FIT2099 Assignment 2
2022 Semester 1
Lab 15 Team 4

Team members:

1) Chan Jia Zheng (32223315)
2) Caelan (30524105)
3) Justin Chuah (32022433)

Rationale

REQ 1
Updating Tree class:

We decided to add an age attribute to the Tree class. Age will be an integer and will increase by one every game tick. This is to help facilitate the growing of the Tree, where there will be a method that updates its display character, and conditionals to direct it to its appropriate function, as a tree does different things at its sprout, sapling and mature stage.

New abstract class, Enemy:

We decided to create an abstract class, Enemy, that extends from Actor. Goomba and the new Koopa class will extend from Enemy. This is because both Goomba and Koopa are similar in function, that they share many of the same behaviours and functions, so it would be simpler to implement now that they extend from a parent class. This also helps facilitate future extensions to the code if we need to add enemy varieties, or help separate any non hostile Actors.

New Coin class and player attribute, wallet:

The coin class will extend from the abstract class Item. This is because a sapling needs to be able to spawn coins and place them on the ground for the player to pick up. The Coin class will also carry an integer attribute, its value. This is so that we can track how much a coin is worth, and easily access that information to add to the player's wallet. The wallet will be an integer attribute added to the Player class to keep track of how many coins the Player has at any given time, and so that mathematical operations can be performed on it whenever the player picks up coins or buys items, spending them.

The method is implemented as we can destroy the Coin objects when picking them up, it is unnecessary to store the coin objects, we only need its value after picking them up.

Assignment 2 changes

Thress classes for Three Tree Stages:

A class was created for each of the Tree's stages, Sprout, Sapling and Mature. This enabled better readability of the code as each stage has a different function, as such should have its own class to contain the respective function. They also inherit from the now abstract Tree class which contains the age counter in its tick function. This is to adhere to the DRY principle. The Tree class has also made abstract now because a Tree object is never directly created, and functions just as a "hub" for its children to inherit their body from.

REQ 2

New class, JumpAction:

The JumpAction class will extend from the abstract class Actions. The plan is that if Player wants to move to a high ground, wall or tree, it will generate a random number and based on requirement specifications, decide if the player will move to said high ground by calling MoveActorAction() or fail, deduct the player's health points and call DoNothingAction() instead.

As such, we can reuse the MoveActorAction() and DoNothingAction() of the original code.

Assignment 2 changes:

JumpAction and a new interface, Jumpable:

The JumpAction class still extends from the abstract class Actions, however it now directly interacts with a new interface, Jumpable. The Jumpable interface contains the default methods that handle jump checks, as the mechanic is the same for all Jumpable objects, with the only differences being success rate and fall damage. This helps adhere to the DRY principle.

The player is moved to the high ground without question if they have the TALL status, otherwise a random number is generated to decide if the player passes the jump check, where health will be deducted if the check fails. The functions of the code are all done within the function implementations, instead of reusing MoveActorAction() or DoNothingAction().

REQ 3
Creating a new abstract class Enemy as both the Goomba and the Koopa operate according to a similar foundation. Both classes use an attacking system that deals a fixed amount of damage with a certain chance of landing the attacks. They will attack other actors that are hostile to them.

Added two status enumerations namely “DORMANT” and “HASWRENCH” as these statuses will be useful for the interaction with the Player and the Koopa class. DORMANT status will be used to represent the beginning for change of the Koopa (e.g: going from K to D when being displayed), causing it to only be able to do nothing in its action list.

Adding a BreakShellAction class whichs extends the abstract action class which allows the action of breaking a Koopa Shell. This action will only be added to the Player's action list if the player owns a wrench which will be represented by the HASWRENCH status, and the Koopa is in Dormant state.

BreakShellAction has 2 Actor attributes, which are the actor who wants to attack the Koopa, and a Koopa actor to be attacked.

Koopa will have an extra item SuperMushroom in its inventory when Koopa is created.

Assignment 2 changes

Removal of the SuperMushroom class diagram in the UML diagram. This is because during our implementation, the only time a SuperMushroom appears for this specific requirement is when the Koopa shell is destroyed by the Player through the BreakShellAction(). And said SuperMushroom was obtained through retrieving the inventory of the Koopa on death(every Koopa has a SuperMushroom in its inventory on spawn), so there is no apparent relation for SuperMushroom with the rest of the class.

