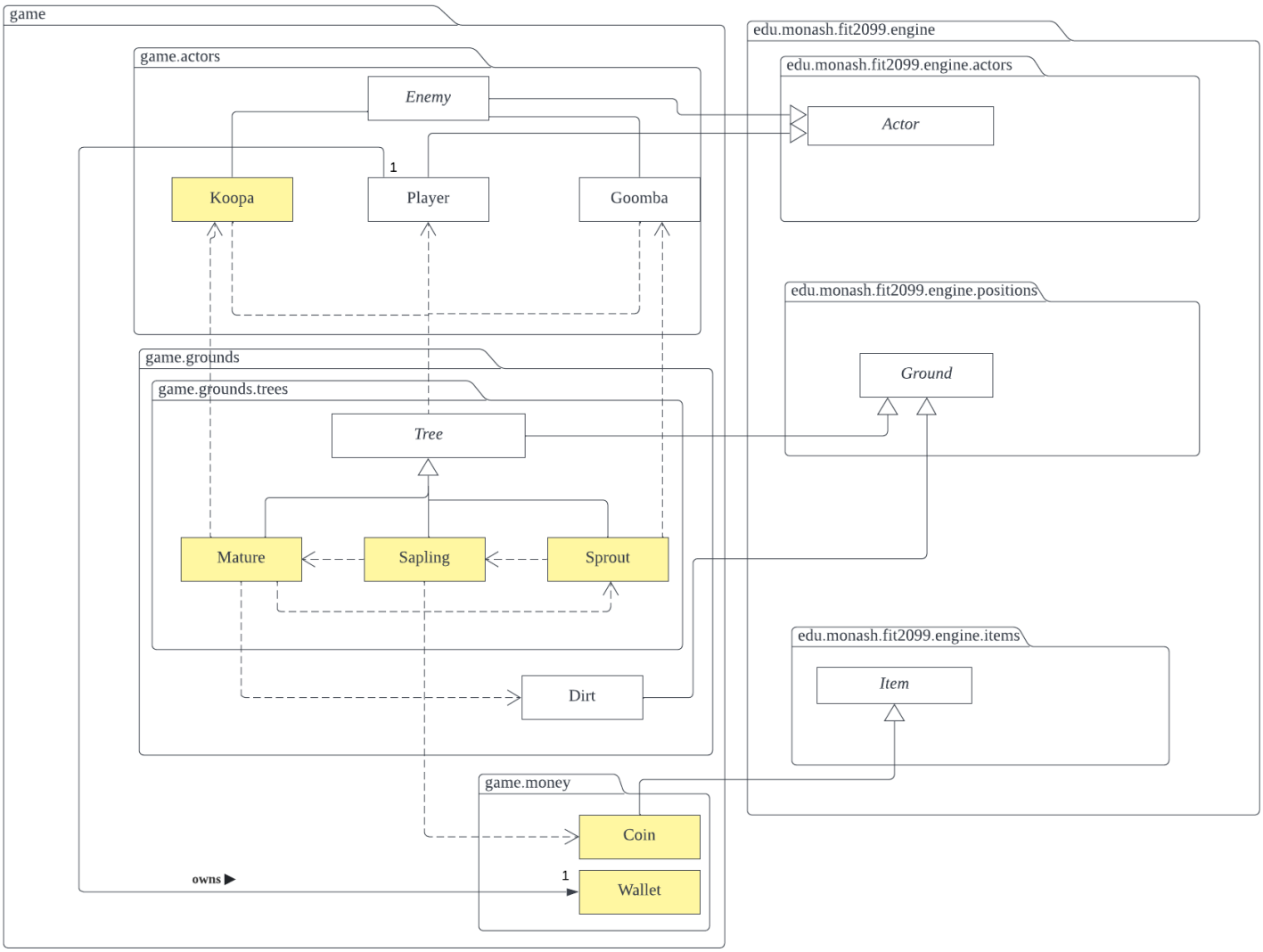
2022

FIT2099 Assignment 1 Diagrams

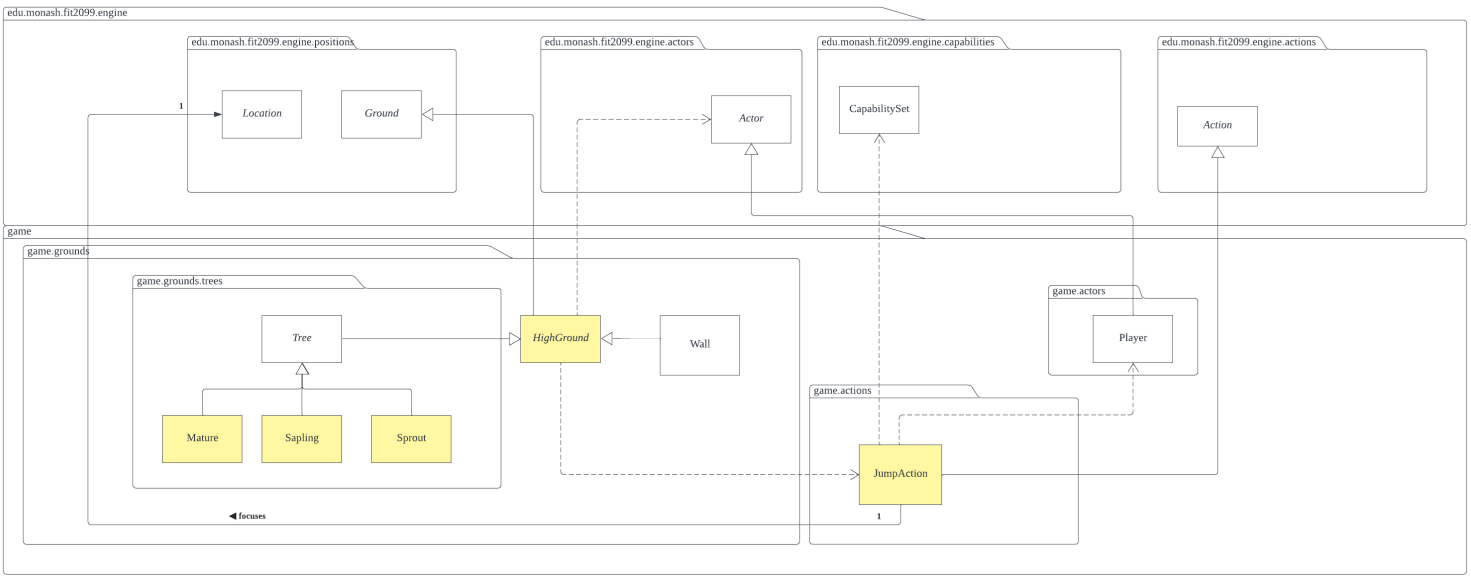
LAB 3 TEAM 2

SHANTANU THILLAI RAJ, EDELYN SEAH, CHAI LI GUANG

