2022

FIT2099 Assignment 2

LAB 3 TEAM 2

SHANTANU THILLAI RAJ, CHAI LI GUANG, EDELYN SEAH

Work Breakdown Agreement for FIT2099 Assignment 2

Team 2 Lab 3:

1. Chai Li Guang (31858988)
2. Edelyn Seah (31107559)
3. Shantanu Thillai Raj (32141580)

	Tasks	Person-in-charge	Date to be completed
1	Implementing requirement 1,2,5	Chai Li Guang	29/4/2022
2	Implementing requirement 2,4,7	Shantanu Thillai Raj	29/4/2022
3	Implementing requirement 2,3,6	Edelyn Seah	29/4/2022
4	Testing implementation of requirement 2,3,6	Chai Li Guang	30/4/2022
5	Testing implementation of requirement 1,2,5	Shantanu Thillai Raj	30/4/2022
6	Testing implementation of requirement 2,4,7	Edelyn Seah	30/4/2022

Chai Li Guang will be responsible for the above tasks. Reviewer: Edelyn Seah, Tester: Shantanu.

Signed by: ***Chai Li Guang***

Edelyn Seah will be responsible for the above tasks. Reviewer: Shantanu, Tester: Li Guang

Signed by: ***Edelyn Seah***

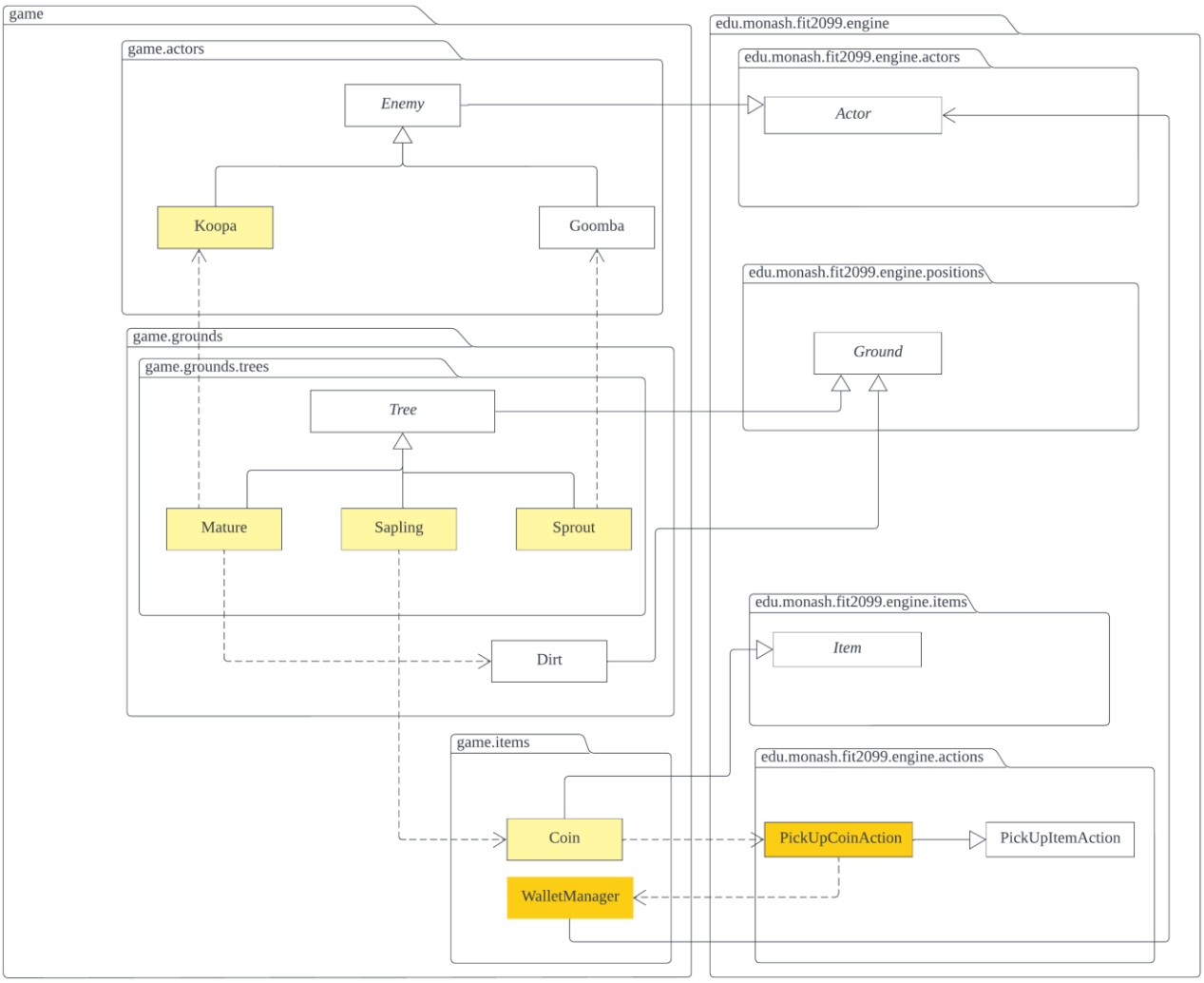
Shantanu will be responsible for the above tasks. Reviewer: Chai Li Guang, Tester: Edelyn Seah

