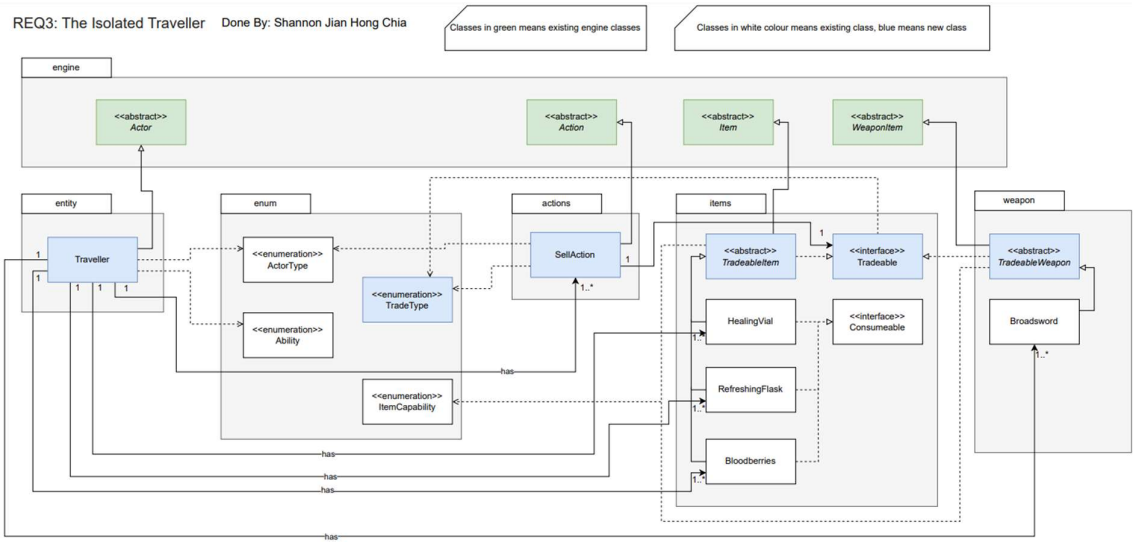
Design Rationale: Req 3 (Done by Shannon)



Brief

The UML Diagram represents requirement 3, where 6 new classes were introduced, namely two new concrete classes, 1 new enumeration, and 1 interface that is implemented by the 2 new abstract class.

Requirement 3 can essentially be subdivided into 2 new parts, where 2 new functionalities will be built upon the current game implementation, namely the introduction of a traveller that sells item, and a new tradable/transaction system between the traveller and the player.

Traveller

The new traveller class is a concrete class, it serves as a non-moving actor in a building within the game, as a trader offering items for sale to the player (SRP).

Key design decision in this implementation, is that it will extend the “Actor” class, inheriting it to allow for leverage existing functionality for interacting with other actors and the game map (LSP). Moreover, the traveller will be given a capability – ‘ActorType.TRADER’ and an ability – ‘Ability.TRADE’, as this allows to distinguish between the trader and a player.

The Travellers inventory can be initialised within the constructor, and prices for each items are also set during the initialisation, providing flexibility for traders to sell items at different price (OCP).

Essentially, the way the Traveller performs the SellAction (the SellAction class will be explain below) for each item is by the ‘getItem’ method, where it generates a list of items that can be traded by an actor, and iterate through the actor’s inventory, if an item within the inventory has the capability – ‘ItemCapability.Tradable’, it will create a corresponding SellAction instance. This allows for better method encapsulation logic and promotes modularity.

Tradable

The Tradable interface is designed to define a set of methods that will be implemented by tradableItems and tradableWeapons that can be traded between actors in the game. The abstraction of the interface is that is contains common methods that fovuses on the essential aspects related to trading, such as getPrice, isPriceAffected, affectedPrice, getTradeType, setPrice, and newItemInstance (SRP).

Methods in this interface

getPrice()	-	Retrieves base price of the tradable item
isPriceAffected()	-	Random generator, that checks whether the price will be affected by the trader (scam)
affectedPrice()	-	The affected price after the scam (Price multiplier)
getTradeType()	-	There are 3 distinguish trade type, not scammable, steal items, or steal runes, that can be performed by the traveller
setPrice()	-	Sets the base price of the tradable item
newItemInstance()	-	Creates a new instance of the tradable item

TradableItem and TradableWeapon

The TradableItem and TradableWeapon class will implement the Tradable interface, which will then be inherited by child class that is able to be traded between the player and the traveller (LSP).

The isPriceAffected() will return a Boolean when called, to see whether a fraud will occur during the transaction.

SellAction

The SellAction class represents an action where an actor sells a tradable item to another actor. This class is designed to handle transaction between the seller and the buyer (SRP), considering the possibility of scams, affected prices, and different actor types (player or traveller).

The way the SellAction is executed, is by first checking whether a scam will occur or not, by calling the TradableItems.isPriceAffected(). Followed, it will check whether the seller is a player or a traveller by checking the sellers player type, if it's a player, it will call the handlePlayerSellAction, whereas if it's a traveller it will invoke the handleTraderSellAction. This is crucial as different type of scams will occur depending on whos the seller.

This design adheres to the OCP, as it accommodates new functionalities without altering its core structure. Moreover, it implements necessary methods for executing and describing the sell action without unnecessary methods (ISP).

*A sequence diagram of the SellAction can be found in the docs.