FIT2099 Work Breakdown Agreement

Team Number	Lab 16 Team 2
Names	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934
Date	29/3/2022

Our team will break down our work by assigning tasks to each member. For each requirement, the assigned member will be responsible for creating the respective UML Class Diagram, UML interaction Diagram.

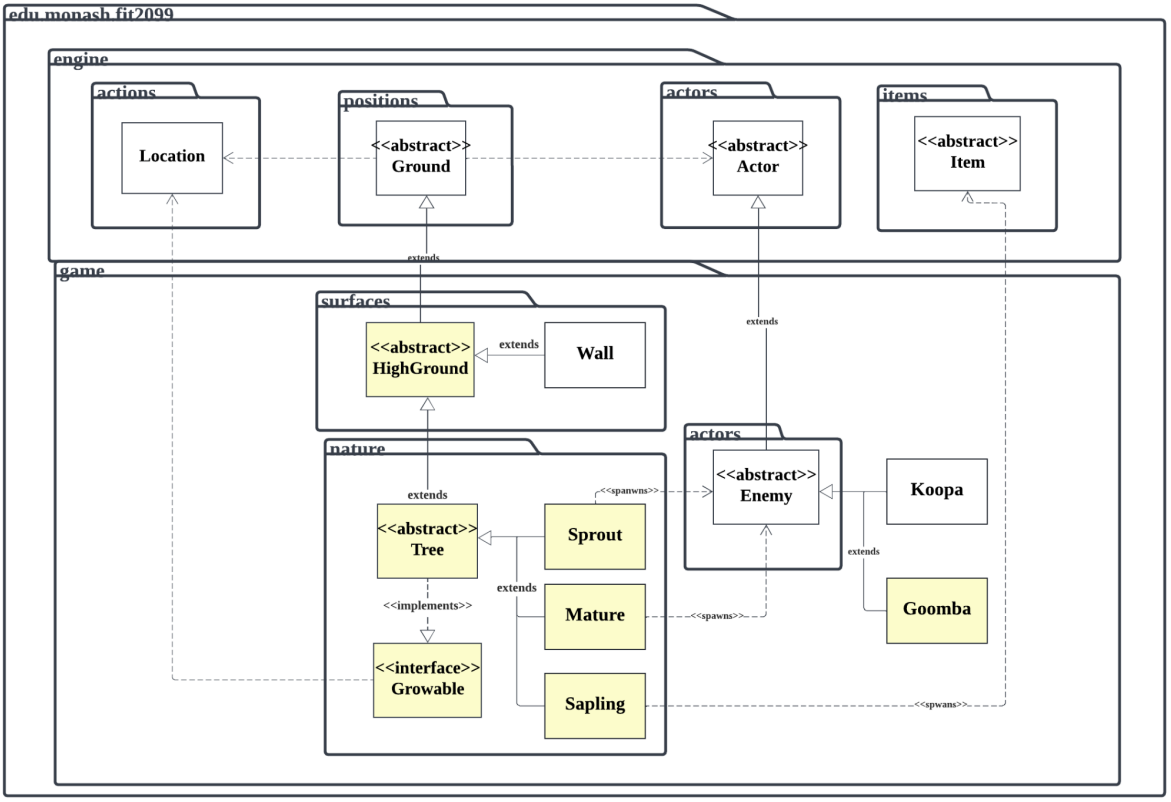
Task	Member	Date of completion
Requirement 1	Low Lup Hoong	20/4/2022
Requirement 2	Low Lup Hoong	20/4/2022
Requirement 3	Garret Yong Shern Min	22/4/2022
Requirement 4	Jastej Singh Gill	24/4/2022
Requirement 5	Jastej Singh Gill	25/4/2022
Requirement 6	Garret Yong Shern Min	28/4/2022
Requirement 7	Garret Yong Shern Min	30/4/2022
Reviewing	All members	2/5/2022

FIT 2099 Object-Oriented Design and Implementation

Title	FIT2099_S1_2022 Assignment 1 Design Documentation
Date	10/4/2022, Sunday
Team Number	Lab16Team2
Team	1. Garret Yong Shern Min 31862616 2. Jastej Singh Gill 31107974 3. Low Lup Hoong 31167934

Requirement 1

UML Diagram



The above UML diagram is our improved version from Assignment 1. The changes will be explained below. Some additional dependencies are shown in the next UML.

