engine

Ground

Actor

game

actors

Koopa

items

Coin

1

Goomba

1

1

spawns

terrain

HighGround

drops

spawns

enums

Status

Tree

Dirt

Floor

Wall

Sprout

1*

Sapling

1*

Mature

1*

Add a Sprout class, Sapling,

Use GroundFactory to spwan new Sprouts at start.

Sprout, Sapling, Tree and Wall are created as an abstract class of HighGround and then Ground to make it more extendable as what does count as a high ground and what doesnt. As a player has to jump on high ground whereas can walk straight onto floor & dirt.
Hence, following the open closed principle. As it open to extension but not modification.

The HighGround's value is stored in status so we can further find out at what stage the high ground has reached like for Trees.

10% chance for goomba to spawn at Sprout(+) every turn
10% chance to drop a 20$ coin at Sapling(t) every turn
15% chance to spawn Koopa every turn
20% chance tree dies and become dirt every turn

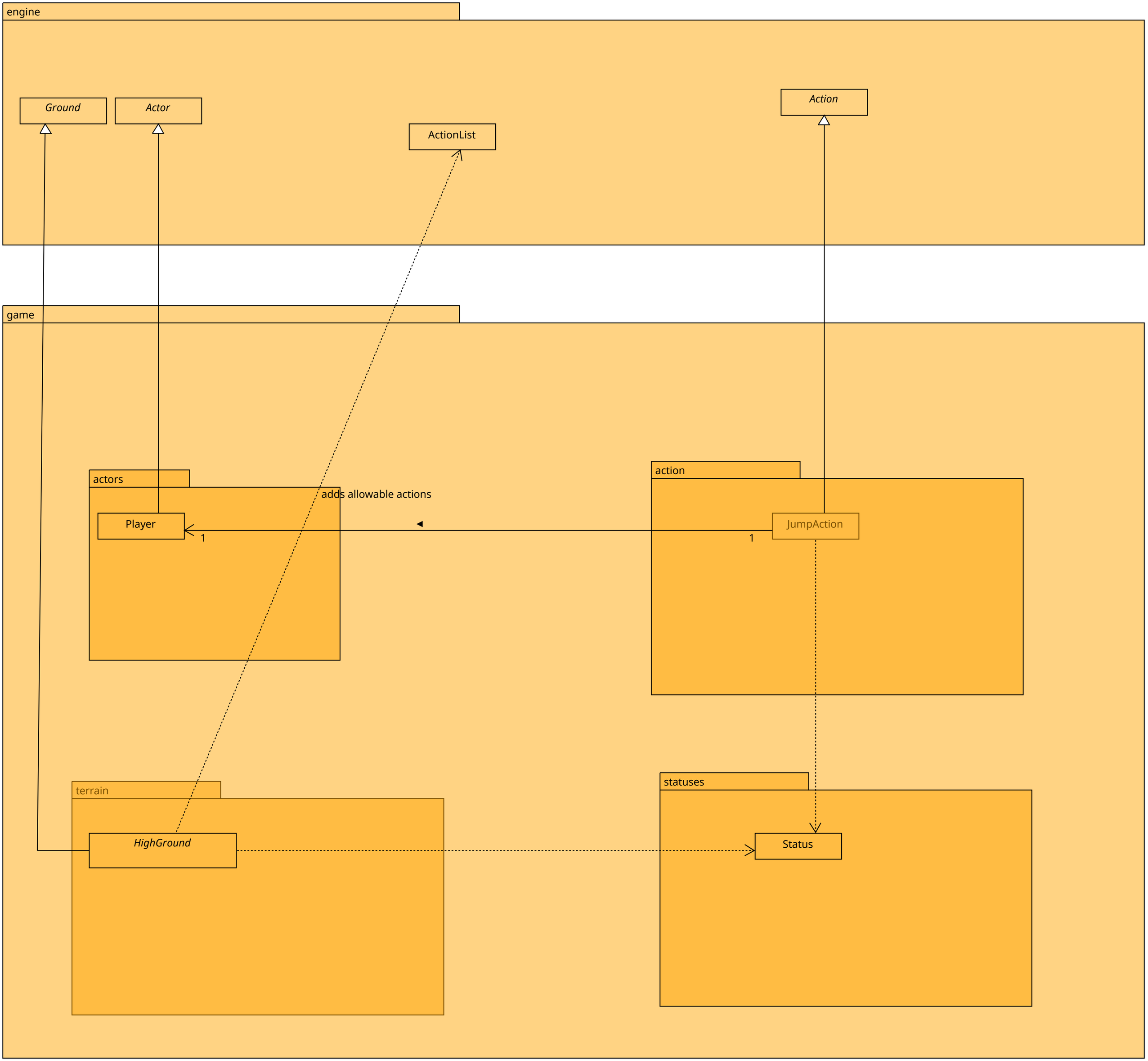
Every turn we have different options possible for each Tree. Which are implemented using the CapabalitySet.

