

Lab 3 – Team 1

Assignment 1 – Design rationale & UML diagrams

Group members:

- 1) Chen Yang Seah 32311168**
- 2) Ong Yong Jun 31861407**
- 3) Yuki Wong Yi Wei 32599862**

Requirement 1

Design rationale

In order to implement the three different stages of trees, we create 3 subclasses of the Tree class, by extending the Tree class through polymorphism. These subclasses include Sprout, Sapling and Mature. This allows the subclasses to inherit the common features of a tree, such as having an age and having the same chances to be removed in a reset (implemented in later requirements). This obeys the Single Responsibility Principle and Open-Closed Principle, where the parent Tree class is only responsible for the common features of all trees and adding new features in the extended subclasses will not affect the parent class itself. The Tree class is an abstract class as a Tree object must exist in one of the tree forms implemented as subclasses, thus it cannot be instantiated as a Tree object.

Since all the 3 subclasses have features for spawning other objects into the game, they will all have a dependency with the Location class as it is needed to determine the spawning point.

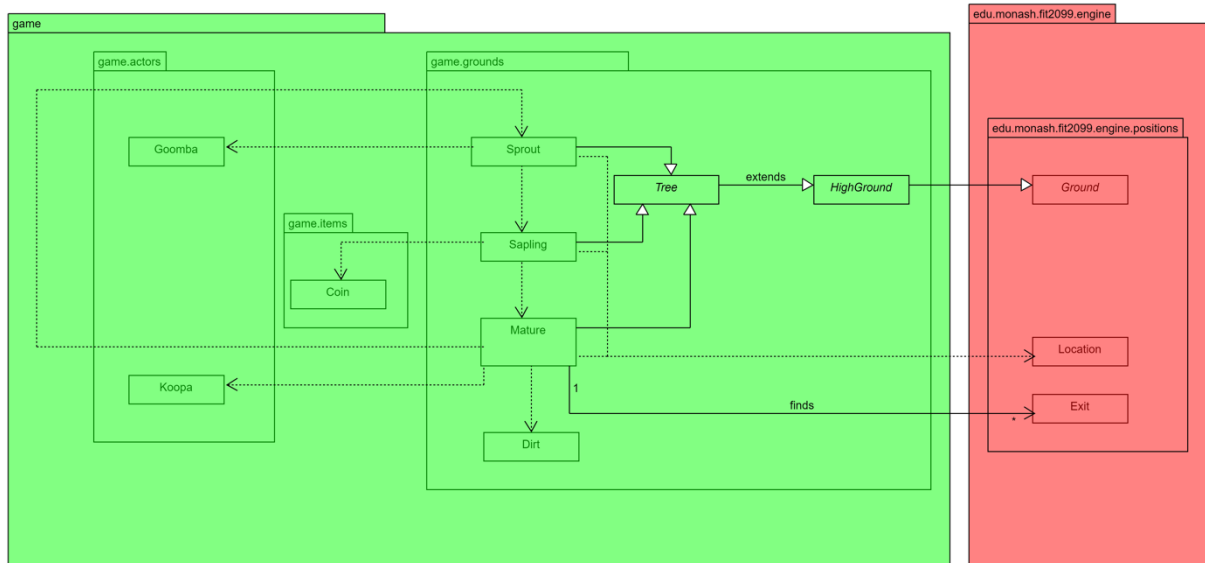
Sprout spawns Goomba, Sapling spawns Coins and Mature spawns Koopa. Therefore, they will have a dependency relationship with their respective spawns.

A Sprout will grow into a Sapling, then a Sapling will grow into a Mature, ending with a Mature eventually spawning a new Sprout around itself, creating a loop. Therefore, all of them will also have a dependency relationship with their next form, as this form of growing is essentially just an action of replacing itself with a spawn of the next form.

A Mature replaces itself with Dirt at the end of its life cycle therefore having a dependency relationship with Dirt. Furthermore, a Mature will also search for an exit and spawn more Sprouts, thus it has an association relationship with Exit as each Mature has their own list of exits that are eligible for spawning sprouts in each turn.

Tree is a form of high ground within the game thus it will be implemented as a subclass of the HighGround class, which is already a subclass of the Ground class. This allows the Tree class to inherit the common features of all HighGround classes and add or modify other features exclusive to Tree classes, which include the common features of all trees stated above. The same fact applies on the inheritance of the HighGround class from the Ground class, which will be further explained in Requirement 2.

