**Assignment 3 Design Rationale (REQ 3, 5)**

**By Jenul Ferdinand**

For this assignment, I worked on requirement 3 and 5. I found it easy to implement the mandatory features: FingerReaderEnia, GoldenRune, and GoldenSeed which was my choice for the creative requirement. FingerReaderEnia is a new trader in the game, which was the first to accept another item as transaction instead of runes. GoldenRune was easy to implement, since our first implementation of Rune was basically how GoldenRune was meant to work. GoldenSeed was a little bit challenging to implement since at first, I thought to access the attributes of the other class (FlaskOfCrimsonTears), but instead I ended up creating a new interface (Enhanceable). In this interface I had a method enchance() which all implementations must use. With the enhance method I can enhance the capacity of the FlaskOfCrimsonTears.

**Changes Made from Last Assignment**

The changes between this assignment and the last are listed below:

* Trader abstract class was implemented to keep code DRY, and this also supports the open/closed principle because I’m now allowing new trader types to be added in the future without modifying the existing Trader class. This also follows the Liskov Substitution Principle, anywhere a Trader object can be used a MerchantKale and FingerReaderEnia object can be used aswell.
* A new way to keep track of how many runes an Actor has. Before we could only store the runes in the Player as an attribute. But now I added the RuneManager singleton class which uses a HashMap with the Actor as the key and an Integer as the value to store the runes that actor has. This class also comes with various helper methods to help us manage the runes of Actors when we need to. Especially in the PurchaseAction or SellAction. By creating this RuneManager singleton class to manage the runes for all Actor instances, the responsibility of managing the runes from each Actor has been seperated, this gives the RuneManager a single responsibility. Also, since the PurchaseAction and SellAction depend on the abstraction (RuneManager) instead of directly the details of how the runes are stored inside an Actor or Player, this follows DIP. PurchaseAction and SellAction are both depending on a higher-level module, thus decoupling them from the direct management of runes and decreasing the risk of changes in one module affecting the other.
* A lot of downcasting issues were solved by programming to an interface or using the capability functionality provided with the engine, which we looked over in the last assignment. I think this is a great improvement from last time.

**Pros and Cons of New Features**

**GoldenRune**

I did not want the Player to be able to consume certain Items while they were on the ground, I wanted it so that they could only consume it while it was in the inventory. This was the case for the items: FlaskOfCrimsonTears and GoldenRune. So I am now adding the ConsumeAction in the tick method instead of the Constructor.

* Pros:
  + The ConsumeAction for the GoldenRune will now only be added once, ensuring that it is not added repeatedly.
* Cons:
  + The ConsumeAction for the FlaskOfCrimsonTears will not appear in the first tick of the game, the player must move first for it to appear.

For the Rune, I want the Player to be able to consume it while it’s on the ground, but not be able to pick it up. So, I made it non-portable and added the ConsumeAction in the constructor.

* Pros:
  + Runes cannot be picked up anymore.
* Cons:
  + I actually like the idea of being able to carry around the Rune instead of consuming it, but now that GoldenRune is implemented, there is no point of that.

In the tick method of FlaskOfCrimsonTears and GoldenRune, I had to make sure that the ConsumeAction wasn’t going to keep adding on every tick. I prevented this by using a capability check, I created a new capability Status.IN\_INVENTORY. First, we will check if we don’t have the capability, then we will add the ConsumeAction and also add the capability. This will ensure that we don’t keep adding to ConsumeActions.

* Pros:
  + This will only be for the GoldenRune, since the FlaskOfCrimsonTears will not be able to drop.
* Cons:
  + The problem with capability checking is if the Player drops the item after picking it up initially, the Item will retain its ConsumeAction. To solve this, I am using the tick method to check if the item is on the ground, and checking if the item has the capability still, then I will remove the capability, and then I will remove the ConsumeAction.

**FingerReaderEnia**

Since FingerReaderEnia will be a new trader, I thought to create an abstract parent class named Trader. This class would manage storing items/weapons, selling, buying. So the child classes (MerchantKale and FingerReaderEnia) would only have to initialise their own inventory with items or weapons.

* Pros:
  + This allows for easy extensibility for new merchants in the future.
* Cons:
  + Since the Trader class does not have functionality for trading, I had to add some extra code to FingerReaderEnia to manage the trading. Since FingerReaderEnia is the only merchant as of now that does trades.

The Trader class will store Item and WeaponItem in two different HashMaps. I added this to prevent prior downcasting used in the PurchaseAction, SellAction and TradeAction. Now there will be two constructors in those action classes, the difference between the constructors will be their parameters, one will need Item and one will need WeaponItem.

* Pros:
  + Downcasting has been eradicated from PurchaseAction, SellAction and TradeAction.
  + Having separate HashMaps for Item and WeaponItem will ensure type safety without needing to downcast. Reducing the risk of a runtime ClassCastException.
  + The code becomes more readable and understandable, as it is now clear which type of object each HashMap is supposed to have.
  + The use of overloaded constructors in the actions classes allows me to handle both Item and WeaponItem types explicitly.
* Cons:
  + The Trader class will be large because of the methods required for managing both HashMaps.
  + Code duplication due to the dual collections of Item and WeaponItem, needed to implement similar logic for both classes.
  + Increased complexity

I had to create some extra classes to showcase the trading functionality of FingerReaderEnia. RemembranceOfTheGrafted, AxeOfGodrick, Grafted Dragon. If the RemembranceOfTheGrafted is in the Player’s inventory, the AxeOfGodrick and GraftedDragon will come up as TradeActions if the Player is in range of FingerReaderEnia’s exits.