EDIT FOR ASSIGNMENT 2

Modifed and added an abstract Tree class and added all the trees inherit that abstarct class. To make resetting the game easier.
All the trees and high ground inherit the HighGround abstract class to make expanding the game easy. Following the Open/Closed Principle, so its open for extension but closed for modification.

And everything else inherits the Ground Absract class.

Added an association with multiplicty 1* between Sprout and Goomba as multiple sprouts have chnaces of spawning goomba every turn.
Added an association with multiplicty 1* between Sapling and Coin as multiple saplings have chances of dropping coins every turn.
Added an association with multiplicty 1* between Mature and Koopa as multiple Mature trees have chances of spawning Koopa every turn.


engine
Ground
Actor
ActionList
Action
game
actors
adds allowable actions
Player
1
action
JumpAction
1
terrain
HighGround
statuses
Status

Added JumpAction that extends Action. The JumpAction will be used by player to jump over higher grounds.

Engine to wall: What actions is the player allowed
wall to engine: The player can perform JumpAction
engine to player: You can do JumpAction on wall

Jump option will be displayed to the user in the menu.

Each object will have a debuff option.
Like Wall has 80% success and 20 fall damage
Sprout has 90% success and 10 fall damage
Sapling has 80% success and 20 fall damage
Mature has 90% success and 30 fall damage

Dependency between HighGround and Status so we can further find out at what stage the high ground has reached like
for Trees or if the player has consumed Super star which means it can go over anything without jumping.

Dependency between Status and CapabilitySet, as for each HighGround terrain we need to implement what capabilities each
terrain has like the succes rates for each jump of differnet terrain.

Dependency between JumpAction and Status as we need to use this to debuff player depening on different HighGround and different
success rates for each jump.