UML diagram



Requirement 2

Design rationale

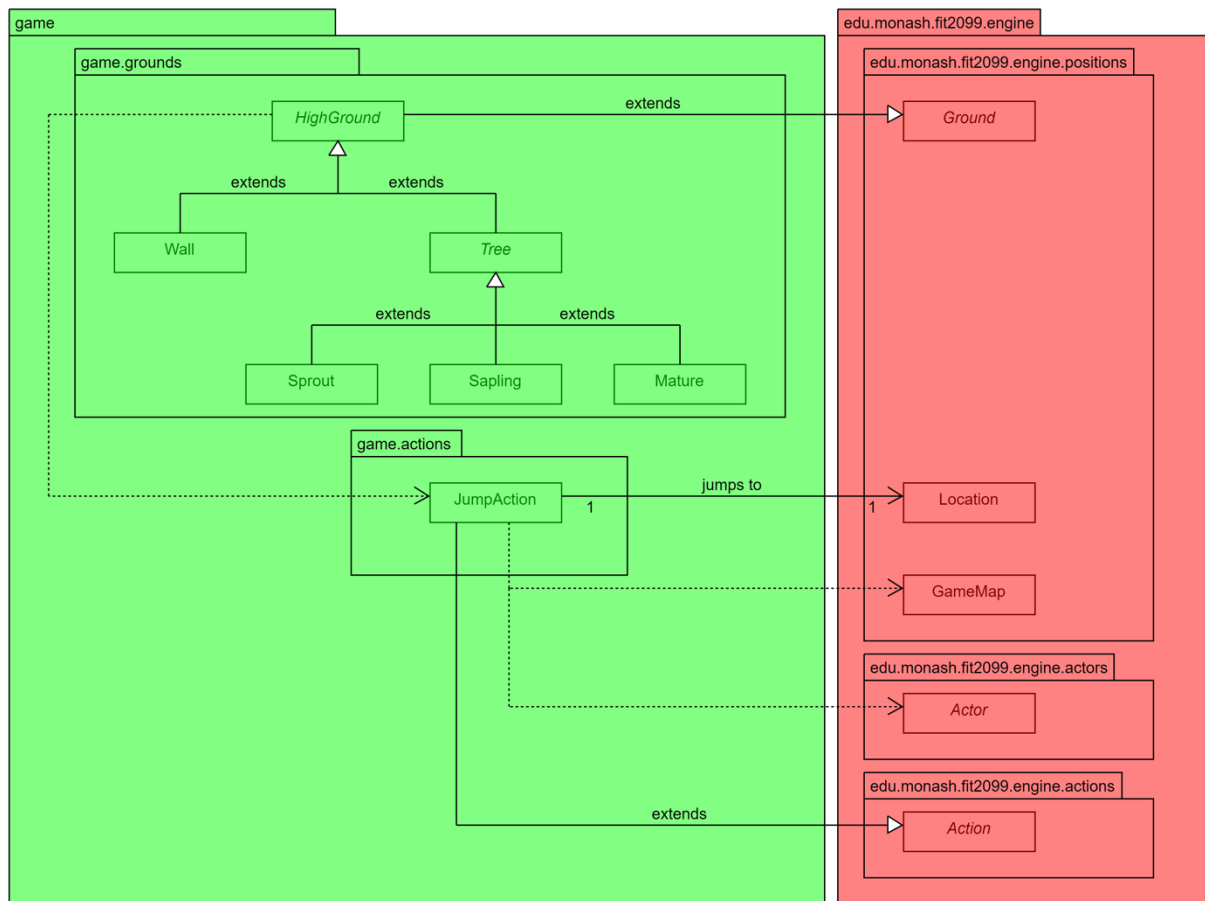
The JumpAction class is extended from the Action class through polymorphism to inherit all of its features to allow for modification to cater to JumpAction's exclusive features. It has an association relationship with the Location class as every JumpAction will have its own target location to jump to, which makes it one of JumpAction's instance variables.

The JumpAction class has dependency relationships with the GameMap class and the Actor class, as the GameMap is needed to check if the target location is occupied with an Actor, as well as to move the actor to the target location, while the Actor class is required to check if the actor has consumed a SuperMushroom, and to hurt the actor in the case of an unsuccessful jump.