Change of the multiplicity for the BreakShellAction() as our initial idea of this action was a mistake, as one BreakShellAction() is only targeted to one actor.

Addition of a PRE_DORMANT enumeration for the Koopa class. The Koopa class will automatically have this status on spawn, and will be removed when it becomes unconscious, replaced with the DORMANT enumeration instead. The addition for this status is mainly to allow the Koopa to be able to continue staying on the game map until it is destroyed through BreakShellAction().

REQ4

Adding another status called INVINCIBLE to the status enumeration. This is to represent the new set of functionalities that the player will be capable of doing after consumption of Power Star.

PickCoinAction is so that once the player picks up a coin on the map, the Coin then gets immediately added into the wallet(which is an attribute of the player), the Coin is then destroyed, which is different from the engine PickUpItemAction that gets added into the inventory instead. It will be returned when the player calls the item getPickUpAction().

PickCoinAction has an attribute Actor, which is the actor who wants to pick up the Coin; and an attribute Coin, which is the Coin to be picked up.

New Abstract class Consumable. Since only SuperMushroom and PowerStar can be consumed by the player, they have been placed under a consumables abstract class to simplify further implementations of them when required.

ConsumeItemAction is a specific action that can only be performed when the specified items of choice falls under the Consumables abstract class. It will be returned when the player calls the item getAllowableActions() from the player’s inventory.

ConsumeItemAction has an attribute Actor, which is the actor who wants to consume an item; and an attribute Consumable, which is the Item to be consumed.

Assignment 2 changes
Changed the class name of PickCoinAction to PickUpCoinAction for clarity purposes.

Changed the relationship of PickUpCoinAction and ConsumeItemAction with the Actor abstract class. The main reason for this is that during our implementation, it was not required for us to create a new Actor attribute in these two actions, instead the Actor in the constructor was sufficient for these two functions to be functional (actor(In this case Player) executes the action on himself). PickUpCoin action gets directly added to the attribute Wallet found in the Player class and the ConsumeItemAction has the effects of the item consumed applied onto himself.

REQ 5 & 6
Create an Action class TalkToToadAction and add the keyword “TALK_TO_TOAD” to Enum Status. The Actor Toad allows other Actors to talk to it if the Actor has capability Status.TALK_TO_TOAD. TalkToToadAction is the class who is responsible for choosing suitable dialogue based on the player status. 1 TalkToToadAction has 1 Actor attribute, which is the actor who wants to talk to Toad.

Also, create an Action class BuyItemAction and add the keyword “BUY_ITEM” to Enum Status. The Actor Toad allows other Actors to buy Items from its inventory if the Actor has capability Status.BUY_ITEM. 1 TalkToToadAction has 2 Actor attributes, which are the buyer and seller; and 1 Item attribute, which is the item be to traded.

The 2 actions will be returned by the actor allowableActions() in processActorTurn(). As such, we don’t need to modify the classes outside the classes created.

Assignment 2 changes:

1) BuyItemAction associates only with an Actor object, which is the seller.

In the previous implementation, it associates with 2 Actor objects, the buyer and seller. In current implementation, the buyer Actor will be provided in method execute() when the buyer calls it.

2) Added a new interface, Buyable.

An Item can only be traded by BuyItemAction if the Item has a price. So, there is a method getPrice() in interface Buyable. For Item that implements Buyable, instead of directly returning the price as a number in getPrice(), it returns the variable 'this.price'. The variable 'this.price' is set in the Item constructor. This is to eliminate magic numbers in the class, It makes future modification easier. This follows Connaissance of Meaning.

3) Set BuyItemAction constructor to private, added a new public method getInstance()

BuyItemAction should only be instantiated if the Item provided is Buyable. BuyItemAction should be responsible to check if the Item provided is Buyable and decided whether to instantiate a new BuyItemAciton for the Item provided. The follows the principle Single

Responsibility.

Besides that, we don't need to write code in other classes to check if the Item is Buyable,
It follows the principle Don't Repeat Yourself.

