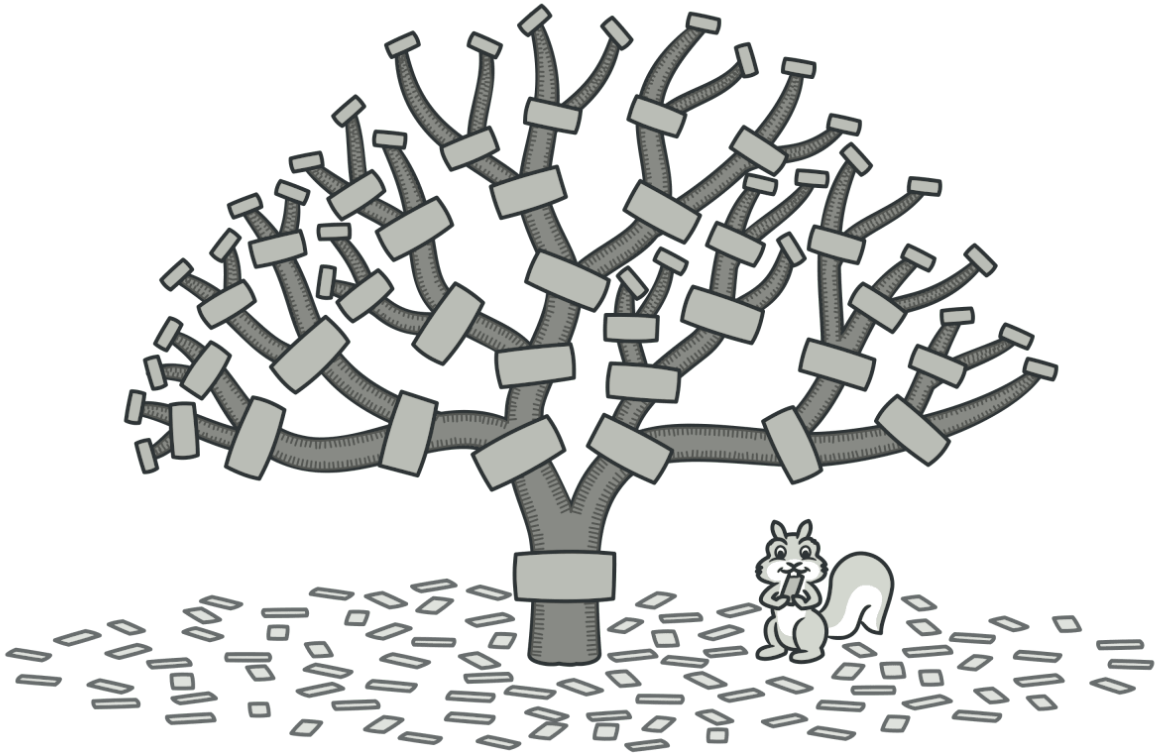
참고 자료

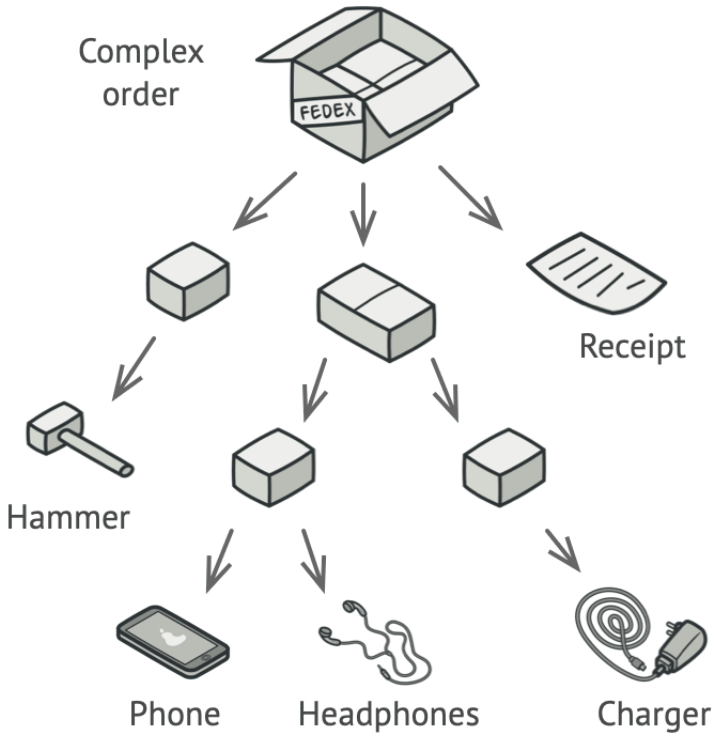
- https://refactoring.guru/design-patterns/composite
- https://en.wikipedia.org/wiki/Composite_pattern

CompositePattern이란?

