FIT2099 Work Breakdown Agreement

Team Number	Lab 16 Team 2
Names	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934
Date	29/3/2022

Our team will break down our work by assigning tasks to each member. For each requirement, the assigned member will be responsible for creating the respective UML Class Diagram, UML interaction Diagram.

Task	Member	Date of completion
Requirement 1	Low Lup Hoong	5/4/2022
Requirement 2	Low Lup Hoong	6/4/2022
Requirement 3	Garret Yong Shern Min	8/4/2022
Requirement 4	Jastej Singh Gill	4/4/2022
Requirement 5	Jastej Singh Gill	5/4/2022
Requirement 6	Garret Yong Shern Min	8/4/2022
Requirement 7	Garret Yong Shern Min	8/4/2022
Reviewing	All members	10/4/2022

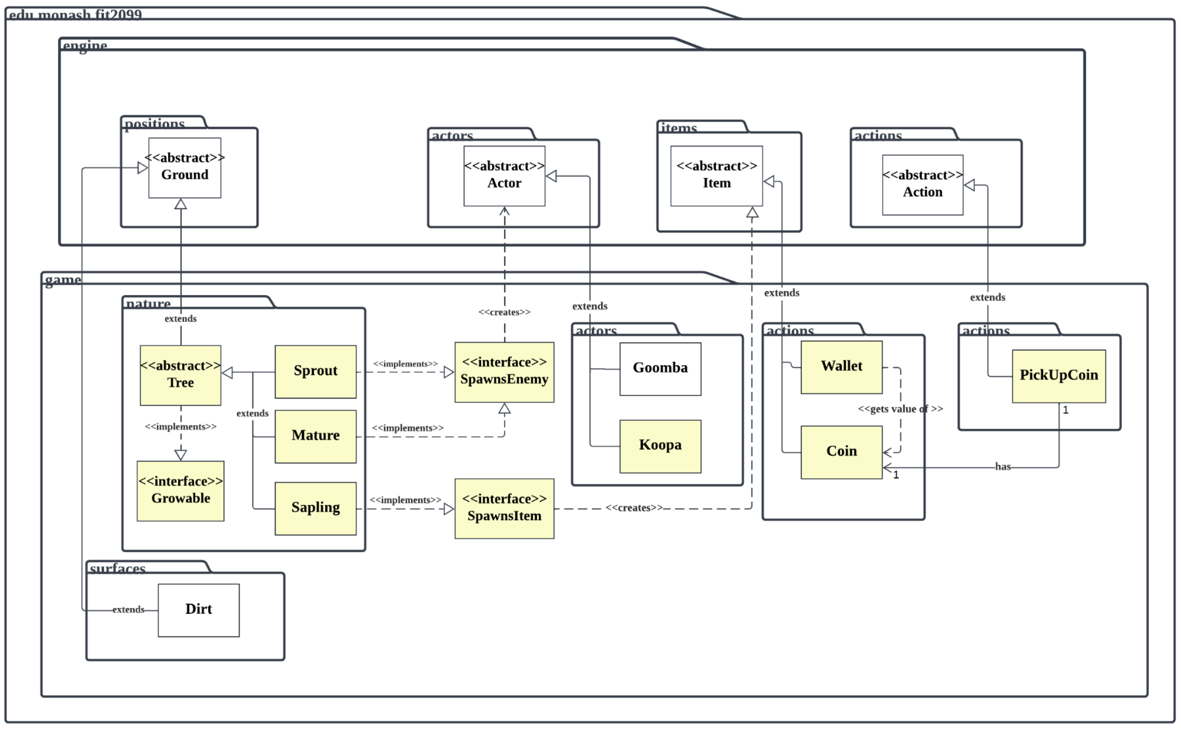
FIT 2099 Object-Oriented Design and Implementation

Title	FIT2099_S1_2022 Assignment 1 Design Documentation
Date	10/4/2022, Sunday
Team Number	Lab16Team2
Team	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934

This file contains the justification for each design choice made for each requirement

Requirement 1

UML Diagram



Referring to the UML Diagram above for Requirement 1, the yellow-shaded boxes are the new classes added in ‘game’.

Design Implementation

New additions and changes made to the existing system:

Interface Growable	Interface methods for tree stages growth with different realisation requirements - Sprout/Sapling/Mature would keep count of the ticks inside its tick(location) method that overrides its parents (Ground and Tree). - If count reached its limit, grow into a new stage location.setGround(newGroundInstance)
Interface SpawnsItem	Interface methods to spawn an item - implemented by entities to spawn an item on the same location of the entity implementing this interface - If a Ground were to spawn an item, the spawn methods would be inside Ground's tick() method, that accepts the location as parameter → location.addItem(itemToSpawn) - If an Actor were to spawn an item, the spawn methods will be inside playTurn(), that has Gamemap as an input parameter → gamemap.locationOf(this.actor).addItem(itemToSpawn)
Interface SpawnsEnemy	Interface methods to spawn an enemy: - implemented by entities that spawn an enemy - checks if conditions are met to spawn an enemy (conditions met if no actor is standing on the entity) - spawns an enemy - similar implementation as SpawnsItem

Abstract Class Tree	Inherits Ground class Implements Interface Growable Parent class of Sprout, Sapling, Mature Contains common attributes and methods of subclasses such as height, width.
Class Sprout	Inherits Tree abstract Class Implements Growable - grows into Sampling after 10 turns Implements SpawnsEnemy, - spawns Goomba with 10% chance
Class Sapling	Inherits Tree abstract Class Implements Growable - grows into Mature after 10 turns Implements SpawnsItem - spawns $20 Coin at the same location
Class Mature	Inherits engine's Tree abstract Class Implements Growable - 20% to wither and die. Will turn into Dirt Implements SpawnsEnemy, - spawns Koopa with 15% chance Has a method to create sprouts randomly in adjacent squares every 5 ticks

Class Coin	Inherits engine’s Item class Has value Provides PickUpCoin Action
Class Wallet	Inherit engine’s Item class A portable item carried by Player Stores total balance of coins - takes coin object as input - get coin value - add to wallet balance

PickUpCoin	Inherits engine’s Action Class This action is added to Coin object with addAction() to: - Allow player to interact with Coin when the player is standing on the same location - Picks up Coin object when an actor is on Coin object - Removes Coin object from its location - Add Coin value into player wallet

Design Rationale

Changes made and design Rationale:

1. **We made the provided Tree class into an abstract class**

Tree class was made abstract as this adheres to DRY, Do not Repeat Yourself principle. This avoids the repetition of code for tree-related attributes (eg height, width) and methods (similar reset settings for all trees).

Making Tree class abstract also achieves “Open to Extension, Closed to Modification”. This allows adding tree-related features for all tree types relatively easier.

2. **Tree abstract class implements interface Growable**, where all subclasses will provide concrete methods to implement different realisation of growth accordingly. In the future, the Tree abstract class can implement more interfaces, which can be passed down to its subclasses. This further reinforces achieving the Open Closed Principle.

3. We then **created 3 new classes that inherit Tree abstract class**: **Sprout, Sampling and Mature**. Creating a separate class for different stages of growth instead of storing the stages as an attribute in a single class achieves the Single Responsibility Principle. This removes using if statements to check the stages to perform different logic.

4. Sprout and Mature will implement to **newly created SpawnsEnemies interface**. Sapling will implement **the newly created SpawnsItem interface.**

   This adheres to the Interface segregation principle, by breaking it down into small interfaces to achieve a more robust interface implementation. If more subclasses of trees are created, they can implement many combinations of interfaces with each of its own unique realisations easily.

5. **PickUpCoin class** is created as a new class. This action will be dedicated to picking up coins and adding the value to the player’s wallet balance which achieves the Single Responsibility Principle.

This action will be added to the Coin class. When the player is in the same square as the coin, the coin will return a PickUpCoin action to the player. By doing this, we are making use of the provided engine code.

This is our most recent design with our current knowledge and will be further improved. Despite the stated strengths of this design, we have identified the trade-offs. It is observed that there is a multiple-level inheritance from abstract Ground class to abstract Tree class.

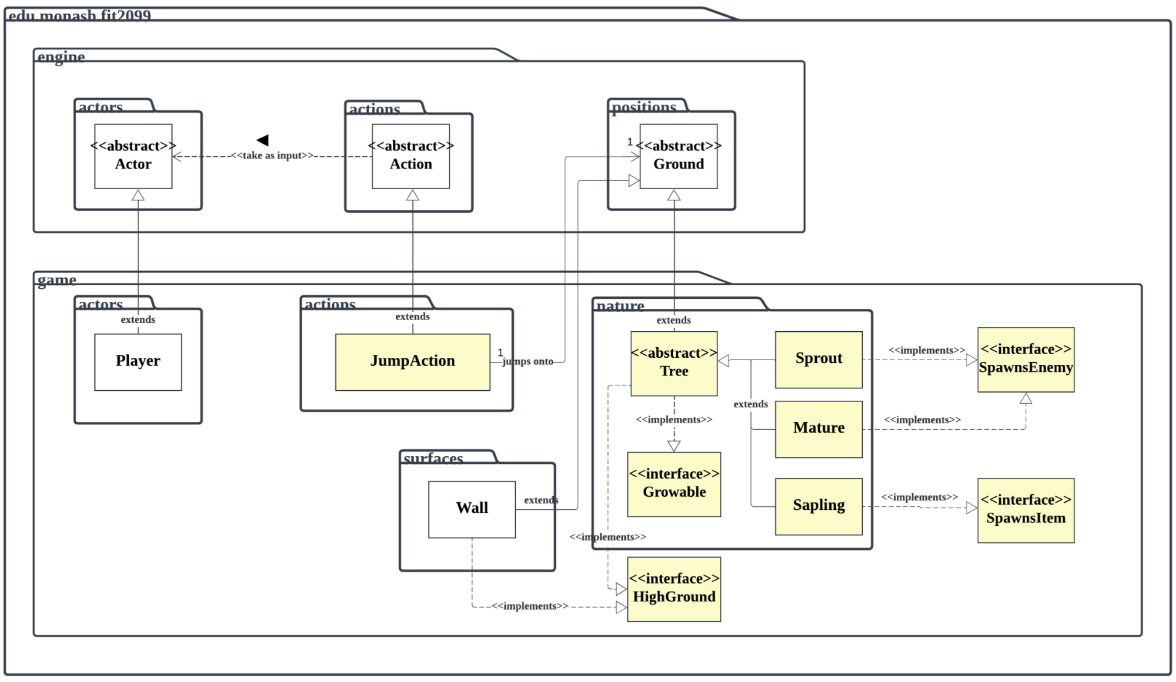
The potential risk is that a method in the Ground class might override methods in the Tree class, which might not be intended. This could lead to unexpected behaviours during gameplay. However, upon digging into the code, the Ground class does not override any methods except for getDisplayChar(). Since this is a getter and does not set or run other methods, the risk is low.

If we were to remove Tree as an abstract class to avoid multiple level inheritance, there would be a repetition of code if we want to add more tree-related features that apply to all Tree subclasses. This is a violation of the DRY principle.

It is evident that the pros outweigh the cons. Therefore, we have decided to go with our design.

Requirement 2

UML Diagram



The above UML Diagram illustrates how the jump action is added on top of requirement 1 from the previous section.

Design Implementation

Changes made to the existing system:

New Capability	CAN_JUMP capability is added to the Status enum class. This Capability is used to indicate that an actor can jump onto high grounds.

Player	The new capability CAN_JUMP is added to Player.
New interface HighGround	Include methods that all HighGround implements, This will be further elaborated in requirement 4.
Wall, Sprout, Sapling, Mature (high grounds)	Implements interface HighGround Two attribute jumpSuccessRate, and damagePoints are added to the four high ground classes. jumpSucessRate indicates the high grounds' jump success rates. damagePoints indicate the damage done to the player upon a failed jump. Ground original canActorEnter method is overridden to return false. This is necessary to ensure that the player cannot move, but must jump. JumpAction is added into all high grounds' ActionList in allowableActions method. This ActionList is returned to the player inside processActorAction, allowing the player to jump onto these high grounds.

New Class JumpAction	Inherits engine’s abstract Action class Has direction as String attribute Has random generator to implement success rates Execution logic: - If the player has SUPER status (from SuperMushroom), then the player will jump successfully. - If the success rate is greater than the high ground’s jump success rate, the player will jump successfully. - Else: player.hurt(damagePoints) If the player jumps successfully, the player will move onto high ground and have no damage incurred. A message is printed onto the console indicating a successful jump and the coordinates of the current high ground. If a player fails to jump, the player stays at the old position as before with damage incurred. A message is printed to indicate a failed jump and will damage their health points. The player could jump (move onto high ground) using Gamemap’s methods. Console messages are printed using Action’s menuDescription method. This uses the existing codes in the engine.

Design Rationale

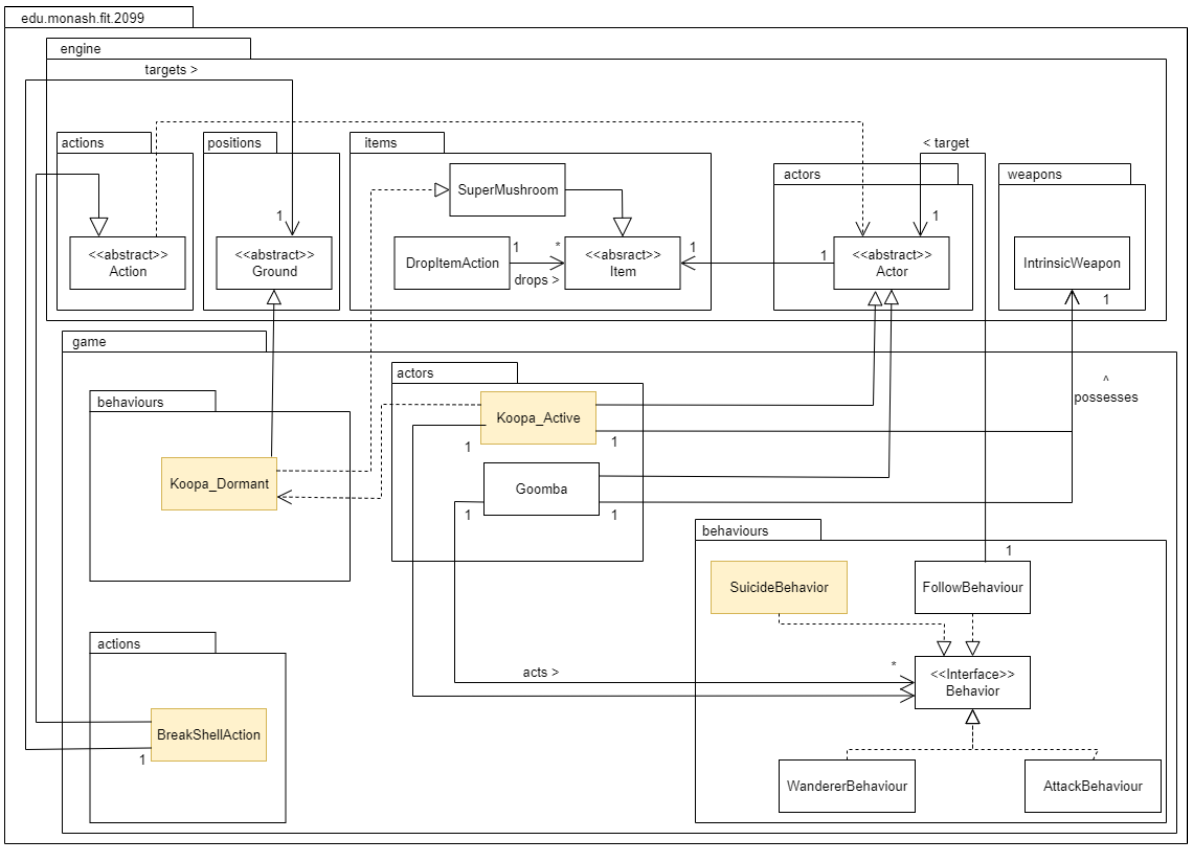
It can be seen that we have arranged our classes within “game” as illustrated in this UML diagram, where all tree-related classes are within a package called “nature”. Other ground classes are inside another package called “surfaces”. The design rationale behind this is to reduce encapsulation boundary crossings. By putting related classes closer to each other, we adhere to the Reduce Dependency, ReD principle.

JumpAction class is created. This class extends the engine’s abstract Action class. In our design, only one jump action is created at the moment. An alternative design would be to create two jump actions: JumpOnTree and JumpOnSurface. This seems to be supporting the Single Responsibility principle, where each class is responsible for one single responsibility. However, this would cause the classes to be too small. This overcomplicates the design as extending the JumpAction generally would require repetition on codes in both classes, violating the Do not Repeat Yourself principle.

Furthermore, the classes within both packages “surfaces” and “nature” are all subclasses of Ground. Therefore, we have decided to have only one JumpAction class for now.

Requirement 3

UML Diagram



Design Implementation

Class Koopa_Active	This class represents a generic Koopa enemy that is currently active This class inherits the Actor class This class will adopt the following behaviour(s): - FollowBehavior (allows Actor to follow Player) - WandererBehaviour (allows Actor to wander the map) - AttackBehaviour (allow Actor to attack)

	This class also has an intrinsic weapon of a punch that deals 30 damage with a 50% hit rate. One method has been modified for this class: - isConsicious() method is modified so that a new Koopa_Dormant replaces the tile that the Koopa_Active is defeated on
Class Koopa_Dormant	This class represents a generic Koopa enemy that is currently dormant This class inherits the Ground class This class does not adopt any behaviours The only allowable action that is performed on the Koopa_Dormant is the BreakShellAction (see next row) When the BreakShellAction is performed on the Koopa_Dormant, the Koopa_Dormant is removed from the map and a SuperMushroom is added to its location (this utilises the tick function of the ground class to attain the location of the Koopa_Dormant to create and add a new instance of SuperMushroom at that location) When returning the allowable action, if the otherActor (Player) is not wielding a wrench, the BreakShellAction is not made available in the menu.
Class BreakShellAction	This class represents the action of breaking the shell of a fallen Koopa This class inherits the ActionClass This class will have several methods - execute() will remove the Koopa_Dormant and replace it with an instance of Dirt. It will also create and drop a new instance of SuperMushroom on that location - menuDescription() functions just as it would with attack action, describing the available action

Class SuicideBehaviour	This class represents the suicidal tendency behaviour This class inherits the behaviour interface This class will repeatedly roll for a 1 in 10 chance to determine whether the actor will kill itself. (This class is meant to be assigned to the Goomba)

Design Rationale

One option would be to modify the child class (Koopa) to possess 2 sets of hp (one inherited from the actor class and this would serve as the dormant state HP while the 2nd set which represents the active state would need to be a new attribute exclusive to this class).

When the Koopa is attacked, while the active state's HP is more than 0, the Koopa's active state HP will be the one decreasing until it drops to 0. This is where the display character will be altered to a D, all behaviour will be cleared from that instance of Koopa, the status of the Koopa will change from HOSTILE_TO_ENEMY to a new status called DORMANT which only allows the user to interact with the dormant Koopa (not attack it).

This is a bad idea as it violates Liskov's Substitution principle as we can no longer substitute the Koopa and actor classes as the Koopa class now has much stricter constraints and this cannot be substituted for each other.

On the flip side, a better option would be to create two classes of Koopa, Koopa_Active and Koopa_Dormant. Koopa_Active inherits from the Actor class so that it is able to take in behaviours and actions to allow for movement and combat. On the other hand, Koopa_Dormant inherits the ground class as it does not require a hitpoint calculator and is supposed to be only interacted with (should not take in any behaviour).

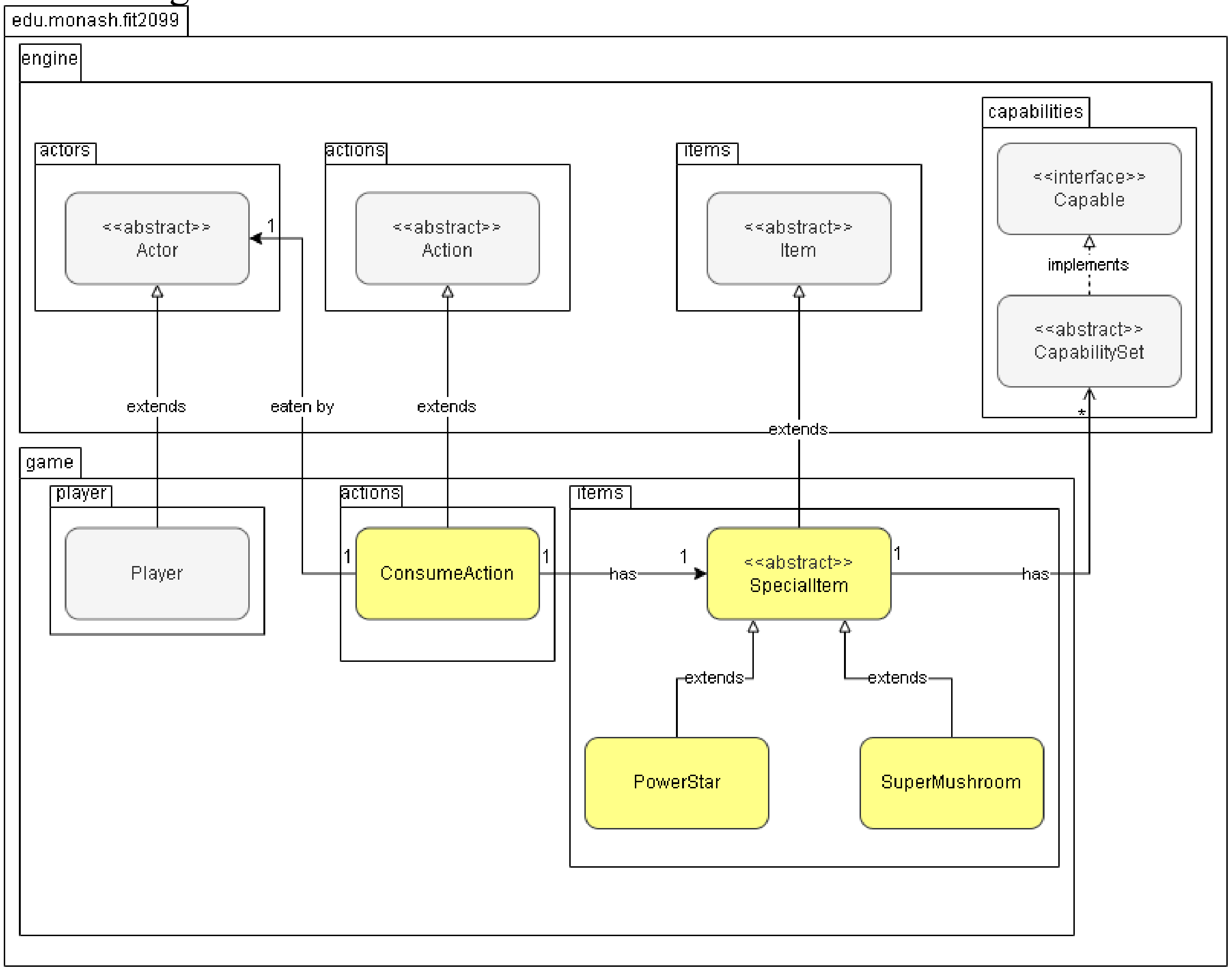
In battle, the Koopa will follow and attack the Player but once it is defeated, Koopa_Active will be removed from the application but a Koopa_Dormant will be spawned in its place. When the player comes into proximity to the Koopa_Dormant. The only allowable action will be to break its shell (BreakShellAction) and that option will display if and only if the player has a wrench in their inventory.

This is a better option than the first-mentioned method as a dormant Koopa has no need for most of the attributes of the Actor class. So by substituting the Actor class with the Ground class, the attributes and patterns of the ground class would complement the dormant Koopa.

In addition to that, SuicideBehaviour is a new class that inherits the Behaviour class. This class is created so that if there are other classes that are meant to commit suicide, those classes can add this class to their list of behaviours instead of rewriting the same code in its own class (Do Not Repeat Yourself principle).

Requirement 4

UML Diagram



Design Implementation

Abstract Class SpecialItem	Inherits Item class - Contains common methods and attributes for SuperMushroom and PowerStar classes - All items are portable and undroppable - Have special capabilities that can be given to the player
Class SuperMushroom	Inherits SpecialItem class - Has capabilities that will make the player: 1. Tall (TALL status) 2. + 50hp (SUPER status) 3. Jump with 100% success rate and no fall damage (SUPER status) - Capabilities are assigned to the item by status using Status enum
Class PowerStar	Inherits SpecialItem class - Has capabilities that will make the player: 1. Invincible (INVINCIBLE status) 2. Able to walk onto higher grounds (INVINCIBLE status) 3. No longer experience fall damage (INVINCIBLE status) - Capabilities are assigned to the item by status using Status enum
Class ConsumeAction	Inherits Action class Has execute(actor, map) method to transfer the capabilities of special items to the player when the player consumes a special item from their inventory

interface HighGround	As mentioned from requirement 2, this interface class is implemented by high grounds (Wall, Sprout, Sapling, Mature). This interface has methods to destroy high grounds. It also includes methods to spawn an item at the high ground's location. In assignment 1, all high grounds will drop a Coin item of value of $5.
Extension to class JumpAction	On top of the specified logic in requirement 2, the following logic is added to its class: If the player has INVINCIBLE status, - Destroy high ground by setting it to Dirt - Spawn an item (Coin) on its location - Player always jumps successfully but returns console message "move" instead of "jump" onto console

Design Rationale

SpecialItem abstract class

A new abstract class SpecialItem is created that extends SpecialItem. This is due to the special items SuperMushroom and PowerStar having similarities in their properties such as both items being undroppable by the player and both of them being able to give the player capabilities in the form of a Status. Therefore, to reduce the amount of repeated code in the SuperMushroom and PowerStar classes, a SpecialItem abstract class is created, adhering to the Do Not Repeat Yourself principle. This approach will also make sense for the ConsumeAction class which will have an association with the SpecialItem class as it will confine the Player to only be able to consume special items and not other items like a wrench. The SuperMushroom and PowerStar classes will then extend the SpecialItem class.

Another alternative would've been to have SuperMushroom and PowerStar inherit directly from the Item class. However, with this approach, there would be much more repeated code compared to the previous approach making this approach not only inefficient as multiple methods will have to be overridden similarly in both classes which violates the Do not Repeat Yourself Principle. Besides that, the ConsumeAction class will be able to have an instance of any item, not just special items and this can be problematic as not all items in the game are meant to be consumed.

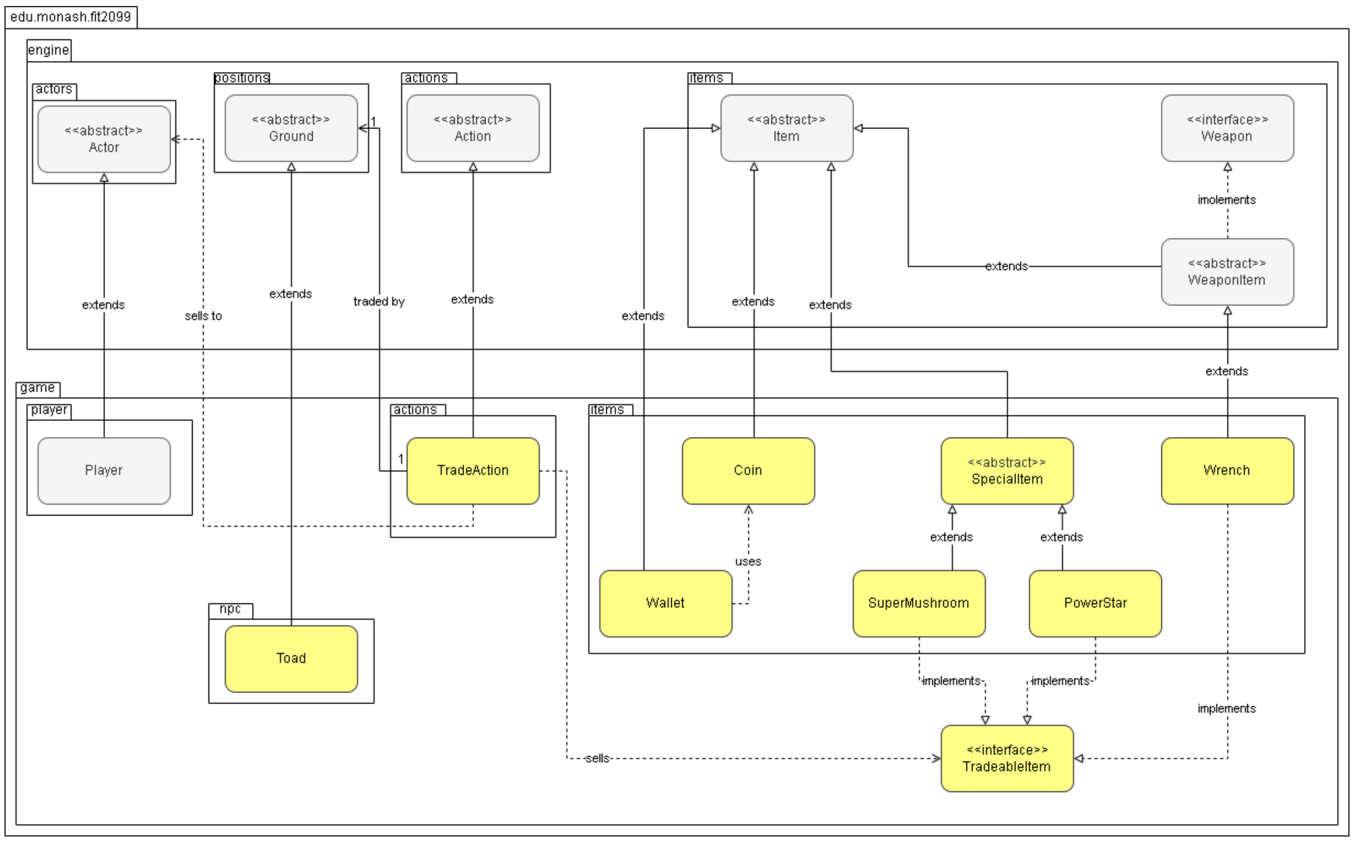
ConsumeAction class

A ConsumeAction class is also added that extends the Action abstract class. The ConsumeAction class is used when the player chooses to consume a SpecialItem in their inventory. Then the ConsumeAction class will be called which transfers all capabilities from the SpecialItem consumed and adds them to the Player's capabilities. This approach adheres to the Single Responsibility Principle. This is because the functionality of consuming an item is confined to the ConsumeAction class and not shared between several classes.

Another alternative was to not implement a ConsumeAction class and instead implement a method for the Player to consume an item in their inventory. This will violate the Single Responsibility Principle and the Dependency Inversion Principle. The Player class will now have more responsibilities such as the responsibility to consume an item along with having multiple associations and dependencies with concrete classes such as the SuperMushroom and PowerStar classes.

Requirement 5

UML Diagram



Design Implementation

Class Toad	Inherits Ground class - Will be set as impassable terrain - Allows Player to perform TradeAction when nearby
Class TradeAction	Inherits Action class - Gets an instance of an actor and checks their wallet - If their wallet has sufficient credits to buy an item, the item will be added to the actor's inventory and the credit will be deducted from total in Wallet - If credits are insufficient, a message will be printed to the console saying "You don't have enough coins!"
Interface TradeableItem	Interface implemented by all items sold by Toad - Gives the item a value as an integer - Allows an item to be traded by Toad - TradeAction can instantiate items that implement TradeableItem - has methods such as getValue() and setValue() so we can access and mutate the value of an item

Design Rationale

Toad class

The Toad inherits the Ground class as the functionality of the Toad is more similar to the Ground class than an Actor class. This is because the Toad does not need to have hit points or an inventory. The toad is also always conscious. Therefore, it makes more sense to let the Toad class inherit from Ground.

If we were to let the Toad inherit an Actor, we would need to change a lot of the inherited methods that may cause the Toad class to behave completely different from the Actor class it is inherited from. This would violate the Liskov Substitution principle. Methods such as isConcious() must be always set to true, inventory related methods will have do-nothing implementations and the hitpoints must be set to an arbitrarily large number. The Toad class behaves nothing like the Actor class it inherits from. Therefore making the Toad class inherit from Ground is much more logical.

TradeAction class

Inherits the Action abstract class. This class is used for all trade actions performed by the player.
Creating this class will allow the implementation of this requirement to adhere to the Single Responsibility Principle. Without this class, the responsibility of trading will be performed by the Player and Toad classes which would violate this principle as the two classes will have to share multiple responsibilities and cause direct relationships in between concrete classes which must be avoided as it will violate the Dependency Inversion principle. Therefore, the TradeAction class is introduced to prevent all these issues and handle trade actions performed by the player.

TradeableItem Interface

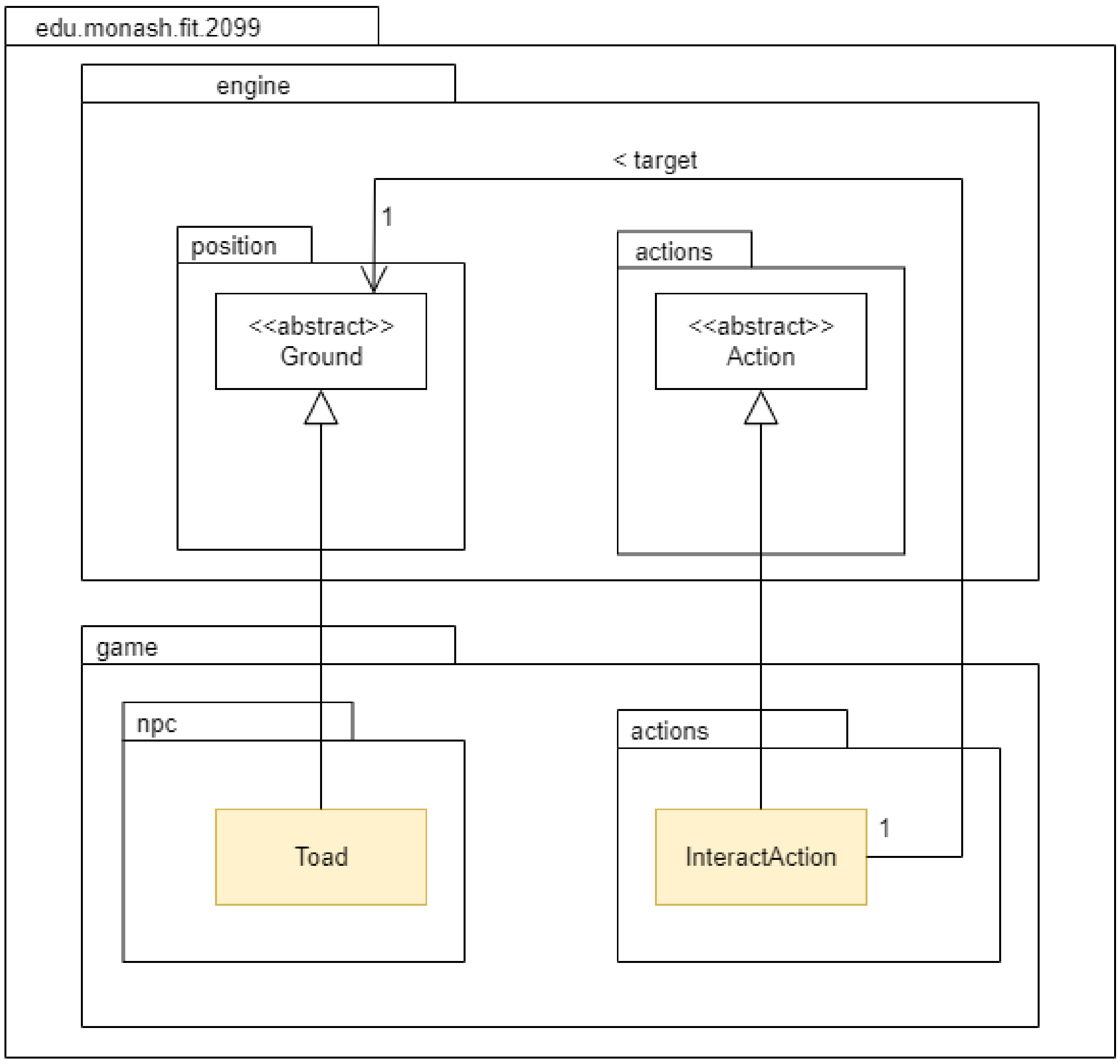
This interface is implemented by all items that are traded by Toad. This interface will ensure all items traded by Toad have the necessary methods and attributes such as the value of an item.

Alternatively, we could have done the implementation without a TradeableItem interface which would make implementation simpler but we could sell items that do not have the necessary attributes and methods. For example, we could sell a coin as the items do not need to have implemented a specific interface. If the coin does not have the necessary attributes and methods required to trade it, we could cause an Exception error.

Therefore, it is better to have the tradeable items implement the TradeableItem interface to ensure that all items sold by Toad are actually tradeable. This also adheres to the dependency inversion principle where instead of the TradeAction class being dependent on the multiple classes of items to be sold, the TradeAction class only has a dependency on the interface TradeableItem.

Requirement 6

UML Diagram



Design Implementation

Class Toad	A class that represents a regular NPC that provides dialogue and vends items to players Inherits the ground class There are some important methods declared in this class: - checkWrench() takes in an actor's inventory as a parameter and returns a boolean value. True if there is no wrench item in the actor's inventory

	- checkSuperStar() takes in an actor's statuses as a parameter and returns a boolean value. True if there is no INVINCIBLE
Class InteractAction	This class represents the action of interacting (speaking) with another NPC This class inherits the Action class This class will have several methods - execute() will take an actor as a parameter whereby that actor's inventory will be scanned for specific item(s). Besides that, the actor's status is also checked for a specific status. This checking is done in order to determine what message an NPC can output - menuDescription() functions just as it would with AttackAction, describing the available action

Design Rationale

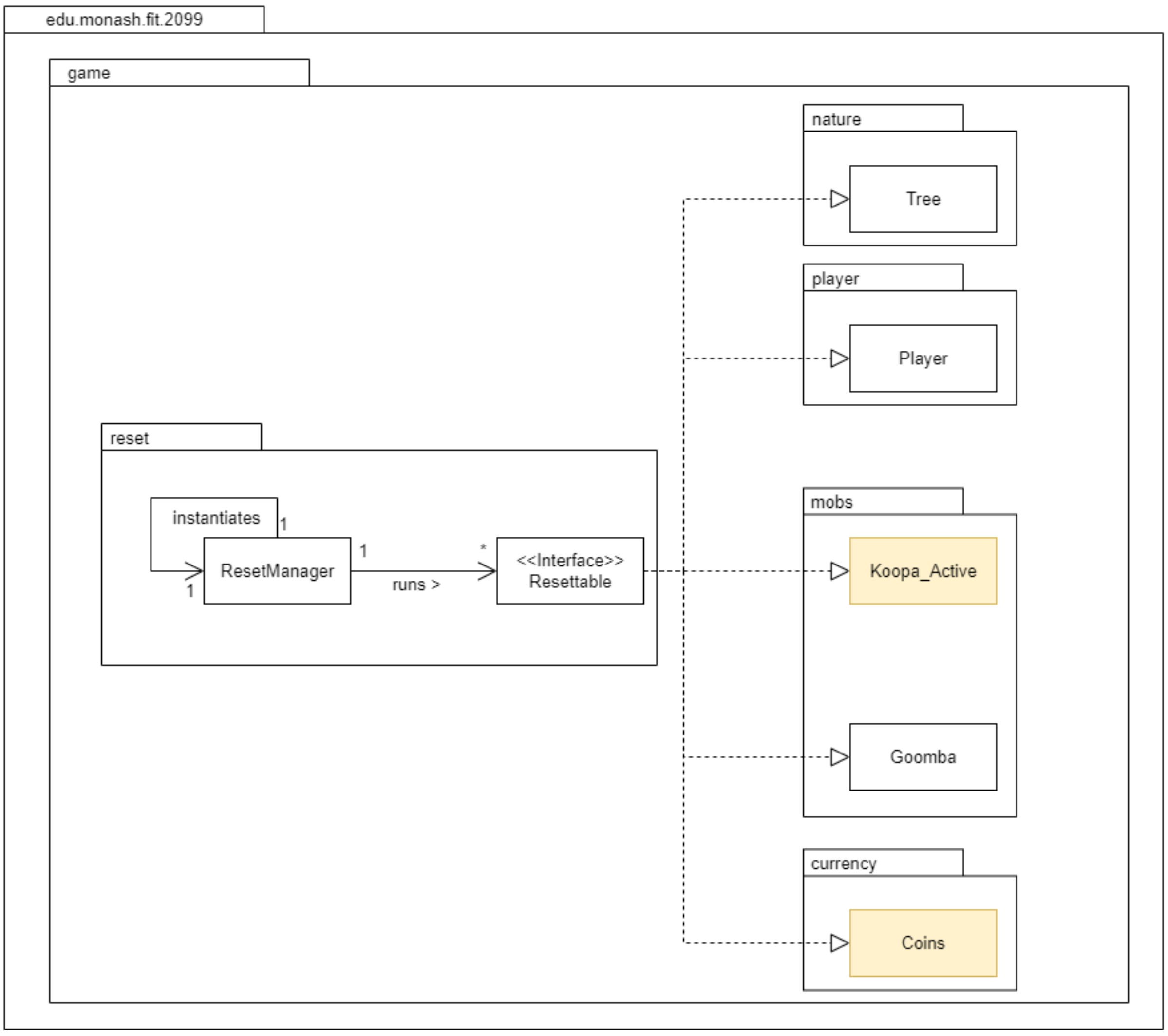
For this requirement, a valid option would be to modify the Player class to include a method to allow the player to interact with objects and sprites in the vicinity. However, this would mean that the player class would be responsible for an extra class. In addition to that, the shopkeeper (Toad he/she/itself) would store the prices for each object but that would make it hard for the prices to be changed.

Hence, to overcome this, a new class implementing the Action abstract class would be created similar in design to the AttackAction and this class would be called the InteractAction class. Much like the AttackAction, it will have a method that is able to select a target however, instead of taking another Actor as a parameter, it would take an object which has inherited the ground class as well. This would then allow for a Toad class to be created which inherits the Ground class to be interacted with. This design takes a page from the O in the SOLID principles whereby the addition of this new class. In future if newer functions were to be added to Toad, it would be easy to extend as this class avoids modifying Toad class directly.

To further elaborate on the Toad class, two methods would be declared the first method is the monologue method which takes in an arraylist of items (player inventory) and status. This method will exclude certain voice lines based on the requirements and then would randomly pick one of the remaining voice lines to be printed. The second method is the vendor method which takes in a Wallet as a parameter, when this method is called upon, the monologue method will be called upon to speak to the player then the shop would open up to sell the items.

Requirement 7

UML Diagram



Design Implementation

Class Tree	An abstract class that is a base class that represents a generic tree Inherits all methods and constructors from the Ground class Possesses only one concrete method: - reset() rolls a 50/50 to determine if an instance of Tree child class will be converted to dirt (this is achieved by the tick method from the ground class which takes in location as a parameter and this will allow us to access that location and set the ground as a new instance of the Dirt Class.
Class Player	Class that represents the playable character (Mario) that the user will control A new method will be included in this class - reset() will use the heal() method to heal the player to full hitpoints
Class Koopa_Active	This class represents the Koopa enemy that is hostile to the Player A new method will be included in this class - reset() will deal maximum damage to the Koopa_Active
Class Goomba	This class represents the Goomba enemy that is hostile to the Player A new method will be included in this class - reset() will deal maximum damage to the Goomba

Class Coins	This class represents a container for the currency of the game A new method will be included in this class - reset() will remove all coins in the map

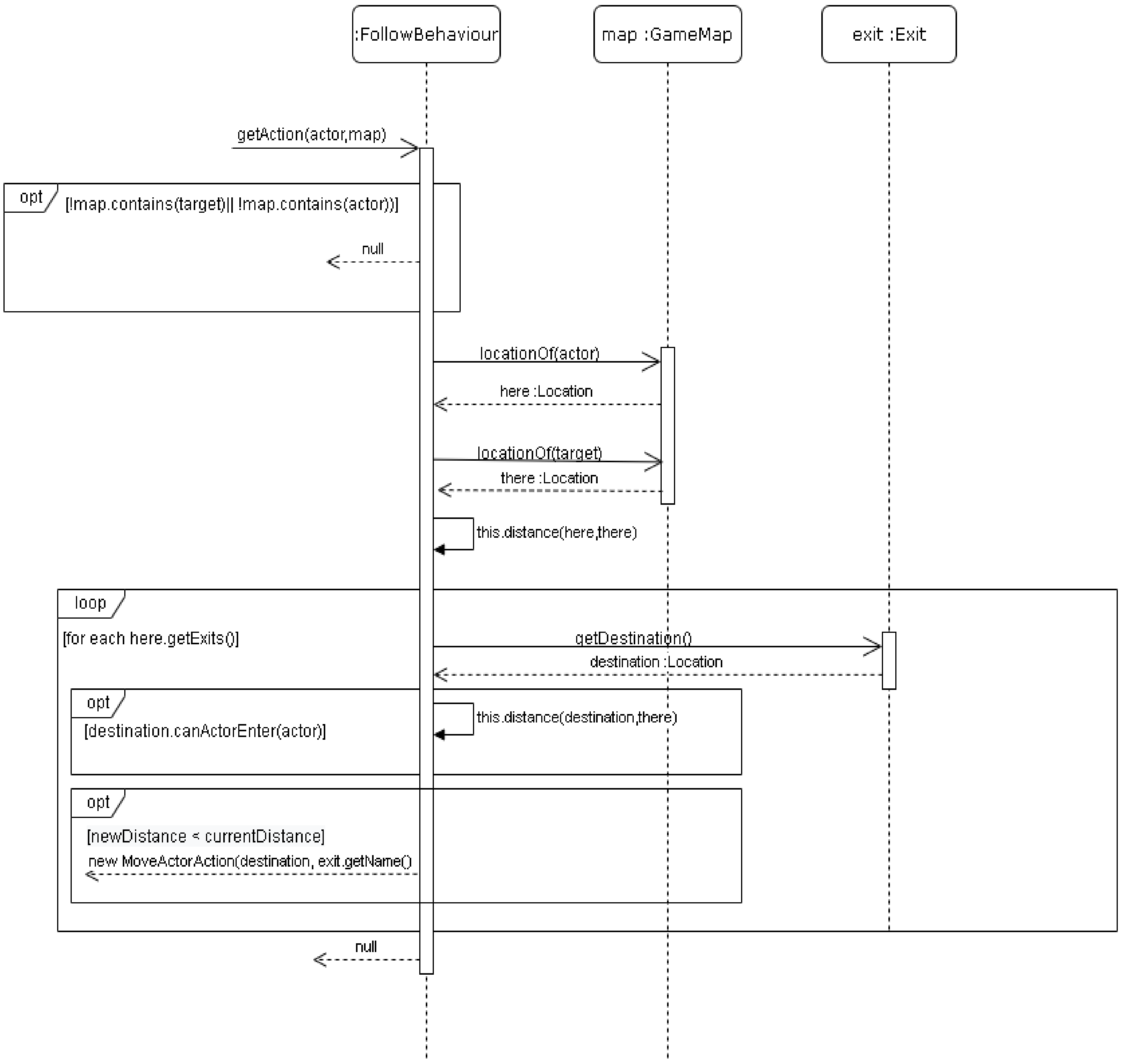
Design Rationale

One approach to achieve this was to use ResetManager to get all the instances of classes that implemented the Resettable interface and manually reset every instance based on their respective classes.

However, to further improve this design, we adapted S of the SOLID principle to further enhance our design. This was achieved by creating a reset method to each class that implements the Resettable interface. This would mean that the ResetManager would only be concerned with obtaining an instance of a Resettable object while each class that implements the interface will be responsible for its own patterns of being reset.

Interaction Diagram

FollowBehaviour.getAction(actor,map)


:FollowBehaviour
map :GameMap
exit :Exit
getAction(actor,map)
opt
[!map.contains(target)|| !map.contains(actor))]
null
locationOf(actor)
here :Location
locationOf(target)
there :Location
this.distance(here,there)
loop
[for each here.getExits()]
getDestination()
destination :Location
opt
[destination.canActorEnter(actor)]
this.distance(destination,there)
opt
[newDistance < currentDistance]
new MoveActorAction(destination, exit.getName()
null