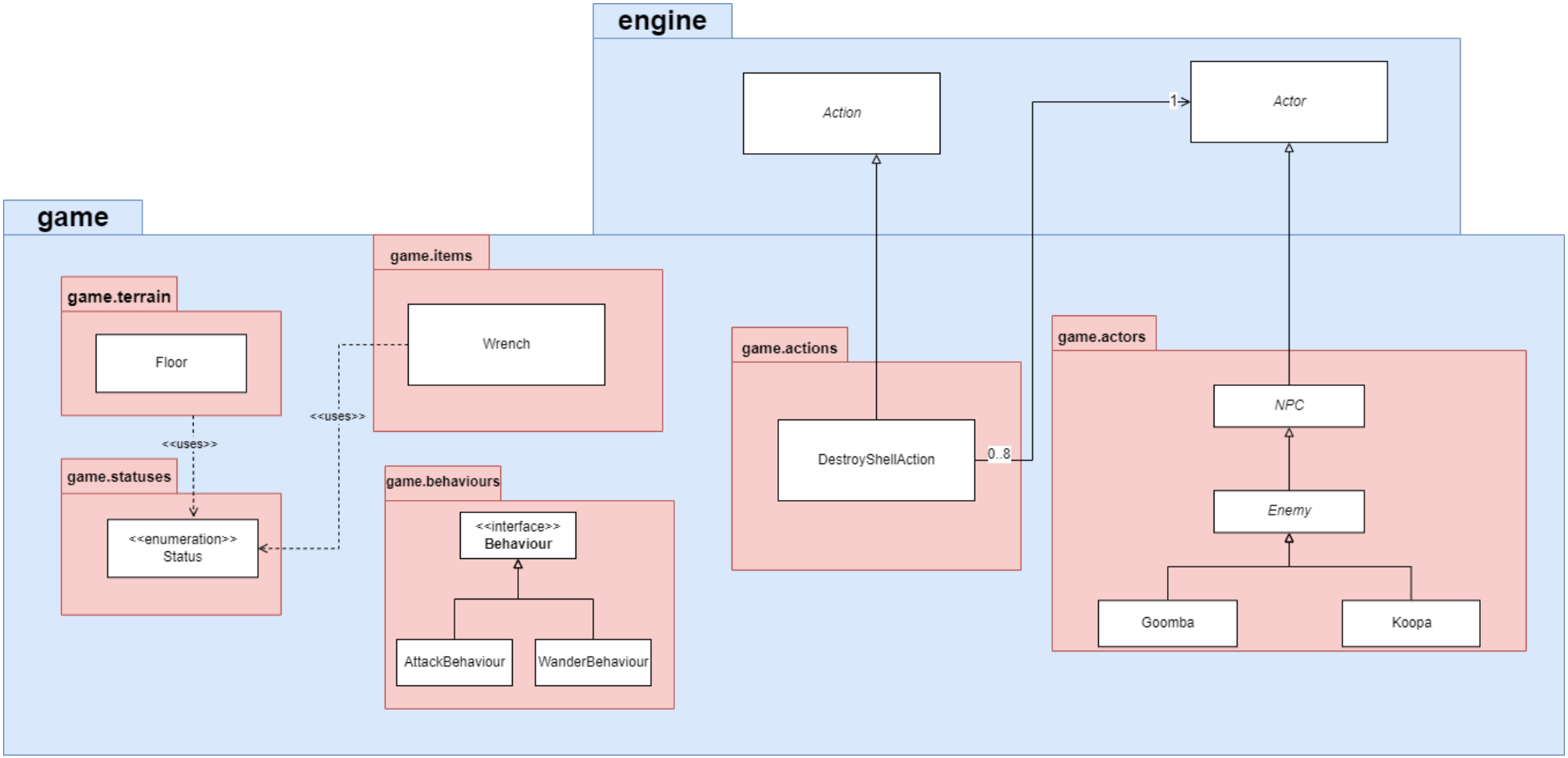
EDIT FOR ASSIGNMENT 2

Modification from previous diagram.
Assocaiation between Player and JumpAction, and a player can use JumpAction to jump once each turn.
Hence the 1 by 1 multiplicity.

HighGround has a dependency with ActionList as with each ground we have JumpAction as an allowable action
which gets added to the ActionList and is accessed by the Player to find out what actions can be
performed.

Requirements 3, 4, 5 UML Diagrams + Documentation by Andy:

Requirement 3 'Enemies' UML Class Diagram



Requirement 3 Documentation

If you believe a relationship between classes is missing, it might be because the relationship is already included in the other UML diagrams

Added NPC abstract class to extend Actor, since all NPCs have a protected TreeMap<Integer, Behaviour> attribute to keep track of the behaviours and the priority associated with the behaviours, friendly and hostile included. Allows for extension. For instance, it is entirely plausible that Toad or some new friendly NPC might have some behaviours in the future.

Added Enemy abstract class extending NPC, reason being that ALL enemies cannot enter floor, and it is likely that in the future there will be some other feature exclusive to enemies implemented, follows DRY.

All future enemy classes added can extend this class

Added Goomba and Koopa class extending Enemy to diagram

Made WanderBehaviour no longer extend Action since it violates SRP trying to handle both Action and Behaviour at simultaneously

Added AttackBehaviour that implements Behaviour, will scan the actor's surroundings. If it finds an actor with HOSTILE_TO_ENEMY it will return an AttackAction

FollowBehaviour is added to the TreeMap of behaviours in allowableActions for the enemies since allowableActions is called in the engine only when the actors are near each other

Modified AttackBehaviour, FollowBehaviour and WanderBehaviour to store a constant PRIORITY indicating the priority of the behaviour in the behaviour system. Avoids connascence of meaning/magic numbers

Modified Floor such that canActorEnter returns False for any actors with CANNOT_ENTER_FLOOR capability. Therefore added a noteworthy dependency between Floor and Status

Both Goomba and Koopa's attack damage are handled by overriding getIntrinsicWeapon

Overriding the PlayTurn method can handle the 10% chance of removing Goomba from the map

