FIT2099: Object-Oriented Design

Assignment 1: Design

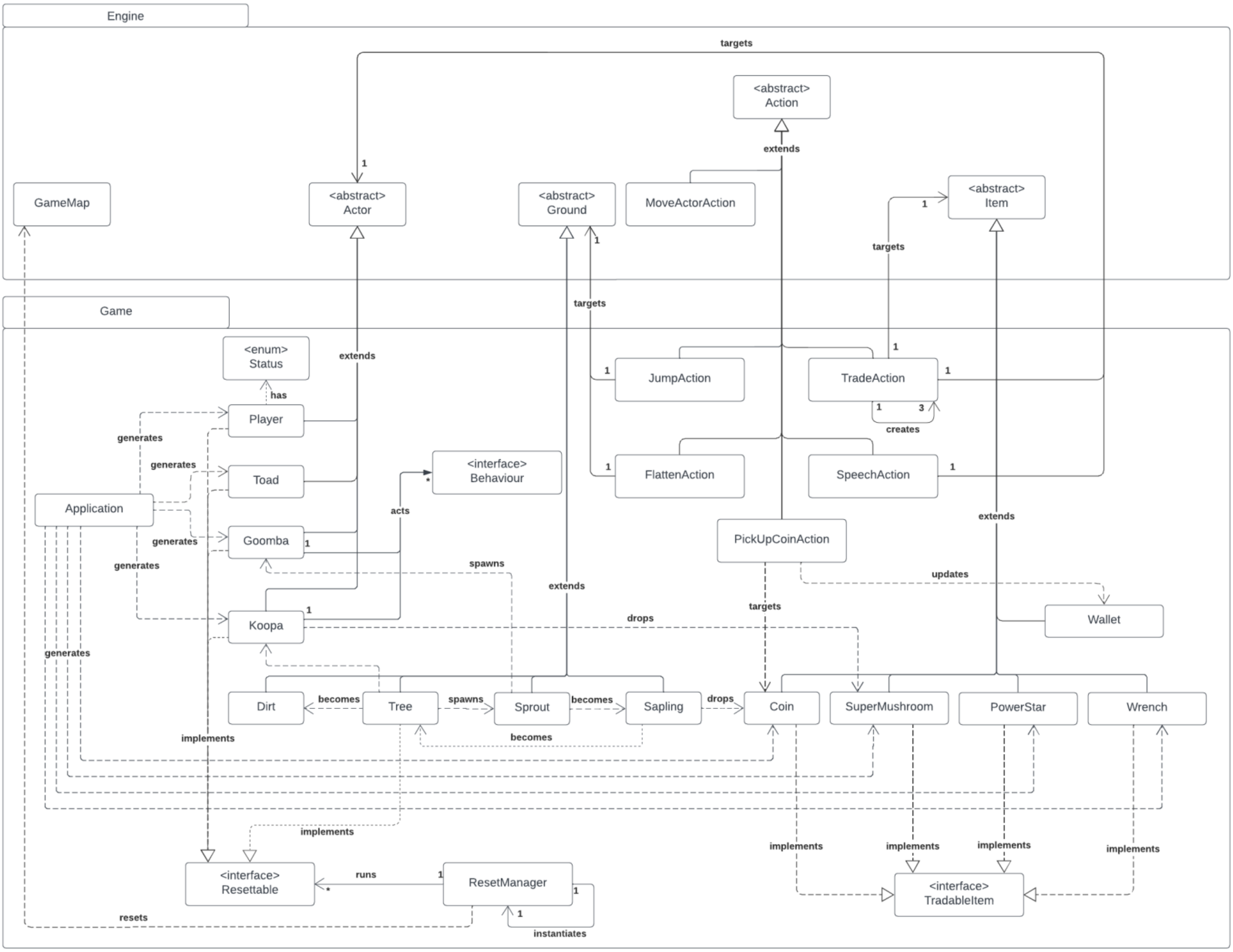
Due Date: 11 April 2022

