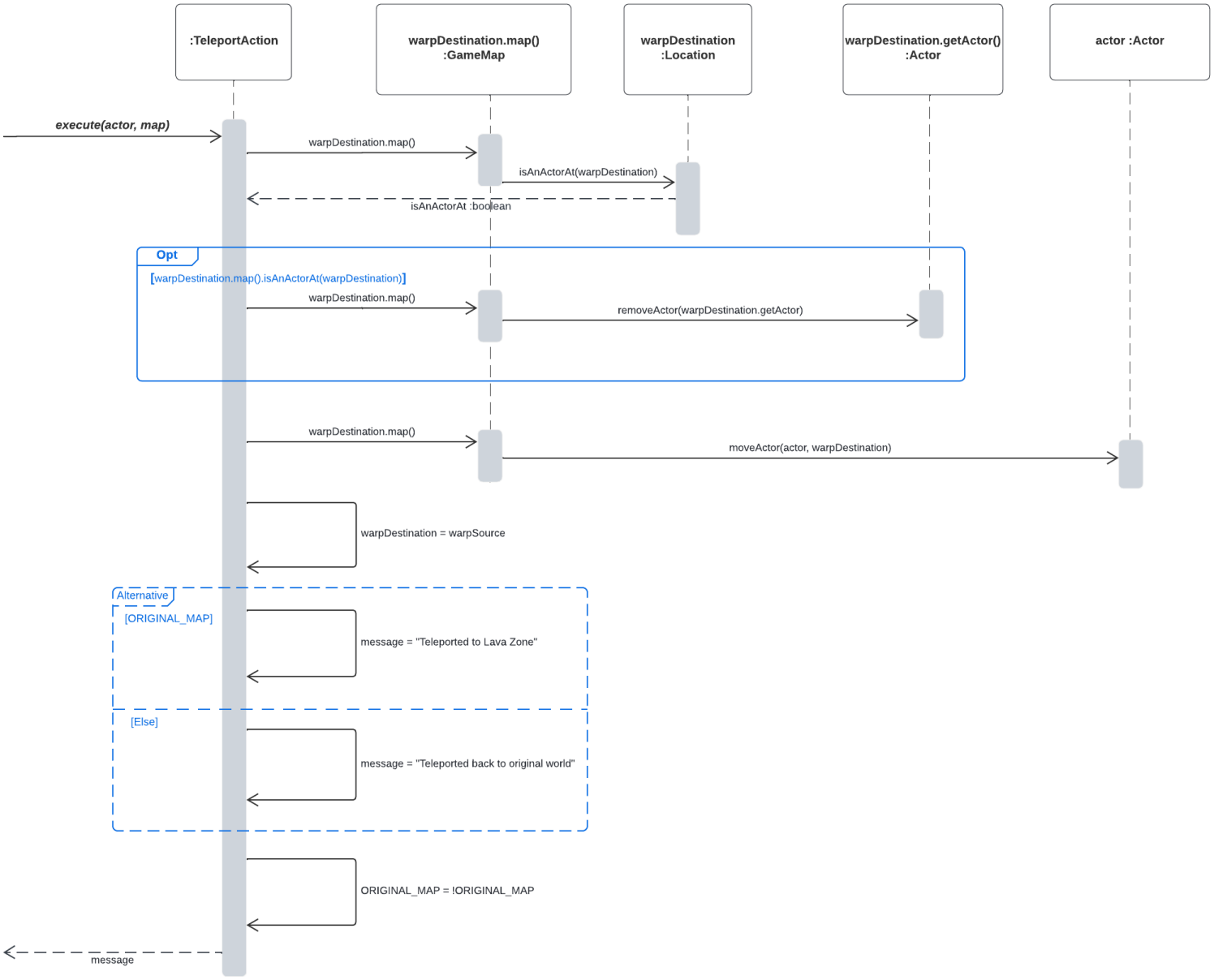
REQ 5:

