Assignment 2

Work Breakdown Agreement

Lab 12 Team 4
Group Member:
Seng Wei Han (32229070)
Chua Jun Jie (32022425)

Work Distribution

Tasks	Person In Charge	Status
Design Req 1, 2, 5	Seng Wei Han	✓
Design Req 3, 4, 7	Chua Jun Jie	✓
Review each other's diagram	Chua Jun Jie/Seng Wei Han	✓
Design Rationale	Chua Jun Jie/ Seng Wei Han	✓

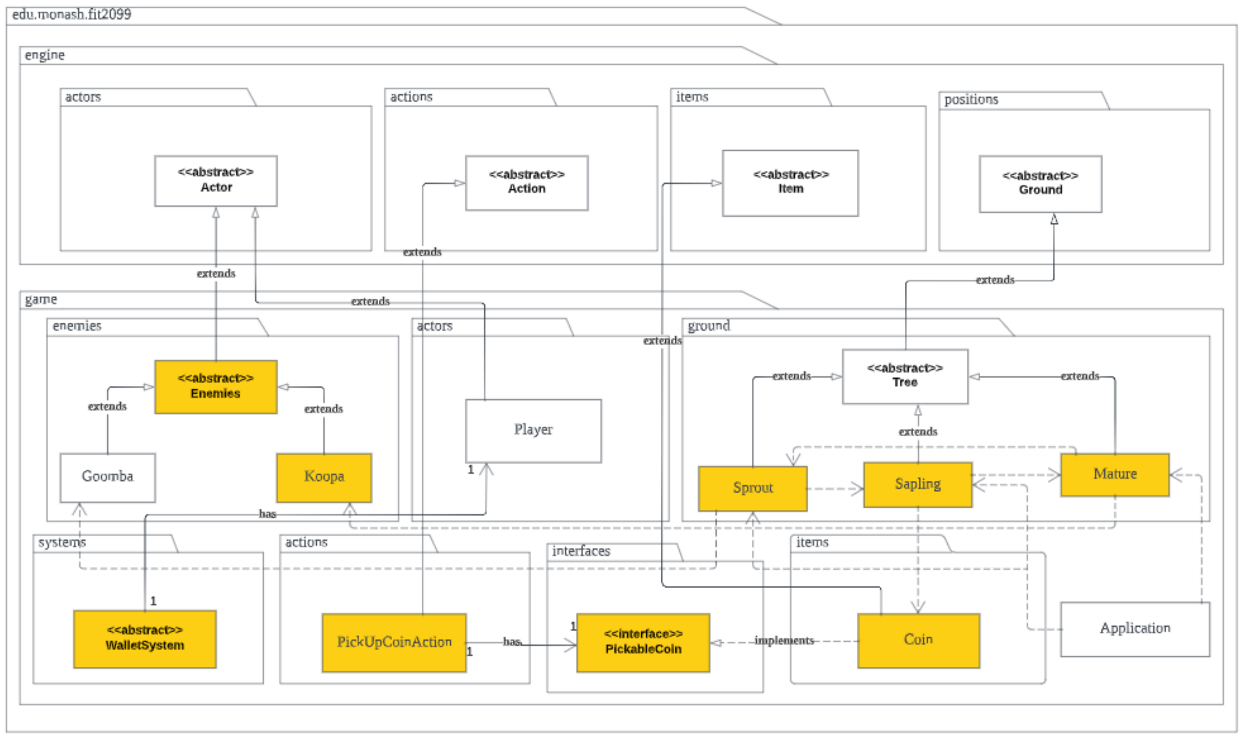
I (Chua Jun Jie) accept this WBA.

I (Seng Wei Han) accept this WBA.

Requirements

REQ1: Let it grow! 🌳

REQUIREMENT 1 CLASS DIAGRAM



(The screenshot shown above is the UML designed for requirement 1)

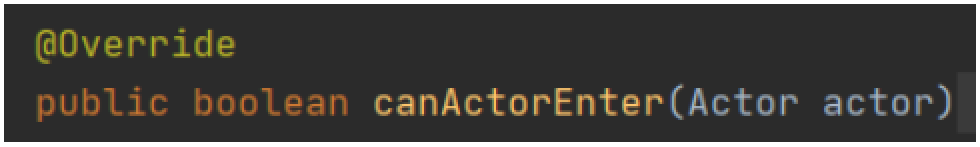
Implementation

Tree Class

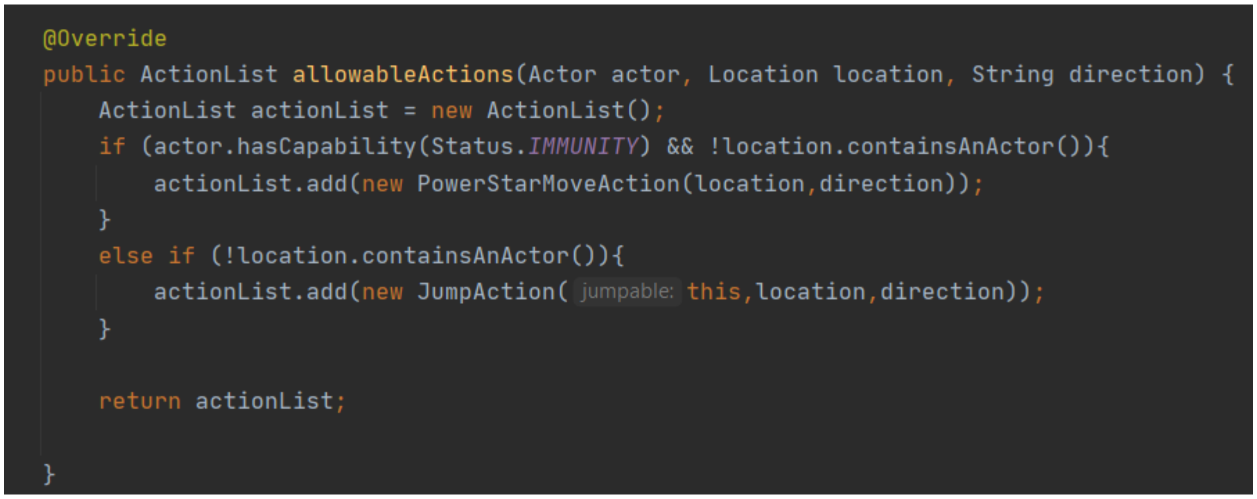
1. The purpose of making the Tree class to be an abstract class is because we knew that we will not instantiate the Tree class in any of our code. Such implementation helped us to achieve abstraction.

2. From requirement 1 of the assignment, it is stated that a Tree has three stages and each stage has a unique spawning ability. On the other hand, each spawning stage will also have additional characteristics, for instance each sprout has 10% chance to spawn Goomba. Therefore, instead of doing all the spawning in one Tree class which will cause the Tree class to be so called the God class, I have decided to split the classes into 3 small new classes namely Sprout , Sapling and Mature and make it inherit the Tree class. In each of these new subclasses, it will have their own responsibility so that this design will adhere to the Single Responsibility Principle (SRP).

3. Since sprout, sapling and mature are the 3 stages that a tree can experience, meaning to say the 3 classes will have the same characteristics for instance, having the same method implementation for the following method:



```java
@Override
public boolean canActorEnter(Actor actor)
```



```java
@Override
public ActionList allowableActions(Actor actor, Location location, String direction) {
    ActionList actionList = new ActionList();
    if (actor.hasCapability(Status.IMMUNITY) && !location.containsAnActor()){
        actionList.add(new PowerStarMoveAction(location,direction));
    }
    else if (!location.containsAnActor()){
        actionList.add(new JumpAction( jumpable: this,location,direction));
    }

    return actionList;

}
```

Thus, it's a better idea to implement those method in the Tree class and allow the other 3 new classes (Sprout, Sapling and Mature) to inherit the method from their parent class. By using this implementation, we can ensure that DRY(Don't Repeat Yourself) Principle will not be violated.

Sprout, Sapling and Mature class

1. The spawning capability of each of these classes can be done by using a tick method together with a counter. World will loop through every single ground and call their tick method if the instance of that ground exists in the gameMap. Therefore, each time a tick method is called, the counter will be incremented by one. Once the counter is equal to 10, meaning that the current instance of ground whether it is sprout, sapling or mature will grow into its next stages.