The HighGround abstract class is extended from the Ground class, thus inheriting all of Ground's features, as HighGround "is a" Ground. The HighGround class then implements the features common to all high grounds, which include adding a JumpAction to its list of allowable actions, not allowing any actor to enter and many other more. These features can be then inherited by HighGround's subclasses, which in this case includes the Wall and Tree class, which also includes all of Tree class's subclasses. This obeys the SRP and OCP as HighGround class is only responsible for features related to high grounds, and more subclasses of HighGround class can be created without the need to modify the HighGround class. The HighGround class is abstract as it should not be instantiated as an object and must only exist in the form of its subclasses.

The HighGround class has a dependency relationship with the JumpAction class as it will instantiate a JumpAction object to be added into its list of allowable actions.

UML diagram



Requirement 3

Design rationale

The Enemy abstract class inherits all of Actor's features through polymorphism to implement its exclusive features. It has an association relationship with the Behaviour interface as it has a hash map containing behaviours as values and their priorities as keys as an instance variable common to all enemies, which determines what automated action the enemy will initiate each turn. The Enemy class has dependency relationships with the AttackAction, ActionList, AttackBehaviour and FollowBehaviour classes, as an AttackAction may be added into the other actor's ActionList of allowable actions in the case where another actor is in range to attack the enemy. The AttackBehaviour and FollowBehaviour classes are needed as they may be put in the list of behaviours in the case where the enemy comes into contact with another actor.

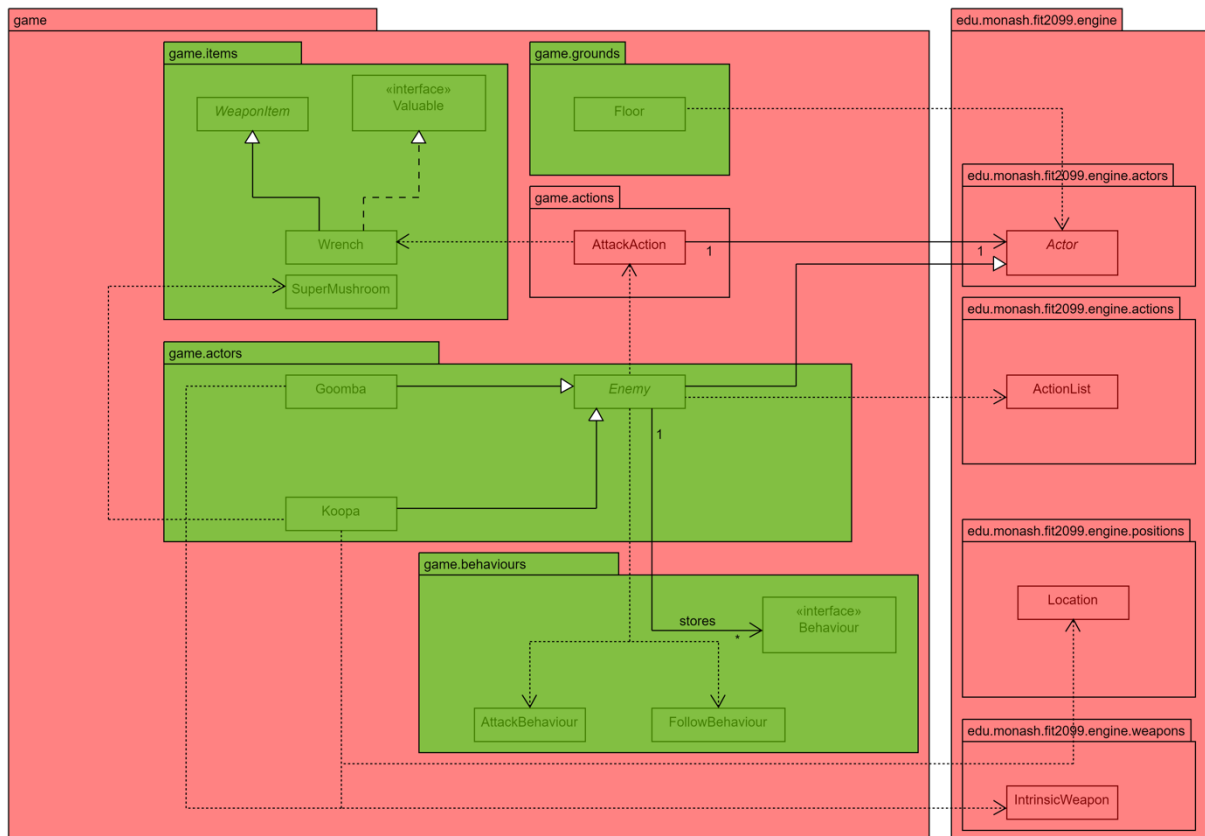
The Enemy class is abstract as it should not be instantiated and can only exist in the form of one of its subclasses, which for now are Goomba and Koopa. This allows the stated classes to inherit all basic features of an enemy and add in their own features and modifications to suit their exclusive features. This obeys the SRP and OCP as the Enemy class is only responsible for the basic features of an enemy and any subclasses extended from it will not require modifications towards the Enemy class.

The Goomba and Koopa classes have dependency relationships with the IntrinsicWeapon class as they have to implement their own intrinsic weapon based on their specifications. Besides, the Koopa class also has dependency relationships with the Location and SuperMushroom classes as those classes are required to drop a new SuperMushroom object on a specific location the Koopa is on whenever its shell breaks. The three different states of Koopa (shell intact, shell broken, dormant) can be implemented through the use of capabilities in the form of a Status enum, which is not part of the UML diagrams.

The Wrench class is extended from the WeaponItem class, inheriting its features, while also implementing the Valuable interface, which will be explained more in later requirements. The AttackAction class has a dependency relationship to it as it needs to know if the AttackAction inflicted towards a certain target involves a Wrench object, as to break the shell of the Koopa if the target actor is indeed a Koopa instance.

The Floor class is extended from the Ground class. It has a dependency to Actor class as it needs to determine whether the actor wanting to enter the Floor is an enemy or not.

UML diagram



Requirement 4

Design rationale

The ConsumeAction class is extended from the Action class to inherit all of its features to allow for modification to cater to ConsumeAction's exclusive features. It has an association relationship with the Item class as every consume action consumes an item which is one of its instance variables. It also has a dependency relationship with the Actor class as it needs to add a certain capability to the actor and remove the item consumed from the actor's inventory.

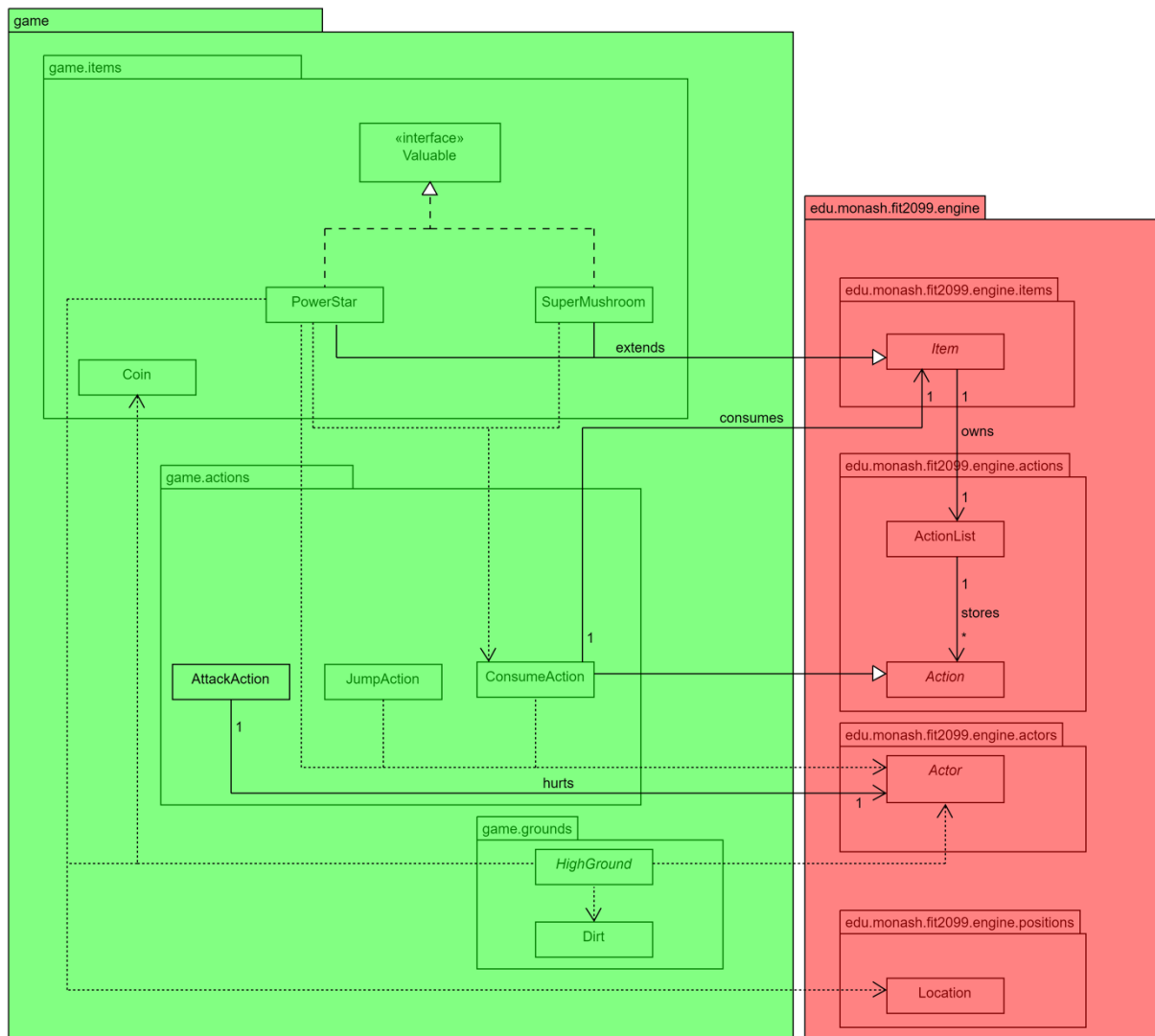
The SuperMushroom and PowerStar classes are extended from the Item class through polymorphism, inheriting the basic features of an Item class. They also implement the Valuable interface, which will be further explained in the later requirements. They both have dependency relationships with the ConsumeAction class as it adds a consume action option to its allowable actions in the case of being in the inventory of an actor. This obeys the SRP as each magical item is responsible for their own specifications and buffs, without the need to modify its parent class in the case of a new subclass creation.

Besides, the PowerStar class has dependency relationships with the Actor and Location class as power stars have a time limit before they fade away regardless of whether they are on the ground in a location or being carried in the inventory of an actor, which requires those classes to check if the power star should fade away or not in every turn.

To implement the buffs given by the magical items, the AttackAction class will have an association relationship with the Actor class as it needs to check if its target instance variable, which is an actor, possesses a buff that makes it immune to any attack. Besides, the JumpAction will have a dependency with the Actor class as it needs to check if the actor possesses a buff that makes its success rate of jumping 100%.

Moreover, the HighGround class will have dependency relationships with the Actor, Coin and Dirt classes, as it needs to check if the actor entering the high ground possesses a buff, which destroys the high ground and converting it into a Dirt instance, while also spawning a Coin instance on the ground.

UML diagram



Requirement 5

Design rationale

Certain items in the game are given value such as [Coin], [SuperMushroom], [SuperStar], and [Wrench]. So when implementing these classes, value should be given to them. There are 3 approaches in doing so.

Approach 1:

They will inherit from the abstract [Item] class because they perform similar functions to the [Item] class but have the added functionality of defining their own value/price through the assigning value in constructor, whereas the [Wrench] class will inherit from the abstract [WeaponItem] class because it is a weapon but also has the ability to define its own value/price through constructor. This shows that all these classes have an inheritance relationship with their respective parent class. It also demonstrates the application of the SPR principle, as each child class is responsible for defining its own value.

Approach 2:

They will implement an abstract [ValuableItem] class inherited from the [Item] class, as this class provides value to all items with value via the getValue() method, except the [Wrench]. Only the [Wrench] class will be extended from [WeaponItem], and its value will be defined via its constructor because [Wrench] is a weapon and must follow the default behaviour of [WeaponItem]. This design obeys the Open-closed Principle where all traded item classes can extend the behaviour in their respective parent class with abstraction.

