WBA for Assignment 3 - FIT 2099 Team: Lab 14 Team 3

Members: Jun Heng Tan (3280 5217), Dayan Perera (3224 2026), Kuan Yi Xuan (3202 3227)

REQ 1: Code, Javadoc, UML Diagram, Sequence Diagram and Design Rationale
--> Managed by Jun Heng Tan (3280 5217)
- Will be reviewed by Dayan Perera (3224 2026) & Kuan Yi Xuan (3202 3227)

REQ 2: Code, Javadoc, UML Diagram, Sequence Diagram and Design Rationale
--> Managed by Jun Heng Tan (3280 5217)
- Will be reviewed by Dayan Perera (3224 2026) & Kuan Yi Xuan (3202 3227)

REQ 3: Code, Javadoc, UML Diagram, Sequence Diagram and Design Rationale
--> Managed by Dayan Perera (3224 2026)
- Will be reviewed by Jun Heng Tan (3280 5217) & Kuan Yi Xuan (3202 3227)

REQ 4: Code, Javadoc, UML Diagram, Sequence Diagram and Design Rationale
--> Managed by Dayan Perera (3224 2026)
- Will be reviewed by Jun Heng Tan (3280 5217) & Kuan Yi Xuan (3202 3227)

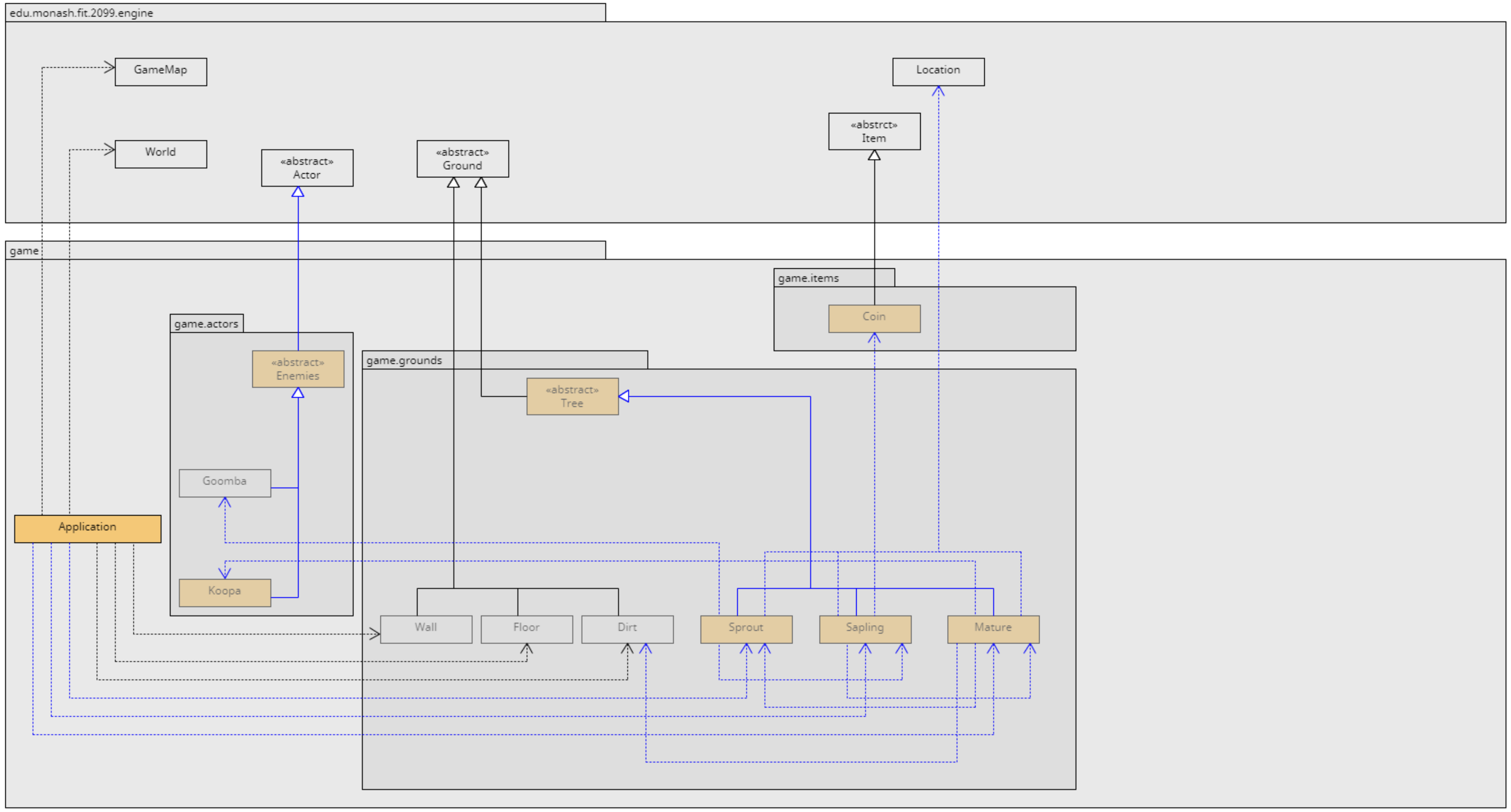
REQ 5: Code, Javadoc, UML Diagram, Sequence Diagram and Design Rationale
--> Managed by Kuan Yi Xuan (3202 3227)
- Will be reviewed by Jun Heng Tan (3280 5217) & Dayan Perera (3224 2026)

Jun Heng Tan: I accept this WBA

Dayan Perera: I accept this WBA

Kuan Yi Xuan: I accept this WBA

Assignment 1- REQ1: Let it grow!



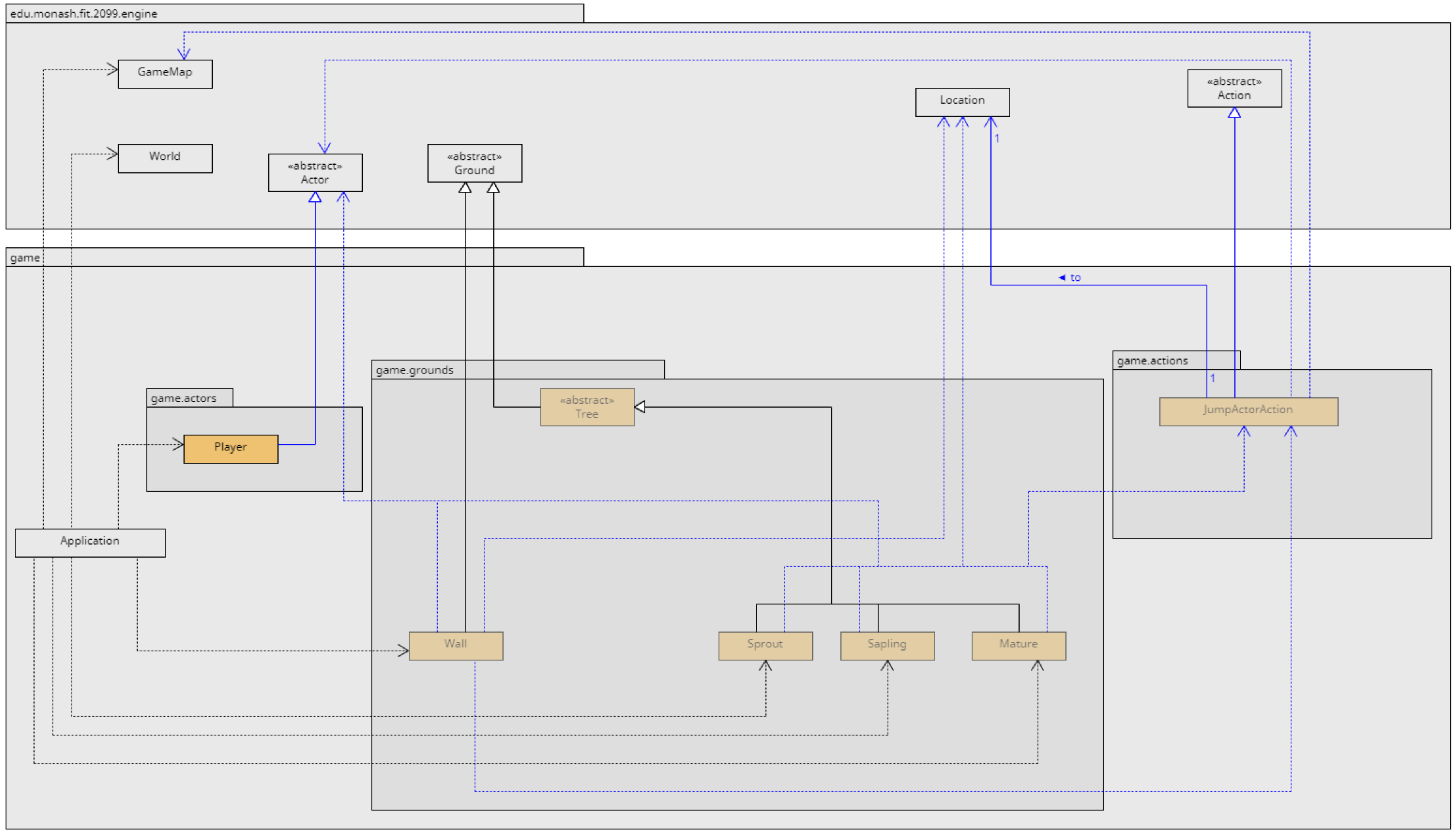
-

- New concrete classes of Sprout, Sapling & Mature. All extending Tree as its parent class.
    - Tree class will thus be made into an abstract class. Prevents DRY.
    - Tree class will only contain a modified canActorEnter methods.

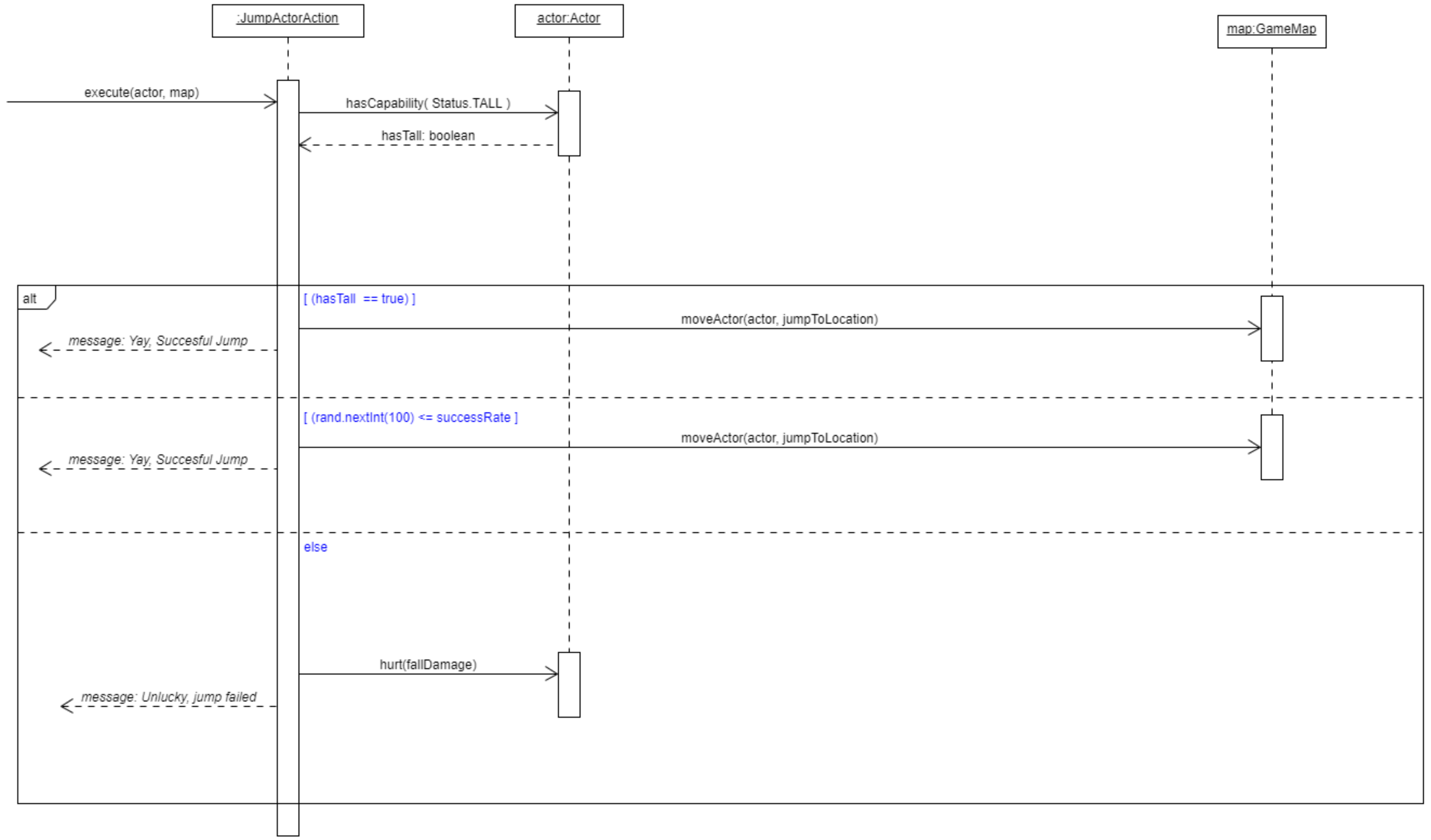
- All 3 classes will have one integral method tick() in their respective classes. Overriding the original method definition in the abstract class Ground.
    - Responsible for central decision making. Will check if criteria for certain actions are met, and if met call those actions. (eg. Sprout class → Call spawnGoomba() method, if the 10 % chance is met)
    - Actions are not performed in method tick() (eg. spawn a Goomba/ Koopa) to minimise method responsibility and to ensure the Single Responsibility Principle is followed for the method.

- New Concrete Class Koopa & Coin will be created (methods in them will be explained later down this document)
    - Class Coin will extend abstract class Item.
    - Class Koopa will inherit from abstract class Enemies. (Explained more later)

Assignment 1 - REQ2: Jump Up, Super Star!



execute.JumpActorAction:


:JumpActorAction
actor:Actor
map:GameMap
execute(actor, map)
hasCapability(Status.TALL)
hasTall: boolean
alt
[(hasTall == true)]
moveActor(actor, jumpToLocation)
message: Yay, Succesful Jump
[(rand.nextInt(100) <= successRate]
moveActor(actor, jumpToLocation)
message: Yay, Succesful Jump
else
hurt(fallDamage)
message: Unlucky, jump failed

*successRate is passed by JumpActorAction's constructor.

- New JumpActorAction class inheriting from class Action.
    - All concrete classes that inherit the abstract Action class have the necessary methods to provide written text before (menuDescription) and after the interaction executes.
        - execute(Actor actor, GameMap map) ⇒ Evaluates the result of the jump.
        - menuDescription(Actor actor) ⇒ menuDescription will output the required menu display.
    - When the JumpActorAction is created using its constructor, information such as the fall damage, success rate, jump to location and direction will be passed. This is done to ensure the open closed principle. Whereby we do not need to check the type of ground actor is jumping to to determine its success rate and fall damage if jump fails. Meaning there will be no use of instance of for checking jump to location's ground type and no multiple if statements.

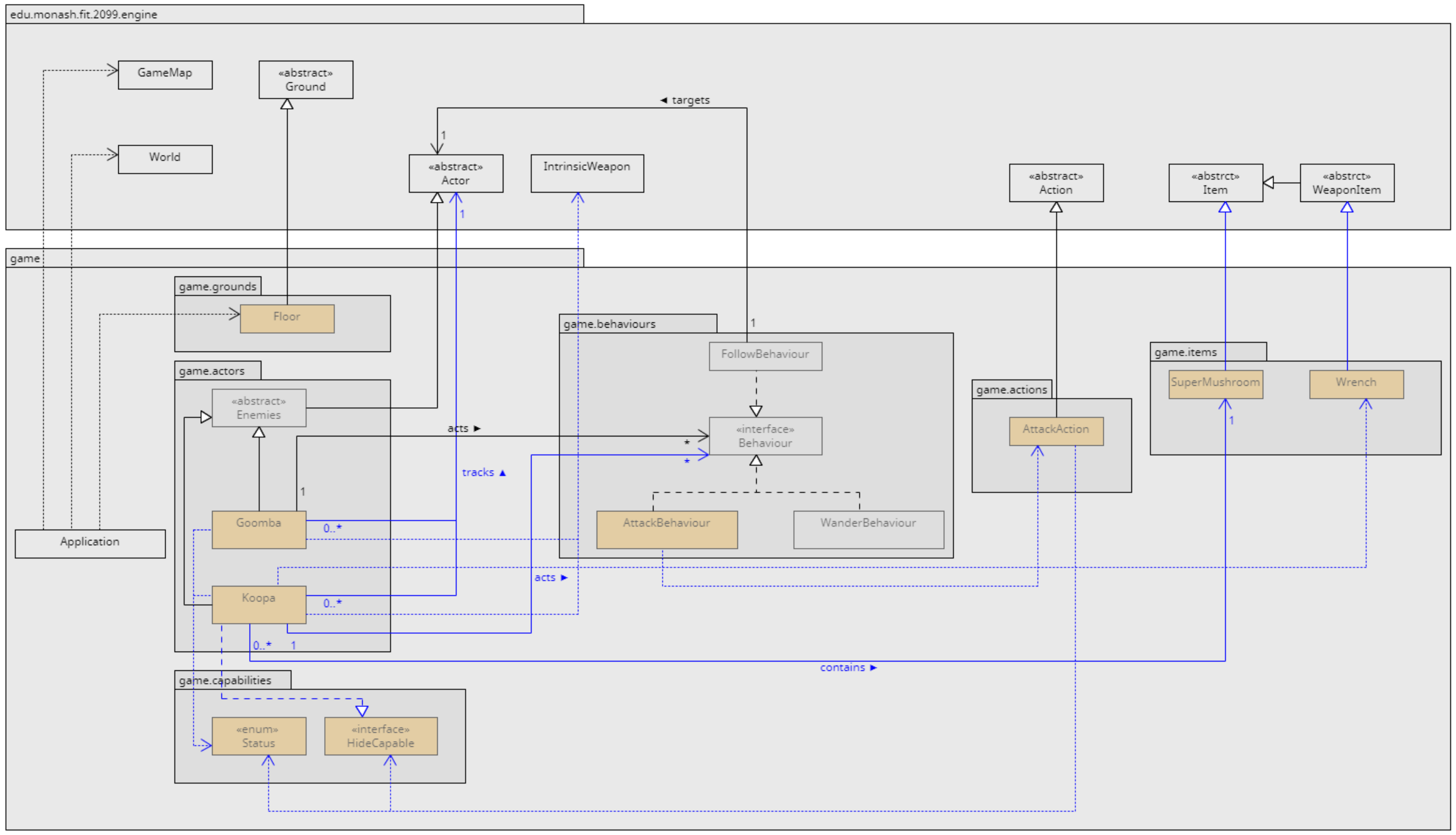
- Grounds that have to be jumped to enter will have overridden/ modified canActorEnter() methods. (i.e. Wall, Tree)
    - returns true if the Actor does have the capability of INVINCIBLE else false.
    - This ensures MoveActorAction returned if INVINCIBLE instead of JumpActorAction.