REQ 7
Class Player, Coin and Tree should extend and abstract class Enemy should implement interface Resettable. Whenever a Resettable object is created, we should call the method registerInstance() to store the object in ResetManager. As such, we have a collection of Resettable objects on the map, actions can be performed on them at once when needed.

Also create another Action class which is ResetAction. There should be an attribute allowReset (boolean) in the class ResetManager to indicate whether reset action is allowed to be performed by the user. We can add the ResetAction into the ArrayList actions in method processActorTurn (in Class World), if the user is allowed to reset. The ResetManager should be the class who decides whether the player can restart (Single Responsibility Principle).

When the ResetAction() is executed, call the run() method in ResetManager. All the objects that are children of the interface Resettable will be reset accordingly.

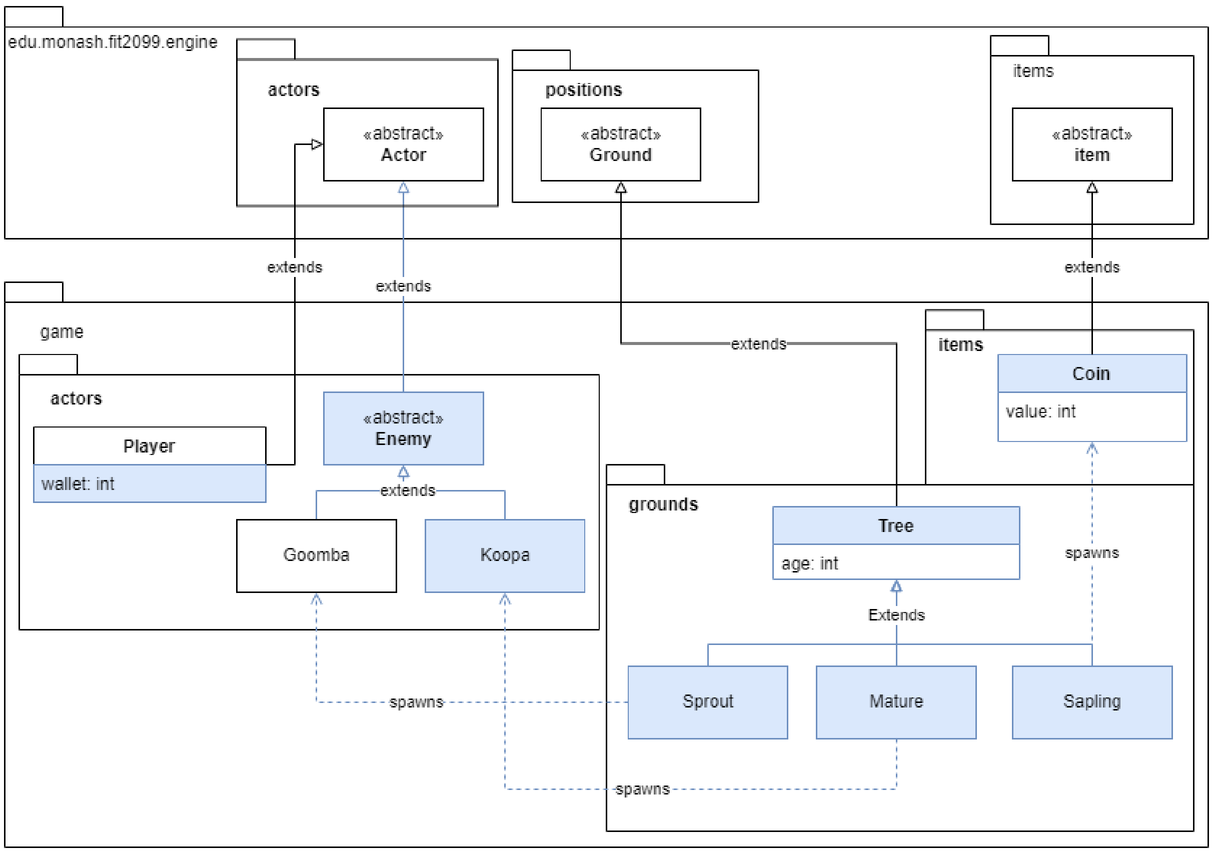
Assignment 2 changes:
1) Added a new Status RESET

When the ResetAction is selected by player, the action to reset the Player is performed in the
Player resetInstance() method.