Approach 3:

Regardless of which class they extended, they must implement the interface [Valuable] as this interface aided them in defining the item's value via the getValue() method. This adheres to the Interface Segregation Principle, as the interface [Valuable] only has the function of obtaining the value of an item.

Which is better?

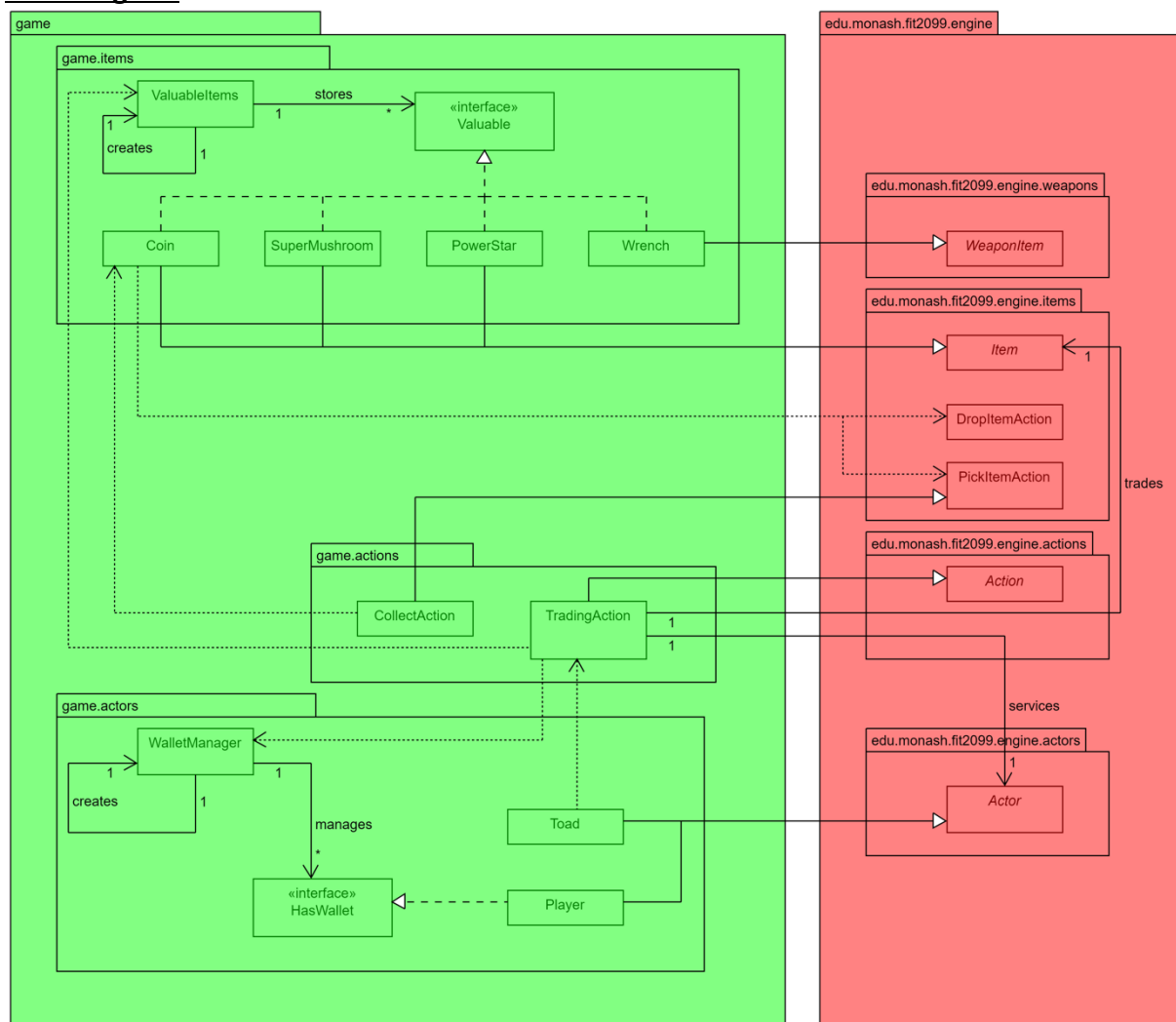
Approach 3

This is due to the fact that all of the traded items have a similar implementation for obtaining the value, so an interface would be the best choice due to the benefit of allowing multiple classes to implement that interface. It also enabled the traded item to have multiple interfaces implemented for different functionality, which is preferable than approach 2 of having an abstract class. Since [Wrench] class must inherit [WeaponItem] class because it is a weapon item, it cannot inherit from abstract [ValuableItem] class because one class can only inherit one parent class, the solution is to have value as an input parameter value when constructing constructor in [Wrench] class, hence approach 2 is not a good idea. For approach 1, it tends to repeat the codes, so it violates Don't Repeat Yourself, hence approach 3 is the best choice.