Signed by: ***Shantanu Thillai Raj***

Changes of Document from Assignment 1 to Assignment 2

- New classes added to the implementation are coloured orange in the UML diagram.
- Rationale updated to include more SOLID principles and to reflet new changed for implementation.
- ONLY sequence diagram for requirement 7 added here.

Requirement 1: Let It Grow!



Design Rationale

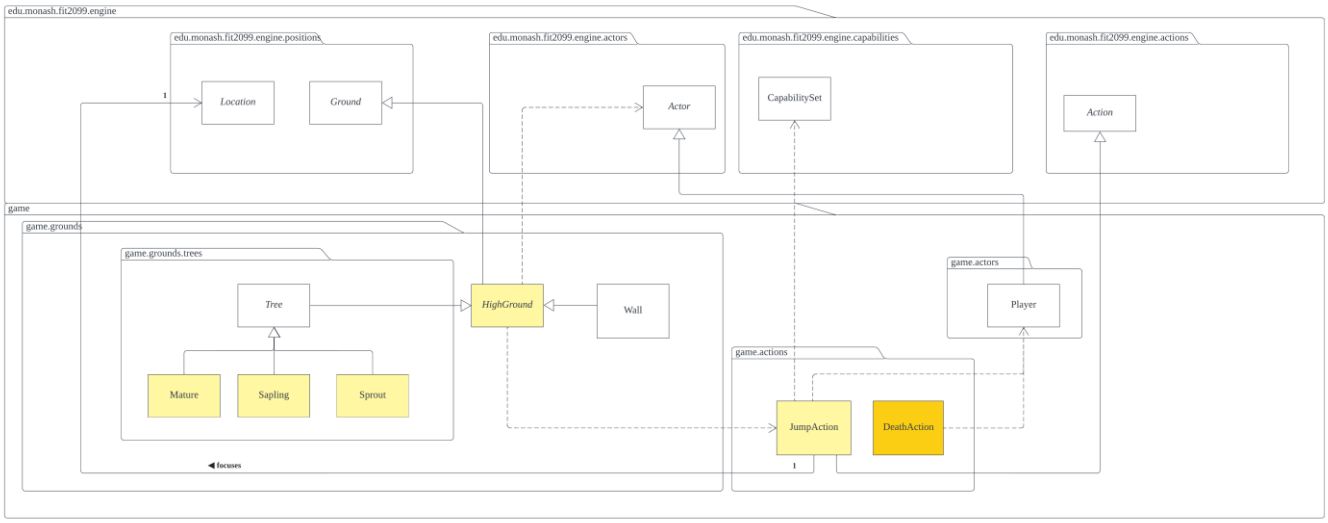
The tree class was extended from the abstract class Ground. The reason behind it was to achieve reusability as much as possible in our code. Since we have been provided with the abstract class Ground, we can reuse the methods like tick() to build our tree class. In addition, we make the tree an abstract 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 all trees have a 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 must 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). However, to avoid overlapping relationship lines, we decided not to include the circulation dependency as dependency relationships are optional to show on class diagram.