- Grounds that have to be jumped to enter will << create >> new JumpActorAction. Done through the use of its allowableActions() method which an actor will call when at its surroundings. This is done here as only grounds that have to be jumped over should have this action as an option for the player. This is not done in Player's playTurn() method to prevent a GOD method. As well as to reduce dependencies on Player's class.
    - Note: Can be done in Players playTurn by checking player's surrounding then adding a JumpActorAction if it is of higher ground

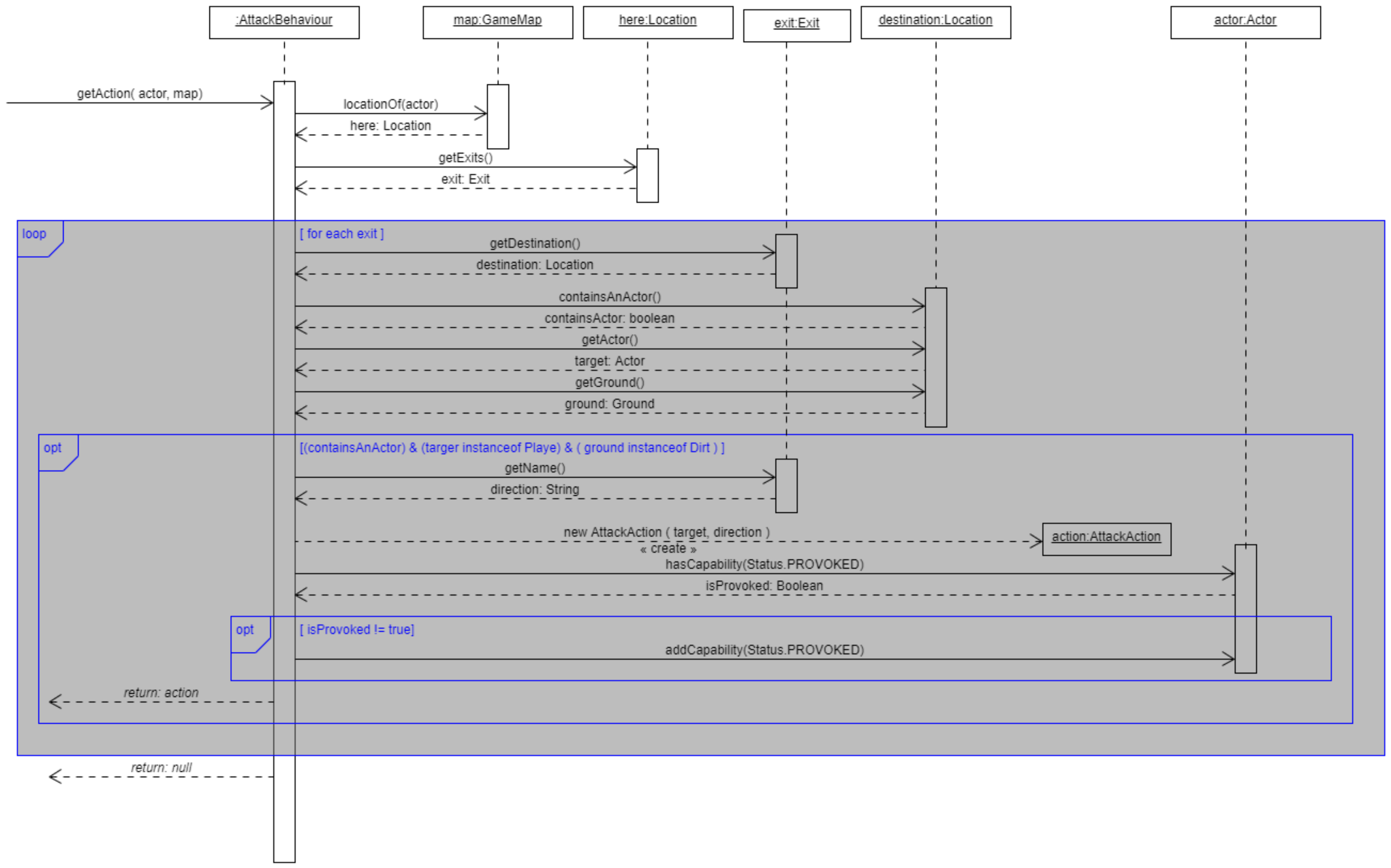
- No changes needed to be made for the enemy Goomba and Koopa classes. Will not be able to jump.
    - PlayTurn() method in their respective classes prevents this.

- Modification of Player class.
    - New method highGround(). Called in playTurn().
        - Checks if player is on top of a high ground. Display its coordinates and its ground type if true.
    - Maintains single responsibility principle for playTurn method. And prevents it from being a GOD method.

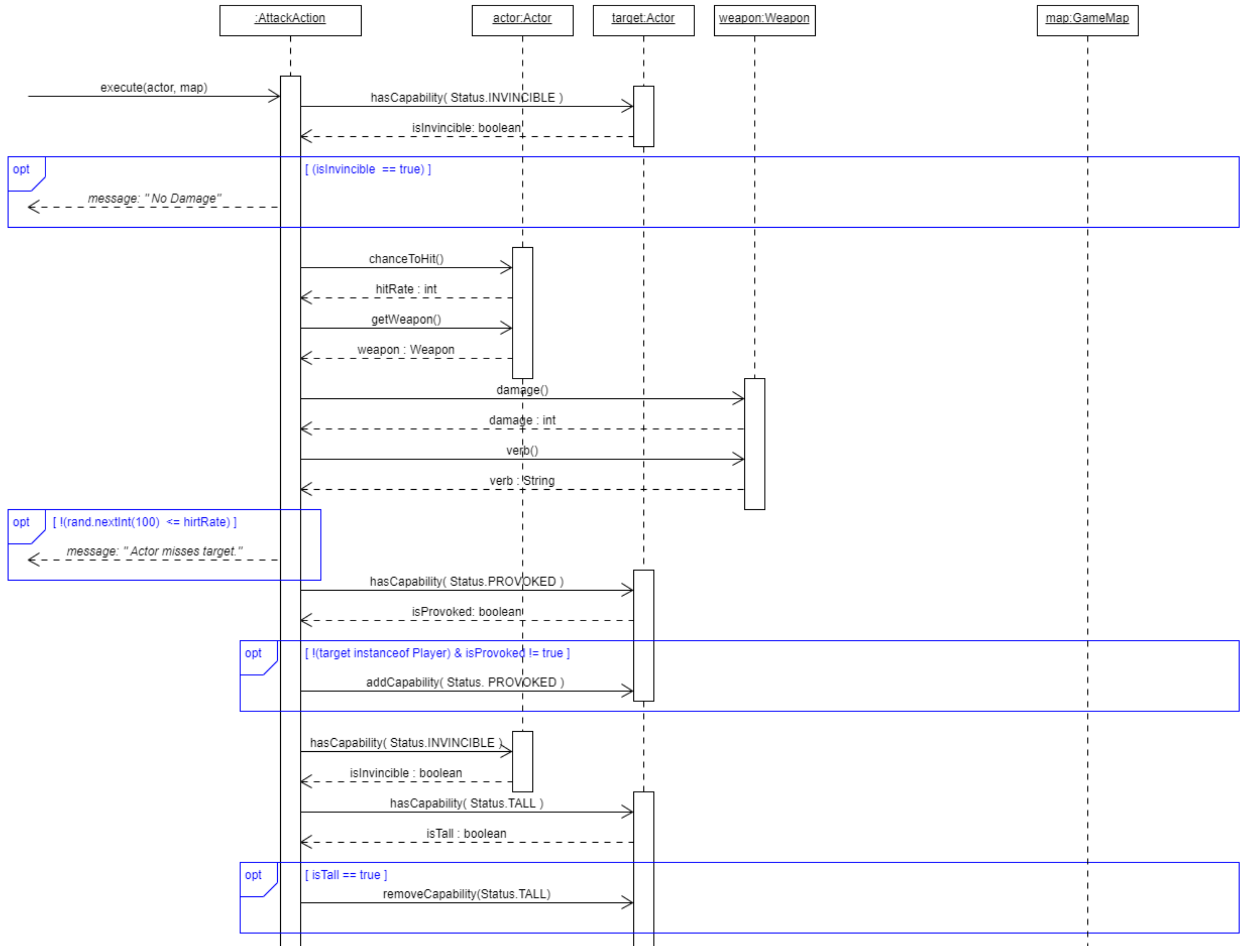
Assignment 1 - REQ3: Enemies



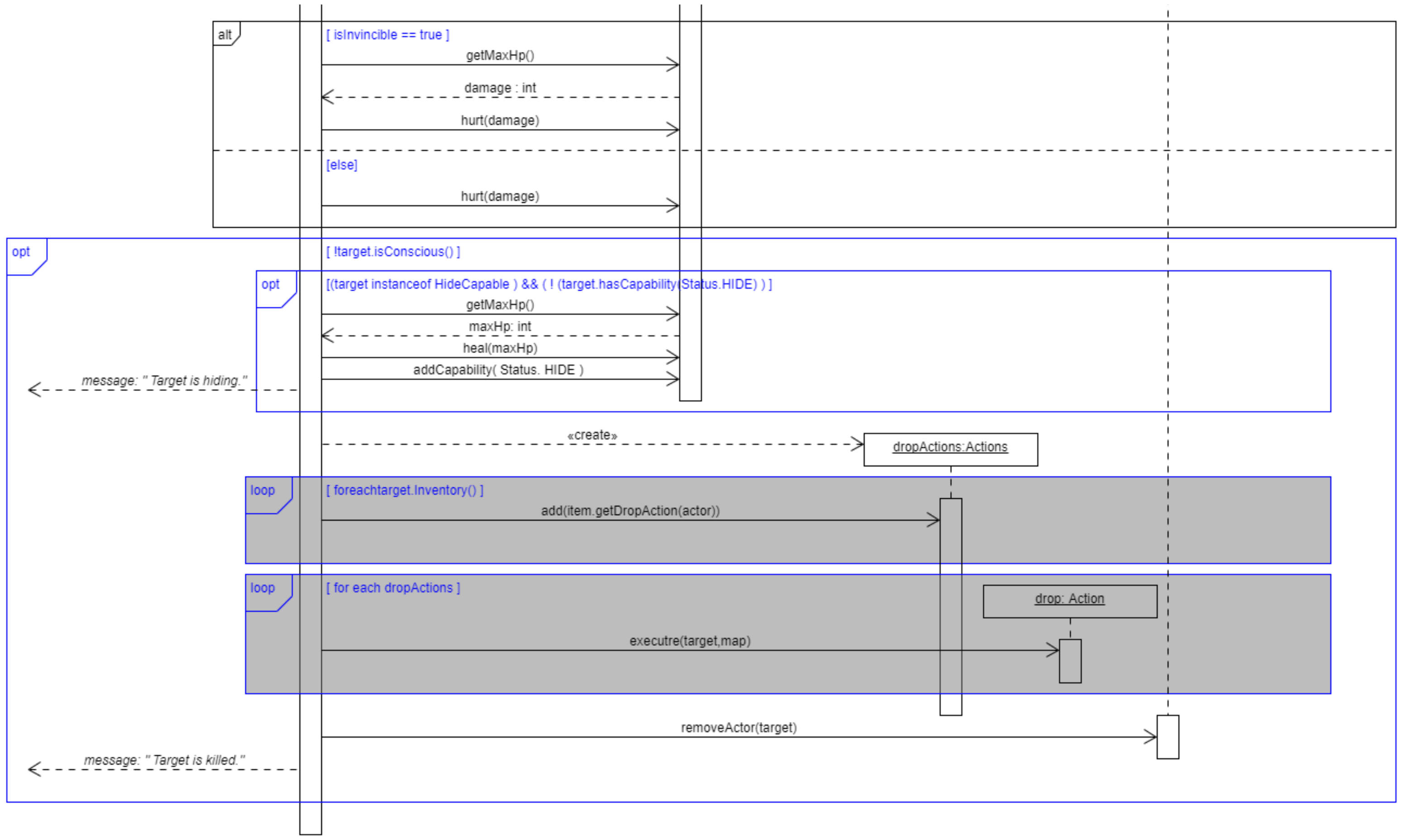
AttackBehaviour.getAction:



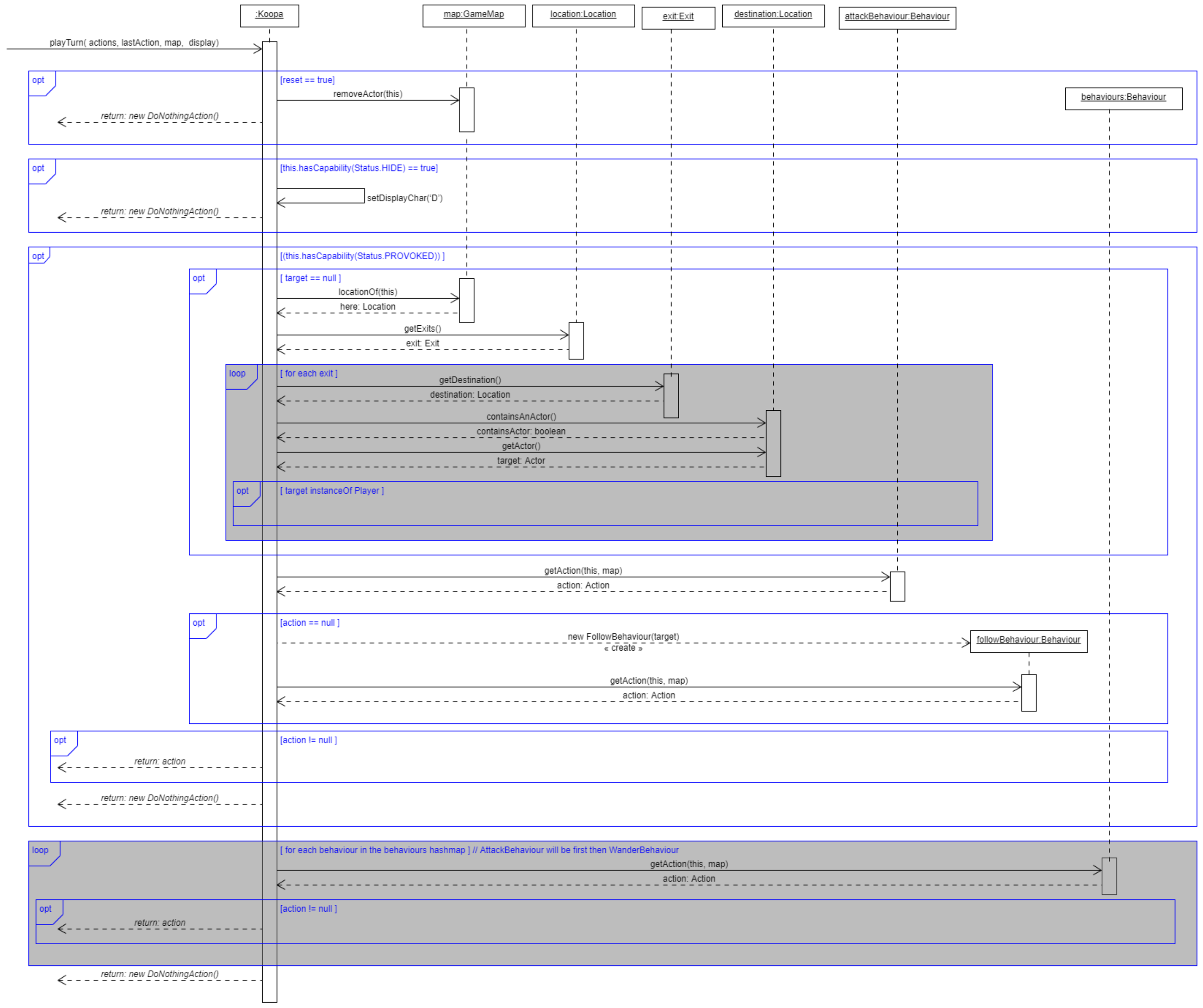
execute.AttackAction:



:AttackAction
actor:Actor
target:Actor
weapon:Weapon
map:GameMap
execute(actor, map)
hasCapability(Status.INVINCIBLE)
isInvincible: boolean
opt
[(isInvincible == true)]
message: " No Damage"
chanceToHit()
hitRate : int
getWeapon()
weapon : Weapon
damage()
damage : int
verb()
verb : String
opt
[!(rand.nextInt(100) <= hirtRate)]
message: " Actor misses target."
hasCapability(Status.PROVOKED)
isProvoked: boolean
opt
[!(target instanceof Player) & isProvoked != true]
addCapability(Status. PROVOKED)
hasCapability(Status.INVINCIBLE)
isInvincible : boolean
hasCapability(Status.TALL)
isTall : boolean
opt
[isTall == true]
removeCapability(Status.TALL)


alt
[isInvincible == true]
getMaxHp()
damage : int
hurt(damage)
[else]
hurt(damage)
opt
[!target.isConscious()]
opt
[(target instanceof HideCapable) && (! (target.hasCapability(Status.HIDE))]
getMaxHp()
maxHp: int
heal(maxHp)
addCapability(Status. HIDE)
message: " Target is hiding."
«create»
dropActions:Actions
loop
[foreachtarget.Inventory()]
add(item.getDropAction(actor))
loop
[for each dropActions]
drop: Action
executre(target,map)
removeActor(target)
message: " Target is killed."

Koopa.playTurn:



* For Goomba replace 2nd opt with 10% suicide calculation. Will removeActor(this) if true, then return a new DoNothingAction.

- Override CanActorEntor method for class Floor. Will only return true if Actor instanceof Player else false.
  - Will currently utilize instanceof Player for check. But if new actors can enter the floor a new interface will be added to mark them. Thus ensuring the open closed principle is followed.

- New class Wrench created, it will extend abstract class WeaponItem
  - Contains a 0 parameter constructor. Which initialises name, displayChar, damage, verb & hitRate to appropriate values.
  - Does not inherit from abstract class Item but WeaponItem to ensure Single Responsibility Principle is followed. Whereby abstract class WeaponItem will be responsible for all weapons.

- New class SuperMushroom created, it will extend abstract class Item.
  - Will contain one 0 parameter constructor. Which initialises name, displayChar & portable to appropriate values.
  - Inherits from abstract class Item to ensure Single Responsibility Principle is followed. Whereby abstract class Item will be responsible for all items.

- Modification to Enum class Status, adding enumeration of PROVOKED and HIDE.
  - Once PROVOKED Goomba/ Koopa will immediately retaliate in the same turn. Once PROVOKED Koopa will immediately retaliate in the same turn. HIDE used to indicate that Koopa has been defeated and now Dormant.
  - Maintains single responsibility principle for Status Enum class. As well as maintaining type safety and preventing any input errors.

- New interface HideCapable created and implemented by Koopa class. Will be utilised as a marking for actors capable of hiding after being attacked.
  - The marking ensures that in the AttackAction method the actor does not die immediately when killed instead implements Status.HIDE. Allowing for instance Koopa to become Dormant.
  - Implemented to ensure Open-closed Principle. Allowing for the addition of more enemies that are capable of hiding.

- AttackBehaviour class will be modified. Will be called when Goomba/ Koopa is PROVOKED or Player is in its surroundings.
  - getAction(Actor actor, GameMap map) ⇒ Will be overridden.
    - For every exit of the Actor's location. Check if an Actor is there & if the Actor is on ground Dirt & instanceOf Player. If yes, return new AttackAction(). Will use Exits of Actor's location to get the target and its direction for the new AttackAction.
    - Before returning attackAction. Check if Actor hasCapability(Status.PROVOKED).
      - If false, actor. addCapability(Status.PROVOKED)
    - Return null otherwise. If it is not possible to have return AttackAction.
  - Maintains single responsibility principle. In which attack action is implemented by AttackBehaviour instead of the Goomba/ Koopa playTurn method.

- FollowBehaviour implements interface Behaviour. Utilised by both Goomba and Koopa.
  - Implements code to follow a target (Player).

    - Only implemented by both Goomba and Koopa if PROVOKED. Additionally, will only be implemented if AttackBehaviour get action returns null, meaning that player is not within striking distance. This check is performed in Goomba & Koopa's respective playTurn() method.
    - This ensures that a GOD method playTurn() is not created and the method follows the single responsibility principle.
    - This further maintains the single responsibility principle for FollowBehaviour. In which move actor action (following actor) is implemented by FollowBehaviour instead of the Goomba/ Koopa playTurn method.

- Modification of execute() method in AttackAction class.
    - Check if the target hasCapability (Status.INVINCIBLE). If true returns " No Damage"
    - Check if the attack was successful [!(rand.nextInt(100) <= hitRate)]. Return appropriate statement if missed.
    - If the attack was successful. Method will perform a check for if target is not an instanceof Player & if target does not have Capability(Status. PROVOKED)
        - If true, meaning target != instance of Player and does not have Status. PROVOKED. It will addCapability(Status. PROVOKED)
    - Check if the target has Capabilities TALL. If true, remove that capability.
    - Check if the Actor has INVINCIBLE Capabilities. If yes, deal full damage to target by getting target.getMaxHp and dealing the MaxHp as its damage.
        - hurt(maxHp)
    - Else deal the target with the weapon's damage.
        - hurt(weapon.damage())
    - Check if (!target.isConscious()).
        - If true, then proceed to check (target instanceof HideCapable) && (! (target.hasCapability(Status.HIDE)).
            - If true, heal to maxHp
            - addCapability(Status.HIDE)
            - Return appropriate message about target hiding.
            - Check for target having capability Hide is done to ensure Koopa can only hide once.
    - Continue Previous implementation.

- New enemies abstract class. Inherits Actor. This is created to prevent DRY being violated.
    - Will only contain one method allowableActions()

- Modification to Goomba/ Koopa class. Extends enemies abstract class.