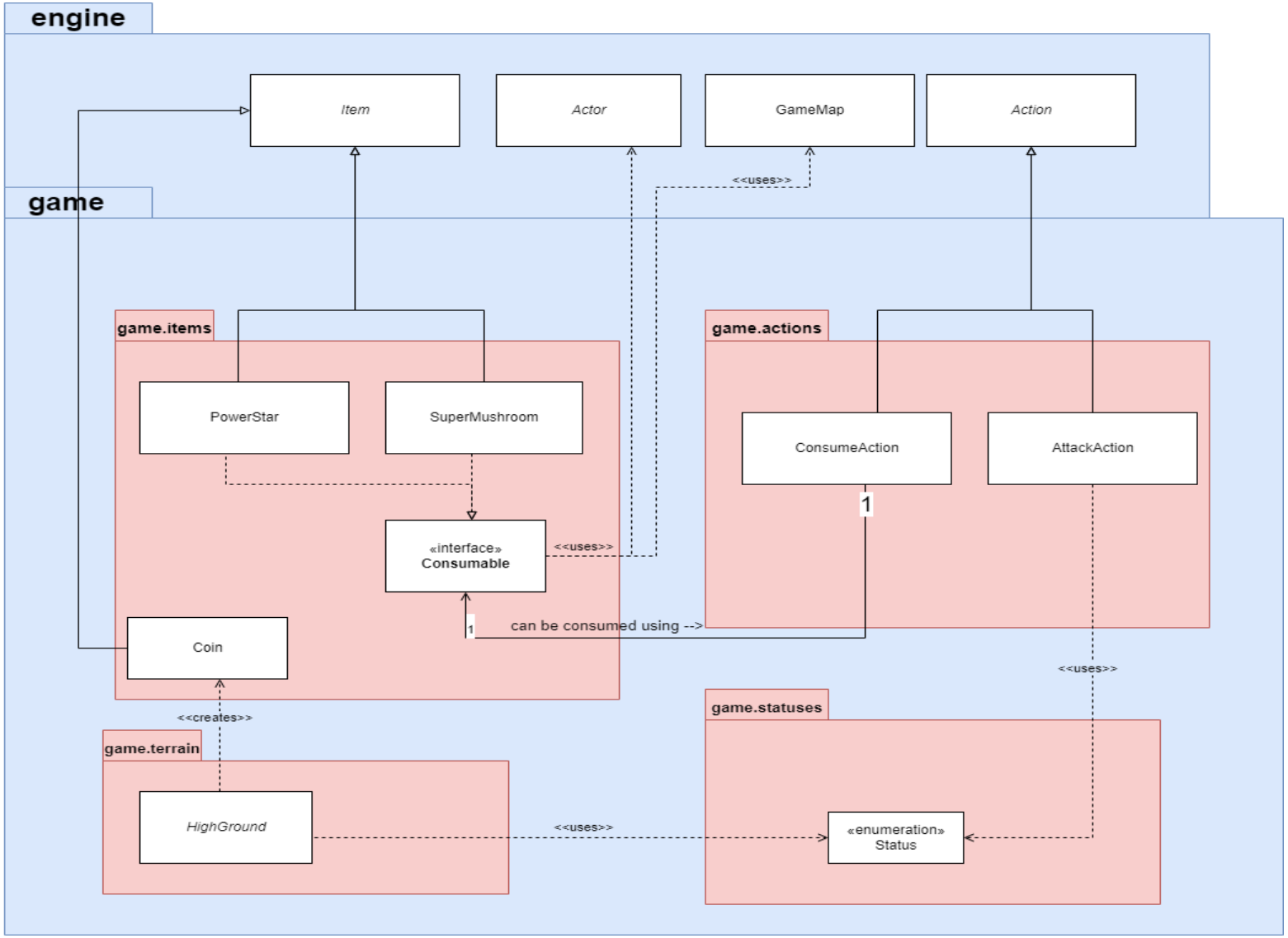
Dormant state will be kept as a Boolean attribute within the Koopa class, and Koopa will behave differently depending on whether it is dormant or not. This is a better way of handling the dormant state than adding a DormantKoopa class because an extra class won't reduce the amount of code/information needed but will create an extra class with unnecessary dependencies/associations.

isConscious method will be overridden for Koopa to always return true, but if its health points is not greater than 0 (checked by using super.isConscious), then the dormant attribute will be set to true, and its display characterwill be changed to D

Wrench's shell destroying capability is implemented by using addCapability in the body of its constructor. Thus, there is a noteworthy dependency between Wrench and Status

Added DestroyShellAction that extends Action. Basically, this action will just remove the target from the map but prints a custom message. DestroyShellAction has association with Actor since it needs to keep track of the target of the action. The 0..8 multiplicity is because theoretically, there can be 8 shells around the player that can be destroyed.

Requirement 4 'Magical Items' UML Class Diagram



Requirement 4 Documentation

Created items package, added Super Mushroom class and Power Star class inside the package, both extending from Item abstract class in the engine

Since Super Mushroom and Power Star are both consumable items, I will add a 'Consumable' interface with a consumedBy method for consumable items so that OCP is followed if any future consumable items are added. Both Super Mushroom and Power Star will implement this interface.

Created actions package and added a ConsumeAction class to handle the action of consuming the consumable items, extends the Action class from engine. Has association with Consumable because it need to be kept as an attribute to use it to call the consumedBy method, as well as to display the correct message in menu

Added dependency between Consumable and Actor

Documenting important dependencies between Consumable, GameMap and Actor:

The consumedBy(Actor actor, GameMap map) method in the Consumable interface will generate the effects of consuming a consumable item (e.g. consuming Super Mushroom grants 50 max hp and TALL status, then the item is removed from inventory, but access to the player is required accomplish this). If the item is consumed from the ground, then we use GameMap to remove it

So far, this diagram follows all the principles. Consumables are easily extendable and existing code do not need to be modified even if a new consumable has new effects of consumption (OCP). ConsumeAction depends on the Consumable interface rather than low levels like the items themselves (DIP). ConsumeAction takes the role of consuming the item, the items themselves take care of their own effects of consumption (SRP)

