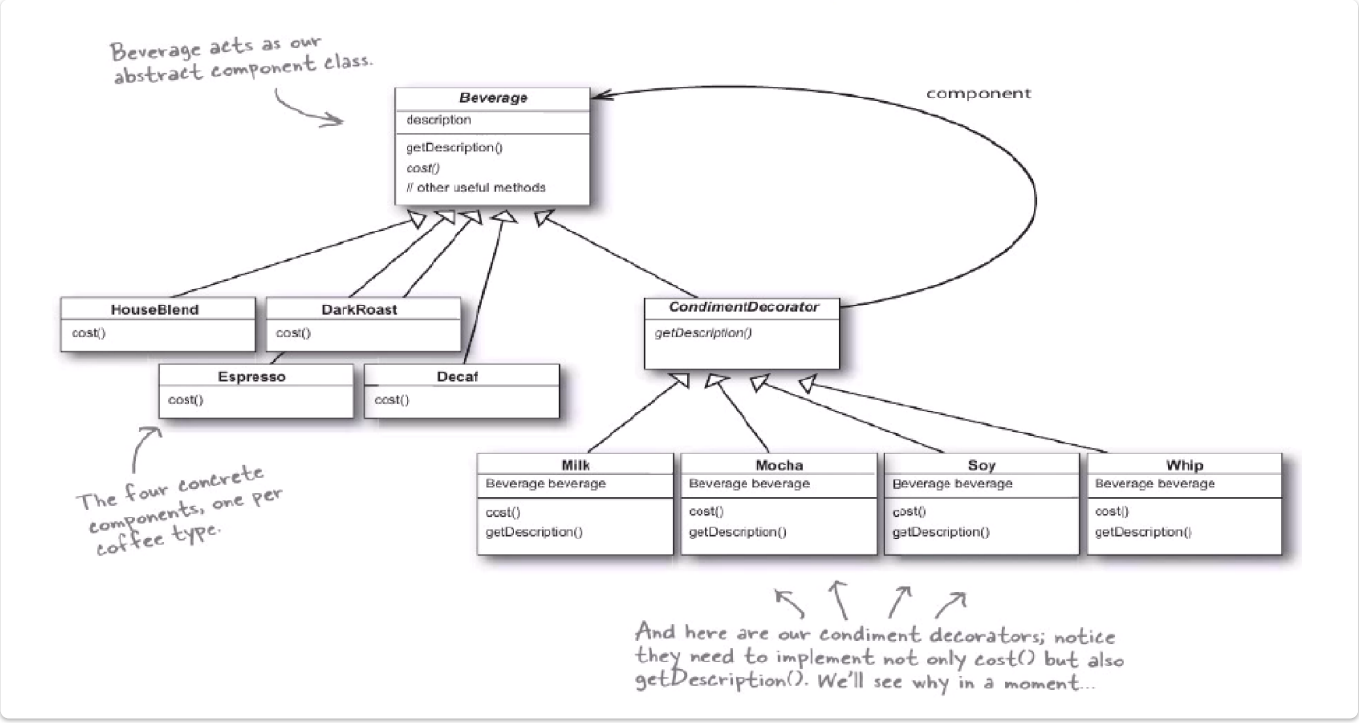
Decorator Pattern

결론 및 정리

정의

Decorator Pattern은 Structual Pattern 중 하나로, **객체의 기능을 동적으로 추가하거나 확장할 수 있게 해주는 패턴**

- ConcreteComponent : 기본적인 기능 (Method 1, Method 2, ...) 들이 작성되어 있는 Class.
- Component : ConcreteComponent 아 Decorator 가 Implements 하거나 Inheritance 해야 하는, Interface 혹은 Abstract Class.
  ConcreteComponent 와 Decorator 가 같은 변수로서 취급될 수 있도록 Polymorphism 을 제공하는 역할을 한다.
- Decorator : Component 를 Implements / Inheritance 한 Abstract, Interface Class.
  Component 타입의 변수를 가지고 있으며, ConcreteComponent 를 그 안에 가지고 있다. 이를 통해 ConcreteComponent 를 wrap 하여 기능을 추가하고 확장하는 역할을 한다.
  - Decorator 가 Component 를 상속/구현 하는 이유 : Decorator 가 Decorator 들을 Wrap (Component 타입의 변수에 넣어) 할 수 있어야 하기 때문에.
- ConcreteDecorator : Decorator 를 상속받아, 실제로 위에서 언급한 기능을 수행한다.



