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, the trader differs from the base Trader, as it can accept an item as transaction instead of runes. GoldenRune was easy to implement, by leveraging our initial Rune implementation, I adapted it to create the GoldenRune, sticking closely to the original design principles.

Previously in the last Assignment, I made my own changes to the requirement of the Rune. I made it so that Rune was an Item that stored a value, and it could be picked up and put in the inventory. I thought that this was a better way of managing runes, but in this assignment, I had to change some things because of the new implementation of GoldenRune.

One thing I did not change was that the runes will not be automatically transferred to the Players balance once an Enemy is killed. Instead, the Enemy will drop a Rune on the last location before they died, and then the Player can then consume that Rune when standing on top of it. This approach increases realism, player agency, strategic depth, and conserves resources by not automatically updating a player's rune balance. Furthermore, it allows us to maintain a certain level of challenge and difficulty within the game by requiring the player to physically collect their rewards.

For the creative requirement, I implemented Golden Seeds, which enhances the capacity of FlaskOfCrimsonTears. The Enhanceable interface, which FlaskOfCrimsonTears implements, provides the enhance() method for increasing the capacity. Despite requiring downcasting to the interface, this approach enables polymorphism, reduces coupling, and allows for future expansion.

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:
    - Better control over game logic by adding the ConsumeAction in the tick() method. I can implement more complex game rules, such as checking whether the item is in the inventory before it can be consumed.
    - Easier bug tracking when the ConsumeAction is in the tick() method because it would be called every tick.
- 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.
    - Potential timing issues could occur since the ConsumeAction is called in the tick() method.
    - Increased complexity

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.
    - Simplicity, since the ConsumeAction is in the constructor, the action is set once when the Rune is created, simplifying the code.
    - This design is different from the other items since it cannot be picked up and put in the Actor’s inventory. It can only be consumed while the Actor is standing on top of it on the ground.
- 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.
    - Potential confusion, players might find it confusing that they can consume the Rune on the ground but not pick it up.

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:
    - Trader class has single responsibility.
    - The Trader class is open for extension but closed for modification. Any unique trading behaviours can be implemented in the subclasses without altering the Trader class.
    - MerchantKale and FingerReaderEnia are substitutable for their parent Trader class without causing issues.
    - The Trader class has method that all traders should have, and no trader is forced to depend on methods they do not use.
    - Depending on the abstraction (Trader)
- 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.
    - Overgeneralisation, since FingerReaderEnia have different ways of managing trading rules, I had to override one of the methods in Trader.

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.
    - Improved immersion
    - Increased gameplay dynamics, this setup introduces an extra layer of strategy and decision making to the game. Players would need to consider holding on to the RemembranceOfTheGrafted to trade for the weapons in the future.
- Cons:
    - The new classes have no function since we chose not to implement Godrick the Grafted
    - Added complexity but this is a Dark Souls game, so it doesn't matter.

Creative Requirement

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

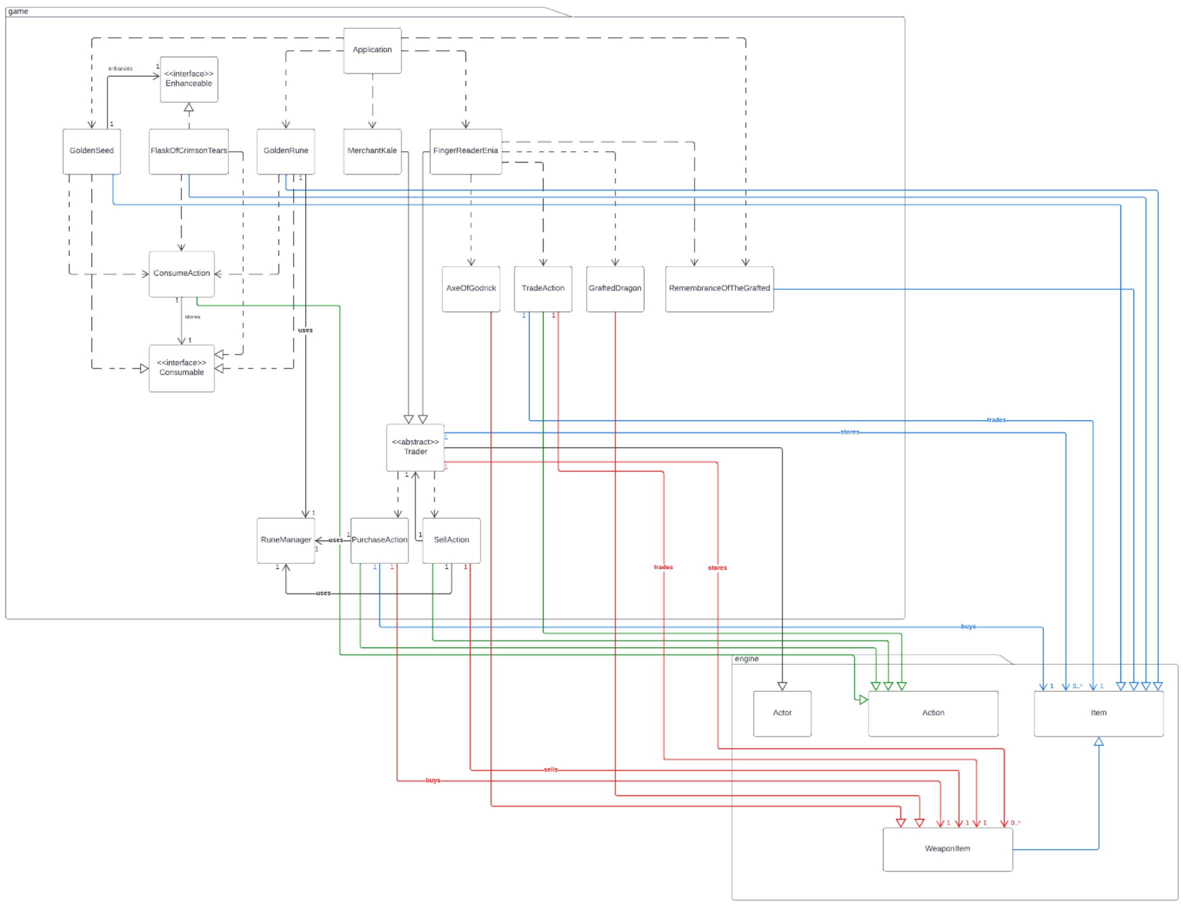
I had to use downcasting to an interface because there was no other way of accessing the data of the FlaskOfCrimsonTears. I created an interface, Enhanceable with method enhance(). The FlaskOfCrimsonTears will implement Enhanceable, and it's method enhance(), inside the enhance() method is where it will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

Overall, the implemented changes and additions should greatly enhance gameplay dynamics, despite the accompanying increased complexity. They provide an engaging and immersive experience, aligning with the challenging and strategic nature of Dark Souls games. Each design decision balances the need for complexity and realism against understandability and maintainability, striving to create a captivating yet manageable game.

UML Diagram



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