<u>Goomba Class:</u> Instance Variables →

- Map<Integer, Behaviour> behaviours = new HashMap<>() → Stores mapping of behaviour and their priority.
    - Only contains <1, AttackBehaviour > and < 10, WanderBehaviour >.
- Actor target = null → Stores the Actor object (Mario/ player) that attacked it. It is utilised to track location of player/ target, allowing the implementation of FollowBehaviour(). playTurn() method will be used to initialise it.

Methods → Goomba(), getIntrinsicWeapon(), playTurn(…), allowableActions(…)

- Goomba() ⇒ Constructor for class Goomba. Will set name, displayChar. hitPoints, initialise possible behaviours
    - this.behaviours.put(1, new AttackBehaviour())
    - this.behaviours.put(10, new WanderBehaviour())
- getIntrinsicWeapon() ⇒ Creates and returns an intrinsic weapon. It is an Overridden method inherited from the Actor class.
    - "Kick", 10
- playTurn(ActionList actions, Action lastAction, GameMap map, Display display) ⇒ Overrides the method in Actor class. Figures out what to do next.
- allowableActions(…) ⇒ No modifications made

PlayTurn(ActionList actions, Action lastAction, GameMap map, Display display):

- Method returns an action to be executed.
- Check if the 10% suicide possibility is met. If true.
    - map.removeActor(this)
    - return new DoNothingAction()
- Checks if Status.PROVOKED. If it is true, it will then check if the target instance variable != null. If == null, will initialised it by looping through Exits and finding the target (the instance variable will be empty only for the initial attack, thus target will be located in one of Goomba's adjacent squares)
    - It will then implement AttackBehaviour.
        - Action action = behaviours.get(1).getAction(this, map) ;
    - Check if action != null, if true, return action. This ensures if Goomba can hit the target it will do it instead of following target.
    - Else FollowBehavior will be implemented.
        - Behaviour behaviour = new FollowBehaviour(target)
        - Action action = behaviour.getAction(this, map)
    - Check if action == null. If true, a new DoNothingAction() will be returned, else action obtained by WanderBehaviour.getAction() will be returned.
- Else will loop through behaviours HashMap to get action.

- So either AttackBehaviour or WonderBehaviour will be implemented. If both behaviours return no actions, a new DoNothingAction() will be returned. AttackBehaviour will be called first, so Goomba will always hit the target if it is in its surroundings instead of Wondering. If AttackBehaviour.getAction() returns an action, add capability PROVOKED to this Goomba object.

- Creation of new concrete class Koopa which extends abstract class Actor

<u>Koopa Class implements HideCapable:</u>

Instance Variables →
- Map<Integer, Behaviour> behaviours = new HashMap<>() → Stores mapping of behaviour and their priority.
    - Only contains <1, AttackBehaviour > and < 10, WanderBehaviour >.
- Actor target = null → Stores the Actor object (Mario/ player) that attacked it. It is utilised to track location of player/ target, allowing the implementation of FollowBehaviour. playTurn() method will be used to initialise it.

Methods → Koopa(), getIntrinsicWeapon(), playTurn(…), allowableActions(…)

- Koopa() ⇒ Constructor for class Koopa. Will set name, displayChar. hitPoints, initialise possible behaviours and add a SuperMushRoom to its inventory.
    - this.addItemToInventory(new SuperMushroom())
    - this.behaviours.put(1, new AttackBehaviour())
    - this.behaviours.put(10, new WanderBehaviour())
- getIntrinsicWeapon() ⇒ Creates and returns an intrinsic weapon. It is an Overridden method inherited from the Actor class.
    - “Punch” , 30
- playTurn(ActionList actions, Action lastAction, GameMap map, Display display) ⇒ Overrides the method in Actor class. Figure out what to do next.
- allowableActions(…) ⇒ Actions that can be performed on Koopa.
    - Check if otherActor.getWeapon() instance of Wrench & this.hasCapability(Status.HIDE) & (otherActor.hasCapability(Status.HOSTILE_TO_ENEMY). If true actions.add(new AttackAction(this,direction)).
        - Allows attack on Koopa with Wrench when Koopa is Dormant.
    - Check if this.hasCapability(Status.HIDE) == false & (otherActor.hasCapability(Status.HOSTILE_TO_ENEMY). If true If true actions.add(new AttackAction(this,direction)).
        - Prevents attack with weapons other than wrench when Koopa is Dormant.

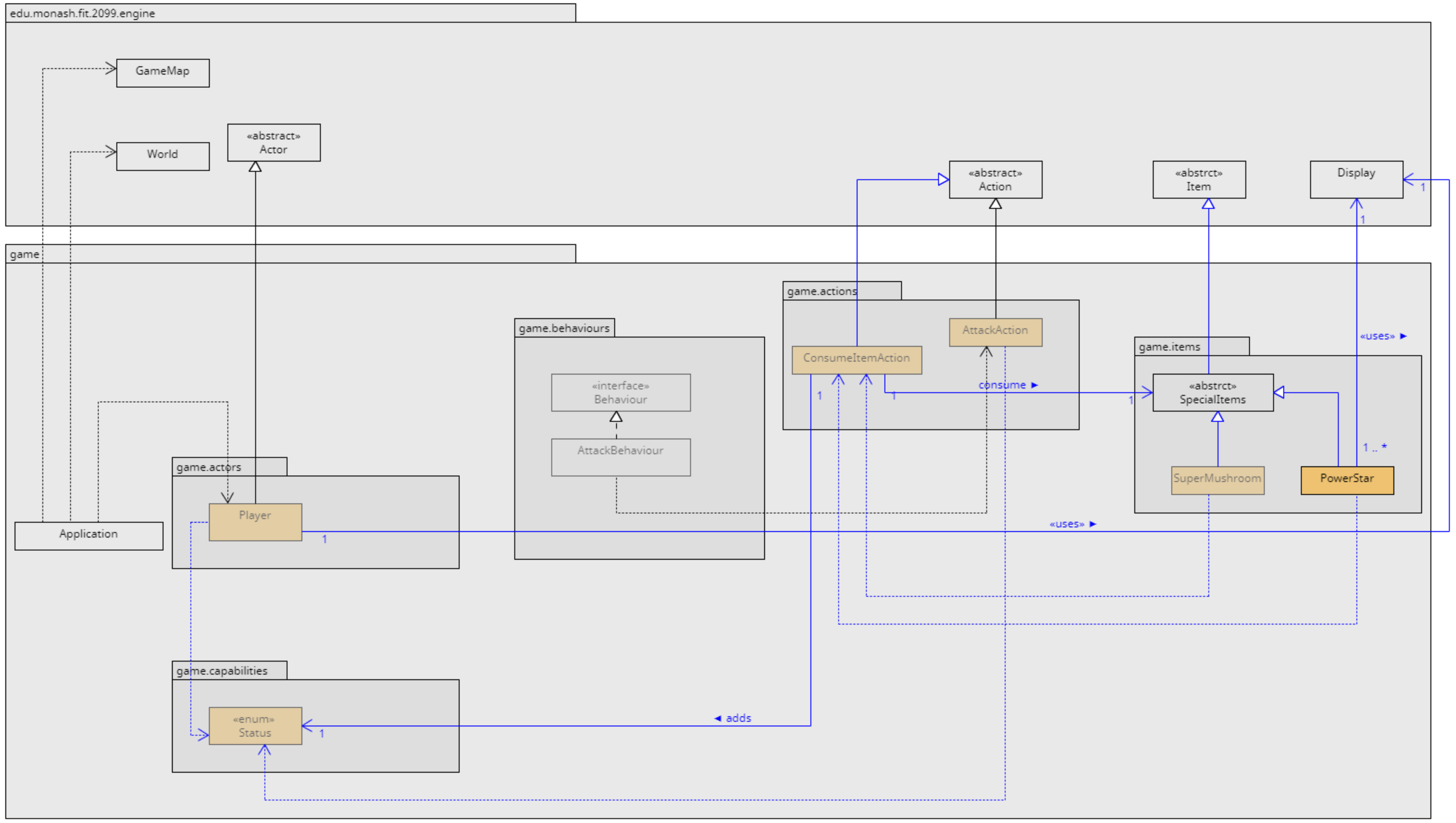
---

playTurn(ActionList actions, Action lastAction, GameMap map, Display display)

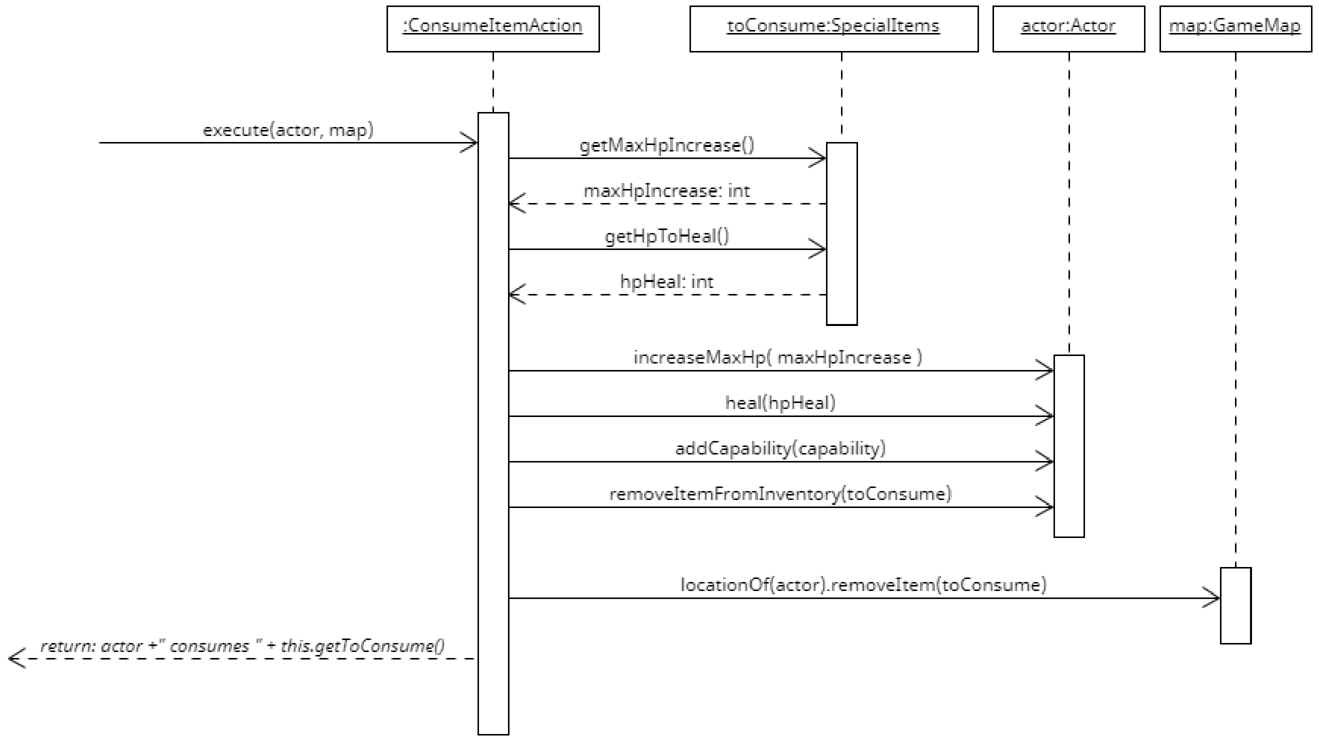
- Method returns an action to be executed.
- Checks if this.hasCapability(Status.HIDE). If it does…

    - this.setDisplayChar(‘D’)
    - return new DoNothingAction()
- Checks if Status.PROVOKED. If it is true, it will then check if the target instance variable != null. If == null, will initialised it by looping through Exits and finding the target (the instance variable will be empty only for the initial attack, thus target will be located in one of Koopa’s adjacent squares)
    - It will then implement AttackBehaviour.
        - Action action = behaviours.get(1).getAction(this, map) ;
    - Check if action != null, if true, return action. This ensures if Koopa can hit the target it will do it instead of following target.
    - Else FollowBehavior will be implemented.
        - Behaviour behaviour = new FollowBehaviour(target)
        - Action action = behaviour.getAction(this, map)
    - Check if action == null. If true, a new DoNothingAction() will be returned, else action obtained by FollowBehaviour.getAction() will be returned.
- Else will loop through behaviours HashMap to get action.
    - So either AttackBehaviour or WonderBehaviour will be implemented. If both behaviours return no actions, a new DoNothingAction() will be returned. AttackBehaviour will be called first, so Koopa will always hit the target if it is in its surroundings instead of Wondering. If AttackBehaviour.getAction() returns an action, add capability PROVOKED to this Koopa object.

Assignment 1 - Req4: Magical Items



execute.ConsumeItemAction:


:ConsumeItemAction
toConsume:SpecialItems
actor:Actor
map:GameMap
execute(actor, map)
getMaxHpIncrease()
maxHpIncrease: int
getHpToHeal()
hpHeal: int
increaseMaxHp(maxHpIncrease)
heal(hpHeal)
addCapability(capability)
removeItemFromInventory(toConsume)
locationOf(actor).removeItem(toConsume)
return: actor +" consumes " + this.getToConsume()

- New abstract class SpecialItems inheriting Item.
    - Contains 2 abstract methods. getHpToHeal() and getMaxHpIncrease().
        - getInsintricDamageIncrease() can be included if a new item performs this action.
    - The abstract class is created to ensure the open closed principle is followed. Allowing for the prevention of instanceof checks in the consumeItemAction to determine whether to increase max hp or heal it.

- A SuperMushroom class and PowerStar class will be created. Both will extend abstract class SpecialItems.
    - Inherits from abstract class SpacialItems to ensure Single Responsibility Principle is followed. Whereby abstract class SpecialItems will be responsible for all special items (capable of changing actor's statistics).
    - Will override the 2 abstract methods. Returning appropriate int values. (0 is no changes to be made and > 0 if there are)
    - For PowerStar class the 2 tick() methods will implement the fading feature of Powerstar. (On a ground/ In player's Inventory) It will also have a static Display display = new Display(). Utilised in outputting the number of remaining turns for PowerStar.

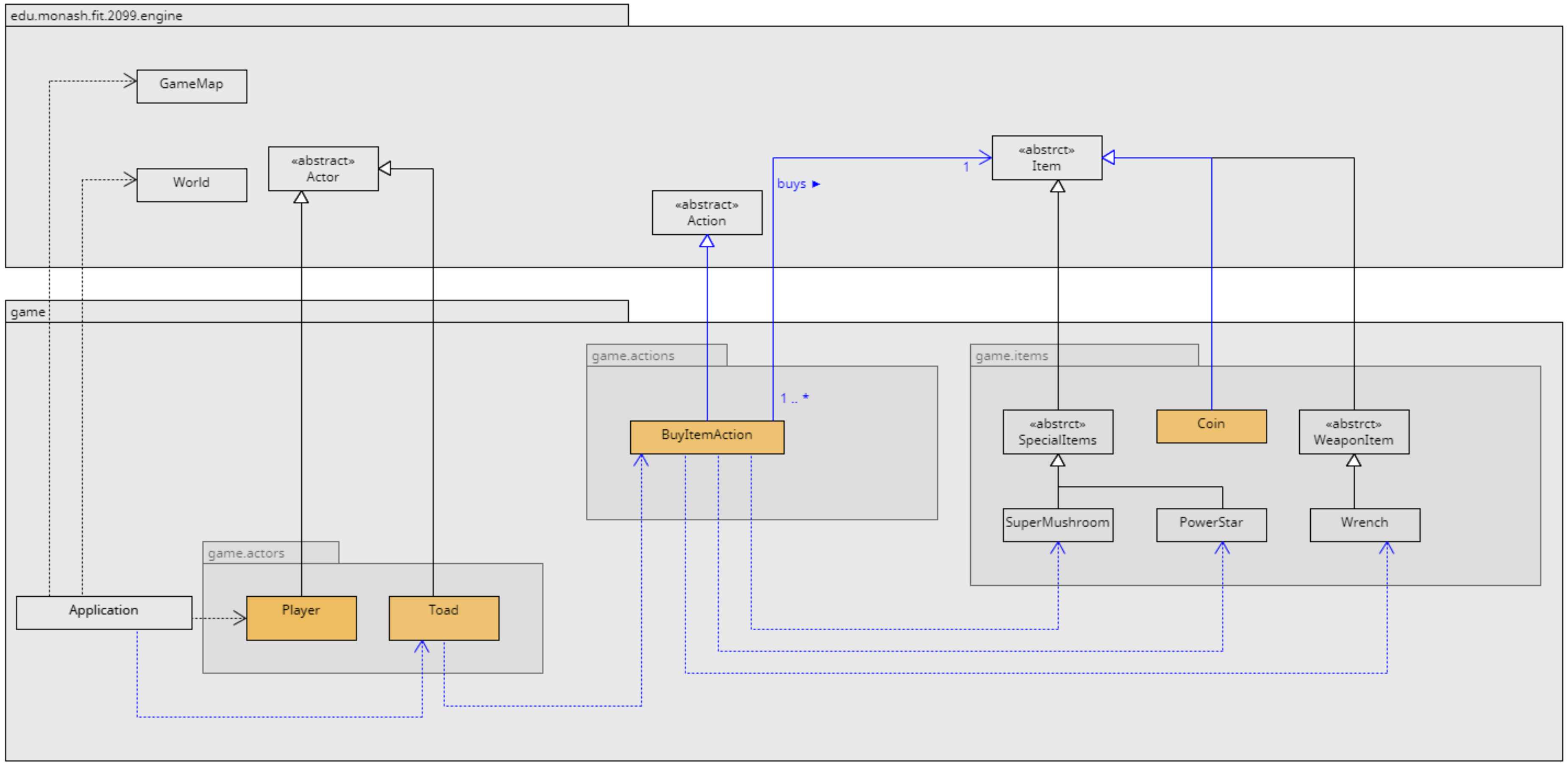
- Add a new enum to Status named INVINCIBLE.
    - TALL = capability gained from the consumption of SuperMushroom.
    - INVINCIBLE = capability gained from the consumption of PowerStar.
    - Maintains single responsibility principle for Status Enum class. As well as maintaining type safety and preventing any input error.

- New Class ConsumeItemAction which extends abstract class Action. All consumable items (special items) will << create >> this action.
    - All concrete classes that inherit the abstract Action class have the necessary methods to provide written text before (menuDescription) and after the interaction executes.
        - execute(Actor actor, GameMap map) ⇒ Evaluates the result of the consumption.
        - menuDescription(Actor actor) ⇒ menuDescription will output the required menu display.
    - When the ConsumeItemAction is created using its constructor, information such as the special item to be consumed and its capability will be passed. This is done to ensure the open closed principle. Whereby we do not need to check the type of Item actor is consuming to determine the course of action.
    - Inside the execute method it contains …
        - actor.increaseMaxHp(toConsume.getMaxHpIncrease());
        - actor.heal(toConsume.getHpToHeal());
        - actor.addCapability(capability);
    - Meaning there will be no use of instance of for type checking and no multiple if statements.