커피로 예를 들어서 생각해보자.

- Component = Beverage
- ConcreteComponent = DarkRoast ...
- Decorator = CondimentDecorator
- ConcreteDecorator = Milk / Mocha ...

구현

1. 기능에 대한 Placeholder 가 존재하는 Component Interface / Abstract Method 를 만든다.
2. Component 를 상속/구현 하면서, Component Type 을 변수로서 가지고 있는 Decorator Class 를 만든다.
3. 기본 기능을 담고 있는 Concrete Component, 추가 기능으로서 적용되는 Concrete Decorator 를 만든다.
4. Concrete Component 를 Concrete Decorate 로 Wrap 하여 기능을 추가하는 방식으로 사용한다.

Context & Problem & Merit

- ConcreteComponent는 기본 기능만을 제공한다.
- Decorator는 Component를 참조하며, 필요한 추가 기능을 확장할 수 있다.
- 사용자는 ConcreteDecorator를 사용하여 ConcreteComponent에 원하는 기능을 추가 / 확장할 수 있다.

1. 기존 코드 수정 없이 객체에 새로운 기능을 추가할 수 있다.
2. Single Responsibility Principle (단일 책임 원칙)을 준수하게 해준다. 각 데코레이터는 하나의 책임만을 가진다.
3. 데코레이터를 조합하여 다양한 기능의 객체를 생성할 수 있다.

Design Pattern Principles

Design Pattern Principle 6 - "Open - Closed Principle"

- 새로운 기능으로의 확장은 가능해야 한다.
- 기존 코드의 수정은 지양해야 한다.
- ex - Singleton 에서, 기능을 상속 받아 manager class 로 만들면, 거기서 또 다른 내용의 추가 없이도 Singleton 의 기능을 사용할 수 있으면서, 기능 자체의 확장은 가능함.
- 내 이해 조금 첨가 : 기능적으로 독립되어 있어야 한다. 확장하여 다른 기능을 쉽게 덧댈 수 있도록 만들어야 한다. 기존 기능을 수정하는 방향성의 경우, 지양해야 한다.

강의 들으며

GPT 기준으로 한번 요약

정의: Decorator 패턴은 객체의 기능을 동적으로 확장하는 구조적 디자인 패턴입니다. 이 패턴은 기본 객체에 추가적인 기능을 런타임에 유연하게 추가할 수 있도록 합니다. 즉, 기본 객체를 수정하지 않고도 기능을 추가하거나 변경할 수 있습니다. => 어떻게 Runtime 에 객체의 기능을 동적으로 확장할 수 있다는 말인가?

주요 구성 요소로는 다음과 같다.

- Component: 기본 객체를 나타내는 인터페이스 또는 추상 클래스입니다. 기본 기능을 정의합니다.
- ConcreteComponent: Component를 구현하는 실제 객체입니다.
- Decorator: Component와 동일한 인터페이스를 구현하는 래퍼(Wrapper) 클래스로, 추가적인 기능을 제공합니다. 또한 내부에 Component 객체를 포함합니다.
- ConcreteDecorator: Decorator를 상속받아 특정 기능을 구현하는 클래스입니다.

Decorator 패턴은 객체의 기능을 계층적으로 쌓을 필요가 있는 경우에 사용됩니다. 기본 객체에 여러 개의 확장 기능을 적용하거나, 적절한 기능을 선택하여 객체를 동적으로 확장하고 싶은 경우에 유용합니다. 또한 상속을 통한 기능 확장보다 유연한 방법을 제공하므로, 객체에 대한 수정 없이도 새로운 기능을 추가할 수 있습니다.