Enemies class

1. The purpose of declaring a new enemies abstract class and allow both Goomba and Koopa classes to extends the enemies class is to provide an abstraction so that

other concrete class will not directly depends on the Goomba or Koopa concrete class instead it should depend on the abstraction which in this case is the enemies class. By doing so, we are adhering to the Dependency Inversion Principle.

PickUpCoinAction

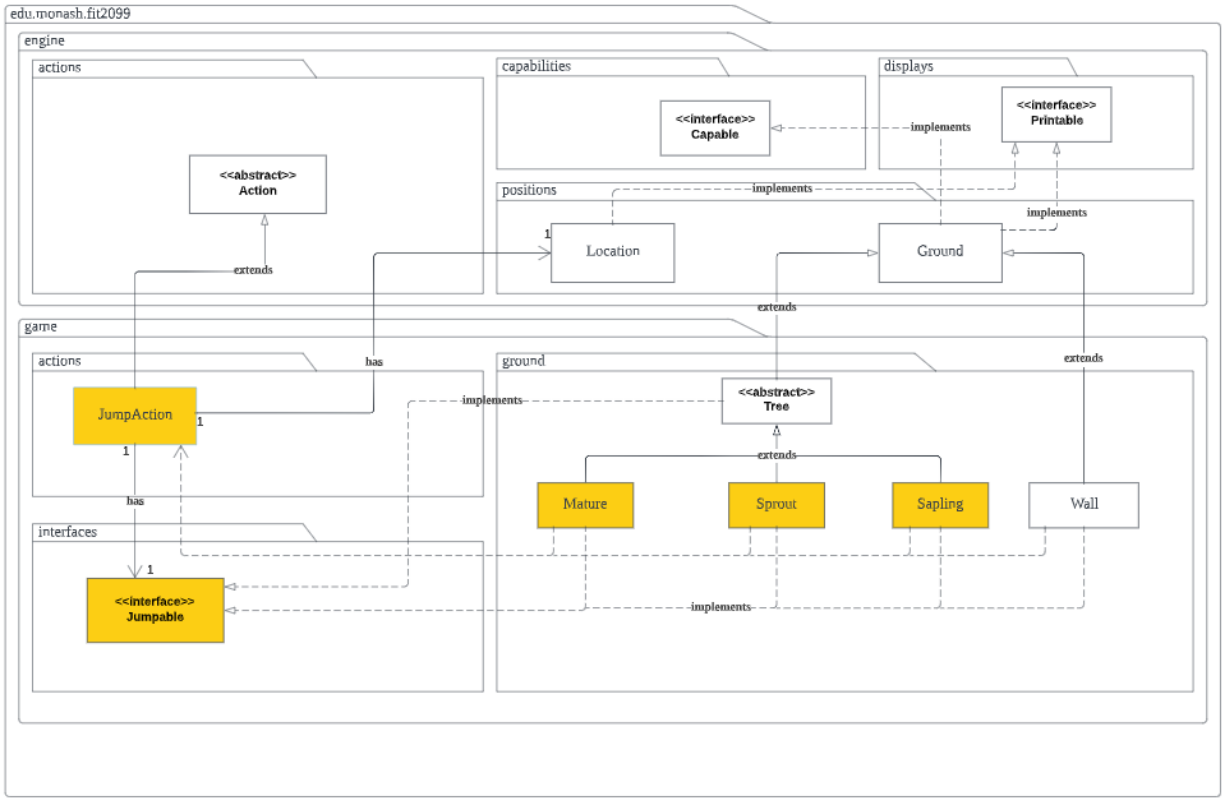
1. An action that a player can take when encountering a coin on the ground.
2. I have decided to create this PickUpCoinAction class instead of using the default PickUpItemAction implemented in the engine class is due to the fact that the PickUpItemAction in engine will basically stores the item that is picked up into the player’s inventory. However, this is not the case for coin items. Instead of storing them into the inventory , I will add the values of the coin into the player’s wallet balance.
3. PickUpCoinAction has direct association with PickableCoin interface instead of Item abstract class is to prevent the usage of downcasting from Item to Coin in order to obtain the coin value to be added into the player’s wallet and as we know that downcasting is a form of code smell which is inefficient and may cause bugs.

WalletSystem

1. The purpose of creating a WalletSystem class is to allow us to keep track of the player’s balance as well as allow us to add to the player's balance when the player collects some coins from the ground which is spawned from sapling.
2. This class is declared as abstract because we will not create any instance of type WalletSystem. This helps to achieve abstraction.

REQ2: Jump Up, Super Star! 🌟

REQUIREMENT 2 CLASS DIAGRAM



(The screenshot shown above is the UML designed for requirement 2)

Implementation

Jumpable

1. An interface which contains a jump method which must be implemented by any ground which implements this jumpable interface except for Tree abstract class. Those grounds which implement this interface are those which do not allow actors to enter unless they use a jump action.

JumpAction

1. Since requirement 2 is about jumping features, then we definitely need to have a JumpAction class implemented. This JumpAction extends Action class and has direct association with Jumpable interface. By having this association, we can prevent the usage of if-else statement in JumpAction class for the purpose of checking whether the current ground is of type either Mature, Sprout, Sapling or Wall because different ground have different success rate and also fall damage.

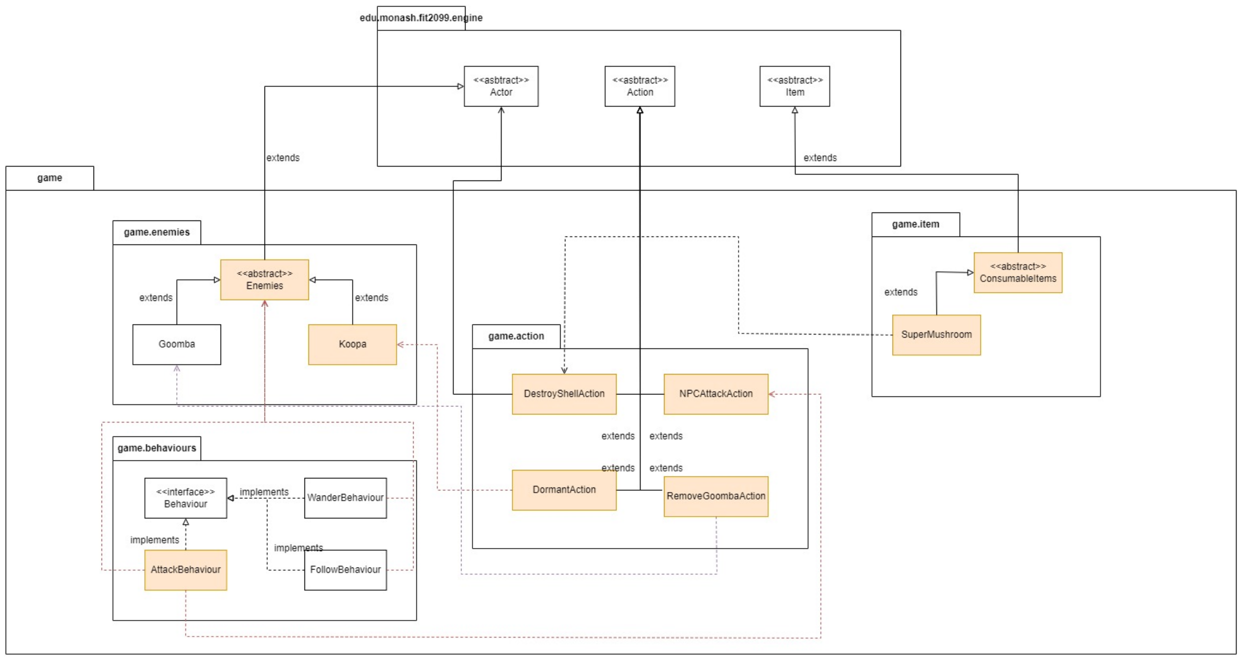
The usage of too many if-else statements in one class can be bad because if there’s a new ground which can be jumped, in order to add this new functionality for this new ground, we will need to modify the existing code and such approach clearly violated Open-Closed Principle.

2. This class also adheres to the Single-Responsibility Principle because it is only responsible for the jumping action of the player and nothing else.

Sprout, Sapling, Mature and Wall

1. These four classes are those which do not allow the player to enter unless the player uses a jump action. Therefore, they will need to implement the Jumpable’s jump method in their class.Each of them will have different implementation of jump method due to different success rate and fall damage as shown below:
2. For the requirement that stated player will have 100% success rate and no fall damage if the player consumes SuperMushroom, we can just do a simple check in the jump method to check whether the player has the EFFECT_SUPER_MUSHROOM status. If they do, then they can just jump without having any fall damage.