Class HighGround

Class HighGround is an newly created class that inherits from engine's abstract Ground class.
Responsibility:

1. It is an abstract parent class to any grounds that will be high grounds. This class contains common high ground attributes such as jumpSuccessRate, jumpDamage rate that child classes such as Tree parent and its child classes. This achieves the DRY principle and Single Responsibility Principle. Wall and Tree parent class now extends from this class.

2. Reduces code repetition for allowableActions methods for all of its child classes. Therefore, for all of its child classes, this parent method will check if actor has "CAN_JUMP_ONTO_HIGH_GROUND" capability and return the JumpAction.

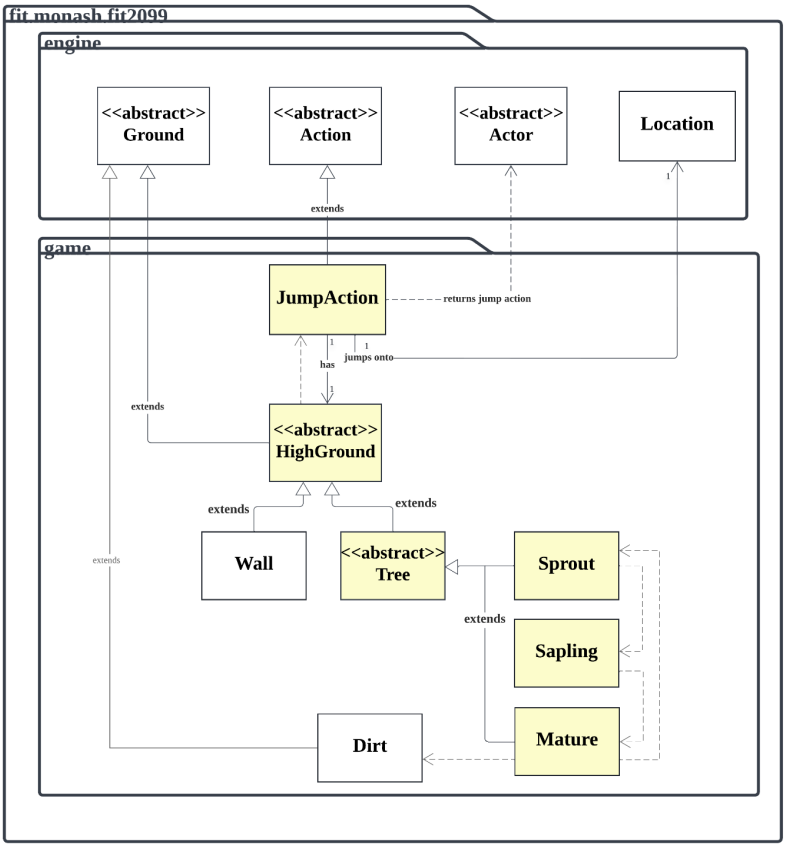
Removed Interface SpawnsEnemy and SpawnsItem

In Assignment 1, we created these 2 interfaces that were implemented by the tree concrete classes according to the requirements. However, as we got more familiar with the engine code after studying it for some time, we realised that we could easily make full use of the provided engine code to spawn these entities.

On top of that, making the concrete classes such as sprout to implement SpawnsEnemy interface means that it could only spawn one type of enemy, We realised that having concrete classes implement these interfaces reduced the flexibility of what we could do. Instead, we could just make use of the given engine code for more flexibility.