Since the value of coins in this game changes as coins are traded, generated, and collected, the [Coin] class uses a wallet to store the balance. A [WalletManager] is required to keep track of the total balance of a [Player], as well as a hasWallet interface, which will be implemented in the [Player] class. In this game, the player can trade with Toad(O) to obtain necessary items. The [Toad] class extends from the abstract [Actor] class and will allow the execute() method

in [TradingAction] to be called in [Player] class for trading purposes, resulting in a dependency relationship between [Toad] and [TradingAction]. As a result, whenever there is trading, [TradingAction] class that extends from [Action] is called. A [ValuableItems] containing a list of [Valuable] interface implemented objects is required to keep track of all instances that implement the [Valuable] interface. Thus, [TradingAction] class also has a dependency relationship with [ValuableItems] to get the value of the item that implements the [Valuable] interface. In an execution of [TradingAction], it will first be informed of the balance in the wallet by [Player] via [WalletManager], and if the balance is greater than the value of the traded item gotten via [ValuableItems], the player is permitted to trade. If they are traded, the traded item will be added to the inventory list and the balance in the wallet will be deducted. Otherwise, the message "You don't have enough coins!" will appear, and the player will be unable to obtain the item. This demonstrates that [WalletManager] has an association relationship with [hasWallet]. When coins are collected from the map, the [CollectAction] class, which extends from [PickUpItemAction], picks up the coin, and the player calls walletAdd() in the interface to increase the amount. This demonstrates that [Coin] is dependent on [CollectAction].

