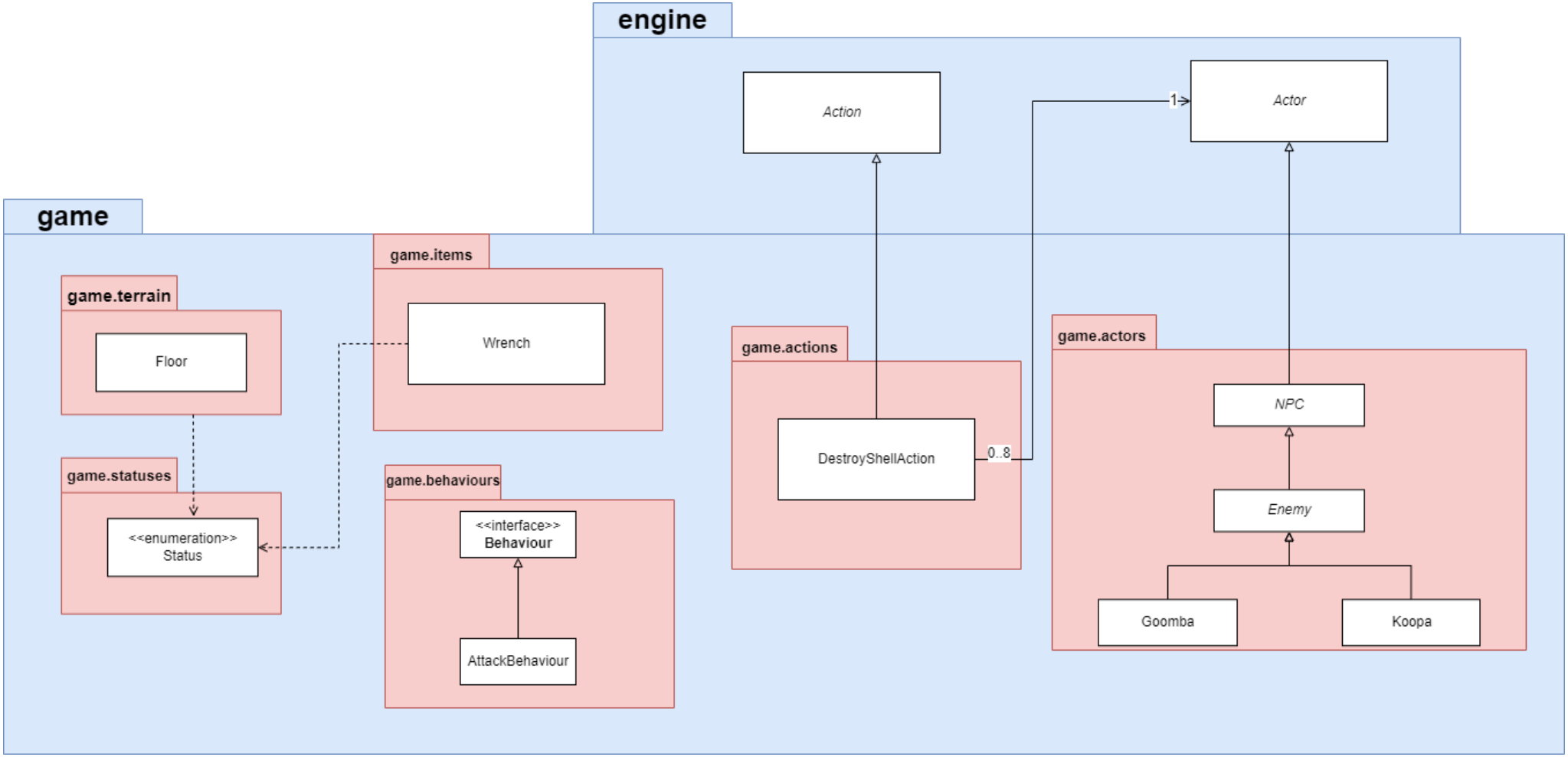
Requirement 3 'Enemies' UML Class Diagram



Requirement 3 Documentation

Added NPC abstract class to extend Actor, since all NPCs have some sort of system to keep track of behaviours (HashMap), friendly and hostile included. Allows for extension.

The HashMap of behaviours will be kept in this class with a getter. Also follows DRY

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.