Title: Yoshi as adventure partner

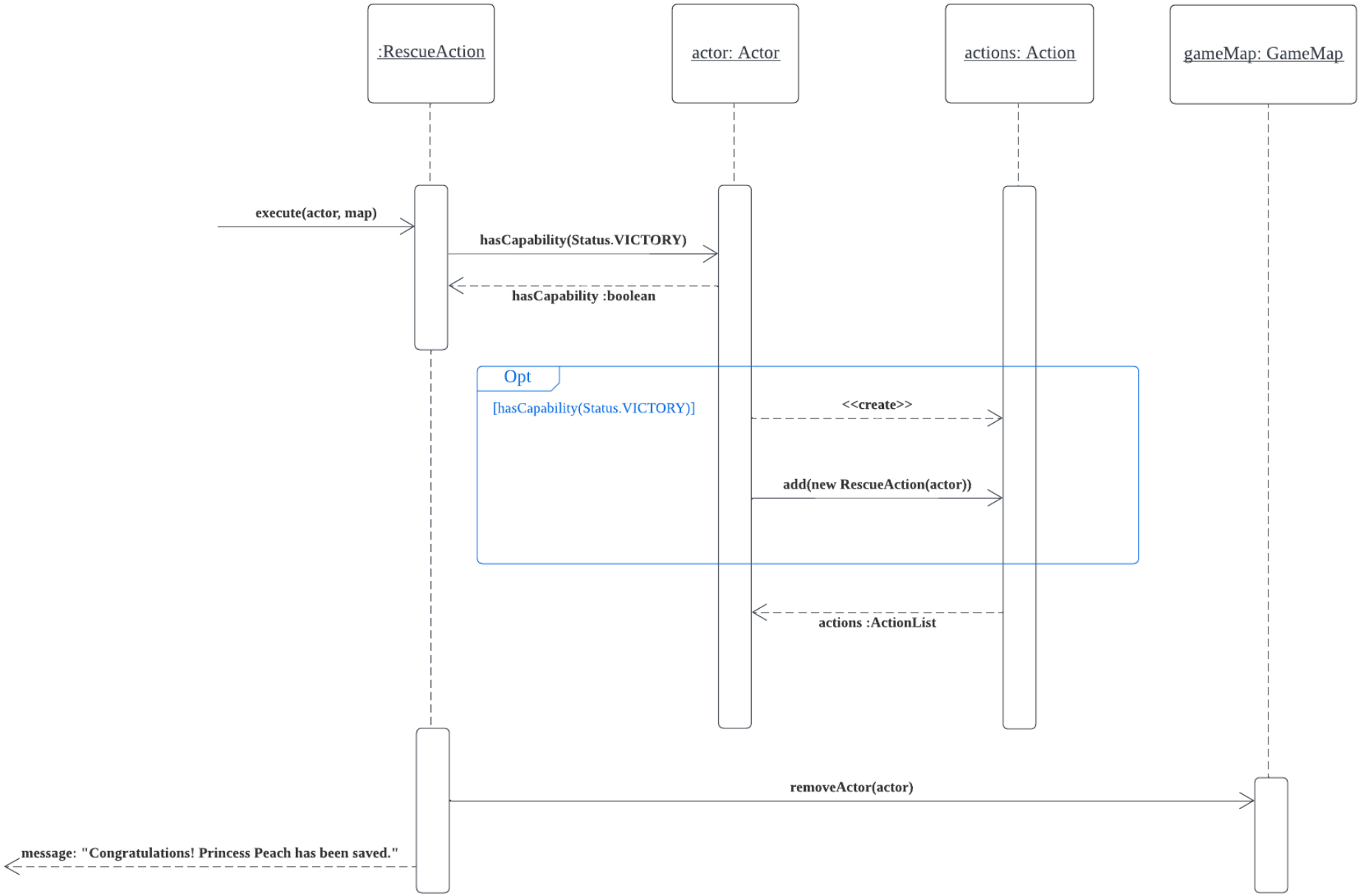


Interaction Diagrams

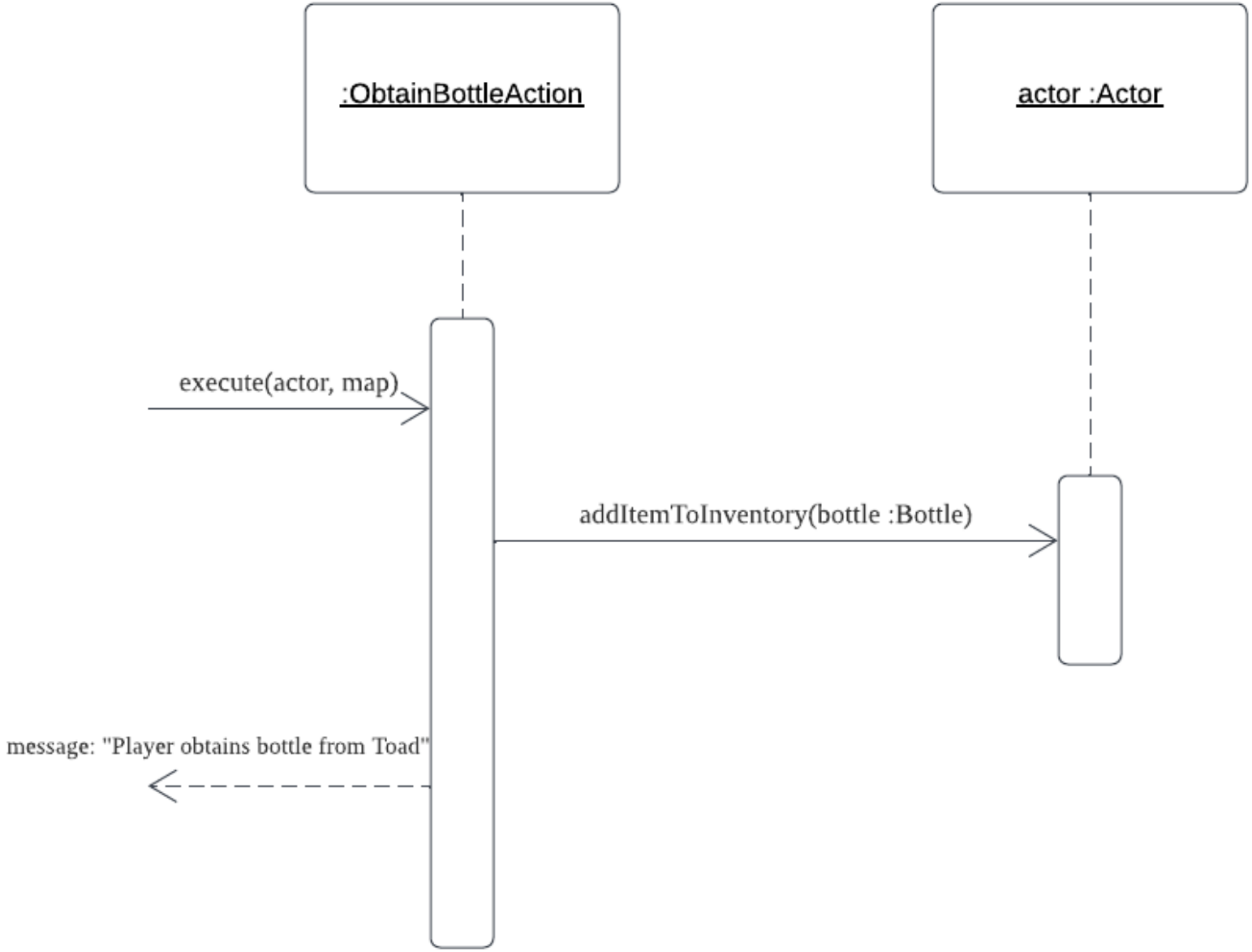
TeleportAction



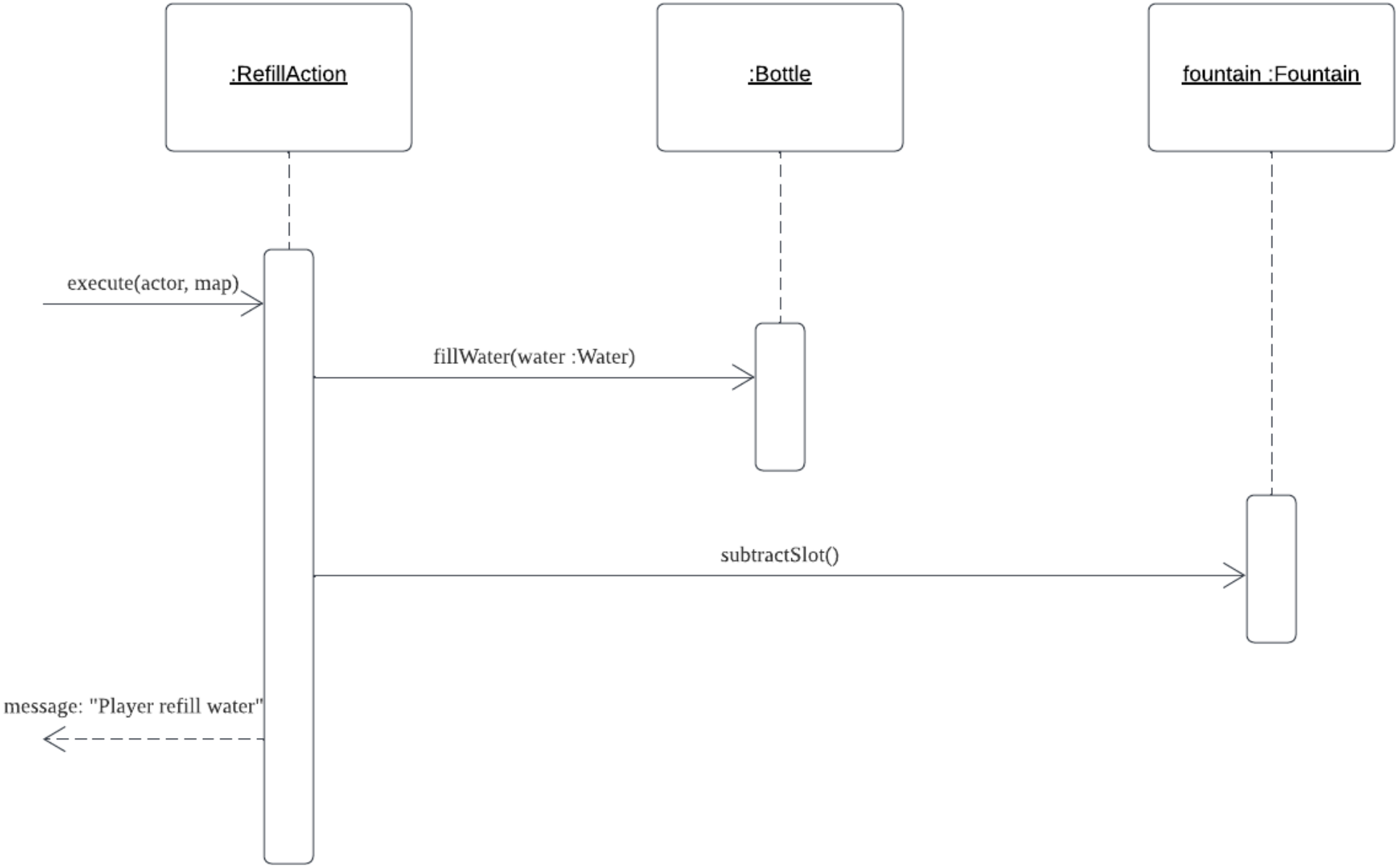
RescueAction



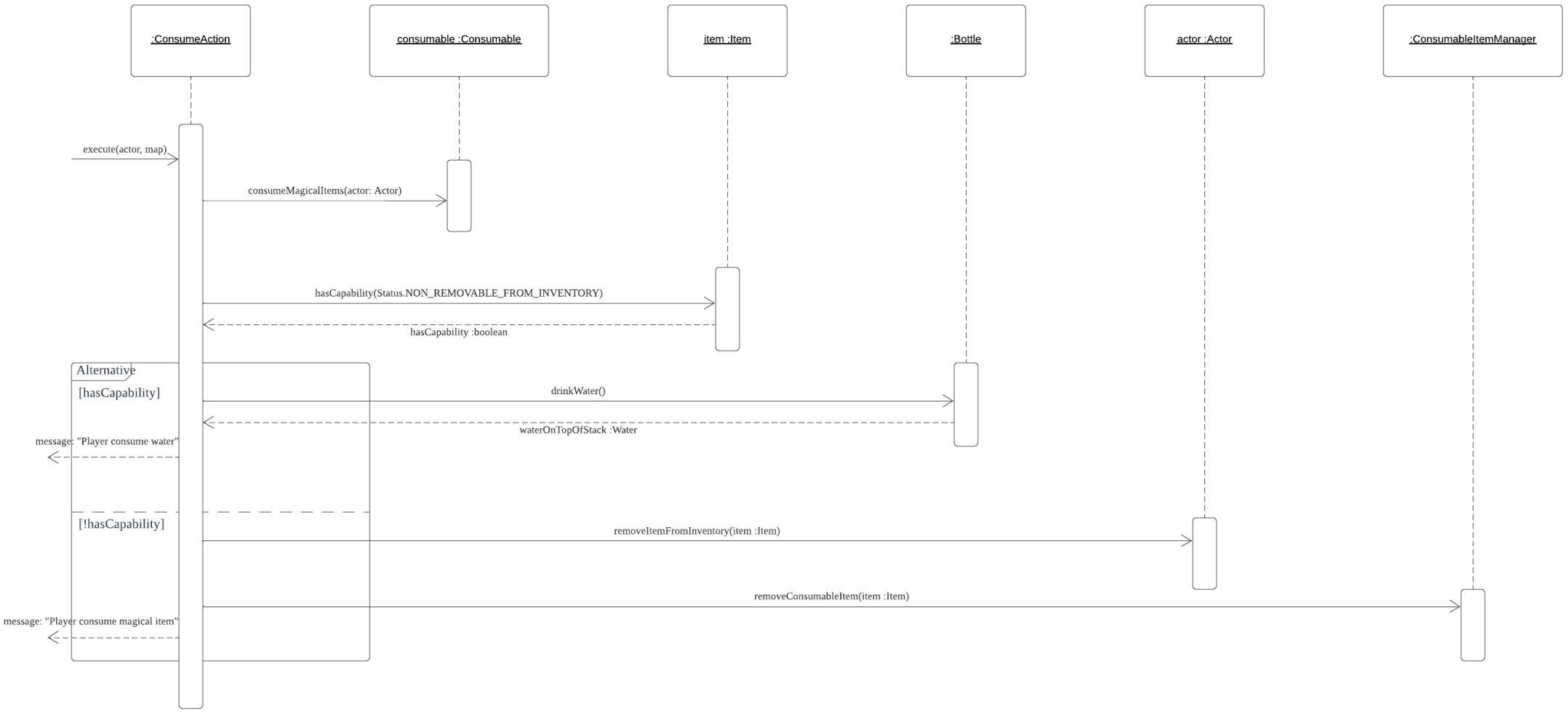
ObtainBottleAction



RefillAction



ConsumeAction



FIT2099 Assignment 3

Lab 13 – Group 33
Members: Ong Di Sheng, Kennedy Tan, Mark Manlangit

Design Rationale

REQ 1: Lava zone

a) Lava

Lava is a class that extends the Ground abstract class. This class represents a blazing fire ground that will damage the player when the player steps on it. Enemies will not be able to step on this ground. Lava has satisfied the **Open-closed Principle** as it has extended the Ground abstract class to provide a new type of Ground to the game that damages players that stand on it, all without requiring any modifications to the Ground class itself.