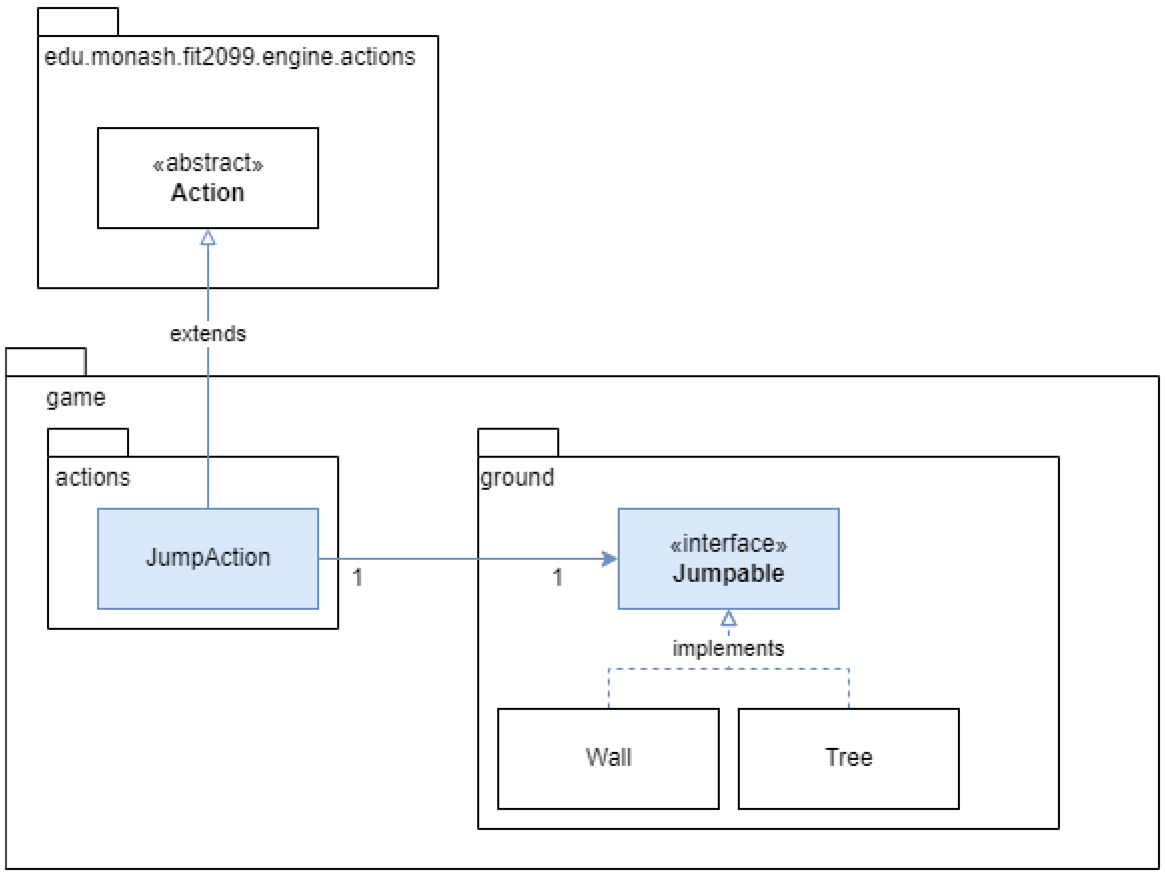
In the resetInstance() method of other object, I will and add the Status.RESET to that object.
Then reset action's logic will be implemented in:
1) PlayTurn() method if the Resettable object is Enemy
2) Tick() method if the Resettable object is Coin or Tree
This is because it needs to access the location of the Resettable object. As such,
we don't need to modify the engine code. This follows Open Close Principle.

All the actions created by above requirements are created such that they will be returned as Action by the code in processActorTurn(). As such, we minimise the need to modify the original code or modify the existing code.

