OOSE Assignment Report Table of Contents

[How Does my Design Work? 1](#_Toc41082811)

[What Design Patterns were used? 1](#_Toc41082812)

[How were they adapted to the situation at hand? 1](#_Toc41082813)

[What do your patterns accomplish? 1](#_Toc41082814)

[How do these patterns impact coupling, cohesion and reuse? 1](#_Toc41082815)

[How these patterns affect future extensibility of my program? 1](#_Toc41082816)

[Two Plausible Alternative Design Choices? 1](#_Toc41082817)

[Design Choice 1 1](#_Toc41082818)

[Design Choice 2 1](#_Toc41082819)

## How Does my Design Work?

### What Design Patterns were used?

For this problem I used 4 design patterns. Strategy, Observer, Factory, and the Decorator Pattern. I knew I would need Strategy Pattern from the start as Menu’s are an easy choice to use Strategy. Since strategy pattern focuses on when there is a repeat of logic. For example, a repeat of menu options. Observer was needed to be able to meet the requirement of displaying any changes to the Main Characters and Enemies health without constantly checking whether there has been a change. This is known as the push vs poll ideology. Where we could be polling the Main Character/Enemy object to see if there is a change but how would we know when to poll… A time limit? A condition… Once every 5 seconds. This could be unreliable and could be late if we were to do it based on time. Therefore, the more elegant solution was to do it via pushing. Which meant that when a change would occur it would notify those that needed to know. The main character would be able to choose items from the shop and battle enemies during runtime. Therefore, it would not be suitable to not use Factory Pattern. I had a factory to create enemies on demand and the program would be none the wiser to which enemy or item was created. Decorator was used to wrap the weapons with functionality.

### How were they adapted to the situation at hand?

I knew I was going to need a Menu from the start so Strategy pattern was an obvious choice for this as we would only be having a single level menu. I created a Selection Interface which every possible menu selection would be implementing or extending. Each selection or strategy would implement a doOption method and have their own algorithmic logic at hand.

### What do your patterns accomplish?

### How do these patterns impact coupling, cohesion, and reuse?

Strategy:

1. Coupling

The strategy pattern decouples the client from knowing about the data that the algorithms would use. For example in my case, my menu controller doesn’t know what happens internally for each of the menu options and only controls which one gets selected. This means the client couples itself to the interface which means if the algorithm that one of the strategies uses changes the client won’t experience any drastic problems when the subclass implementation changes. This may not remove coupling but is minimizes coupling.

Having this lack of knowledge about the Observer/Observable is the core component of decoupling components.

1. Cohesion

Strategy Pattern is great for Cohesion. Rather than having a class that has each method for the algorithm. You can use the strategy pattern to separate each Algorithm into their own classes/object which means they no longer are coupled together. These objects are treated separately and can be interchanged at runtime depending on the users whim.

1. Reuse

Strategy Pattern contains only one reuse between them and that is the interface.

Observer

1. Coupling

The observer pattern came about as we needed a way to allow objects to communicate without strongly coupling them together. They are still coupled together through what known as Abstract coupling but not as tightly. By using the Observer Pattern the Subject doesn’t need to know about the Concrete class just the Interface. And it no longer explicitly calls the Observer. Observer doesn’t check the observers state for a change it receives the update instead. And the observable doesn’t know the concrete type of the Observer.

1. Cohesion

Allows for the observer and the subject to have specific purpose. For example the subject’s purpose is to notify the observer when an event occurs and the observer is responsible for reacting to that update. So rather than the Observer having to check if the state has changed and then react. There is no longer a need to check if a change has occurred and the observer can wait unaware of what is going on in the subject until and notification occurs.

1. Reuse

Factory

1. Coupling
2. Cohesion
3. Reuse

Decorator

1. Coupling
2. Cohesion
3. Reuse

### How these patterns affect future extensibility of my program?

## Two Plausible Alternative Design Choices?

### Design Choice 1

Advantages:

Disadvantages:

### Design Choice 2

Advantages:

Disadvantages: