Assignment 3

FIT2099

Zoe Low Pei Ee (31989985)
Chua Shin Herh (31902456)
Loh Zhun Guan (32245327)

Overview

This documentation includes the class diagrams, sequence diagrams and the design rationale of the extended system. It covers the explanation of the roles of the new classes as well as the interaction between the existing and extended classes. This documentation also discusses the design principles that the classes followed and utilized while implementing their functionalities.

Design Principles Utilized

[1] Avoid excessive use of literals

[2] Don't Repeat Yourself

[3] Single Responsibility Principle

[4] Open-closed Principle

[5] Liskov Substitution Principle

[6] Interface Segregation Principle

[7] Dependency Inversion Principle

Note: Highlighted are the sections that are modified/updated in Assignment 3

REQ 1 - Let it grow!

SpawnCapable

SpawnCapable is an interface which can be implemented by any object that has the ability to spawn something. An interface is used because we can adhere to the **Open-Closed Principle[4]**. For now, only the *Tree* class and its subclasses will implement this interface. Suppose we want to add new functionalities or capabilities to these classes. In that case, we can just implement another interface so we don't have to alter anything in this interface because the methods in this interface can only be used by spawnable objects, which fulfils the **Single Responsibility Principle[3]** and **Interface Segregation Principle[6]**.

We assume that trees might not be the only object that has the spawn capability hence we chose to implement an interface instead so we are able to maintain the flexibility of our code and it is also easier to keep track of the capabilities of the classes by implementing interfaces.

Tree

Tree is a class that extends the *Ground* class as it will inherit all the attributes and methods of the ground. It is used to represent the parent class of all the trees. *Tree* is an abstract class as we do not want to instantiate it. The *Tree* class is extended by several classes *Sprout*, *Sapling* and *Mature*. They all share a similar attribute which is the age where the tree will grow after a certain number of turns. This design is implemented based on the **Don't Repeat Yourself principle[2]** as the similar attributes are not repeated within each of the *Tree* subclasses which ensures the maintainability of our code.

Tree class implements the *SpawnCapable* interface as each of its subclasses will implement the drop() method to represent each of their unique spawning abilities. This makes it easier to maintain the code as we do not need to modify the parent class code so new or different functionality can just be added in the child class which adheres to the **Open-Closed principle[4]**.

Sprout

Sprout is a subclass of *Tree* because sprout is a type of tree in the game. As a subclass it will inherit from the parent class the constructors, methods with the same parameters and the return values**.** Thus, *Sprout* has to override the *drop()* method of the *SpawnCapable* interface as the *Tree* class implements the *SpawnCapable* interface. *Sprout* starts with the age 0 and in the *drop()* method it will implement its functionality which has a 10% chance of spawning a *Goomba*.

Sapling

Sapling is a subclass of *Tree* because *Sapling* is a type of tree in the game. Similar to the *Sprout* class, it will inherit the parent class constructors, methods with the same parameters and the return values**.** Thus, *Sapling also* has to override the *drop()* method of the *SpawnCapable* interface. *Sapling* starts with the age 10 and in the *drop()* method it will implement its functionality which has a 10% chance of dropping a coin with a value of $20.

Mature

Mature is a subclass of *Tree* because *Mature* is a type of tree in the game. Similar to *Sprout* and *Sapling*, *Mature* starts with the age of 20 and will override the *drop()* method where it has a 15% chance of spawning a *Koopa* every turn, spawn a new sprout in one of the surrounding fertile squares randomly every five turns and a 20% chance of withering and turning into dirt.

Wallet

Wallet is a class that represents the balance of the player. *Wallet* has a balance as a HashMap with *Actor* as key and Integer as a value. Balance was created as a hashmap because it would be easier to just map the players to their respective balance if there are more than one player in the game.

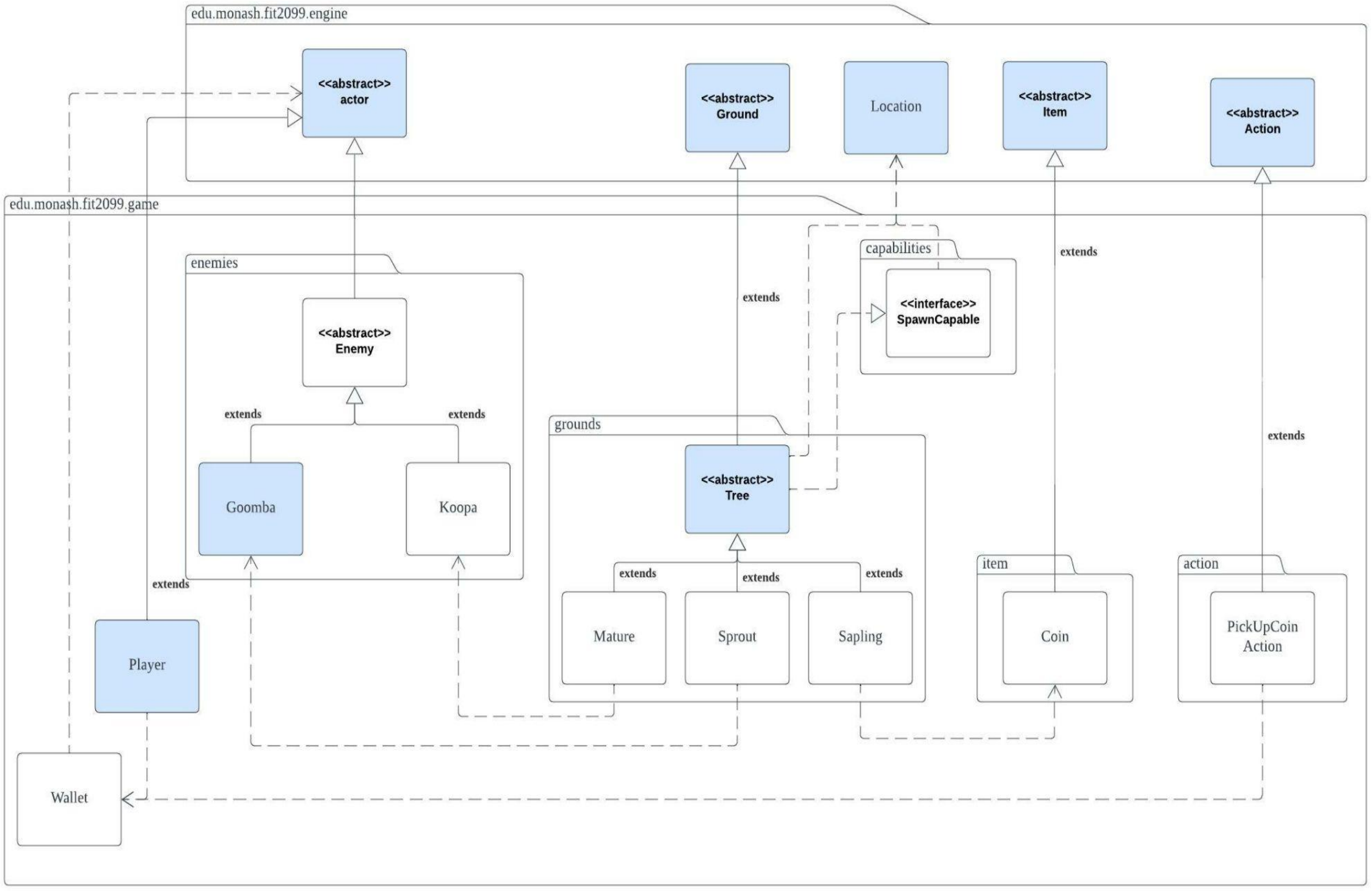
PickUpCoinAction

PickUpCoinAction is a subclass of *Action* as it is a type of action in the game. *PickUpCoinAction* is used to pick up the coins from the map and remove them. It adheres to the **Single-Responsibility Principle[3]** and **Don't Repeat Yourself principle[2]** as the class is only used to handle the pick up coin action.

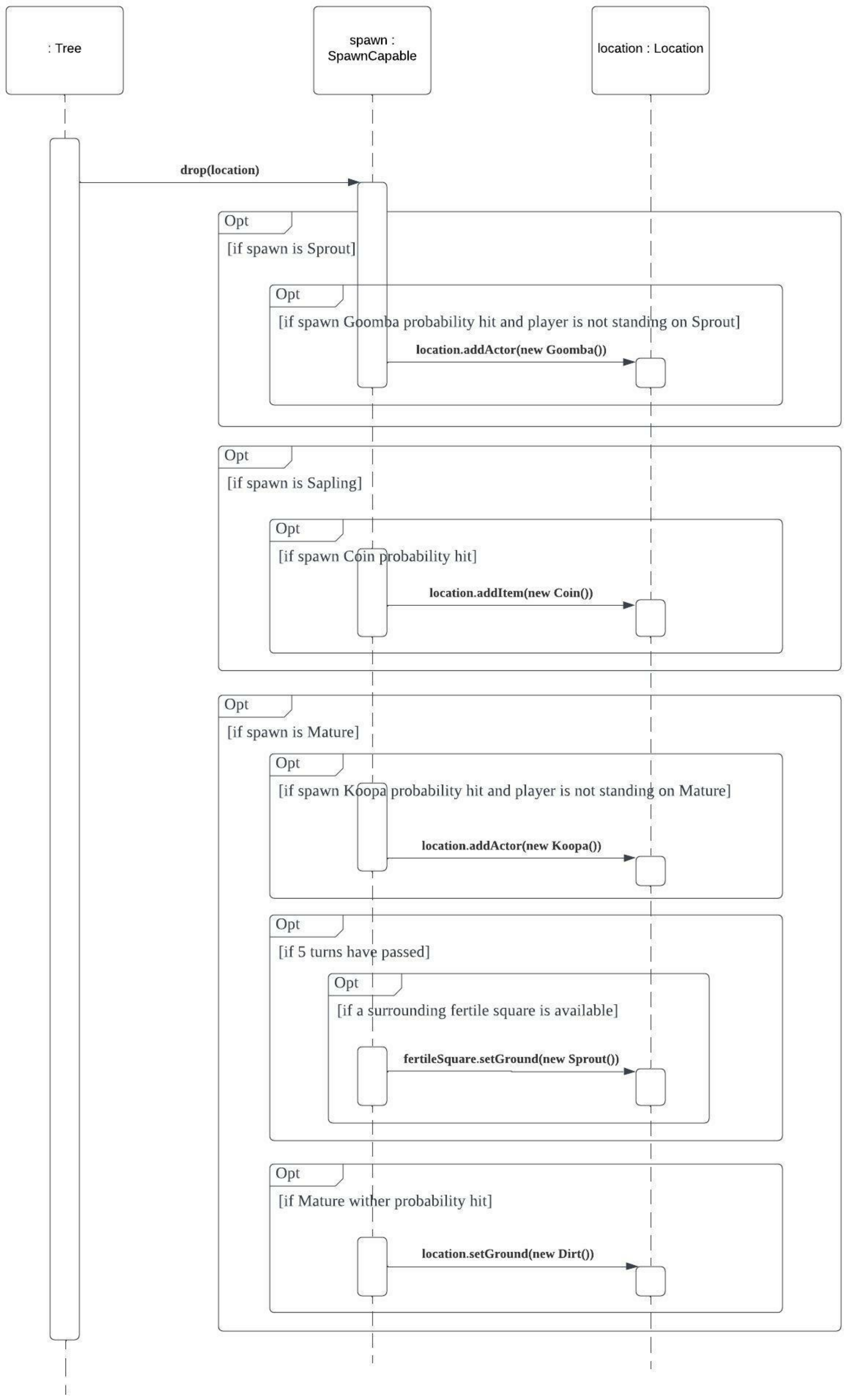
Coin

Coin is a class that represents the coin that is dropped from the *Sapling*. It creates a new action that will then be added to the allowable actions list that the player can perform when the player is on top of a coin.

Class Diagram



Sequence Diagram


: Tree
spawn : SpawnCapable
location : Location
drop(location)
Opt
[if spawn is Sprout]
Opt
[if spawn Goomba probability hit and player is not standing on Sprout]
location.addActor(new Goomba())
Opt
[if spawn is Sapling]
Opt
[if spawn Coin probability hit]
location.addItem(new Coin())
Opt
[if spawn is Mature]
Opt
[if spawn Koopa probability hit and player is not standing on Mature]
location.addActor(new Koopa())
Opt
[if 5 turns have passed]
Opt
[if a surrounding fertile square is available]
fertileSquare.setGround(new Sprout())
Opt
[if Mature wither probability hit]
location.setGround(new Dirt())

REQ 2 - Jump Up, Super Star!

JumpCapable

JumpCapable is an interface which can be implemented by the high grounds. An interface is used because we can adhere to the **Open-Closed Principle[4]**. The high ground classes will implement this interface and if we want to add more features to these classes, we don't need to change anything inside the interface, as the methods in the interface are standard across all high grounds.

This interface also lets us adhere to the **Dependency Inversion Principle[7]**. Instead of directly having a dependency between *JumpAction* and the high ground classes, which can cause problems if we add more high ground classes in the future, we have this interface between *JumpAction* and the high ground classes, so changes in one class wouldn't affect the others.

This interface also adheres to the **Don't Repeat Yourself[2]** principle. We can have a default method in the interface which performs the jump for all the high ground classes rather than having the jump method in all the high ground classes, which reduces duplicate code.

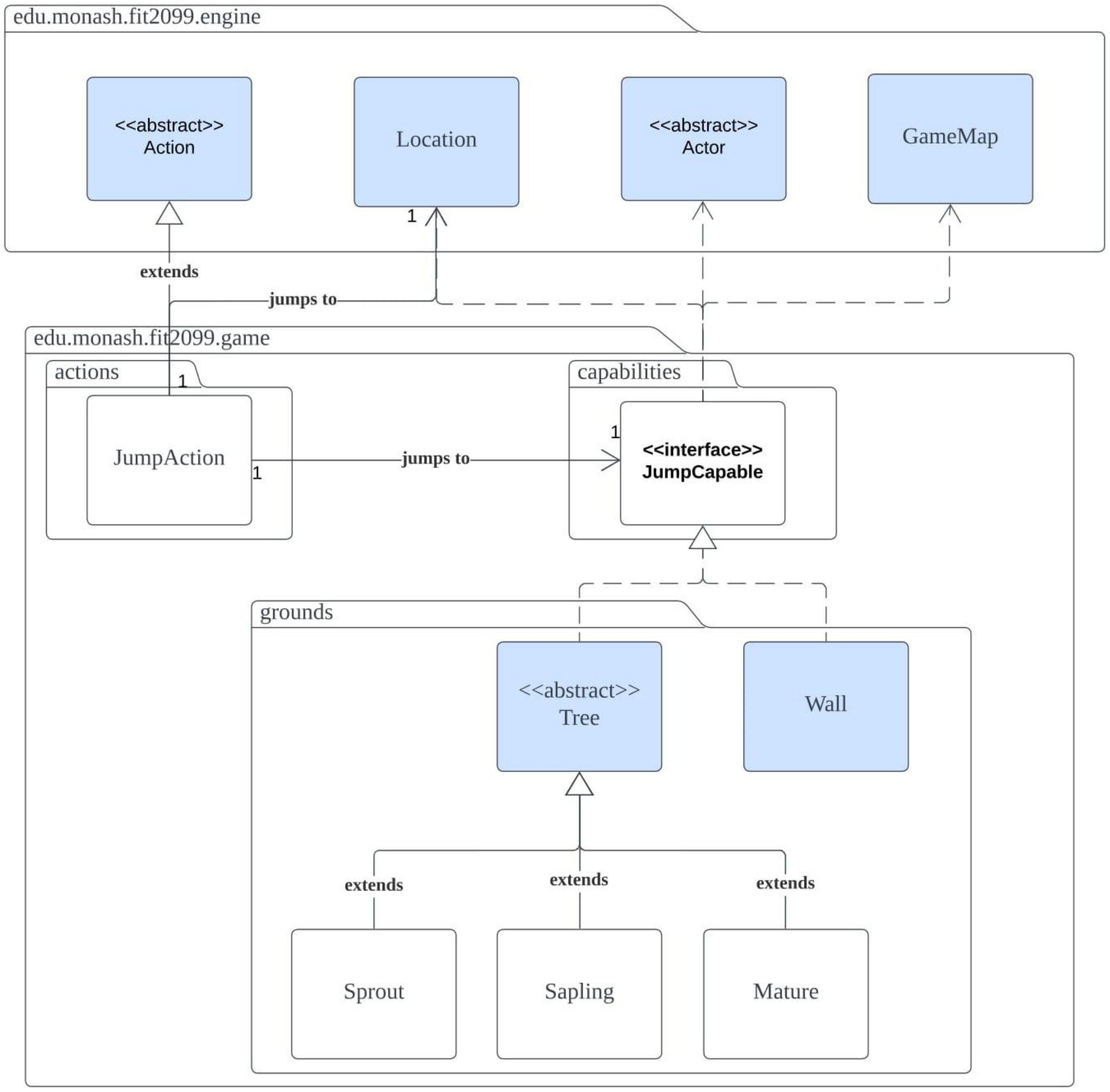
JumpAction

JumpAction is a class used by Actors to jump to a high ground. *JumpAction* extends the *Action* abstract class since jumping is an action, and to adhere to the **Single Responsibility Principle[3]** where this class only handles the jumping actions.

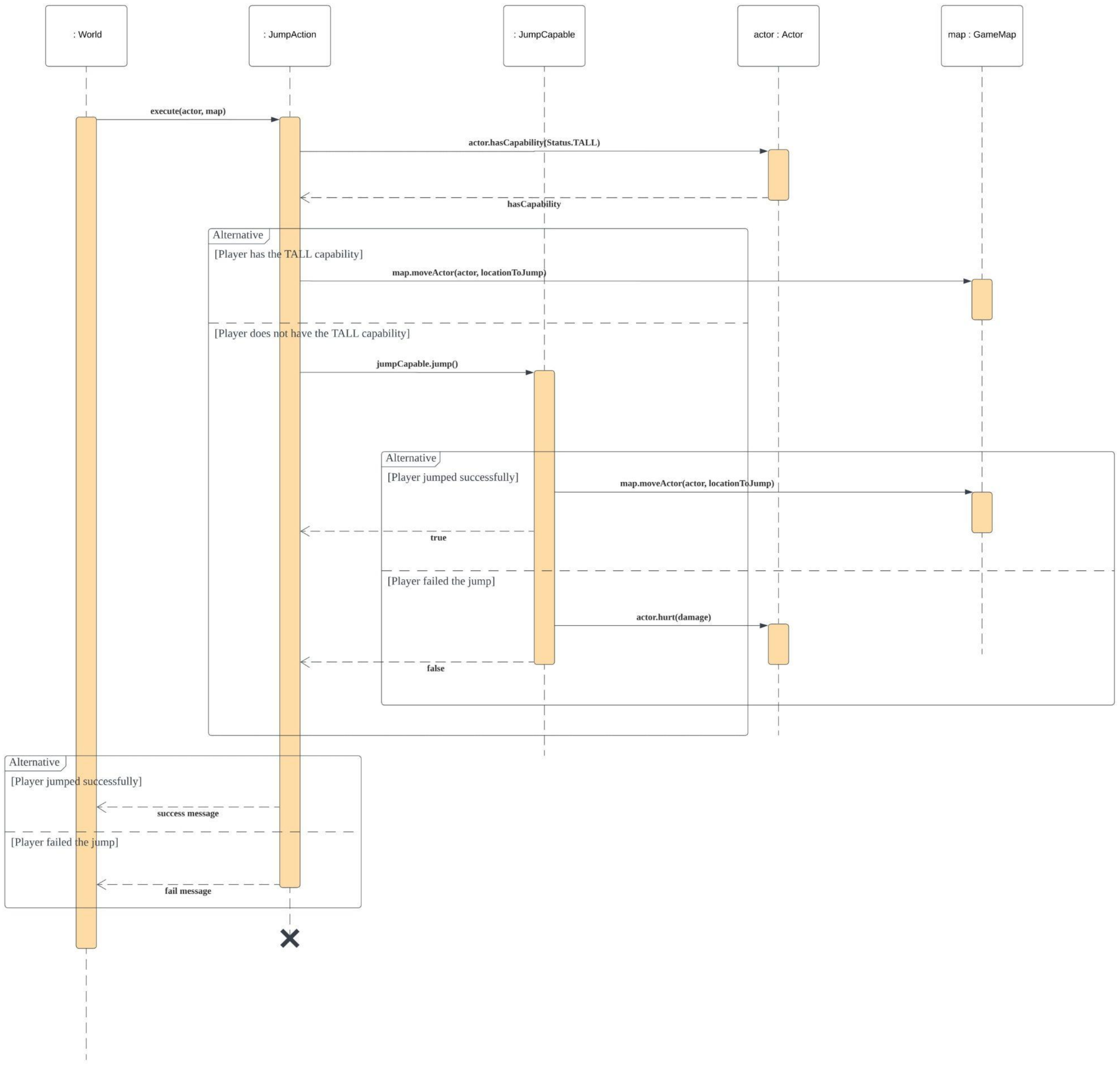
JumpAction has an association to the *Location* class because it needs the location of the high ground to jump to. *JumpAction* has an association to *JumpCapable* because the *JumpAction* can execute the jump method in the *JumpCapable* interface, and it needs a *JumpCapable* instance to know which *JumpCapable* object to execute the jump.

JumpAction also has a dependency on the *Status* enumeration which is not shown in the UML Class Diagram. This dependency is because when the player consumes a Super Mushroom, the player will jump to any high ground with a 100% success rate, and instead of using a magic string to determine whether the player has consumed a Super Mushroom, we can use the *Status*.TALL constant, which will follow the **Avoid Excessive Use of Literals[1]** principle.

Class Diagram



Sequence Diagram



REQ 3 - Enemies

Enemy

Enemy abstract class extends from *Actor* class as it will inherit the attributes and methods of an *Actor*. It represents the parent class of all enemies and we do not want to instantiate an *Enemy* object so we make it an abstract class. The *Enemy* class is extended by several classes which are *Goomba* and *Koopa*. They share similar attributes as all enemies are able to attack the player when the player stands in the enemy's surroundings and will follow the player once it is engaged in a fight. This design is implemented based on the **Don't Repeat Yourself principle[2]** as the similar attributes and methods would not be repeated within each *Enemy* subclass thus making maintenance of the code easier.

Enemy also stores a *behaviours* HashMap with its priority as key and behavior as value so that each enemy can have its own behavior. Hence if any enemy has a specific new behaviour then it is able to utilize the existing HashMap and just put the new behavior into it. This HashMap will be set as private access modifier and final to create a tighter control and avoid unnecessary or unintentional modification on the HashMap, so a separate *addBehaviour* method with a modifier of protected is created to allow only sub-classes that inherits from this Enemy class to be able to add the behaviours into the HashMap. *Enemy will* add a *FollowBehaviour* to the list of *behaviours* (attribute of the *Enemy* class) as it will follow the *Player* once they are engaged in a fight and also an *AttackBehaviour* to attack the player. This maintains the flexibility of the code as it does not need to modify the parent class if new behaviours or implementations are added, thus fulfilling the **Open-Closed principle[4]**. As each enemy has more than one behaviours by default and might have other different behaviours, so the multiplicity of Enemy to *behaviours* HashMap is one to many.

Goomba

Goomba is a subclass of the *Enemy* abstract class because *Goomba* is a type of enemy in the game. *Goomba* starts with 20HP and it has a 10% chance to be removed from the map. *Goomba* will overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a kick and 10 damage, the hit rate is the same as player so we can just overrides this method instead of creating duplicate methods.

As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed whenever we add a behaviour which adheres to the **Open-Closed Principle[4]**. Next, the *Goomba* would implement a *SuicideAction* because the Goomba has the ability to suicide during the game thus it overrides the *playTurn* method from the *Actor* class to have a 10% chance of calling the *SuicideAction*. This approach fulfills both the **Single Responsibility Principle[3]** and the **Don't Repeat Yourself Principle[2].**

Koopa

Koopa is a subclass of the *Enemy* abstract class because *Koopa* is a type of enemy in the game. *Koopa* starts with 100HP and will go into dormant state (D) when it is defeated by the player. *Koopa* will overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a punch and 30 damage, the hit rate is the same as player and *Goomba* so we don't have to extend *IntrinsicWeapon* class or create other methods to modify the hit rate value.

Koopa would override the *playTurn* method as it will go to a dormant state and stay on the ground when it is not conscious and will then be removed from the map and drop a Super Mushroom once its shell is destroyed by the player by a Wrench.

Koopa will execute the *playTurn* method of the parent class (Enemy abstract class) through the "super" keyword. The default implementation of *playTurn* is to execute the *WanderBehaviour*, *FollowBehaviour* and *AttackBehaviour* which are the default behaviour of the enemies based on the *Status* of the *Player* and *Enemy*. As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed. Hence we won't need to repeat the same chunk of code which fulfills the **Don't Repeat Yourself principle[2]** as repeating code is avoided.

Wrench

Wrench will be extending from *WeaponItem* abstract class as it is a type of *Weapon* and also an *Item*. As a subclass, it will inherit and use the base class constructors and it can call the methods of the *WeaponItem* class when needed without implementing duplication methods which fulfills the **Don't Repeat Yourself principle[2].**

AttackKoopaAction

AttackKoopaAction will be an extension from *Action* class, in which the constructors and methods will have the same signatures. It stores the attributes of an *Actor* target (which is the one to be attacked -- *Koopa*) and a String of direction (to identify the direction of incoming attack). The random number attribute is also stored as a class level attribute, it is to generate the probability of successful attacks chances.

AttackKoopaAction fulfills the **Single-Responsibility principle[3]** as it will only be called while attacking *Koopa*. It will do nothing else such as the action of "checking if the target is *Koopa*" or "the *Koopa* attack the *Player*" are not within the class as its main responsibility is just to attack and defeat the *Koopa* (destroy its shell) but it will ensure that the weapon used by the *Player* is a *Wrench* as it is the only weapon that can destroy the shell.

AttackKoopaAction inherits *Action* class and only overrides or add methods that will have different functioning logic so that we can avoid code duplication as both classes shared most of the logic which is to attack the target, hence adhering to the principle **Don't Repeat Yourself [2].** The main difference between two classes is the *execute* method, where the Player has to destroy *Koopa*'s shell with a *Wrench* which will then drop a *Super Mushroom* when the *Koopa* is defeated. As it is a special condition only for *Koopa*, hence the design decision made to create a new subclass is also to support different execution of attack separately rather than creating multiple check conditions and dependencies (i.e. Super Mushroom) in the base class.