Lucus Choy

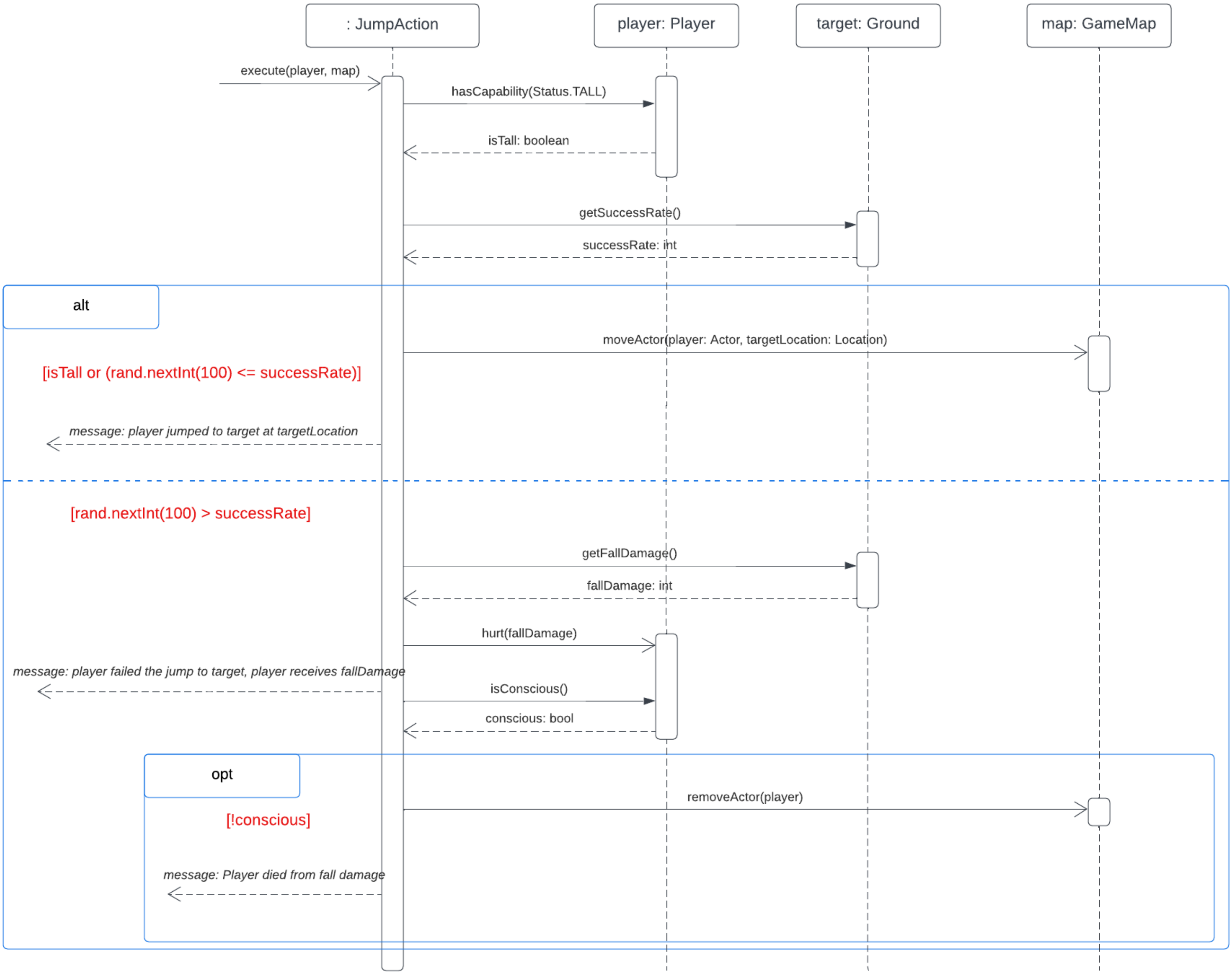
Connor McCloud-Gibson

Shang-Fu Tsou