Requirement 2

UML Diagram

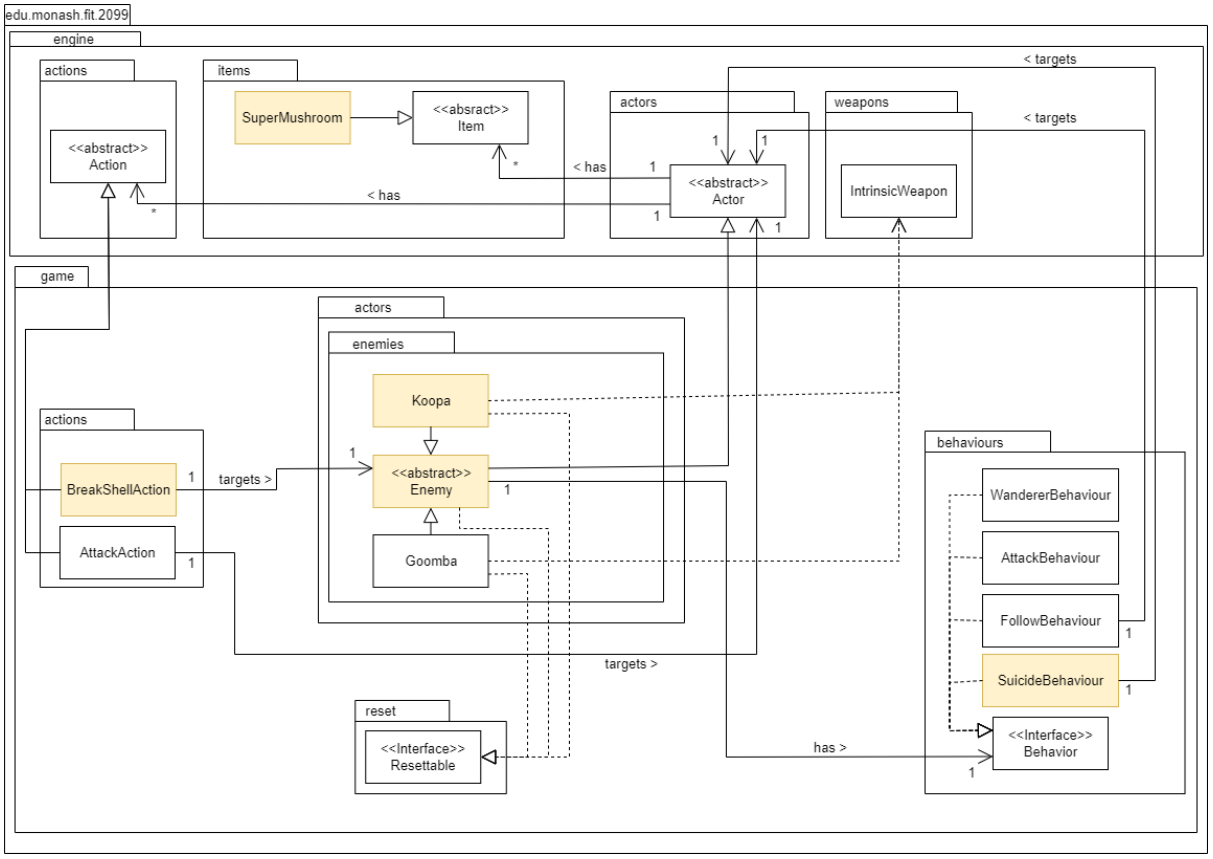


JumpAction now has an association with HighGround abstract class. HighGround has a dependency with JumpAction. All high grounds will return this JumpAction action to the actor if the actor has CAN_JUMP_ONTO_HIGH_GROUND capability.

Designing it this way reduces code repetition for high ground subclasses to implement the jump action.

Requirement 3

UML Diagram



The above UML diagram is our improved version from Assignment 1. Design's rationale will be explained below

Abstract Class Enemy

Abstract Class Enemy is a new abstract class which extends the abstract class, Actor. This class will be the parent class of all enemies created in future

Responsibility:

1. This class will provide most, if not all the appropriate behaviour required to fufill a basic enemy class. Upon extending this class, an enemy will already possess the FollowBehaviour, AttackBehaviour and WandererBehaviour so future classes do not need to repeat this block of code (DRY principle)
2. This class also has little to no additional constraints that would set it too far apart from the Actor class (Liskov's Substitution principle)

Koopa Class

Koopa class extends Enemy class hence inheriting many of the attributes and features provided by the Enemy Abstract class (DRY principle). This enemy has 2 sets of hp, one would be used when combating with any attacker while the other would act as a mediation to when the Koopa's shell is broken.

Changes

1. Additional item added to the Koopa's inventory, Super Mushroom. This will allow for the Koopa to drop the Super Mushroom upon death
2. Additional capabilities added to the Koopa, KOOPA_ACTIVE. This indicates that the Koopa is still in it's ative state
3. Method hurt is altered to ensure that during combat the Koopa's active hp will drain instead of the dormant hp. When the active hp is depleted, the Koopa's status changes from active to dormant and all behaviours is cleared
4. Instrinsic weapon is changed to a 30 damage punch

BreakShellAction

BreakShellAction is an action class that extends the Action abstract class. This class is used to break the shell of the Koopa during its dormant state.

Modifications

1. Will check the attacker's status to see whether the attacker possesses a wrench which is indicated with a HAS_WRENCH capability. If not, the attacker will fail to break the Koopa's shell and would have wasted a turn
2. If the attacker possesses a wrench, the attacker will remove the target from the map and the target will truly be killed

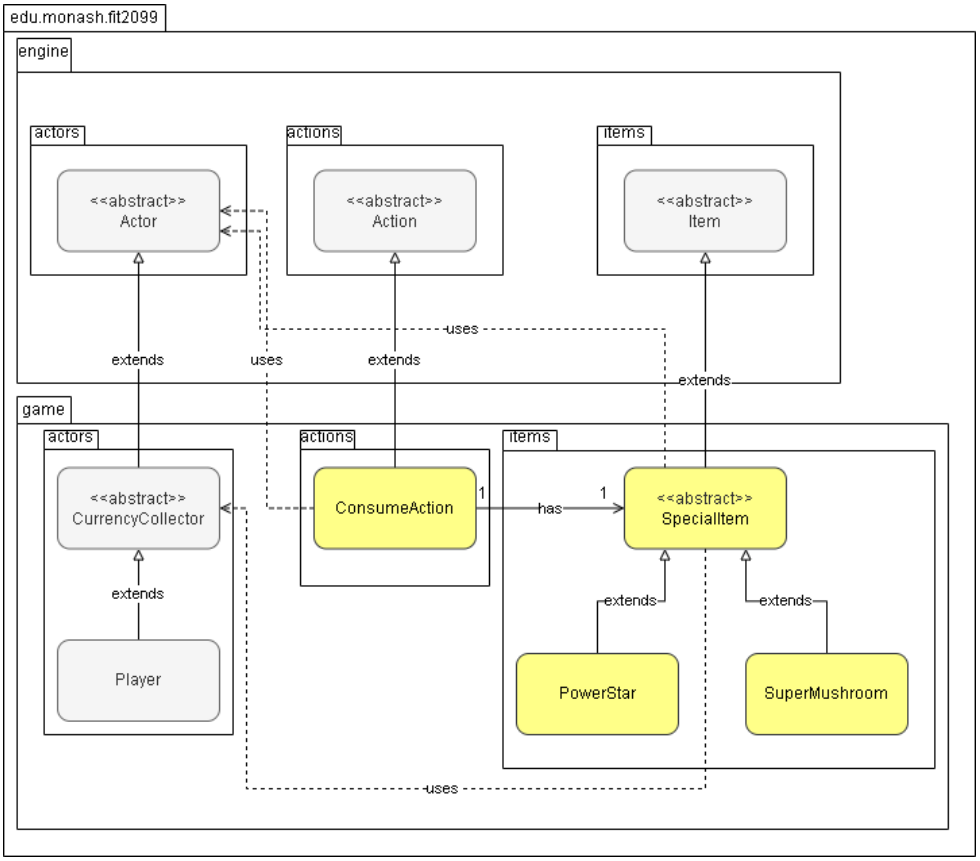
SuicideBehaviour

SuicideBehaviour is a behaviour class that would roll a chance for an actor to commit suicide. This class rolls for a 1 in x chance to cause instant death for an Actor.

This method's constructor accepts the any integer value to determine the chances, x, for the suicide to occur (open to future implementation)

Requirement 4

UML Diagram



Above is the UML Diagram for requirement 4. Any changes from the previous UML diagram is outlined below.

Class CurrencyCollector

A new abstract class inherited by the Player.

Responsibility:

1. Only allow certain actors to be able to pick up and store currency
2. Each actor will be assigned a wallet to store currency picked up
3. Has an assessor to access the wallet of each currency collector

Without this class every actor that can pick up currency will have to reimplement common methods and cause a breach in the do not repeat yourself (DRY) principle.

This class also leaves room for extension. If other actors besides the player require the functionality of picking up currency, we can achieve that by simply extending the CurrencyCollector class again which fulfils the open-closed principle.