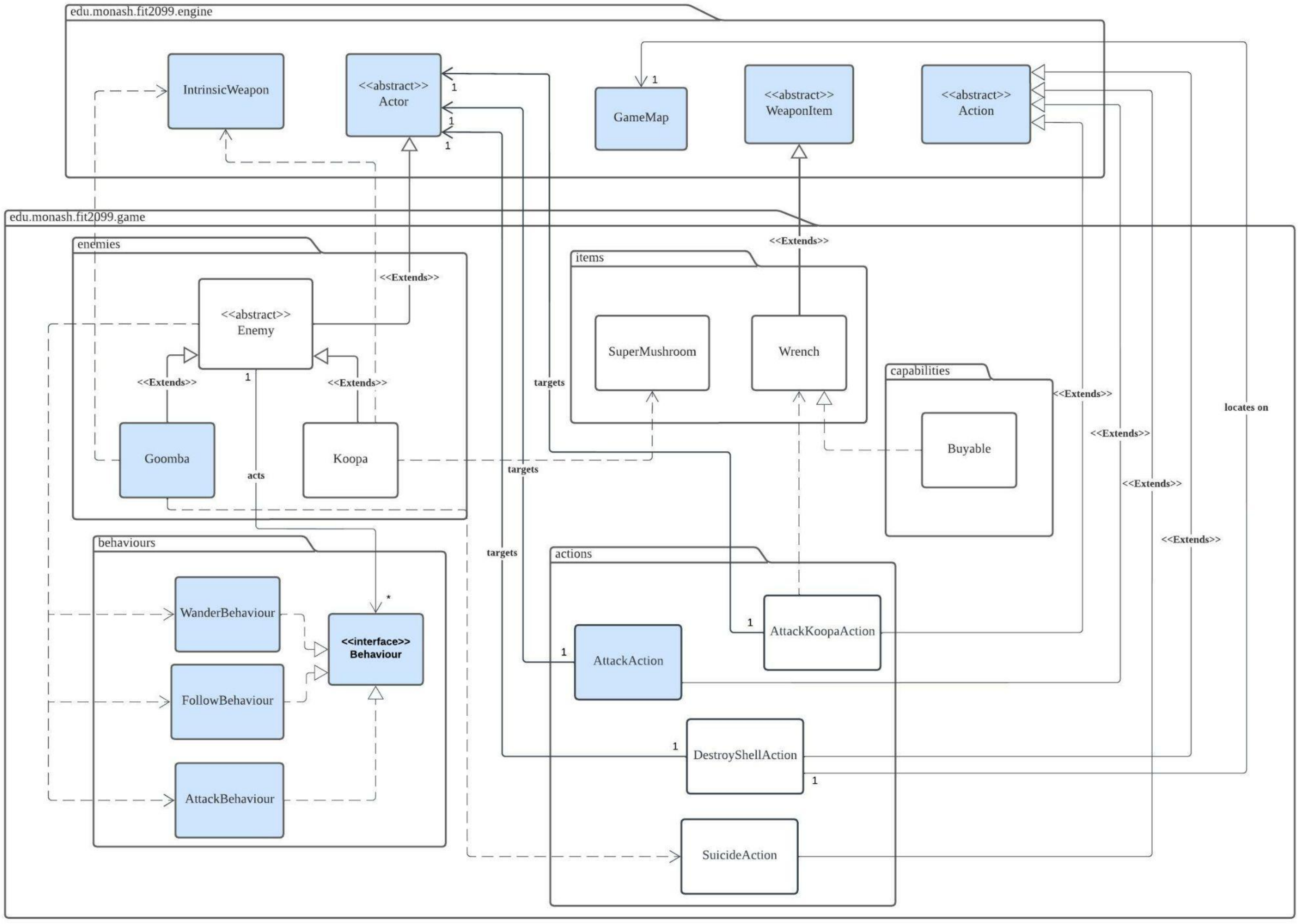
SuicideAction

SuicideAction will be extending from the *Action* abstract class. *SuicideAction* is used to remove *Goomba* from the map and its main implementation is just calling the *removeActor*() function and printing out a String message indicating the *Goomba* has suicide. It is a general suicide action so when there are other actors who will perform suicide, we can avoid code duplication which adheres to the **Don't Repeat Yourself principle [2].**

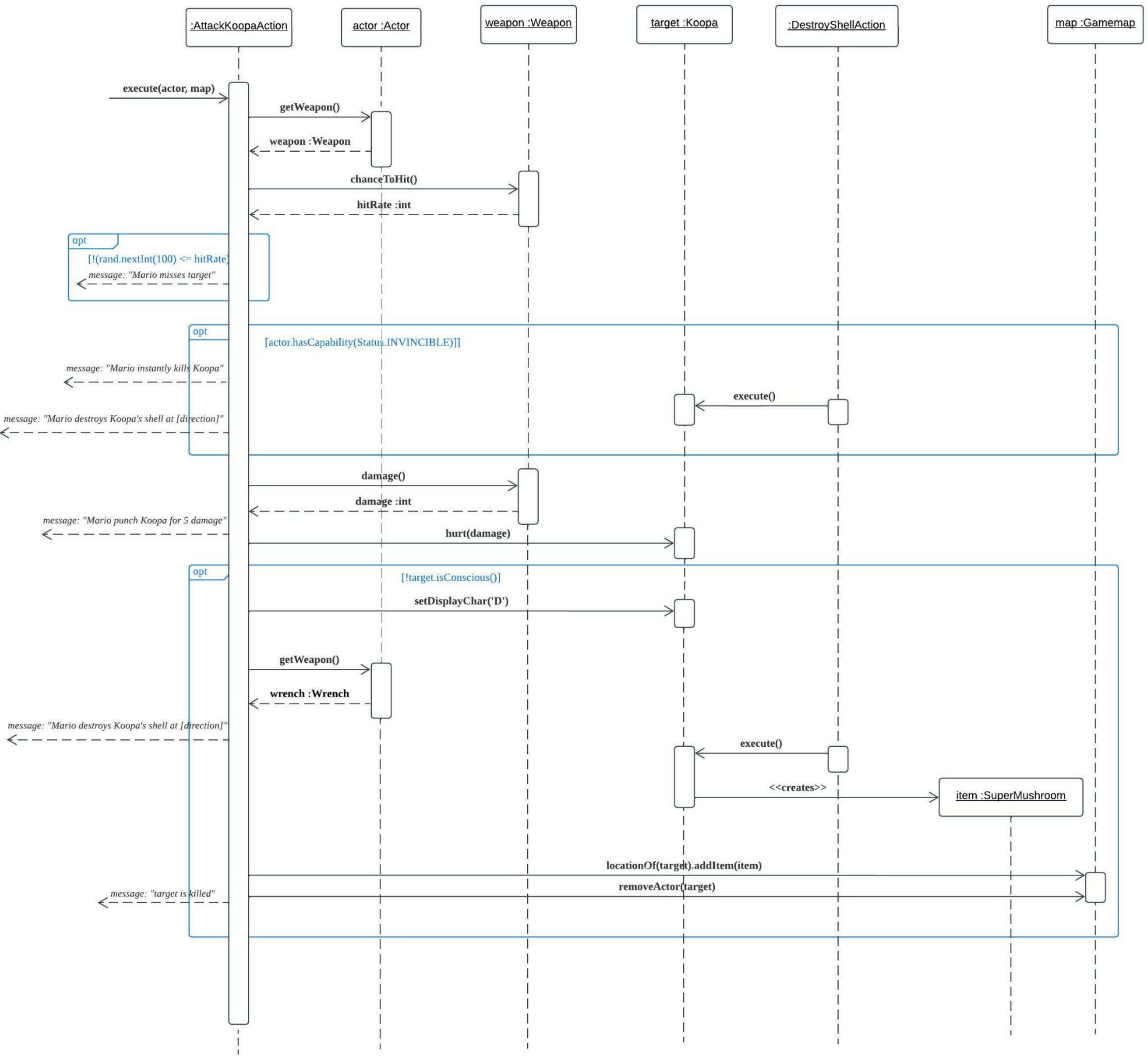
DestroyShellAction

DestroyShellAction will be extending from the *Action* abstract class. *DestroyShellAction* is used to destroy the target's shell when defeated and drops a Super Mushroom only when the player has a Wrench. We make it as a general destroy shell action so when there is another enemy whose shell can be destroyed, we don't need to repeat the same code again which adheres to the **Don't Repeat Yourself principle [2].**

Class Diagram



Sequence Diagram


:AttackKoopaAction
actor :Actor
weapon :Weapon
target :Koopa
:DestroyShellAction
map :Gamemap
execute(actor, map)
getWeapon()
weapon :Weapon
chanceToHit()
hitRate :int
opt
[!(rand.nextInt(100) <= hitRate)]
message: "Mario misses target"
opt
[actor.hasCapability(Status.INVINCIBLE)]]
message: "Mario instantly kills Koopa"
execute()
message: "Mario destroys Koopa's shell at [direction]"
damage()
damage :int
message: "Mario punch Koopa for 5 damage"
hurt(damage)
opt
[!target.isConscious()]
setDisplayChar('D')
getWeapon()
wrench :Wrench
message: "Mario destroys Koopa's shell at [direction]"
execute()
<<creates>>
item :SuperMushroom
locationOf(target).addItem(item)
removeActor(target)
message: "target is killed"

REQ 4 - Magical Items

MagicalItem

MagicalItem will be extending from *Item* abstract class as it is a type of *Item*. It represents the parent class of all magical items (PowerStar and SuperMushroom) and we do not want it to be instantiated by other classes. We make it as an abstract class to ensure we follow the **Dependency Inversion Principle[7]** where a concrete class should not depend on another concrete class and should depend on abstractions instead. A *MagicalItem* inherits the attributes and methods of the *Item* class. It has an abstract method *consumedBy()* which the subclasses have to provide the implementation of this method with this same signature. This design decision is made as each magical item will have different effects on the *Player* hence require different implementation of that method. Our design assumes that by default, *MagicalItem* is already on the same ground and the player is able to pick it up or drop it. As *MagicalItem* inherits *Item* abstract class, the extended class would only need to use the existing attributes and methods through the "super" keyword which fulfills the **Don't Repeat Yourself (DRY) principle[2]**.

SuperMushroom

SuperMushroom will be extending from *MagicalItem* abstract class. It will override the *consumedBy*() method of its parent class which executes the special features after the Actor consumes it. It will change the status of the actor to Status.TALL, which is already provided in the *Status enumeration*, hence we can **avoid excessive use of literals[1]**.

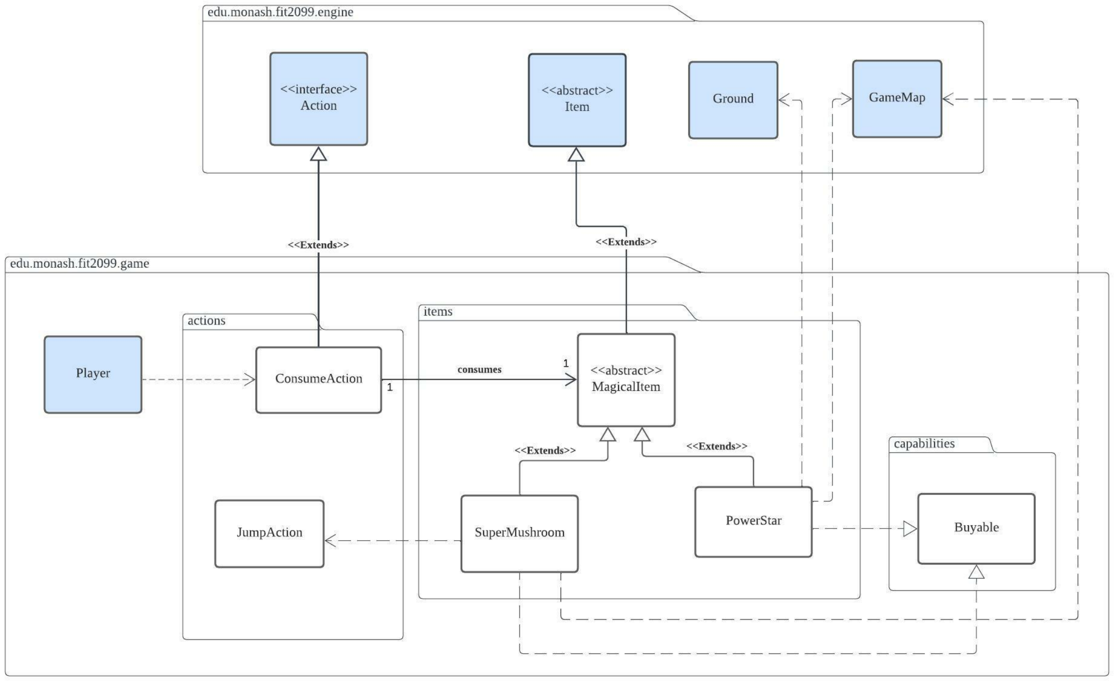
PowerStar

PowerStar will be extending from the *MagicalItem* abstract class. It will override the *consumedBy*() method of its parent class which executes the special features after the Actor consumes it. It will change the status of the actor to *Status*.INVINCIBLE, hence we can **avoid excessive use of literals[1]**. The difference between *PowerStar* and *SuperMushroom* is the dependency relationships of *Ground* class, *Dirt* class and *Coin* class as we will have to convert the higher grounds to dirt and drop a Coin after destroying the ground thus instantiating an instance of these classes. Also, *PowerStar* has a *tick*() method to keep track of the remaining turns of the effect as it will self-destruct after 10 turns.