UML diagram

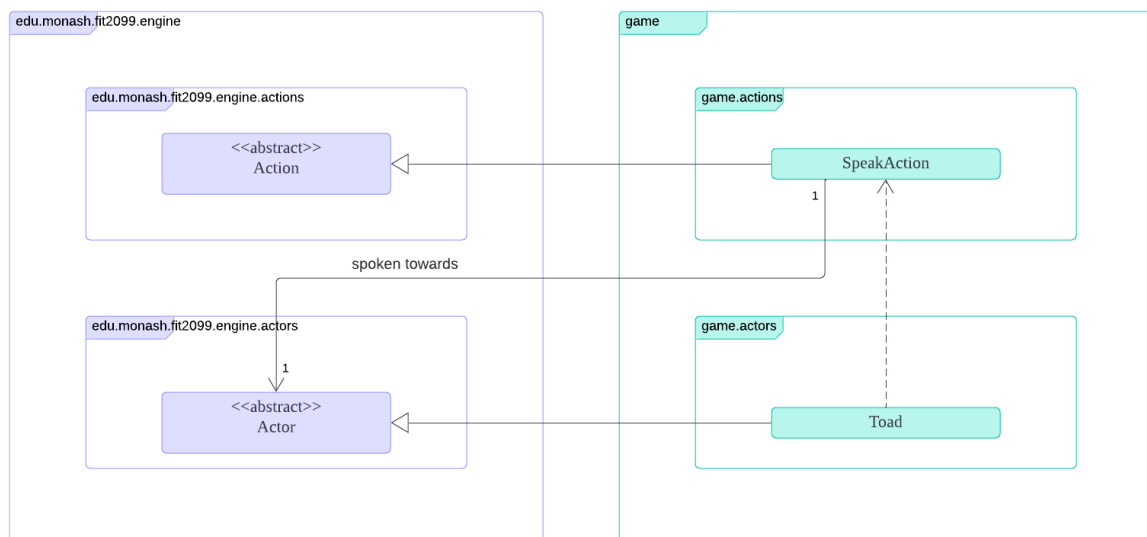


Requirement 6

Design rationale

In this game, whenever the player walks near [Toad], it can speak with [Toad], and [Toad] will randomly speak one sentence at a time. To accomplish this, a [SpeakAction] will be created that extends from [Action]. It works in the same way as [AttackAction]. It is designed in such a way that four sentences are chosen at random from an arraylist that contain the sentences. But we know that in this game, once the player has a Wrench, he will not say, "You might need a wrench to smash Koopa's hard shells." To do so, he must first determine the player's capability, such as whether it has the status of having Wrench. If it doesn't, this sentence will be added to the arraylist because there's a chance he'll speak it; otherwise, it won't. The same goes for PowerStar. The sentence "You better get back to finding the Power Stars." will not be spoken once the Power Star effect is activated. The same approach is used, but this time it determines the player's capability, whether it has the status of activating [PowerStar]. The other two sentences, on the other hand, will be added to the arraylist regardless of what happens. When [SpeakAction] is called, he will select one at random from the arraylist to speak, indicating that [Toad] has a dependency on [SpeakAction].

UML diagram



Requirement 7

Design rationale

The Resettable interface requires the classes that implement it to implement a `resetInstance()` method that resets an instance of the class. Thus, the Coin, Tree, Player, and Enemy classes and their respective subclasses if any will implement the Resettable interface as they are the classes that are resettable.

For the Coin, Tree, and Enemy classes, their implementation of `resetInstance()` will only involve adding a status called RESET into its list of capabilities, indicating that the instance is to be reset. Their respective implementations of resetting will be done in their `playTurn()` or `tick()` methods as they require other properties like location to convert the ground into Dirt and remove coins, map to remove enemies and many more. These reset implementations will only be conducted if the instance possesses the RESET status, indicating that the Player has decided to reset the game.

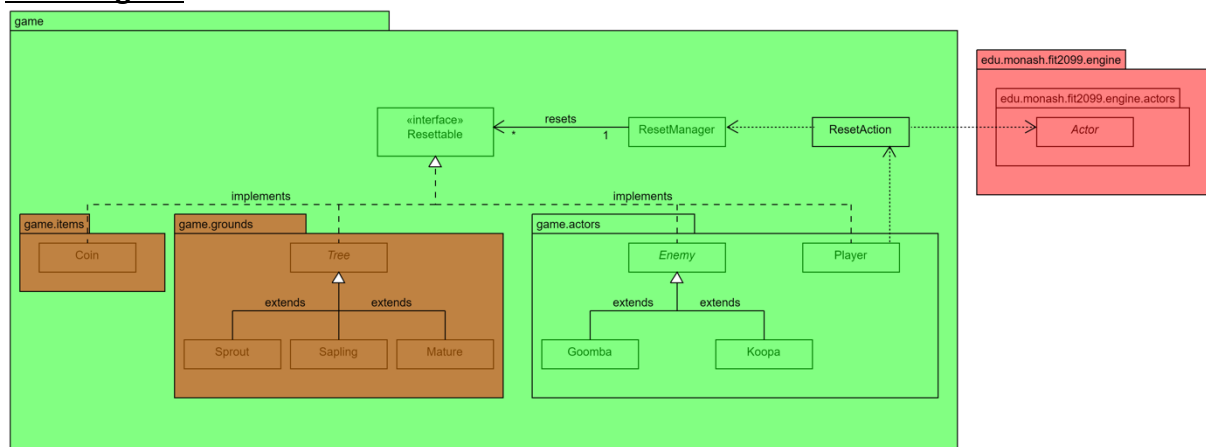
For the Player class, its implementation can be done right in `resetInstance()` as it only involves healing and removing capabilities. To ensure that the Player only gets to reset once in the entire game, it can be given a capability called CAN_RESET and that capability can be removed in the implementation of `resetInstance()`.

The ResetManager class has a list of Resettable instances as an instance variable, to keep track of all instances that are resettable, thus it has an association relationship with the Resettable interface. It is responsible to run the `resetInstance()` implementations of all resettable instances to reset them in an event where the player resets the game.

The ResetAction class executes by running the ResetManager's run method and it needs to know which actor initiated the reset action, thus it has dependency relationships with both the Actor and ResetManager class.

To allow the player to be given a reset option in the menu, the ResetAction class is created as an allowable action in Player's `playTurn`, thus the Player class has a dependency relationship with the ResetAction class.

UML diagram



Sequence diagram for TradingAction execute()