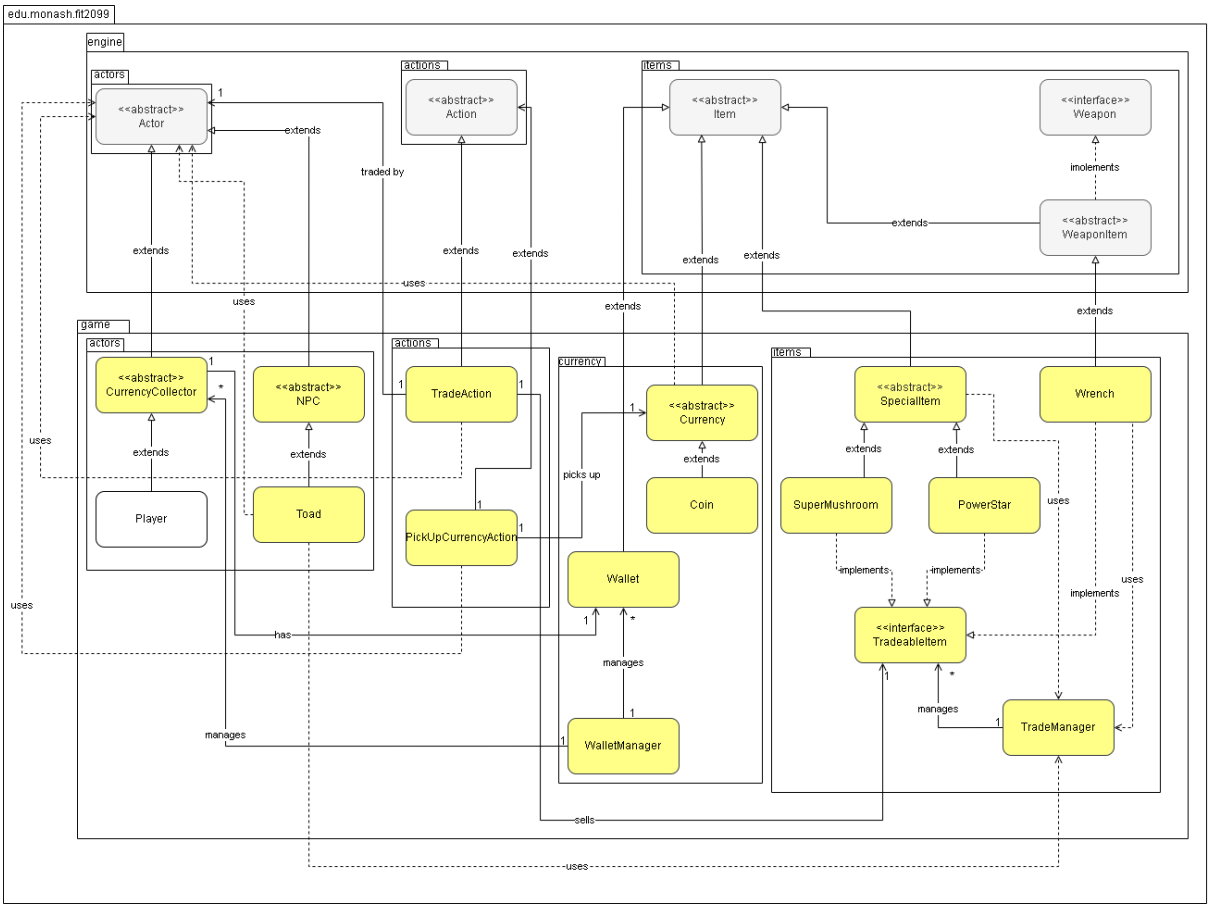
Removing Association Between Actor and ConsumeAction

The Actor consumer is no longer an instance variable in the ConsumeAction class.

The consumer actor is now taken as just a parameter to the menuDescription and execute methods. This parameter will be passed in by the Actor class consuming the item hence reducing connascence between the actor and consumeAction class .

Requirement 5

UML Diagram



Above is the UML Diagram for requirement 4. Any changes from the previous UML diagram is outlined below.

NPC class

An abstract class for all non hostile actors.

Responsibility:

1. Ensure non hostile actors can't attack or be attacked
2. Reduce repeated code for actors with similar characteristics such as having non-hostile actions

Although for now the NPC class is only inherited by Toad, this class allows for future extension if the need arises for more NPC-like actors. This adheres to the Open-Closed principle as well as the do not repeat yourself (DRY) principle by reducing code repeated by NPC actors.

Currency class
An abstract class for all currency.
Responsibility:

1. Have similar methods for all currency items along with common attributes such as value.
2. Reduces repetition for methods such as pickedUp() which is called whenever a currency item is picked up as well as the allowableActions which allows an Actor to pick up a currency item.