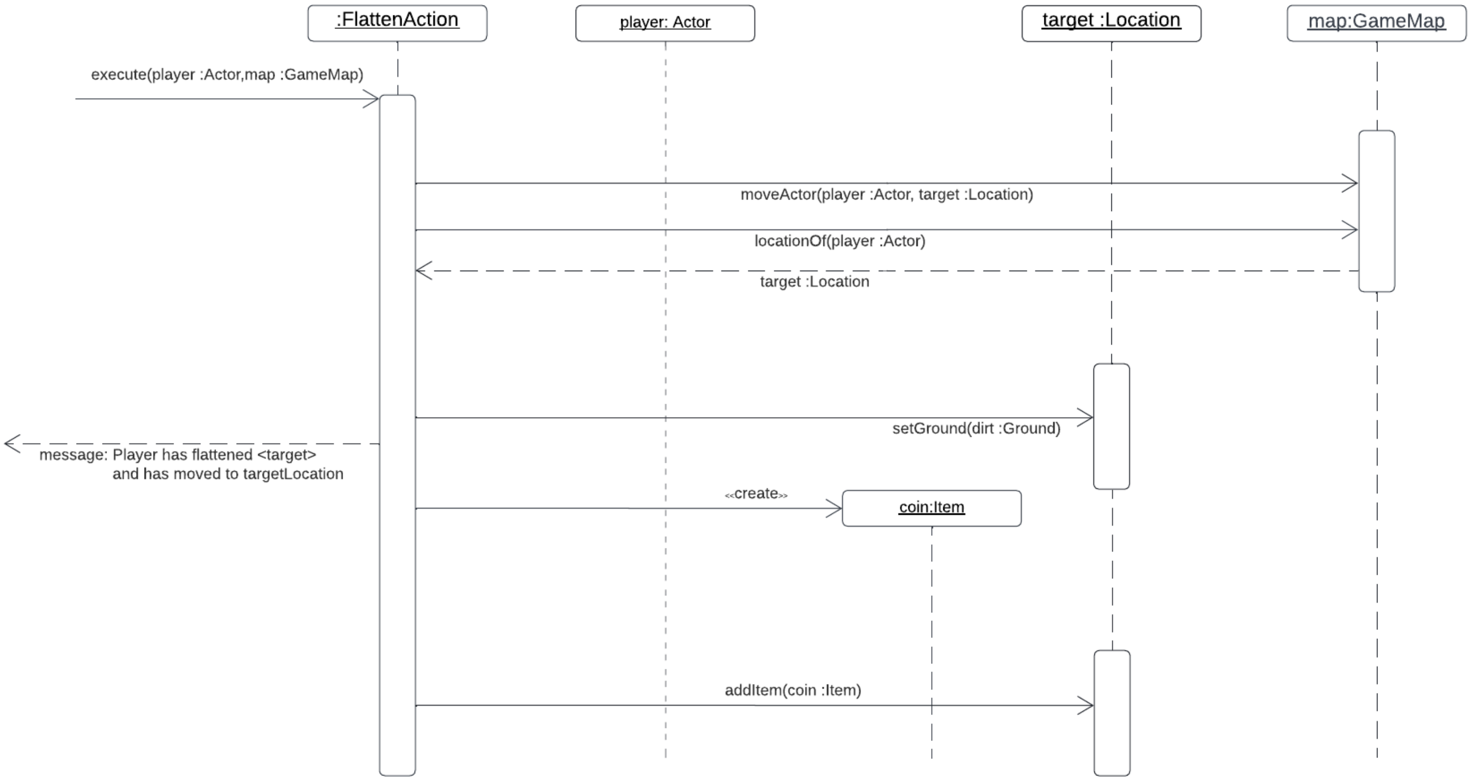
Overall UML Diagram For Added Functionalities



Requirement 2 – JumpAction.execute()

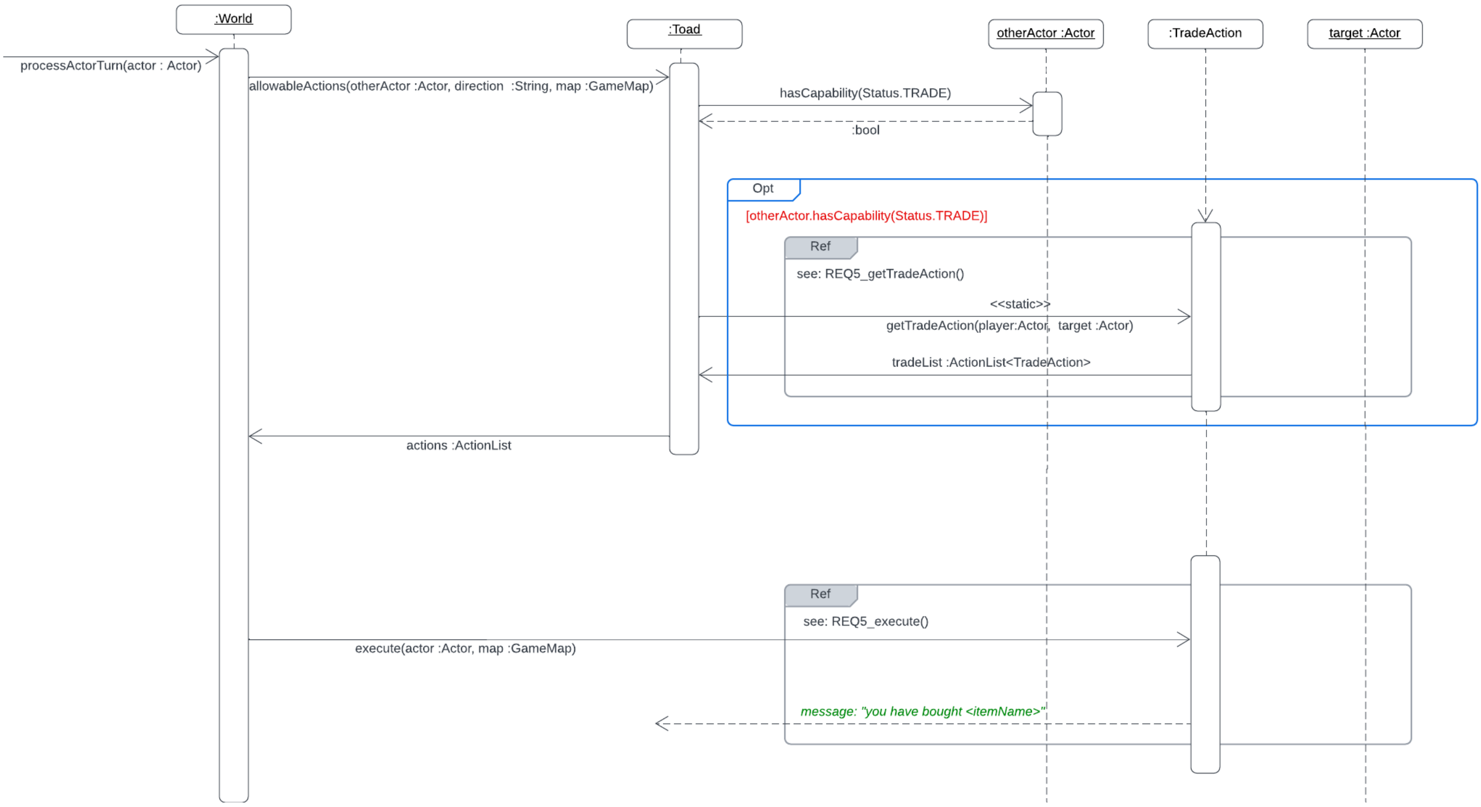

: JumpAction
player: Player
target: Ground
map: GameMap
execute(player, map)
hasCapability(Status.TALL)
isTall: boolean
getSuccessRate()
successRate: int
alt
moveActor(player: Actor, targetLocation: Location)
[isTall or (rand.nextInt(100) <= successRate)]
message: player jumped to target at targetLocation
[rand.nextInt(100) > successRate]
getFallDamage()
fallDamage: int
hurt(fallDamage)
message: player failed the jump to target, player receives fallDamage
isConscious()
conscious: bool
opt
removeActor(player)
[!conscious]
message: Player died from fall damage

Requirement 4 – FlattenAction.execute()

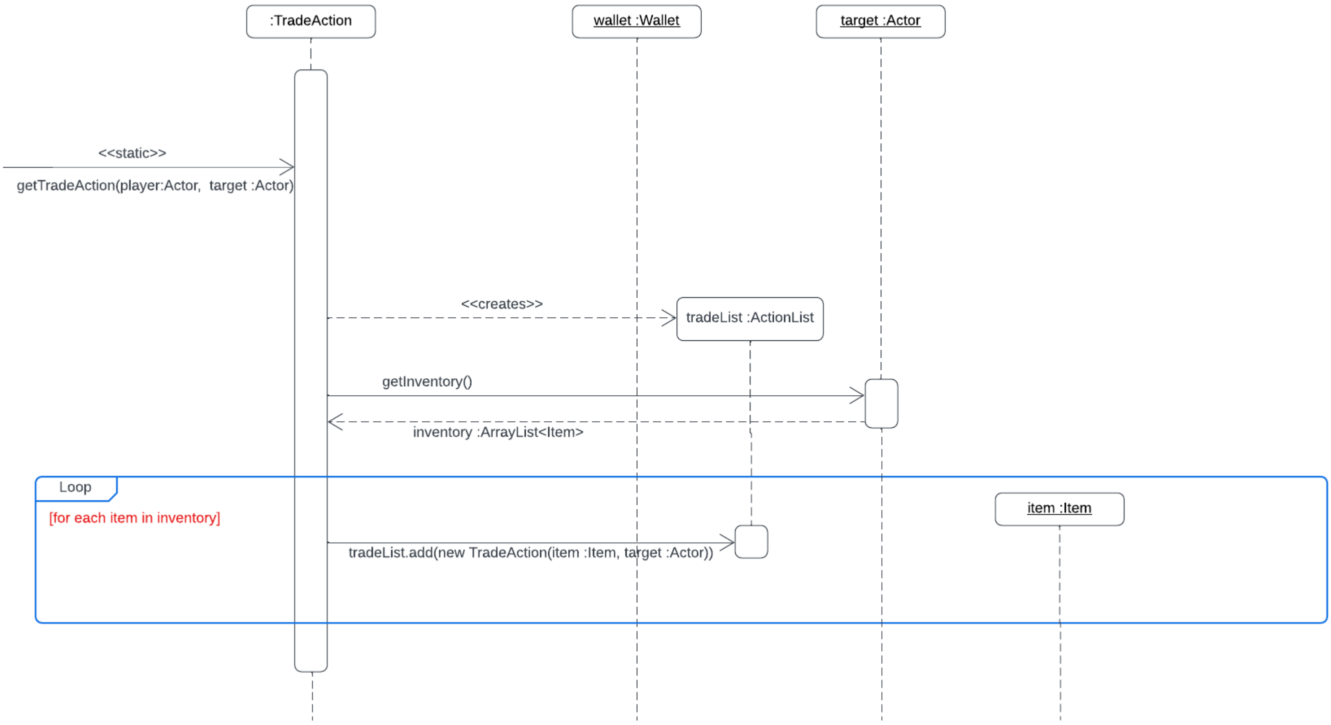

:FlattenAction
player: Actor
target :Location
map:GameMap
execute(player :Actor,map :GameMap)
moveActor(player :Actor, target :Location)
locationOf(player :Actor)
target :Location
setGround(dirt :Ground)
message: Player has flattened <target>
and has moved to targetLocation
<<create>>
coin:Item
addItem(coin :Item)

Requirement 5 – TradeAction

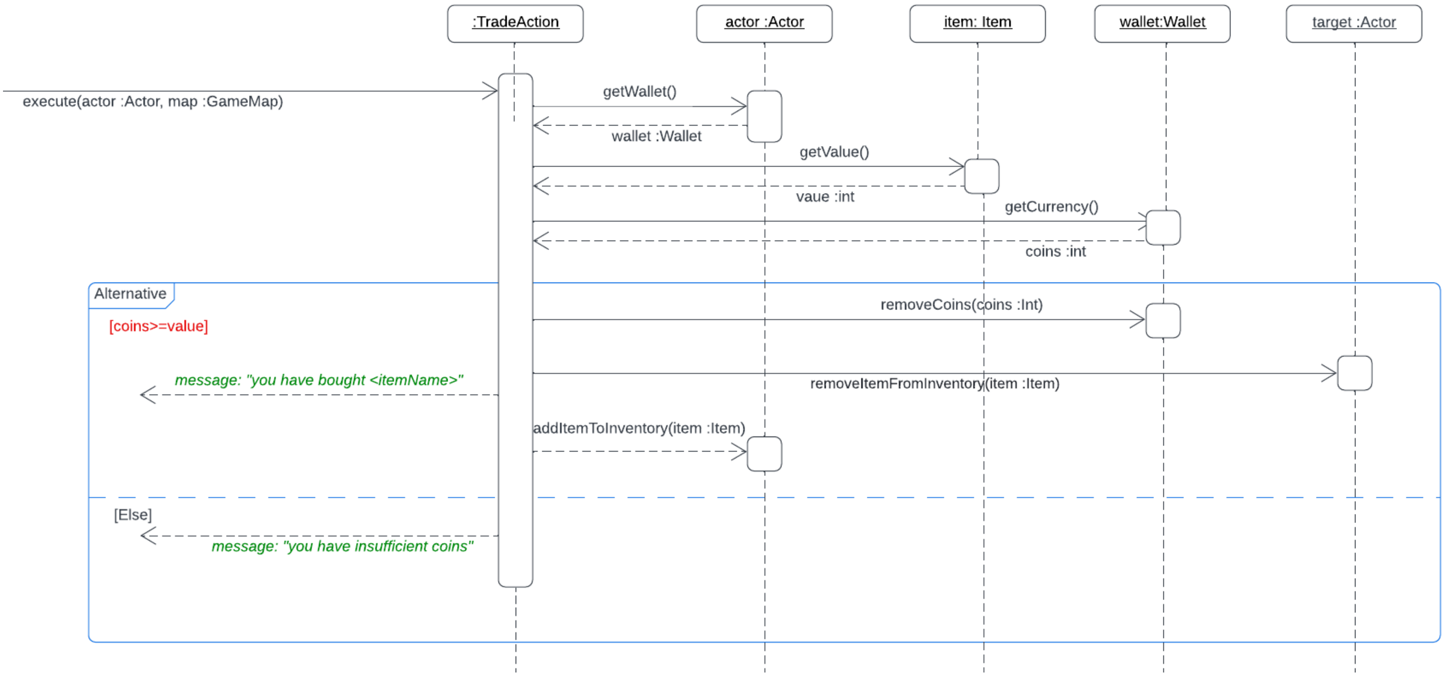
TradeAction Overall Functionality


:World
:Toad
otherActor :Actor
:TradeAction
target :Actor
processActorTurn(actor : Actor)
allowableActions(otherActor :Actor, direction :String, map :GameMap)
hasCapability(Status.TRADE)
:bool
Opt
[otherActor.hasCapability(Status.TRADE)]
Ref
see: REQ5_getTradeAction()
<<static>>
getTradeAction(player:Actor, target :Actor)
tradeList :ActionList<TradeAction>
actions :ActionList
Ref
see: REQ5_execute()
execute(actor :Actor, map :GameMap)
message: "you have bought <itemName>"

TradeAction.getTradeAction()


:TradeAction
wallet :Wallet
target :Actor
<<static>>
getTradeAction(player:Actor, target :Actor)
<<creates>>
tradeList :ActionList
getInventory()
inventory :ArrayList<Item>
Loop
[for each item in inventory]
item :Item
tradeList.add(new TradeAction(item :Item, target :Actor))

TradeAction.execute()



//////REQ1//////

While Sprout, Sapling, and Mature are stages of a tree, they have unique spawning abilities. To obey the Single-Responsibility Principle, it was decided to make them separate classes that inherit the Ground Class.

As Sprout can spawn a Goomba, it implied that it would have a dependency, that is, a method to potentially spawn a Goomba when conditions are met.

Similarly, a Sapling can drop coins every turn, which implies that Sapling knows about Coin, and therefore has a dependency on the Coin Class.

Lastly, a Tree can spawn a Koopa, which is responsible for its dependency on the Koopa Class.

As Sprout grows into Sapling, Sapling grows into Tree, and Tree can spawn Sprouts or become Dirt, Sprout has a dependency on Sapling, Sapling has a dependency on Tree, and Tree has dependencies on Sprout and Dirt.

//////REQ2//////

The JumpAction is implemented similarly to the existing class AttackAction. It also inherits from the base class Action. It has an attribute target of class Ground which forms its association with the Ground class. This is done to obey the Liskov Substitution Principle. In reality, JumpAction will only target Wall, Sprout, Sapling, and Tree which all inherit from the Ground class. Success rates and fall damages will be added as attributes to the relevant Ground subclasses as they have unique values. These values will then be retrieved when JumpAction.execute() is called. As JumpAction already has an association with the Ground class, this design does not add on to the dependencies between classes, and therefore follows the idea of reducing dependencies (ReD).

To further implement the ReD principle, JumpAction will check the consumption of SuperMushroom via the Actor method hasCapabilities() to check if the Player has the Status TALL. Since Action has an existing dependency on Actor, this method does not add more dependencies.

//////REQ3//////

Goomba and Koopa would inherit certain aspects of the Actor class as they both contain attributes that are shared amongst other characters such as health and the ability to move around (except for Toad). As a result, they would have an inheritance relationship with the Actor class. Additionally, Goomba and Koopa have relationships with Sprout and Tree where a Sprout or a Tree have the chance to spawn a Goomba or Koopa respectively.