With the tree abstract class method spawn(), we applied the Open/Closed principle where the tree abstract class is opened for extension (implementing the spawn() in the sub classes of tree) and closed for modification (did not change any code in the tree abstract class).

Since Sapling spawns Coin, we created a Coin class which extends from Item class. We also added a few new classes, PickUpCoinAction and WalletManager to handle the money system. The PickUpCoinAction applies the Liskov principle as it is a subclass of PickUpItemAction, and when overriding getPickUpAction() to return PickUpCoinAction in the Coin class, the program still works fine even though the required return type of the method is PickUpItemAction. Creating PickUpCoinAction class is also applying Single Responsibility Principle as we don't have to cramp everything in player class(we previously did it in the player class). We used the Singleton WalletManager to store the wallet balance of each actor(currently only player has wallet). The advantage of using the singleton manager is since it only has one instance, this allows all the actors that has wallet in the future to only refer to this one instance. WalletManager has an association relationship with actor as it has a hashmap in which the key is the actor object and value being the walletbalance.

Requirement 2: Jump Up, Super Star!



Design Rationale

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 cannot 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 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 must check if the player has any super mushroom buff activated so it has a relationship with CapabilitySet.

Addition for A2:

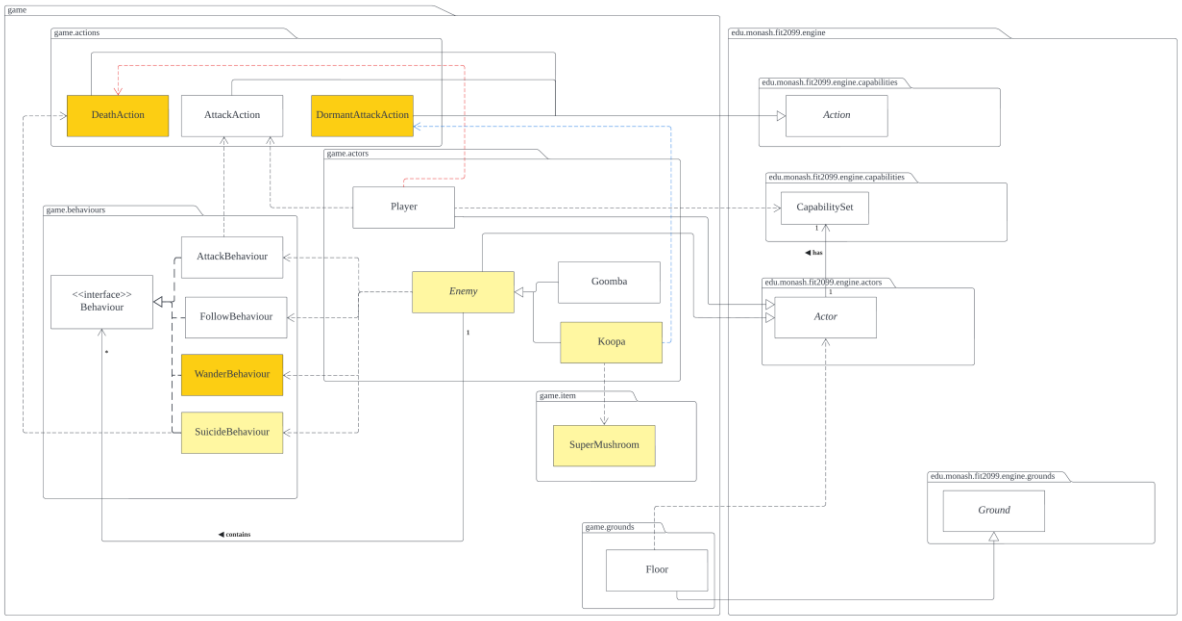
If actor dies from Jump Action damage, adds status DEAD to actors' capability set. On actors PlayTurn, when DEAD Status is seen: returns new DeathAction() which drops all actors item from inventory and removes actor from the map.

Solid Principles Applied

Single Responsibility Principle

- JumpAction only does the jump action for the actor between it and a highground
- DeathAction only does a death (removes actor from the map and drops the actor inventory items onto the ground)

Requirement 3: Enemies



Design Rationale

How it works?

As we know, the Player, Goomba and Koopa extends from the abstract Actor class. The abstract Actor class follows the Dependency Inversion Principle. 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 follows the Open-Closed Principle because there is room for extension for other future enemies but to prevent further modification of the present enemy classes.

The AttackBehaviour, FollowBehaviour, WanderBehaviour and SuicideBehaviour (self-made) classes implement interface Behaviour to enable modularization since the behaviours can be reused for multiple actors. This follows the Interface Segregation principle; actors should not be forced to depend upon interfaces that they do not use.

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.

DormantAttackAction extends Action abstract class in the engine and Single-Responsibility Principle: Special action for when attacker has a wrench and the Koopa is in a dormant state.

DeathAction extends Action abstract class in the engine. It removed the DEAD actor from the map. (hasCapability(DEAD)) and drops all the items from the actors inventory. Useful for Koopa where it has to drop SuperMushroom on death.

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

SOLID Principles Applied

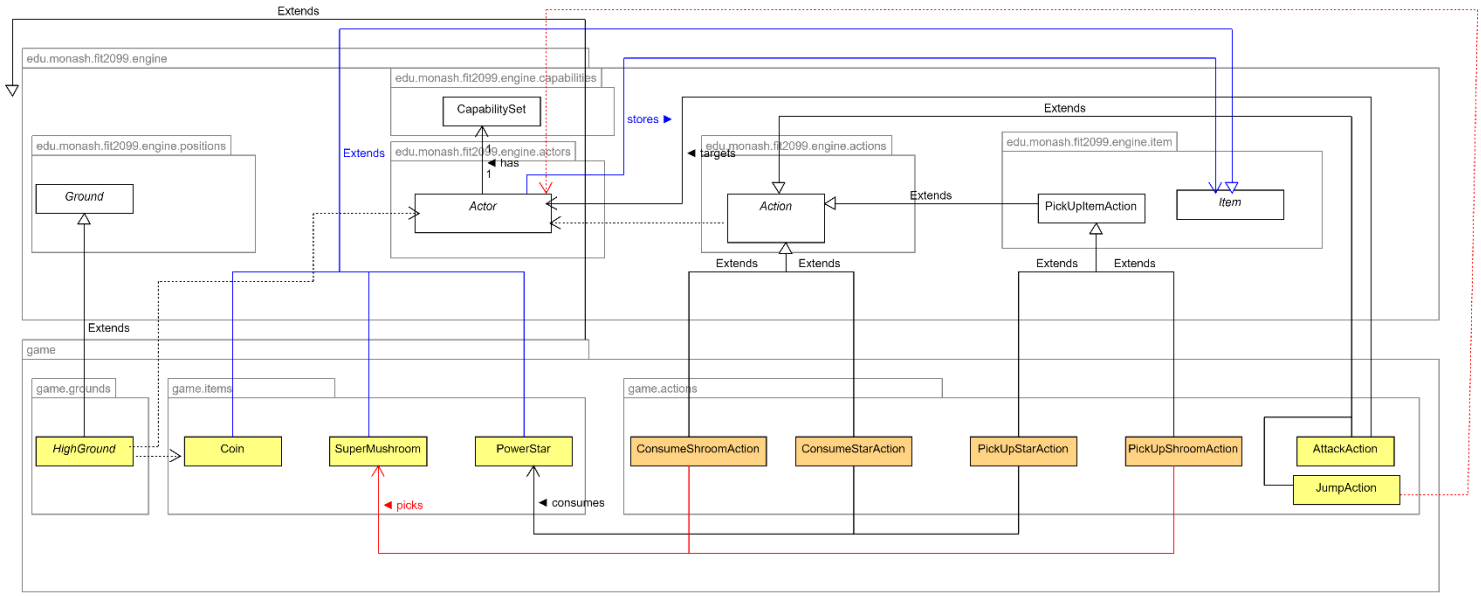
Open-Closed Principle

- This follows the Open-Closed Principle because there is room for extension for other future enemies but to prevent further modification of the present enemy classes.

Interface Segregation Principle

