1. 팩토리 메소드 패턴

1)정의&장점

"객체 생성 처리를 서브 클래스로 분리해 처리하도록 캡슐화 하는 패턴"이 사전적 정의이다. 객체 생성하는 작업을 다른 클래스에 외주 준다고 이해할 수 있다. 팩토리 메소드 패턴에서는 객체 생성 처리를 해주는 new 키워드 업무를 Main클래스에서 하지 않고 factory클래스라는 서브 클래스가 담당하여 구상 클래스(Concrete class : 추상 클래스와 반대되는 개념. 실제 내용이 구현되어 있는 클래스를 가리킴. )에 대한 의존도를 줄인다. 객체 생성이 캡슐화 됨에 따라 자연히 인터페이스를 바탕으로 한 프로그래밍이 가능하므로 유연성과 확장성면에서 편리한 코드를 만들 수 있게 된다.

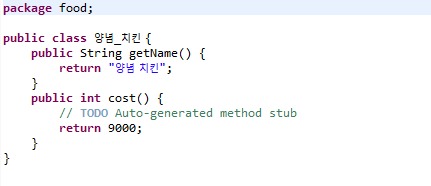
2)예제(1에서의 내용을 증명)

주문에 따라 주문한 음식에 해당되는 객체를 내오는 프로그램을 작성할 것이다. 디자인 패턴 적용 전후를 비교해서 패턴의 장점을 알아 보도록 한다.

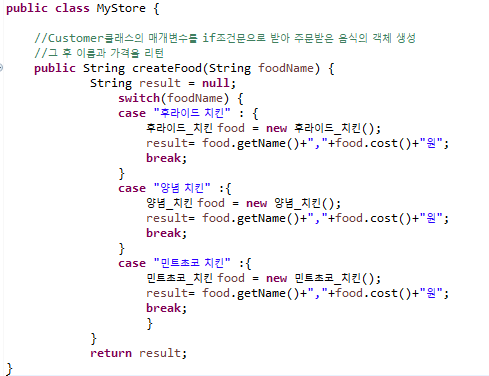
|  |
| --- |
| -음식 객체에는 양념치킨, 후라이드 치킨, 민트초코 치킨 3개를 생성  -각 객체에는 이름과 가격을 출력하는 메소드 생성  -main메소드에서 주문 내용에 따라 다른 객체를 생성하는 MyStore 클래스 생성 |

먼저 OOP를 무시한 채로 코드를 짜보자.

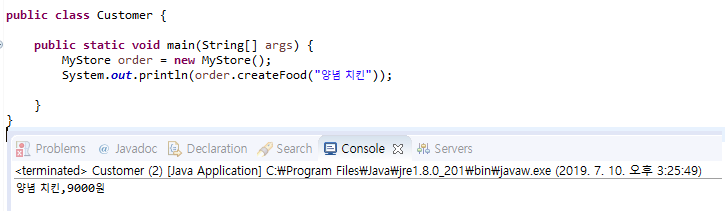
(치킨 객체 x3)



(MyStore 클래스)

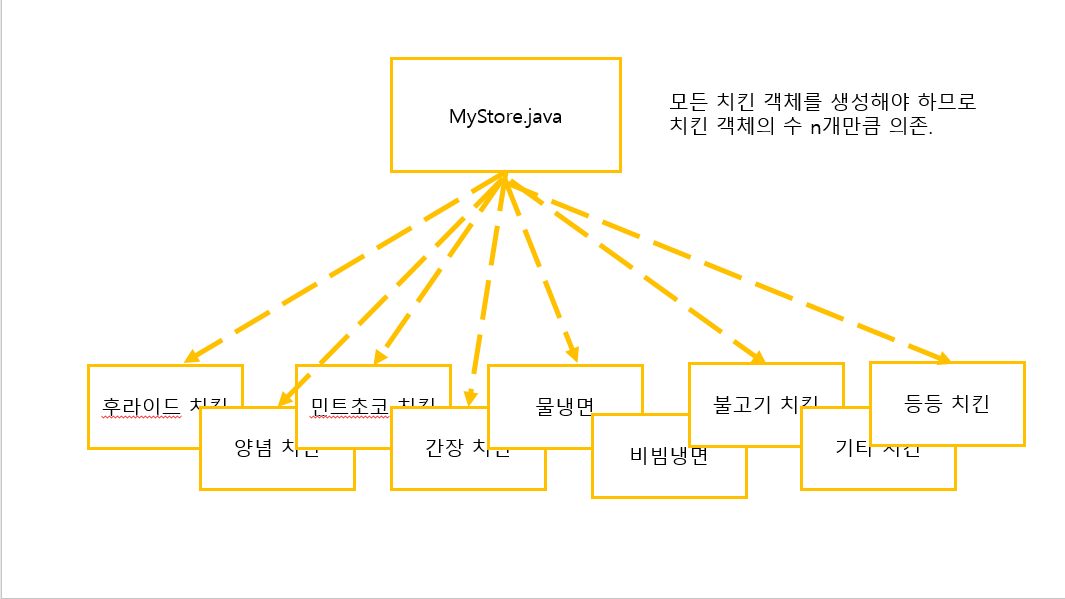


(주문하는 Custom 클래스 & 결과)



정상적으로 동작한다. 그러나 문제가 많다. 기능의 확장이나 수정에 애로사항이 생길 수 있기 때문이다.