REQ3: Enemies



(The screenshot shown above is the UML designed for requirement 3)

Implementation

Enemies — Abstract Class

1. Enemies is an abstract class that extends Actor abstract class. It is created to group the enemies together, and also creating an Enemies abstract class is essentially using Single Responsibility Principle (SRP). Goomba and Koopa extends Enemies abstract class. By creating this class, it allows us to combine the code that both Goomba and Koopa has, simplifying the coding process and makes the code cleaner.

Goomba ('G')

1. The Goomba class is given already, thus we don't need to create another class just for the character. So, at first, Goomba will extend Enemies abstract class, and its basic stats will be declared in the constructor. This uses mainly Open-closed Principle (OCP) as whatever changes Goomba class has made, it will not affect the other classes that extends Enemies as well.
2. Goomba attacks deal 10 damage with a kick, and has 50% chance to hit. This is achieved by overriding getIntrinsicWeapon() method, putting the damage as "10", and the verb as "kicks".

3. For every turn, Goomba has 10% chance to be removed from the map. Thus, a method called “remove()” that returns a boolean is created to remove Goomba if Goomba hits the 10% chance. “remove()” method is placed in Goomba’s playTurn method so that it will be called every turn of the game.

RemoveGoombaAction

1. RemoveGoombaAction class is created to remove Goomba through an action. This is called in the playTurn method inside Goomba, after passing the “remove()” method check. As the name goes, it is to remove goomba from the map. Basically, if removed() returns a true, playTurn method will return a RemoveGoombaAction, where the current Goomba object will be removed from the game.

Koopa (’K’)

1. Since there is no given Koopa class before, we shall create its own dedicated class. Similar to Goomba, Koopa will extend Enemies abstract class, and its stats will be declared in the constructor. Similar to Goomba, this uses mainly Open-closed Principle (OCP) as whatever changes Koopa class has made, it will not affect the other classes that extends Enemies as well.
2. Similar to Goomba, Koopa attacks deals 30 damage on successful hit with a punch, and it has 50% chance to hit as well. Thus, the same method getIntrinsicWeapon() is overrode , putting damage as “30” and verb as “punches”.
3. Koopa will become dormant if it’s HP falls below 0. When its HP is 0, its display character should become ‘D’ instead of ‘K’. This can be set inside Koopa’s playTurn method, by checking if Koopa has the “DORMANT” capability or no. If yes, its display character is set to ‘D’.
4. If Koopa is in dormant state, it will return a new action called DormantAction. This action simply displays a message that says “Koopa is in dormant state” in the console. Koopa is unable to move or attack during dormant state.

Wrench (’w’)

1. Wrench is a weapon that deals 50 damage and with a 80% hit chance. The Wrench class is a subclass of WeaponItem. Its stats is declared in the constructor. This also uses Open-closed Principle (OCP) and Single Responsibility Principle as it allows

extension for future similar wrench class to be made, while also being specific that Wrench's class responsibility is to create a wrench in the game.

2. Player can only have the option to defeat Koopa if player has a wrench inside his inventory. If there is no wrench inside player's inventory, there will not be an option for player to hit Koopa that is in dormant state.

Dormant Status

1. The "DORMANT" status is placed inside the Status enum class. This status can be reused to prevent excessive use of literals.

Enemies Behaviour — How it works

1. Since Enemies abstract class is only for NPCs that will attack the player, we would use AttackBehaviour, WanderBehaviour and FollowBehaviour (hereafter, AB, WB, and FB) to make Enemies subclasses do certain actions naturally. The three behaviours mentioned is used inside Enemies playTurn method. Thus, Goomba and Koopa naturally uses the coded playTurn inside Enemies abstract class.
2. All the behaviours are stored in a TreeMap with an integer as key. WB is stored with a key value of 10, and this storing action is coded inside the constructor of the Enemies abstract class so that all subclasses will have a WB preset in their TreeMap, allowing the enemies created to wander around the map automatically.
3. The playTurn method contains a for-loop that checks the exits of the current object, if there contains a player in one of the exits of the enemy. Then, it runs a method to check if the enemy lands a successful attack, the method (successAttack()) returns a boolean, so if true, a new AB will be stored inside the TreeMap, with a key value of 0. This allows the enemy to attack the player automatically. If the method returns false, the enemy will continue to wander.
4. Assuming the enemy successfully attacked the player, then, the "attacked" attribute will set to true. This "attacked" attribute is then used to check the upcoming turns the player make.
5. If the player moves away from the enemy, and "attacked" is true, that means the enemy has attacked the player already, and FB will be stored inside the TreeMap with the key value of 5. This makes the enemy to essentially follow the player around the map, until the enemy is defeated.

6. If the player still remains inside the exits of the enemy, then it will run an if-statement to check if the TreeMap contains FB with key value of 5, if yes, that means the enemy has been following the player around, and has successfully caught up to the player. Then, it will remove FB from the TreeMap. If the enemy passes the "successAttack()" check, then a new AB with key value 0 will be stored.
7. The reason why we use TreeMap instead of HashMap is because TreeMap will sort the keys by ascending order. And since our action takes the first behaviour, it means if we have AB inside TreeMap, it will always prioritise AB, then FB, then WB.

WanderBehaviour

1. WanderBehaviour is also a given class that allow the Enemies to wander randomly around the map until they encounter player.

FollowBehaviour

1. FollowBehaviour is a given class that allows the Enemies to follow the player around the map.

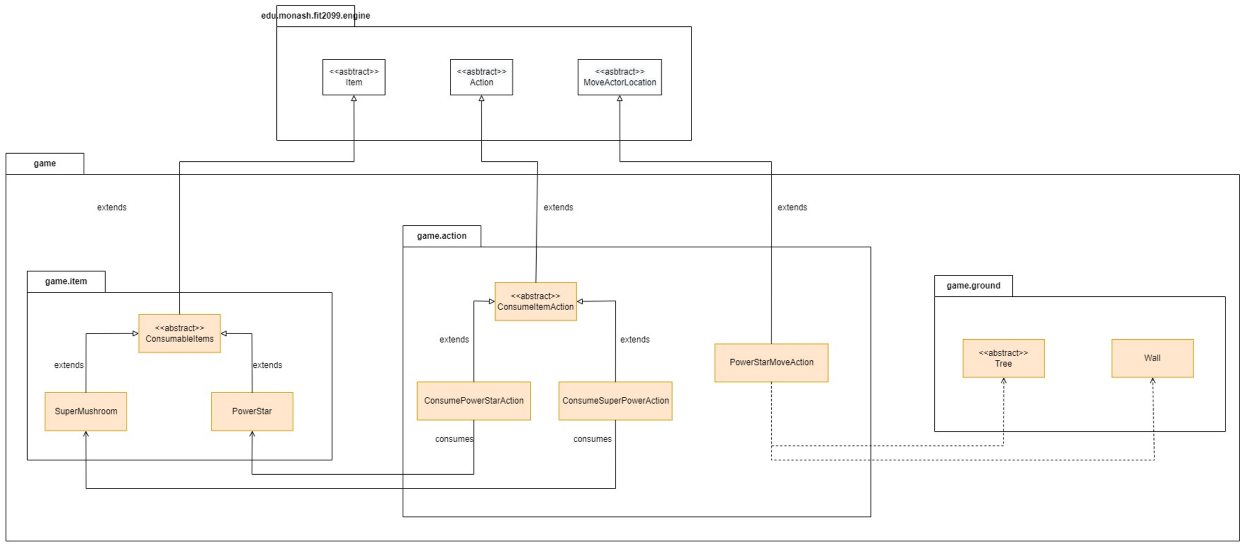
AttackBehaviour

1. AttackBehaviour is used so that the Enemies subclasses can generate a NPCAttackAction to attack the player.

NPCAttackAction

1. NPCAttackAction class is similar to the AttackAction class, but this class is solely for the Enemies subclasses to generate an AttackAction towards the player. Simply put, player uses AttackAction to attack Enemies, Enemies uses NPCAttackAction to attack players.

REQ4: Magical Items



(The screenshot shown above is the UML designed for requirement 4)

Implementation

ConsumableItems

1. Before creating SuperMushroom and PowerStar classes, we first create an abstract class called ConsumableItems. This is to indicate that the subclasses of the class can be consumed by the player. ConsumableItems also extends Item abstract class as it counts as an Item the player can obtain.