- structure design pattern
- object들을 트리 구조로 구성하도록 하고 (**part-whole hierarchies**)
- 이 object를 마치 개개의 object들인 것 처럼 다루게 하는 패턴입니다.



상황



- 주문 시스템을 생각해봅시다. (ordering system)
- 박스와 상품이라는 2개의 타입을 가진 오브젝트가 있습니다.
- 큰 박스 안에는 좀 더 작은 박스가 들어있을 수 있고, 상품이 들어있을 수 있습니다.
- 주문은 박스로 포장되지 않은 **상품**일 수도 있으며, 상품이나 박스들을 포장하고 있는 **박스**일 수도 있습니다.

목표

주문된 특정 박스 또는 특정 상품 가격을 구해봅시다!

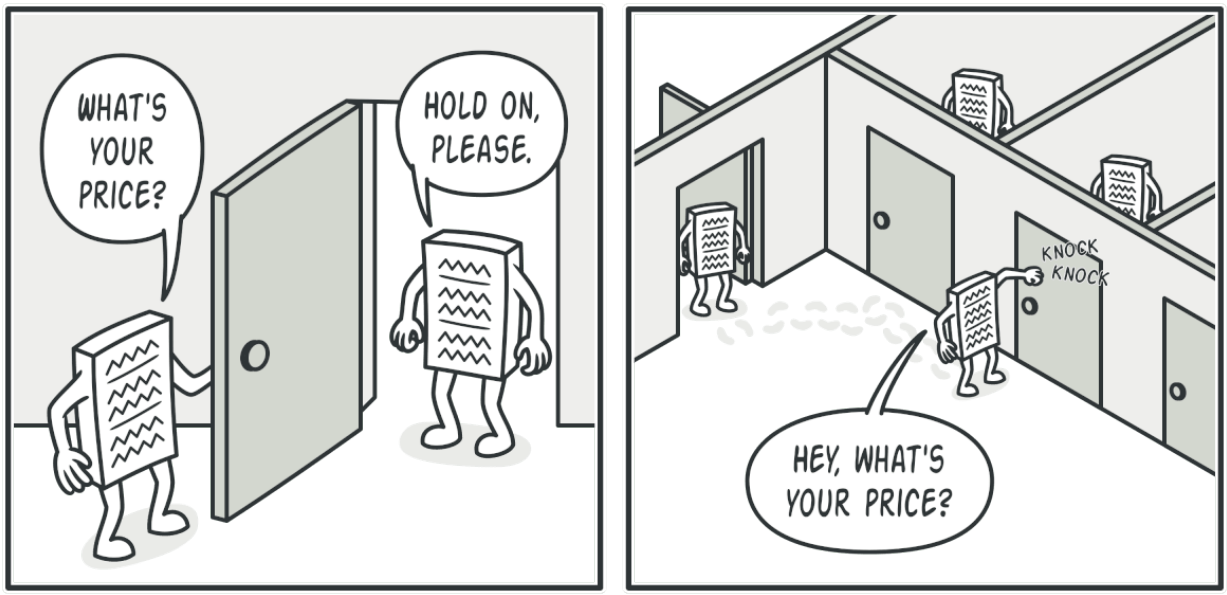
단순히 다 포장을 까서 아이템을 확인하면 어떨까?

코드는 현실만큼 간단하지 않습니다.

1. 매 object마다 이 object의 타입이 Products class타입인지, Boxses 타입인지 확인해야 합니다.
2. 그리고 각 object별 level도 확인해야 합니다. (특정 박스 하위에 있는 아이템들을 알아야 하므로)

이런 방법은 사이즈가 커진다면 거의 불가능한 방법이 됩니다.

해결 방법



답은 Composite pattern!

point : app의 핵심 모델인 상품과 박스의 관계가 tree구조를 이루고 있고, 상위 레벨의 박스가 하위 레벨의 박스 및 상품들(child node)을 포함하고 있습니다.

