2. 데코레이터 패턴

(1) 정의 & 장점

"객체를 동적으로 서브 클래스를 이용해 확장한다"로 정의된다. 데코레이터 패턴은 추가적인 기능을 넣고자 하는 객체의 추상 객체를 상속받은 데코레이터 추상 객체를 만든다. 추가적인 기능을 담을 틀을 만드는 것이다. 여기에다 필요한 기능을 구현하여 클래스를 추가하는 것으로 객체에 새로운 기능을 동적으로 부여할 수 있다. 기존 코드를 수정할 필요 없이 필요한 데코레이터를 적용하는 식으로 확장이 이루어지기 때문이다.

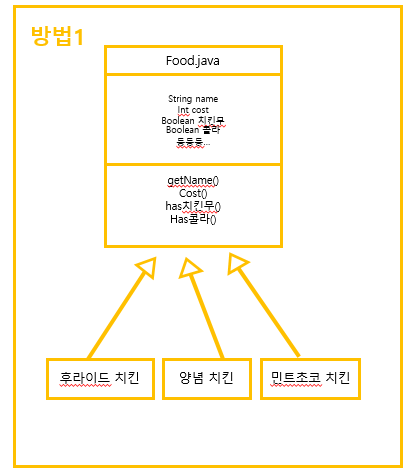
(동적 = Main메소드(=사용단)에서 기능을 넣었다 뺐다 할 수 있음. 정적에서는 기능이 전부 정의되어 있어 명령한 것 밖에 실행이 안됨.(=조합을 할 수 없음)

(2)예제 (1에서의 내용을 증명)

팩토리 메소드때와 마찬가지로 주문에 따라 해당되는 음식 객체를 내오는 프로그램을 작성할 것이다. 그러나 이번에는 치킨무, 음료와 같은 추가 메뉴가 들어가 있다. 추가 메뉴가 있을 때의 분기를 감안하여 코드를 작성해야 하는 것이다. 게다가 음료를 2개 이상 주문한다거나 여러 추가 메뉴를 같이 준비하는 경우도 감안해야 한다. 요약하자면 다음과 같다.

|  |
| --- |
| -음식 객체에는 양념치킨, 후라이드 치킨, 민트초코 치킨 3개를 생성  -추가 메뉴를 넣을 수 있는 메소드 혹은 클래스 생성  (추가 메뉴는 중복과 복수 주문이 가능 )  -각 객체에 이름과 가격을 출력하는 메소드 생성 |

역시 OOP를 무시한 채로 코드를 구상해보자. 생각해봄직한 해결 방안은 인스턴스 변수와 수퍼클래스 상속을 써서 추가 사항을 관리하는 것이다. 이를 다이어그램으로 표현하면 다음과 같다.



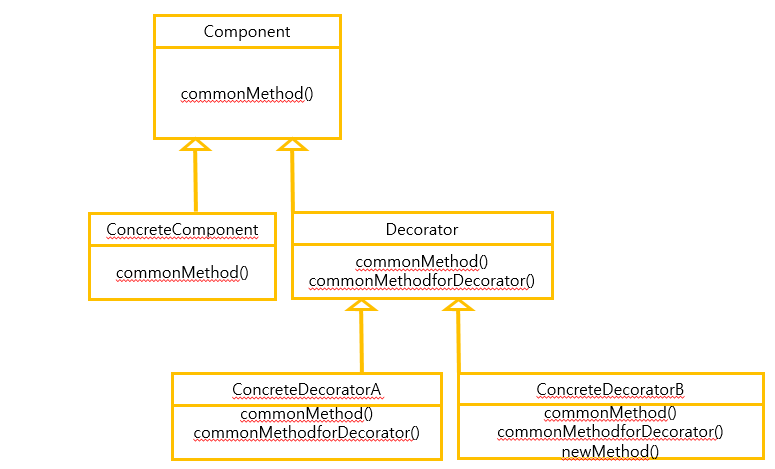
이름과 가격을 출력할 메소드를 작성하고, 거기에 추가 메뉴의 유무(boolean)에 따라 cost에 가격을 추가하고 이름에 추가 메뉴를 덧붙이는 has추가메뉴() 메소드를 만들었다. 중복이나 복수처리의 경우에도 이러한 메소드를 여러 번 사용하면 해결 가능하다. 실제로도 동작 자체는 정상적으로 할 가능성이 높다.

그러나 치명적인 단점들이 있다. 먼저 이 코드는 확장에 대해 닫혀있고 수정에 대해 열려있는 OCP원칙을 지키지 못한다. 새로운 메뉴를 추가하거나 가격을 수정하기 위해서 부모 클래스인 Food를 건드려야 하기 때문이다.

게다가, 추가 메뉴를 시키지 않은 객체들이 추가 메뉴를 생성하는 메소드를 상속받았다는 것도 문제다. 이 경우, 원하지 않은 메소드가 실행되거나, 특정 메소드만을 막기 위해 모든 객체를 수정하는 상황이 벌어질 여지가 있다. 모든 서브클래스에 모든 추가 메뉴가 필요한 것은 아니므로 여기서 상속을 사용하는 것은 올바른 해결책이 아니다.