b) WarpPipe

WarpPipe is a class that extends from the HighGround abstract class. This class represents a warp pipe that spawns a piranha plant on top of it. If the player kills the blocking piranha plant, the player can jump on top of the warp pipe and initiate a teleport to another pipe. If the player initiates a reset, a new Piranha plant will spawn on empty warp pipes. WarpPipe has satisfied the **Open-closed Principle** as it has extended the HighGround abstract class to provide new functionality to the game such as spawning PiranhaPlants and providing a TeleportAction, all without requiring any modifications to the HighGround class itself. It also satisfies the **Single Responsibility Principle** as it is only concerned with the responsibility of spawning Piranha plants and providing the player with the option to teleport, but has delegated the implementation of the actual teleport to a different Action class.

c) TeleportAction

TeleportAction is a class that extends from the Action abstract class. This class allows a player to teleport from a warp pipe to another warp pipe on another map. Once there, the player will have an option to teleport back to the original world at the same warp pipe that teleported him before. TeleportAction has satisfied the **Open-closed Principle** as it has extended the Action abstract class to provide new functionality by allowing players to travel to different maps in the game, all without requiring any modifications to the Action class itself. It also satisfies the **Single Responsibility Principle** as it has separated the responsibility and implementation of performing a teleport action from the WarpPipe, ensuring that each class is only concerned with a single part of the game's functionality.

REQ2: More allies and enemies!

a) **Enemy**

Enemy is a new abstract class that extends from Actor abstract class. It contains the general characteristics of the actors that can deal damage to the player. With abstraction, we managed to extend the code functionality without modifying the existing Enemy class. This follows the **Open-Closed principle** which has define as, classes should be open for extension but closed for modification. In Assignment 3, we did not change anything on this class. In Assignment 2, our implementation follows the design in Assignment 1. In Assignment 1, we have two enemies, which are Goomba and Koopa. Goomba and Koopa inherit from Enemy abstract class. In Enemy class's constructor, WanderBehaviour and AttackBehaviour are added into the Behaviours Hash map in the Enemy class. As they are the same characteristics of all enemies.

