Design Rationale REQ2

Design Rationale

Implementation & Why:

Blacksmith:
Blacksmith extends from the actor class, where his purpose is to allow the player to upgrade their weapons. In order to only allow the player to upgrade, it checks for the actors in his exits in the allowableActions method for only actors with EntitiyTypes.PLAYABLE.

Similar to TradeableItems in our Trader class, we have an getItems() method that returns an ActionList, which consists of all the upgradeAction of every item that has the capability to be upgraded (Ability.UPGRADE). This method utillizes downcasting converting Item to Upgradeable in order to access the interface methods, in this case so that we can get the price of the upgrade and the boolean singleUpgrade value that would be passed into the UpgradeAction. This ensures single responsibility principle is adhered as the Blacksmith class takes care of all the upgrade logic, instead of relying on the individual

Upgradable (interface):
Every item that can be upgraded in the game would implement the Upgradable interface in order to maintain Interface Segregation Principle so that only items that can be upgraded have the functionality needed. Ensures all the items that can be upgraded has the upgradePrice, the upgrade method itself, as well as whether it is a single upgrade.

UpgradeAction:
An action that takes the weapon to be upgraded and upgrades the item if the player has enough balance. This ensures Single Responsibility Principle is adhered to as the upgrade change is handled by the item itself, the action just calls the upgrade method of the item and removes the capability to upgrade if the item is single upgrade.

HealingVial and RefreshingFlask upgrade:
Implements the Upgradable interface. IncreasePercentage is initiated as a field, where the values are changed to the upgraded value when the upgrade method is called. Single Upgrade is set to true for these 2 items as they can only be upgraded once. The value is passed into the action where the upgrade capability is removed after one upgrade. The selection to upgrade the item is not included in the actionlist in getItems() if the capability to upgrade is removed, thus it cannot be upgraded.

Broadsword upgrade:
Implements the Upgradable interface where the price and singleUpgrade is defined. The implementation of the Broadsword's upgrade is to override the damage method in the parent WeaponItem to calculate the final damage output of the requirement. The initial damage is stored as a static variable, hence we just multiply the initial damage value with the multiplier if there is a change in damage which indicates that FocusAction has been executed on the

Broadsword. The additional damage is added back into the multiplied initial damage to produce the final damage.

However, the previous implementation of our focusAction takes advantage of the WeaponItem class (all weapons would extend from WeaponItem) with the intention of extensibility where the focusAction can be implemented into different weapons in the game without storing the instance of that weaponItem in the focusAction class itself, just pass the item in the constructor as a WeaponItem (Example, we want GreatKnife to have FocusAction in the future). However, we soon realised that when we call the increaseDamageMultiplier in our focus action when the action is executed, the damage multiplier would only be updated in the WeaponItem's class, where the field is stored as a private variable, so that means that our broadsword instance's damage multiplier is unchanged. The easy workaround to that would be to use a getter, but WeaponItem is an engine class so we are not able to modify the given code. In this case, we have to modify our focusAction to take an instance of the Broadsword, and in our broadsword class, we override the parent (WeaponItem) updateDamageMultiplier and increaseDamageMultiplier methods so that the Broadsword damageMultiplier field is updated to achieve the correct results when doing calculation for the final damage in the damage method.

Great Knife upgrade:
Implements the Upgradeable interface to include the methods of an upgrade. The price and singleUpgrade is defined. The upgrade method utilises the increaseHitRate method of WeaponItem (parent class) to increase the hit rate value whenever the Great Knife is upgraded.

Pros:

Upgradable is used to adhere to Interface segregation principle where items that cannot be upgraded just doesn't have to implement the interface, like Giant Hammer, so there would not be any unimplemented methods in the child class.

UpgradeAction is a general action that takes the item to be upgraded, it allows for single responsibility principle as the upgrade method of each item is defined by the item itself, the action only calls the method to be executed as well as deducting the player's balance.

Cons:

If you forgot to add ability, you would not be able to upgrade the item. Human error might occur and takes time to debug.

Extendability:

Implementing Upgradable as an interface allows for future extendability of items that can be upgraded, just implement the interface to an item that can be upgraded.