이번에는 어느정도 객체지향적 요소를 가미하여 프로그램을 구상해보자. 바로 치킨과 추가메뉴를 조합한 모든 경우의 수를 자녀 클래스로 만드는 것이다. 후라이드 치킨만 시켰을 때, 후라이드와 치킨무를 시켰을 때, 후라이드와 치킨무 2개를 시켰을 때 등등… 주문이 생길때마다 클래스를 추가하는 것이다. 물론, 중복 코드를 전혀 제외하지 않았으므로 굉장히 비효율적이다.

문제 상황을 해결하기 위해서는 데코레이터 패턴을 사용해야 한다. 객체로 만들 구성요소 Component를 상속받는(행동을 물려받는게 아니라 인터페이스처럼 형식을 맞추는데 목적을 둠으로 구성이라고도 할 수 있다.) 데코레이터 추상 클래스를 만들고, 그것을 다시 상속받은 서브클래스에서 공통적으로 필요한 기능이 갖춰지거나 새로운 기능이 구현된다. 이를 구성요소의 구상 클래스에 동적으로 추가해 최종적으로 객체의 기능이 확장된다. 이를 UML로 옮기면 다음과 같다.



Component : 생성되는 객체가 모두 공유할 인터페이스 혹은 추상 클래스를 구현한다.

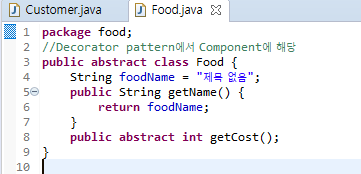
ConcreteComponent : Component의 구상클래스. 여기에 데코레이터가 추가된다. 직접 쓰일 수도 있고 데코레이터로 감싸져서 쓰일 수도 있다.

Decorator : 장식에 해당하는 클래스들이 공유할 같은 인터페이스 혹은 추상 클래스를 구현한다.

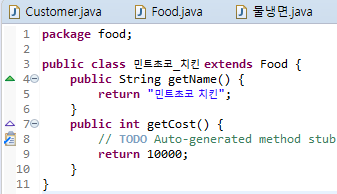
ConcreteDecorator : Decorator의 구상클래스. 공용 메소드를 상속받고 새로운 기능을 추가할 수도 있다.

실제로 UML을 따라 데코레이터 디자인을 구현해보자.

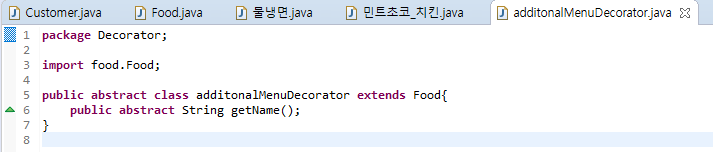
Component : Food.java



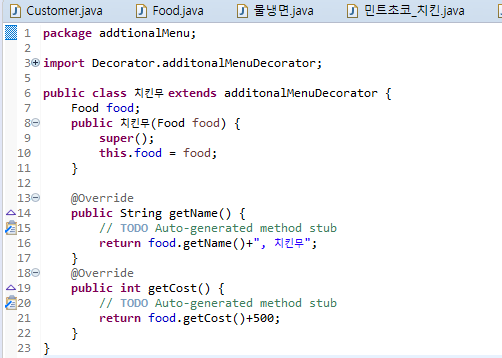
ComponentConcrete: 치킨.java x3



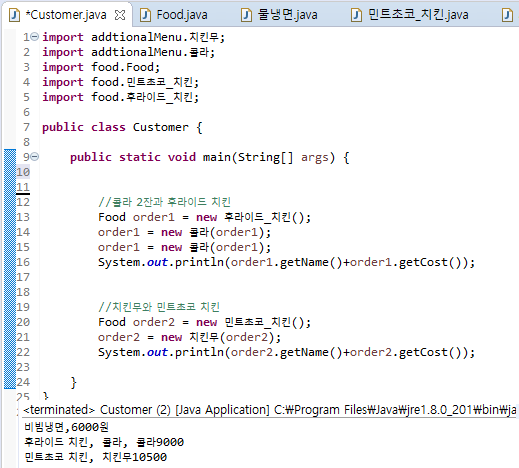
Decorator: additonalMenuDecorator.java



ConcreteDecorator: 치킨무.java



사용 및 결과 : Customer.java



메인 메뉴라고 할 수 있는 치킨 객체 뒤로 서브 메뉴가 추가 되었으며, 가격 또한 더해진 채로 출력된 것을 확인 할 수 있다. 무엇보다도 Main메뉴에서 유동적으로 필요한 객체를 생성할 수 있게 되었다.

또한 새로운 기능을 새로운 서브 메뉴를 추가하거나 내용을 수정하는데도 효율적이다. 필요한 기능을 담은 서브 클래스를 새로 작성하거나 기존 클래스를 상속받은 뒤 오버라이드 하는 식으로 수정 하면 되기 때문이다. 변경에 닫히고 확장에 열린 OCP가 구현된 것이다.

부록) cost 메소드에서 객체가 추가되는 과정은 다음과 같다.