- Each interface serves a single purpose and actors do not have to depend on it if they are not using it.

Requirement 4: Magical Items



Design Rationale

How it works?

- Power Star and Super Mushroom extend from the Item Abstract Class in the engine.
- Each have their own specialized pick-up action and consume action created:
    - PickUpStarAction, ConsumeStarAction for Power Star
    - PickUpShroomAction, ConsumeShroomAction for Super Mushroom
- When a Power Star/Super Mushroom is picked up (PickUpStarAction/PickUpShroomAction execute occurs), a consume action (ConsumeStarAction/ConsumeShroomAction execute) instantly occurs therefore allowing the player to be instantly buffed the moment they pick up the items.
- For the Power Star, the buff timer and item life span timer is done in the Power Star class itself using the tick method to countdown.
- The consume actions add their specific buff to the players capability set (TALL for Super Mushroom and INVINCIBLE for Power Star) alongside doing the other buffs such as healing and increasing max HP.
- If item is bought from toad, a new consume action for that specific item is provided for player.
- The effect of these items on the JumpAction is handled in both the
    - HighGround abstract class:
        - If the player has status INVINCIBLE, actor can enter and the ground is destroyed (replaced with dirt, coin added when actor is on that location)
    - JumpAction class:
        - If the player has status TALL, actor has 100% success rate on the jump.
- The effect of these 2 items on attacks is handled in
    - AttackAction class:
        - Checks if the attacker/target have the statuses and does the corresponding correct actions based on spec. E.g.: If target has INVINCIBLE, damage of attacker set to 0, returns a specific string to say the target is invincible.