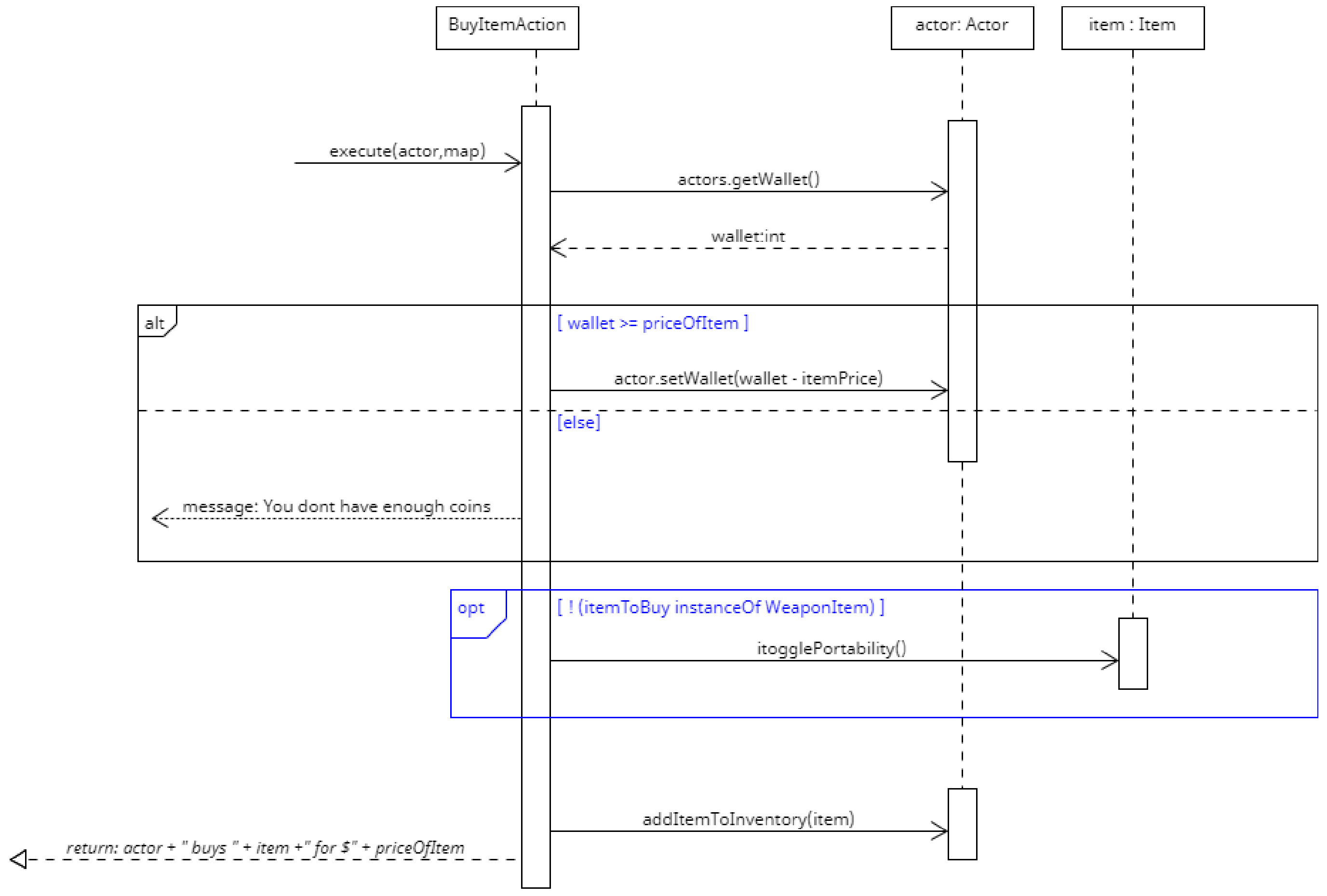
- Modification of Player class. New method invincible.
    - Manages capabilities gained by Player when PowerStar is consumed as well as the fading method after its consumption.
    - Called in playTurn().

- Maintains single responsibility principle for playTurn method. And prevents it from being a GOD method.

Assignment 1 - Req5:Trading



execute.BuyItemAction:


BuyItemAction
actor: Actor
item : Item
execute(actor,map)
actors.getWallet()
wallet:int
alt
[wallet >= priceOfItem]
actor.setWallet(wallet - itemPrice)
[else]
message: You dont have enough coins
opt
[! (itemToBuy instanceOf WeaponItem)]
itogglePortability()
addItemToInventory(item)
return: actor + " buys " + item +" for $" + priceOfItem

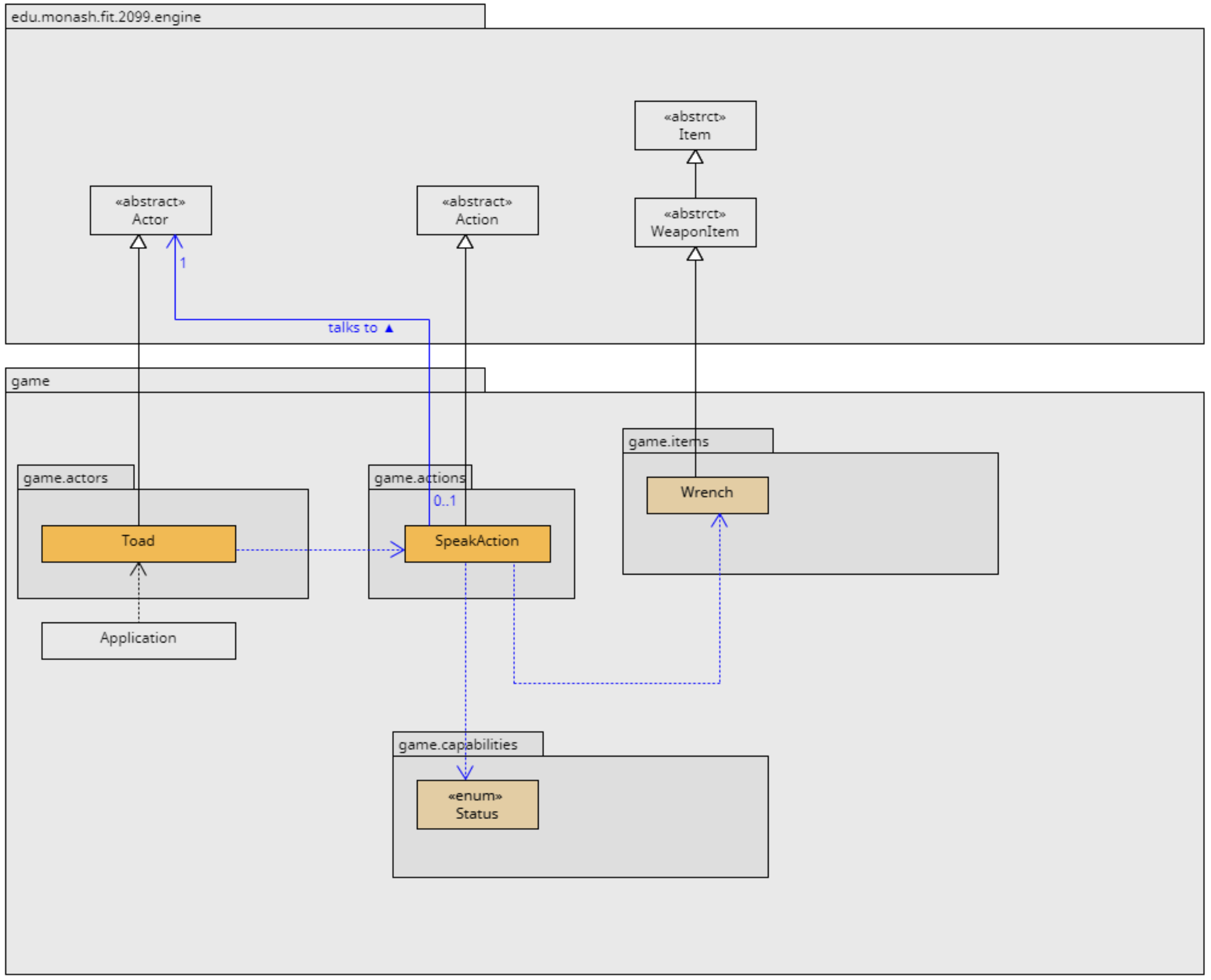
- Modification of Player class.
    - New instance variable int wallet. Stores the total value of all coins collected by Player. Initialised to 0 in its declaration.
    - New method updateWallet(). Called in playTurn().
        - Checks the player's inventory for coins and adds it to the wallet.
    - Maintains single responsibility principle for playTurn method. And prevents it from being a GOD method.

- New Class Coin extends abstract class Item.
    - Will have an instance variable int value. Which stores the value of this Coin object. Will have a constructor method. Will take one parameter, that being the value of the Coin. Sets name, displayChar, portability and value of Coin
    - Contains method, getValue() which returns value of Coin object.
    - Inherits from abstract class Item to ensure Single Responsibility Principle is followed. Whereby abstract class Item will be responsible for all items.

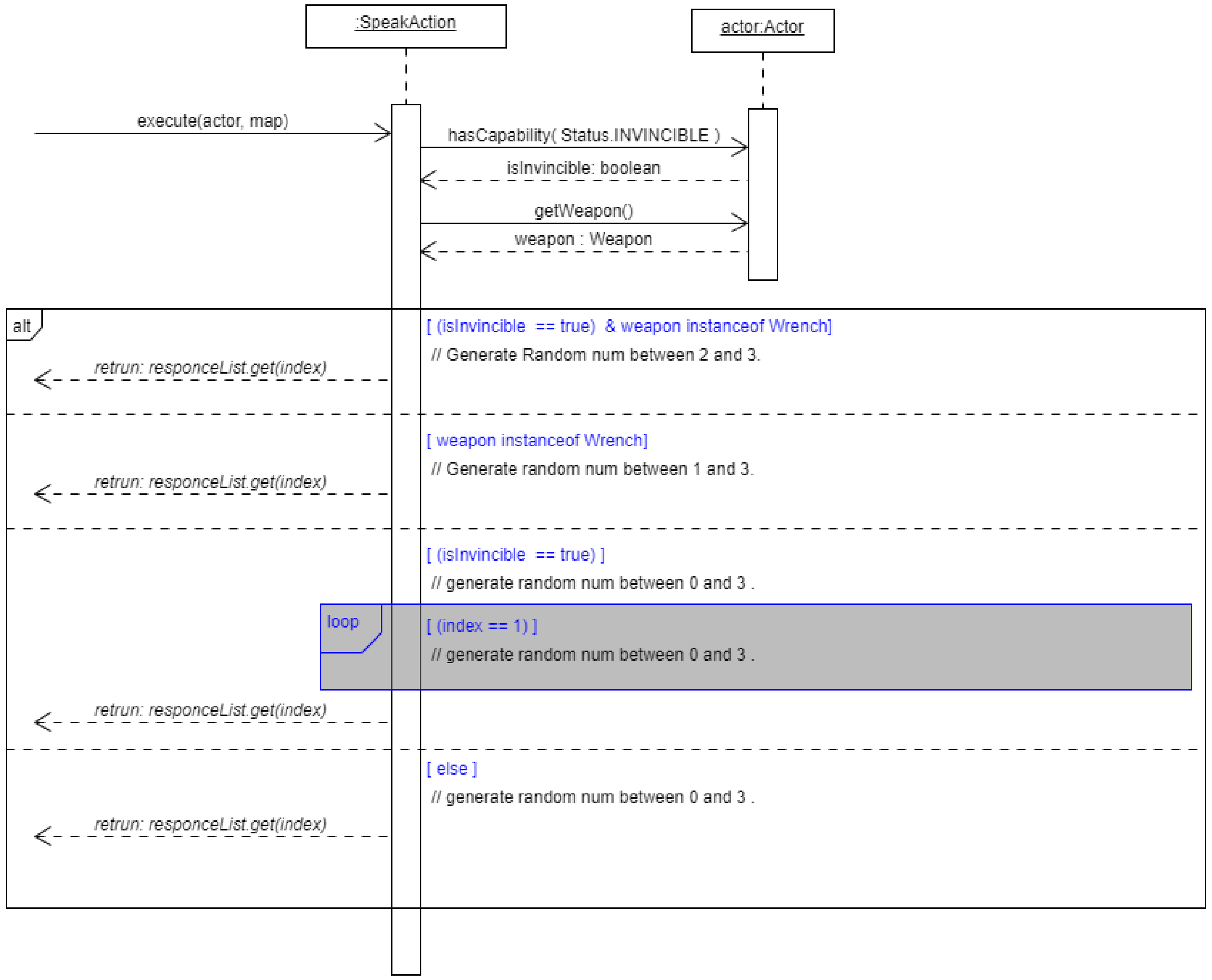
- New Class Toad which extends abstract class Actor.
    - Toad is placed in the centre of the map, surrounded by walls which are manually placed. So changes in application will need to be made.
    - Inherits from abstract class Actor to ensure Single Responsibility Principle is followed. Whereby abstract class actor will be responsible for all actors.
    - Toad << creates >> the BuyItemAction in its allowableActions method.
        - 3 new BuyItemAction for the 3 Items of SuperMushroom, PowerStar & Wrench.

- New BuyItemAction class extends the abstract class Action.
    - All concrete classes that inherit the abstract Action class have the necessary methods to provide written text before (menuDescription) and after the interaction executes.
        - execute(Actor actor, GameMap map) ⇒ Evaluates the result of the transaction.
        - menuDescription(Actor actor) ⇒ menuDescription will output the required menu display.
    - Will check if the player's wallet has the required amount to buy the items. Return appropriate string representing the result of the transaction.
    - buyFromActor, item to be bought and its price is passed by its constructor.

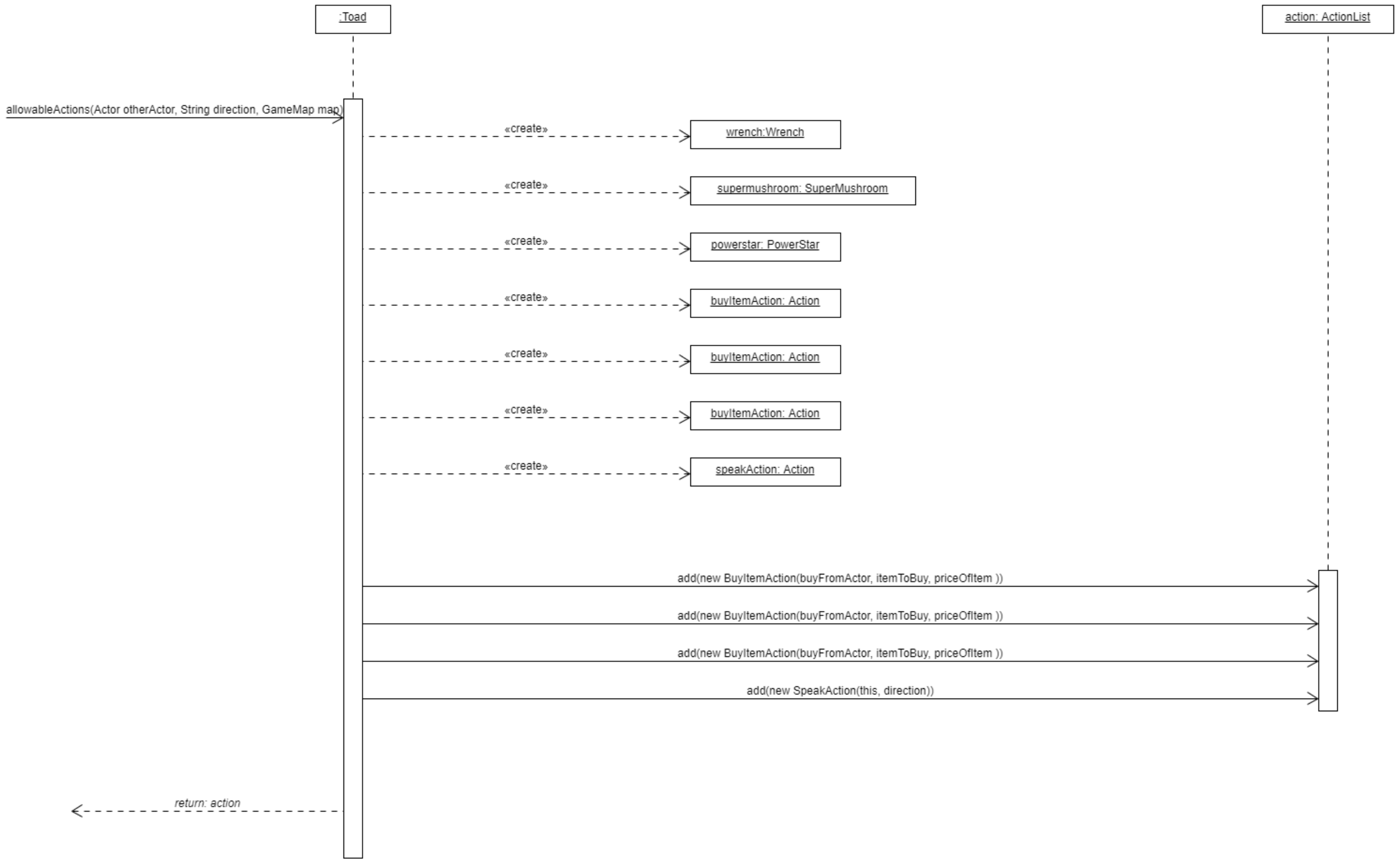
Assignment 1- REQ6: Monologue.



execute.speakAction:


:SpeakAction
actor:Actor
execute(actor, map)
hasCapability(Status.INVINCIBLE)
isInvincible: boolean
getWeapon()
weapon : Weapon
alt
[(isInvincible == true) & weapon instanceof Wrench]
// Generate Random num between 2 and 3.
retrun: responceList.get(index)
[weapon instanceof Wrench]
// Generate random num between 1 and 3.
retrun: responceList.get(index)
[(isInvincible == true)]
// generate random num between 0 and 3 .
loop
[(index == 1)]
// generate random num between 0 and 3 .
retrun: responceList.get(index)
[else]
// generate random num between 0 and 3 .
retrun: responceList.get(index)

toad.allowableActions:



:Toad
action: ActionList
allowableActions(Actor otherActor, String direction, GameMap map)
«create»
wrench:Wrench
«create»
supermushroom: SuperMushroom
«create»
powerstar: PowerStar
«create»
buyItemAction: Action
«create»
buyItemAction: Action
«create»
buyItemAction: Action
«create»
speakAction: Action
add(new BuyItemAction(buyFromActor, itemToBuy, priceOfItem))
add(new BuyItemAction(buyFromActor, itemToBuy, priceOfItem))
add(new BuyItemAction(buyFromActor, itemToBuy, priceOfItem))
add(new SpeakAction(this, direction))
return: action

- Modification of Toad Class.
    - Modification of AllowableActions(). Adding a new SpeakAction to actions ActionList after adding the 3 BuyItemAction().
        - actions.add(new SpeakAction(Actor talkToTarget))

- New class SpeakAction extends abstract class Action.

<u>SpeakAction Class:</u>

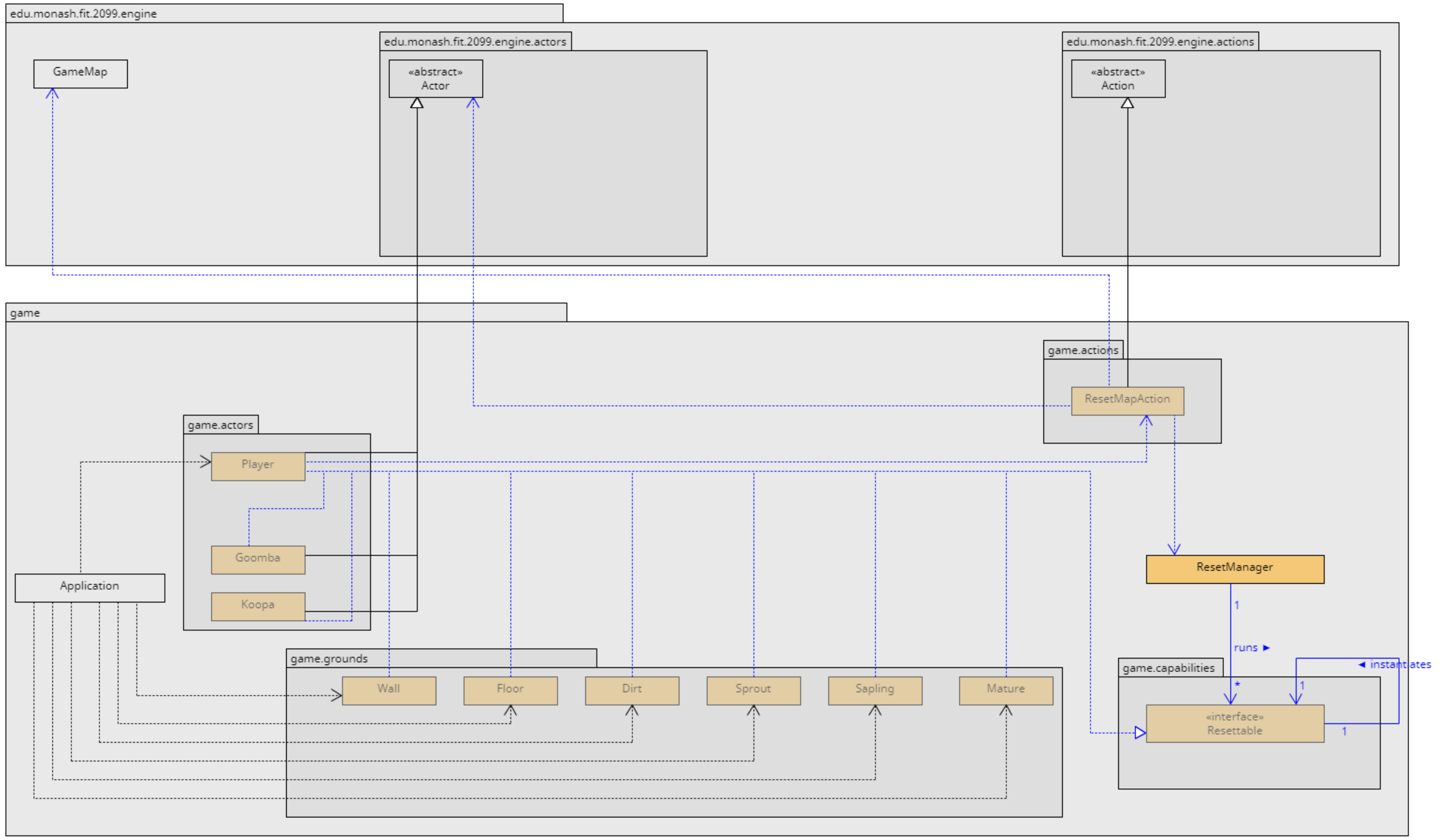
Instance Variables → Actor talkToTarget, Random rand = new Random()
- talkToTarget stores the Actor in which player will talk to

Methods → SpeakAction(...), execute(...), menuDescription(...)