1. 아이템의 가격을 계산해주는 method를 추상화 해줍니다.

   이렇게 한다면, concrete class인 products나 Boxses가 와도 동일한 메서드를 수행할 수 있게 됩니다.

2. method의 내용은 다음과 같습니다.
   1. item일 경우

      단순히 아이템의 가격을 반환해줍니다.

   2. box일 경우

      box에 속해있는 아이템들에 대해 가격을 묻고(위임한다) 이를 합산하여, 이 box의 가격을 반환해줍니다.

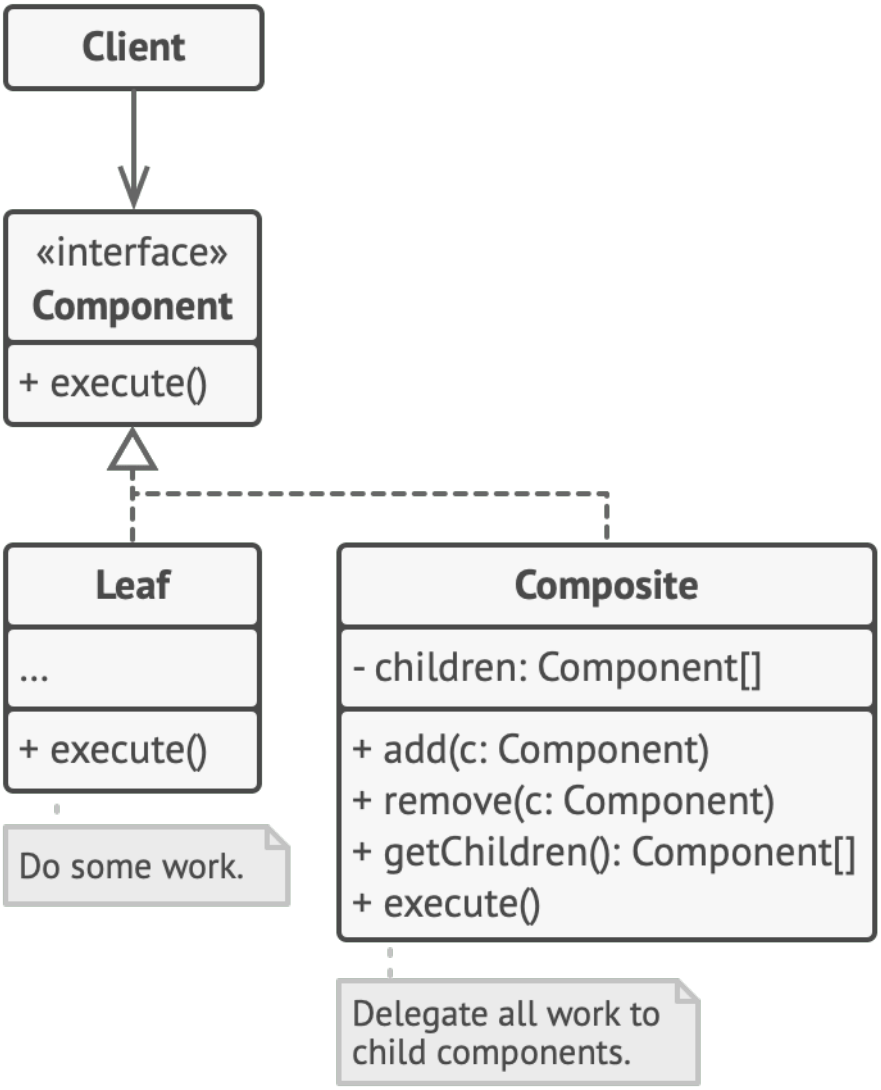
      따라서 box에 속해있는 더 작은 박스가 있을 경우 이 박스 또한 recursive하게 위 연산을 수행합니다.