* Pros:
  + Trading works with these items.
* Cons:
  + The new classes have no function since we chose not to implement Godrick the Grafted

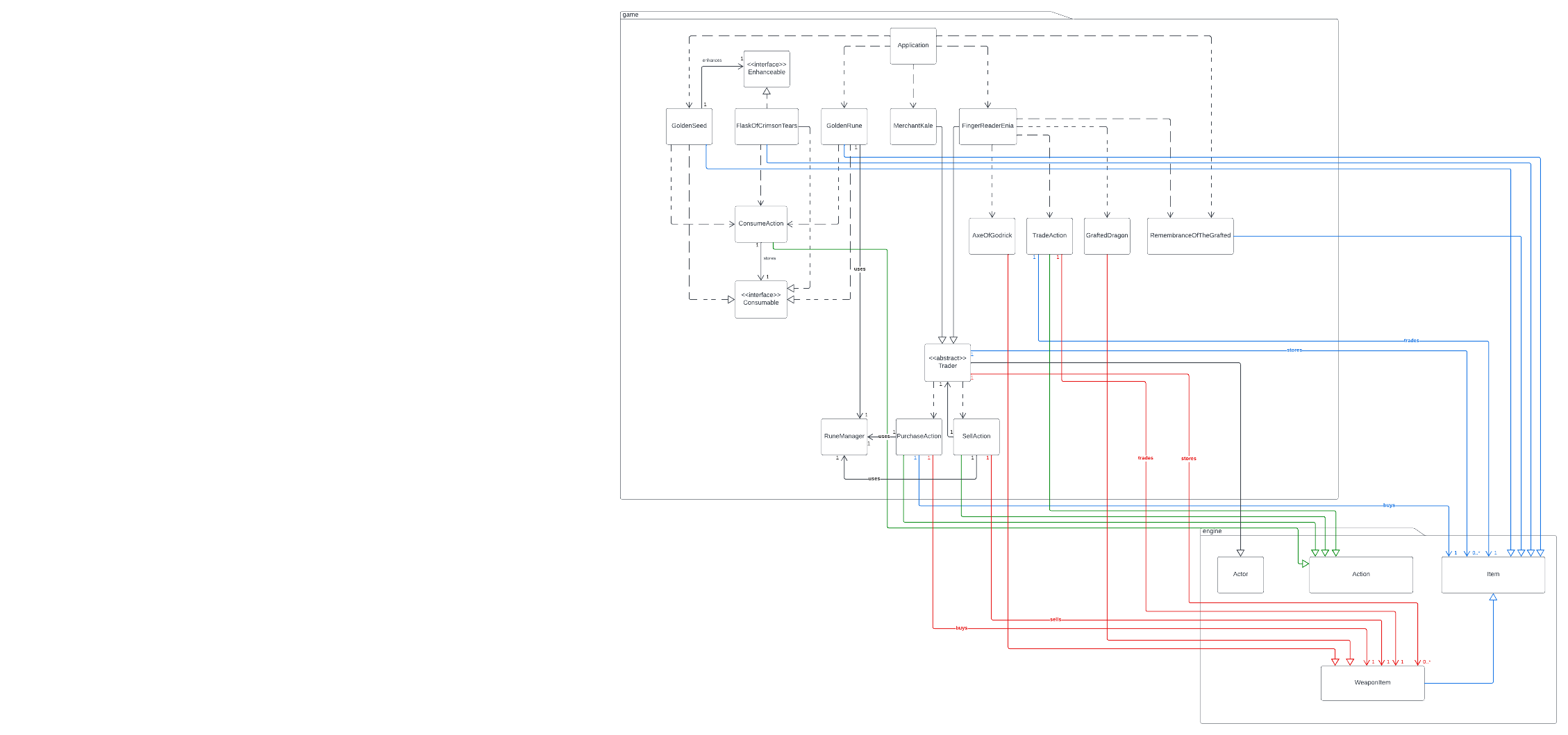
**Creative Requirement**

For the creative requirement, I will be implementing Golden Seeds. Golden Seeds will increase the maximum capacity of the FlaskOfCrimsonTears.

I had to use downcasting to an interface because there was no other way of accessing the data of the FlaskOfCrimsonTears. I created an interface, Enhanceable with method enhance(). The FlaskOfCrimsonTears will implement Enhanceable, and it’s method enhance(), inside the enhance() method is where it will increase its capacity.

* Pros:
  + By using interfaces, I’m enabling polymorphism. This means that I can refer to any object which implements the Enhanceable interface as an Enhanceable object, which allows me to call the enhance() method without knowing the exact type of the object.
  + This approach reduces coupling between different parts of the game. Other classes don’t need to know anything about FlaskOfCrimsonTears except that it’s Enhanceable.
  + Flexibility and scalability for future Enhanceable objects that could be added.
* Cons:
  + Had to downcast to interface, but I think this is the only way for this scenario.

**UML Diagram**

****

PDF of UML Diagram can be found at docs/Assignment 3 – UML Diagram (REQ 3, 5).