Similar to the the NPC class, this class allows for future extension in case the need arises for other currency items besides Coin which adheres to the Open-Closed principle while also reducing repeated code between multiple currency items which adheres to the do not repeat yourself (DRY) principle.

PickUpCurrencyAction class
An action performed by a CurrencyCollector to obtain a coin.
Responsibility:

1. Allows currency collectors to pick up currency off the ground
2. Updates the balance in the currency collector's wallet with the coin's value by calling currency's pickedUp method.

A specific pick up action is created for currency items separate from PickUpItemAction. This is because the functionality of a currency being picked up is different from picking up an item. The key differences are the PickUpCurrencyAction is performed by CurrencyCollectors and can only pick up Currency objects. An alternative approach would've been to extend PickUpItemAction class but this would've led to unexpected behaviour as the CurrencyCollector and Currency classes perform methods that aren't available in the Actor and Item class. This would violate Liskov's Substitution Principle as the PickUpCurrencyAction would behave differently in comparison to the PickUpAction class. Hence why the PickUpCurrencyAction is created.

WalletManager class
A class that manages all wallets and their owners.
Responsibility:

1. Has an ArrayList of all wallets and one for their respective owners
2. Allows us to check if a wallet or an owner of a wallet(currency collector) exists so we can safely cast an item as a wallet or an actor as a currency collector.

The wallet manager prevents any errors from occurring when the need arises to perform casting. Before casting an item or an actor to a wallet or currency collector, we check if their respective arraylists contain the item or actor.

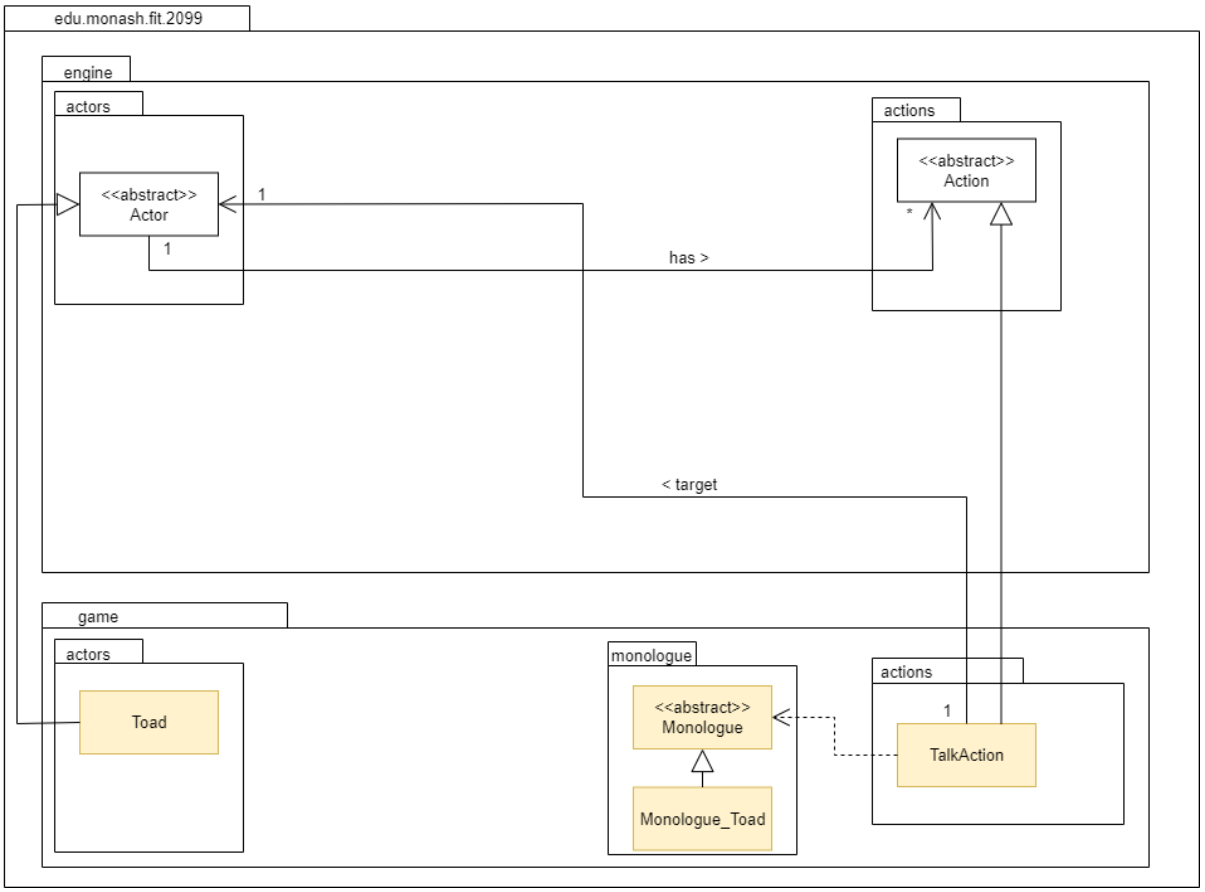
TradeManager class
A class that manages all tradeable items.
Responsibility:

1. Has an arraylist of all tradeable items, each tradeable item is added to the arraylist when they are instantiated in their respective constructors.
2. Manages all tradeable items

The TradeManager class is used by the Toad and prevents the toad from having any dependencies or associations with every single tradeable item.
This adheres to the dependency inversion principle where the tradeable items now only have a dependency with the trade manager which is then used by toad to get the items to be traded with other actors(currency collectors).

Requirement 6

UML Diagram



The above UML diagram is our improved version from Assignment 1. Design's rationale will be explained below

Abstract Class Monologue

Abstract class Monologue is a generic monologue class that will serve as the base class for all monologue related class going forward.
Responsibility:

1. This class provides the structure for all monologue classes going forward ensuring that other monologue related classes would follow a similar pattern.
2. This class will contain all of the dialogue for a certain character based on the monologue class created to alleviate the burden on the character itself (Single Responsibility Principle). This also allows for modification to be made to the actor's dialogue patterns without having to edit the code of the actor itself hence reducing the risk of messing up the character's code (Open-Closed Principle).

Class Monologue_Toad

Class Monologue_Toad is the monologue class created specifically for Toad, this class extends the Monologue abstract class to be able to adopt its structure.

This class contains 4 lines of dialogue to be randomly delivered to the player when the player talks to Toad. However, in the speak() method, several conditions are laid out to suppress some of the dialogue from being outputted.

In this class there are 4 statements:

1 : The Princess is depending on you! You are our only hope.
2 : Being imprisoned in these walls can drive a fungus crazy :(
3 : You might need a wrench to smash Koopa's hard shells.
4 : You better get back to finding the Power Stars.

However, statement 3 cannot be returned if the actor has a Wrench and statement 4 cannot be returned if the actor has consumed a Power Star

Hence, the speak() method will roll a random number from 0 to 3 to determine which of the statements will be returned else it would be rerolled.

This class has no additional constraints to cause it to differ from its parent class (Liskov's substitution principle) and this would be helpful during implementation of this class in the TalkAction class