가장 큰 이점

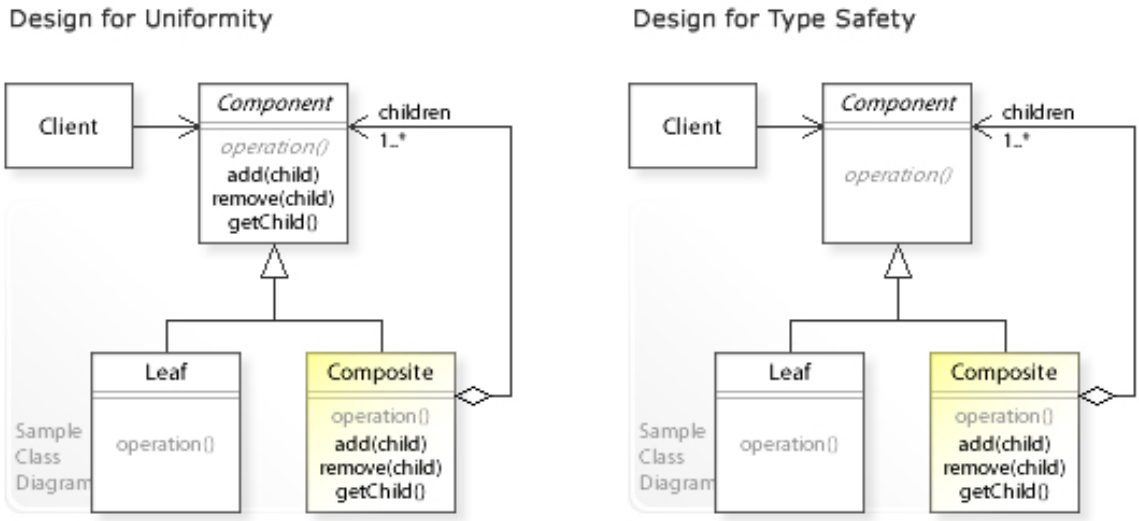
1. tree를 구성하는 concrete 클래스가 무엇인지 신경쓰지 않고 가격을 구할 수 있습니다.

   다시 말해 interface에 의존하기 때문에 이게 아이템인지 아주 복잡한 box인지 신경쓰지 않아도 된다는 의미입니다.

Structure



1. Component (요소, 부품)



> guru에서 따로 언급하는 내용을 위키에서 설명해주고 있어 위 사진을 추가하였습니다.

구현 1 Uniformity를 위한 디자인

simple하거나, complex element가 수행해야할 operation 메서드만을 가지고 있습니다. 이와 같이 구현하면 모든 element들이 이 interface와 동일하게 작동한다고 생각할 수 있습니다.

하지만, 이와 같이 구현하면 Leaf노드는 자식을 가지고 있지 않음에도 자식을 추가하고 지우는 등의 메서드를 가지고 있게 됩니다.

구현 2 Type Safety를 위한 디자인

Leaf노드가 Composite만의 연산을 수행하는 것을 막기 위해 Component에는 공통의 메서드만 넣어줍니다. 그리고 Composite에 composite만의 메서드를 추가해주는 디자인이지요.

하지만 이와 같이 구현하면 반대로 Type Safety를 가지나, Uniformity는 잃는다 보시면 됩니다.

2. Leaf

- 리프 노드는 basic element에 속합니다. 서브 엘레먼트를 가지고 있지 않지요.

3. Composite(aka. Container)

- 리프노드 또는 container를 서브 엘레먼트로 지니고 있습니다.
- 이 container는 서브 엘레먼트의 concrete class를 알지 못합니다. Component를 통해서 알고 있지요
- 요청을 받으면, 서브 엘레먼트들에게 요청을 전달하고,중간 결과를 처리한다음에 그 최종 결과를 client에게 반환해줍니다.

4.Client

- Component를 통해 element들과 커뮤니케이션을 합니다. 어떤 element가 오더라도 동일한 방식으로 처리할 수 있습니다.

간단정리

> Component는 Leaf, Composite를 일반화시켜 부르는 용어이고,
>
> Component를 분류하면 leaf와 Composite로 된다는 걸 알 수 있습니다.
>
> 그 Composite는 또 다른 Composite와 leaf를 구성할 수 있지요.

장,단점

장점

1. 복잡한 tree 구조를 편하게 다룰 수 있게 됩니다.
2. 새로운 element type을 추가하더라도 Component라는 추상 클래스에 의존하기 때문에 기존의 코드를 수정할 필요가 없습니다.

단점

1. 서로 다른 요소에 대해 공통의 interface를 적용하기 어려울 수 있습니다. 따라서 component interface를 over generalize할 여지가 있기 때문에 추후 이해하기 어려워질 수 있습니다.