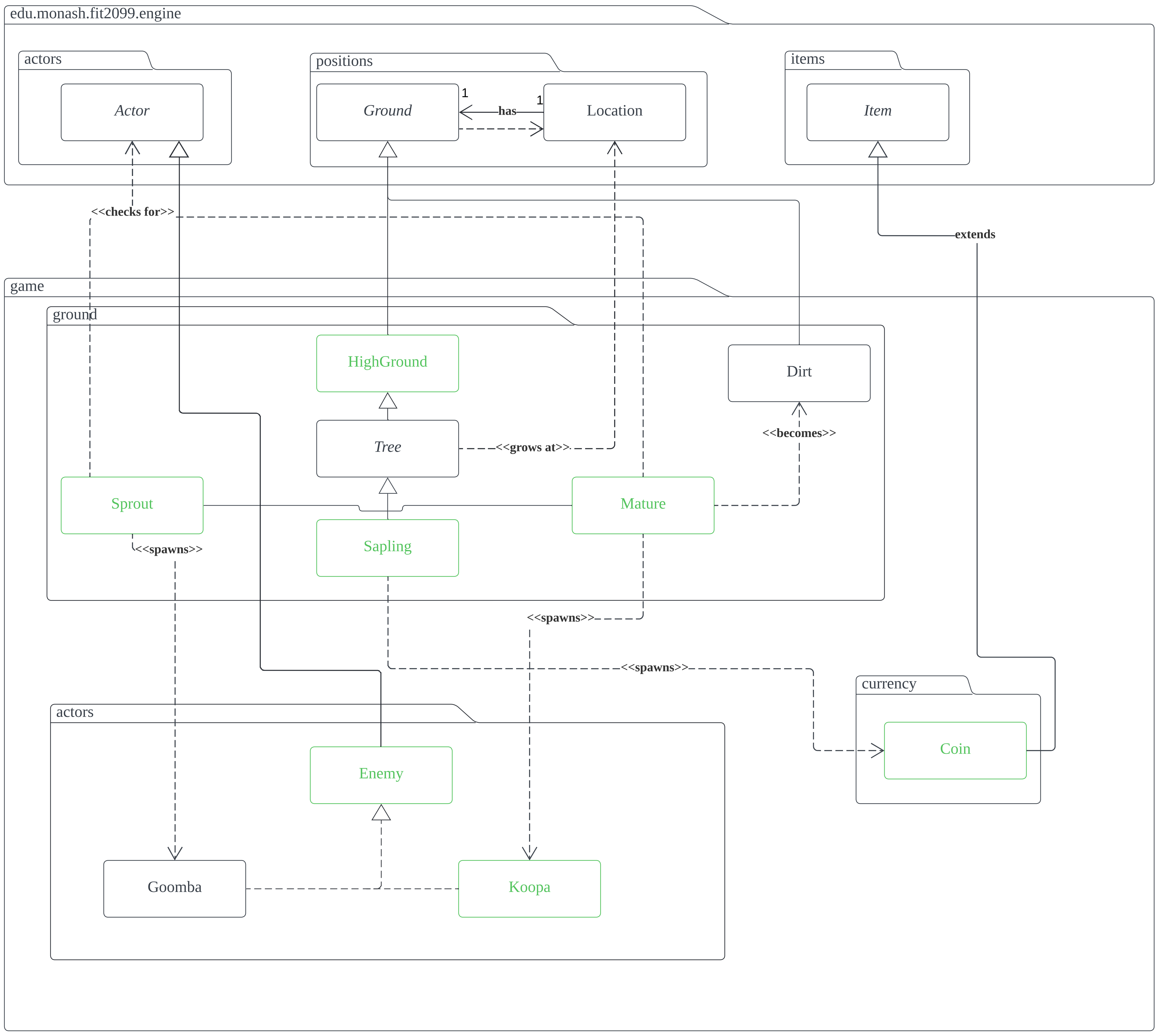
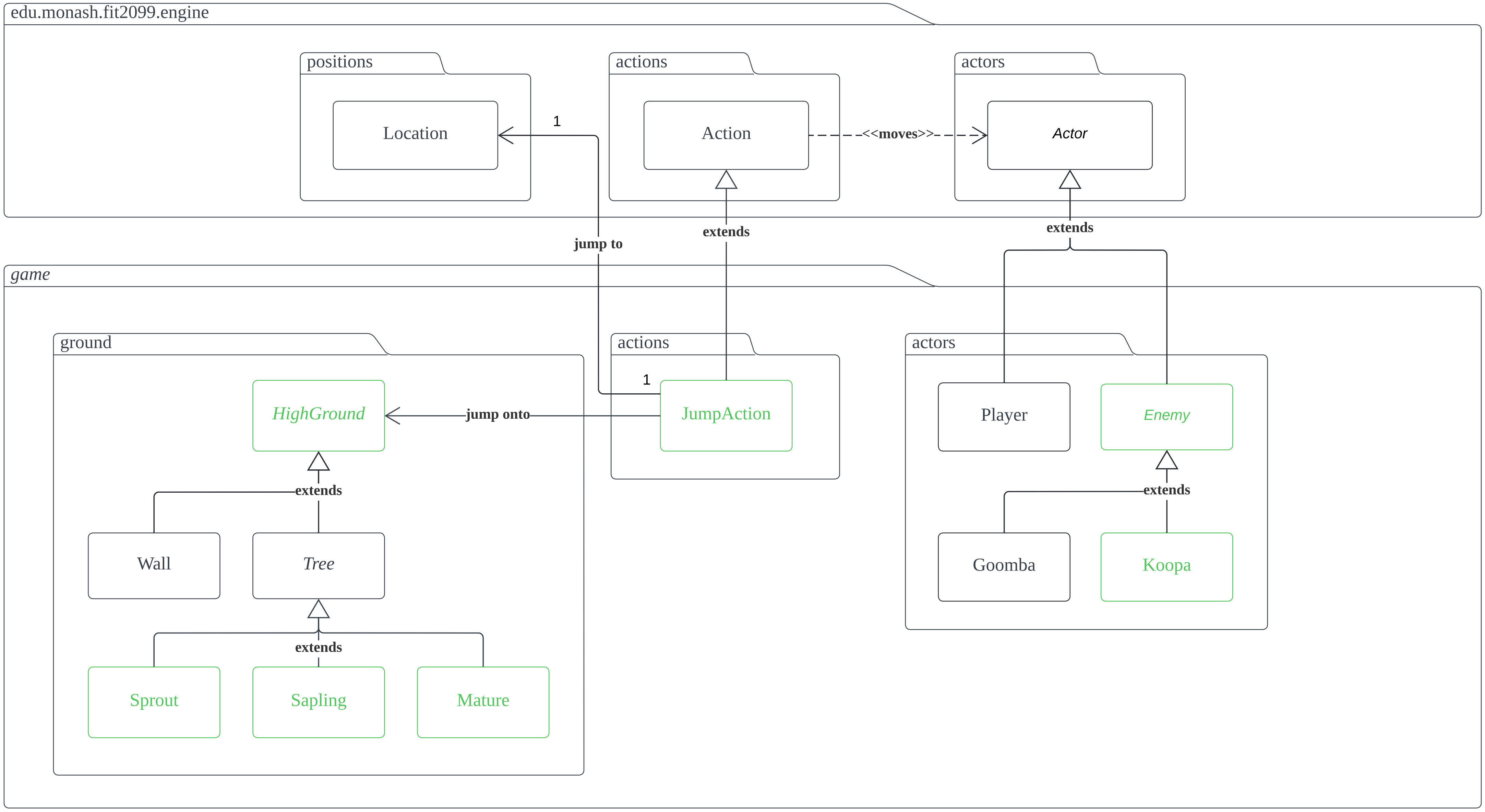
UML DIAGRAMS

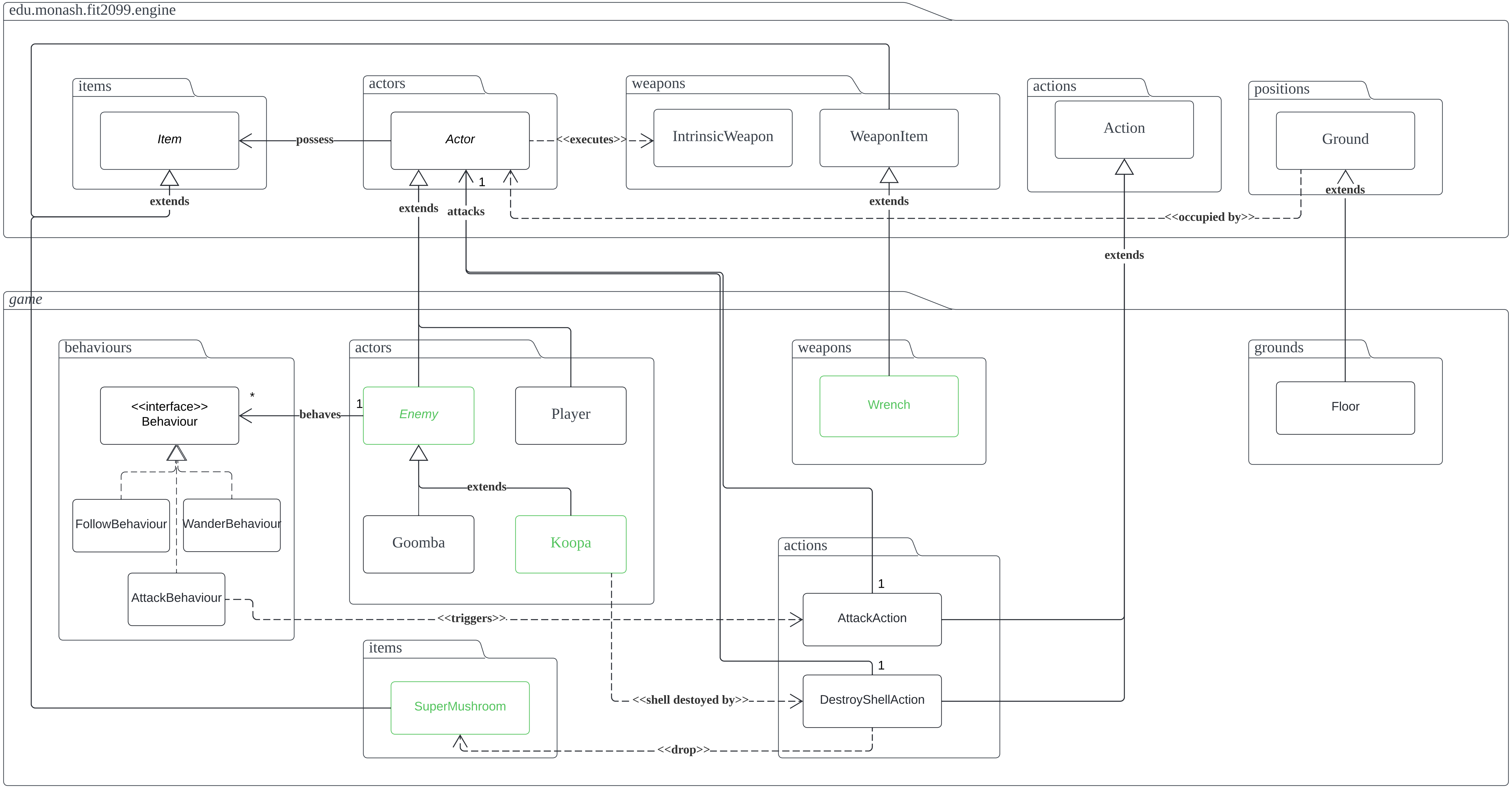
REQ1: Let It Grow



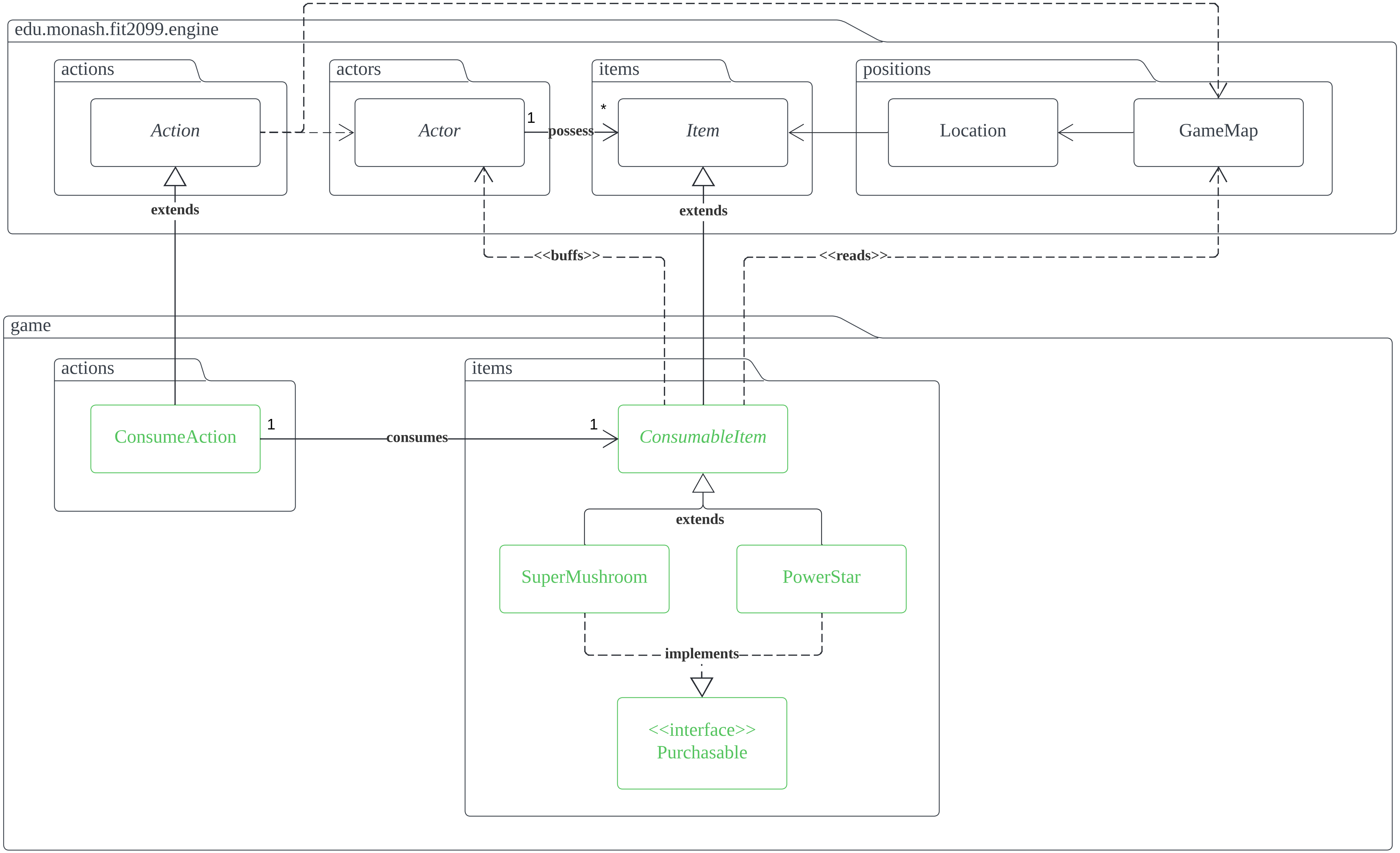
REQ2: Jump Up, Super Star



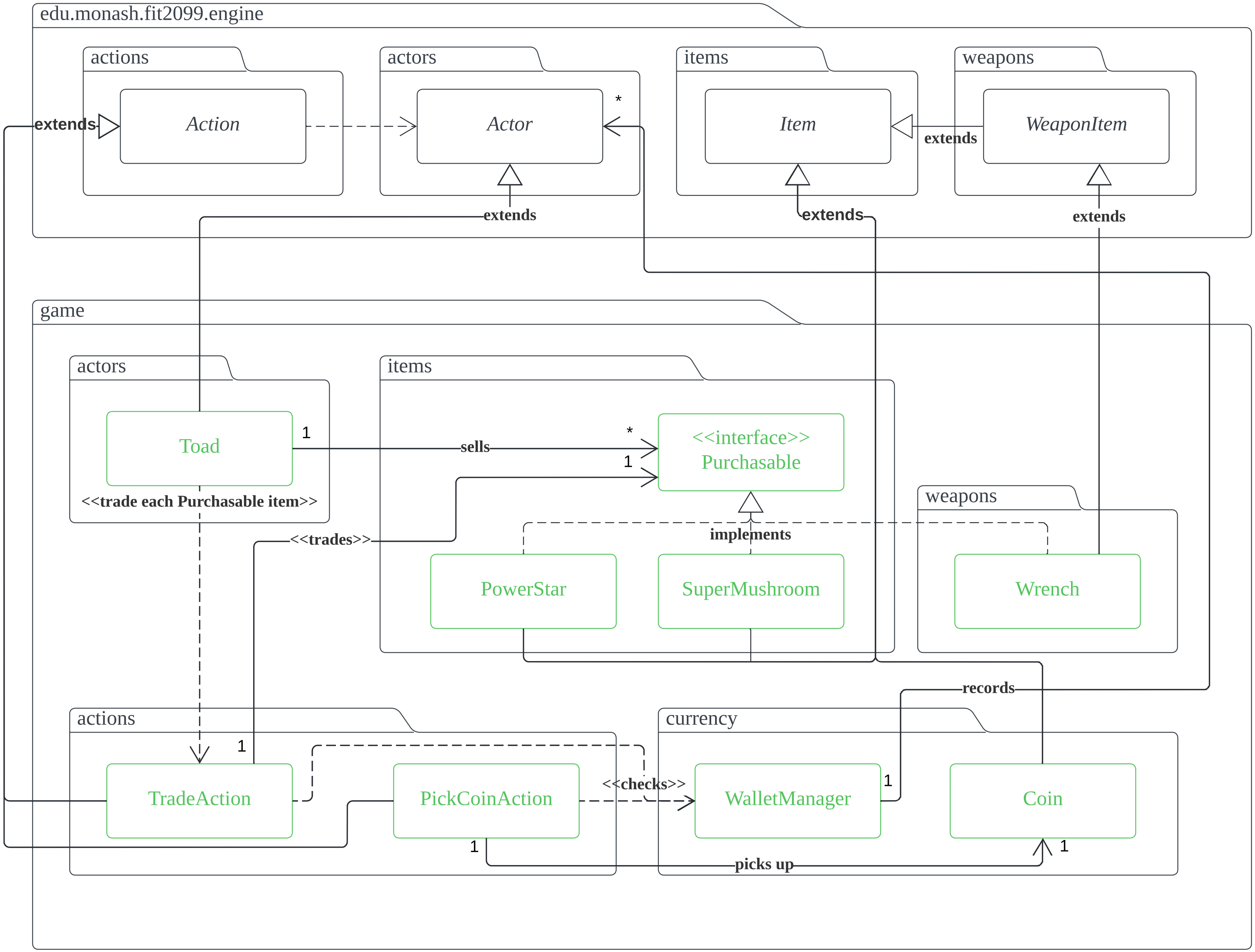
REQ3: Enemies



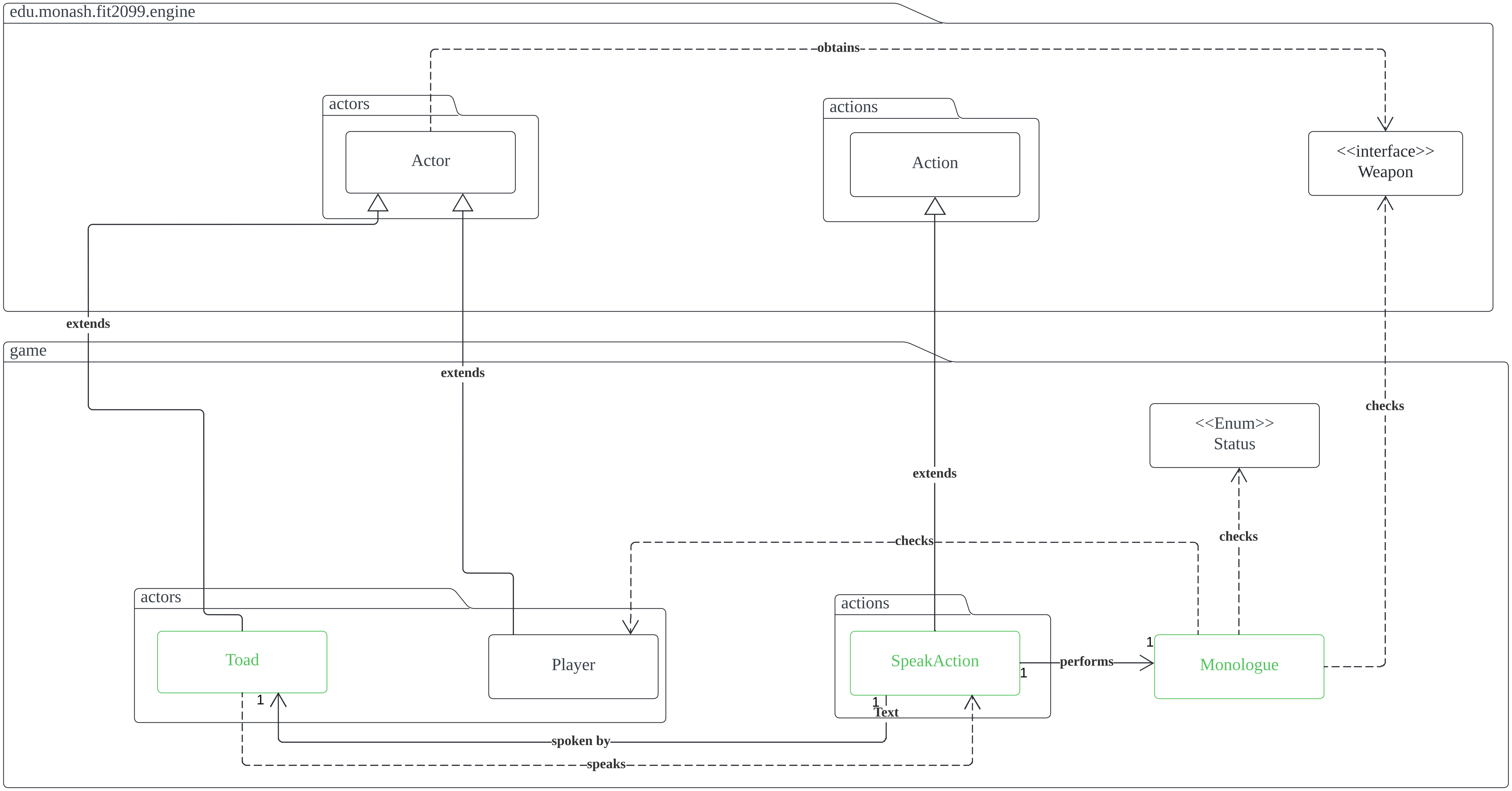
REQ4: Magical Items



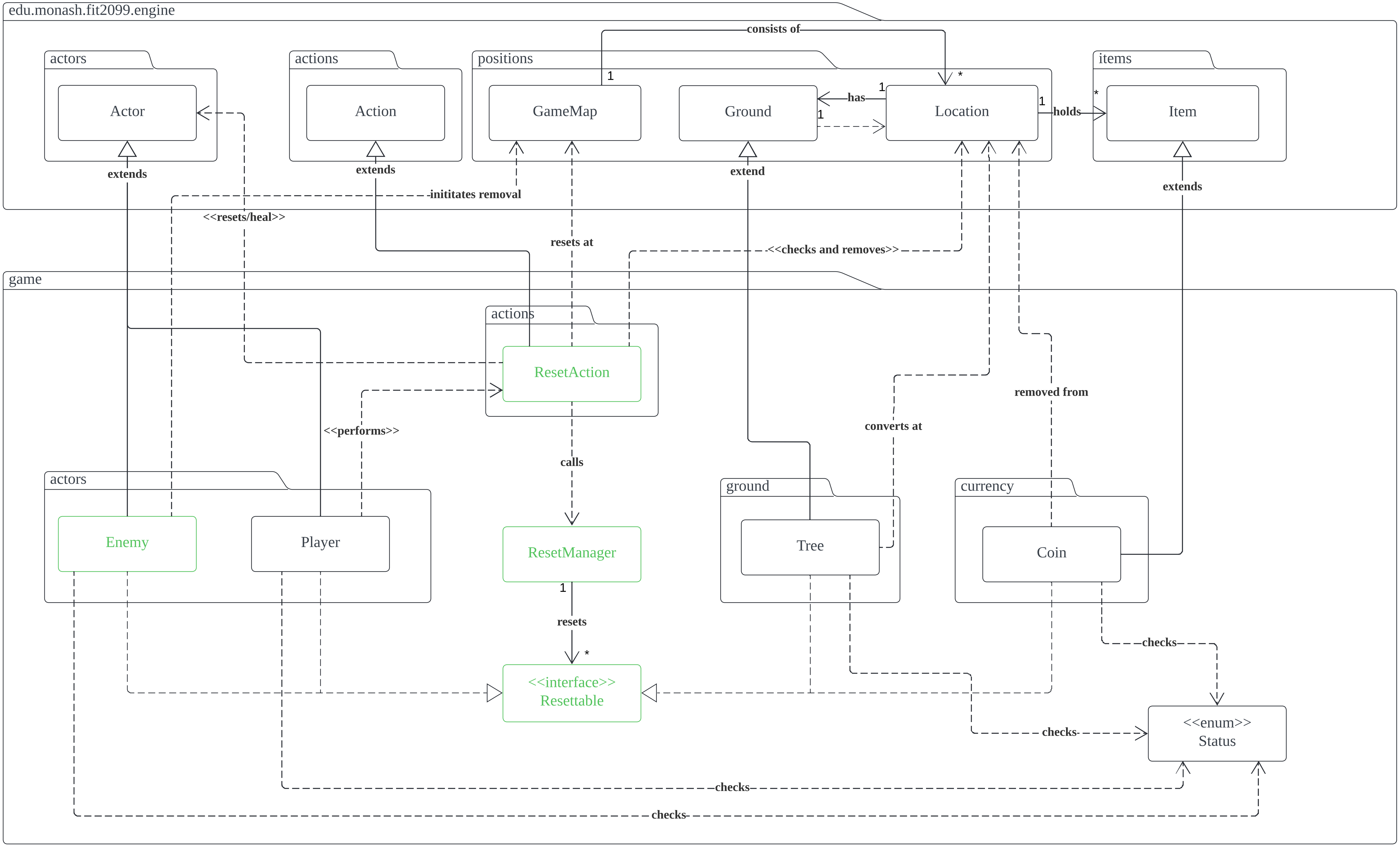
REQ5: Trading



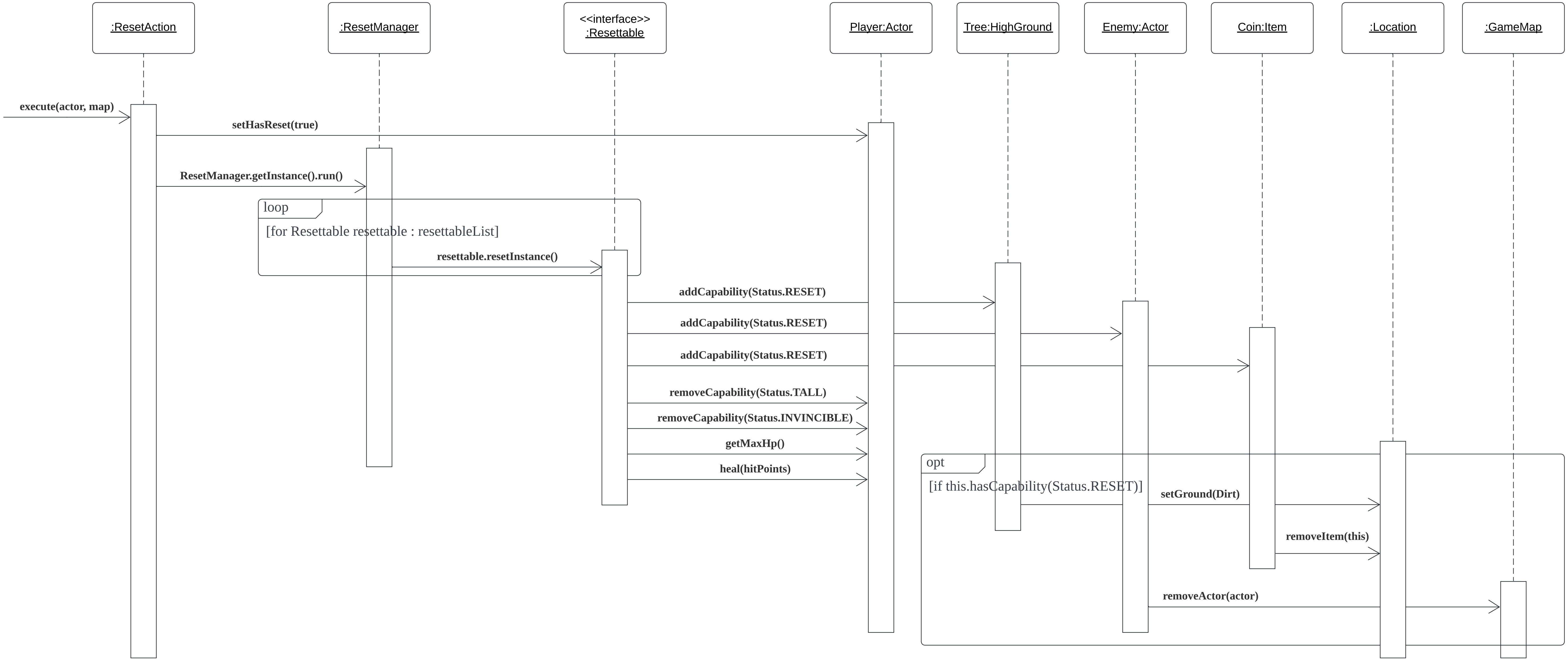
REQ6: Monologue



REQ7: Reset Game



REQ 7 UML Sequence Diagram


:ResetAction
:ResetManager
<<interface>>
:Resettable
Player:Actor
Tree:HighGround
Enemy:Actor
Coin:Item
:Location
:GameMap
execute(actor, map)
setHasReset(true)
ResetManager.getInstance().run()
loop
[for Resettable resettable : resettableList]
resettable.resetInstance()
addCapability(Status.RESET)
addCapability(Status.RESET)
addCapability(Status.RESET)
removeCapability(Status.TALL)
removeCapability(Status.INVINCIBLE)
getMaxHp()
heal(hitPoints)
opt
[if this.hasCapability(Status.RESET)]
setGround(Dirt)
removeItem(this)
removeActor(actor)

DESIGN RATIONALE

REQ1: Let it Grow!
Written by: Matthew Tan, Kar Weng Choong, Jing Ee Lim

Class	Responsibilities
HighGround	- Abstract class that will be inherited by the Tree class.
Tree	- The parent class of the Sprout, Sapling and Mature classes
Sprout	- The first growth stage of a Tree. - 10% chance to spawn a Goomba per turn. - Grows into a Sapling after 10 turns
Sapling	- The second growth stage of a Tree. - 10% chance to drop a $20 coin every turn. - Grows into a Mature after 10 turns.
Mature	- The third and final growth stage of a Tree - 15% chance to spawn a Koopa every turn - 20% chance to convert into Dirt every turn
Enemy	- Abstract class to generalise Goomba and Koopa. - Makes it easier to invoke methods that apply to both Goomba and Koopa.
Goomba	- The enemy actor that will be spawned by a Sapling - Will not spawn if an actor is standing on top of the Sapling
Koopa	- The enemy actor that will be spawned by a Mature. - Will not spawn if an actor is standing on top of the Mature
Dirt	- The state of a Tree before it becomes a Sprout. - Mature converts into Dirt, thus beginning the growth cycle of a Tree.
Coin	- The item that will be dropped by a Sapling

Note: The Location class in the engine is responsible for growing a Tree into its respective stages due to the functionality of setGround(ground) method.

Design Rationale (REQ1)

Requirement 1 did not have many major changes. The only changes are the addition of the HighGround class and the Enemy class, which are required for the later tasks.

Requirement 1 was conceptualised around the **Open/Closed Principle** by extending three new classes from an abstract Tree class to represent its three growth stages. This will allow us to extend new growth stages for a Tree (such as Death) if such a need would arise in the future, without having to modify any single existing class. **Dependency Inversion Principle** can also be seen applied as we have Tree abstract class being extended by the Sprout, Sapling and Mature. Reason being each subclass of Tree may be located at a specific location and doing so allows the Location class to only depend on the abstract Tree class instead of multiple