SOLID Principles Applied

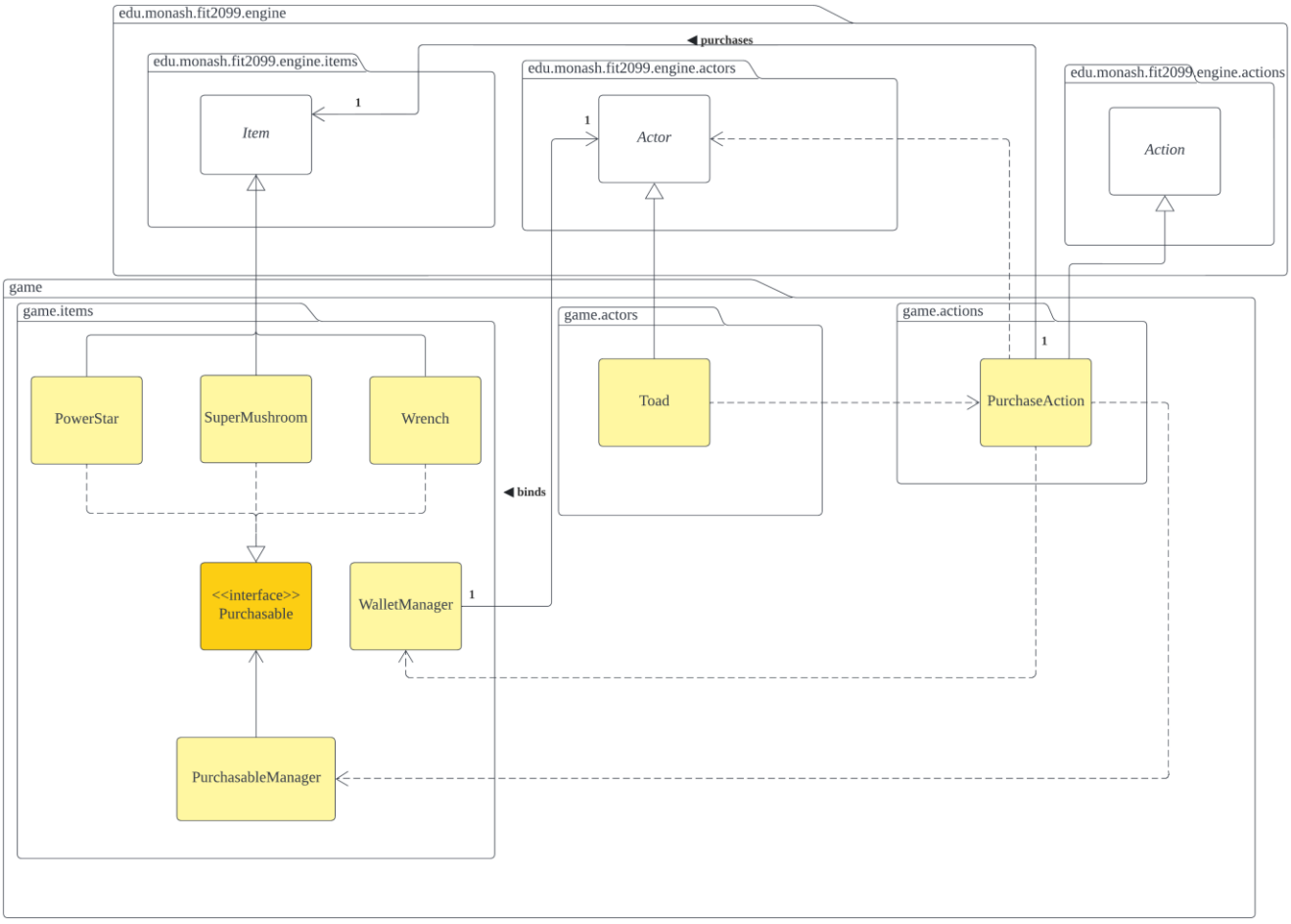
Single Responsibility Principle

- Instead of having a single Consume action as we had designed in assignment 1, we have 1 for each item, this way we can avoid the use of if-else and down casting to check the item type, and each consume action (ConsumeStarAction/ConsumeShroomAction) is only responsible for actor consuming their specific item.
- Similarly, having a pickup action for each item (PickUpStarAction/PickUpShroomAction) to avoid if-else, down casting and each pick up action is only for picking up the specific item they are for.

Liskov Substitution Principle

- The Power Star and Super Mushroom extend the Item abstract class and do not add any special methods therefore can be replaced without disrupting the system.
- Avoid redundancy by just transferring the capability of the item into the actor on consumption.