This also follows the principle of **Don't Repeat Yourself**. In the future, if the enemies have new common behaviours, we just have to add it in the Enemy abstract class, instead of repeating them in Goomba, Koopa and other new enemies' classes. For their own special characteristics, they should be added in their own classes. For instance, Koopa will be change to dormant state if defeated.

b) **Behaviour**

Behaviour is an interface that included in the based code which determines the current behaviour of the enemy. It is split into three behaviour which are FollowBehaviour, WanderBehaviour and AttackBehaviour. So that the enemy can only implement the methods they need at that specific situation. This follows the **Interface Segregation Principle** which is defined as larger interface should be split into smaller ones.

c) **AnnoyingKoopa**

AnnoyingKoopa is an extend of Enemy class. For Assignment 3, we have change the actual Koopa class to AnnoyingKoopa class, which is an abstract class. The implementation of AnnoyingKoopa class is same as Koopa class that we design in Assignment 2. Thus, we extend Koopa and FlyingKoopa classes from the AnnoyingKoopa class. Koopa and FlyingKoopa will both turn to dormant state whenever it is unconscious and drops SuperMushroom when it is cracked by wrench. However, FlyingKoopa is allowed to move around high ground when it wanders around, which makes it different from the actual Koopa. Instead of repeating the

same code in their own classes, we extend them from AnnoyingKoopa, and add on their own implementations. This satisfied the principle **Don't Repeat Yourself.** For instance, if we have SwimmingKoopa in the future, we do not have to repeat the code that turn Koopa to dormant state, we just have to extend it from AnnoyingKoopa.

Koopa and FlyingKoopa are the enemies in this system, but it has different characteristics compared to Goomba. Koopa and FlyingKoopa will go to dormant state by using Status enumeration and it is not allowed to move around and when it is destroyed by the player using wrench, it will drop a SuperMushroom by DropItemAction. These characteristics make it much different compared to Goomba. So, instead of putting all types of enemies inside Enemy class and using if statement to implement their characteristics, we create classes for every enemy and extends from Enemy class. In this way, also their general characteristic is defined in the Enemy abstract class. In the future, if we need to add enemies to the system, we simply just create classes for them and extends from Enemy abstract class.

This satisfied the SOLID principle, **Open-Closed Principle**. Enemy is the abstract class closed for modification, special characteristics for enemy should only present in their own classes.

d) **Bowser**

For Assignment 3, we have a new enemy called Bowser. Instead of implementing it in the Enemy class, we extend Bowser class from Enemy class and implement its own characteristics in the class. So that, we do not have to repeat the same code like adding wander and attack behaviour to the Bowser class, as it has already been added in Enemy class. Thus, we satisfied the principle **Don't Repeat Yourself**. For example, if we have other kind of Bowser with different characteristics, we can make Bowser an abstract class, then the other Bowsers extends from it. So that we do not have to repeat the same code again and again.

e) **Princess Peach**

Princess Peach extends from Actor class, she cannot be moved, attacked or attack. It will interact will Player only if the Player has the key to unlock her cuff.

f) **Piranha Plant**

Piranha Plant extends from Enemy class that will only spawn at the second turn. Once it was killed, Player can jumps on it to teleport to the second map.

g) **DestroyAction**

For Assignment 3, there is nothing change to this action. This is a new class called DestroyAction is newly created in Assignment 2, which extends from the Action in the engine code. In Assignment 2, this action is use to destroy Koopa's shell when Koopa is in dormant state and drop SuperMushroom at that specific location. This action will have its own option for the user to choose to destroy Koopa's shell if the player has wrench in their inventory. Then, it will display specific result to the console.

This follows the **Single Responsibility Principle**, which define as a class should have only one job. In this case, DestroyAction only used for the player to destroy the enemies in certain state that they cannot be attacked but can be destroyed. AttackAction is specifically used for actor to attack target, while DestoryAction is used for actor to destroy target in certain state that they cannot be attacked but can be destroyed to turn into different state.

h) **RescueAction**

For Assignment 3, we created a new RescueAction extends from Action. This action is mainly to interact with Princess Peach for now. When the Player have the key after killing Bowser, it can interact with Princess Peach and rescue her to end the game. This satisfied the principle, **Single Responsibility Principle,** which define as a class should have only one job. So, instead of doing everything in one Action class, we create an RescueAction that only focus on Player and Princess Peach.