2. Creating an ConsumableItems abstract class is essentially using Single Responsibility Principle (SRP). This is to allows the classes extending ConsumableItems to have the identity of “this item can be consumed”. Furthermore, creating an abstract superclass allows us to group codes that exist in all its subclasses, and putting the code in the abstract class.

3. Inside ConsumableItems abstract class, the method getDropAction() is overrode and it returns null. This is because Magical Items can only be picked up but cannot be dropped. Other than that, an abstract method called consumeItem() that takes in an actor as parameter is created, allowing subclasses that extends ConsumableItems to override this method and use it accordingly.

Super Mushroom (^)

1. In order to implement Super Mushroom as a Magical Item into the game, a class called SuperMushroom is created, and it is a subclass of the ConsumableItems

abstract class, as shown below. Extending ConsumableItems utitlises Open-Closed Principle (OCP).

2. Inside SuperMushroom class, its basic information like name, display character and such are declared in the constructor. Meanwhile, a new action called ConsumeSuperMushroomAction is added inside the constructor as well. This is so that the super mushroom objects created will always have a consume action available.

3. The consumeItem() method is overrode here, and when this method is called, the actor will increase its max HP by 50, fully heals the actor, and also add a status effect called EFFECT_SUPER_MUSHROOM to the actor.

4. Since the SuperMushroom effect lasts for an indefinite time until the player receives damage from enemies, we can utilize the given enum class Status and use it to set the status of the player. This is to prevent using excessive literals in the code.

Power Star (*)

1. In order to implement Power Star class, we would do exactly the same as to how we implemented Super Mushroom, by first extending the ConsnumableItems abstract class. Similar to Super Mushroom, extending ConsumableItems utitlises Open-Closed Principle (OCP).

2. Inside PowerStar class, its basic information like name, display character and such are also declared in the constructor. Meanwhile, a new action called ConsumePowerStarAction is added inside the constructor as well. This is so that the power star objects created will always have a consume action available.

3. The consumeItem() method is overrode here, and when this method is called, the actor will heal its HP by 200, and also receives a status effect called IMMUNITY to the actor.

4. Power Star will disappear after 10 ticks, regardless if its on the ground or inside player's inventory. This can be achieved by overriding the two built-in methods tick() from Item abstract class. Then, a constant instance variable called TICK_COUNT is created that has the integer value of 10, and another instance variable called tick is created that has the integer value of 0. Inside the two tick() methods, an if-statement is placed to check if tick is smaller than TICK_COUNT, if yes, tick is incremented, else, the Power Star is removed.

5. Even though IMMUNITY effect only lasts for 10 turns, we can still utilize the given enum class Status and use it to set the status of the player. This is to prevent using excessive literals in the code, just like Super Mushroom's effect.

Status — Enum Class

1. The Status class is important here as using Enum allows us to not repeat ourselves and prevent us from using excessive literals. In the Enum class, besides the given status, there are two extra status in the class: EFFECT_SUPER_MUSHROOM and IMMUNITY, the former being related to Super Mushroom, while the latter is related to Power Star. The statuses here are only given after the player consumes the magical items.

AttackAction

1. We can utilize the special status effect to grants the subsequent special effects to the player. For example, under the special effect IMMUNITY, each successful attack made by the player is an instant kill to the enemies. This is achieved by adding an if-statement to check if the actor executing AttackAction (in our implementation, we only allow the player to call AttackAction, so its the player executing) has IMMUNITY status: if yes, it will deal a large amount of damage and kills the enemy. Under IMMUNITY special effect, the AttackAction bypasses Koopa's dormant state as well, killing the Koopa instantly.

ConsumeItemAction — Abstract ConsumeAction class

1. Creating ConsumeItemAction and extending Action abstract class essentially uses OCP and SRP because we wish to have specificity to consume the consumable items, so that's using SRP. Like every other abstract class, creating ConsumeItemAction allows us to not repeat the same code that contains in the subclasses extending ConsumeItemAction, and this uses OCP.
2. This class is used for the items to be consumed. Thus, PowerStar and SuperMushroom will both have a consume action specifically for itself. This is to separate the different effects given to the player by consuming one of the items. Player will have the specific consume action to consume the items
3. Inside ConsumeItemAction abstract class, there exists a constructor that initialises a ConsumableItem type variable called item. This uses Liskov-Substitution Principle

(LSP) as the any subclasses extending ConsumableItems should have a ConsumeItemAction. Besides that, the menuDescription() method is overrode here as all the subclasses extending ConsumeItemAction will have the same menu description, thus grouping all the code inside this abstract class.

ConsumeSuperPowerAction & ConsumePowerStarAction

1. Both ConsumeSuperPowerAction and ConsumePowerStarAction extend ConsumeItemAction abstract class. Inside both classes, there exists a execute() method that overrode Action abstract class execute() method.
2. For ConsumeSuperPowerAction, its execute() method calls for SuperMushroom's consumeItem() action to apply the buffs to the player. For SuperMushroom, we allow the player to consume it off the ground. This only happens when player defeats a Koopa, and Koopa drops a SuperMushroom.
3. As for ConsumePowerStarAction , its execute() method calls for PowerStar's consumeItem(). For PowerStar, if the player already consumed a power star, we prevent the player to consume a second power star while the player's immunity effect is still ongoing. Thus, the execute() method checks if the player has immunity status, if yes, it will return a message that he already consumed a power star; else, the method calls for PowerStar's consumeItem() and apply the buffs to the player.

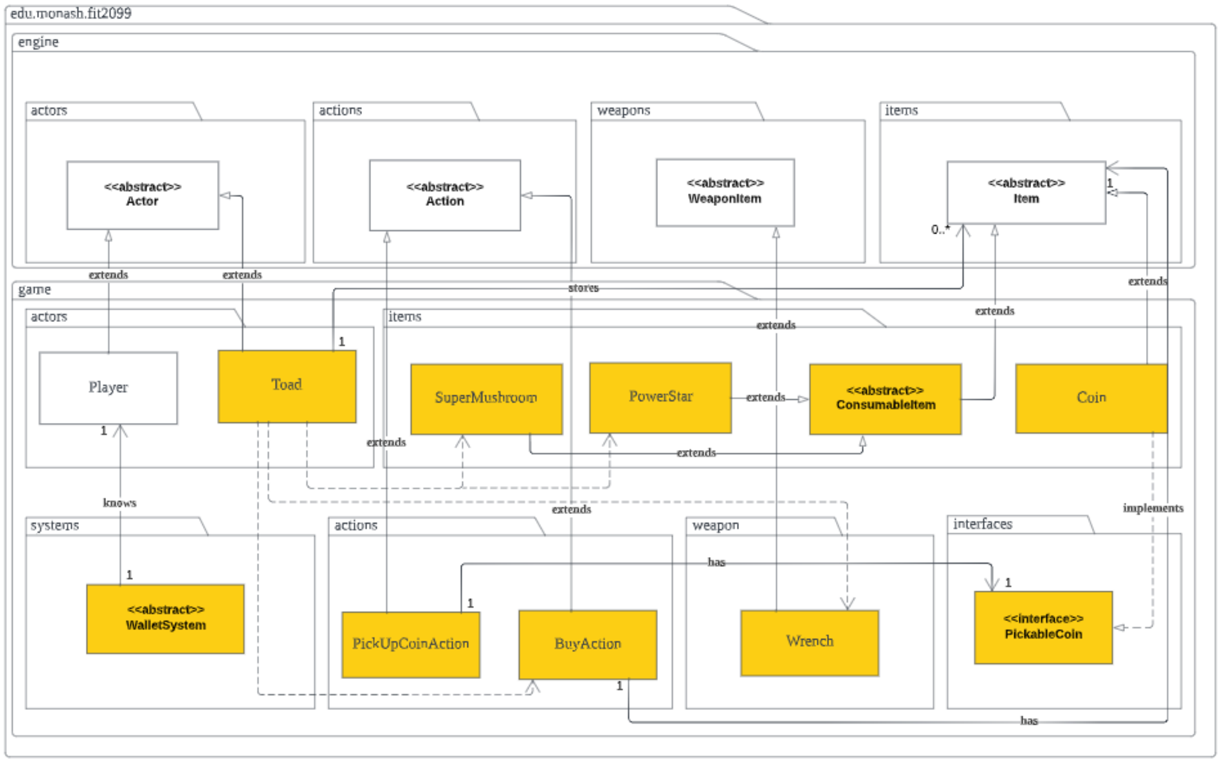
PowerStarMoveAction

1. PowerStarMoveAction is created and used specifically when the player is under the effect of power star and when player wishes to move to higher grounds (such as Trees and Walls). This class is created and extended from MoveActorLocation class because this class is used solely for moving the actor (player, in this case) when the actor is under the power star effect. This obeys both SRP and OCP.
2. Inside PowerStarMoveAction class, there contains a constructor and an execute() method. The execute() method overrides the MoveActorLocation execute() method. In this method, it simply moves the actor to the respective location it needs to go, then it gets the location of the actor is in (this location is the location after the actor moves), and set the ground of the location to be dirt (destroying the high ground and turning it to dirt), and on the ground it adds a Coin object (destroyed ground will drop a coin) that has a value of 5$.