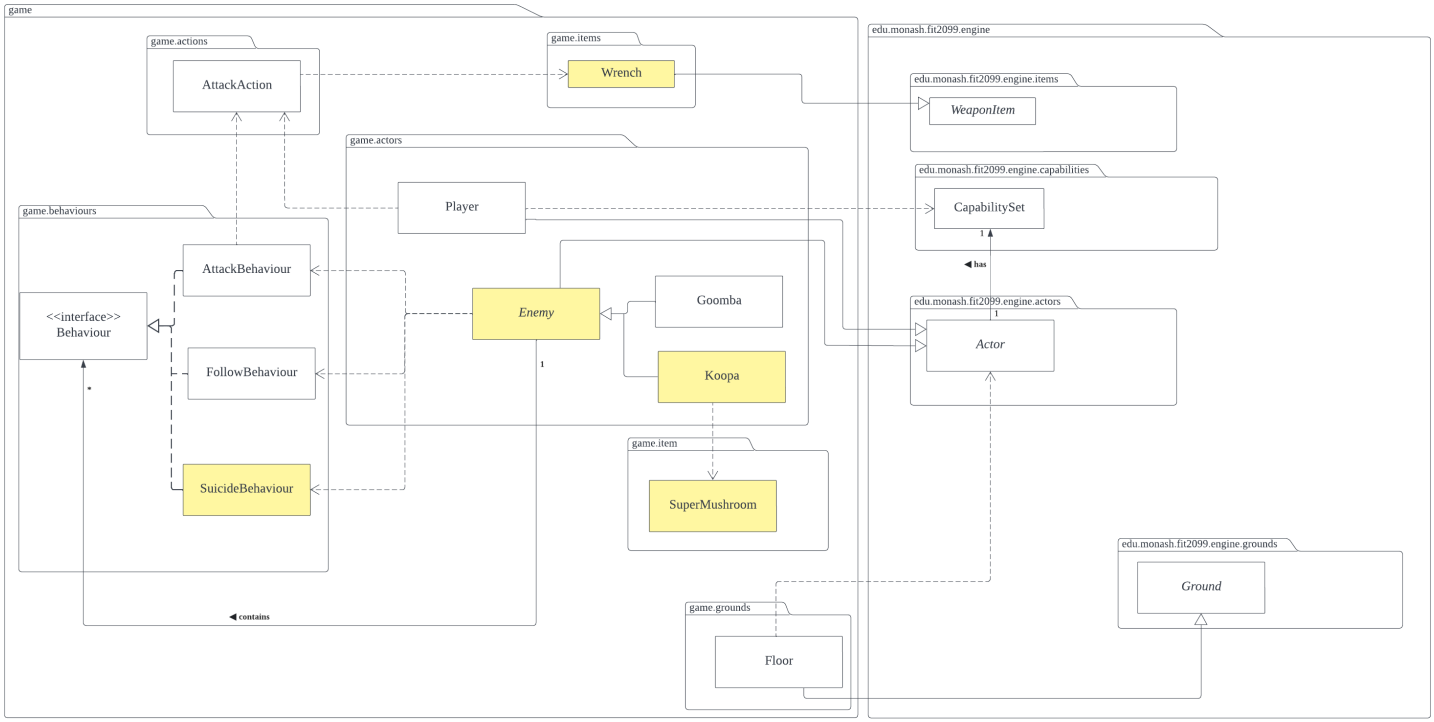
Requirement 1: Let It Grow UML Diagram



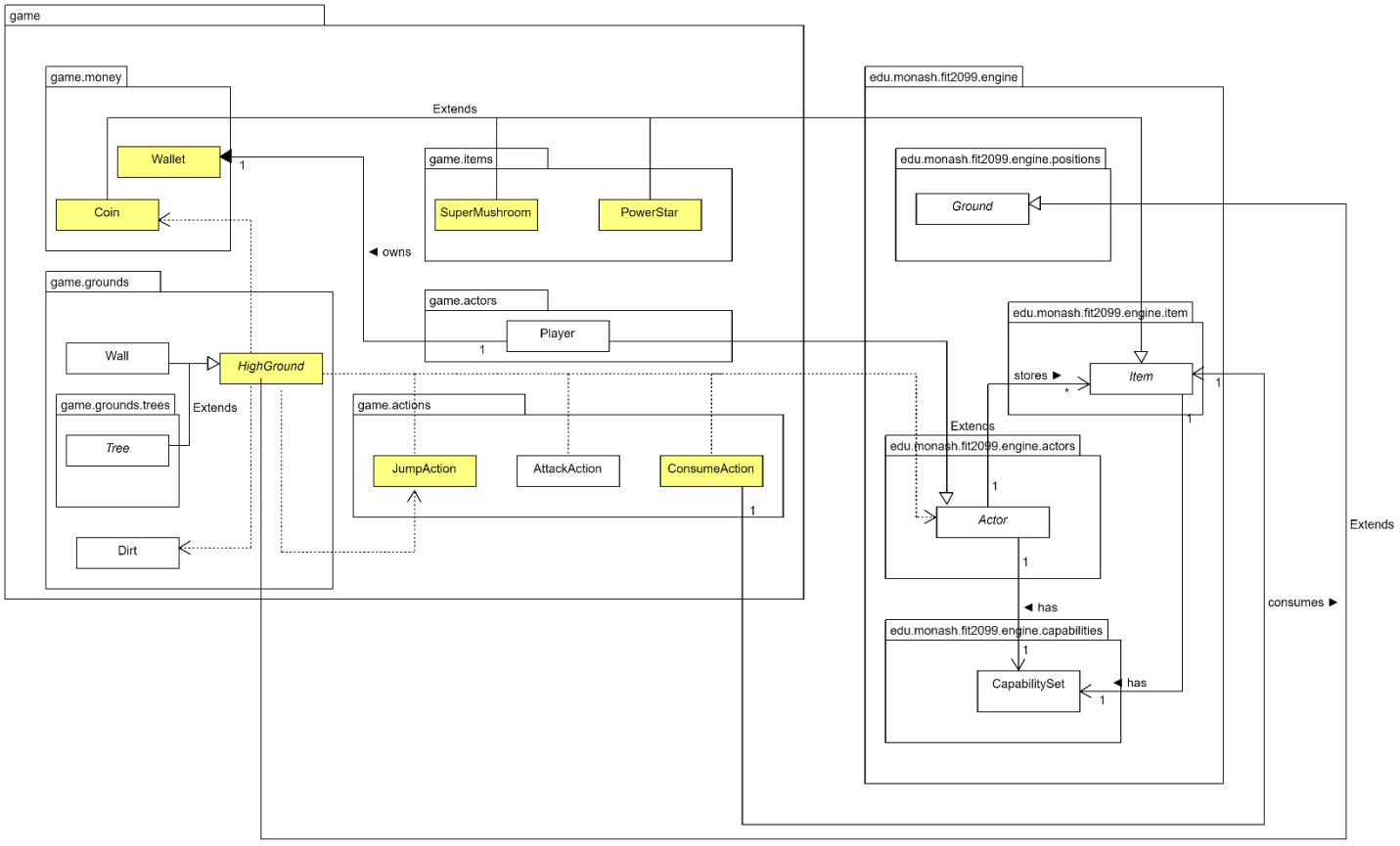
Requirement 2: Jump Up, Super Star! UML Diagram