REQ 3: Magical Fountain (With Optional Challenges)

a) RefillAction

RefillAction is an extension from Action abstract class, which allows the player to refill water from the fountain. When the player executes this action, it will first fill the water into the bottle and subtract 1 water from the fountain. RefillAction satisfies the **Single Responsibility Principle**, because it only focuses on the process of player refilling water from the fountain. This is to separate the action of refilling water from other actions, which have their own implementations.

b) ObtainBottleAction

ObtainBottleAction is an extension from Action abstract class, which allows the player to obtain bottle from the Toad. When the player executes this action, it will add the bottle into the player's inventory. ObtainBottleAction satisfies the **Single Responsibility Principle**, because it only focuses on the process of player obtaining bottle from the Toad. This is to separate the action of obtaining bottle from other actions, which have their own implementations.

c) ConsumeAction

ConsumeAction is an extension from Action abstract class, which allows an actor to consume item that implements Consumable interface. When the actor executes this action, the consumed item will give the actor some effects by adding capabilities to them. ConsumeAction satisfies the **Single Responsibility Principle**, because it only focuses on the process of consuming item that is consumable. This is to separate the action of consuming consumable item from other actions, which have their own implementations.

d) ConsumeBehaviour

ConsumeBehaviour provides an actor a consuming behavioural property depending on the ground where the actor steps on such as fountain. This behaviour allows the actor to consume item that implements Consumable interface such as Water. ConsumeBehaviour also satisfies the **Single Responsibility Principle**, because it only provides an actor the specific consuming behaviour and not other behaviours, which have their own implementations.

e)

- **Fountain**
  Fountain is an abstract class that is an extension from the Ground abstract class.

- **PowerFountain**
  PowerFountain is an extension from the Fountain abstract class. This class is responsible for allowing the player to refill PowerWater.

- **HealthFountain**
  HealthFountain is an extension from the Fountain abstract class. This class is responsible for allowing the player to refill HealingWater.

  Fountain satisfies the **Open-Closed Principle** as the functionality of Fountain can be extended without modifying the Fountain abstract class. This means that the extended class is able to use the inherited methods while providing its own implementations as in PowerFountain and HealthFountain. For example, PowerFountain which is an extension from the Fountain abstract class is able to provide PowerWater for the player to refill.

f)

- **Water**
  Water is an abstract class that is an extension from the Item abstract class.

- **PowerWater**
  PowerWater is an extension from the Water abstract class. This class is able to increase the drinker's base attack damage by 15 when this water is consumed.

- **HealingWater**
  HealingWater is an extension from the Water abstract class. This class is able to increase the drinker's hit points by 50 hit points when this water is consumed.

  Water satisfies the **Open-Closed Principle** as the functionality of Water can be extended without modifying the Water abstract class. This means that the extended class is able to use the inherited methods while providing its own implementations as in PowerWater and HealingWater. For example, HealingWater which is an extension from the Water abstract class is able to heal the drinker's hit points by 50 hit points.

g) Bottle

Bottle is an extension from the Item abstract class. Bottle satisfies the **Single Responsibility Principle** as it only holds Water and not other items. This is to separate the Water storage from other storages such as the actor's inventory, which have their own implementations.

REQ 4:

Title: Blink action + Patrol behaviour

Description: When the player steps on BlinkingTower, the player is able to blink to a location that is several steps (predefined) away within one turn. There is also another Actor called Luigi who exhibits a patrol behaviour that moves at the set path, back and forth.

Classes created:

- BlinkAction: an Action used by the player to blink to a location that is several steps (predefined) away within one turn.
- BlinkingTower: a HighGround where the player can jump onto to blink to a location that is several steps (predefined) away.
- Luigi: an Actor who will patrol at the set path, back and forth.

- PatrolBehaviour: a Behaviour that enables an Actor to patrol at the set path, back and forth.

Explanation why it adheres to SOLID principles:

- BlinkAction is an extension from Action abstract class, which allows the player to blink to a location that is several steps (predefined) away within one turn. BlinkAction satisfies the **Single Responsibility Principle**, because it only focuses on the blinking process of the player from one location to another. This is to separate the action of blinking from other actions, which have their own implementations.
- Luigi is an actor who will patrol at the set path, back and forth. Since the characteristics of Luigi is so much different with other actors (e.g. Toad, PrincessPeach, etc.), a new Luigi class is created and extends from Actor class as the general characteristics of actor is defined in the Actor abstract class. Luigi class satisfies the **Open-Closed Principle** as Actor is an abstract class closed for modification and the specific characteristics for Luigi only appear in Luigi class. This means that we can extend the implementations and characteristics of Luigi without modifying the Actor abstract class.
- PatrolBehaviour provides an actor a patrolling behavioural property, which allows it to move at the set path, back and forth. PatrolBehaviour also satisfies the **Single Responsibility Principle**, because it only provides an actor the specific patrolling behaviour and not other behaviours, which have their own implementations.
- BlinkingTower is a HighGround where the player can jump onto to blink to a location that is several steps away. Since BlinkingTower has this extra functionality of blink, apart from the original functionality of jump in the HighGround abstract class where it extends from. Therefore, BlinkingTower satisfies the **Open-Closed Principle** as the functionalities of BlinkingTower have been extended without modifying the High Ground abstract class.