3. PowerStarMoveAction is used in Wall class and Tree abstract class, inside their allowableAction() method. Inside the method, it runs an if-statement to check if the player has IMMUNITY status: if yes (means player consumed a power star), it will create a new PowerStarMoveAction; else, it proceeds as normal.

REQ5: Trading 💰

REQUIREMENT 5 CLASS DIAGRAM



(The screenshot above shows the UML class diagram for REQ 5)

Implementation

Toad

1. A friendly actor which serves a purpose of selling items to the player.
2. Since Toad is selling the 3 items (Wrench, Super Mushroom and Power Star) , therefore instead of creating 3 separates arrayList that stores each of those items, I have decided to create a HashMap that stores the items that the Toad is currently

selling together with the price of the items. By doing so, we are adhering to the Dependency Inversion Principle by depending on abstraction instead of concrete classes and also the Liskov-Substitution Principle.

3. Toad will also add an instance of BuyAction into its allowableActions method so that when a player approaches the toad in the gameMap, they can use the BuyAction to buy the items that the toad currently has , provided that the player has sufficient wallet balance.

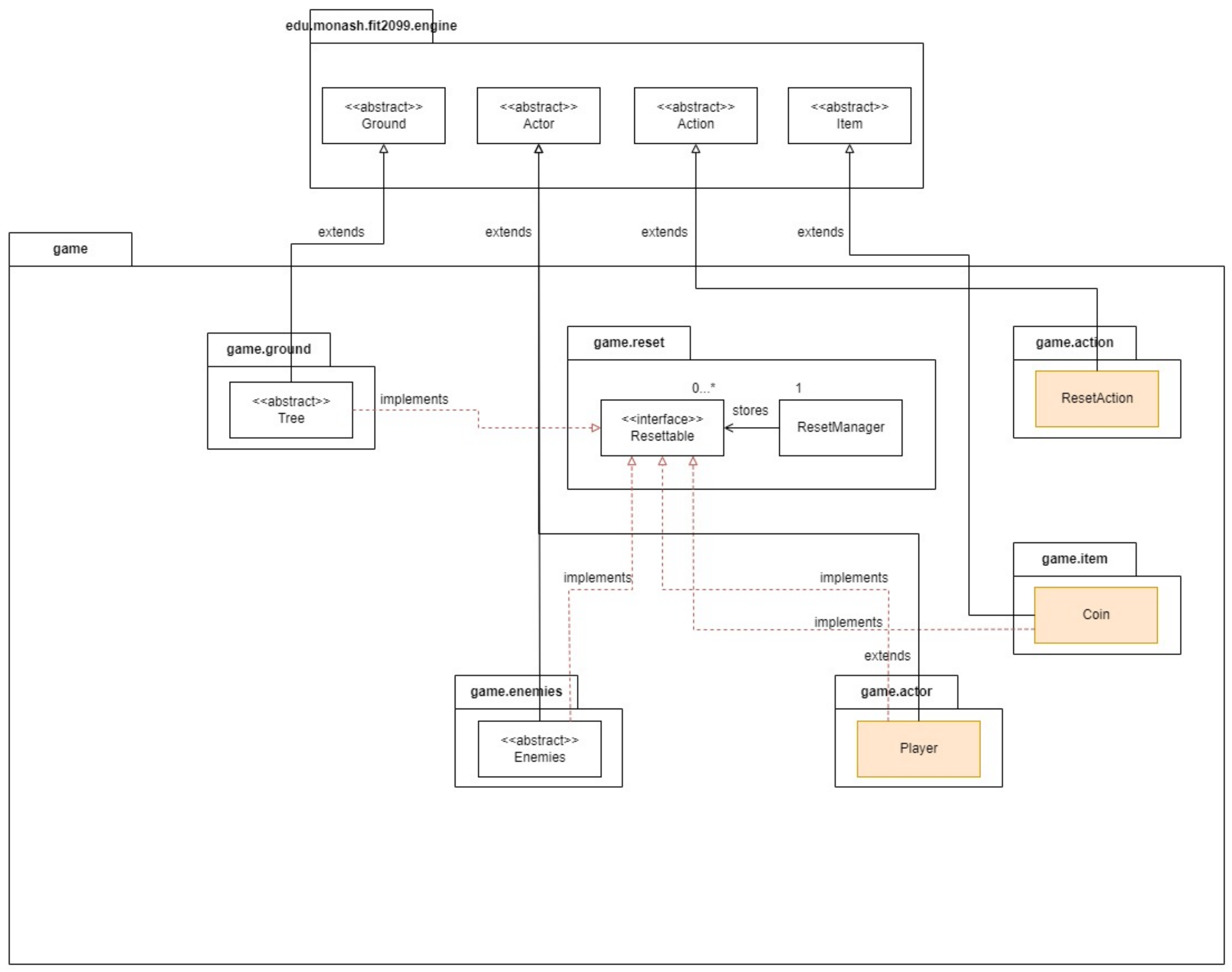
BuyAction

1. An action that a player can take to buy items from Toad.
2. Inside the execute method of the BuyAction class, it will check whether the player’s wallet balance is sufficient to buy those items from Toad , if it does then the item will be added into the player’s inventory and the wallet balance will be deducted. However, if the player’s wallet balance is insufficient then a message will be printed in the console.

WalletSystem

1. The purpose of creating a WalletSystem class is to allow us to keep track of the player’s balance as well as allow us to add to player’s balance when the player collected some coins and also to allow the player’s balance to be deducted from the wallet when the player is buying items from the Toad.
2. This class is declared as abstract because we will not create any instance of type WalletSystem. This helps us achieve abstraction.

REQ7: Reset Game



(The screenshot shown above is the UML designed for requirement 7)

Implementation

There are two classes given already in the game package, Resettable interface and ResetManager. We can use these classes to implement resetting the game.

Resettable

1. We can start making the classes that are required to be reset implements Resettable interface. This action is essentially using Interface Segregation Principle (ISP). By implementing Resettable interface, all the constructor's of the classes implementing Resettable will add a new line of code called "this.registerInstance()". This line of code makes the objects created under the classes that implements Resettable to be added into the ResetManger resettableList ArrayList. This allows us to keep track which objects are to be reset when the ResetAction is called

2. Classes that implements Resettable needs to override resetInstance() method that exists in Resettable interface. For Tree abstract class and Coin class, their resetInstance() method basically gets the entire map, and loop every ground to check if the current ground is its object
3. For trees, it has 50% chance to be removed, so the if-statement checks for the 50% chance: if yes, the tree will be removed; else, it will stay.
4. For coins, it will always be removed from the ground. Thus, if the ground contains the coin object, it will be removed.