concrete classes. Moreover, each of the growth stages will simply use the method setGround(ground) to imply that the Tree object has 'grown' after a certain amount of turns.
In the future, newer classes added to the growth cycle can simply inherit the Tree class and have its tick() method coded accordingly.

REQ2 : Jump Up, Super Star!
Written by: Jing Ee Lim

Class(es)	Responsibilities
JumpAction	- Jump is a class that allows an actor to move onto higher ground. - An extension of the MoveActorAction class
Location	- It represents the coordinates of the map. - Determines where the Player will jump onto.
HighGround	- An abstract class that represents all types of high grounds in the game, which are ground types meant for a Player to perform a JumpAction onto. High grounds include: a) Wall b) Tree - Sprout, Sapling, Mature
Wall, Tree	- Wall and Tree are types(extensions) of High ground. Meant to be jumped onto. - They each have different jump success rates and fall damage: a) Wall: 80% success rate, 20 fall damage b) Tree: (a parent class, look at child class for details)
Sprout, Sapling, Mature	- Sprout, Sapling, Mature are parts of a process of a Tree in the game. - They determine the success rate and fall damage of the jump action performed by the player: - Sprout: 90% success rate, 10 fall damage - Sapling: 80% success rate, 20 fall damage - Mature: 70% success rate, 30 fall damage
Actor	- An abstract class that represents the characters in the game.
Player	- Player is an actor of the game - Player might have consumed item SuperMushroom that can assist gameplay/enhance actions giving it Status.TALL. Thus, the player's performance will depend on items that they have.
Enemy	- Enemy is an Actor in the game. - An abstract class that contains base attributes and methods of an enemy.
Goomba, Koopa	- Goomba and Koopa are enemies of the game. - They are not allowed to perform jump actions. They are not allowed onto high grounds.
Ground	- An abstract class that represents the Grounds in the map.
Floor	- A ground type that is only accessible by Player and not Enemies.