Requirement 5: Trading



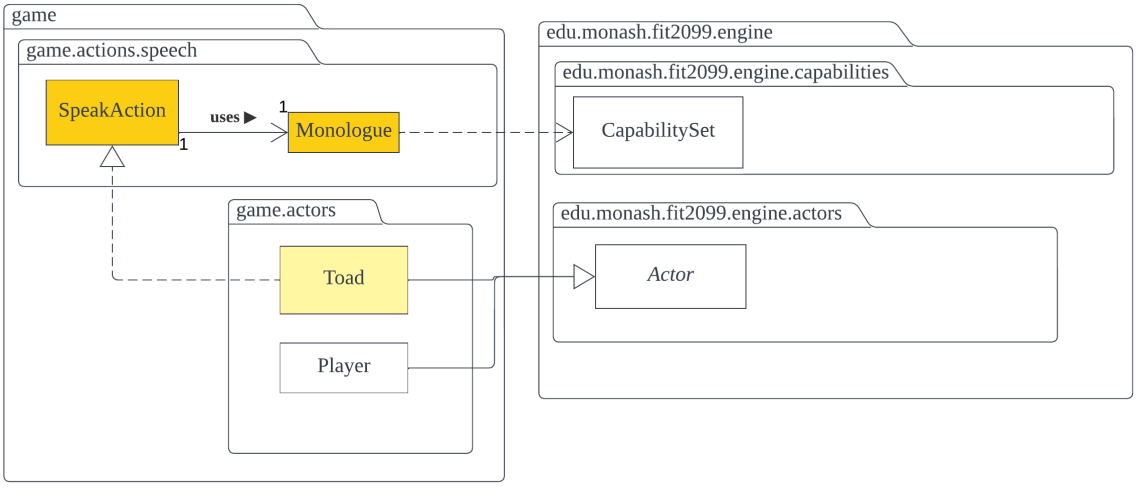
Design Rationale

For the trading interaction between the player and toad to happen, the toad must be 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. We will be adding 3 purchaseActions as currently there are 3 purchasable items (Wrench, SuperMushroom and Power star). Since purchaseAction will take in the mentioned item object, it will have a dependency relationship with the purchasable items. We chose not to show it on the diagram as it will cause overlapping lines and become very messy.

We created a Purchasable interface so that all purchasable items can implement it and completes the price(), which it will just return the price of the item. Then, we have a PurchasableManager that manages the purchasableItems so that it can prevent the code smell of downcasting/ using instanceOf(), as not all items are purchasable (in the future), by having an arraylist of purchasableItem objects. It also complies with the Single Responsibility Principle as PurchasableManager only handles storing all the purchasable items and make it accessible to other classes that call creates this PurchasableManager instance.

Moving on, the PurchaseAction will create an PurchasableManager instance to access the purchasableItemList so that it can loop through it and find the matching item, then have access to the item price (down casting is prevented this way). Then it 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, and minus the balance if the actor has sufficient money, this explains the relationships between PurchaseAction and WalletManager.

Requirement 6: Monologue



Design Rationale

How it works?

We extend 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.

We created a SpeakAction which serves the purpose of displaying the given monologues from the actor. This specific class follows Single-responsiblity Principle because the class only has one function.

The Monologue class is where we do the related checking according to what is in the inventory of the Player class - therefore it is a dependency.