ResetManager

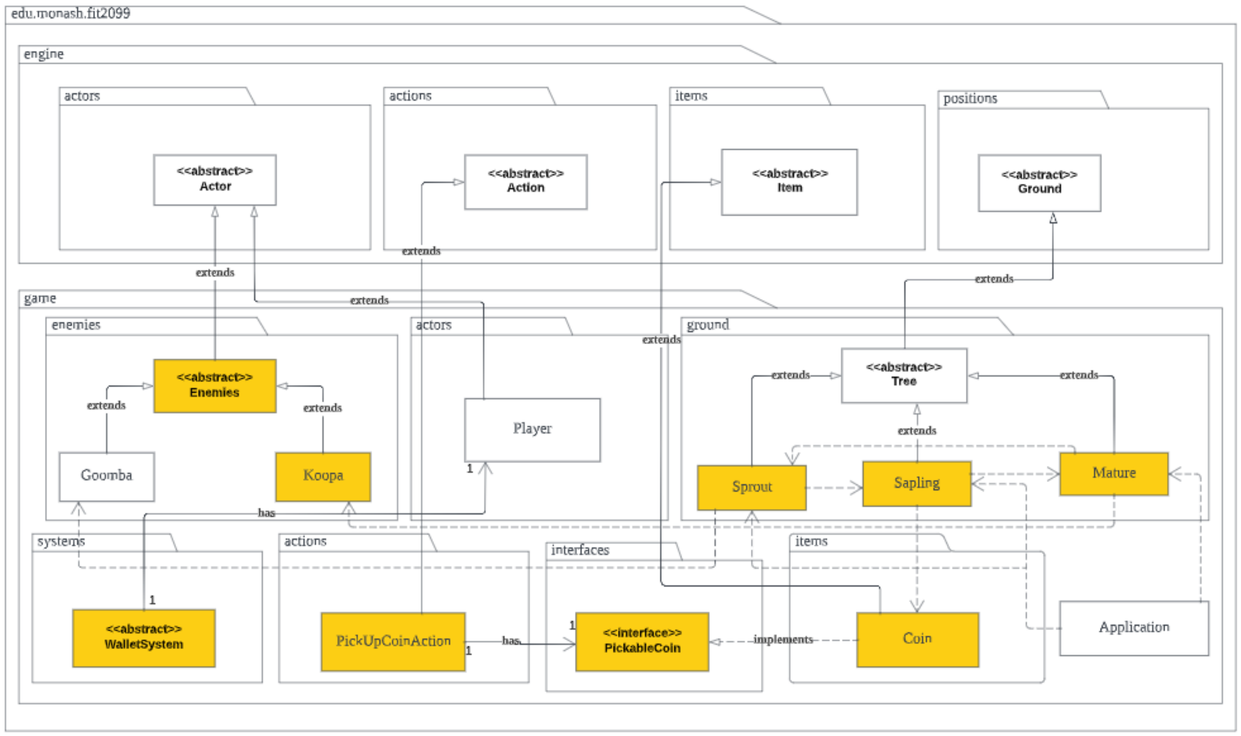
1. Inside ResetManager class, it has a boolean variable called “reset”. This variable is used to check if the player has reset the game already or not.
2. Besides that, there exists a run() method in ResetManager class. Inside the run() method, it loops all objects in the resettableList ArrayList in ResetManager, and call its resetInstance() method. After running the for loop, it will set the “reset” variable to true.

ResetAction

1. A ResetAction class is created to execute the run() method. Creating ResetAction class and extending from Action abstract class obeys OCP and SRP. This action is added inside the Player class.
2. In Player class playTurn() method, we would check if the “reset” variable is true inside ResetManger class: if its false, the player will have the option to reset the game, and this is achieved by creating a ResetAction and adding it inside’s Player’s ActionList; else, it does not have a ResetAction. Doing this allows the player to only reset the game once.
3. If the player executes the ResetAction, inside its execute() method, it will run the run() method in ResetManager. And such, the entire game is reset.