Updated Design Rationale (REQ2)

REQ2 was updated to have MoveActorAction by Action to maintain consistency throughout the program. Now jumpAction will have an association with Location which was previously associated from MoveActorAction to Location since it must know a Location to jump to. Other than that, the item class which is extended by SuperMushroom and the SuperMushroom class is removed from the UML. JumpAction extends the Action class since it is an action that moves an actor in the game. It must know a location to jump onto and is dependent on the highgrounds and items carried by the player to determine the success rate and fall damage on the player. **Open/Closed Principle** and **Dependency Inversion Principle** were applied in several cases. For example, the game takes the growth process(Sprout, Sapling, Mature) of a Tree into account throughout gameplay since they each have different jump success rates and fall damage. It can also be seen by the abstract enemy class since there are more than 1 kind of enemy. Abstract class Tree and Enemy aligns with the **Open/Closed Principle** since they allow extensions of the class without having to go through modification. In the future, when more Enemy/Tree type characters are introduced. **Dependency Inversion Principle** is achieved by having the JumpAction class point to an abstract class, HighGrounds instead of multiple concrete classes such as Wall, Sprout etc. Hence when jumpAction depends on more highGround types, the jumpAction class will still only need one dependency relationship to HighGround instead to each HighGround type class. Moreover, these examples obey the rules of **DRY Principle** since the content in the Tree class and Enemy class need not to be repeated for each subtypes.

REQ3: Enemies
Written by: Jing Ee Lim

Class	Responsibilities
Actor	- An abstract class that represents the characters in the game.
Player	- Player is an Actor of the game. - The Player may collect weapon items that can be used to fight against enemies of the game etc. Helping them survive
Enemy	- Enemy is an Actor in the game. - An abstract class that contains base attributes and methods of an enemy. - Each Enemy has an IntrinsicWeapon to be executed to attack the player
Goomba	- Goomba is a type of enemy of the game.
Koopa	- Koopa is a type of enemy of the game. - Koopa will drop a SuperMushroom when its shell is destroyed by the player. An instance of SuperMushroom will be created then.
Behaviour	- An interface class that represents the behaviour of an enemy
FollowBehaviour	- A class that represents a type of behaviour (follow behaviour). Allows an actor to follow another actor
WanderBehaviour	- A class that represents a type of behaviour (wander behaviour). Allows an actor to wander around the map
AttackBehaviour	- A class that represents a type of behaviour (attack behaviour).

	Allows an actor to attack another actor.
IntrinsicWeapon	- A final class that represents an attack by an actor unarmed.
Item	- An abstract class that represents collectable items in the game.
SuperMushroom	- SuperMushroom is an Item to be collected throughout gameplay, when consumed it will allow 100% jump success rate for the player without fall damage - It is dropped by a Koopa when its shell is destroyed
WeaponItem	- An abstract class representing items to be collected throughout the game to help perform attacks
Action	- An abstract class that represents the action of an actor.
AttackAction	- An action to be performed by an actor to attack an actor.
DesrtoyShellAction	- An action to be performed by an actor to destroy Koopa's shell
Ground	- A class that represents the grounds in the map
Floor	- A ground type that is not accessible by Enemy

Updated Design Rationale (REQ3)

REQ3 was updated to have another behaviour which is AttackBehaviour which will trigger an Actor of the game to attack. Furthermore, SuperMushroom will not be instantiated in the Koopa class but instead in a new DestroyShellAction that will be called and executed to destroy Koopa's shell, then when destroyed, it will create an instance of the SuperMushroom. Each actor of the game will have an intrinsic weapon to perform an attack. A player however may collect additional weapon items along the game and use them for attacks. A wrench is a weapon item and can be used to attack a Koopa, upon destroying a Koopa's shell will produce a SuperMushroom which is also a collectable item. An enemy will have behaviours, follow behaviour, wander behaviour and attackBehaviour. Enemies also have an attack action on actors of the game. **Open/Closed Principle** can be seen applied. One of few examples include the Wrench class extending the WeaponItem abstract class since we know an actor may possess multiple items one of them being a wrench. That said, when more items are to be added to the game, it can too be extended from the item class. Another principle, **Dependency Inversion Principle** can be seen by having the behaviour interface class responsible for all behaviours of the enemy, in this case a follow behaviour, a wander behaviour and an AttackBehaviour. This allows abstract class Enemy to only depend on a single interface class, Behaviour. Instead of pointing to multiple behaviour classes even more behaviours are added to the game.

REQ4: Magical Items
Written by: Kar Weng Choong

Class	Responsibilities
ConsumeAction	- An action performed by actors to consume a ConsumableItem.
ConsumableItem	- An abstract class to represent items that are consumable.

	- Implements an abstract method consume, to be overridden in subclasses for the effect of buffing the consumer with the effects of a consumable item.
Purchasable	- An interface representing Items that are Purchasable, i.e, to be sold by Toad.
SuperMushroom	- A consumable item, provides a Player with buffs: a) increased max HP, b) better jumps, and also c) changes the display character of a Player.
PowerStar	- A consumable item, heals a Player and provides them with buffs: a) can walk on higher grounds without jumping (higher grounds turn to Dirt and drops a Coin), b) immune to enemy attacks, c) instant-kill attack on enemies. - Keeps track of its own lifetime in the game (max of 10 turns). - Displays its remaining time in the game map or inventory, after which it is removed entirely if not consumed.

