**Work Breakdown Agreement for FIT2099 Assignment 1/2/3**

Team 3 Lab 11

1. Seow Zheng Hao (32558414)
2. Muhammad Abdullah Akif (31275036)
3. Danesh Mariapan (31965601)

We hereby agree to work on FIT2099 Assignment 1 according to the following breakdown:

- Class Diagrams
    - Completion: Sunday, 10th April 2022
    - Initial commit: Sunday, 10th April 2022
    - Authors:
        - REQ 1 & REQ 2 will be done by Seow Zheng Hao
        - REQ 3 & REQ 4 will be done by Muhammad Abdullah Akif
        - REQ 5 & REQ 6 will be done by Danesh Mariapan
        - REQ 7 will be done together
    - Reviewer:
        - REQ 1 & REQ 2 will be reviewed by Muhammad Abdullah Akif
        - REQ 3 & REQ 4 will be reviewed by Danesh Mariapan
        - REQ 5 & REQ 6 will be reviewed by Seow Zheng Hao
        - REQ 7 will be reviewed as a group, as all members participated in making this diagram.
- Interaction Diagrams
    - Completion: Sunday, 10th April 2022
    - Initial commit: Sunday, 10th April 2022
    - Authors: Seow Zheng Hao, Muhammad Abdullah Akif, Danesh Mariapan
    - Reviewer: Will be reviewed as a group, as all members participated in creating the diagram.

**Work Breakdown Agreement for FIT2099 Assignment 1/2/3**

- Design Rationale
    - Completion: Sunday, 10th April 2022
    - Initial commit: Sunday, 10th April 2022
    - Authors:
        - REQ 1 & REQ 2 will be done by Seow Zheng Hao
        - REQ 3 & REQ 4 will be done by Muhammad Abdullah Akif
        - REQ 5 & REQ 6 will be done by Danesh Mariapan
        - REQ 7 will be done together
    - Reviewer:
        - REQ 1 & REQ 2 will be reviewed by Muhammad Abdullah Akif
        - REQ 3 & REQ 4 will be reviewed by Danesh Mariapan
        - REQ 5 & REQ 6 will be reviewed by Seow Zheng Hao
        - REQ 7 will be reviewed as a group, as all members participated in making this diagram.

- Implementation
    - Completion: Monday, 2nd May 2022
    - Authors:
        - REQ 1 & REQ 2 will be done by Seow Zheng Hao
        - REQ 3 & REQ 4 will be done by Muhammad Abdullah Akif
        - REQ 5 & REQ 6 will be done by Danesh Mariapan

**Work Breakdown Agreement for FIT2099 Assignment 1/2/3**

- Add-ons (Assignment 3)
    - Completion: Monday, 22nd May 2022
    - Authors:
        - REQ 2 will be done all together **[ALL MEMBERS]**

          Each person does tasks related to their previous tasks

          [i.e anything related to tree class was done by Zheng Hao in previous tasks, so he will do that part]
        - REQ 1 & 3 will be done by Muhammad Abdullah Akif
        - REQ 4 will be done by Seow Zheng Hao
        - REQ 5 will be done by Danesh Mariapan

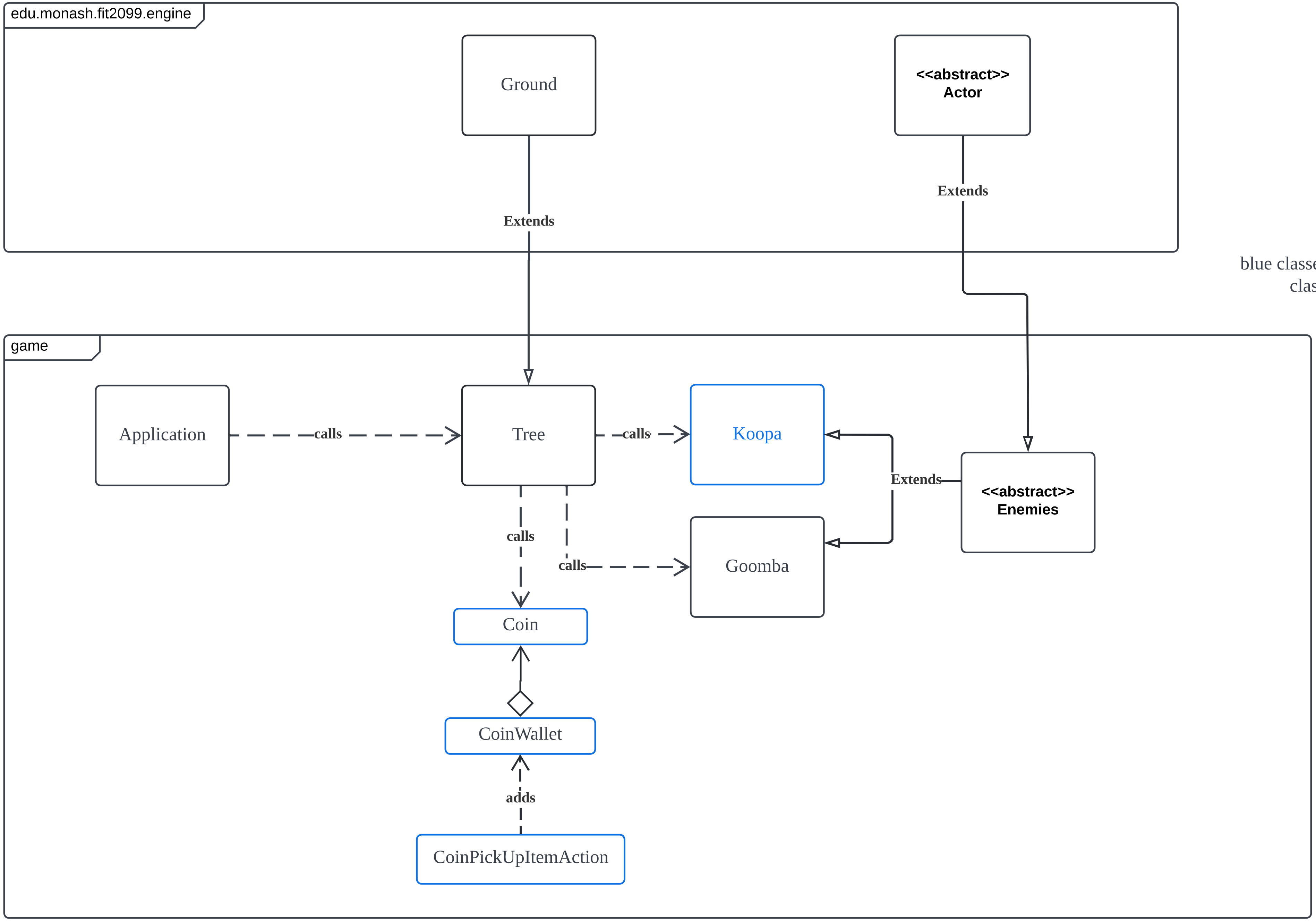
I accept this WBA - **Muhammad Abdullah Akif**