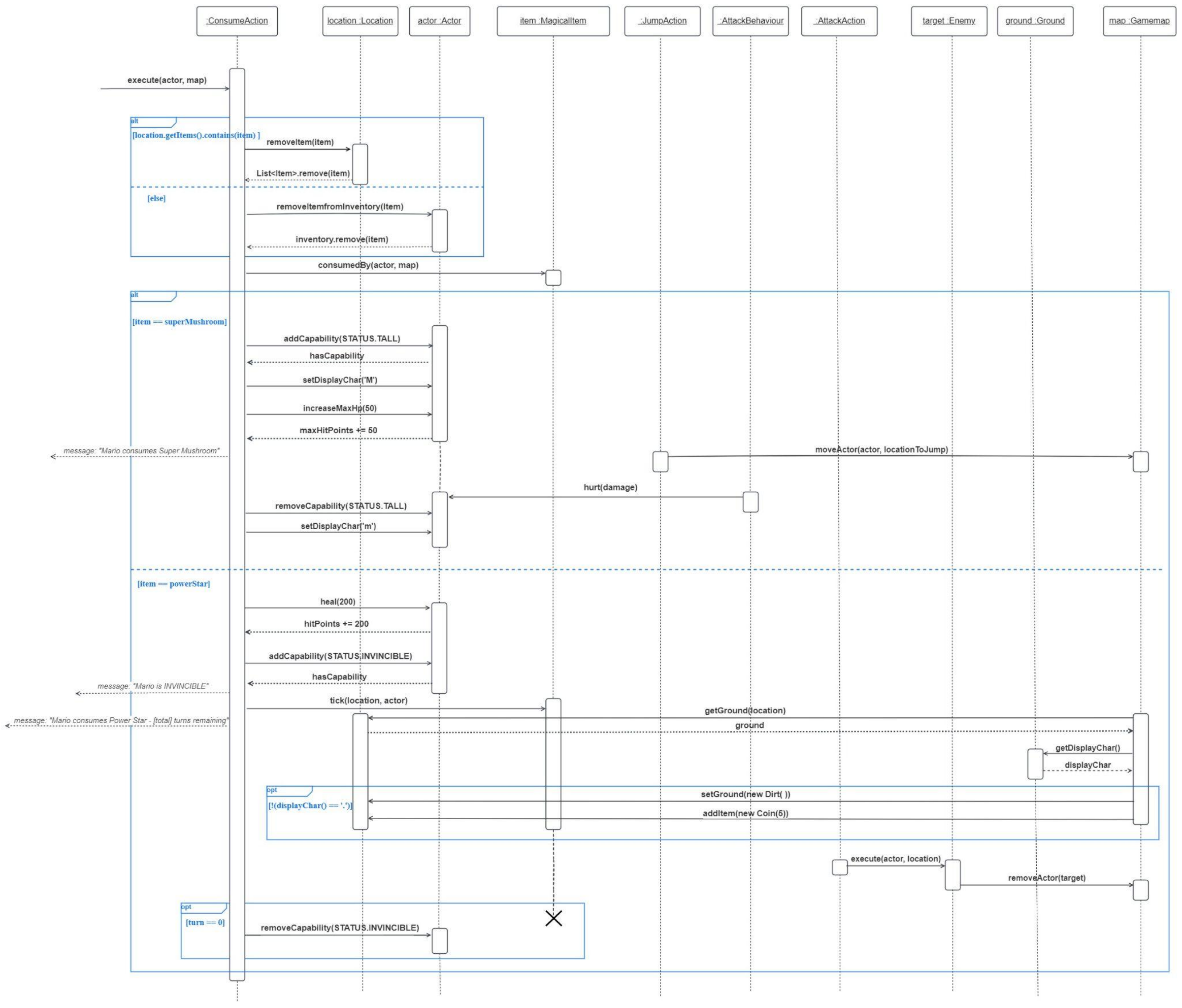
ConsumeAction

ConsumeAction will be a subclass of *Action* abstract class, it inherits to **minimize code duplication[2]**. *ConsumeAction* fulfills the **Single Responsibility principle[3]** because it will only remove the *MagicalItem* from the inventory or on the ground and consumes the *MagicalItem*. The *execute*() method is overridden, by checking whether the magical item is on the ground or in the Actor's inventory to remove it, followed by calling the consumedBy() method of the related subclass to consume the *MagicalItem*. The menu description is overwritten as well to output the message "[*Actor*] consumes [*MagicalItem*]".

Class Diagram



Sequence Diagram



REQ 5 - Trading

Buyable

Buyable is an interface which can be implemented by the buyable items. An interface is used because we can adhere to the **Open-Closed Principle[4]**. The buyable items will implement this interface and if we want to add more features to these classes, we don't need to change anything inside the interface, as the methods in the interface are standard across all buyable items.

This interface also lets us adhere to the **Dependency Inversion Principle[7]**. Instead of directly having a dependency between *BuyAction* and the buyable items, which can cause problems if we add more buyable items in the future, we have this interface between *BuyAction* and the buyable item classes, so changes in one class wouldn't affect the others.

This interface also adheres to the **Don't Repeat Yourself[2]** principle. We can have a default method in the interface which performs the buying for all the buyable items rather than having the buy method in all the buyable item classes, which reduces duplicate code.

Buyable has a dependency on the *Wallet* class as buying items would require access to the wallet balance of the player, and to deduct coins from the wallet's balance. *Buyable* has a dependency on the *Item* class because when the player buys something, the item will be added to the player's inventory, and adding an item to the player's inventory uses the item, but doesn't require an association to the item.

BuyAction

BuyAction is a class used by the player to buy items from *Toad*. *BuyAction* extends the *Action* abstract class because buying is an action, and to adhere to the **Single Responsibility Principle[3]** where this class only handles the buying actions.

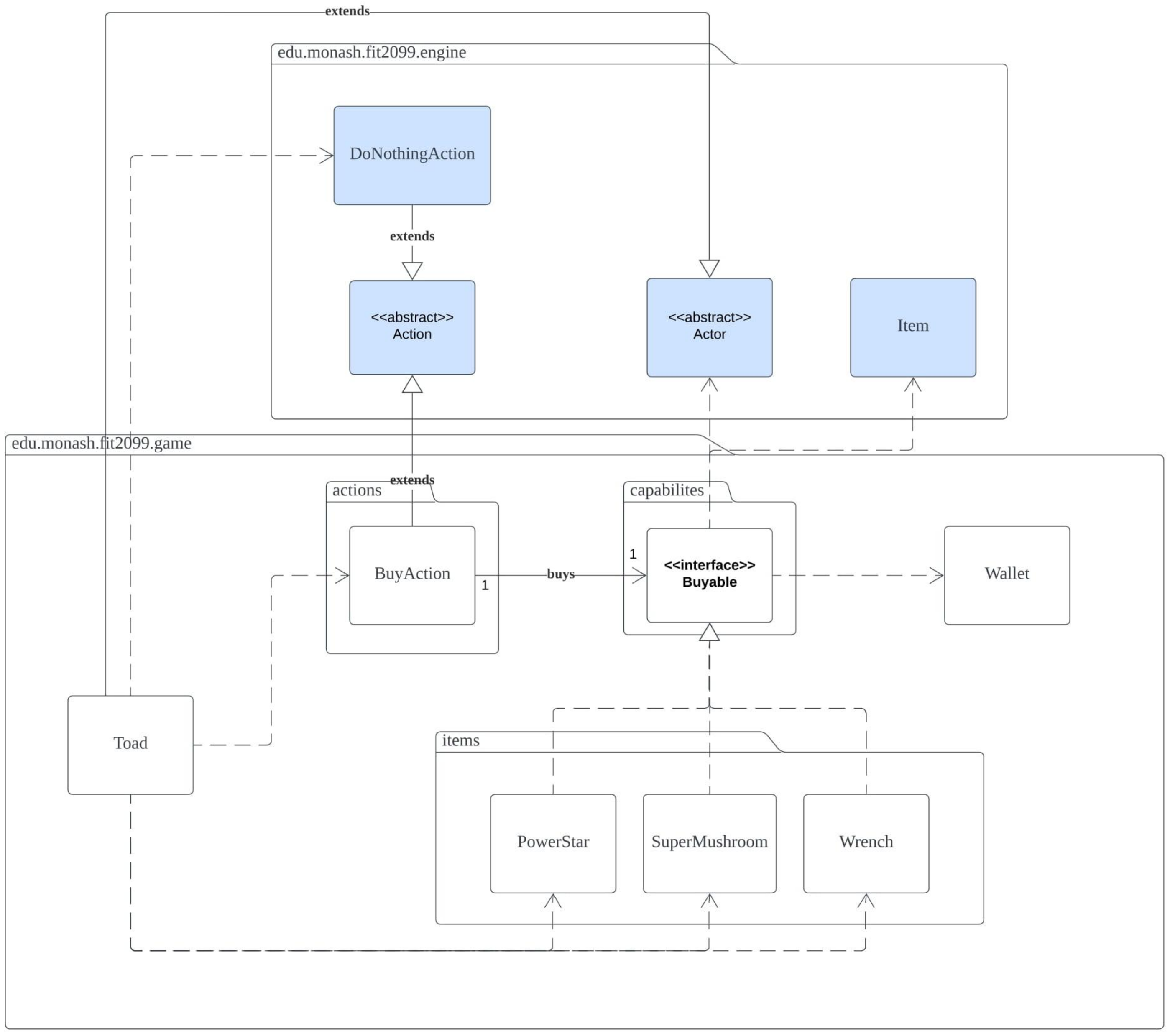
BuyAction has an association to *Buyable* because the *BuyAction* can execute the buy method in the *Buyable* interface, and it needs a *Buyable* instance to know which *Buyable* object to execute the buy process.

Toad

Toad is a class which will provide the player with items to buy. *Toad* extends the *Actor* class as *Toad* is an actor. *Toad* has dependencies to the classes *PowerStar*, *SuperMushroom* and *Wrench* because *Toad* uses new instances of these classes to sell these items to the player. *Toad* has a dependency on *BuyAction* because buying items from *Toad* are part of Toad's allowable actions.

Toad has a dependency on *DoNothingAction* because *Toad* will not do anything on every turn.

<u>Class Diagram</u>

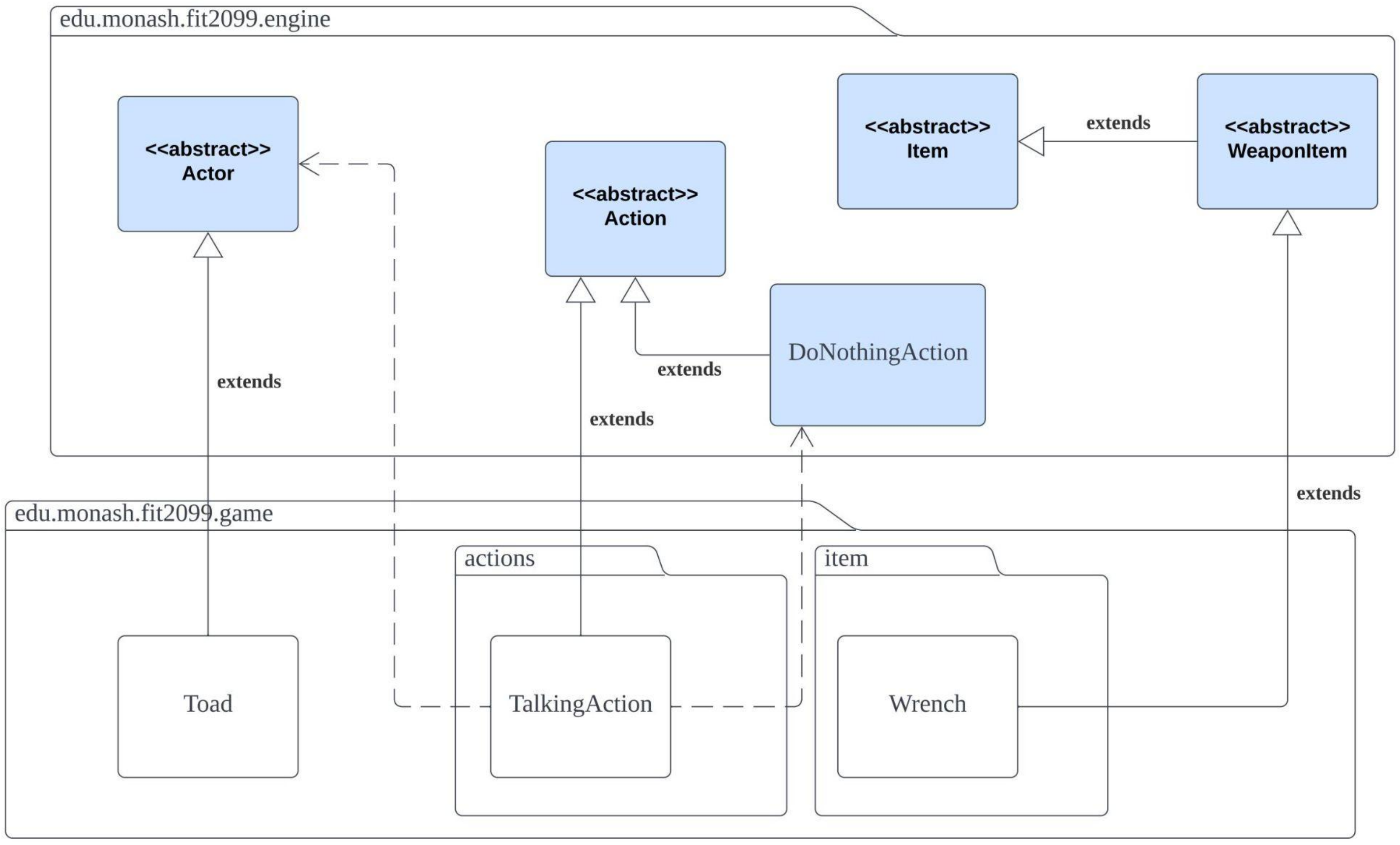


REQ 6 - Monologue

TalkingAction

TalkingAction is a subclass of *Action* used to represent the conversation between *Toad* and the player. *TalkingAction* adheres to the **Single Responsibility Principle[3]** as this class only handles the talking actions. *TalkingAction* also has a dependency on *Actor* as it needs to check if the actor, which is the player, has eaten a Power Star and has a wrench in the inventory to react with appropriate dialogue.