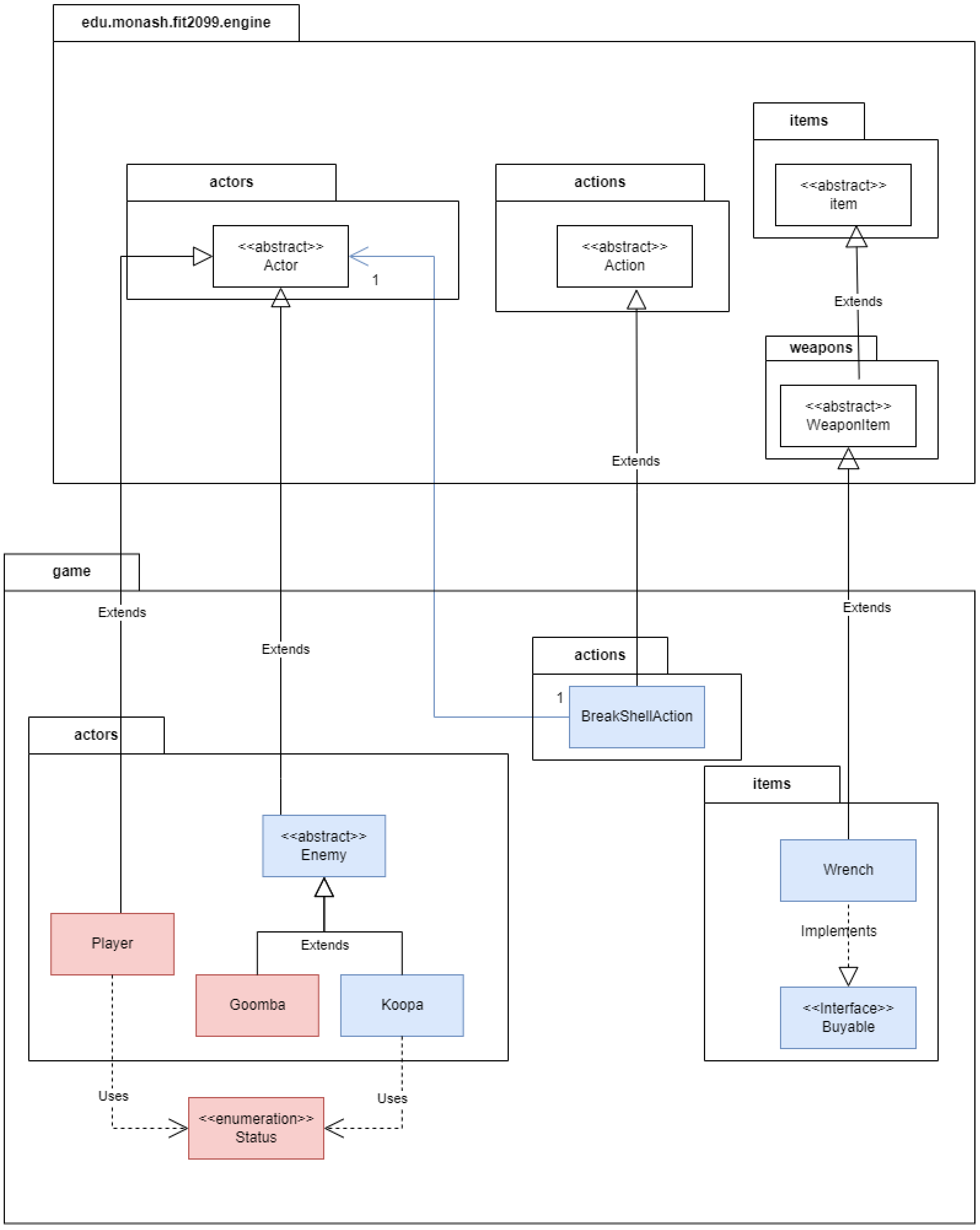
UML Class Diagram for Requirement 1



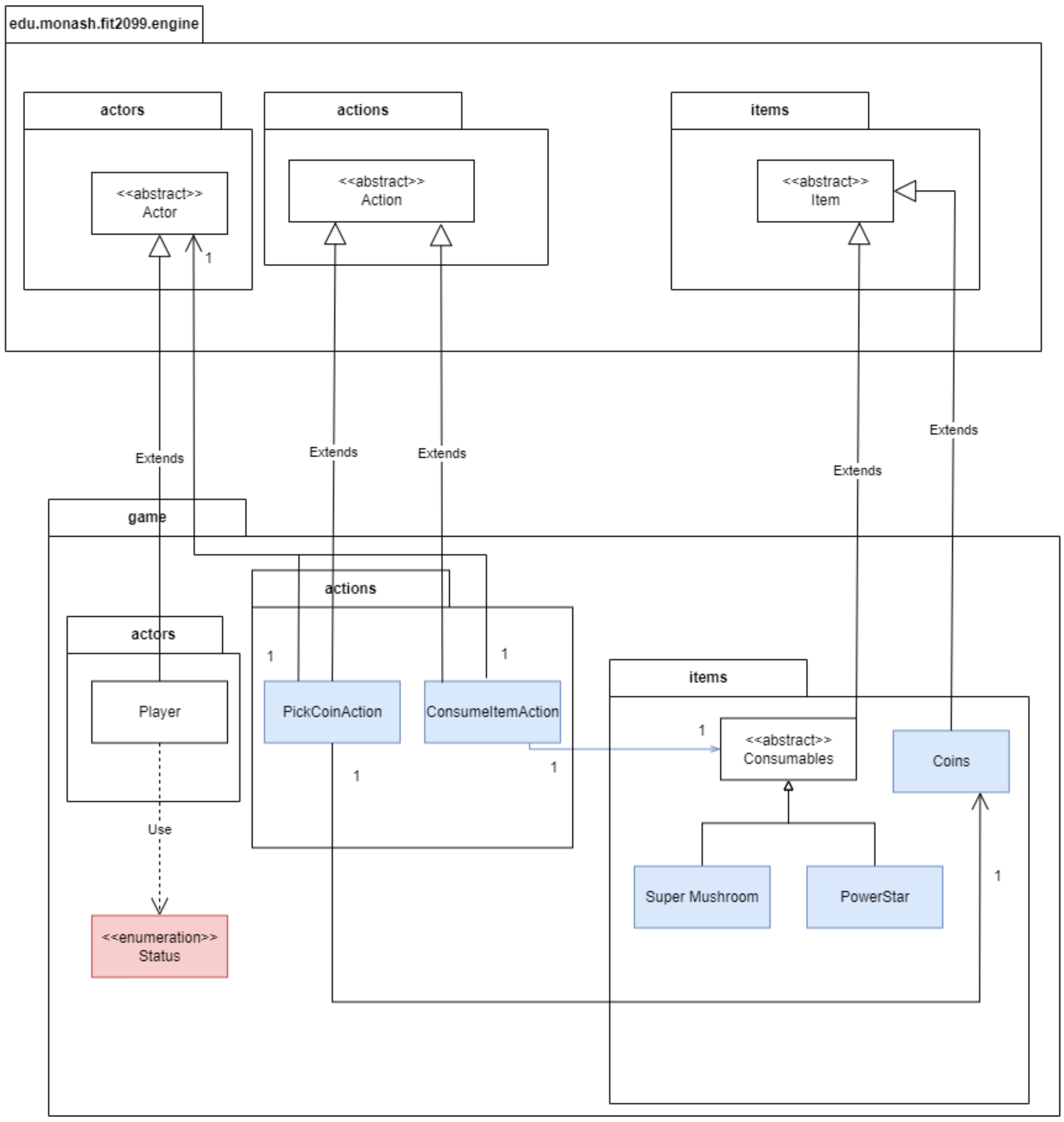
UML Class Diagram for Requirement 2



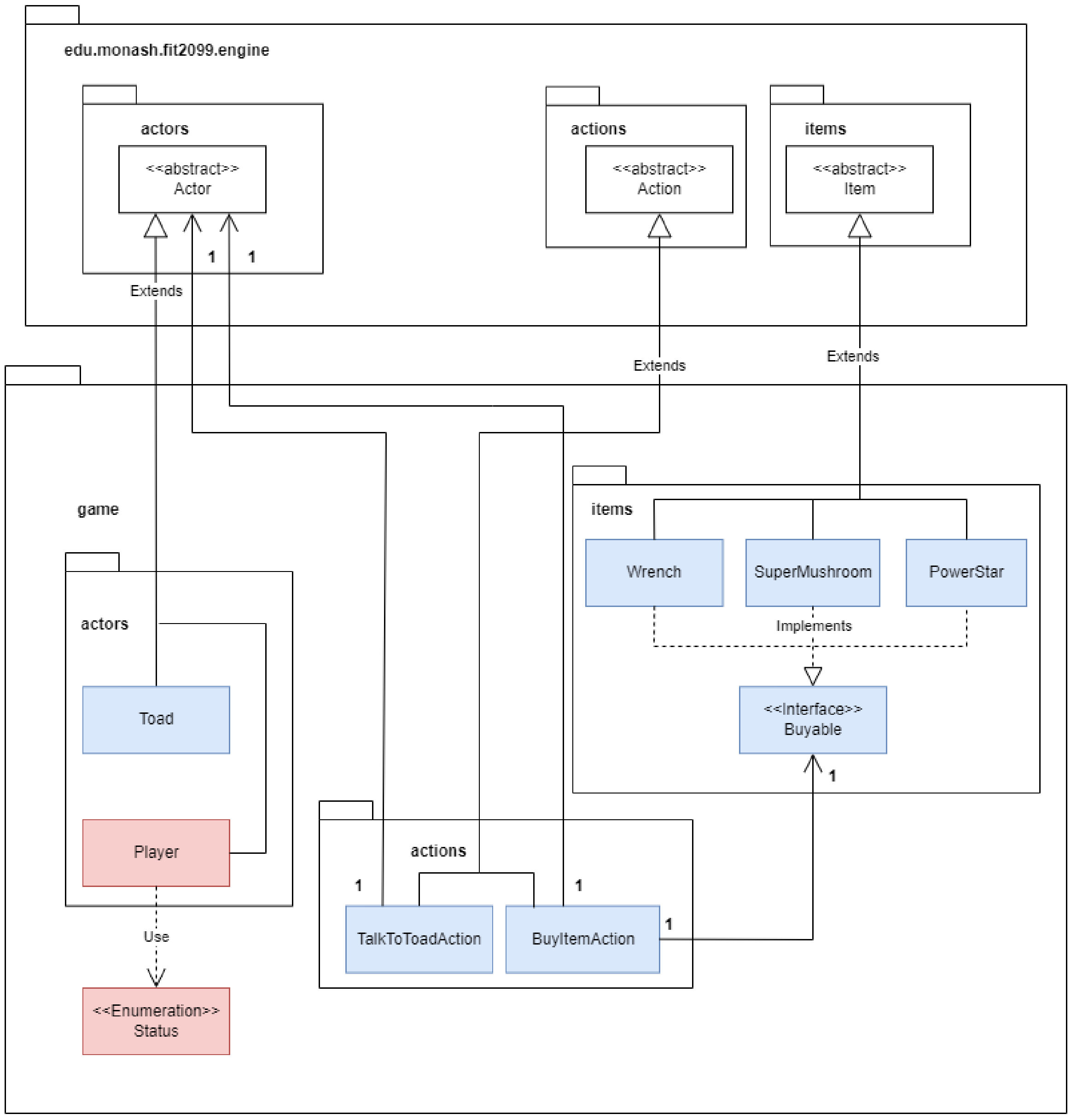
UML Class Diagram for Requirement 3



UML Class Diagram for Requirement 4



UML Class Diagram for Requirement 5 & 6



Work Breakdown Agreement

We hereby agree to work on FIT2099 Assignment 2 according to the following breakdown:

Assignment 1:

Implementation planning:

- Requirement 1 to 4 - Caelan & Justin
- Requirement 5 to 7 - Jia Zheng

UML diagrams & Design Rationale:

- Requirement 1 & 2 - Caelan
- Requirement 3 & 4 - Justin
- Requirement 5 to 7 - Jia Zheng

I accept and agree to this WBA - Caelan Kao (30524105)
I accept and agree to this WBA - Chan Jia Zheng (32223315)
I accept and agree to this WBA - Justin Chuah (32022433)