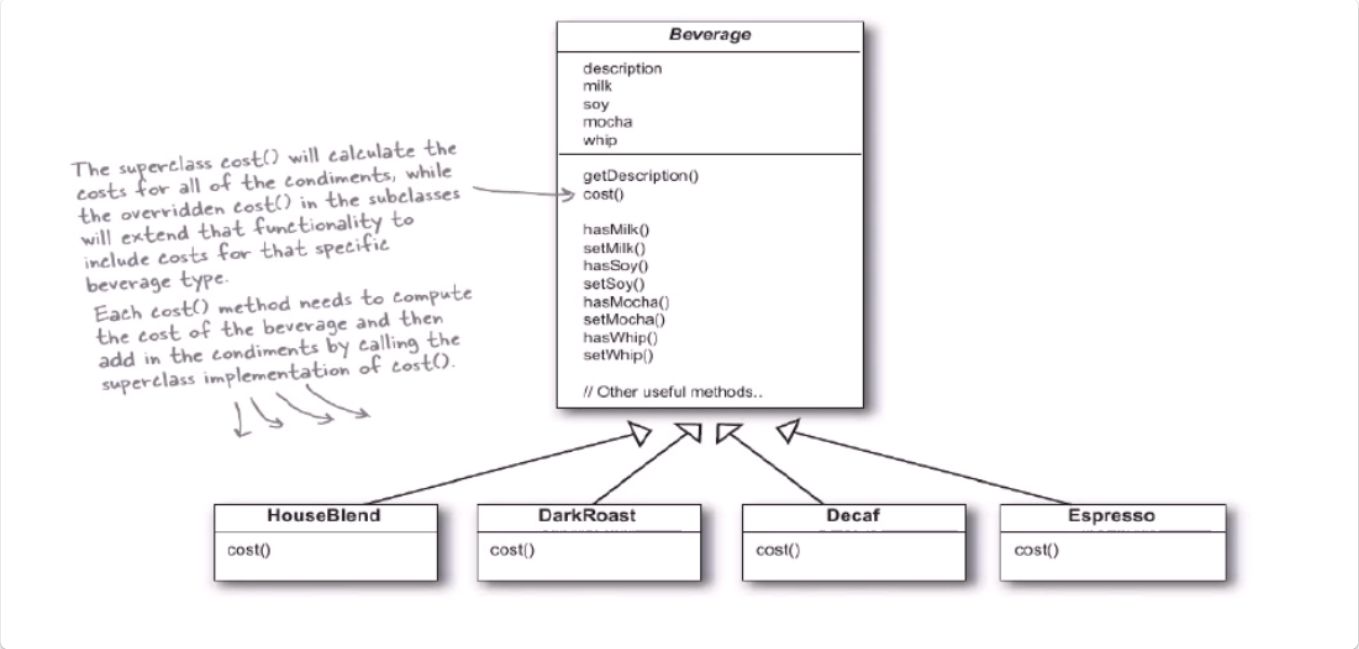
강의 들으며

(Structural Pattern 임.)

추가 기능들을 계층적으로 점점 쌓을 수 있다고 하심.

알림을 이메일 만으로 보내다가, 다른 추가 되는 것들이 더 많아지게 된다면?
각각에 대한 내용을 상속 받으면 될지 몰름. => 하지만, 그것들의 조합으로 보내야 하는 경우가 생기게 되면, 그 조합의 개수만큼 Class 를 생성해야 하므로 복잡해지게 됨.
(Application Class 가 Notifier Class 를 변수로서 가지고. 그 Notifier 변수를 통해 호출하는 Send() 가 실제 Notify 를 보낸다고 가정할 때, 그렇게 작용하게 된다는 뜻.)

이런 경우도 Decorator Pattern 이 필요한 예시로서 들 수 있음.