Koopa would have an dependency with Super Mushroom as when the shell of a Koopa is destroyed, a Super Mushroom is dropped. Furthermore, Goomba and Koopa would implement the Behaviour interface which allows them to have their own 'objectives' such as killing the player.

//////REQ4//////

Both Super Mushroom and Power Star are objects that the player can use to gain a benefit/advantage in the game, as a result, they would inherit aspects of the Item class.

Super Mushroom and Power Star would implement the TradableItem interface which would list them as items Toad has a potential to carry when the player interacts with him to trade.

Koopa would have an association with Super Mushroom as when the shell of a Koopa is destroyed, a Super Mushroom is dropped. A class called FlattenAction would create the functionality where whenever, the player character moves to higher ground, instead of a JumpAction, the target position would be flattened and become dirt. This class would have an association with the abstract class Ground as it would turn Ground objects (Sapling, Sprout, Tree, Wall) into Dirt.

FlattenAction would also inherit attributes of the abstract class Action as it is an action the player can make after using a Power Star item.

//////REQ5//////

Requirement 5 is a more involved example than requirement 6, as it requires multiple actions to be generated simultaneously to populate the menu with trading options for the player. Furthermore, to trade items the functionality of certain items will need to be extended, such that they can have a value and can be identified as being tradeable.

This will be implemented using an interface "TradeableItem", which will add new attributes and methods to certain item subclasses. Foremost, the use of an interface allows for the extension of the sub-classes of Item – which will reduce code repetition/redundancy by using default methods. The use of the interface also enforces a few different SOLID design principles. Firstly, it allows for interface segregation – not all subclasses of the Item class will require the attributes and methods associated with trading, so an interface allows for the extension of specific Item sub-classes. Using an interface in this scenario also ensures that entities derived from the Item class are open for extension but closed for modification. Furthermore, using an interface would allow the methods/classes related to trading mechanics to only accept TradableItems(s) as inputs – which could allow for better exception handling.

In this design the trading mechanics are implemented using the TradeAction class, which is an extension of the Action class. TradeAction(s) are generated within the Toad class through the "getAllowableActions'' function. Here, if the Toad class has the correct capability, it should generate an ActionList, which contains the TradeAction(s). The inventory of the Actor (Toad in this case) trading with the player will be iterated through, and a TradeAction would be generated for each TradableItem in the inventory (each item will lead to different menu prompts and trading outcomes). While it would be possible to implement the logic for this process in the "getAllowableActions" function of the Toad class, this could violate some of the SOLID principles. Instead, as depicted in the sequence diagram, the proposed implementation uses a static method within the TradeAction class to generate an ActionList containing Items, using the "target" Actor class's inventory.

Through this design, the static function allows all of the logic required for creating multiple instances of the TradeAction class to be encapsulated within the TradeAction class itself, instead of existing externally within a single Actor class's method. This improves the reusability of the trade mechanics – if other trading characters were introduced, they would be able to implement trading behavior by using TradeAction methods and having the correct capabilitySet. Furthermore, this is a continuation of the implementation of the Capability system of the game engine – the behavior of the Actor can be primarily defined by its capabilitySet and not necessarily its methods. This design is also an example of the single responsibility principle – by implementing all the trading logic in the TradingAction class, any Actor class is only concerned with the logic required to supply actions governed by its capabilityList and Behaviors. All of this leads to reduced dependency as there is lower coupling and better organization as all functionality pertinent to trading is centralized.

//////REQ6//////

Requirement 5 and requirement 6 deal with similar design problems – both require an action to be generated when interacting with Toad (when proximate to his location in the game world).

Requirement 6 will be implemented using the SpeechAction class, which extends the Action class. The Action will be created from within Toad's "getAllowableActions" using the class's capabilitySet. This class will access the capabilitySet of the player to determine whether a player is holding/has activated a particular item. By determining this, the SpeechAction class can display the appropriate message when its execute method is called.

//////REQ7//////

The ResetManager class manages a list of objects that implements the Resettable interface. For the purpose of resetting the game, Player, Toad, Goomba, Koopa, and Tree will implement the Resettable Interface. Doing so supports the Dependency Inversion Principle, as ResetManager does not directly associate with any low-level modules, but only associates with the Resettable Interface itself. The list of Resettable in ResetManager would then contain any resettable objects.

ResetManager would then have a dependency to GameMap via a method such that it would be able to reset the GameMap to its initial state.