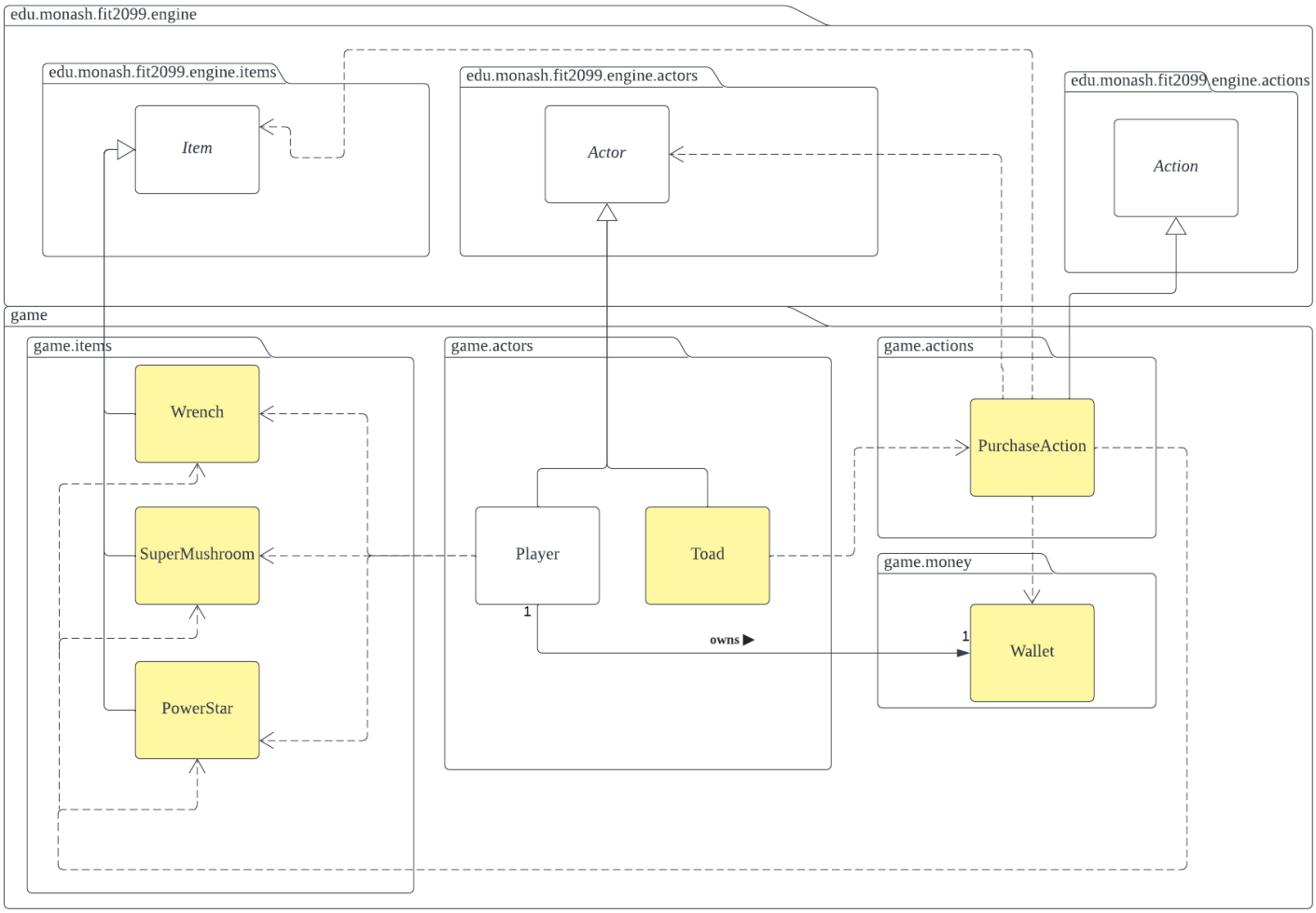
Requirement 3: Enemies UML Diagram



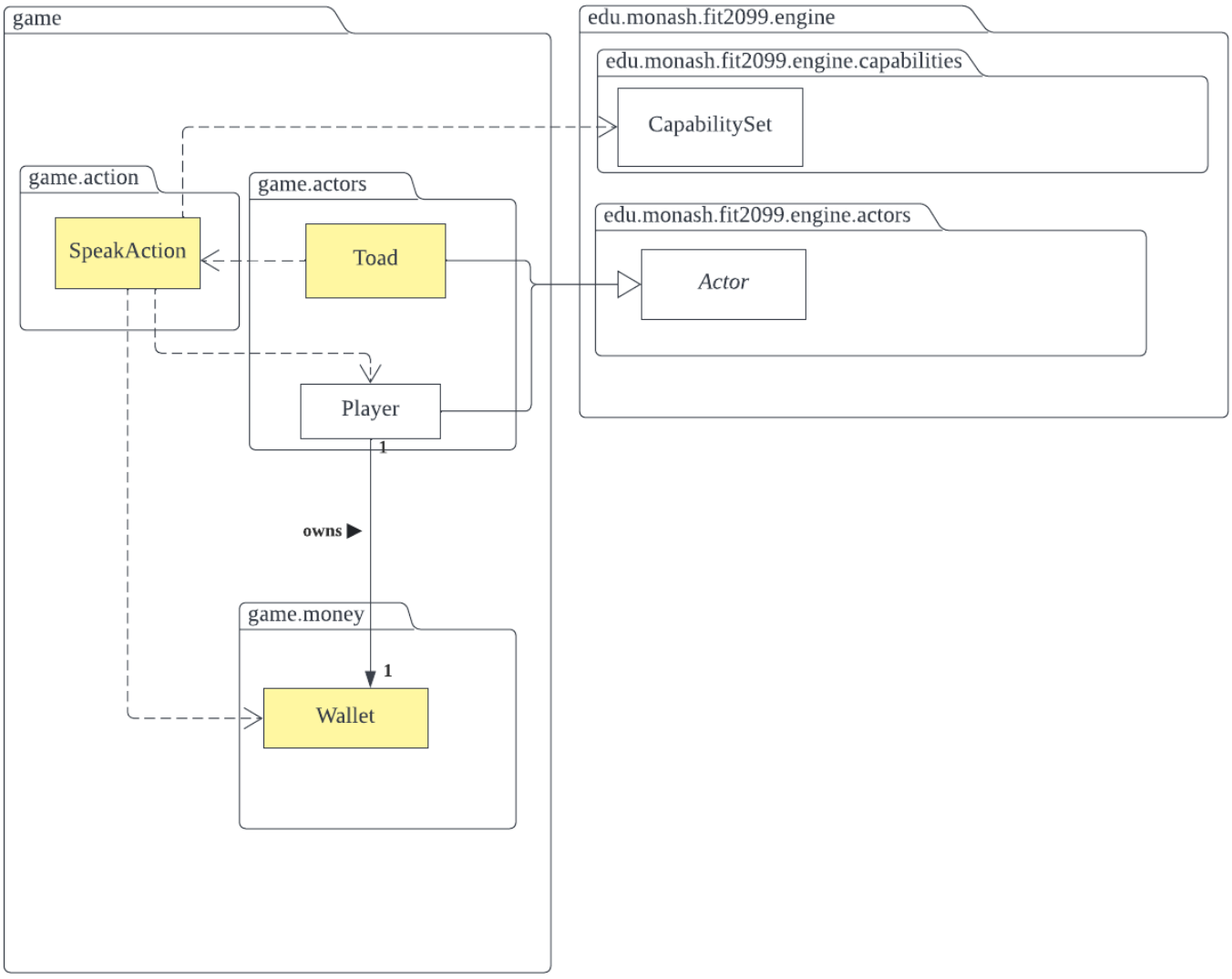
Requirement 4: Magical Items UML Diagram