LSP and ISP are clearly not violated here

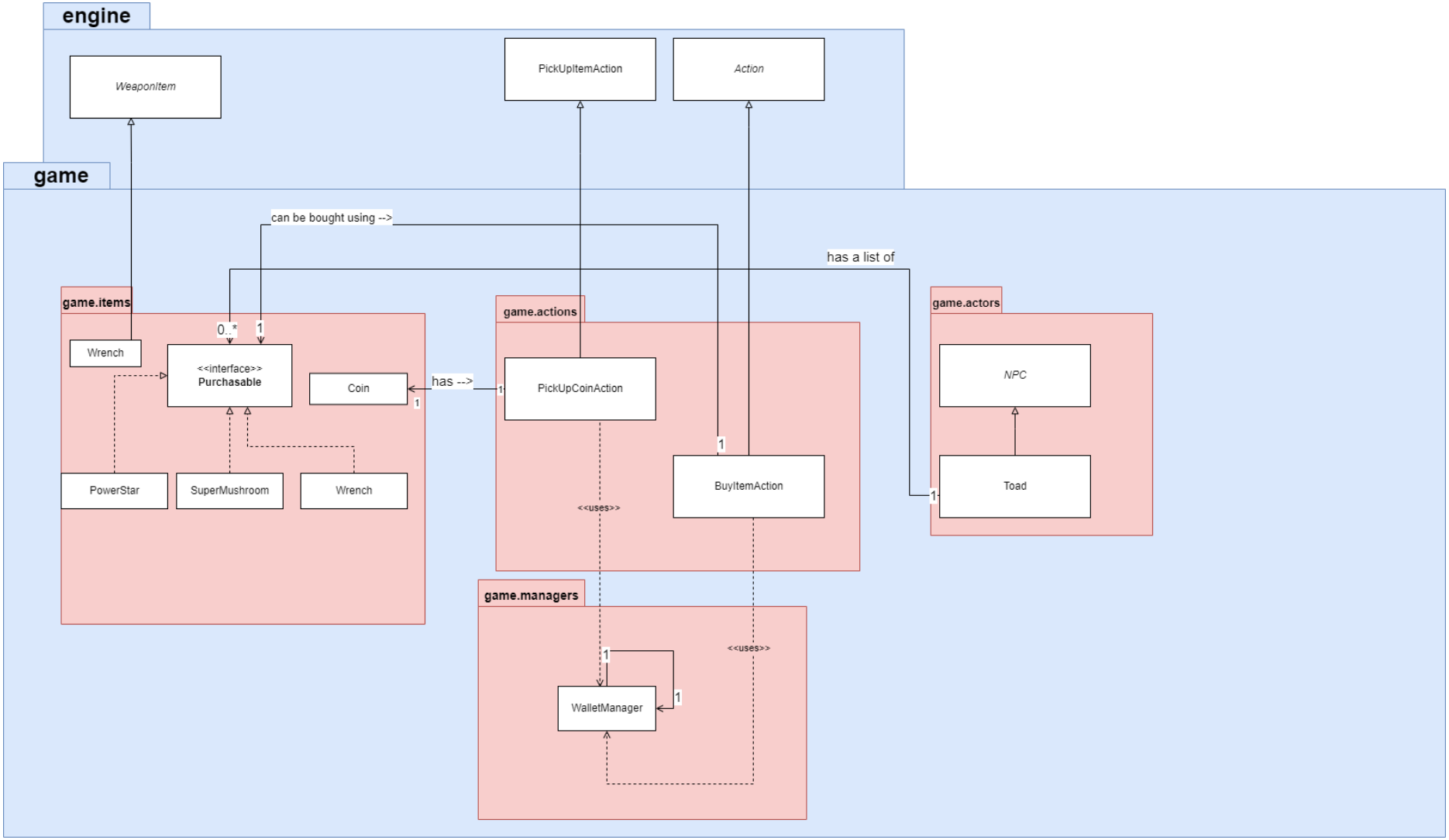
Added dependency between High Ground and Status because canActorEnter should return true if player has the GLOWING Status (from consuming Super Star) and also because High Ground should convert to Dirt IF actor is standing on that location.

The 100% jump chance is implemented by a capability check for TALL in JumpAction

Added Coin class in items package and made it extend from Item abstract class in engine. High Ground has dependency with Coin because tick method should add a Coin to the location of the (former) high ground if a GLOWING player stands on it

Added dependency between AttackAction and Status because of the GLOWING status check when attacking or receiving damage

Requirement 5 'Trading" UML Class Diagram



Requirement 5 Documentation

Recommending reading REQ4 Documentation and UML diagram first since some classes that appear to be missing relationships have already been added in previous UML diagrams

Added BuyItemAction that extends Action

Added singleton class WalletManager that has a private constructor to prevent instantiation and has a private attribute of itself which can be retrieved by calling getInstance. Thus, it has an association with itself

I chose to add WalletManager instead of adding a private attribute called wallet into player so I don't need to downcast Actor to Player when using BuyItemAction to check for wallet. Additionally, by logic, having a singleton wallet manager makes sense because only the player of the game will need access to the wallet. This also allows for extension of the wallet system since it also allows other actors to collect coins FOR the player, such as coin collectors, which wouldn't be possible if we used a wallet attribute for specific actors which requires downcasting.

Added PickupCoinAction that extends PickUpItemAction and handles picking up coins that instantly add credits to WalletManager

Reason for extending PickUpItemAction is that the engine already has the functionalities to support picking up items (getPickupAction)

Overrode getPickUpAction in the Coin class to return a new PickUpCoinAction instead

PickupCoinAction has an association with the Coin class since it needs to be kept as an attribute to remove it from the map as well as show the appropriate message in menu.

The value of the coin also needs to be accessible.