Class Diagram

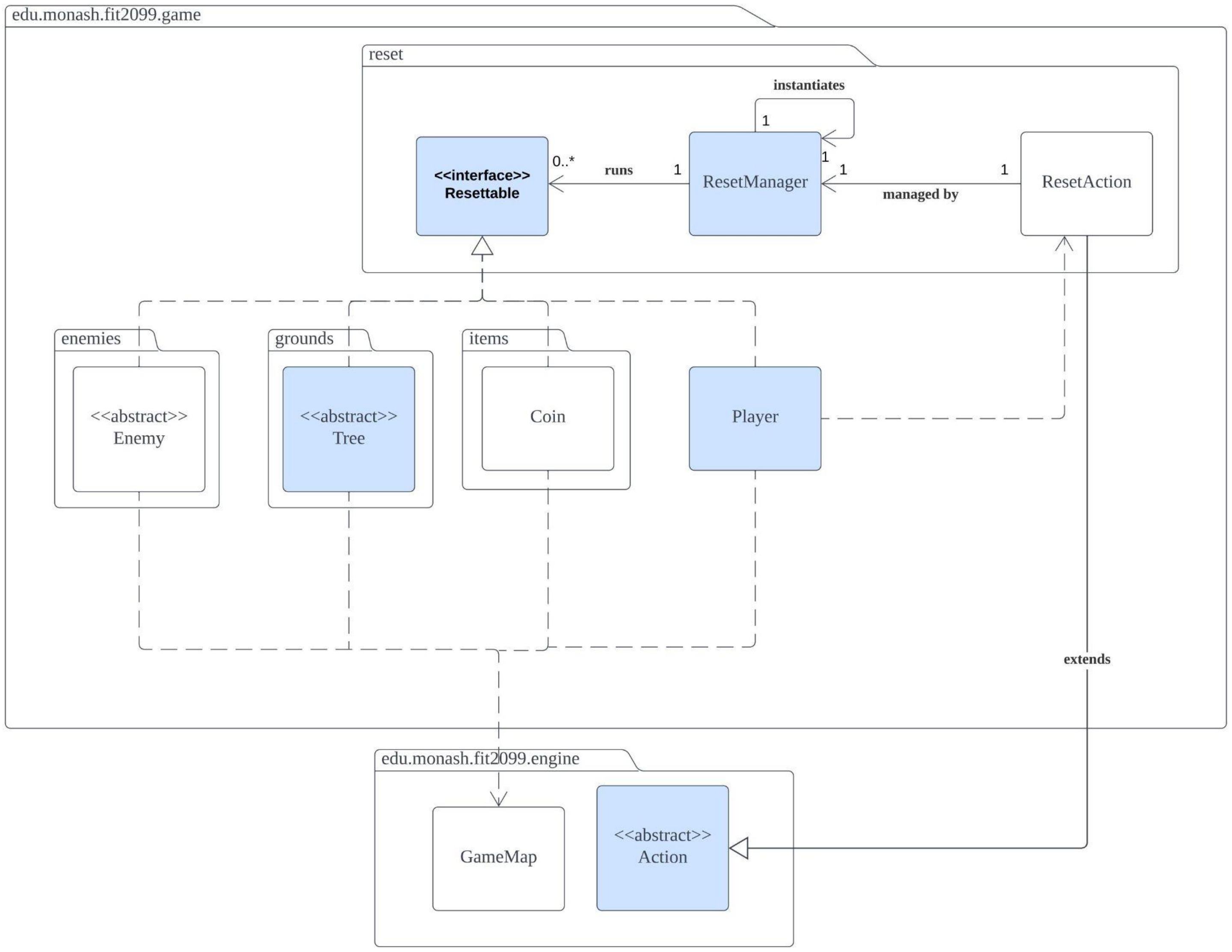


REQ 7 - Reset Game

ResetAction

ResetAction will be extended from the *Action* abstract class. In this class, we get the singleton *ResetManager* instance as the class attribute by calling the public static factory method *getInstance*() of the class *ResetManager, resulting* in a one-to-one association relationship between *ResetAction* and *ResetManager*. The main responsibility of this class is to execute the *run*() method of the *ResetManager* to clean the map and reset the game, which adheres to the **Single Responsibility Principle[3]**. Furthermore, by doing this way it will avoid using "instanceof" all over the place, to ensure that we are not violating the **Open Closed Principle[4]**. As the Player class will be the only class that can create *ResetAction* in the *playTurn*() method, hence it will have a dependency relationship with the *Player* class.

Class Diagram



Assignment 3

REQ 1 - Lava Zone

Lava

Lava is a class that will deal damage to the player if the player is standing on it. It extends the Ground class as it is a ground.

WarpPipe

WarpPipe is a class that will teleport a player to another map in the game. It extends the Ground class as it is a ground. It implements the JumpCapable interface as it can be jumped on, the Resettable interface as it can be resetted to spawn Piranha Plant again, and the TeleportCapable interface as it can teleport the player to another location. It has an association to the Location class because we need the location of the destination to teleport to.

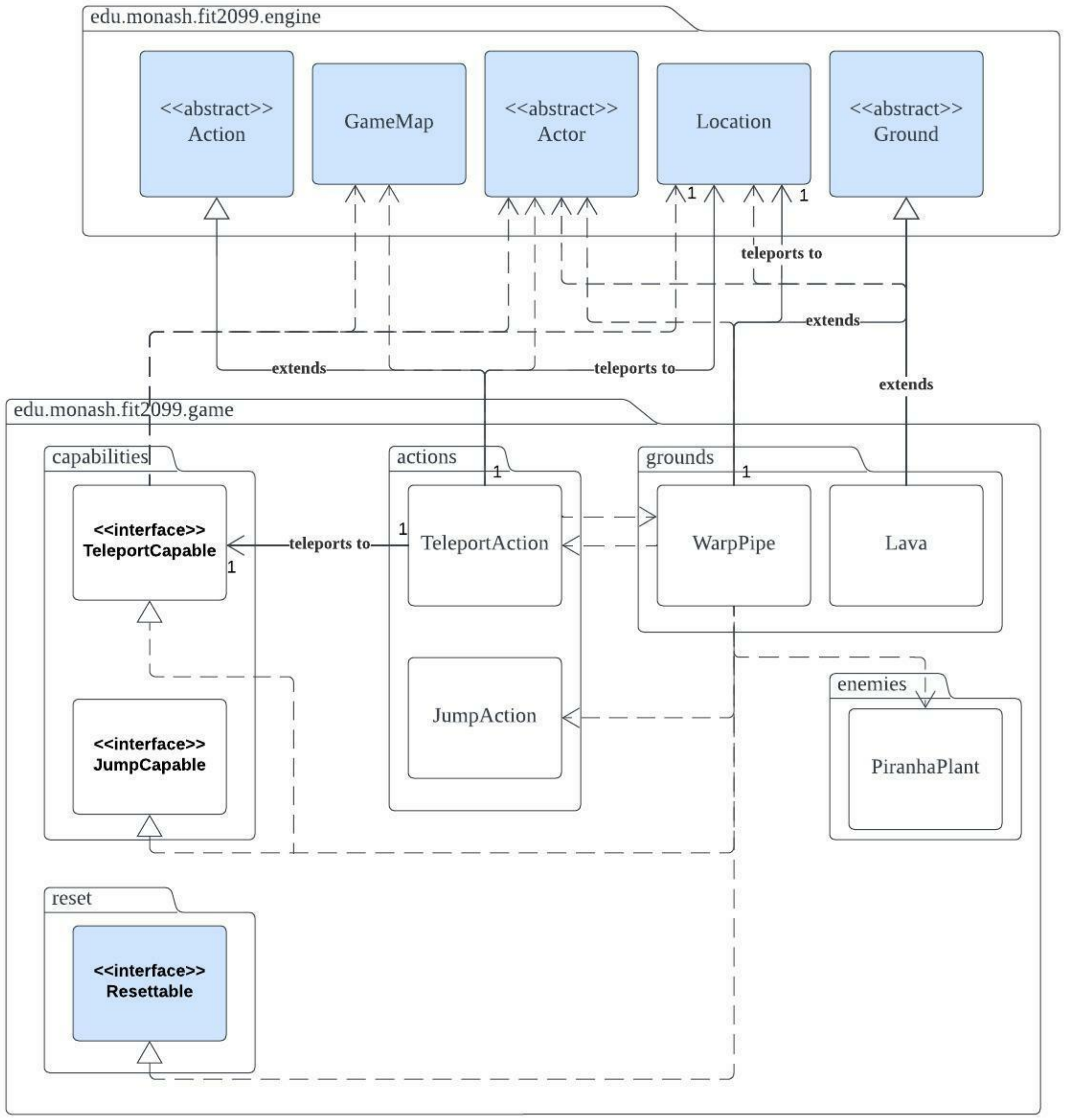
TeleportCapable

TeleportCapable is an interface which can be implemented by grounds that can teleport actors to another location, which will be called teleporters in here. An interface is used because we can adhere to the Open / Closed Principle. The teleporters will implement this interface and if we want to add more features to these classes, we don't need to change anything inside the interface, as the methods in the interface are standard across all teleporters. This interface also adheres to the Don't Repeat Yourself principle. We can have a default method in the interface which performs the teleportation for all the teleporters rather than having the teleport method in all the teleporter classes, which reduces duplicate code. TeleportCapable has a dependency on Location because we need the location information of the source location and destination location to know where to teleport to and which teleporter we came from. In the teleport method of this class, there is a downcasting of Ground to TeleportCapable, this is because in this case, we will always teleport to another teleporter, so we can confirm that the ground that the player is standing on right after teleporting is a TeleportCapable ground.

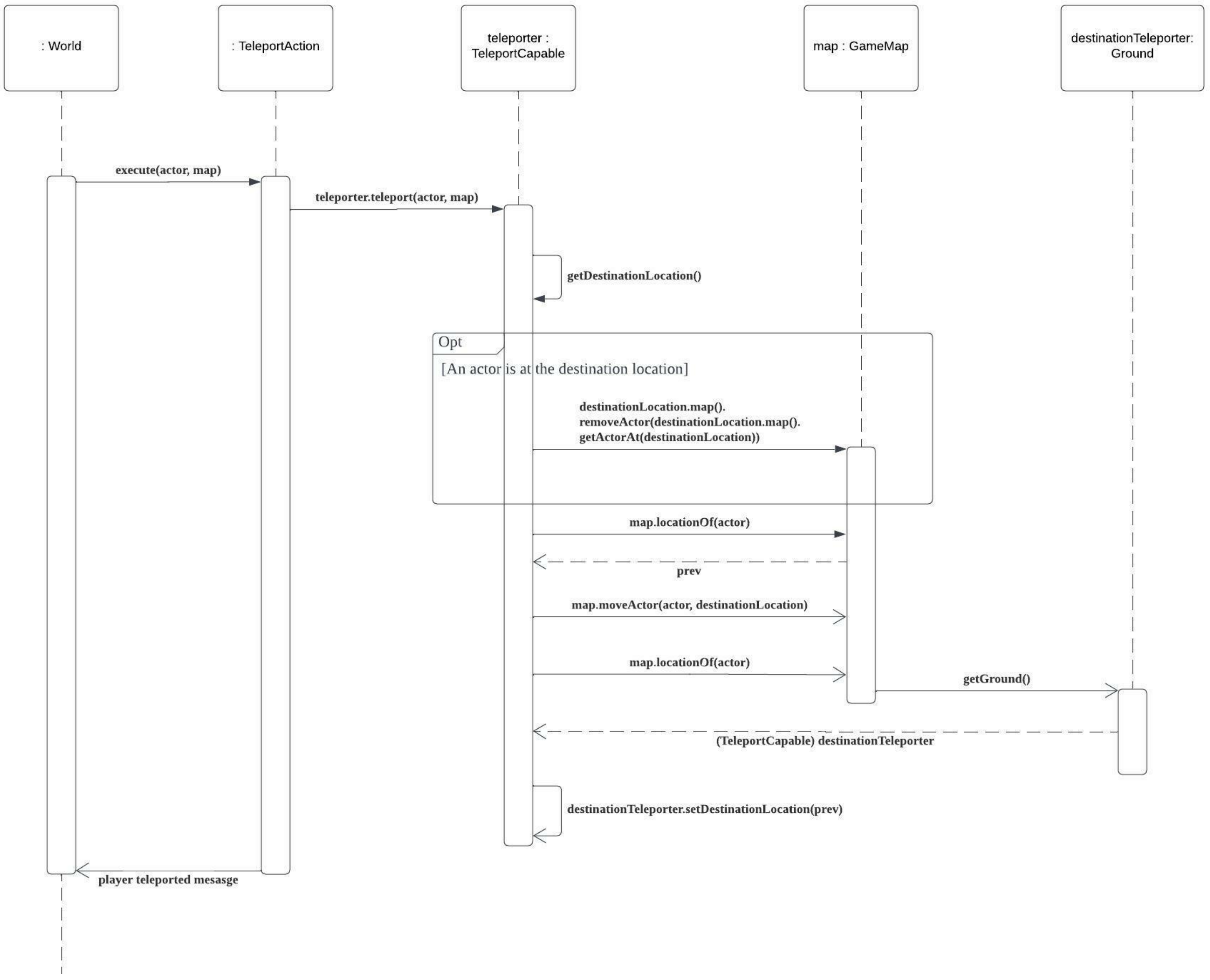
TeleportAction

TeleportAction is a class used by the player to teleport to another location. TeleportAction extends the Action abstract class because teleporting is an action, and to adhere to the Single Responsibility Principle where this class only handles the teleporting actions. TeleportAction has an association to TeleportCapable because the TeleportAction can execute the teleport default method in the TeleportCapable interface, and it needs a TeleportCapable instance to know which TeleportCapable object to execute the teleportation process.