SOLID Principles Applied

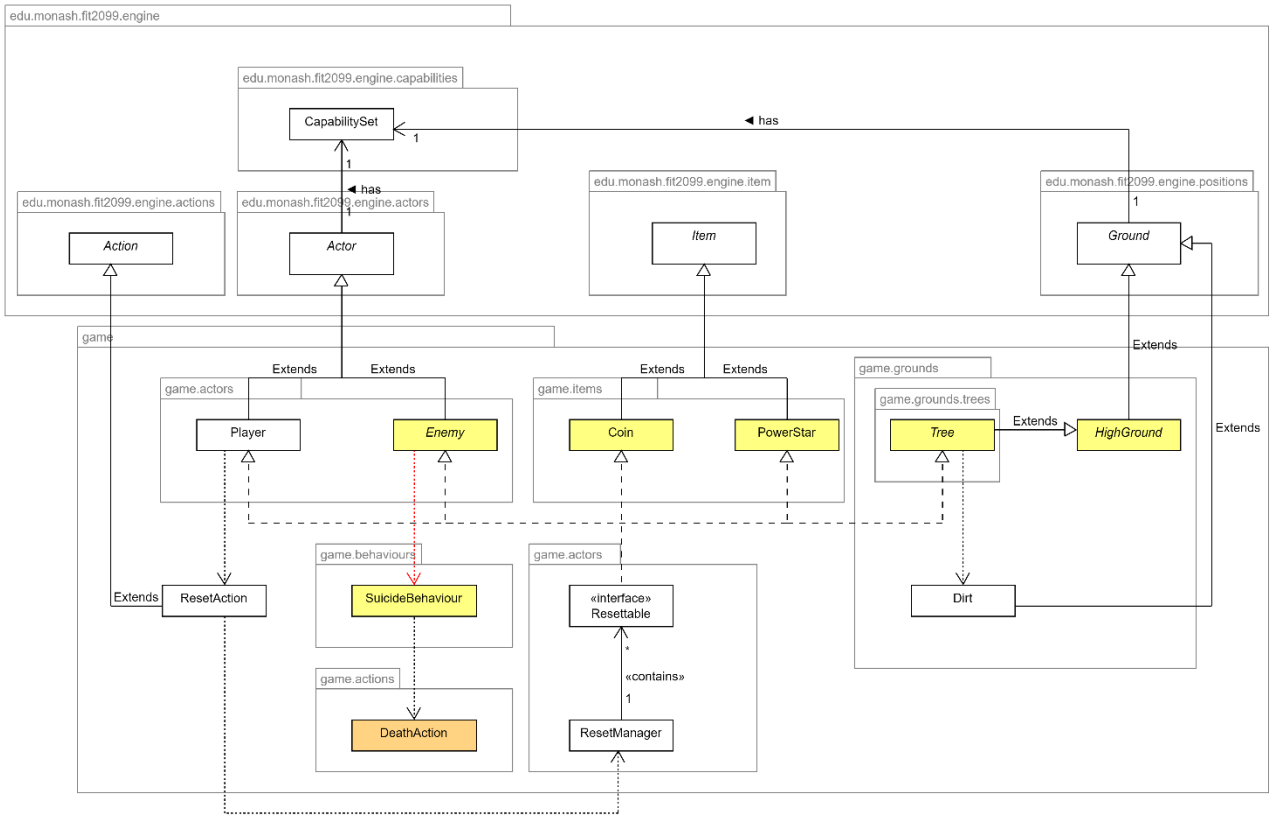
Liskov substitution principle

- Player class will inherit all the methods from the Actor class.

Single-responsibility Principle

- SpeakAction is only used to do the speaking (monologue printing)
- Monologue classes is only used to store the monologues and return the allowed monologues given conditions to the SpeakAction when it occurs.

Requirement 7: Reset Game



Design Rationale

How it works?

- All resettable classes implement the Resettable interface.
- The ResetManager stores each of these instances that must be reset.
- If the resettable item/actor/ground was removed from the map for other reasons outside of reset, using the cleanUp() method on it to remove that instance from the ResetManager.
- When a reset occurs, all the objects to be removed from the map are given the Status REMOVED.
- Before the start of next turn when the tick method, play turn methods run, the SuicideBehaviour and tick methods will remove all the enemies/items/grounds that should be removed (a.k.a has the REMOVED Status)
- Using of extra utility class Probability to do the probability for removal of trees.
- When a reset occurs, Player is given capability Status.RESET, when player has this status the reset action will no longer be available to the player.

SOLID Principles Applied

Single Responsibility 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 interface instead of each tree (sprout, sapling, mature) and each enemy (Goomba and Koopa), allows us to avoid repeating the implementation of resetInstance() in every one of those subclasses.

Liskov Substitution 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 down casting/using the taboo instanceOf and getClass() methods as we know that any object which 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.

Sequence Diagram for Requirement 7 Reset Action

- Updated to add power star and death action to reflect actual implementation.