Enemy's constructor's body adds new capability CANNOT_ENTER_FLOOR

All future enemy classes added can extend this class

Added Goomba and Koopa class extending Enemy to diagram

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

Modify 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 can be handled by overriding getIntrinsicWeapon

Overriding the PlayTurn method can handle the 10% chance of removing Goomba from the map. Additionally, the PlayTurn method can be used to check the enemy's surroundings, if the enemy spots an actor with HOSTILE_TO_ENEMY, a FollowBehaviour will be added to their behaviours

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.

isConscious method will be overriden for Koopa to always return true, but if its health points is not greater than 0, then the dormant attribute will be set to true, and its display character will be changed to D

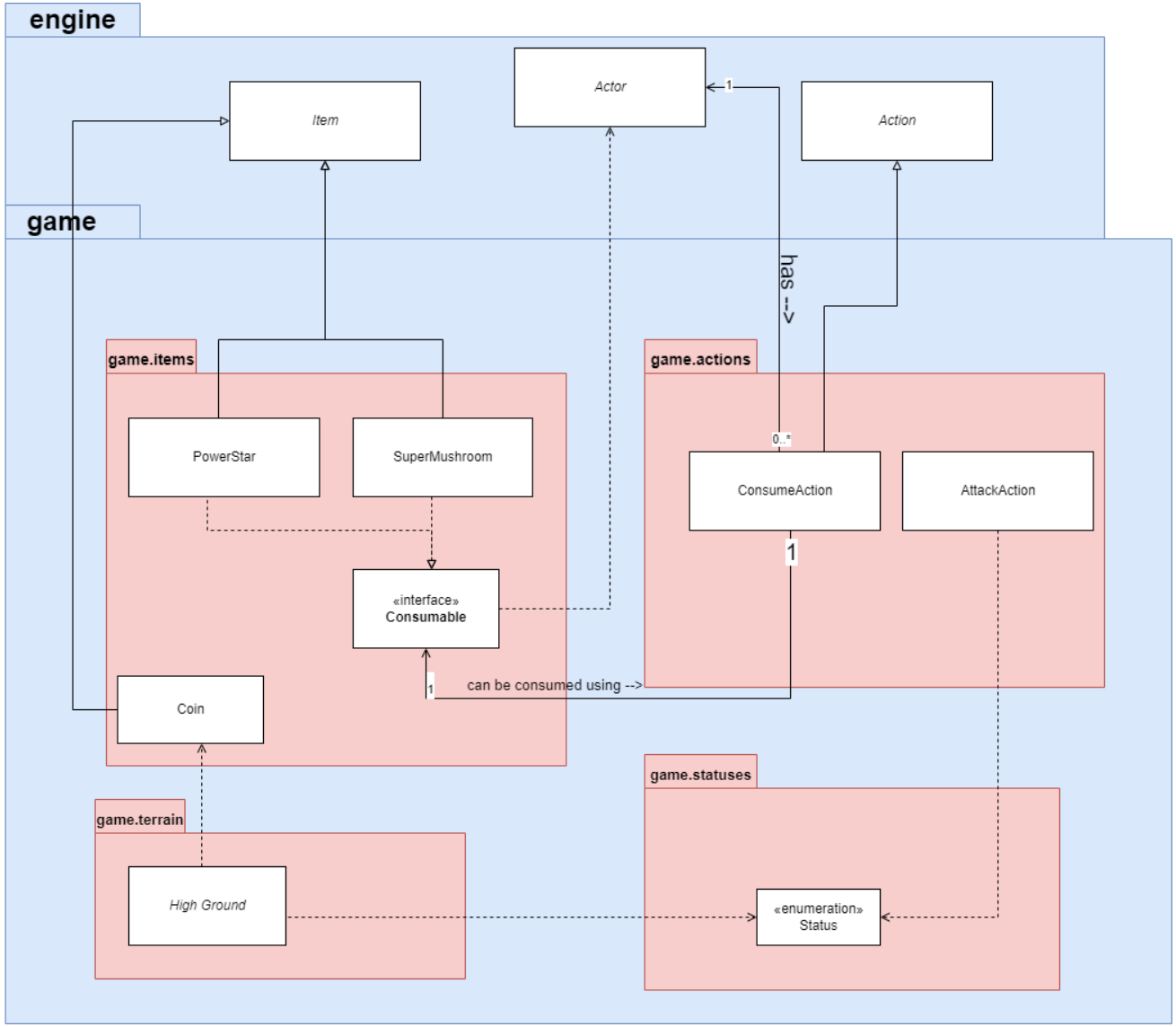
Wrench's shell destroying capability can be implemented by using addCapability in the body of its constructor. Thus, there is a noteworthy dependency between Wrench and Status. (Note: All the relationships missing here for Wrench was already added in REQ5 diagram)

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 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 Actor and Consumable because both need to be kept track of to complete the action.