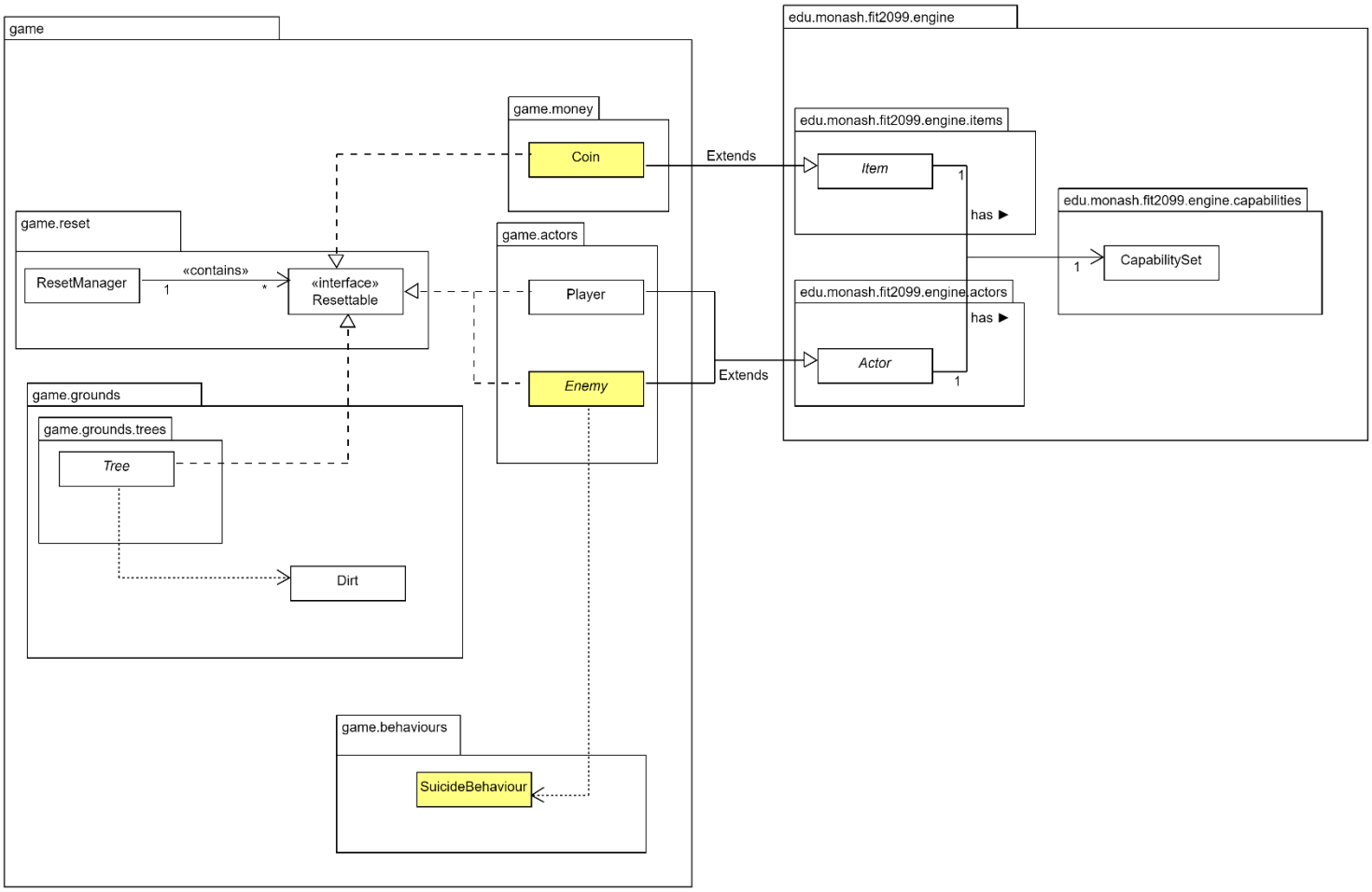
Requirement 5: Trading UML Diagram



Requirement 6: Monologue UML Diagram

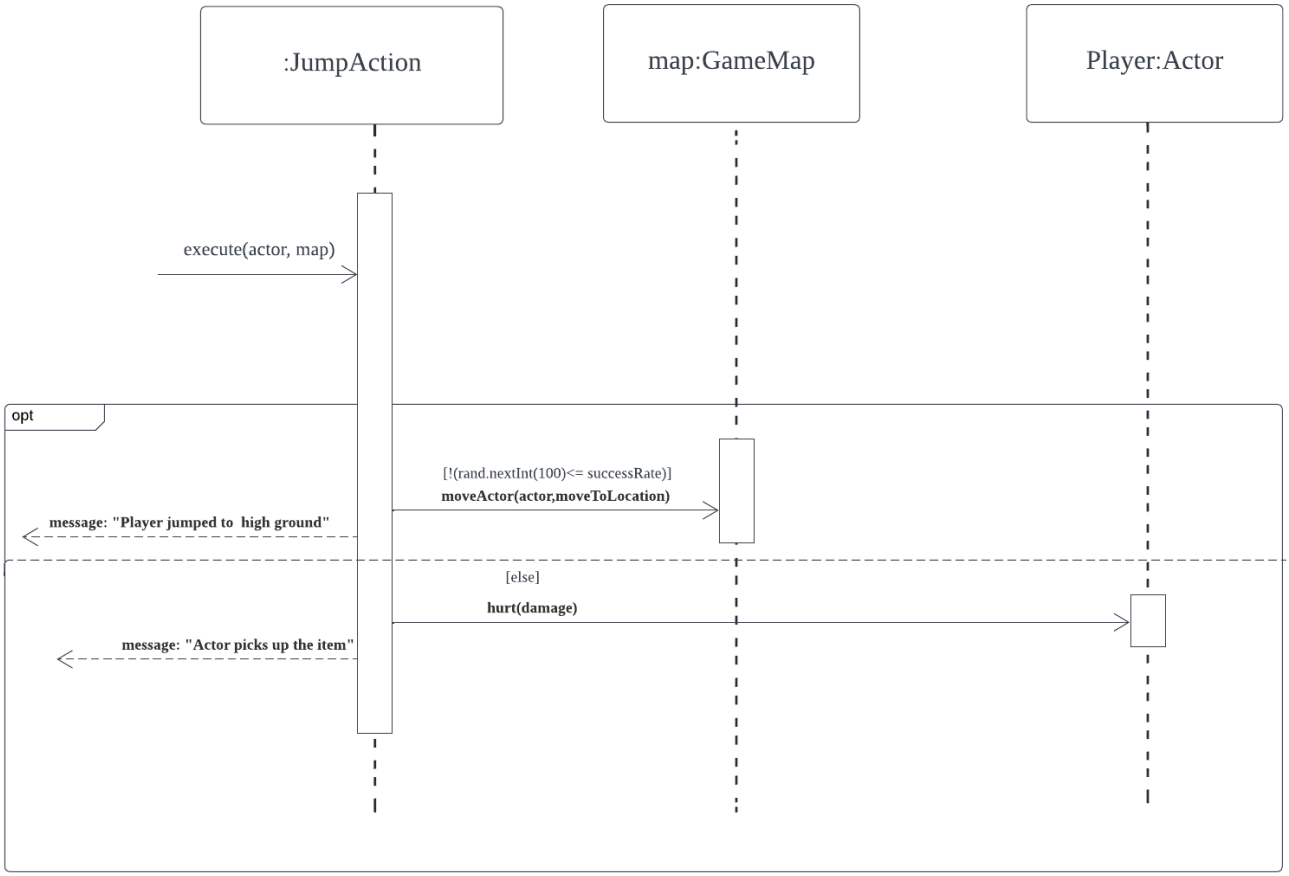


Requirement 7: Reset Game UML Diagram

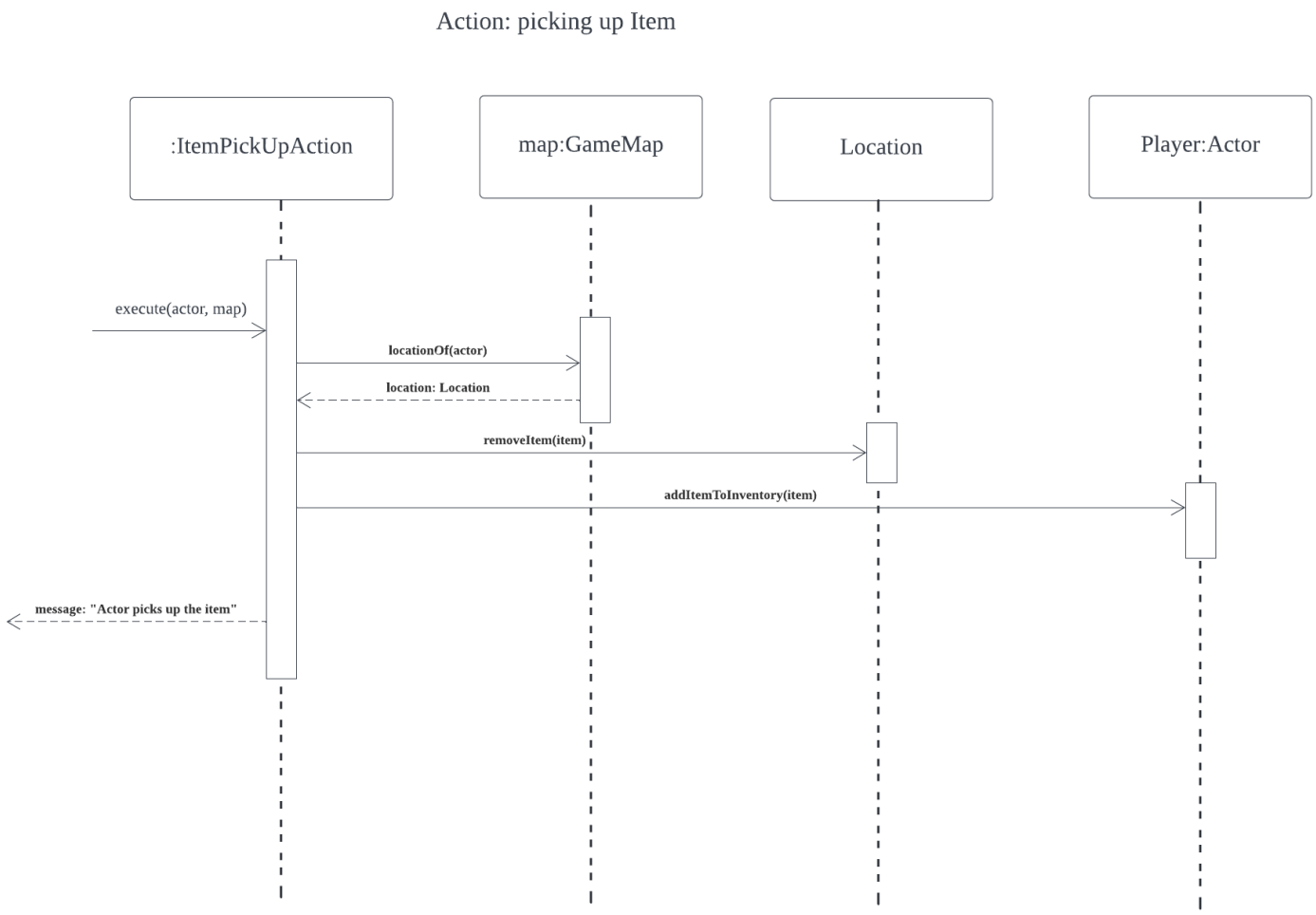


JumpAction Sequence Diagram

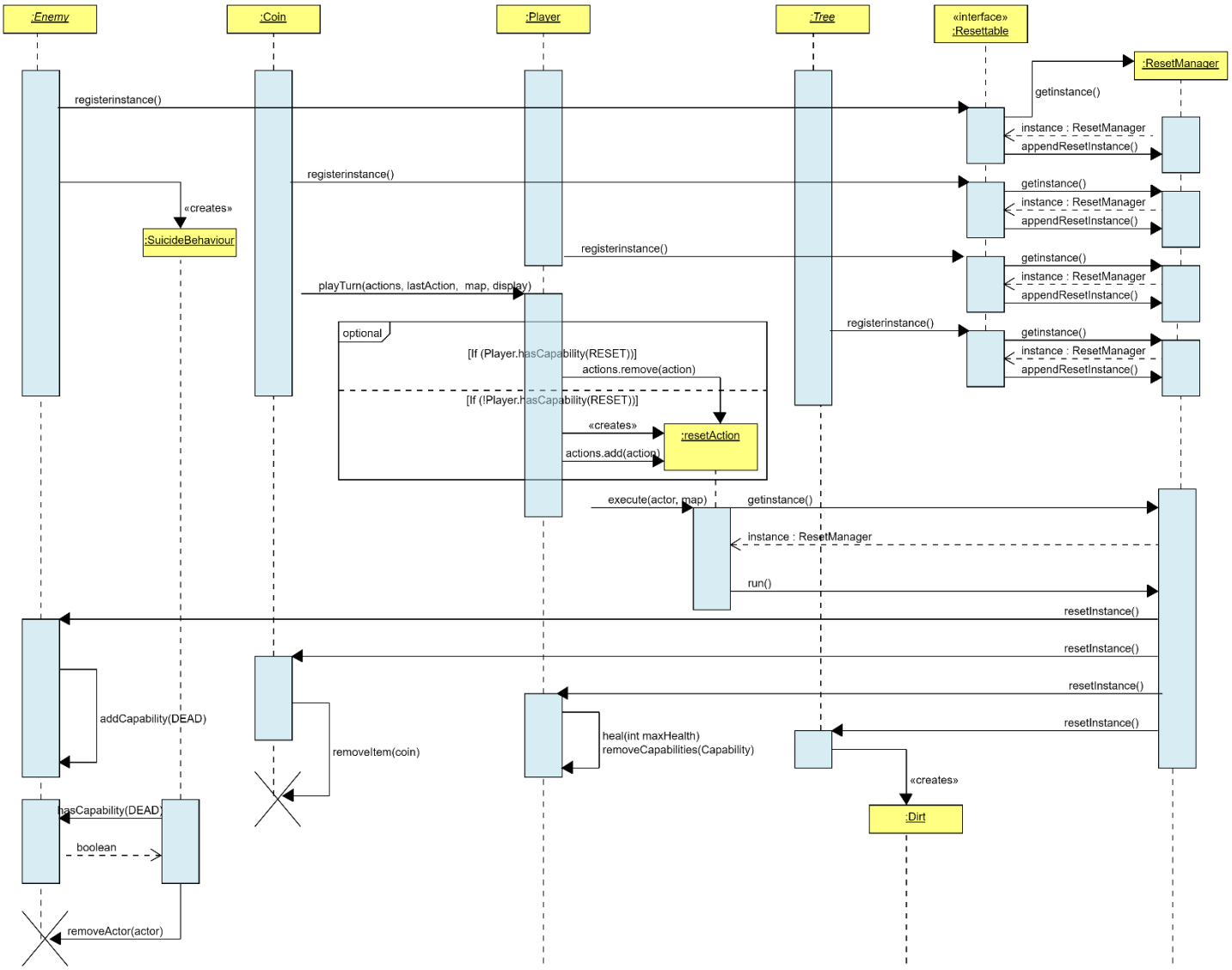
JumpAction sequence diagram



PickUpAction Sequence Diagram



Requirement 7: Reset Game Sequence Diagram



General Rationale

Below is the explanation of the classes we have created to supplement other classes for some of the requirements:

Instead of implementing the wallet inside the player’s inventory, we specifically created a wallet class because we wish to comply with the Single Responsibility Principle(SRP), such that we don't have to worry about handling other responsibilities such as adding or removing items from the inventory. It is also easier to maintain and extend if the class only focuses on one responsibility.

Requirement 1 : Let It Grow!

The tree class was extended from the abstract class Ground. The reason behind it was to achieve reusability as much as possible in our code. Since we have been provided with the abstract class Ground, we can reuse the methods like tick() to build our tree class. In addition, we make the tree an abstract class and have the three stages of the tree (sprout, sapling and mature) extend the tree abstract class. This is because there are common attributes for all the tree stages. For instance, all the trees have a variable called age which is used to keep track of the tree growth stage (every 10 turns will grow to the next stage) and every trees have an unique spawning ability (Sprout spawns goomba, sapling spawns coin, mature spawns, so the abstract Tree class should have an abstract method called spawn().

We know that each tree will grow into the next stage every 10 turns (except for Mature),we have to override the tick() to keep track of the game turns and create instance of the next tree stage on its current location. That is why we have a dependency relationship between the 3 types of trees. (Sprout → Sapling → Mature). Since the Sapling will drop the coin on 10% chance every turn, we will need to implement a Coin class for that. The Sapling will have a dependency relationship with Coin because in our Sapling class, we will have to override/implement the abstract method in the Tree abstract class where we will create an instance of Coin and display it on the current location of the Sapling. Same goes for mature and sprout class, as they need to override the abstract spawn() method to spawn the enemies at a specific chance (10% for sprout to spawn goomba every turn, and 15% for mature to spawn koopa every turn), so mature and sprout have a dependency relationship with their respective spawning enemies.

Besides, when the player picks up the coin, it will increase the balance of the player's wallet. So instead of creating a PickCoinAction class that is actually similar to PickUpItemAction, we just override the addItemToInventory() method in Player class so that it will check if the item it is going to pick up is coin or not, if yes it will add the coin value into the wallet instance declared as the Player instance variable. That is why we have an association relationship between the Player and wallet.

Requirement 2 : Jump Up, Super Star!

One of the main concerns of this requirement is to determine if the actor is standing right beside a high ground (any ground that the actors can not enter directly, i.e walls and trees). However, it is already implemented in the engine class where in the location class it will check if the actor is able to enter the particular ground by executing the canActorEnter() method in ground class. So instead of overriding the canActorEnter() method in every single highground concrete class (sprout, sapling, mature, wall), we created a HighGround abstract class which currently contains 2 subclasses (tree and wall) to reduce the duplicate code which is one of the code smells. The common functionality for all the highground classes is the canActorEnter() method which will return false. unless the actor has the power star buff active. In conclusion, this explains why we have a dependency relationship between the HighGround abstract class and the Actor class and also the dependency relationship between CapabilitySet class and HighGround abstract class.

The JumpAction class extends the Action class in the engine implementing methods execute and menuDescription. JumpAction is only used by the player thus the dependency relationship between the Player and JumpAction and not the other actors like the enemies. Besides, the JumpAction class also has dependency relationships with the high ground abstract class, this is because the HighGround abstract class has the overridden allowableAction() method that will add JumpAction into the action list. In addition, since the actor(player) will move to the high ground location when the jump is successful, JumpAction also have an association relationship with Location because we need to pass the location of the actor and the direction to the high ground into the JumpAction input argument when adding it into the action list in allowableAction() method. In that case, JumpAction is forced to have an instance variable of Location type, making them associative with each other. Lastly, the JumpAction has to check if the player has any super mushroom buff activated so it has a relationship with CapabilitySet.

Requirement 3 : Enemies

As we know, the Player, Goomba and Koopa extends from the abstract Actor class. To make the design more reusability-friendly, another abstract class specifically for Enemies is created for the Goomba and Koopa and other future enemies to extend. This is to accommodate the similarity of specific allowable actions and reset methods.

The Wrench class (self made class) inherits from the abstract WeaponItem class. The damage value given by the Wrench class is then used as a dependency for the AttackAction class. This is because the damage value of the AttackAction (if the wrench is used by the Player) is will be affected based on the damage and hitRate from the Wrench class. The AttackAction and AttackBehaviour classes have a dependency relationship as well since the enemies require AttackAction to attack the Player.

The AttackBehaviour, FollowBehaviour and SuicideBehaviour (self-made) classes implement interface Behaviour to enable modularization since the behaviours can be reused for multiple actors. The Behaviour is an association of the Enemy class because the Behaviour interface has a hashmap- which has the purpose of storing the behaviours of each enemy. SuicideBehaviours checks the capabilities (for an enum called REMOVED) to automatically remove the enemy from the map on the next turn– because for requirement 3, there is a 10% that the Goomba will suicide each turn. The other enemies are not affected because they do not have the REMOVED enum.

There is a dependency between the Player and the CapabilitySet because the status of the player depends on the CapabilitySet to tell if it has any power ups - eg: if the SuperMushroom is consumed, max HP is increased by 50.

For enemy actors, there will be an enum (ENEMY) which will be already known by the CapabilitySet Class due to the association relationship between the Actor Class. The Floor Class will implement a canActorEnter method which will require the enum ENEMY, since by transitivity, the Floor can detect it from the Actor Class from abstraction.

Lastly, the SuperMushroom Class is a dependency with Koopa Class because it creates an instance of a SuperMushroom.

Requirement 4 : Magical Items

The rationale behind the design of requirement 4.

Necessary info for understanding rationale

- In the engine provided, all items, players and grounds have a capability set linked to them.

The Design

- In the Status enum class, we add 2 constants (for this explanation I'll be calling these constant GIANT and INVINCIBLE).
- Both the `SuperMushroom` and `PowerStar` classes extend the `Item` class present in the engine.
- In the constructor for the SuperMushroom and PowerStar we have use the `addCapability()` method to add GIANT (mushroom) and INVINCILE (star) to the capability list of the items.
- For the power star 10 turn to dissapear if on the ground :
    - Making use of the `tick` method in the class, we just check if the item is still on the ground and not in the players inventory before the tick method runs 10 times (using an int counter attribute which starts at 0)
    - If the tick method runs 10 times it means (counter reaches 10), the power star will be removed from the map using :
      `currentLocation.removeItem(this)`
      in the `tick` method of the item.
    - If the player picks up the star : the counter still goes (does not reset back to 0) on until the player consumes the item. If it hits 10 before the player consumes the item, it is removed from the players inventory using :
      `actor.removeItemFromInventory(this)`
      in the `tick` method of the item.
- To consume an item from the inventory, the `consumeAction` class which extends abstract `Action` is used
    - This action provides the menu option to consume the item.
    - If executed (`execute(Actor actor, GameMap map)` method in the `consumeAction` class) :
        - Mushroom item : Uses `addCapability()` to add the `GIANT` capability to the actors capability list. Increases the actors max HP by 50 using the `increaseMaxHp(int)` method provided in the `Actor` class. Removes the item from the actors inventory using method `removeItemFromInventory(item)`
        - Power Star item : Uses `addCapability()` to add the `INVINCIBLE` capability to the actors capability list. Heals the actor by 200 HP using the `heal(int)` method provided in the `Actor` abstract class.
        - Resets the counter in the Power Star class using a `resetCounter()` method which sets the counter value back to 0. When the counter hits 10 this time, the capability is removed from the actor using `removeCapability` method.
- Mushroom GIANT Capability for jumps/attacks :
    - The JumpAction (created in the `HighGround` abstract class method `allowableActions()`) would take probability of jump success as an input in the constructor which is normally based on each ground type (wall is 80%, sprout is 90% etc.) When the player has capability of `GIANT`, checked

using the `hasCapability(GIANT)` on the actor the probability is set to 100 to ensure the player always suceeds in the jump.
    - In the `attackAction`, in the exexute method, when an enemy (Goomba or Koopa) attacks the player, it checks if the player has `GIANT` capability using `hasCapability(GIANT)` method on the actor, if true, player takes 0 damage and the capability is removed using `removeCapability(GIANT)`
- Power Star INVINCIBLE for attacks :
    - The JumpAction (created in the `HighGround` abstract class method `allowableActions()`) would not be created if actor `hasCapability(INVINCIBLE)` and the `canActorEnter` method is used where it returns : `actor.hasCapability(ItemCapabilities.INVINCIBLE)` which allows the actor to move onto the high ground without jumping. Using the `tick` method in `HighGround`, we check if the location of the actor is the same as the location of the `HighGround` and if the actor has the capability `INVINCIBLE`. If both true, the ground there is set to be dirt (destroyed) and a new Coin item with value 5 is mode on the same location.
    - For attacks, if attacking, in the `attackAction` execute method we check using `hasCapability()`. If the actor has this capability, they hit the enemy for their full health (insta kill).
    - For attacks, if getting attacked, in the `attackAction` execute method we check using `hasCapability()`. If the target has this capability, they will take 0 damage from the enemy. Code : `target.hurt(0)`

Principles Applied

Liskov Substituion Principle

- The Power Star and Super Mushroom extend the Item abstract class and do not add any special methods as we just set capabilites and handle each scenario based on the capabilities therefore allowing "replacement" without disrupting the system.
- Avoid downcasting or using getClass()/instanceof to check which item is in the players inventory as we can just get the items capabilites and we would know exactly what item it is.
- Avoid redundancy by just transferring the capability of the item into the actor on consumption.