Updated Design Rationale (REQ4)

REQ4 was updated to show only the main classes relevant to the design, not implementation of the requirement. In other words, the UML diagram now focuses on the magical items and their consumption only, instead of the implementation of logic behind the status effects granted by each consumable item. Here, we applied the **Open/Closed Principle** by introducing an abstract class ConsumableItem for further extensibility in the future, so we could add any amount of items that can be consumed by an Actor in the game. An interface Purchasable is also implemented by these magical items as they can be sold by Toad, therefore allowing us to add more items for Toad to sell in the future.

REQ5: Trading
Written by: Kar Weng Choong

Class	Responsibilities
Toad	- A peaceful Actor that remains stationary in the middle of the map, selling useful Items in exchange for coins: a) Wrench, b) Super Mushroom, and c) Power Star.
TradeAction	- An action performed by actors to trade with another actor on the map. - Prints the items available for sale by Toad. - Checks if a Buyer has sufficient balance to purchase an Item. - Adds the item to their inventory and deducts the Buyer’s balance if it is sufficient. - If not, prints a message in the UI indicating insufficient balance.
WalletManager	- A singleton class to manage a list of Actors that own a wallet.
Purchasable	- An interface representing items that can be sold by Toad. - Contains methods to be implemented by each Item sold.
Wrench	- A weapon item sold by Toad, costs 200 Coins.

SuperMushroom	- A consumable item sold by Toad, costs 400 Coins.
PowerStar	- A consumable item sold by Toad, costs 600 Coins.
Coin	- An item that represents the currency in the game. - Has a value ($) attached to it.
PickCoinAction	- An action performed by an Actor to pick up a Coin, after which increases the Actor's wallet balance.

Updated Design Rationale (REQ5)

REQ5 was updated to exclude the Player class, as Toad can trade with any Actor that is also a buyer. The Wallet class from the previous diagram was also updated with a WalletManager. Wrench was relocated into its own package "weapons", and all items sold by Toad are now implementing a Purchasable interface. Nevertheless, the concept remains the same with the **Dependency Inversion Principle** and **Open/Closed Principle** being the main motivation behind the implementation of the Purchasable interface, as it allows us to extend for more purchasable items for sale in the future without the need to modify TradeAction. Not only that, Toad stores a list of Purchasable items instead of storing multiple concrete classes in multiple attributes, thus reducing associations.

REQ6: Monologue
Written by: Matthew Tan Yau Koon

Class	Responsibilities
Player	- The main actor who will be interacting with the Toad. - The Player will interact with Toad, who will initiate the monologue.
Toad	- The actor who will be performing the SpeakAction class. - Will speak one of four lines by random, and certain lines are excluded depending on whether the Player has a Wrench in its inventory or is in a powered up state.
SpeakAction	- The main class will contain the necessary methods and attributes to perform the act of speaking. - Prints one of the four possible lines that Toad will speak.
Monologue	- A helper class that stores the possible monologue to be spoken by an Actor. - In this case, this class stores the monologues spoken by Toad. - This class contains the logic of which dialogue to exclude, based on the Player's condition. - Uses the Status enum to identify if the Player is currently under the effects of a Power Star.
Status	- The enum to check if the player is currently under the effects of a PowerStar item.

Design Rationale (REQ6)

Requirement 6 was updated to exclude the Display and Wrench class as we can just use the attributes of a Player instance to check if the Player is holding a Wrench object with the help of the getWeapon() class. The Display object was removed as the engine has already standardised the printing function. The Toad class will now add SpeakAction as one of its allowable actions in the allowableActions method. It was found to be easier to implement a separate Monologue class as it further emphasises the Open-Closed Principle, in which additional methods can be added into the Monologue class, if there comes a time where other Actors from Toad can speak. Additionally, the Liskov Substitution Principle The Monologue class will now contain the logic (i.e. checking Player's weapon and capabilities) before filtering out the monologue to display in the menu.

REQ7: Reset Game
Written by: Matthew Tan Yau Koon

Class	Responsibilities
Player	- The main actor that will invoke a method from the ResetAction class. - When reset, all capabilities are reset and Player is healed to maximum. - The Player can reset the game at any point, but this can happen only once per playthrough.
Enemy	- An abstract class for the Goomba and Koopa class. - Generalises the Goomba and Koopa class into one to make it easier to remove Enemies from the map. - When reset, all subclass instances of this class will be removed from the map.
ResetAction	- The main class that will initiate the process of resetting the game. - The ResetAction class will invoke a method the run() method in ResetManager
ResetManager	- A singleton class that does the soft reset on the instances - Responsible for adding and removing resettable objects in the list. - Has a public static factory method.
Resettable	- Interface that is implemented by classes that can be reset. - Contains the method that registers the class as a resettable instance in ResetManager - Contains the method that implements the details of an object being reset.
Status	- The enum that checks if the Resettable can be reset with the help of Status.RESET
Tree	- The terrain object which has a 50% chance of being converted back into Dirt when a reset occurs.
Coin	- The item in the map that will be cleared off when a reset occurs.

Design Rationale (REQ7)

Firstly, the UML class diagram was updated to remove the Goomba and Koopa class. This is just to simplify the diagram by making it less dense. Next, a ResetManager class and a Resettable interface was added to make it easier to implement this requirement. The Resettable interface will be implemented by the classes that will be reset (i.e Tree, Player, Coin and Enemy), however their actual implementation lies in their playTurn() or tick() methods with the exception of Player where its implementation is in the resetInstance() method. In this updated version, instead of ResetAction handling all of the processing, each of the Resettable classes will have their own processing and are invoked collectively in the run() method of ResetManager(). This implementation opens up the possibility that in the case of a new class that can be reset, it can simply implement the Resettable interface.

This also further emphasises the Open-Closed Principle, where it can be observed in the implementation of the Resettable interface.