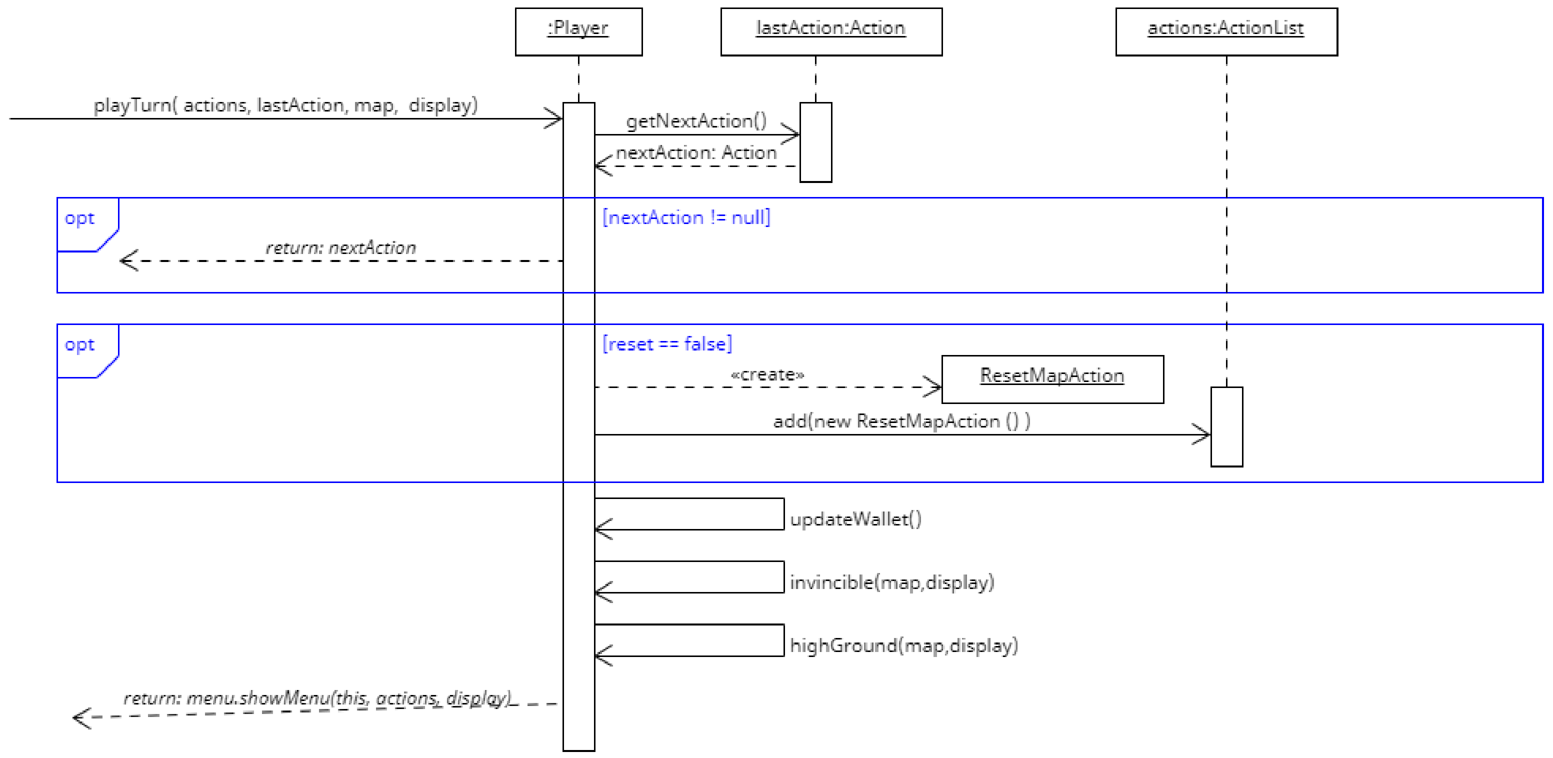
- SpeakAction (Actor talkToTarget) ⇒ Constructor for SpeakAction class. Will set instance variable talkToTarget.
- execute(Actor actor, GameMap map) ⇒ Returns appropriate message.
    - Stores the 4 Strings into an arrayList response..
        - 0: "You might need a wrench to smash Koopa's hard shells."
        - 1: "The Princess is depending on you! You are our only hope."
        - 2: "Being imprisoned in these walls can drive a fungus crazy :("
        - 3: "You better get back to finding the Power Stars."
    - Check if actor has Capability INVINCIBLE and actor.getWeapon instanceof Wrench.
        - If true. Generate random index between 1 and 2. And return response.get(index)
    - Elif check if actor.getWeapon instanceof Wrench
        - If true. Generate random index between 1 and 3. And return response.get(index)
    - Elif check if actor has Capability INVINCIBLE
        - If true. Generate random index between 0 and 2. And return response.get(index)
    - Else generate random index between 0 and 3. And return response.get(index)
- menuDescription(Actor actor) ⇒ what to display for the menu.
    - Return actor + " talks with" talkToTarget.

Random index between a range generated by index = rand.nextInt(max - min + 1) + min
- max = maximum value and min = minimum value.

Assignment 1 - REQ7: Reset Game



Player.playTurn:



- Interface Resettable will be implemented by all resettable objects.
    - Acts as a marking of all resettable objects.
    - Ensures that the open close principle will be met. Allowing the addition of more ressetable objects.

- ResetManager Class manages all resettable objects.Will only have one instance of it for the whole game.
    - Modification of run() ⇒ loop through all ressetable objects in the resettableList and call their resetInstance() method
    - Modification of appendResetInstance(Resettable reset) ⇒ Add ressetable objects to resettableList
        - resettableList.add(reset)
    - Modification of cleanUp(Resettable resettable ⇒ Remove ressetable objects from resettableList
        - resettableList.remove(resettable)

- Create a new ResetMapAction Class which extends abstract class Action.
    - Ensure single responsibility principle is followed. All actions should inherit from the abstract class action.

<u>ResetMapAction Class:</u>

Attributes → String hotKey
- hotKey stores hot key for implementing the action

Methods → ResetMapAction(), execute(Actor actor, GameMap map), menuDescription(Actor actor)

- ResetMapAction() ⇒ Constructor for ResetMapAction Class. Will have no parameters. Initialises hotKey to “r”
- execute(Actor actor, GameMap map) ⇒ Executes action. Will call ResetManager.getInstance().run(). And return appropriate String stating Game has been reset.
- menuDescription(Actor actor) ⇒ method for returning appropriate message on menu.
    - Return “ Reset the game.”

- Modification of Player Class. Will now implement interface Ressetable. It also << creates >> a ResetMapAction.
    - New instance variable boolean reset = false. Stores if the Player object has been called to reset.
    - Modification of Constructor ⇒
        - Will now include this.registerInstance(), which adds this player object to the ressetableList in ResetManager
    - Modification of playTurn() method ⇒
        - Check if (lastAction.getNextAction() != null). If true return lastAction.getNextAction();
        - Check if reset == false. If true, add a new ResetMapAction to the passed ActionList actions. This allows the ResetMapAction to be displayed on the menu.

        - Then continue the previous implementation.
    - Override resetInstance() method ⇒
        - Set instance variable reset to true.
        - Remove the capability of TALL and INVISIBLE from the player.
        - getMaxHp and heal(MaxHp)

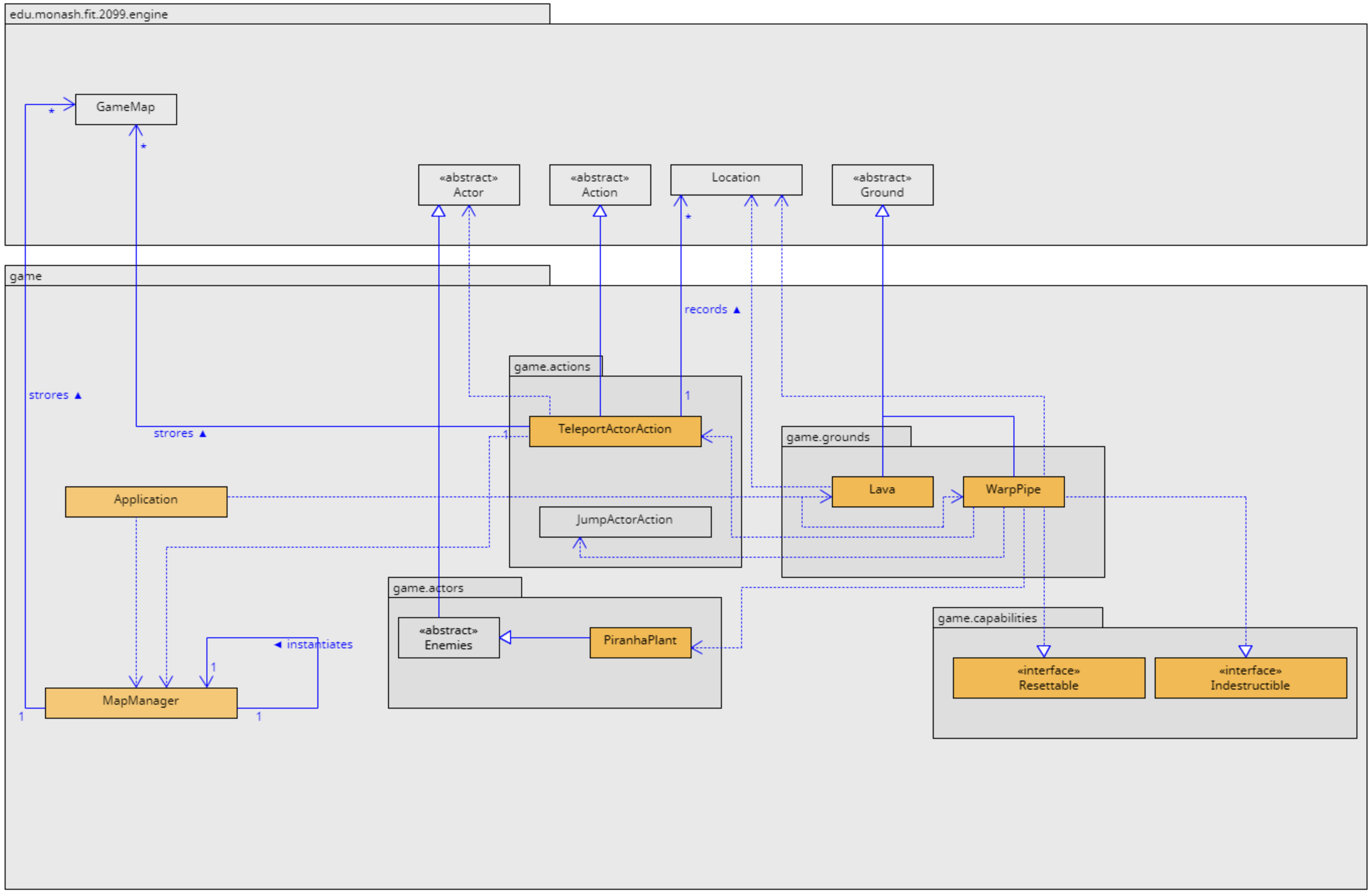
- Modification of Sprout, Sapling & Mature Classes. Will now implement interface Ressetable. Changes will apply and be the same for all.
    - New instance variables. boolean reset = false & boolean hasResseted = false.
        - reset stores whether resetInstance() method have been called.
        - Stores if this object has already been reset
    - Modification of Constructor ⇒
        - Will now include this.registerInstance(), which adds this object to the ressetableList in ResetManager
    - Override resetInstance() method ⇒
        - Sets instance variable reset to true.
    - Modification of tick(Location location) method ⇒
        - When the method is now called. It will now start by checking if [(this.reset == true) & (!hasResetted)]. If true …
            - Check if (random.nextInt(100) <= 50). If true, set current location to Dirt.
                - location.setGround(new Dirt())
            - Loop through all Item objects stored at the current location's inventory. And check if they are instanceOf Coin. If true, remove the Item object from location's inventory.
            - Set hasRessetted to true. This ensures reset of objects are only done once.
        - Then continue previous implementation.

- Modification of Dirt, Floor and Wall clases. Changes will apply and be the same for all.
    - New instance variables. boolean reset = false & boolean hasResseted = false.
        - reset stores whether resetInstance() method have been called.
        - resetCount stores num of time resetInstance() method have been called.
    - Modification of Constructor ⇒
        - Will now include this.registerInstance(), which adds this object to the ressetableList in ResetManager
    - Override resetInstance() method ⇒
        - Sets instance variable reset to true.
    - Override tick(Location location) method from abstract class Ground ⇒
        - Check if [(this.reset == true) & !hasResetted)]. If true …

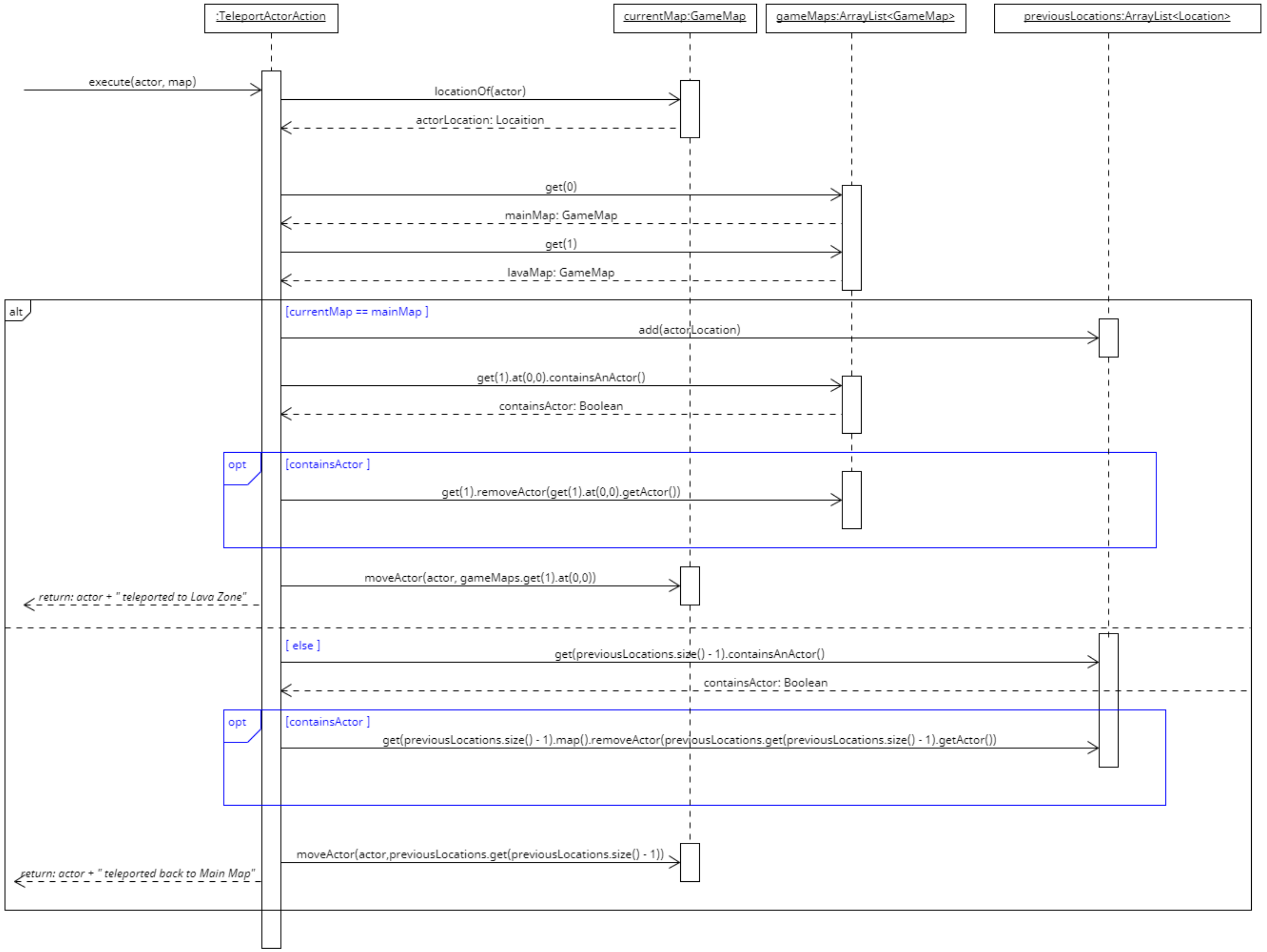
        - Loop through all Item objects stored at the current location's inventory. And check if they are instanceOf Coin. If true, remove the Item object from location's inventory.
        - Set hasRessetted to true. This ensures reset of objects are only done once.

- Modification of Goomba and Koopa class. Changes will apply and be the same for both.
    - New instance variables. boolean reset = false
        - reset stores whether resetInstance() method have been called.
    - Modification of Constructor ⇒
        - Will now include this.registerInstance(), which adds this object to the ressetableList in ResetManager
    - Override resetInstance() method ⇒
        - Sets instance variable reset to true.
    - Modification of playTurn(ActionList actions, Action lastAction, GameMap map, Display display) ⇒
        - When the method is now called. It will now start by checking (this.reset == true). If true …
            - remove actor from map.
                - map.removeActor(this)
            - return new DoNothingAction()
        - Then continue previous implementations.

Assignment 3 - REQ 1: New Map & Teleportation (Warp Pipe)



execute.TeleportActorAction:


:TeleportActorAction
currentMap:GameMap
gameMaps:ArrayList<GameMap>
previousLocations:ArrayList<Location>
execute(actor, map)
locationOf(actor)
actorLocation: Locaition
get(0)
mainMap: GameMap
get(1)
lavaMap: GameMap
alt
[currentMap == mainMap]
add(actorLocation)
get(1).at(0,0).containsAnActor()
containsActor: Boolean
opt
[containsActor]
get(1).removeActor(get(1).at(0,0).getActor())
moveActor(actor, gameMaps.get(1).at(0,0))
return: actor + " teleported to Lava Zone"
[else]
get(previousLocations.size() - 1).containsAnActor()
containsActor: Boolean
opt
[containsActor]
get(previousLocations.size() - 1).map().removeActor(previousLocations.get(previousLocations.size() - 1).getActor())
moveActor(actor,previousLocations.get(previousLocations.size() - 1))
return: actor + " teleported back to Main Map"

- New Class Lava extending abstract class Ground.
    - Inherits from ground class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all grounds are managed by abstract class ground.
    - Newly created dealDamage() method is responsible for dealing damage to an actor standing on its location. This is called by tick(), meaning the dealDamage method is called every turn.
        - tick() does not deal the damage or check if an actor is at its current location to prevent the creation of a GOD method (possible if more features are added in the future). It also ensures that the single responsibility principle is upheld. Where tick() is utilised in ensuring that the Lava object experiences the passage of time and dealDamage() responsible for hurting the actor on its location.

- Modification of application class.
    - Creation of a new map and adding it to the world.
    - Adding the new Lava class to the groundFactory.

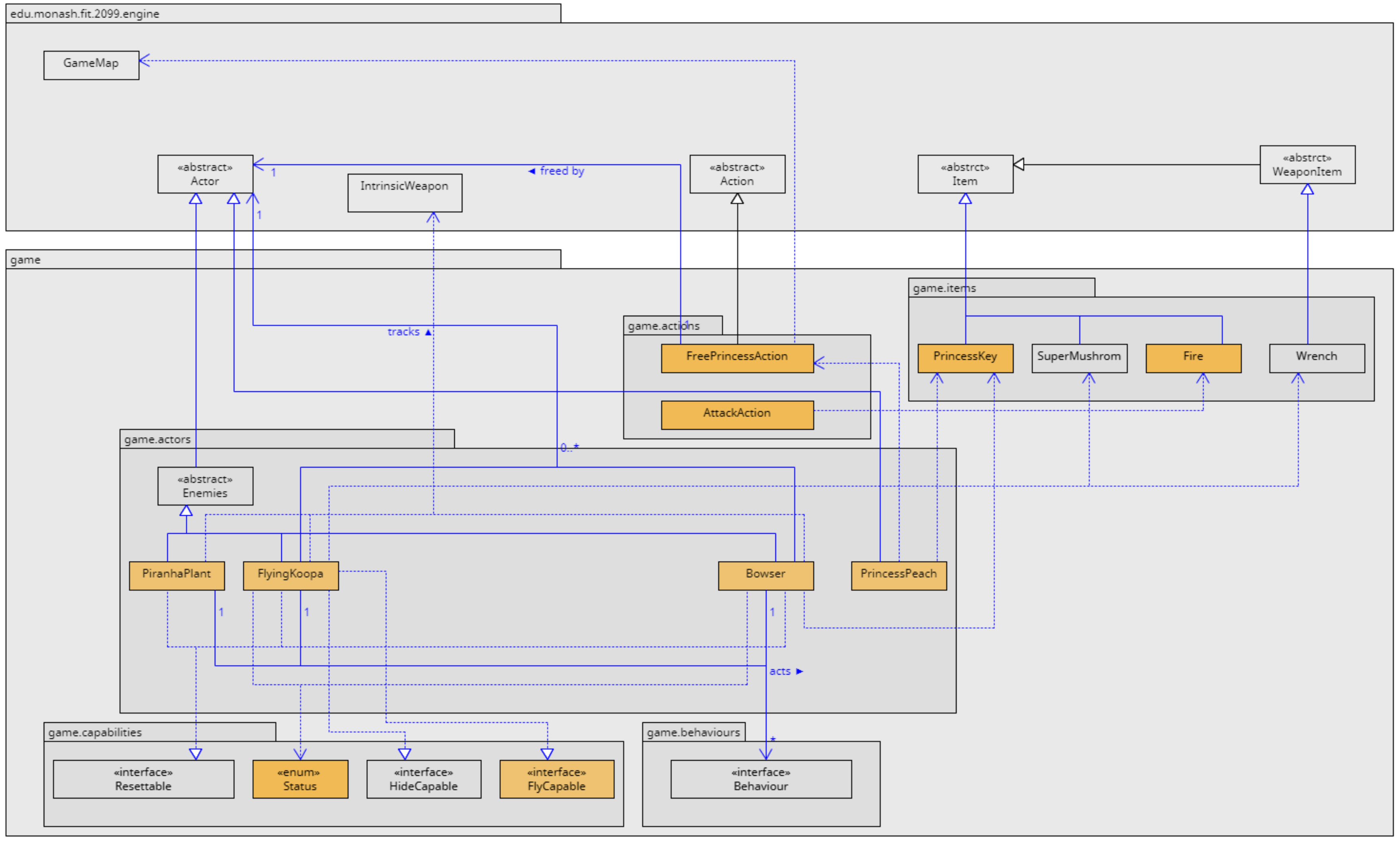
- New Class MapManager. Utilising a static factory method. This class is responsible for storing all maps created in our Mario game using an arrayList.
    - This creation of this class stems from the inability to modify World class in engine. Thereby, there is no access to the known maps.
    - The maps are appended in the Application class With MapManager.getInstance().appendMap (gameMap);
    - Static factory method is used as only one MapManager object is needed. This thus removes the need for the constant creation and deletion of the object.
    - MapManages allows for the Open Closed Principle to be maintained as new maps can be easily appended with zero to no modifications.

- New TeleportActorAction class. Extending abstract class action.
    - Inherits from Action class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all actions are managed by abstract class Action
    - TeleportActorAction will contain overriden execute() and menuDescription() methods.
    - Its constructor will pass the Location of the object which called it.
    - Will utilise MapManager instance in execute() method in determining which map to jump to.
    - A Location arrayList will be utilised in storing previous teleport from location.

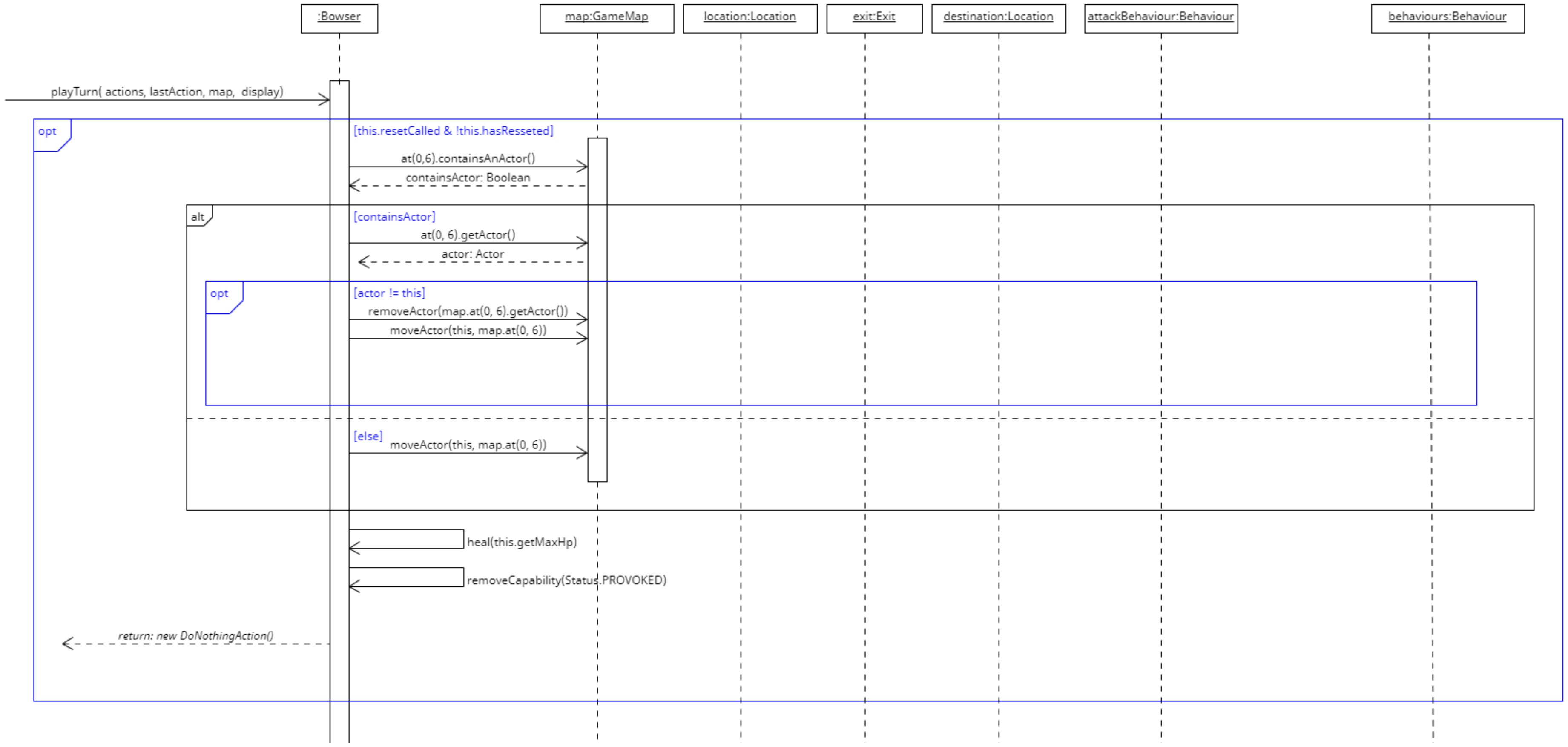
- New WarpPipe Class. Extending abstract class Ground implementing Resettable, Indestructible
    - Inherits from Ground class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all grounds are managed by abstract class Ground
    - Implements Ressetable, acting as a marking and ensuring that it is able to spawn a Piranha Plant again after reset.
    - Implements Indestructible as it acts as a marking as grounds that can’t be destroyed when Player is INVINCIBLE. This is done to maintain the open closed principle. Removing the need of multiple instanceOf checks and if statements.

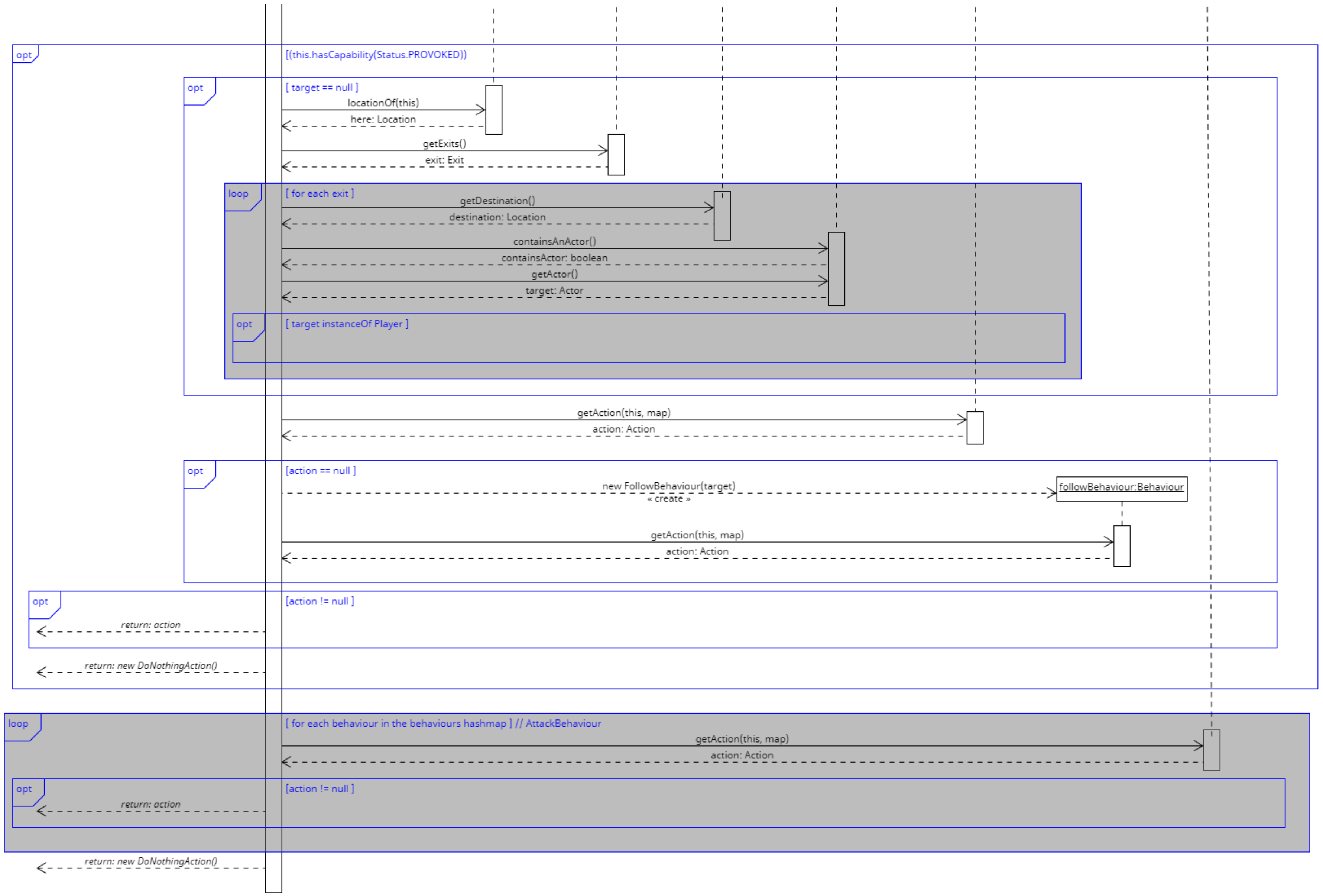
- tick() method will call spawnPiranhaPlant() method when one turn has passed. This is done to reduce the possibility of tick() becoming a GOD method in the future if more features are added. It is also done to maintain the single responsibility principle.
- allowableActions << creates >> JumpActorAction and TeleportActorAction.
    - Actions created here to remove risk of GOD method in Player playTurn and to reduce dependencies in Player class.

Assignment 3 - REQ 2: More allies and enemies!



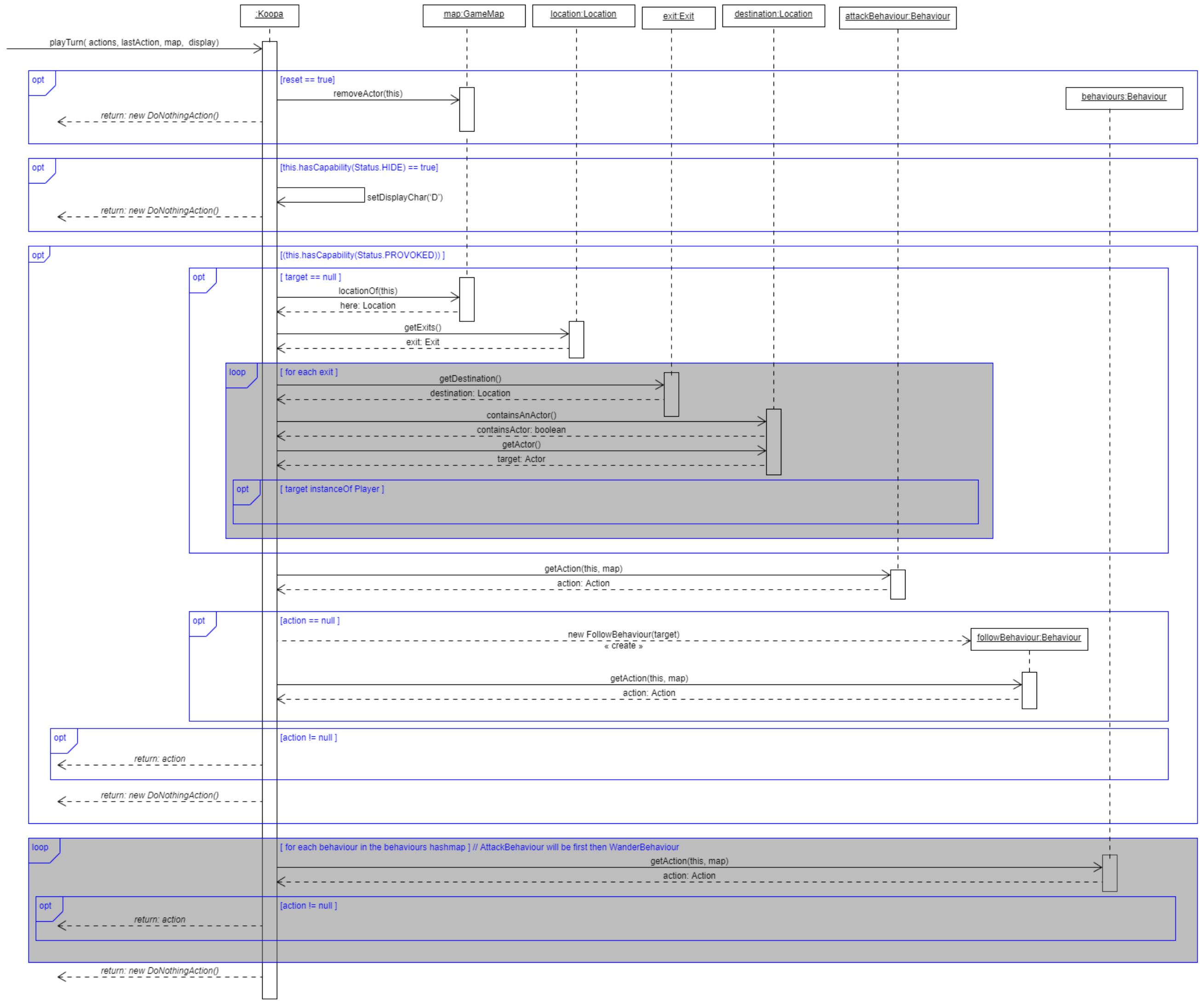
Bowser.playTurn():


:Bowser
map:GameMap
location:Location
exit:Exit
destination:Location
attackBehaviour:Behaviour
behaviours:Behaviour
playTurn(actions, lastAction, map, display)
opt
[this.resetCalled & !this.hasResseted]
at(0,6).containsAnActor()
containsActor: Boolean
alt
[containsActor]
at(0, 6).getActor()
actor: Actor
opt
[actor != this]
removeActor(map.at(0, 6).getActor())
moveActor(this, map.at(0, 6))
[else]
moveActor(this, map.at(0, 6))
heal(this.getMaxHp)
removeCapability(Status.PROVOKED)
return: new DoNothingAction()

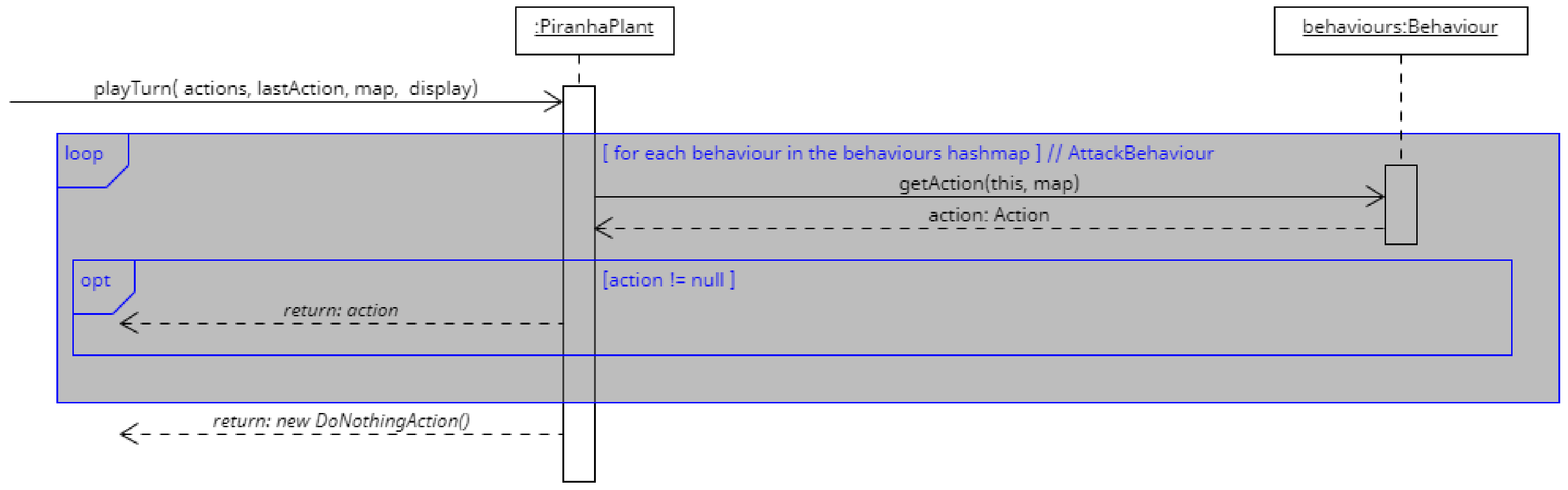


opt
[(this.hasCapability(Status.PROVOKED))
opt
[target == null]
locationOf(this)
here: Location
getExits()
exit: Exit
loop
[for each exit]
getDestination()
destination: Location
containsAnActor()
containsActor: boolean
getActor()
target: Actor
opt
[target instanceOf Player]
getAction(this, map)
action: Action
opt
[action == null]
new FollowBehaviour(target)
« create »
followBehaviour:Behaviour
getAction(this, map)
action: Action
opt
[action != null]
return: action
return: new DoNothingAction()
loop
[for each behaviour in the behaviours hashmap] // AttackBehaviour
getAction(this, map)
action: Action
opt
[action != null]
return: action
return: new DoNothingAction()

FlyingKoopa.playTurn():



PiranhaPlant.playTurn():


:PiranhaPlant
behaviours:Behaviour
playTurn(actions, lastAction, map, display)
loop
[for each behaviour in the behaviours hashmap] // AttackBehaviour
getAction(this, map)
action: Action
opt
[action != null]
return: action
return: new DoNothingAction()

- New class PrincessKey. Extending abstract class Item.
    - Inherits from Item class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all items are managed by abstract class item

- New class FreePrincessAction. Extending abstract class action.
    - Inherits from Action class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all actions are managed by abstract class Action
    - Contain overriden execute() and menuDescription() methods.
    - execute() method will remove actor from map and return appropriate string stating the ending of the game.
    - menuDescription() returns a description of the free princess action suitable to display in the menu.

- New class PrincessPeach. Extending abstract class Actor.
    - Inherits from Actor class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all actors are managed by abstract class Actor
    - allowableActions() method << creates new >> FreePrincessAction if the other actor has the princess key.
    - Checks for the princess key is done by a separate method. Which returns true if the princess key is found else false. This is done to maintain the single responsibility principle.