Open-Closed Principle

- By using capabilites to do the checks, we are able to extend the items without modifying the item abstract clss itself.
    - There would be no extra methods implemented specially for PowerStar or SuperMushroom that does not exist in the item abstract class nor will there be any special methods that we would have to call in each subclass. Therefore we once again avoided using instanceof/getClass() and downcasting.

DRY Principle

- Instead of creating methods in each item for when we consume them to do, we just use the methods provided in the engine like heal, increaseMaxHp etc. on consumption.

Requirement 5 : Trading

In order for the trading interaction between the player and toad to happen, the toad has to be located in one of the player's exits (8 directions). Fortunately, this has already been implemented in the engine code (allowableAction() method). So we would only need to override that method in the toad class such that it would add PurchaseAction into the ActionList. Thus, there is a dependency relationship between Toad and PurchaseAction.

Looking into the PurchaseAction class, it has dependency relationships with all the items (Wrench, SuperMushroom and Power Star) because we would need to create instances of the items in the execute() to store it inside the player's inventory when the player bought the particular item. Before that, the PurchaseAction will check if the player has enough money to buy the items, so it will get the player's wallet and check its current amount, which explains the relationships between PurchaseAction and wallet. Furthermore, the wallet class has an association relationship with the player since we would only need to instantiate the wallet one time as an instance variable so we can use it in the other methods to update the same wallet instance.

REQ6

We extends the Player class from the abstract Actor class because we wish to follow the Liskov substitution principle (LSP), as the player class will inherits all the methods from the actor class, making the subclass(player) is able to replace the superclass(actor) without breaking the program. This will only work for player class as it will utilize all the methods in the actor class unlike other subclasses such as toad and koopa. For example, player class uses heal() and increaseMaxHp(), where other subclasses of actor class don't.

The Actor implements Capable interface because the Actor class contains a method (hasCapability()) which can detect whether the power star is consumed. There is an association between the abstract Actor class and the Item class to detect if the player is currently holding the wrench because the actor class contains an instance variable of Item type (an array list of items representing the actor's inventory).

Requirement 7 : Reset Game

The rationale behind the design of requirement 7.

Necessary info for understanding rationale

- All enemies extend from the Enemy abstract class.
- All tree types (Sprout, Sapling, Mature) extend from the Tree abstract class.
- The tick method occurs every turn regardless of situation.

The Design

- In the Status enum class, we add 2 constants (for this explanation I'll be calling these constant REMOVED and RESET).
- The classes `Enemy`, `Tree`, `Coin` and `Player` implement the `Resettable` interface.
- In each of those classes that implemented resettable we add the following to the constructor : `registerinstance()`
    - The `registerinstance()` method will append this resettable object into the array list in the resetManager class which is responsible for the resets when the object is created.
- In each class implementing Resettable, we have to implement the method `resetInstance()`. The `resetInstance()` method should do the following :
    - For the `Coin` and `Enemy` class when a reset occurs, we just use the `addCapability()` method to add the REMOVED to their capabilities
    - For the `Tree` class when a reset occurs, we use random (50% probability) to check if the tree should be removed or not, if to be removed, use `addCapability()` method to add the REMOVED to the trees capability list.
    - For the `Player` class when a reset occurs, use `removeCapability()` in a loop to remove every capability from the capability list of the player and use `heal()` alonsgside `getMaxHp()` to heal the player back to full.
- After `resetInstance()`, none of the trees/enemies/coins have been removed yet. This is where we add to the tick methods of tree and coin to remove them and create a new behaviour `SuicideBehaviour` to kill all the enemies.
    - In the tick methods for the `Coin` and `Tree` class, we should have a check for if "this" (since tick occurs in each instance of an object) has the capability of REMOVED using `hasCapability`. If it does have the capability REMOVED, then we just use :
      `currentLocation.removeItem(this)`
      to get rid of the coins and use :
      `location.setGround(new Dirt())`
      to change the ground from a tree to a dirt object.
    - The `Enemy` class does not have a tick method as such instead we create a behaviour called `SuicideBehaviour` that does the same check using `hasCapability`, if it returns true, remove the enemy from the map using :
      `map.removeActor(actor)`
- `ResetAction` class implements the abstract class Action from the engine. All it does is gives the menu description of the choice to reset and in the execute method :
    - Creates a new instance of reset Manager which has all the resettable items.

    - Calls the `run` method in ResetManager which runs all the resetInstance() methods in each resettable object.
    - Adds a capability into the Player, called RESET using addCapability()
- The Player does a check on every `PlayTurn()` method call for the RESET capability using `hasCapability()`. If it has the RESET capability it means that we should not add the `resetAction` into the players action list.

Principles Applied

Single Reponsibility Principle

The `SuicideBehaviour` class and `ResetManager` class only have 1 responsibility each :-

- `SuicideBehaviour` handles removing enemies off the map.
- `ResetManager` handles storing all the resettable items and calling all the `resetInstance()` method in each of those items when a reset occurs.

DRY Principle

- Making the `Tree` and `Enemy` classes implement the inteface instead of each tree (sprout, sapling, mature) and each enemy (Goomba and Koopa), allows us to avoid repeating the implementation of `resetInstance()` in everyone of those subclasses.

Liskov Substituion Principle

- By implementing the interface into `Enemy` and `Tree` instead of in each of its subclasses. It allows for all the enemies and trees to have the resettable method we would need, letting each one (of the same subclass) to be able to replace the other
- Implementing an interface for the resets in every class instead of an abstract class or a normal method allows the ResetManager to avoid downcasting/using the taboo `instanceOf` and `getClass()` methods as we know that any object whih is of `Resettable` does have a resetInstance() and we can just call that method instead of having to down cast and calling a normal method in the class.