Class Diagram



Sequence Diagram



: World
: TeleportAction
teleporter : TeleportCapable
map : GameMap
destinationTeleporter: Ground
execute(actor, map)
teleporter.teleport(actor, map)
getDestinationLocation()
Opt
[An actor is at the destination location]
destinationLocation.map().
removeActor(destinationLocation.map().
getActorAt(destinationLocation))
map.locationOf(actor)
prev
map.moveActor(actor, destinationLocation)
map.locationOf(actor)
getGround()
(TeleportCapable) destinationTeleporter
destinationTeleporter.setDestinationLocation(prev)
player teleported mesasge

REQ 2 - More allies and enemies!

Bowser

Bowser is a subclass of the *Enemy* abstract class because *Bowser* is a type of enemy in the game. *Koopa* starts with 500HP and will drop a *Key* to unlock *PrincessPeach's* handcuffs when the player defeats it. *Bowser* will overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a punch and 80 damage, the hit rate is the same as *Player* so we don't have to create other methods to modify the hit rate value.

Bowser would override the *allowableActions* method as it will add an *AttackBehaviour* and a *FollowBehaviour* to the behaviours HashMap which will be cleared during the instantiation of Bowser.

Koopa will execute the *playTurn* method of the parent class (Enemy abstract class) through the "super" keyword. As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed. Hence we won't need to repeat the same chunk of code which fulfills the **Don't Repeat Yourself principle[2]** as repeating code is avoided.

The *resetInstance* method is also overridden to move *Bowser* back to the original position, heal it to maximum, and stand there until *Player* is within its attack range.

Fire

Fire is a subclass of *Item* that lasts for 3 turns on the ground and damages any actor that is standing on top of it.

Key

Key is an *Item* which is used to free *PrincessPeach*, which extends from the *Item* class.

PrincessPeach

PrincessPeach is a class that allows the player to interact with in order to win the game. *PrincessPeach* extends the actor class as it is an actor. *PrincessPeach* has a dependency on the *FreePrincessPeachAction* because it is used to determine whether the player has won the game.

FreePrincessPeachAction

FreePrincessPeachAction is a subclass of *Action* which will provide the player with an action to free *PrincessPeach* if the player has the key in their inventory. Once this action is executed, the player will be removed from the map, which will cause the game to stop running, and the game is over. It's adhering to the **Single ResponsibilityPrinciple[3]**.

PiranhaPlant

PiranhaPlant is a subclass of the *Enemy* abstract class because *PiranhaPlant* is a type of enemy in the game. *PiranhaPlant* starts with 150hp. *PiranhaPlant* will override the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that attacks with a chomp and 90 damage, the hit rate is the same as player so we can just overrides this method instead of creating duplicate methods.

Koopa

Koopa will be extending from the *Enemy* abstract class as it is a type of *Enemy*. It represents the parent class of all types of Koopa (*WalkingKoopa*, the normal Koopa and *FlyingKoopa*). We make it an abstract class to ensure we follow the **Dependency Inversion Principle[7]**.

Koopa would override the *playTurn* method as it will go to a dormant state and stay on the ground when it is not conscious and will then be removed from the map and drop a Super Mushroom once its shell is destroyed by the player by a Wrench. If *Koopa* is conscious, *Koopa* will execute the *playTurn* method of the parent class (Enemy abstract class) through the "super" keyword. The default implementation of *playTurn* is to execute the *WanderBehaviour*, *FollowBehaviour* and *AttackBehaviour* which are the default behaviour of the enemies based on the *Status* of the *Player* and *Enemy*. As the behaviours are stored in the form of HashMap in the *Enemy* abstract class, the existing structure will not be changed. Hence we won't need to repeat the same chunk of code which fulfills the **Don't Repeat Yourself principle[2]** as repeating code is avoided.

Koopa will override the *allowableActions* method to check if the *Player* has a *Wrench*. If the *Koopa* is not conscious but the *Player* doesn't have a *Wrench*, it will just return. *Koopa* will execute the default implementation of *allowableActions* method of the parent class through the "super" keyword and just add *AttackKoopaAction* to cater the additional attack action of *Player* to *Koopa* which fulfills the **Don't Repeat Yourself principle[2]**.

We created this class instead of just letting *FlyingKoopa* extend from the concrete *Koopa* class, to avoid violating the **Liskov Substitution principle[2]**. Hence to avoid changing the behaviour of a normal *Koopa*, we created subclasses for the *FlyingKoopa* and *WalkingKoopa* (representing the normal *Koopa*) that extend from this abstract class.

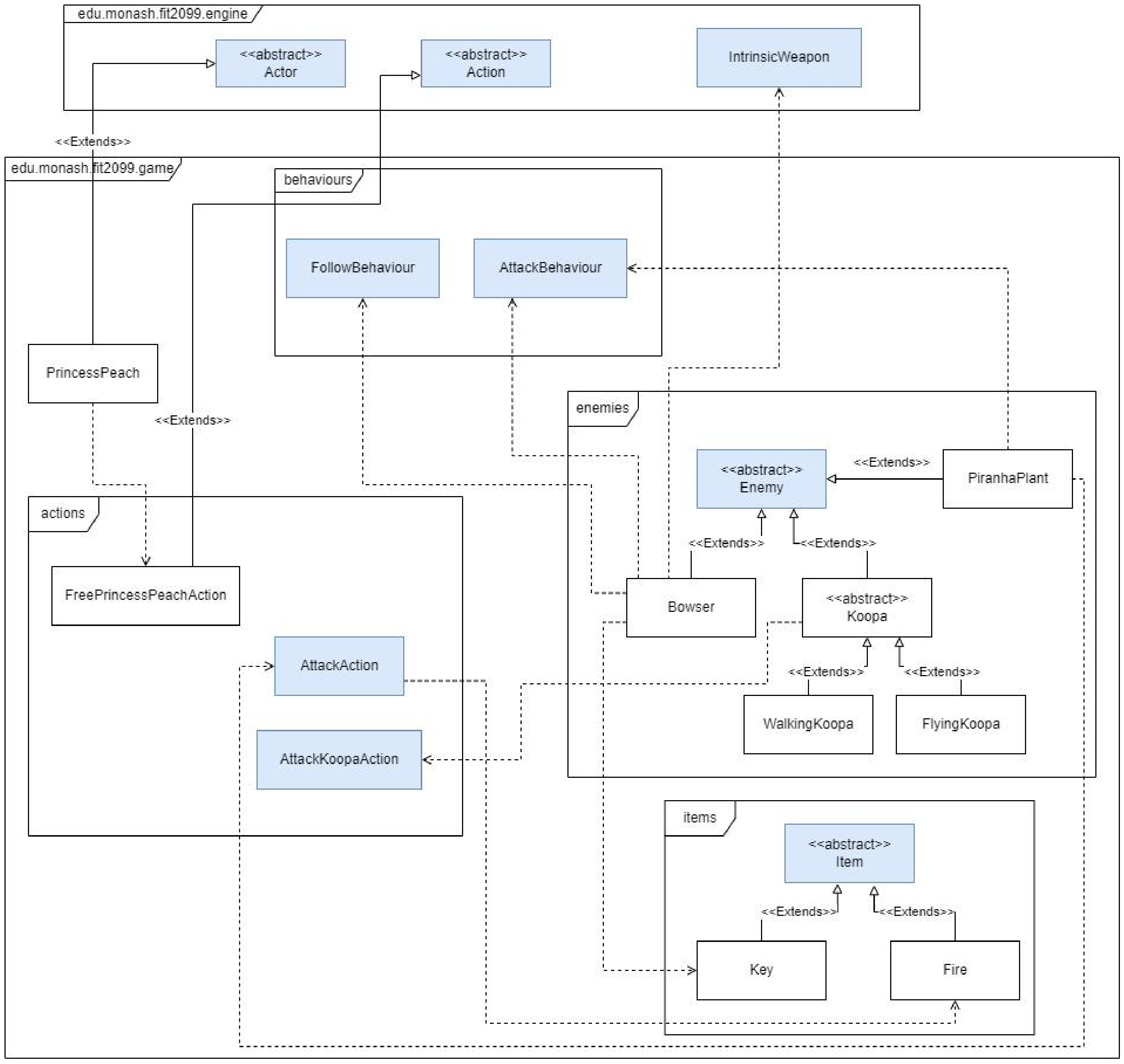
WalkingKoopa

WalkingKoopa is a subclass of the *Koopa* abstract class because *WalkingKoopa* is a type of *Koopa* in the game. *WalkingKoopa* is the initial normal that starts with 100HP and will go into dormant state (D) when the player defeats it. hence, it will just execute the *Koopa* abstract class's *allowableActions* and *playTurn* method to avoid code duplication, adhering to the **Don't Repeat Yourself principle[2]**.

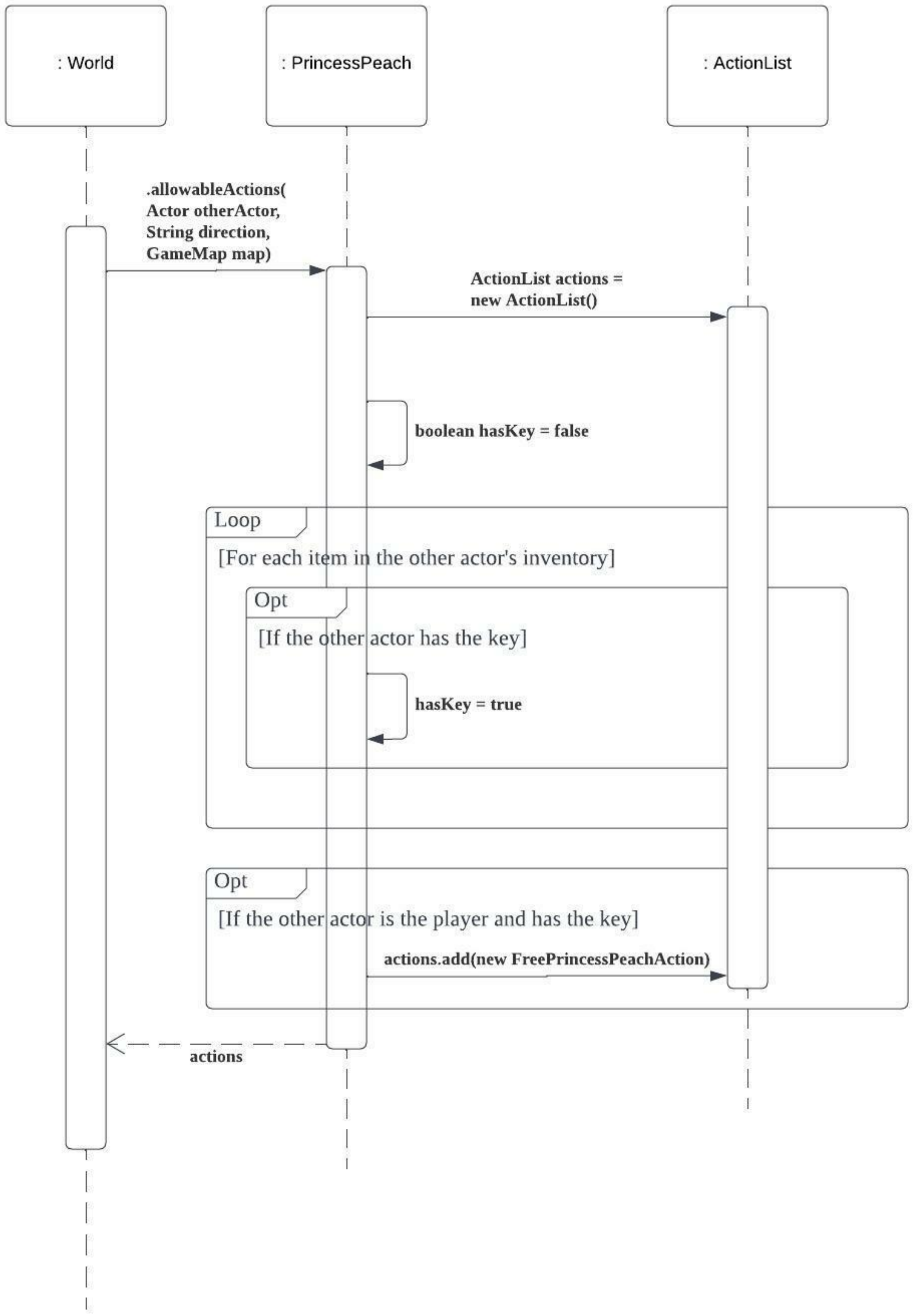
FlyingKoopa

FlyingKoopa is a subclass of the *Koopa* abstract class because *FlyingKoopa* is a type of *Koopa* in the game. *FlyingKoopa* starts with 150HP and will go into dormant state (D) when the player defeats it. It will also be removed from the map and drop a Super Mushroom once its shell is destroyed by the player by a Wrench. It has almost the same characteristics as a normal Koopa, except it will fly over high grounds while wandering. Hence, it will just execute the parent class's *allowableActions* and *playTurn* method.