Justification for the 0..* multiplicity between Actor and PickUpCoinAction is that an actor can have multiple PickUpCoinActions if there are multiple coins in the same location.

PickupCoinAction has an important dependency with WalletManager since it must call getInstance then add the appropriate amount of credits corresponding to the coin's worth.

Added Purchasable interface with the methods purchasedBy(Actor actor) and getPrice() so that all buyable items present and future can implement this interface. Follows OCP

The purchasedBy method allows us to add the item to the user's inventory without typecasting

Created Wrench class that extends WeaponItem and implements Purchasable

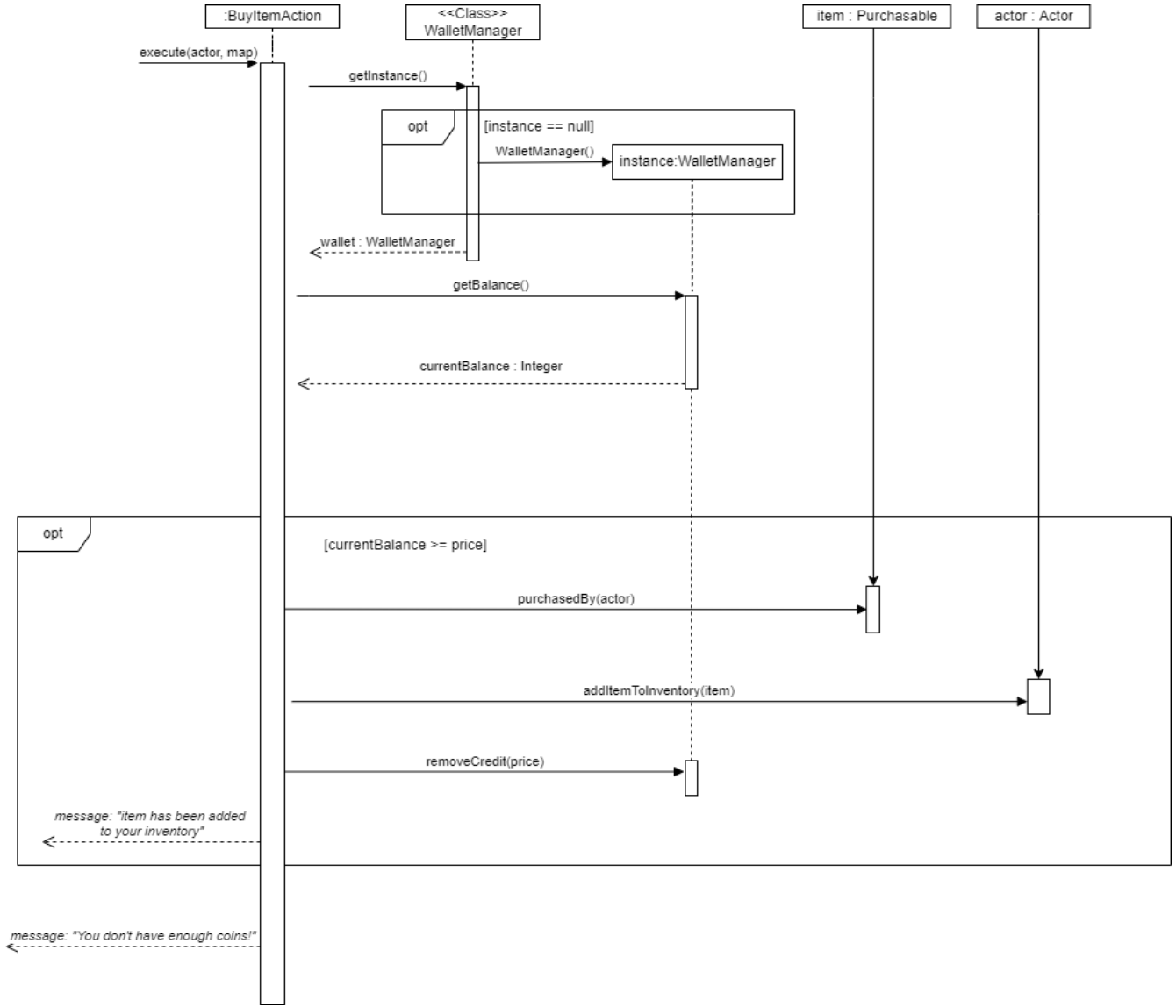
Made PowerStar and SuperMushroom also implement Purchasable

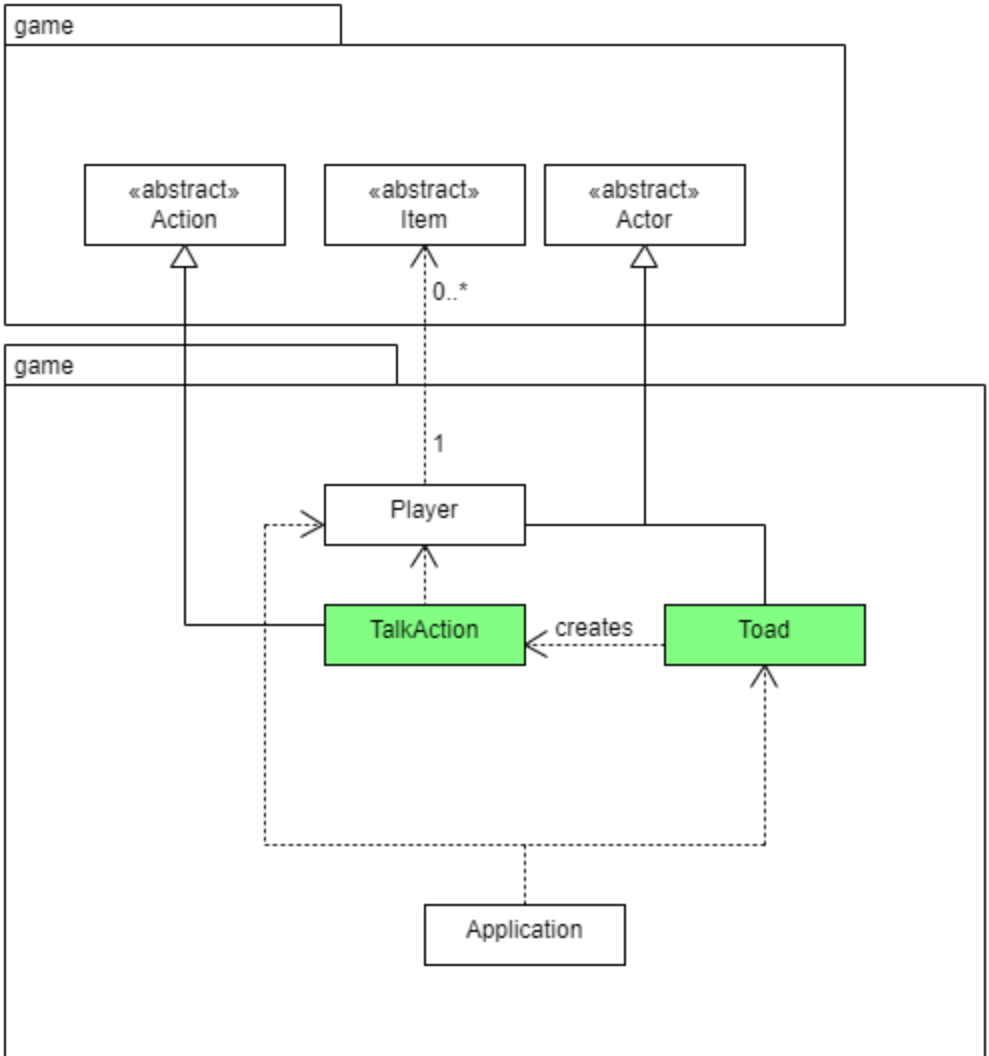
Added Toad class that extends NPC and has an association with Purchasable because he keeps an ArrayList of Purchable items that he can sell by iterating through the array and creating BuyItemActions for each item

BuyItemAction has a dependency with WalletManager and an association with Purchasable because it needs to be able to call the purchasedBy method, (which will add the buyable item to the player's inventory) after deducting the balance from WalletManager, or if there is not enough credits, then the item isn't added.