Requirements	Features (HOW) / Your Approach / Answer
Must use at least two (2) classes from the engine package	We use three classes from the engine package: Action, Ground and Actor. BlinkAction class extends Action, BlinkingTower class extends HighGround which extends Ground and Luigi class extends Actor.
Must use/re-use at least one (1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Existing features used: • Jump feature (assignment 2): The player needs to jump onto the BlinkingTower before he sees the action to blink to a location that is several steps away. • Reset game (assignment 2): When the game is reset, Luigi will move back to its initial position in the next round.
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	Existing abstract class used: • HighGround: extended by BlinkingTower class so that the player will need to jump onto it

	before seeing the action of blinking to a location that is several steps away. Existing interfaces used: • Resettable: Implemented by Luigi class and override resetInstance method to provide RESET status to Luigi, so that Luigi will move back to its initial position in the next round when the game is reset. • Behaviour: Implemented by PatrolBehaviour class and override getAction method to return MoveActorAction, which is used by Luigi to patrol at the set path, back and forth.
Must use existing or create new capabilities	Existing capability used: • RESET: When the game is reset, this status will be given to Luigi, so that it will move back to its initial position in the next round.

REQ 5:

Title: Yoshi as adventure partner

Description: When the game starts, Yoshi who is beside the player will follow the player wherever the player goes. When the player takes damage from the enemies, Yoshi will also heal the player with a constant hitpoints of 5.

Classes created:

- Yoshi: an Actor who acts as an adventure partner for the player and will follow the player wherever he goes.
- HealBehaviour: a Behaviour used by Yoshi to get HealAction when the player takes damage from the enemies
- HealAction: an Action used by Yoshi to heal the player when the player takes damage from the enemies.

Explanation why it adheres to SOLID principles:

- HealAction is an extension from Action abstract class, which allows an actor to heal a target with a constant amount of hitpoints. HealAction satisfies the **Single Responsibility Principle**, because it only focuses on the healing process between an actor and a target. This is to separate the action of healing from other actions, which have their own implementations.
- HealBehaviour provides an actor a healing behavioural property, which allows it to heal a target with a constant amount of hitpoints. HealBehaviour also satisfies the **Single Responsibility Principle**, because it only provides an actor the specific healing behaviour and not other behaviours, which have their own implementations.

- Yoshi is an actor who will follow the player wherever he goes and heal the player when the player takes damage from the enemies. Since the characteristics of Yoshi is so much different with other actors (e.g. Toad, Luigi, etc.), a new Yoshi class is created and extends from Actor class as the general characteristics of actor is defined in the Actor abstract class. Yoshi class satisfies the **Open-Closed Principle** as Actor is an abstract class closed for modification and the specific characteristics for Yoshi only appear in Yoshi class. This means that we can extend the implementations and characteristics of Yoshi without modifying the Actor abstract class.

Requirements	Features (HOW) / Your Approach / Answer
Must use at least two (2) classes from the engine package	We use two classes from the engine package: Actor and Action. Yoshi class extends Actor and HealAction class extends Action.
Must use/re-use at least one (1) existing feature (either from assignment 2 and/or fixed requirements from assignment 3)	Reset game feature from assignment 2 is used. When the game is reset, Yoshi will move back to its initial position in the next round.
Must use existing or create new abstractions (e.g., abstract or interface, apart from the engine code)	Existing interfaces used: • Resettable: Implemented by Yoshi class and override resetInstance method to provide RESET status to Yoshi, so that Yoshi will move back to its initial position in the next round when the game is reset. • Behaviour: Implemented by HealBehaviour class and override getAction method to return HealAction, which is used by Yoshi to heal the player when the player takes damage from the enemies.
Must use existing or create new capabilities	Existing capabilities used: • RESET: When the game is reset, this status will be given to Yoshi, so that it will move back to its initial position in the next round. • HOSTILE_TO_ENEMY: This status is used to retrieve the player in the map, which will then be used to calculate the Manhattan distance between the player and Yoshi. Note: Yoshi will only follow the player when the player is nearby to Yoshi. • CAN_ENTER_HIGH_GROUND: This status is given to Yoshi, so that Yoshi is able to follow the player even if the player is currently on a high ground. New capability used: • INJURED: When the player takes damage from the enemies, the player will be given INJURED status. This status will be checked by Yoshi before healing the player.