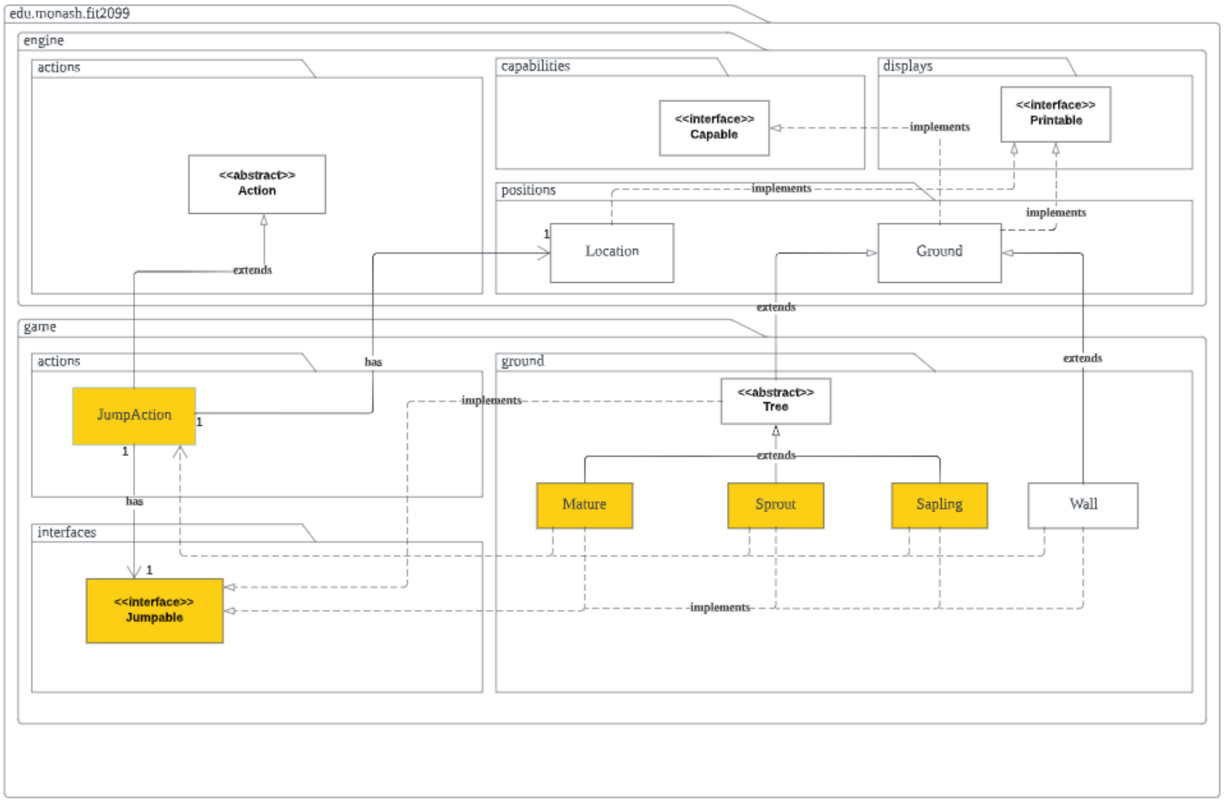
UML Class Diagrams

REQUIREMENT 1 CLASS DIAGRAM

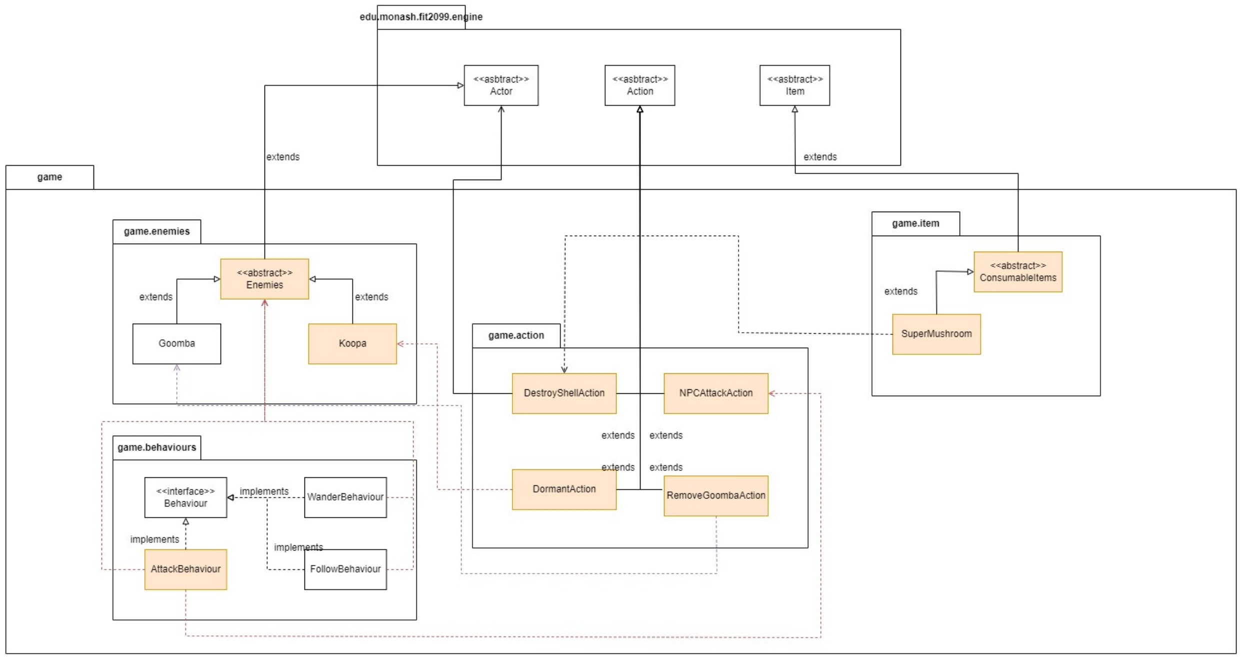


Requirement 1 Class Diagram

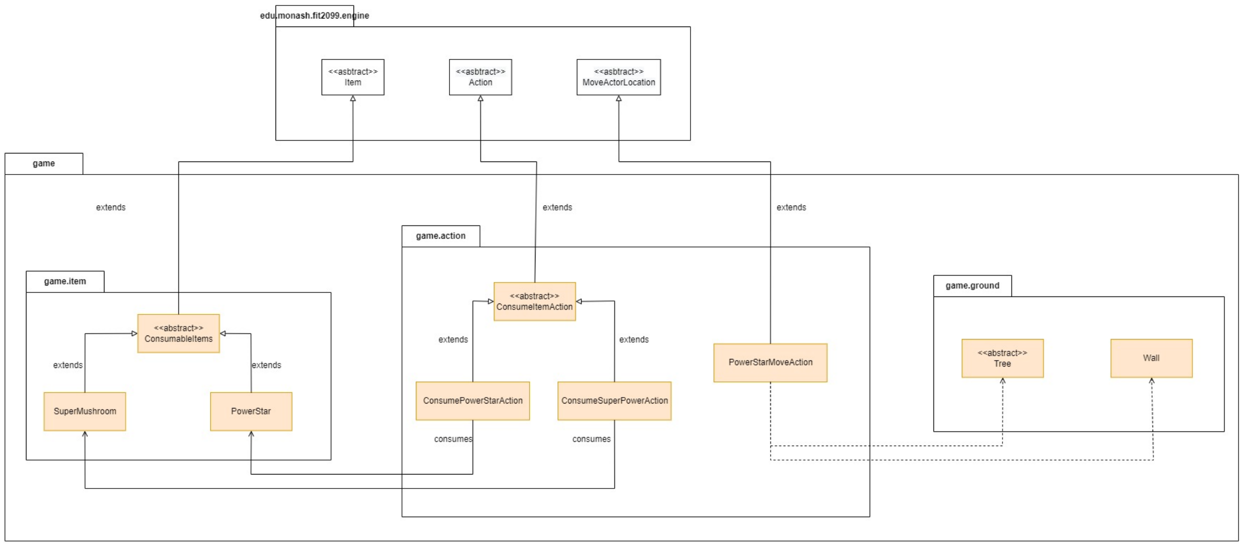
REQUIREMENT 2 CLASS DIAGRAM



Requirement 2 Class Diagram

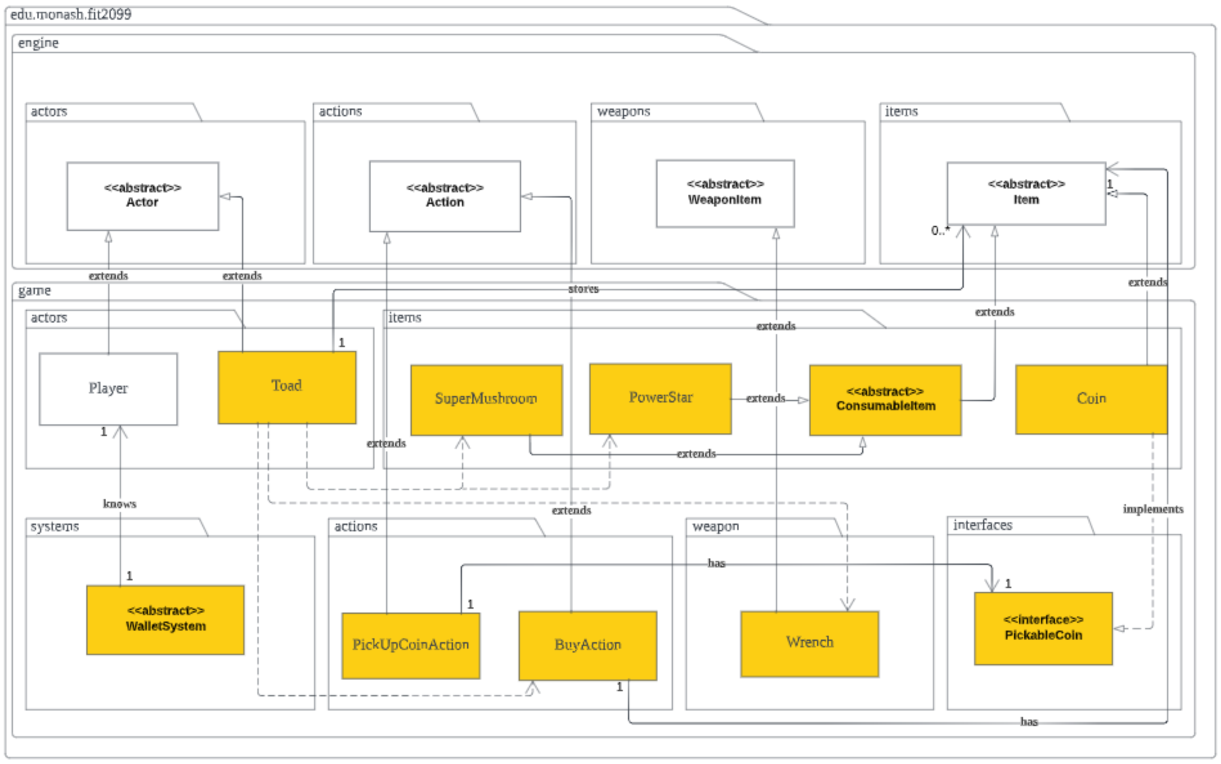


Requirement 3 Class Diagram

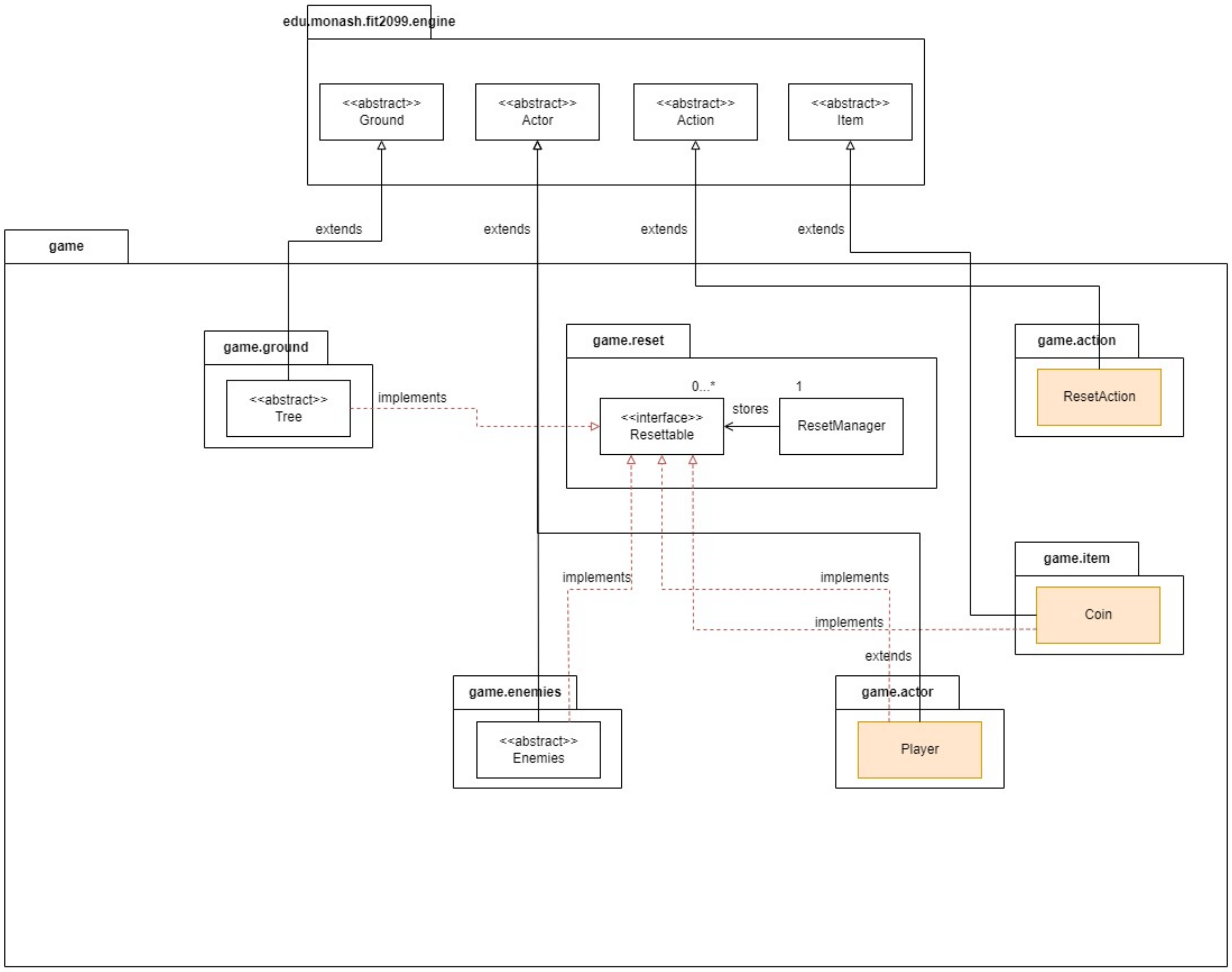


Requirement 4 Class Diagram

REQUIREMENT 5 CLASS DIAGRAM

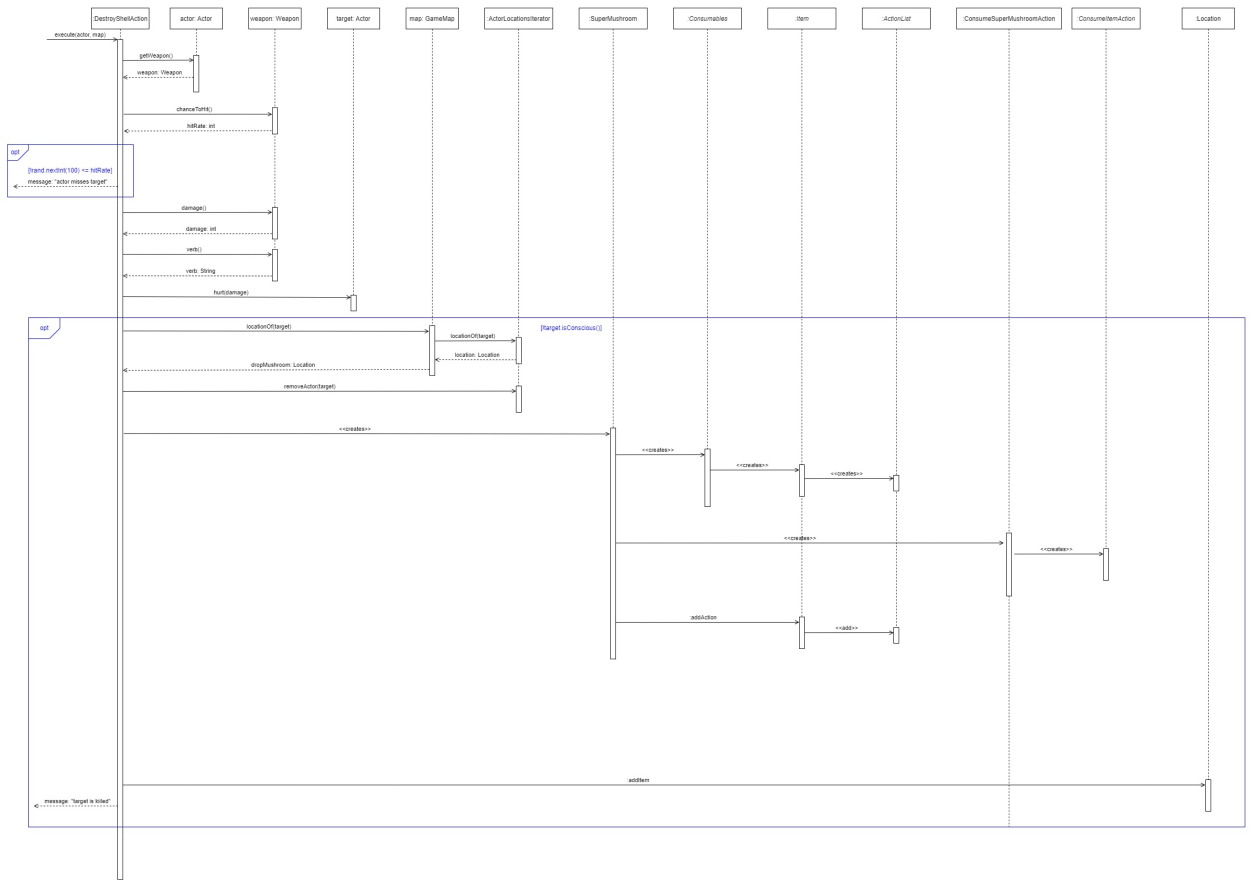