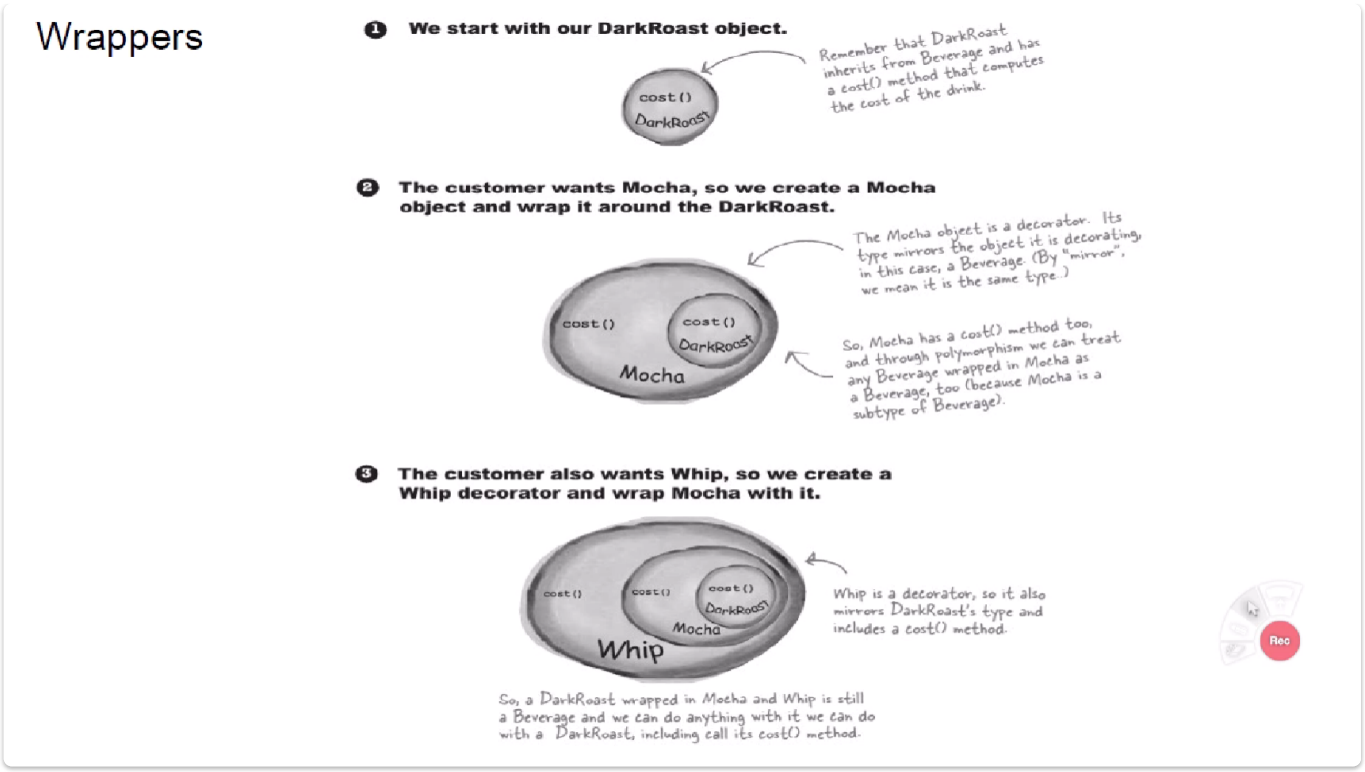
각각에 해당 하는 내용들이 다 독자적으로 기능한다면 모르겠지만, Espresso 의 경우, hasMilk, hasSoy() 등등등, super class 에 모든 내용을 넣게 되면 버려지는 method 들이 너무 많음.
또한, cost() 등의 method 가 super class 의 내용을 아예 override 하는 것이 아니라. superclass의 내용을 기반으로, 혹은 다른 class 들 안에 든 내용을 기반으로 하여 새롭게 만들고 싶은 경우는 어떻게 하나?

위와 같은 경우 다음과 같은 문제가 발생함을 문제 삼을 수 있음.