Class Diagram



Sequence Diagram



REQ 3 - Magical Fountain

MagicalFountain

MagicalFountain will be extending from the *Ground* abstract class as it is a type of *Ground*. It represents the parent class of all fountains (Health Fountain and Power Fountain) and we do not want it to be instantiated by other classes. As each water from a fountain provides different effects, we make it an abstract class to ensure we follow the **Dependency Inversion Principle[7]** where a concrete class should not depend on another concrete class and should depend on abstractions instead. As the subclasses will have the same way of counting the "five turns" timer when the waters in the Fountain are fully exhausted, the *tick* method is implemented in this class to avoid violating **Don't Repeat Yourself principle[2].**

HealthFountain

HealthFountain is a subclass of the *MagicalFountain* class. Adhering to the **Don't Repeat Yourself principle[2]**, this class will only implement the abstract methods and call the methods implemented by the parent class when needed. The implementation of the abstract methods is similar to the *PowerFountain* but as they are storing different types of water, an ArrayList is instantiated as a class attribute to represent the slots of the *HealingWater* in the fountain.

PowerFountain

PowerFountain is a subclass of the *MagicalFountain* class. This class will also only implement the abstract methods to fulfill the **Don't Repeat Yourself principle[2]**. The implementation of the abstract methods is similar to the *HealthFountain* but as they are storing different types of water, an ArrayList is instantiated as a class attribute to represent the slots of the *PowerWater* in the fountain. So the amount of water slots left in the fountain is separated from the *HealthFountain*.

HealingWater

HealingWater is a subclass of the *MagicalItem* class as it will be consumed by the actors and provide special effects after being consumed. As we extend from the MagicalItem class, we have to implement the abstract *consumedBy* method, which increases the actor's hit points by 50 every time it is called.

PowerWater

PowerWater is a subclass of the *MagicalItem* class as it will be consumed by the actors and provide special effects after being consumed. As we extend from the MagicalItem class, we have to implement the abstract *consumedBy* method, which increases the actor's intrinsic damage by 15. The intrinsic damage of an actor is set as protected, hence we can't access it from different package classes. To solve this issue, we decided to implement a public *setDamage* method in the *Player* and *Enemy* class and call them in the *consumedBy* method by downcasting the *Actor* type to the respective type of the actor after checking via its display character which we could avoid using "instanceOf" keyword which violates the **Open-Closed Principle[4]**.

Bottle

Bottle is a subclass of the *Item* class. As we will only have one bottle which will be given to the Player, this class is created as a singleton class. Hence, only one instance of the *Bottle* will be created during the game. A *Stack* is used to store the water of the *MagicalFountain.* We fill in the water to the bottle via *push*, and let the actors consume the water by *popping* out from the stack.

GiveBottleAction

GiveBottleAction will be extended from the *Action* abstract class. *Player* doesn't have a bottle in his inventory at the start of the game. Instead, *Player* will obtain it from *Toad*. In this class, we get the singleton *Bottle* instance as the class attribute by calling the public static factory method getInstance() of the class *Bottle*, resulting in a one-to-one association relationship between *GiveBottleAction* and *Bottle*. It overrides the *Action* class *execute* method to add a bottle to the player's inventory if a bottle doesn't exist. This class fulfills the **Single Responsibility Principle [3]**.

RefillAction

RefillAction will be extended from the *Action* abstract class. This class adheres to the **Single Responsibility Principle [3]** as its main function is to just refill the bottle with the water of the fountain the *Player* is standing on. It overrides the parent class's *execute* method by pushing the water into the *Stack* if a bottle exists in the *Player's* inventory.

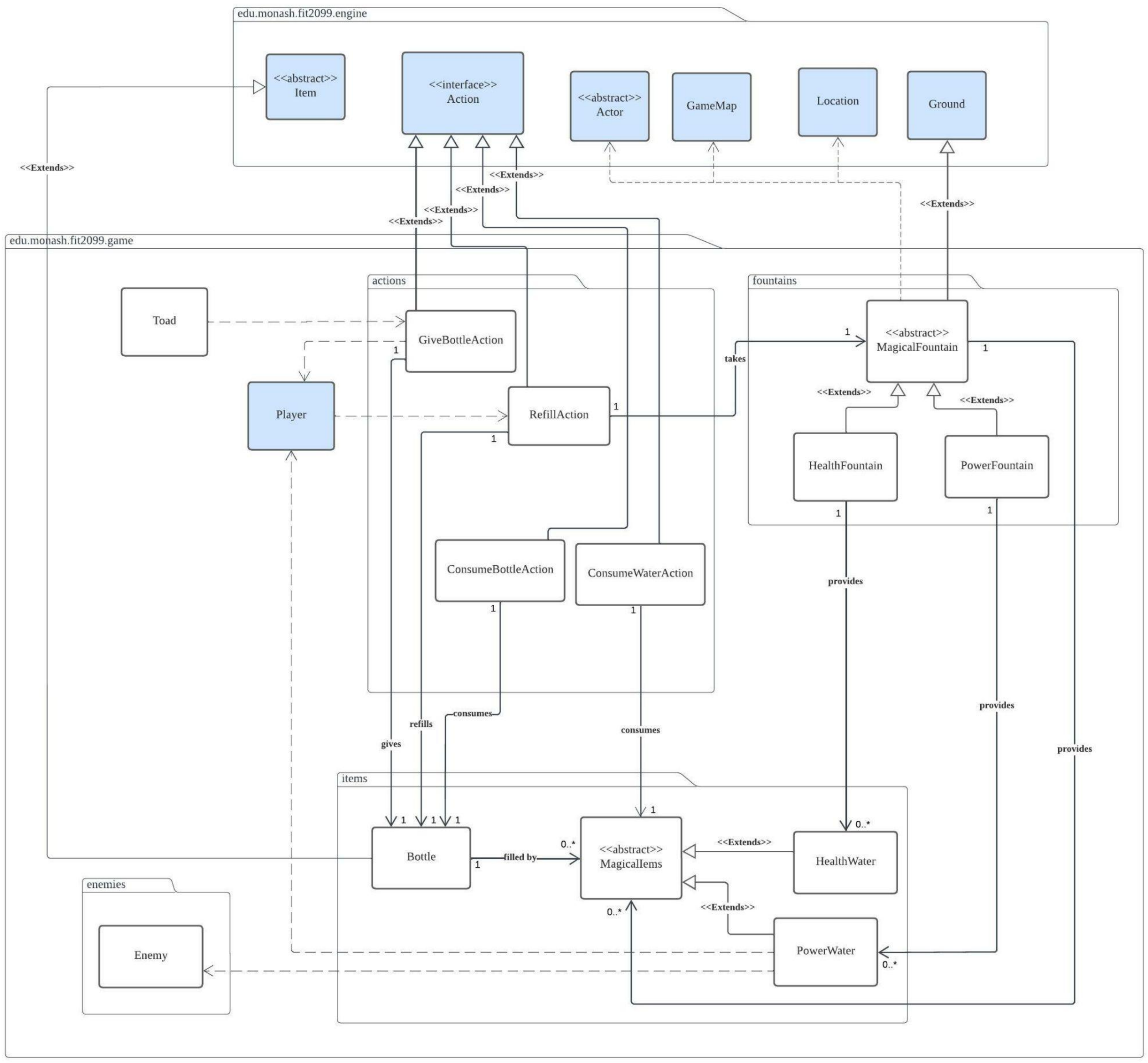
ConsumeBottleAction

ConsumeBottleAction will be a subclass of *Action* abstract class, it inherits to **minimize code duplication[2]**. *ConsumeBottleAction* fulfills the **Single Responsibility principle[3]** because it will only pop the magical water from the *Bottle* and consume the *water*. The *execute*() method is overridden, checking whether the stack of the *Bottle* is not empty to pop the water out from the stack, followed by calling the consumedBy() method of the related *HealingWater* or *PowerWater* class to consume the water*,* as the water is a subclass of *MagicalItem* abstract class. The menu description is overwritten as well to output the message "[*Actor*] consumes bottle[a mixture of *HealingWater* and *PowerWater*]".

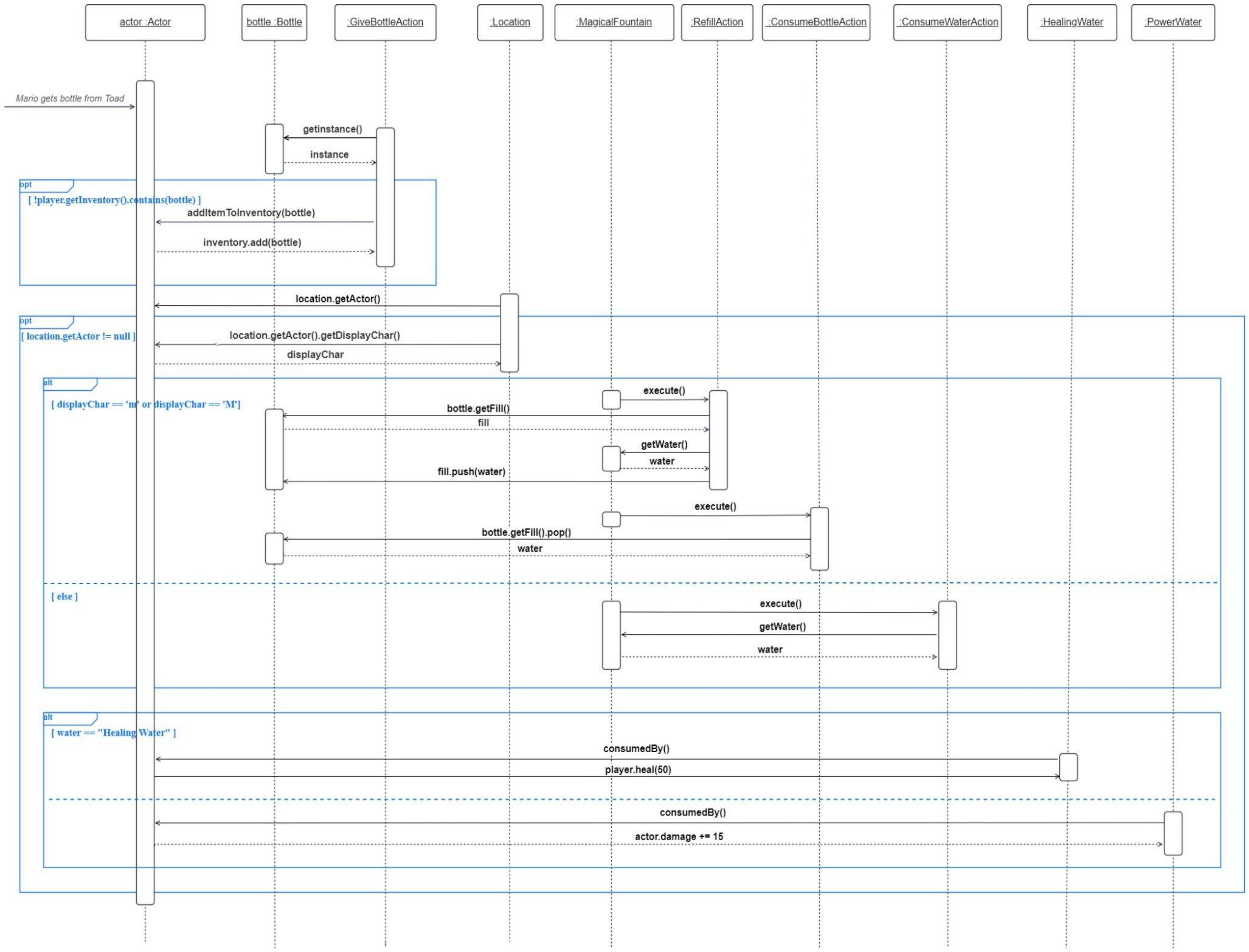
ConsumeWaterAction

ConsumeWaterAction will be extended from the *Action* abstract class. This class adheres to the **Single Responsibility Principle [3]** as its main function is to let the enemies consume the magical water. It overrides the parent class *execute* method by just calling the *consumedBy* method of the respective *HealingWater* or *PowerWater* class which depends on the fountain the enemies are standing on. It will then print a message "[*Enemy*] consumes [*Water*]" after the *consumedBy* method is called. The *menuDescription* method will return null as we would not have the options to control this action. The enemies will directly consume the water when they stand on top of the *MagicalFountain,* hence we would not need a ConsumeBehaviour class to cater to this action's priority, since it will always call this class's execute method before other behaviours are executed.

Class Diagram



Sequence Diagram

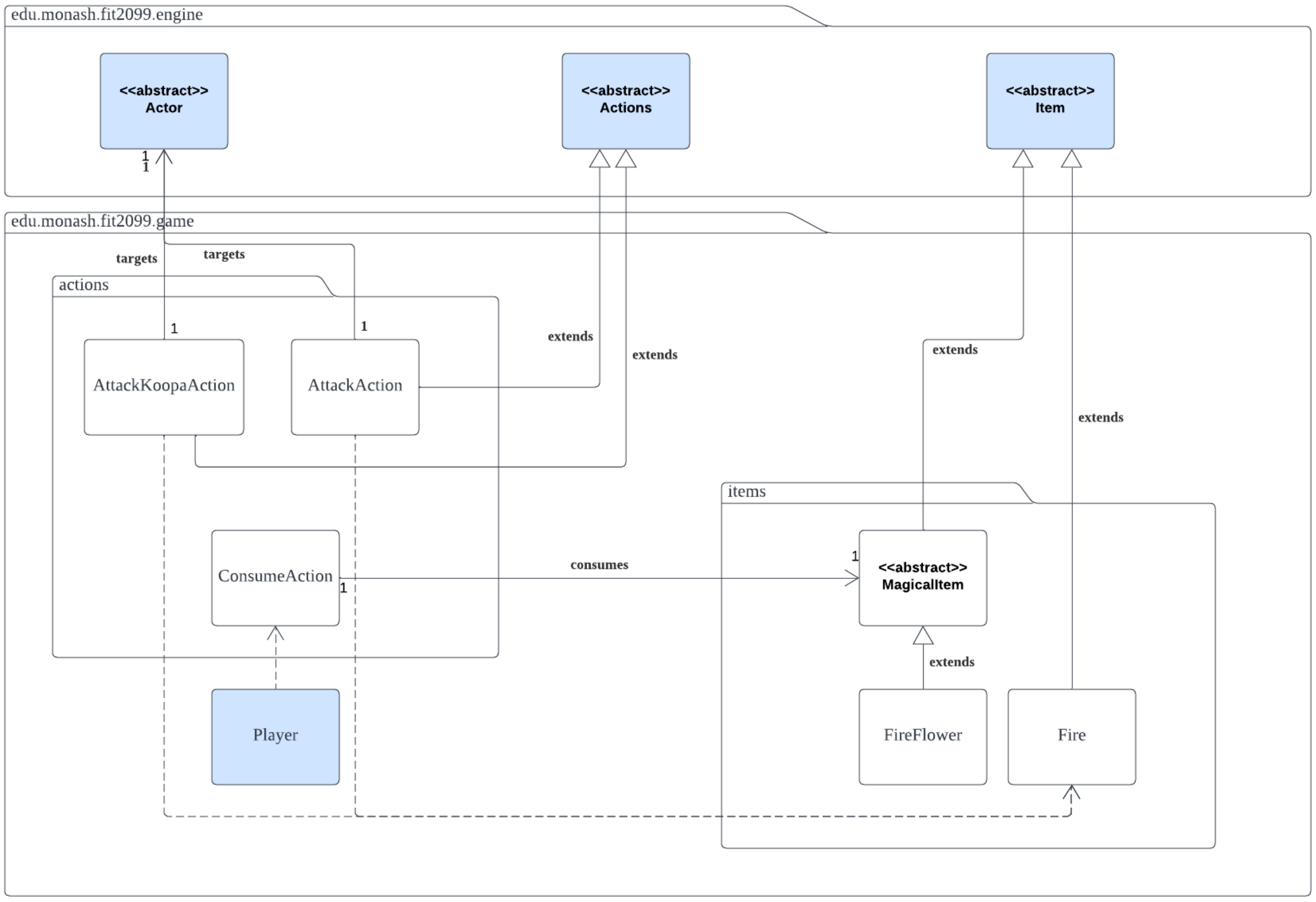


REQ 4 - Flowers (Structured mode)

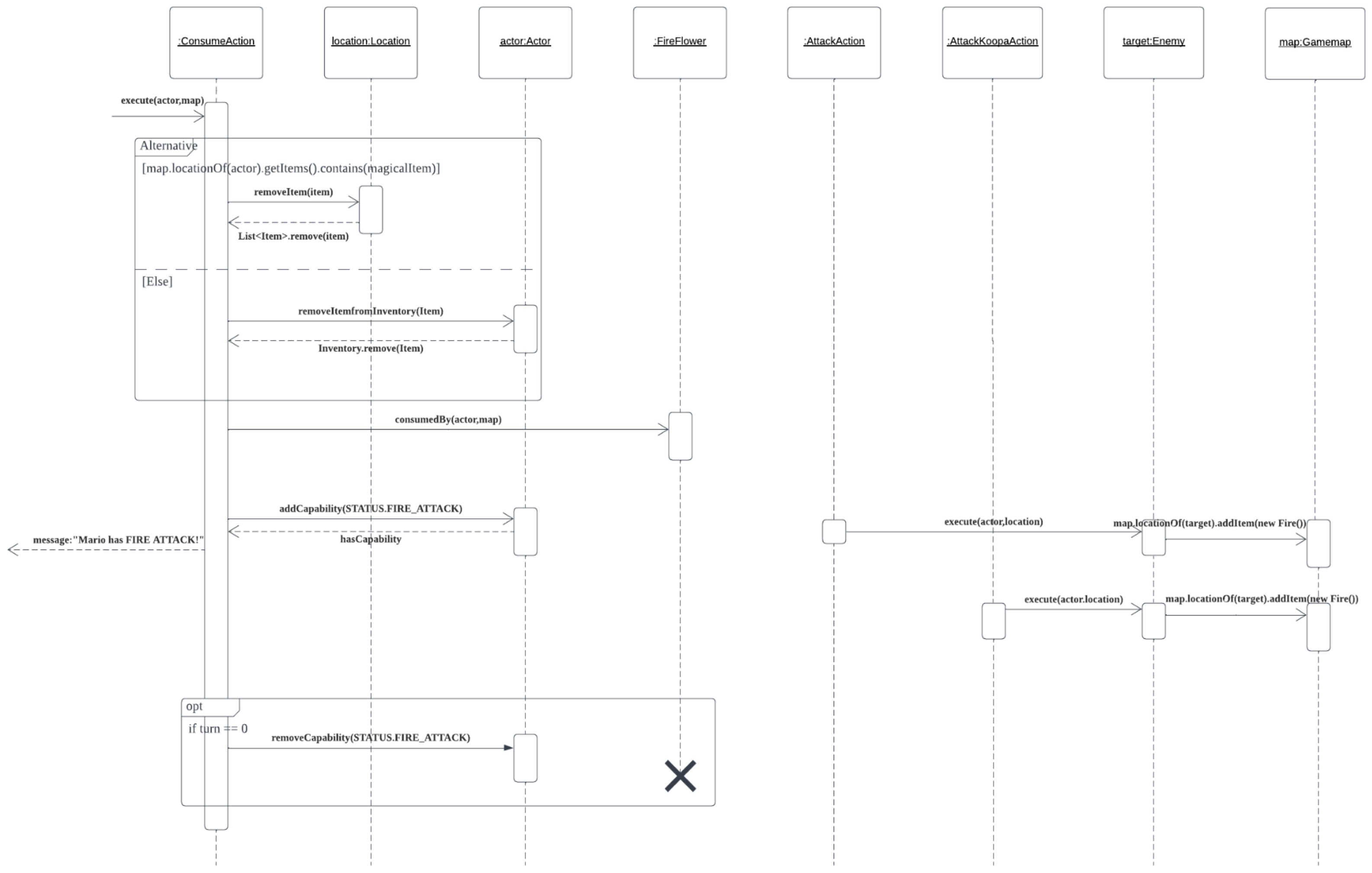
FireFlower

FireFlower will be extending from the MagicalItem abstract class. It will override the consumedBy() method of its parent class which executes the special features after the Actor consumes it. It will change the status of the actor to Status.FIRE_ATTACK and **avoid excessive use of literals[1]**.

Class Diagram



Sequence Diagram

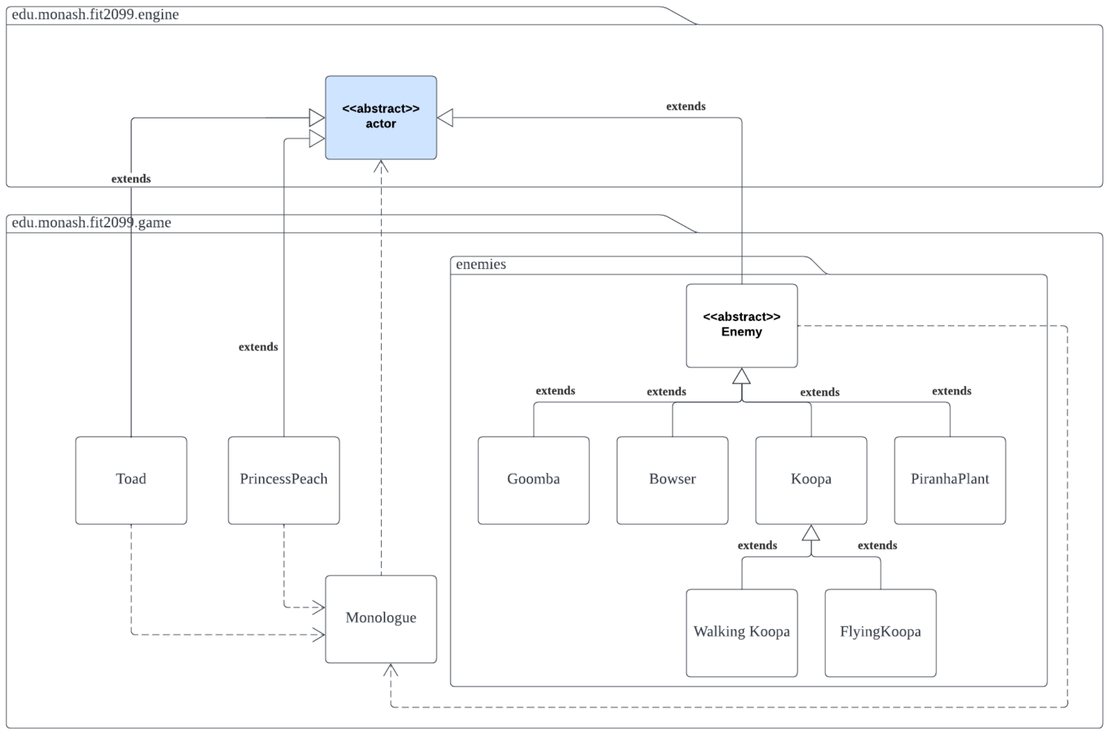


REQ 5 - Speaking (Structured mode)

Monologue

Monologue is a class that picks random sentences from a list of monologues by the respective actors. *Monologue* has “actors” as a HashMap with *Actor* as the key and *Integer* as a value to represent the number of turns that each actor has. “Actors” was created as a HashMap as there are multiple actors in a game and it is easier to just map the actors to their respective turn counter. *Monologue* also has another HashMap “actorsTalkList” with *Actor* as the key and *ArrayList<String>* as the value to represent the actors and their respective monologue lines.

Class Diagram



Work Breakdown Agreement Assignment 3

Zoe Low Pei Ee	31989985	zlow0011@student.monash.edu
Chua Shin Herh	31902456	schu0064@student.monash.edu
Loh Zhun Guan	32245327	zloh0009@student.monash.edu

Task: Req 1: Create and update Lava Zone Class Diagram, Sequence Diagram and the related classes

Person In Charge: Chua Shin Herh

Deadlines: 19th May 2022

Task: Req 2: Create and update the allies and enemies classes, Class Diagram and Sequence Diagram

Person In Charge: Chua Shin Herh, Zoe Low Pei Ee

Deadlines: 19th May 2022

Task: Req 3: Create Magical Fountain related classes, Class Diagram and Sequence Diagram.

Person In Charge: Zoe Low Pei Ee

Deadlines: 19th May 2022

Task: Req 4: Create Fire flower and the related classes, Class Diagram and Sequence Diagram

Person In Charge: Loh Zhun Guan

Deadlines: 19th May 2022

Task: Req 5: Create Monologue class, Class Diagram and Sequence Diagram and update the affected classes.

Person In Charge: Loh Zhun Guan

Deadlines: 19th May 2022

Task: Create Readme.md file and do amendments for Assignment 2 mistakes

Person In Charge: Zoe Low Pei Ee

Deadlines: 19th May 2022

Task: Update the design rationale of each REQ

Person In Charge: Collaborative

Deadlines: 19th May 2022

Task: Review and testing

Person In Charge: Collaborative

Deadlines: 19th May 2022

1. Each task, when completed, is to be pushed onto the repository.

2. Work will be pushed to the git repository frequently to ensure each member gets the latest update of the work, push partially completed tasks with the commit message “TO BE COMPLETED”.

3. Diagrams are required to be pushed after any changes to ensure consistency. Diagrams will be made through lucidchart.com.

4. All work to be done before the deadline, no work will be done later than the 19th May 2022.

If the member agrees to the terms, please sign your name, and date, with “I accept this WBA”.

- Zoe Low Pei Ee, 10th May 2022, I accept this WBA.
- Chua Shin Herh, 10th May 2022, I accept this WBA.
- Loh Zhun Guan, 10th May 2022, I accept this WBA.