(The screenshot above shows the UML class diagram for REQ 5)



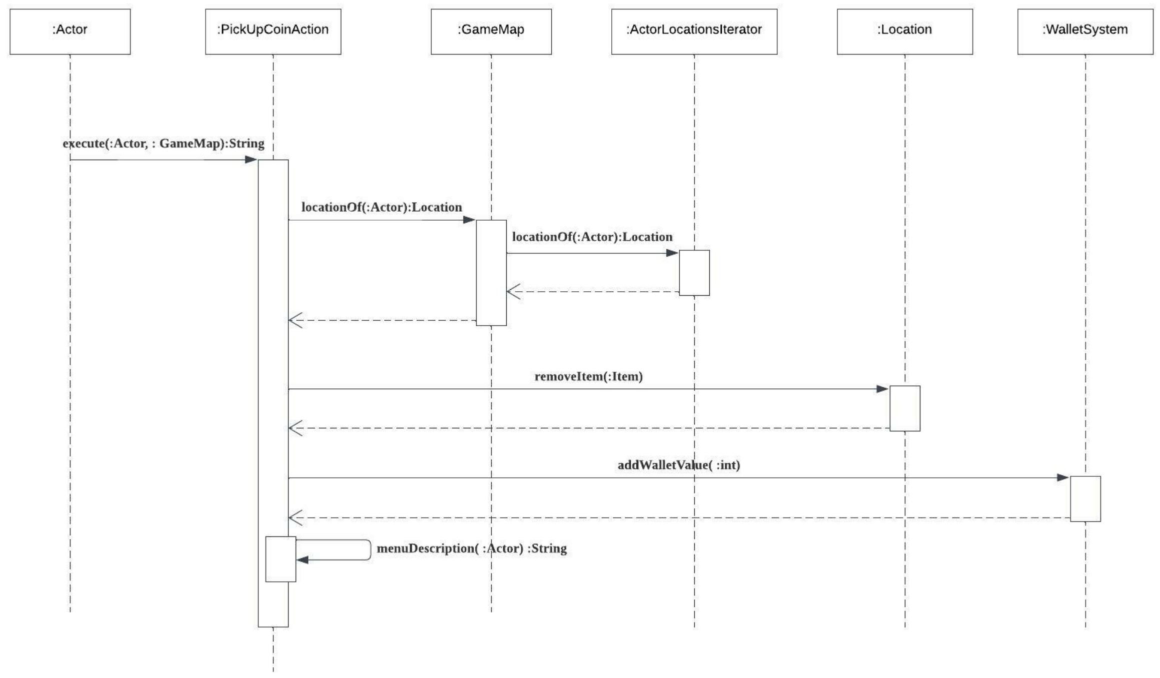
Requirement 7 Class Diagram

Interaction Diagrams



Sequence Diagram for DestroyShellAction

REQUIREMENT 5 : PICKUPCOINACTION SEQUENCE DIAGRAM


:Actor
:PickUpCoinAction
:GameMap
:ActorLocationsIterator
:Location
:WalletSystem
execute(:Actor, : GameMap):String
locationOf(:Actor):Location
locationOf(:Actor):Location
removeItem(:Item)
addWalletValue(:int)
menuDescription(:Actor) :String

Sequence Diagram for PickUpCoinAction