I accept this WBA - **Seow Zheng Hao**

I accept this WBA - **Danesh Mariapan**

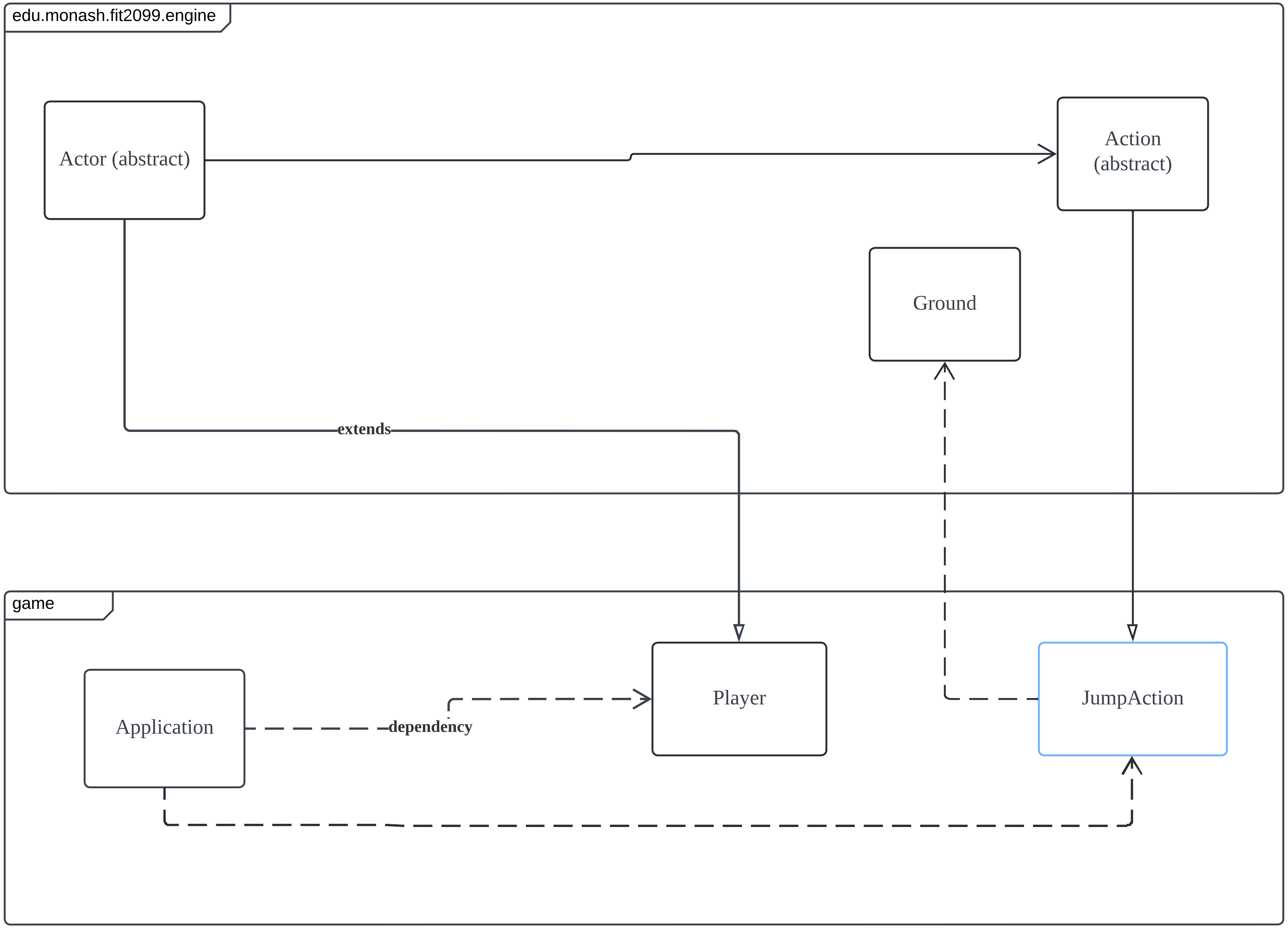
# REQ 1

Seow Zheng Hao



blue classes are new classes

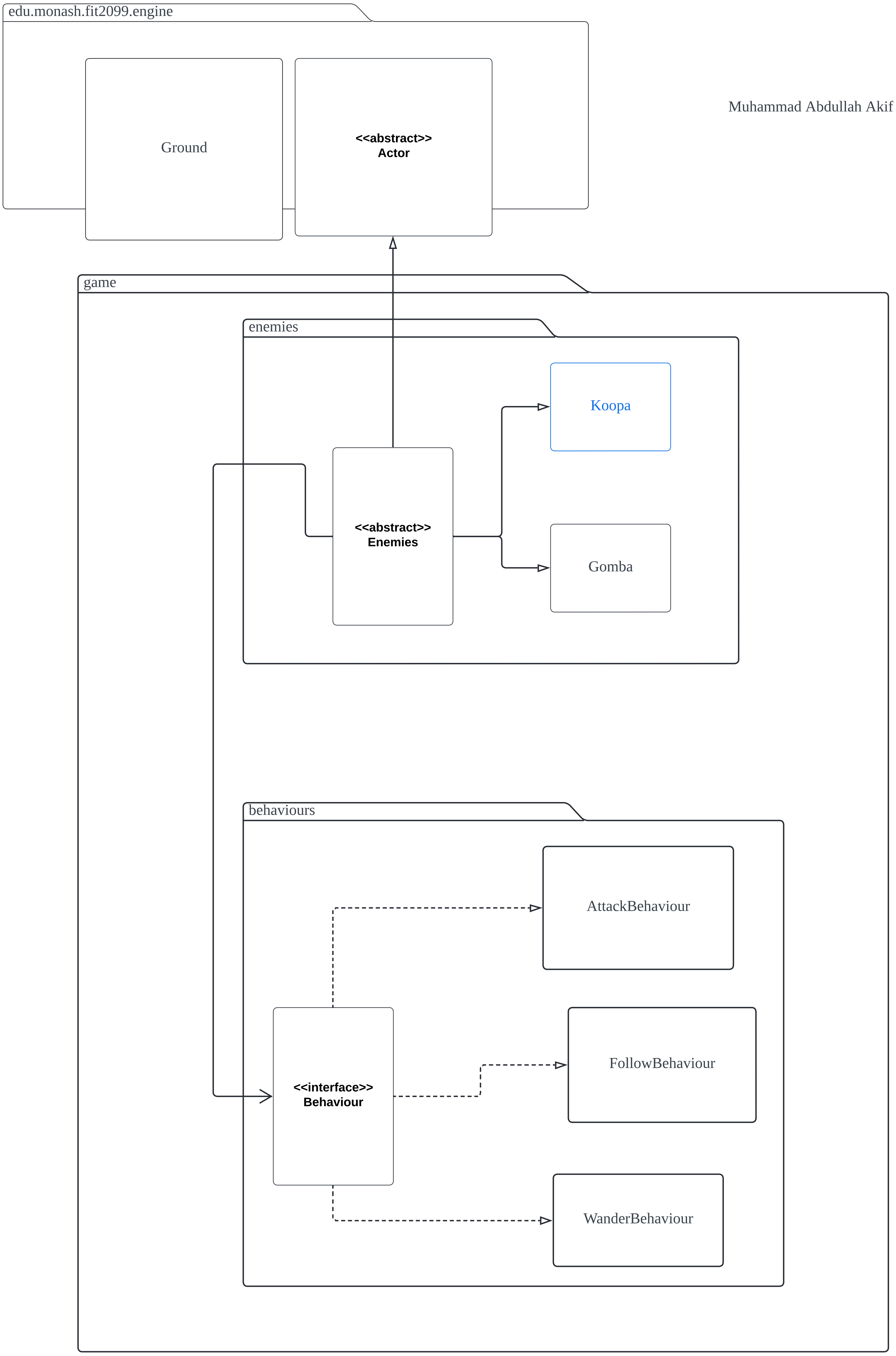
# REQ 2


edu.monash.fit2099.engine
Actor (abstract)
Action (abstract)
Ground
extends
game
Application
dependency
Player
JumpAction