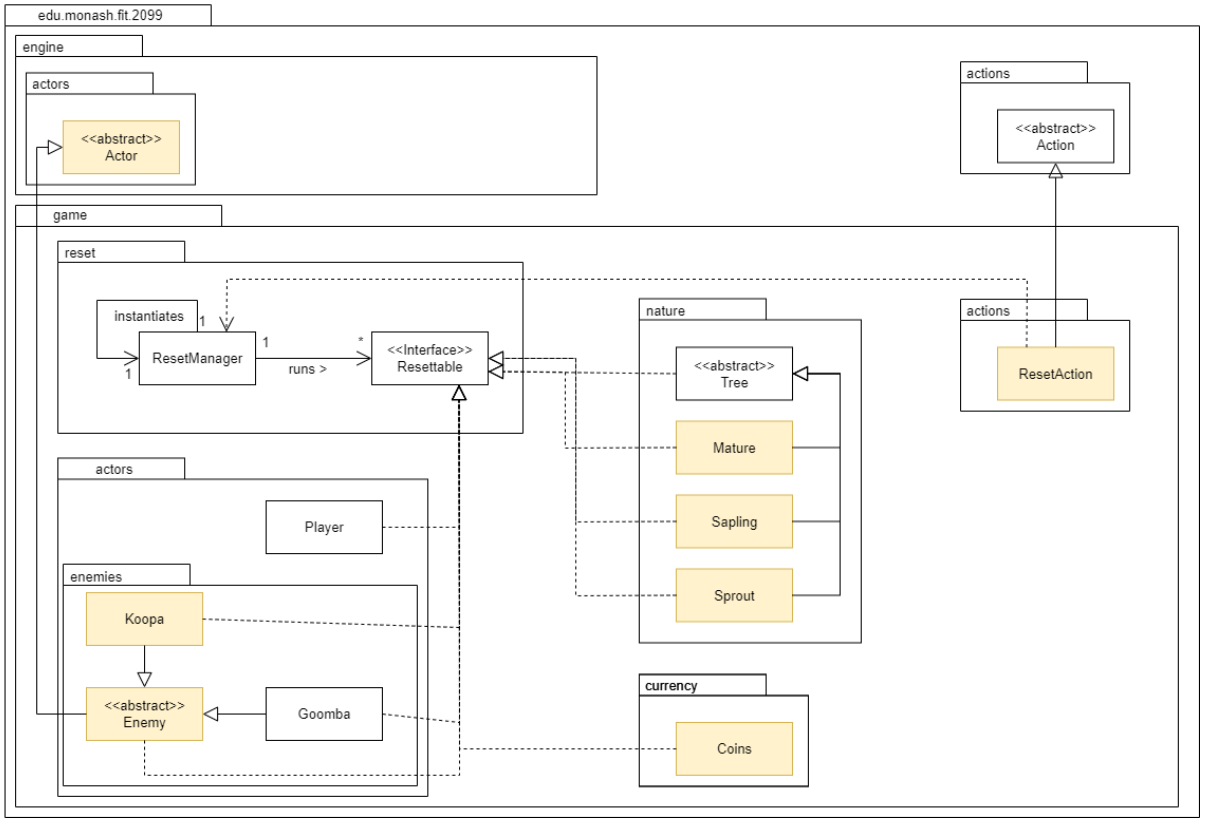
Class TalkAction

Class TalkAction is an action class that extends the Action abstract class. This class is meant to carry out the action of talking to another actor.

This class contains an attribute monologue that is of type Monologue. Since child classes for the Monologue abstract class should be substitutable for its parent class and likewise, after identification of the class which the talkAction is being directed at, a new instance of a Monologue child class would be created and the speak() method would be called to provide dialogue.

Requirement 7

UML Diagram



The above UML diagram is our improved version from Assignment 1. Design's rationale will be explained below

Main Changes:
The following classes and their child classes have implemented the Resettable interface:

- Enemy
    - Koopa
    - Goomba
- Coins
- Tree
    - Sprout
    - Sapling
    - Mature
- Player

The 4 main classes (Enemy, Coins, Tree and Player) would have used the registerInstance() method to register that particular class as well as any child classes as an instance of a resettable object. In addition, these 4 classes would have implemented the interface method, resetInstance(). In addition to that, all these 4 classes would have implemented the interface method, resetInstance() whereby a new status would be added to that object, RESET_QUEUED.

When this status is queued, a reset will be queued for that instance of object to be executed at the next tick or turn.

Class ResetAction

Class ResetAction is a simple class that extends the Action abstract class. This class acts as the medium to broadcast a reset to all instances that have registered itself as a resettable object

Implementation:

1. When the Player class is first instantiated, a new capability would be added to the list of statuses called RESET_AVAILABLE. While the Player maintains this status, the reset option will be made available for the user to activate by selecting the ResetAction on the menu which has “r” as its hotkey.
2. Upon the commencement of the ResetAction, the RESET_AVAILABLE status will be removed from the Player and no more option to reset the map would be made available for the user.
3. Upon the commencement of the ResetAction, all items registered as a resettable object would have their resetInstance() method ran and all of them would be given the status of RESET_QUEUED and the logic for the reset that would be planted in the tick() method or playTurn method would be carried out and the reset would truly occur.

Reset for Tree class and its child classes

For all instances of Tree and its children classes would roll a random number or either 0 or 1 (50%) to determine if that instance of Tree would be converted back into dirt.

Reset for Player class

When RESET_QUEUED is added, the player would be fully healed and INVINCIBLE status and SUPER status will be stripped from the player (if present).

Reset for Coins

During the reset, all coins would be removed from that location

Reset for Enemy and its child classes

All actors extending Enemy abstract class would be purged from the map and removed instantly, not killed but immediately removal.