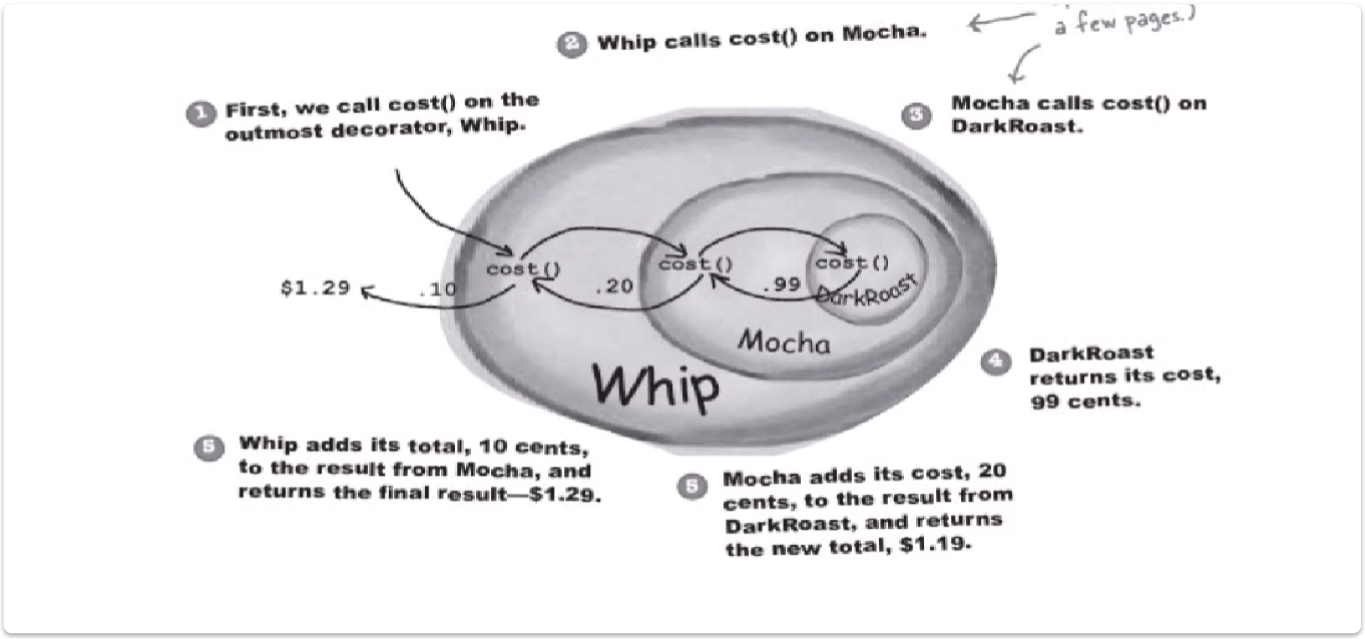
1. 새로운 variable 등이 추가되거나 할 때, 우리는 새로운 method 를 더하고, 이를 sub class 상에서 구현하여야 함.
2. 쓰지 않는 게 상속 될 때도 있음! (사용하지 않는 Variable 이나 Method 도 있을 수 있기 때문에,)
3. Dobule 을 원할 때는 이걸 Double 로 할 수가 없음.

> *Design Principle 6 : Class 는 extension을 위해선 열려있으나, 수정에 대해선 닫혀있어야 함.*
>
> - *Open - Closed Principle*
> - *새 행동으로 확장하는 것은 가능하도록 열려 있어야 하나, 기존 코드를 수정하는 방향성의 경우, 막혀 있어야 한다.*
> - *우리가 Singleton 의 기능을 상속받는 manager class 를 만들면, 그를 수정할 필요 없이 자동으로 signleton 의 역할을 수행할 수 있도록 만들어진다고 생각하면 편하다.*

이 커피와 Decorator Pattern 에 대해 배워보자.
Mocha 와 Whip 이 더해진 DarkRoast 커피를 만든다고 생각해보자.
DarkRoast 커피를 마련하고, 이걸 Mocha Object 로 wrap 하고, Whip Object 로 wrap 하고.



이런 Wrap 의 형식이라고 생각하면 된다.
맨 마지막의 경우에는, 계산할 때, 안에 담겨 있는 친구를 이용하여 Cost 를 계산하는 형태로 만들면, cost 를 마지막 부분까지 충분히 계산할 수 있다.



Decorator 는 object 에 추가적인 동작을 붙인다. inheritance 해서 기능적인 요소를 추가하는 것보다, 유연하게 기능적인 요소를 추가하는 것을 가능하도록 한다.
순서대로 Behavoir 를 실행해야 하는 경우에도 용이하게 사용할 수 있다.

대강 wrap 하여 기능적인 요소를 추가하여 구현한다는 것은 알겠는데, 이걸 실제로 어떻게 구현하는가?

이후에 실습 관련 내용이 쭉 나옴.

아 뭔가 이해가 될 것 같다!
커피에 대응시켜서 이야기 해보자.
Component Interface 가 실제로 마실 것들에 대한 Interface.
마실 음료들은 전부 해당 Interface 의 상속을 받음.
재료가 될 Decorator 의 경우에는, 해당 Component Interface 를 상속 받는 Interface 임.
Component 를 변수로서도 가지고 있음. 왜? 이 Decorator 의 경우, 실제 "음료"나 "Decorator 에 의해 이미 Wrap 된 음료" 를 변수로서 가지고 연산에 사용함으로써 "감쌀" 것이기 때문에.
Decorator 가 Decorator 스스로를 wrap 해야 되기 때문에, 왜 Decorator abstract class 가 Component Interface 를 implements 하는지 이해가 되네.