# REQ 3

Updated Version

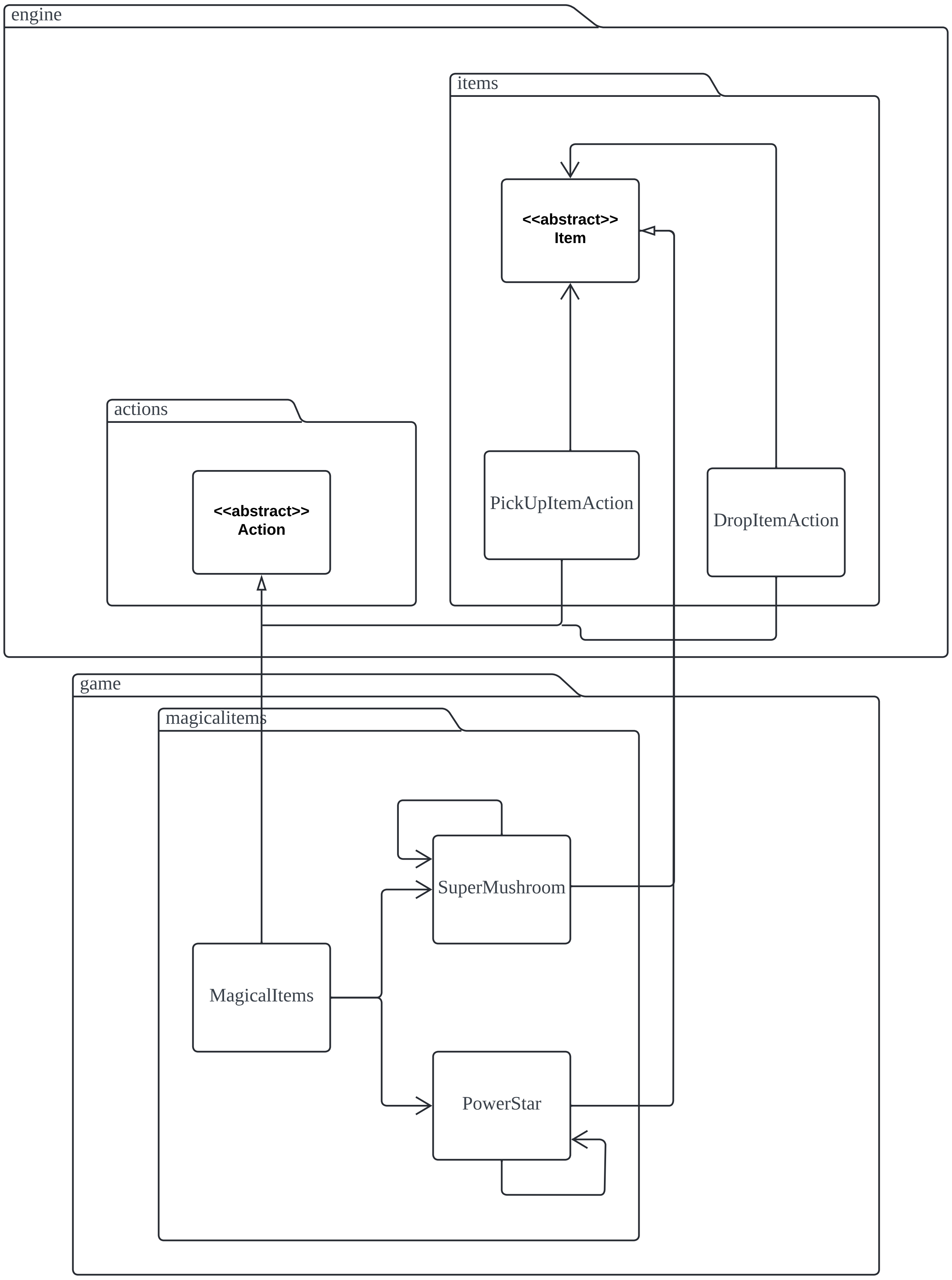
Muhammad Abdullah Akif



# REQ 4

Updated Version

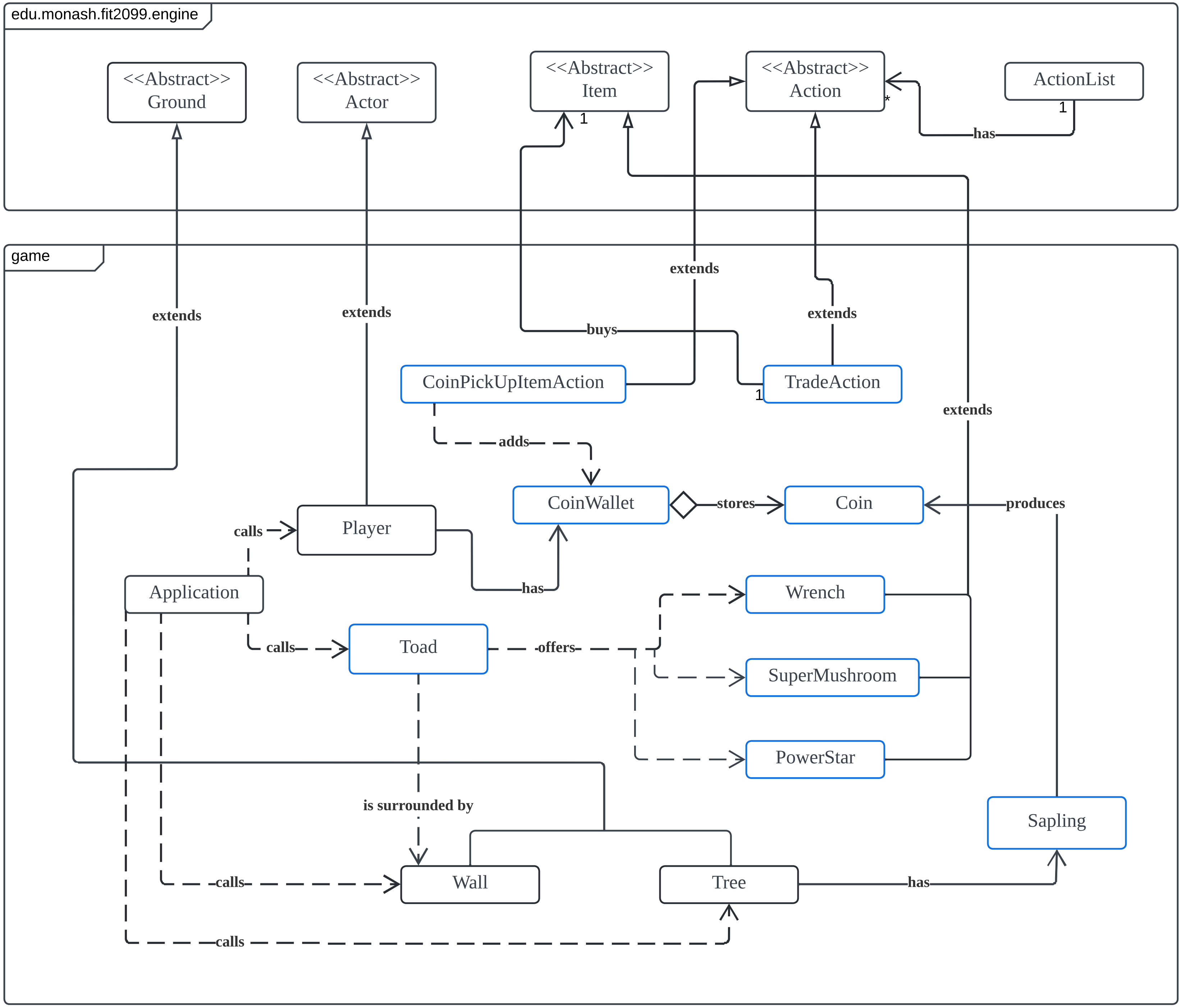
Muhammad Abdullah Akif



Danesh Mariapan

# REQ5: Trading

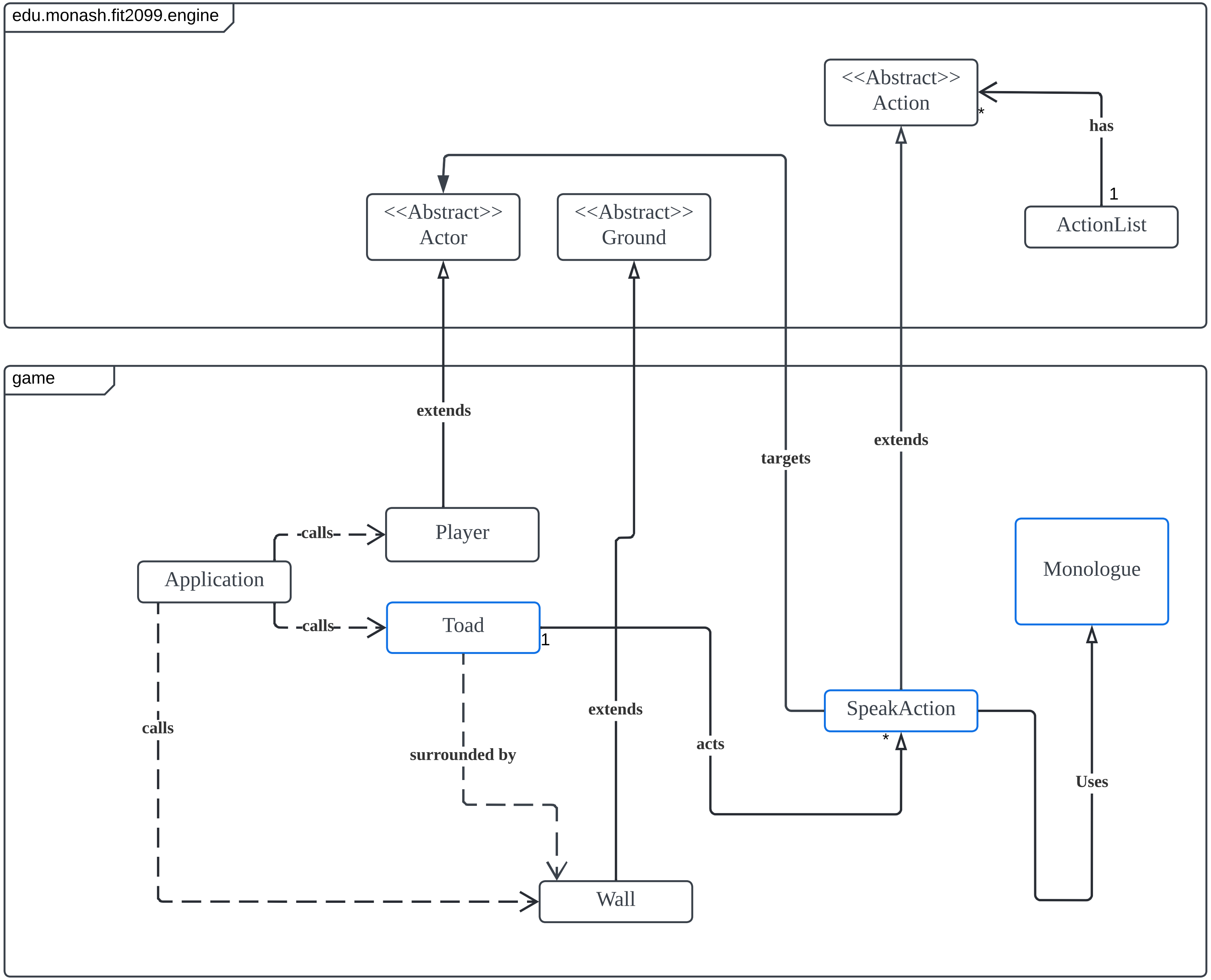
Updated for Assignment 2



Danesh Mariapan

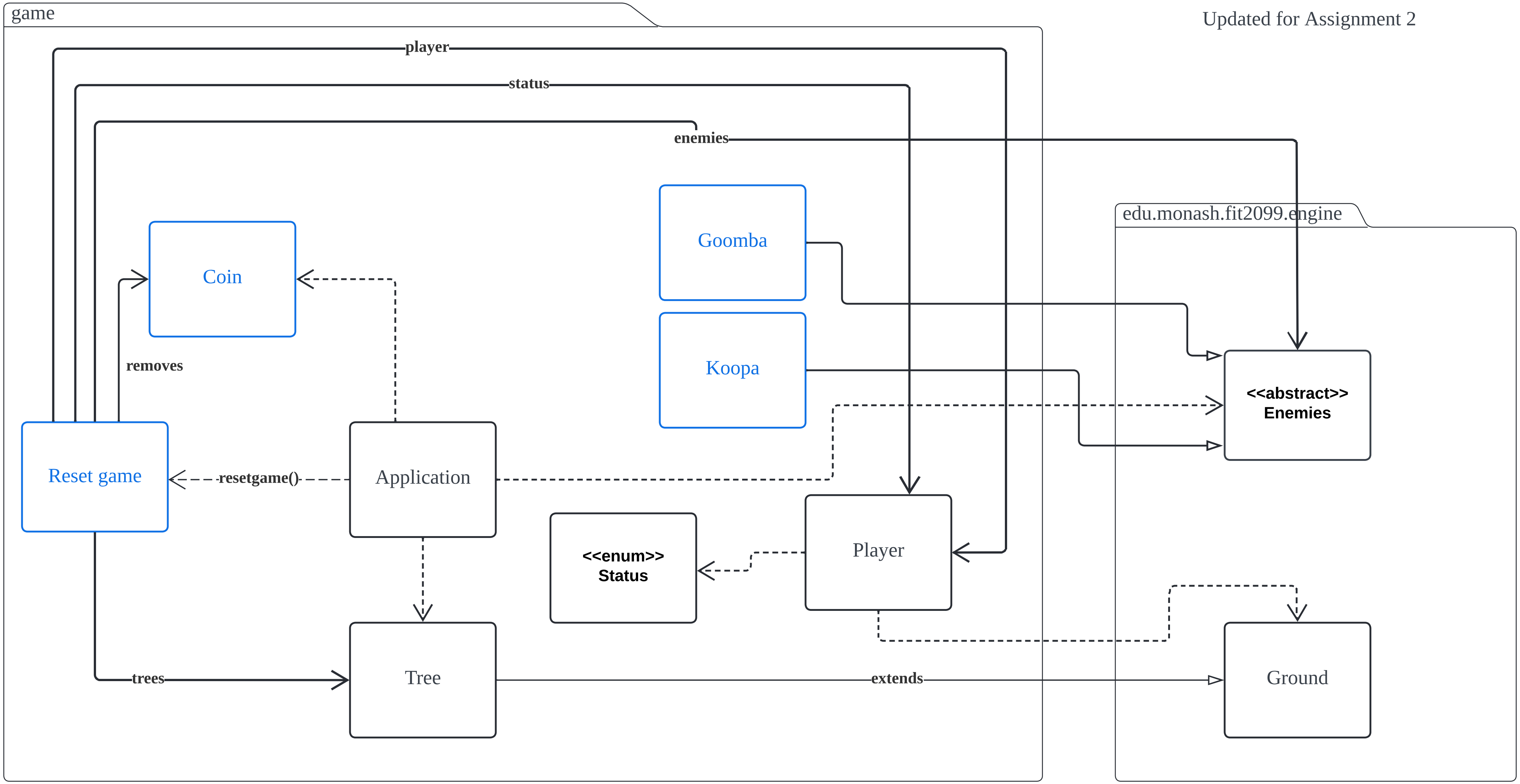
# REQ6: Monologue

Updated for Assignment 2



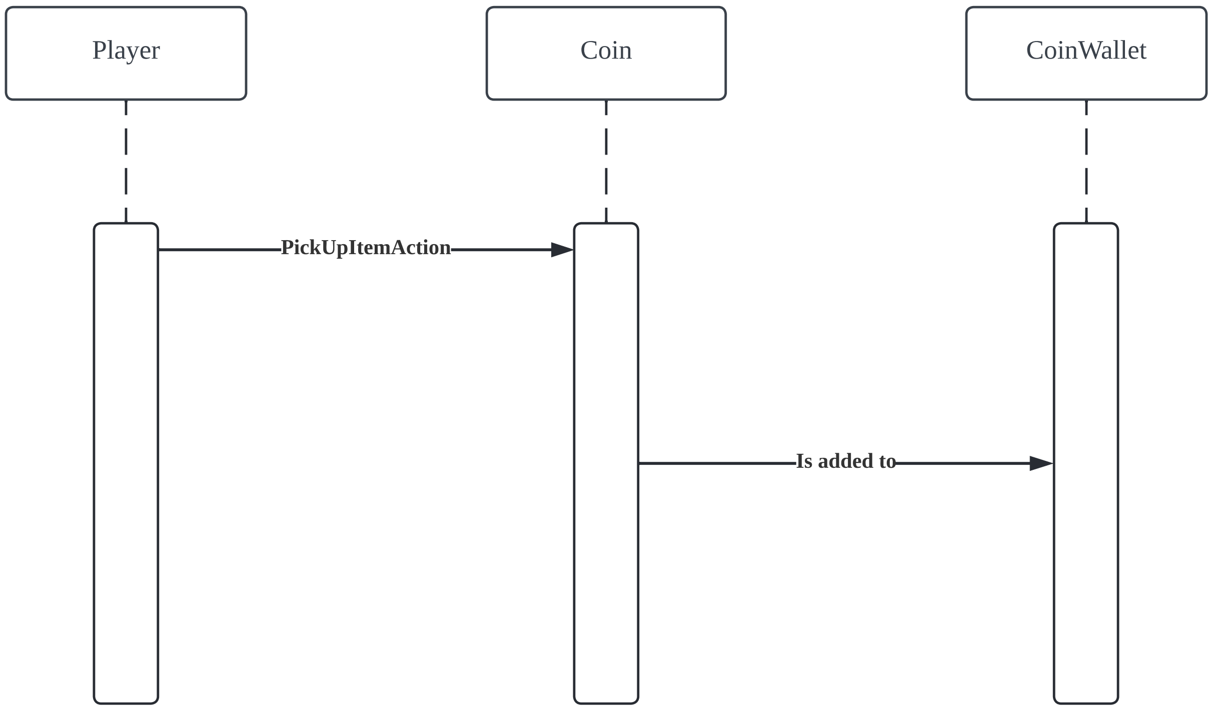
# REQ 7

Updated for Assignment 2

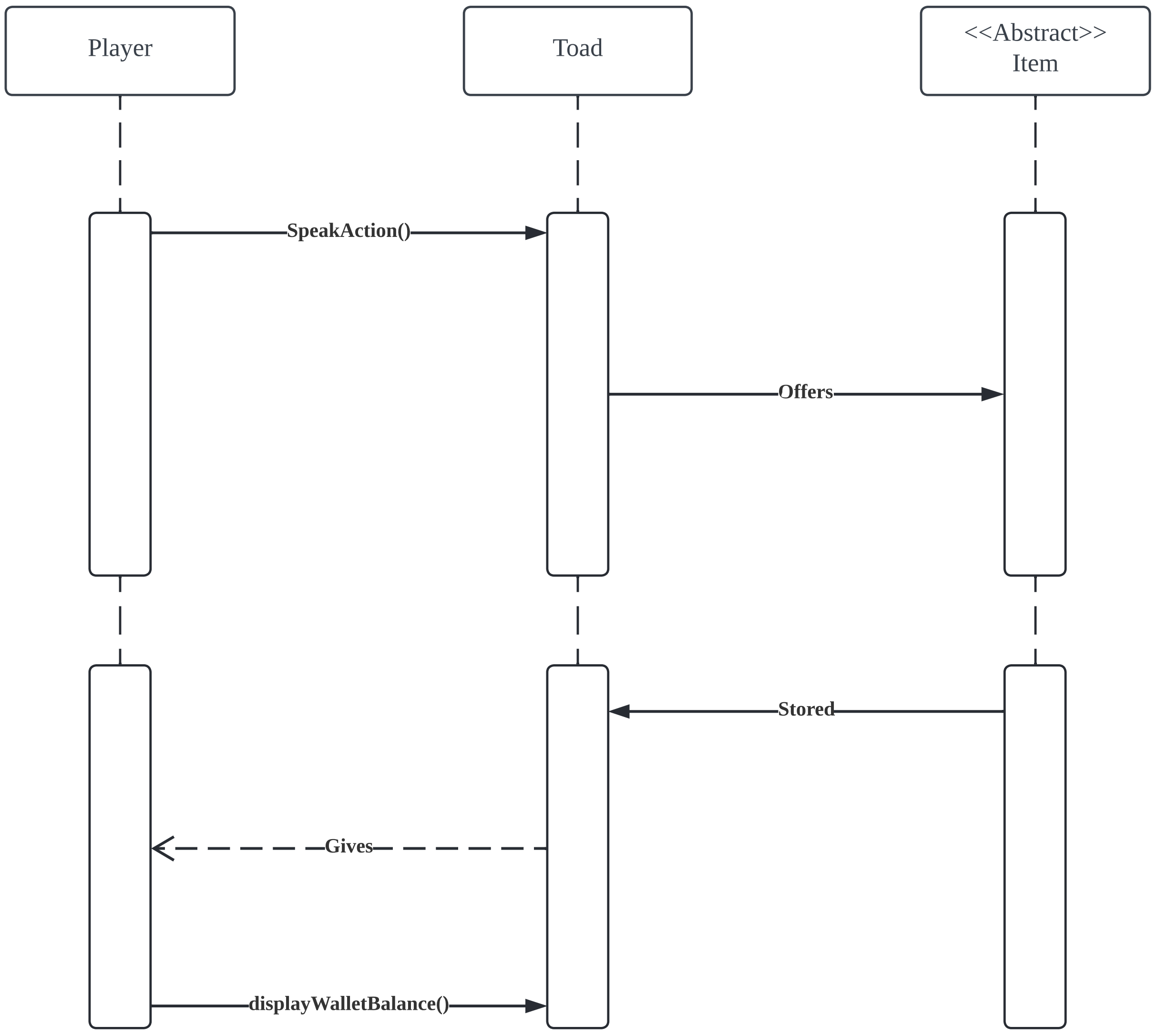


# Interaction Diagram

Picking Up Coins



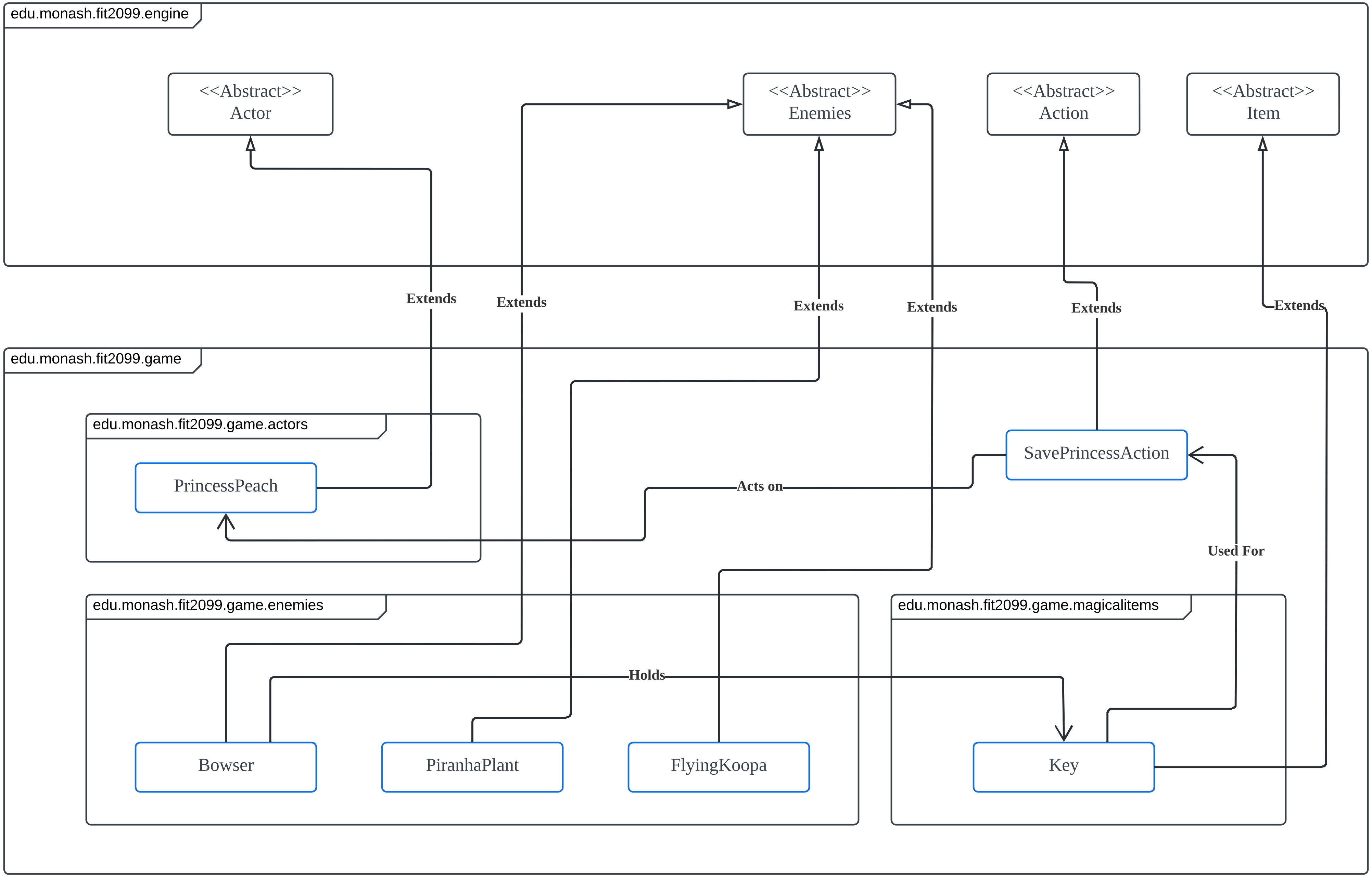
Buying Items from Toad



# **ASSIGNMENT 3 - REQ 2:** More allies and enemies!

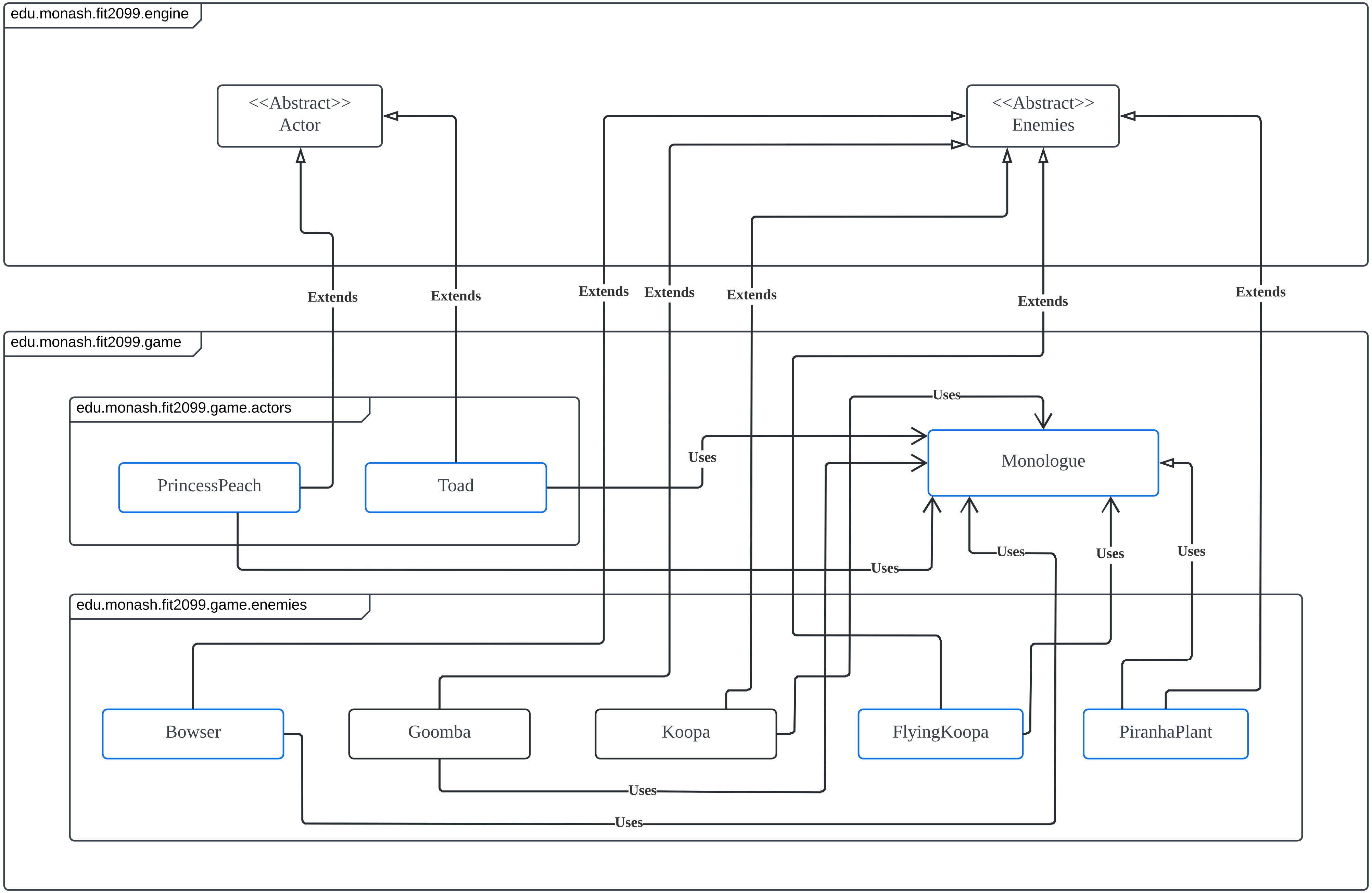
Danesh Mariapan

Princess Peach & End Game Parts



# **ASSIGNMENT 3 - REQ 5:** Speaking (Structured Mode)

Danesh Mariapan



**Lab11 Team3**
**Group:** Seow Zheng Hao, Muhammad Abdullah Akif, Danesh Mariapan

# Design Rationale

- REQ 1

  Assignment 1

  The 3 Sprout, Sapling, Mature classes will extend the tree class to get similar base attributes, since the 3 types of trees have different rates of growth and different functions, extending them to a base class will reduce redundancy.
  **Sprout class:** a small tree with different growth rate extending the tree class
  **Sapling class:** a medium tree with different growth rate extending the tree class
  **Mature class:** a big tree with different growth rate extending the tree class

  Assignment 2

  1) removed sprout, sapling and mature tree classes as there is no need to create too many inheritance. The 3 child classes do not share any methods i.e there are no functions that all 3 child classes share, so parent class will just be abstract, it will be easier to manage the functions