Justification for 0..* multiplicity between Actor and ConsumeAction is because there can be multiple items in the actor's inventory that may be consumed

Added dependency between Consumable and Actor

Documenting important dependency between Consumable and Actor:

I will add a consumedBy(Actor player) method to the Consumable interface that generates 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)

Can use tick method to generate ConsumeAction and pass the Actor instance

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 the player is standing on that location. (Can be implemented by overriding Tick method)

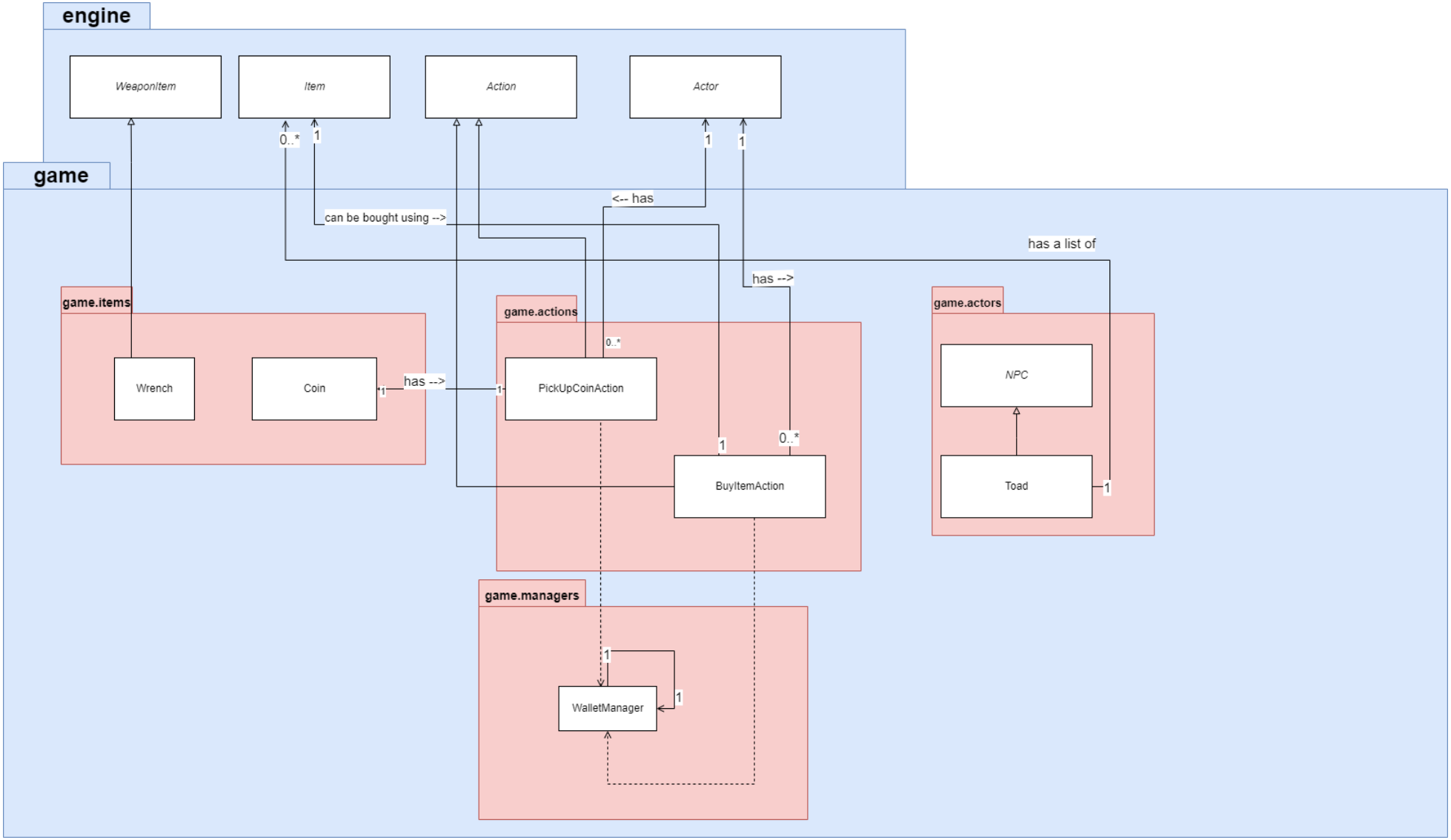
Added Coin class in items package and made it extend from Item abstract class in engine

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

Added BuyItemAction that extends Action and WalletManager

WalletManager 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 handles picking up coins that instantly add credits to WalletManager

PickupCoinAction has association to Actor and Coin class since they need to be kept as attributes to show the appropriate message in menu.

Also, the execute method needs to know the coin to remove the coin from the map afterwards.

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 number of credits corresponding to the coin's worth.

Initially, I wanted to rename Coin to Valuables and make it abstract, create a PickupValuableAction class, then create a Coin class which extends Valuable so that the game can add new valuable items such as Diamonds or Sapphires that give some other functionalities too when picked up. Even though this would follow OCP, I concluded it to be not worth the unnecessary complexity because it seemed highly unlikely a Mario game would add other instantly consumed currency items with different functionalities.

Created Wrench class that extends WeaponItem

Doesn't have relationship between Item and Super Mushroom/Power Star because it's in REQ4 UML Class diagram

Added Toad class that extends NPC and has an association with Item because he keeps an ArrayList of 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 Item and Actor because it needs to be able to add the buyable item to the player's inventory after deducting the balance from WalletManager, or if there are not enough credits, then the item isn't added.

See BuyItemAction sequence diagram below

BuyItemAction Sequence Diagram