See BuyItemAction sequence diagram below

BuyItemAction Sequence Diagram by Andy

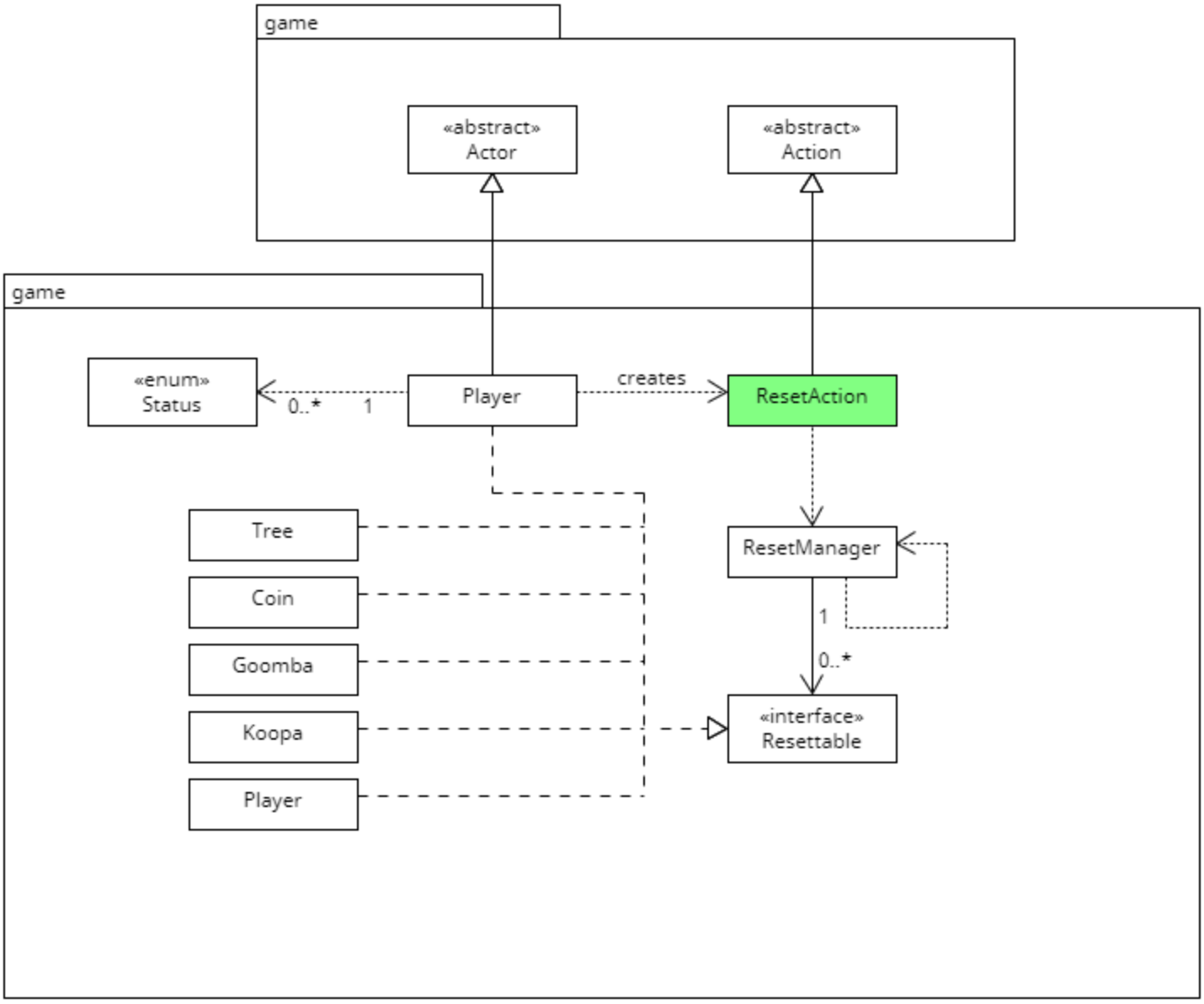



game
«abstract»
Action
«abstract»
Item
«abstract»
Actor
0..*
game
1
Player
TalkAction
creates
Toad
Application

Add TalkAction to the ActionList in Toad's allowableActions method.
This means anytime Player is adjacent to Toad the TalkAction will be allowed.
This follows the system already in place for actions the player can take.

TalkAction is a class that extends from the Action class.
When the execute method is called from TalkAction it will iterate through the players inventory and capabilityList
and will add sentences to a list based on the given requirements not adding the ones that shouldn't be displayed.
It will then randomly select one to be returned which will then be displayed through the processActorTurn method in World class.

This implementation doesn't modify any existing classes and only adds new ones showing and more conditional dialogue options can be easily added
clearly showing the Open-Closed Principle.


game
«abstract»
Actor
«abstract»
Action
game
«enum»
Status
0..*
1
Player
creates
ResetAction
Tree
Coin
Goomba
Koopa
Player
ResetManager
1
0..*
«interface»
Resettable

ResetAction is a new class that extends from Action.
ResetAction will be added to the Player's actionList in the playTurn method if the player currently has the status "CAN_RESET".
The enum "CAN_RESET" will be added to the capabilityList in the Player's constructor.
Once the user has used ResetAction the capability will be removed and as a result will no longer be able to use ResetAction again.

ResetAction class has a dependency on the GameMap and the Player. It will iterate through each Location in the GameMap and check the reset
conditions eg. if location contains tree, if location contains actor. It will then "reset" those things eg. if it contains a tree 50%
chance to reset to dirt. Then it will iterate through the Player's CapabilityList and remove any statuses and heal the player to a maximum.

This implementation will only need to add one extra class. As ResetAction extends from Action no other changes need to be made in order to
implement this class other than adding a few things to the Player class and dependencies are kept to a minimum
clearly showing adherence to the Open-Closed Principle and Dependency Inversion Principle.
Implementing the actual "reset" is made easier as there are many helpful methods in Location and Actor that can be utilized.

EDIT 30/04/22

Now implemented using Resettable interface and ResetManager
This allows for an easy way for the game to keep track of what objects need to be reset and also reduces the amount of
code that is repeated.
Any class that needs to be reset can simply implement the Resettable interface and its resetInstance() method to reset
according to the class's needs.

For objects that need to be removed, that is Coin, Tree, Goomba and Koopa, a boolean attribute "remove" is added.
Then in the tick or playTurn methods it can be removed from the game based on whether remove is true or false.
This amount of dependencies is kept to a minimum adhering to the Dependency Inversion Principle and future objects
that need to be reset can easily be implemented adhering to the Open-Closed Principle.