of each tree type in the tree class itself, as seen by a different class method being called for every stage of a trees life. Every tree on the map has to go through all 3 stages. Following the **single responsibility rule**, one tree class manages every stage of the tree and contains all the possible functions.
  2) added koopa and goomba classes as sprouts and saplings have a fixed chance to spawn enemies every turn as well as the classes they extend (enemies and actors)
  3) added the coin class along with its dependent classes as saplings have a chance to spawn coins every turn, there is no need to create additional classes to handle actor or item creation.

  Tree class

  Manages all the life cycles of a tree (sprout,sapling,mature,dead) Contains a tick counter that checks the current life stage of the tree to direct to the correct life stage methods. Has a method that is called by mature trees to spread to nearby fertile grounds. Has a callable resetinstance to reset trees when resetaction is chosen by player as well as the registerinstance to register tree classes as resettable.

  Koopa,Goomba and Coin class

  The 3 classes can be called by tree classes in different life stages. Koopa can be spawned on top of the tree at a fixed chance once a tree reaches mature stage. Goomba can be spawned by tree at a fixed chance during the sprout stage. The sapling tree stage can spawn coins at a fixed chance.

- REQ 2

  Assignment 1

  Jump action is only going to be called by mario as enemies cannot jump to higher ground. Walking off from high ground to low ground is guaranteed and can be done by mario and other enemies. Needs to perform a check for player status for super mushroom before calculating percentage. then check what object Mario is trying to

jump to, then call the item class and prompt them for a jump. The jump function in the tree classes will save the chances and output a boolean for success or failure.
**JumpAction class:** will depend on the ground class to check for jumps, only player can jump, enemies cannot
**Walk down class:** all actors can walk down from high ground

Assignment 2
1) Corrected jumpaction into the game files, removed walkdown entity as there is no need for extra classes.
2) added extension for jumpaction class to action class as the class uses similar methods and overrides the differences. This follows the **interface segregation principle** and ensures that there is **no unnecessary repeating code.**
3) added dependency to ground class for jumpaction as jumpaction knows the existence of ground class objects to check for jump success chances.
4) added dependency to jumpaction class to application class as the application calls jumpaction class when an actor gets the option to jump as well as display the result of the jump given from jumpaction class

Jumpaction
The jumpaction class can be chosen by the player when the exit is a tree or wall. Players have a fixed chance to succeed the jump depending on the jump target, and failure leads to damage done to the player.

- REQ 3: (Updated Version)
  Goomba and Koopa will be created on the map in the Application class, at the start of the game. Both Goomba and Koopa will extend the abstract class, Enemy because both these will be using the methods present in the Enemy abstract class. This will reduce redundancy and follows the DRY principle. The enemy abstract class will extend the Actor class. When a Koopa's shell is destroyed it will drop a Super Mushroom, hence it would have a dependency with a new class called SuperMushroom. (We will be talking about the SuperMushroom class in REQ4)