- Modification of Enum Status.
    - New FIREATTACK. Representing the capability of attacking with fire.

- New class Bowser. Extending abstract class Enemies. Implements Ressetable.
    - Inherits from Enemies class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all enemies are managed by abstract class Enemies.
    - Implements AttackBehaviour and FollowBehaviour under playTurn(), contains item PrincessKey and has capability FIREATTACK.
    - << creates >> new Insintric Weapon.
    - Implements Ressetable, acting as a marking and ensuring that it returns to its starting position when reset.

- New class PiranhaPlant. Extending abstract class Enemies. Implements Ressetable.
    - Inherits from Enemies class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all enemies are managed by abstract class Enemies.
    - Implements AttackBehaviour under playTurn().
    - << creates >> new Insintric Weapon.

    - Implements Ressetable, acting as a marking and ensuring that it increases its maxHp by 50 when reset.

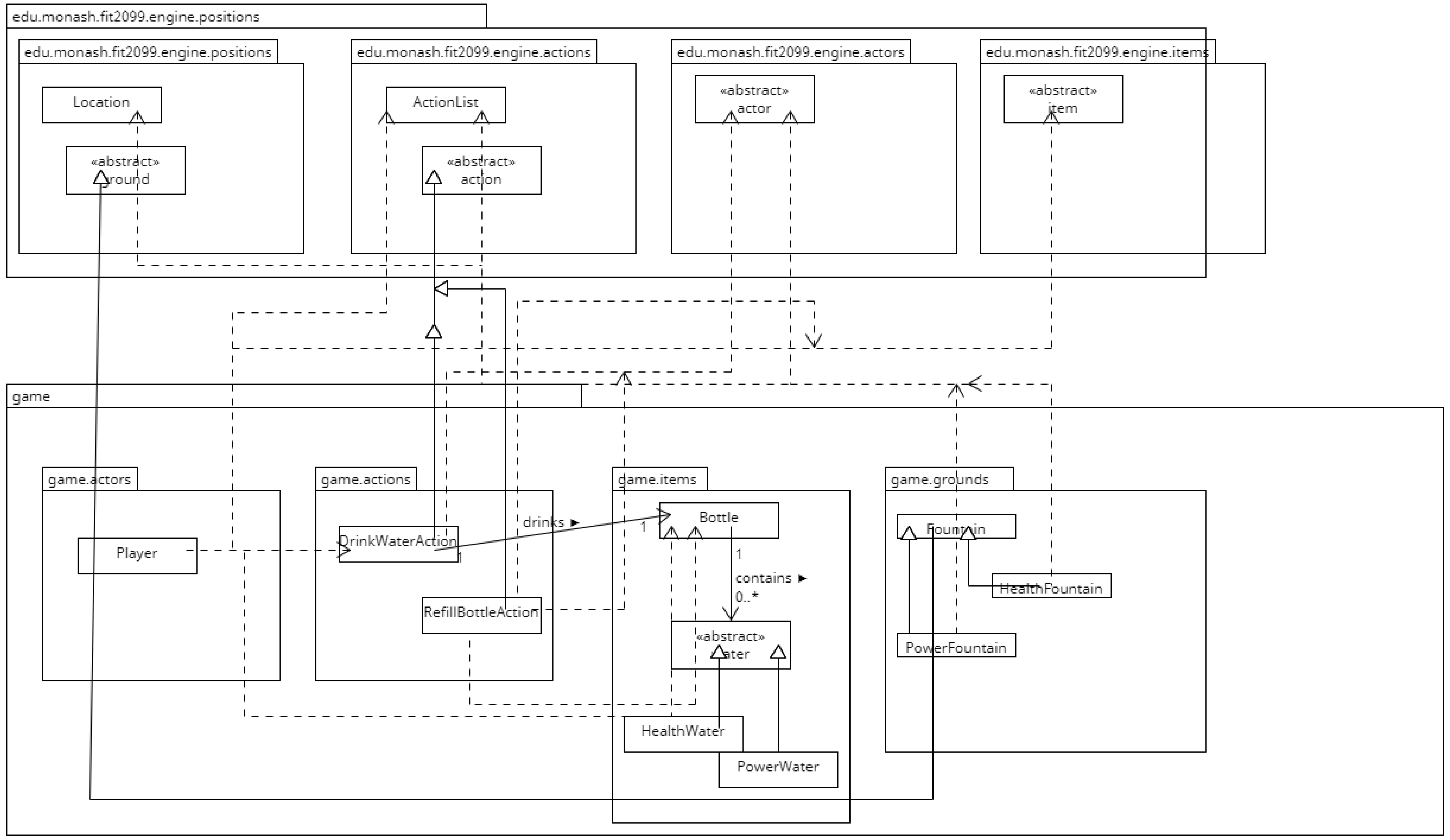
- New interface FlyCapable. Marking for all actors that are able to fly. Meaning that they are able to pass through high grounds.
    - This is done to maintain the open closed principle when checking canActorEnter for highgrounds. Removing the need for multiple instance of checks and if statements.

- New class FlyingKoopa. Extending abstract class Enemies. Implements Ressetable, FlyCapable, HideCapable.
    - Inherits from Enemies class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all enemies are managed by abstract class Enemies.
    - Implements AttackBehaviour, FollowBehaviour and WanderBehaviour under playTurn(). Contains a SuperMushroom Item.
    - << creates >> new Insintric Weapon.
    - allowableActions() will check if other actor has WeaponItem Wrench if it is Dormant else no AttackAction() will be returned.
    - Implements Ressetable, acting as a marking and ensuring that it kills itself when reset.
    - Implement HideCapable to ensure it is able to become Dormant.

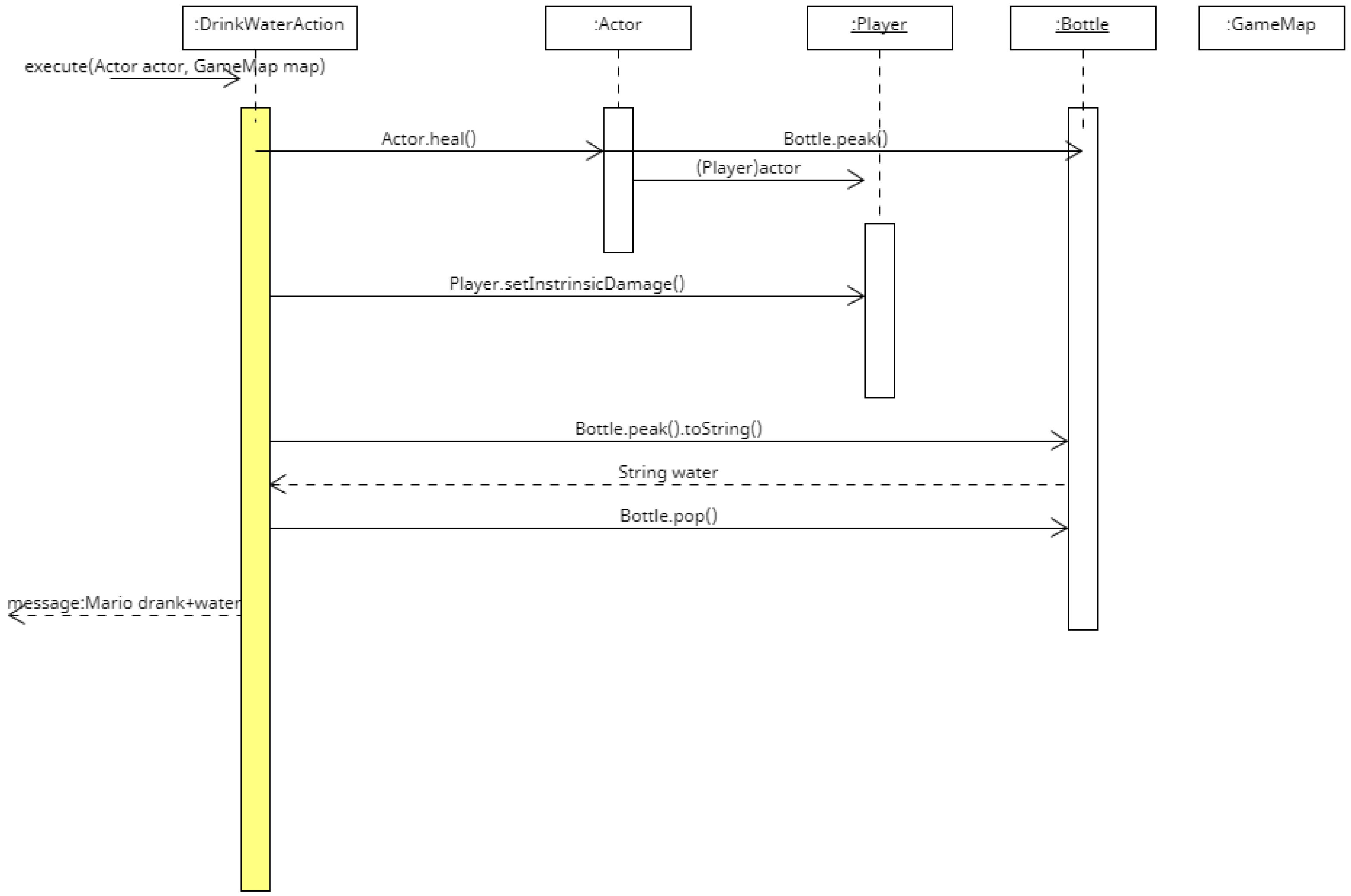
- New class Fire(). Extends Item.
    - Inherits from Item class to ensure DRY is not violated and to ensure Single Responsibility Principle is followed. Whereby all items are managed by abstract class Item.
    - Newly created dealDamage() method is responsible for dealing damage to an actor standing on its location. This is called by tick(), meaning the dealDamage method is called every turn.
        - tick() does not deal the damage or check if an actor is at its current location to prevent the creation of a GOD method (possible if more features are added in the future). It also ensures that the single responsibility principle is upheld. Where tick() is utilised in ensuring that the Lava object experiences the passage of time and dealDamage() responsible for hurting the actor on its location.
    - tick() is responsible for removing the item from the map after 3 turns.

- Modification of AttackAction class.
    - Will now check for capability FIREATTACK. If present, appropriate return string and menu description will be returned. As well as adding a Fire item to a target's location.
    - This is done here to maintain the single responsibility principle. Where the AttackAction class should be responsible for all attacking actions.
    - This is further done to reduce dependencies of classes that can gain the FIREATTACK capability. (no new relationship with FIRE class).

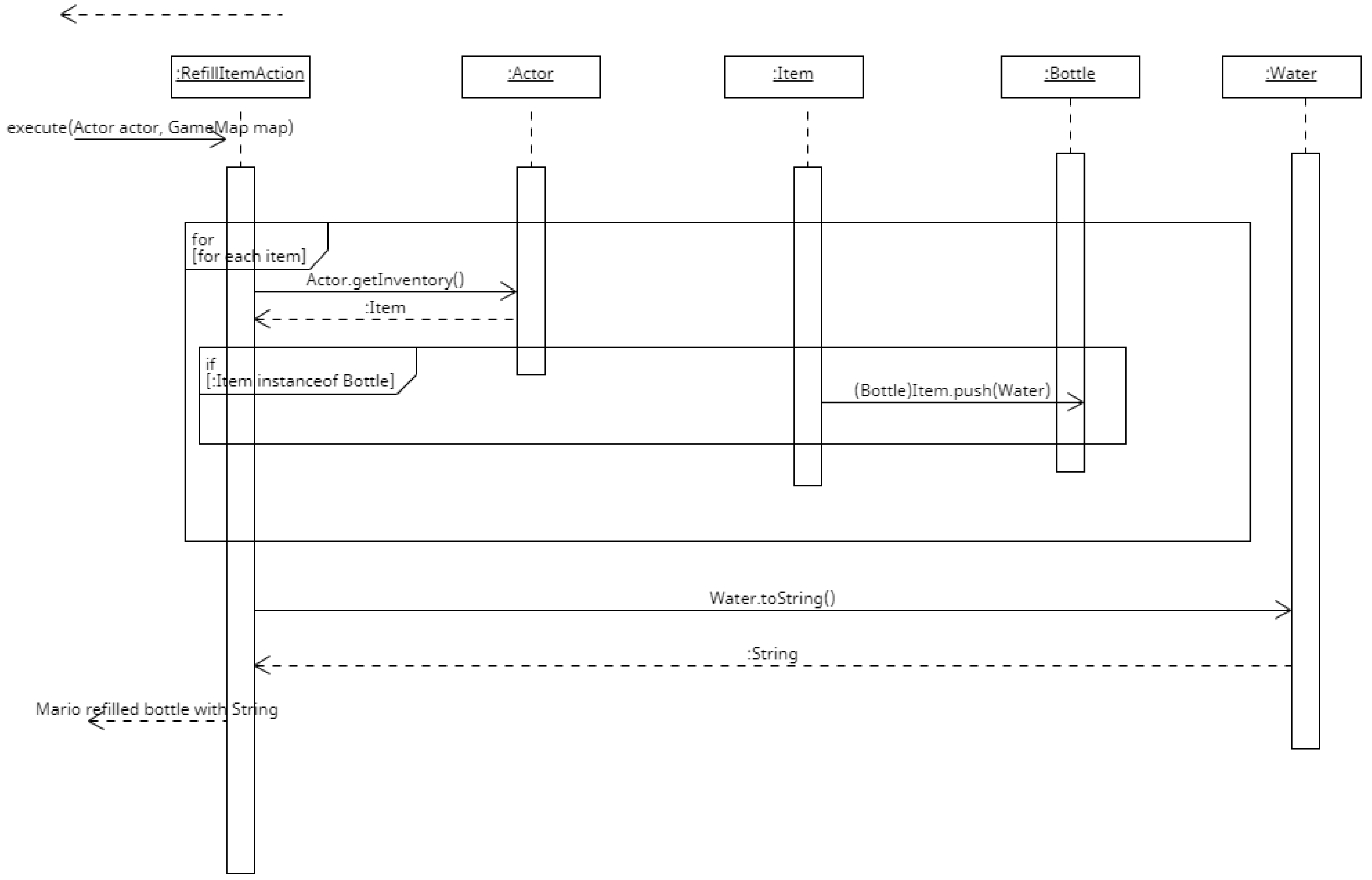
Assignment 3 - REQ 3: Magical fountain



execute.DrinkWaterAction:

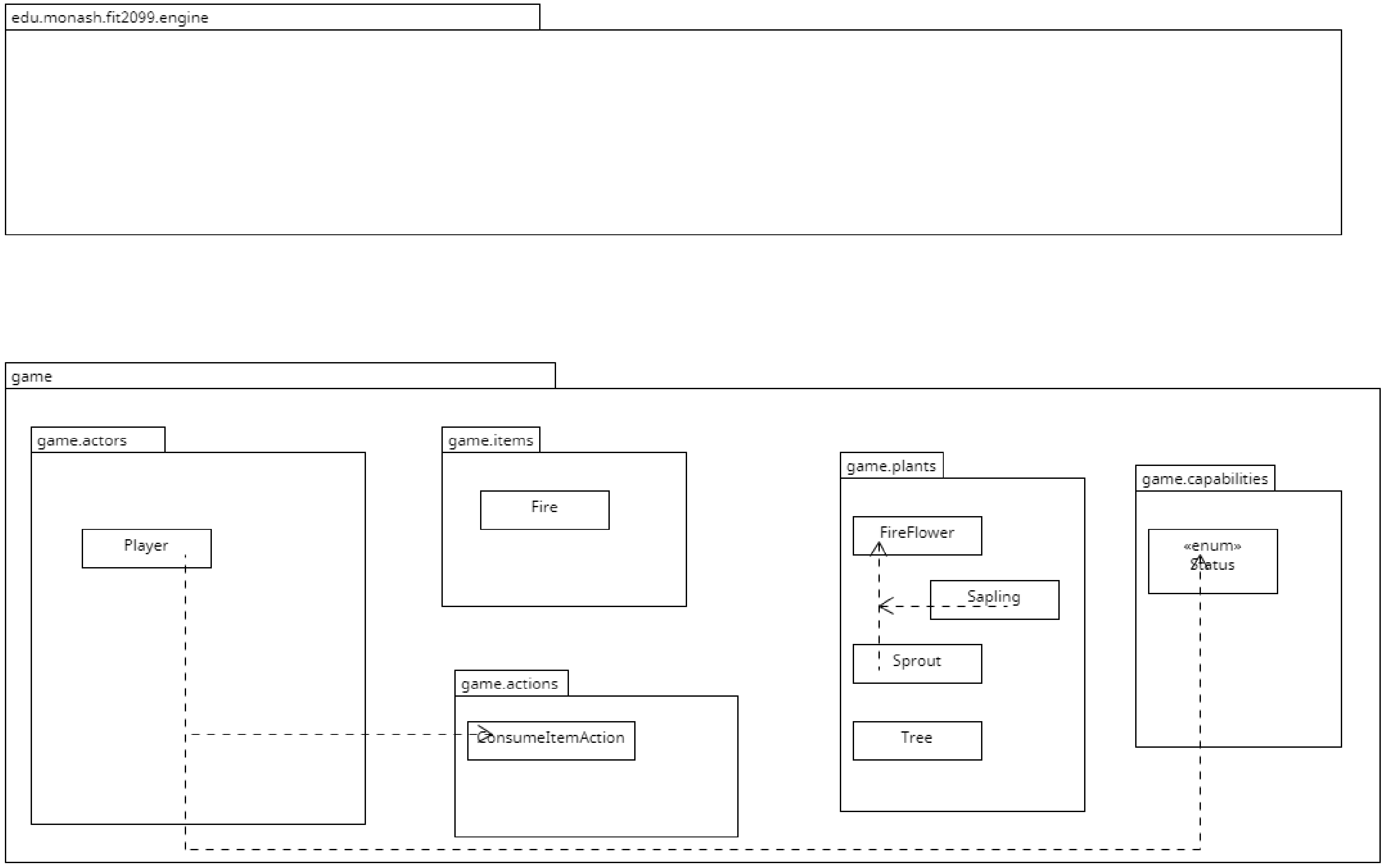


execute.RefillBottleAction:

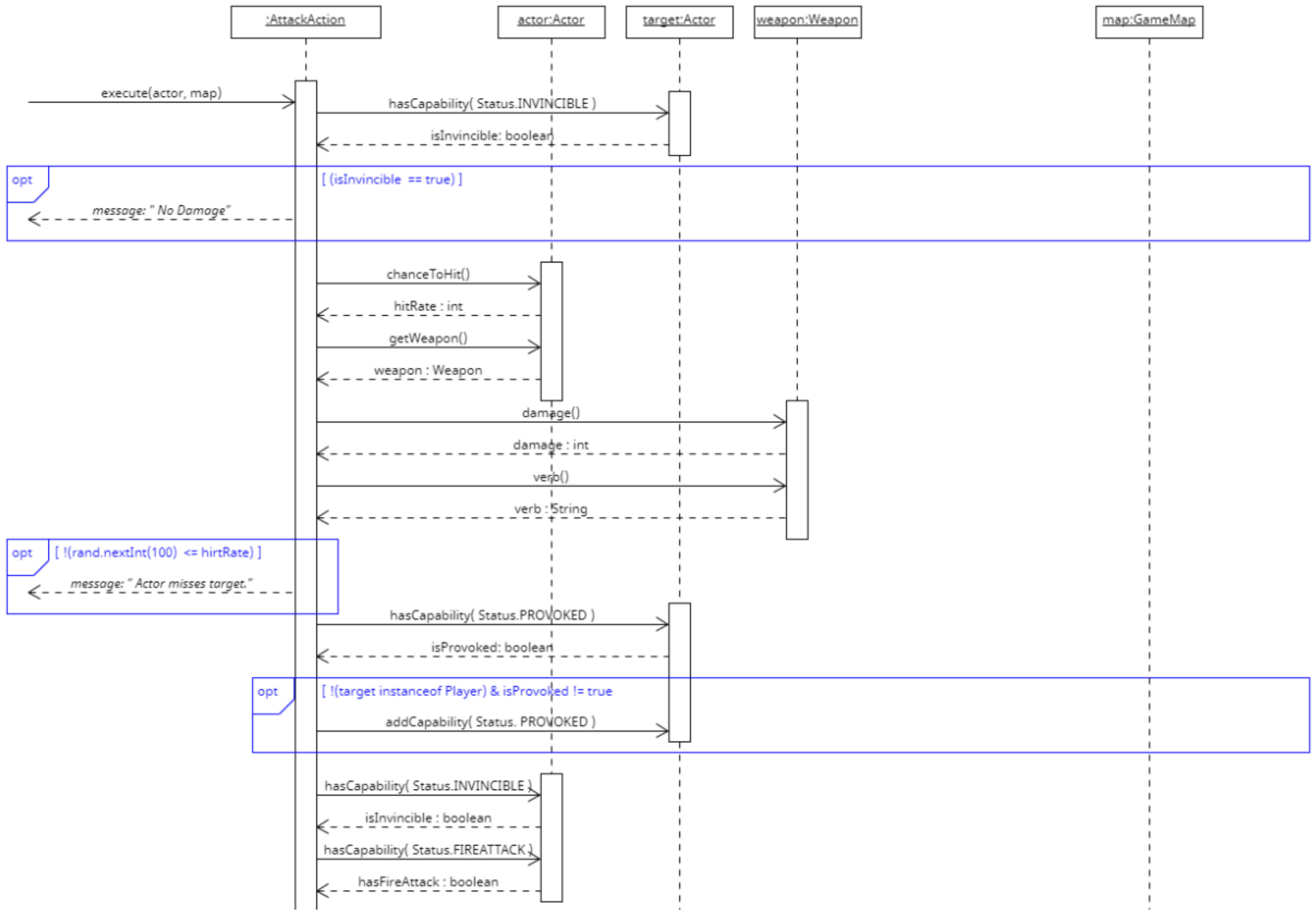


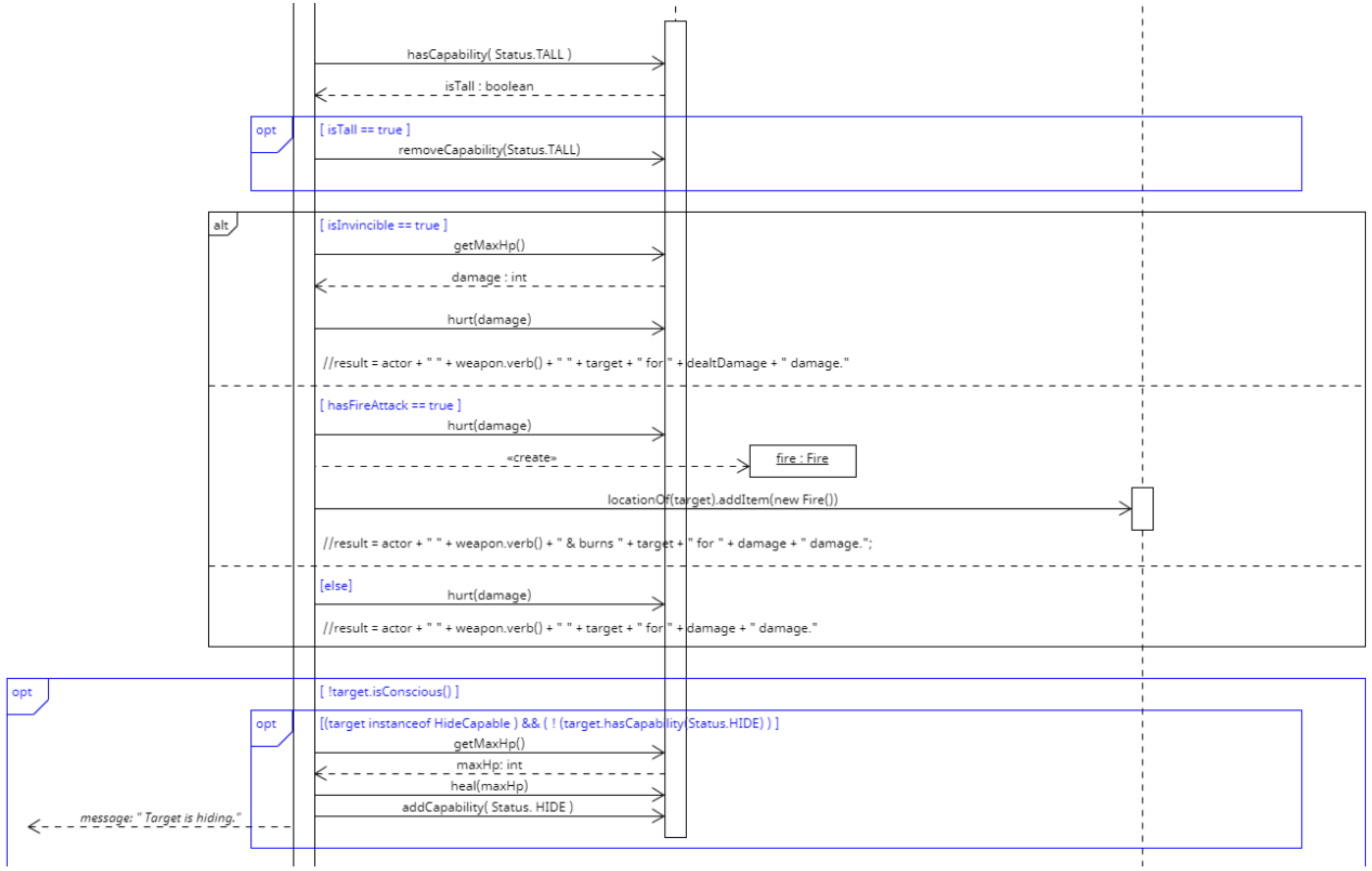
- There will be a bottle class that extends Item and a drinkWater action to complete the item.
    - The new action specially for water is cause consuming water is a fundamentally different action compared to consuming other items.
    - The new bottle class is due to it being easier to add different types of behaviour to the bottle in the future.
- There will be a fountain base class and a two more classes healthFountain and PowerFountain that extends it.
    - The base class exists due to some behaviours being common across all the different types of fountains and having a base class and extending it, is in accordance with DRY as it reduces redundant code.
- An action for refill bottle is created.

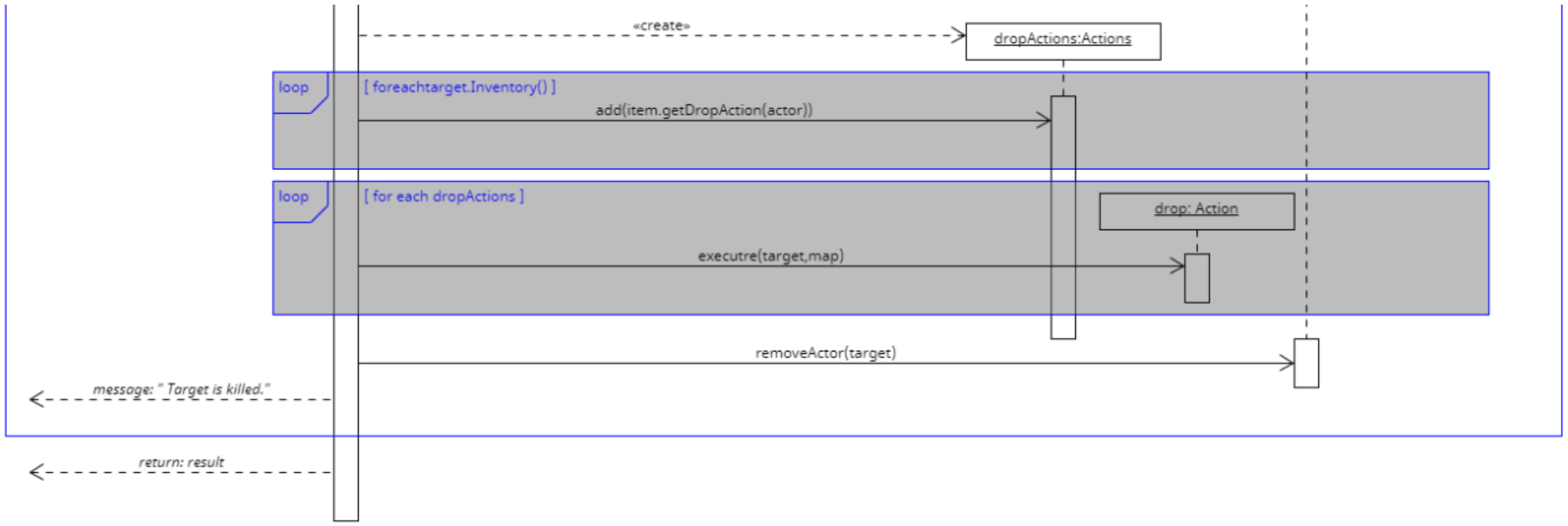
Assignment 3 - REQ 4: Flowers 🌻 (Structured Mode)



execute.AttackAction:

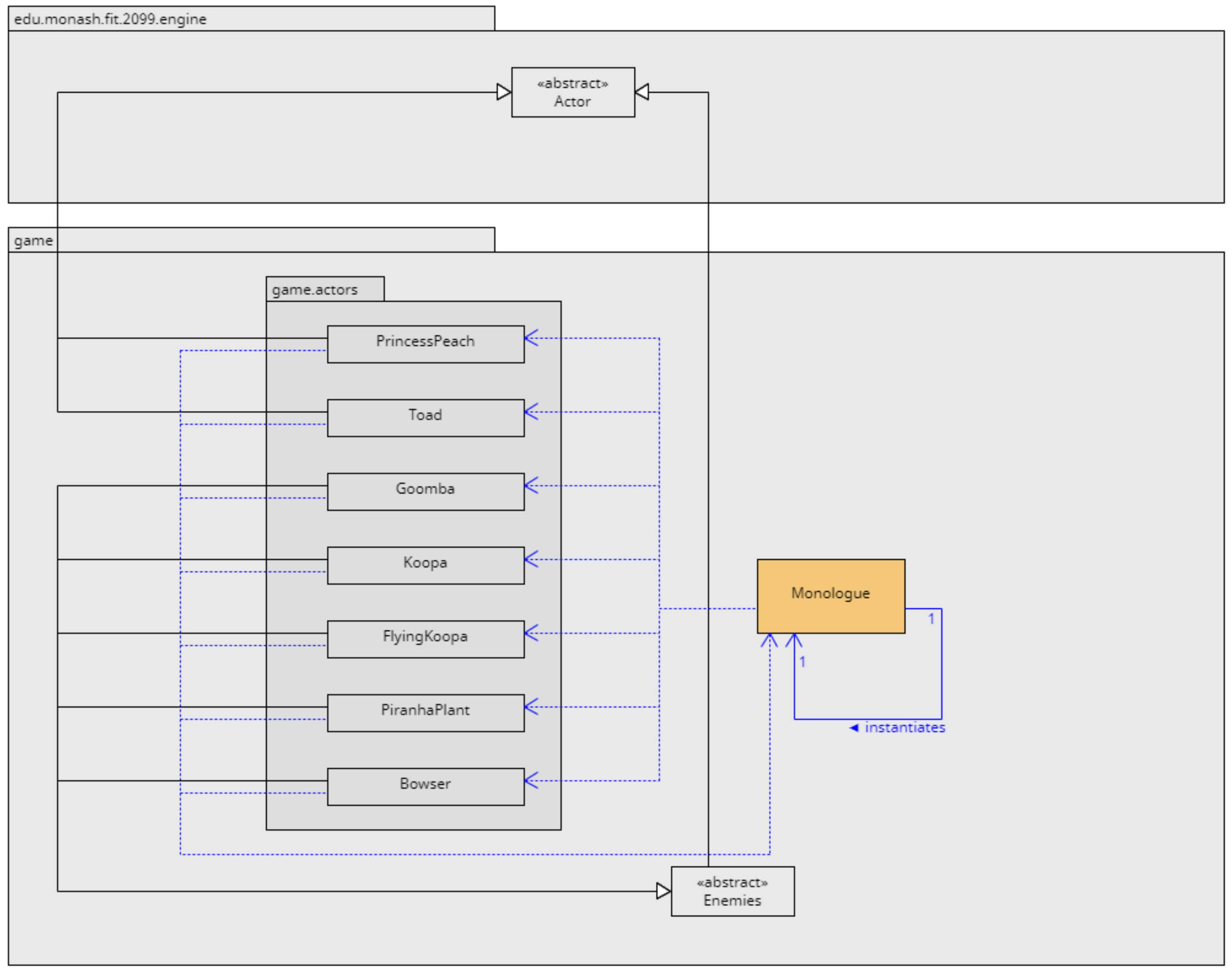

:AttackAction
actor:Actor
target:Actor
weapon:Weapon
map:GameMap
execute(actor, map)
hasCapability(Status.INVINCIBLE)
isInvincible: boolean
opt
[(isInvincible == true)]
message: " No Damage"
chanceToHit()
hitRate : int
getWeapon()
weapon : Weapon
damage()
damage : int
verb()
verb : String
opt
[!(rand.nextInt(100) <= hirtRate)]
message: " Actor misses target."
hasCapability(Status.PROVOKED)
isProvoked: boolean
opt
[!(target instanceof Player) & isProvoked != true
addCapability(Status. PROVOKED)
hasCapability(Status.INVINCIBLE)
isInvincible : boolean
hasCapability(Status.FIREATTACK)
hasFireAttack : boolean


hasCapability(Status.TALL)
isTall : boolean
opt
[isTall == true]
removeCapability(Status.TALL)
alt
[isInvincible == true]
getMaxHp()
damage : int
hurt(damage)
//result = actor + " " + weapon.verb() + " " + target + " for " + dealtDamage + " damage."
[hasFireAttack == true]
hurt(damage)
«create»
fire : Fire
locationOf(target).addItem(new Fire())
//result = actor + " " + weapon.verb() + " & burns " + target + " for " + damage + " damage.";
[else]
hurt(damage)
//result = actor + " " + weapon.verb() + " " + target + " for " + damage + " damage."
opt
[!target.isConscious()]
opt
[(target instanceof HideCapable) && (! (target.hasCapability(Status.HIDE))]
getMaxHp()
maxHp: int
heal(maxHp)
addCapability(Status. HIDE)
message: " Target is hiding."

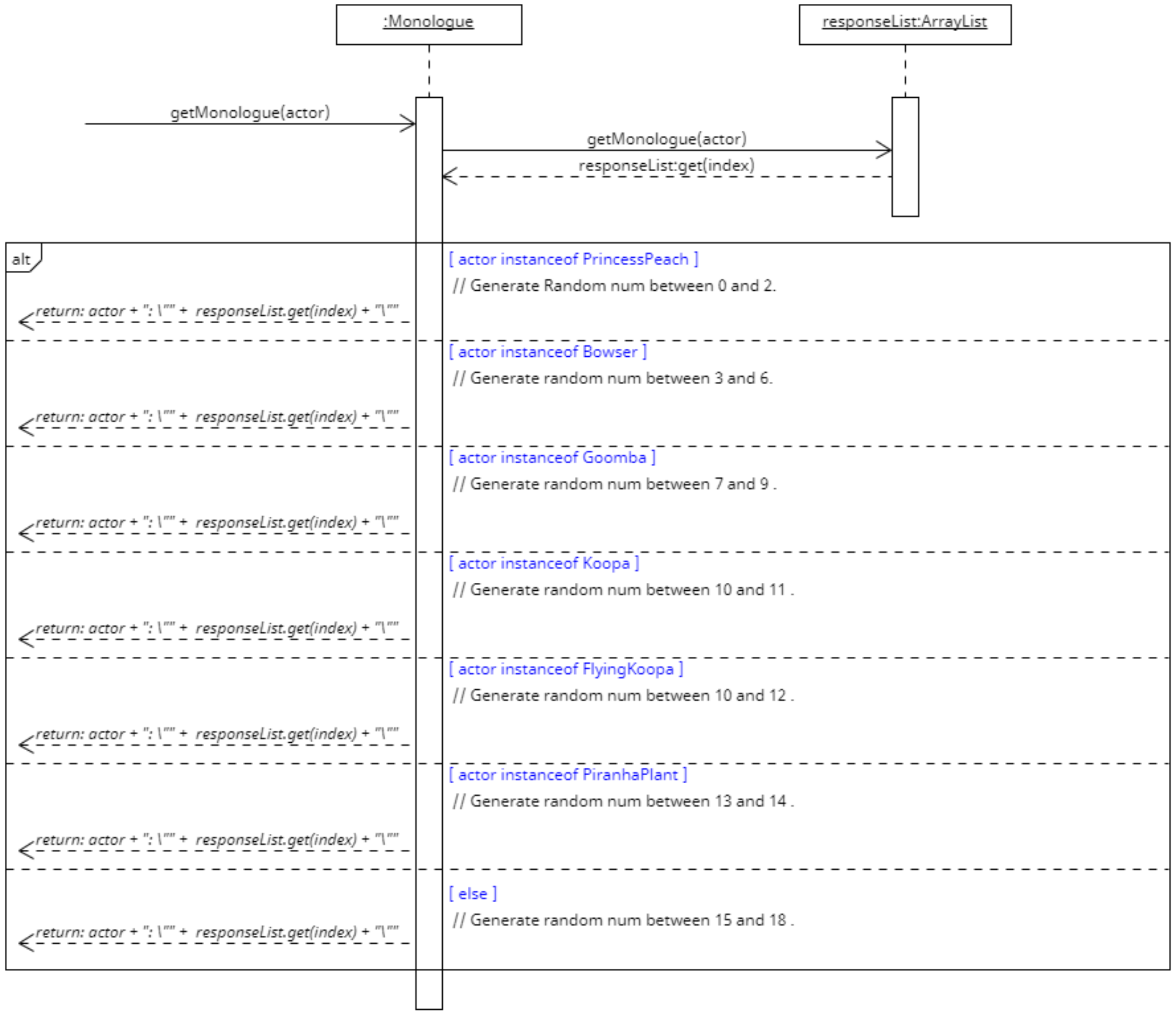


- New class FireFlower is created.
    - Created in sprout and sapling through the help of the random class in java.
    - Since a fire flower is technically a magic item and has similar properties/abilities to the other magical items(excluding water), we make use of the consume item action class to reduce code.
- Uses the Fire item that was created previously.
- Attacking with fire managed by attack action by checking Player’s status.
    - See if if has FIREATTACK
    - This is done as this capability is only temporary and thus an interface is unsuitable.

Assignment 3 - REQ 5: Speaking 🗣(Structured Mode)



Monologue.getMonologue:


:Monologue
responseList:ArrayList
getMonologue(actor)
getMonologue(actor)
responseList:get(index)
alt
[actor instanceof PrincessPeach]
// Generate Random num between 0 and 2.
return: actor + ": \"" + responseList.get(index) + "\""
[actor instanceof Bowser]
// Generate random num between 3 and 6.
return: actor + ": \"" + responseList.get(index) + "\""
[actor instanceof Goomba]
// Generate random num between 7 and 9 .
return: actor + ": \"" + responseList.get(index) + "\""
[actor instanceof Koopa]
// Generate random num between 10 and 11 .
return: actor + ": \"" + responseList.get(index) + "\""
[actor instanceof FlyingKoopa]
// Generate random num between 10 and 12 .
return: actor + ": \"" + responseList.get(index) + "\""
[actor instanceof PiranhaPlant]
// Generate random num between 13 and 14 .
return: actor + ": \"" + responseList.get(index) + "\""
[else]
// Generate random num between 15 and 18 .
return: actor + ": \"" + responseList.get(index) + "\""

- Create a new Monologue class.
- Declare monologue as a private static instance variable.
  → So that for each instance, they hold their own values
- Create a private static getInstance method.
  → To return the value stored in each instance
- Create a public String getMonologue method which takes in Actor as a parameter. Which:
  → Adds all 19 monologues into an ArrayList called monologue
  → Uses if, else if, else statements to see which actor is instantiated, if actor is instance of Actor, generate a random sentence that Actor would say. (example: checks which actor is instantiated, if actor instantiated is Toad, randomly chooses a sentence Toad would say.)
  → Returns the actor instantiated: "a random sentence the actor instantiated would say" using the formula actor + ": \"" + monologue.get(num) + "\""
  → An example output will be: Toad: "You might want a wrench to smash Koopa's hard shells."
- Under playTurn for each Actor that has monologues, add numTurns and check, where check = numTurns % 2, when check == 0, randomly print a monologue of the actor instantiated.
  → numTurns will increase by one every turn the actor is instantiated, and check is recalculated every turn
  → To ensure that actors only speak in alternate turns
  → A rough idea of it would be that:
    ★ For the first turn of the game no actor speaks
    ★ For the second turn, Bowser and Goomba(1) speaks
    ★ For the third turn, Bowser and Goomba(1) doesn't speak but goomba(2) which was spawned during the second turn speaks
    ★ For the fourth turn, Bowser and Goomba(1) speaks whereas Goomba(2) doesn't
- Displays the return value of getMonologue in the console menu depending on which actor was instantiated using the execute method.