  **Enemy abstract class**: class that extends the Actor abstract class.
  **Koopa class:** an enemy class that extends the Enemy abstract class.

- REQ 4: (Updated Version)
  The two classes, SuperMushroom and PowerStar will have an association with the MagicalItems class as an object of each of the above two classes will be made. The SuperMushroom and PowerStar classes both will extend the Item abstract class as both will be using methods from the item class, this reduces redundancy and follows the DRY principle. The MagicalItems class will extend the Action abstract class as item uses methods and hence, will reduce redundancy. Which will result in following the DRY principle.
  **MagicalItem class:** a class that extends the Action abstract class
  **SuperMushroom class:** a class that facilitates SuperMushroom and it extends the Item abstract class
  **PowerStar class:** a class that facilitates PowerStar, it extends the Item abstract class.

**Lab11 Team3**
**Group:** Seow Zheng Hao, Muhammad Abdullah Akif, Danesh Mariapan

Assignment 1:

- Danesh REQ 5:
  **Toad Class:** Will have a Dependency linking all the other three usable Item Classes (Wrench, SuperMushroom & PowerStar), since the Toad is supplying the items to the Player, and they can only be obtained through the Toad.
  **Coin Class:** Has an Association from the Sapling Class to itself as the Saplings produce/spawn the Coins on the game map. Each coin object will have an integer / currency value which adds to the Player's wealth when picked up
  **CoinWallet Class:** is added as a wallet system to the Player, and is an Aggregation relationship to the Coin Class. This is where the currency value of the Player's total wealth will be stored.
  **CoinPickUpItemAction:** This class is an added action in which the player uses when he/she is next to a Coin on the game map, the Coin is then removed from the map and the integer value (currency value) of the Coin is added to the player's CoinWallet (which stores the integer value of the Player's total wealth)
  **TradeAction**: This class is an added action used when the Player is trading with the Toad. When used, the Player "buys" an item from the Toad a currency transaction is made - the cost of the item is deducted from the Player's CoinWallet's total wealth, and the item is added to the Player's Inventory.

  Assignment 2 Implementation:

  We follow the **Single Responsibility Principle** by having separate Classes for each of the Coin functionality (managing the Objects, managing the values separately in the Wallet, Picking up the Coins and Trading with Toad. This ensures that each of these classes have their own role / task - which makes adding functionality easy. We also follow the **Dependency Inversion Principle** as we are overriding methods of other Classes in the engine to create our own functionality. This also ties in with the **Interface Segregation Principle** - for example with the Resettable interface, allows for reset methods in the Classes that need these. These also apply to **Liskov's Substitution Principle** as most of these subclasses can be easily implemented in different ways without disrupting the behaviour of the program as most are well modularised and abstracted. When implementing all functionalities **Open-Close Principle** was also used with the abstraction taken from the engine Classes for the cleanest and best design implementation and integration.

- Danesh REQ 5:
  **Coin Class:** Here, Coin extends Item Class and implements Resettable Class as we want to be able to use most of its functionality (tick, resetInstance, registerInstance, etc). However, I have overridden PickUpItemAction and DropItemAction, and set them to return null (we do not want to use this functionality from Item Class). As we want Coin objects not to be stored in the inventory, but in our CoinWallet instead and also we do not have any use for the DropItemAction implementation (dropping coins). We will then create our own CoinPickUpItemAction for pick up implementation to be added to CoinWallet, just for Coin types and not generic Item types.

**CoinWallet Class:** This also extends Item Class, which is not portable and is added into the Player actor's inventory (for practicality seen in like most games). Here, a HashMap implementation is added to store the Coin objects (I store it using key: Coin number - representing the order in which the Coin was picked up, and value: Coin Object). A displayWalletBalance() method is also added which goes through all the coins in the HashMap and shows the total currency value the Player has and how many Coin objects are in the wallet.

**CoinPickUpItemAction:** This extends Action class and implements the picking up action a bit differently - by overriding the execute() method, we set it so that the Coin object is added to the CoinWallet directly and the Player's wallet balance after every Coin picked up will also be visible in the console.

**TradeAction:** This extends the Action class, and we override the execute() method here to implement functionality to calculate if there is enough Coin value stored in the wallet to buy each item the Toad offers. If the Player does have enough, he/she can buy the item and the item is added directly to the Player's inventory and the cost of the item is deducted from the wallet, as well as the Coin object used to create the value for payment, will be removed from the HashMap.

**Toad:** Toad extends the Actor Class as it is a friendly NPC. The items for sale and their costs respectively are also stored in a HashMap. We also override the allowableActions() method in the Actor Class to add for the new Trade Actions that should be allowed to be used on the Toad.

Assignment 1

- Danesh REQ 6:
  **SpeakAction:** Is an added Action that has an Association from the Player Class and extends the Abstract Action Class, as it is a type of Action in which the player can use when he/she is near the Toad and wants to interact with it. Only after this Action is used will the Player see the Toad's monologue and shop items to buy.
  **ToadSpeakBehaviour:** Is an added Behaviour that has an Association from the Toad Class and extends the Interface Behaviour, as it is an NPC behaviour in which the Toad has to follow when being interacted with by the Player. The behaviours then target the Player using the FollowBehaviour Class.

Assignment 2 Implementation

- Danesh REQ 6:
  **Monologue:** This is a generic public class created to hold/store all of Toad's lines (the sentence options that Toad can choose to say to the Player actor). All of the sentences are declared as static strings and they are stored as a static list object, so that they can be accessed from other Classes.

**SpeakAction:** This extends Action class, and I have overwritten the execute() function to choose a sentence randomly from the sentences stored in the Monologue Class. If the player is holding a wrench or has Power Star Effect as well, certain sentences cannot be chosen. For the randomness of the choosing, I've created some methods that generate random numbers between different index values in the Utils Class. The target here is the actor that we want to speak to, being the Toad.

- REQ 7:
  Assignment 1
  A new class reset game is created and calls the different classes that need to be removed once the game is reset.
  **Rest Game class:** calls the classes that will be affected by resetting the game

  Assignment 2
  1) corrected direct association to enemy classes, resetgame now goes through the enemy abstract classes instead to access enemy actors.
  2) resetgame now directed towards the player class to clear status rather than the status enum
  3) In the game, reset game can only be called after all entities are ticked at least once as the tick function gives each entity affected their location on the map.
  Resetaction class
  Reset action class calls the reset manager and runs the run() method. The run method then goes through all the resettable classes in the list and calls the resetinstance() method. The resetinstance() method in the resettable classes will then reset the individual resettable classes. Only Coin,Tree,Enemies,Player classes are resettable.Each class has different probability to be reset when resetaction is chosen.

## Assignment 3

Assignment 3 Req 2
1) change canactorenter in tree for flying koopa so that it can 'fly'
Req 4
1) Assume cannot create fire on top of pipe
2) Aded FloorOnfire() method in enemies as all enemies will use it to check for damage from fire attack. Doing it in the enemies class prevents repeating code in all enemy classes. This aligns with **Liskov's Substitution Principle** as all enemy classes will have access to this method.
3) Added ConsumeFireFlower action class for actors to consume fireflower on the ground when standing on it, similar to the coin, disappears after picking up, multiple flowers can be at the same location
4) show multiple options to consume different fireflowers in one location in case the player wants to consume all to keep refreshing fire attack duration.

**Lab11 Team3**

**Group:** Seow Zheng Hao, Muhammad Abdullah Akif, Danesh Mariapan

Assignment 3 Req 2 [Princess Peach & End Game] - Danesh Mariapan

- **PrincessPeach:** This extends the Actor class and is another character in the game. Princess Peach cannot move/attack. She must be saved by Player (Mario) after defeating Bowser and obtaining a Key to win the game. The new action created for the endgame - SavePrincessAction is implemented here and targets this object (Princess Peach).
- **SavePrincessAction:** This extends Action. The Player (Mario) can use this action when he is next to Princess Peach. If he has the Key (bowserKey) in his inventory, he will be able to interact with PrincessPeach and save her. A congratulations, Winning message is then printed to the Console. However, if he does not have the Key, Princess Peach will say a line, hinting on how to get the key to save her, printed to the console. I have overridden the execute() function from the parent Action class to create the functionality, and have set the target to Princess Peach in the PrincessPeach class to allow this new Action to be done on her.
- **Key:** This class extends Item Class and implements all of the functionality of a normal Item. It is portable and is held by Bowser (a Key is put into Bowser's inventory when Bowser spawns). Only when Bowser has died/been killed, the Key item is dropped onto the map, where the Player can pick it up to save Princess Peach.

Assignment 3 Req 5 [Speaking] - Danesh Mariapan

- **Monologue:** I created this Monologue Class to hold all the possible sentence lines of all Enemies/Allies in the game. They are stored as Static final Strings, as each Character can only say a few specific lines that don't change and to allow for easy access of these lines in other Classes. I then grouped the sentences by storing them in different String Java Arrays for each Character - this makes it easier to randomly access a sentence that corresponds to only that specific type of Character that can say it. I've then implemented getters for all these String Java Arrays which then returns their corresponding String Java Array when used in other Classes.
- To implement the Monologue, I've overridden the playTurn() function in all of the individual Characters' classes and implemented a turnCounter (to count the number of turns since instantiation). The String Java Array getters are then utilised here in the playTurn() functions along with random number generators from Utils class to access that Character's possible sentences and have them output a random one.

- **Single Responsibility Principle** is followed, new Actions and Items are created for a specific purpose to keep the codebase concise.
  **Open-Closed Principle** is Applied, as I extended new Classes from generic parent classes like Action & Item in order to keep the general functionality.
  **Liskov's Substitution Principle** is also Applied, to prevent high coupling problems, for instance implementing the Monologue Class, in which can be used by all the other Character classes to access their own Sentences.
  **Interface Segregation Principle** is also used - in the case of the Resettable Interface being used by multiple Classes in order to add the reset functionality.
  **Dependency Inversion Principle** is also followed as a lot of our used Classes have extended the more generic parent classes to use its functionality and apply better abstraction - more specific classes can easily be added/deleted for the Game.