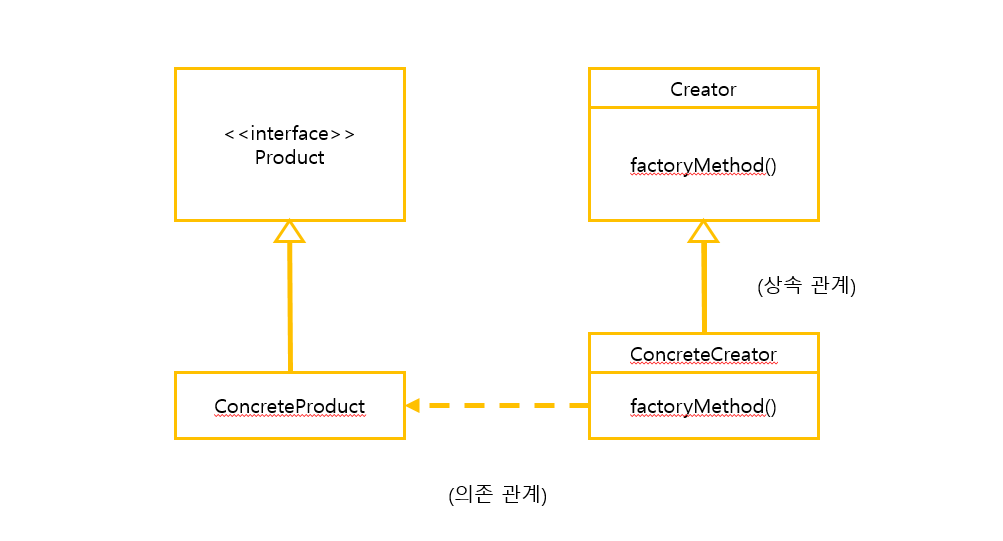
역시 예를 들어보자. 위 코드에 냉면을 추가로 팔고 싶어졌다. 방금까지 했던 작업을 반복한다. 객체를 만들고, 구상 클래스 MyStore에서 냉면 객체를 생성하는 분기를 추가한다. 어렵지 않다. 하지만 그렇다고 방치한다면 대략 이런 상황이 발생한다.



새로 기능을 추가할 때마다 객체를 생성해야 하며, 그만큼 구상 클래스 MyStore가 의존하는 객체의 수가 늘어난다. 이렇게 되면 객체(=메뉴)하나를 수정하기 위해서 이미 만들어 둔 구상 클래스의 코드를 뒤적여야 한다. 이는 완성된 클래스를 변경하는 것을 지양하는 OCP(Open-Closed Principle :개방-폐쇄 원칙)에 위배된다.

의존의 방향도 문제다. 위 그림에서는 고수준 구성요소(다른 구성요소를 이용하는 구성요소. 여기서는 MyStore.java 클래스)가 저수준 구성요소(음식 객체)에 심하게 의존하고 있다. 달리 표현하자면, 구상 클래스()에 대한 의존도가 높은 것이다. 그러므로 해당 코드는 의존도를 줄이자는 목적으로 제시된 DIP(Dependency Inversion Principal : 의존성 역전 원칙)를 역시 위반한다.

문제 상황을 해결하기 위해 팩토리 메소드 패턴을 사용할 수 있다. 객체를 만드는 Factory클래스를 추상 클래스로 정의하고 실제 객체 생성은 Factory클래스를 상속받은 서브 클래스에서 하도록 만드는 것이다. 이를 UML로 옮기면 다음과 같다.



Creator : 객체를 구현하기 위한 팩토리 메소드를 추상 메소드 형태로 정의해 놓았다.

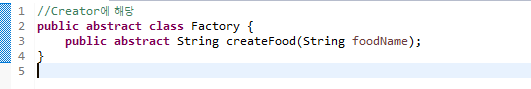
ConcreteCreator : 팩토리 메소드가 실제로 구현되는 서브 클래스. Creator를 상속 받았다.

ConcreteProduct : 구상 클래스. 팩토리 메소드에서 만들어지는 객체를 정의해 놓았다.

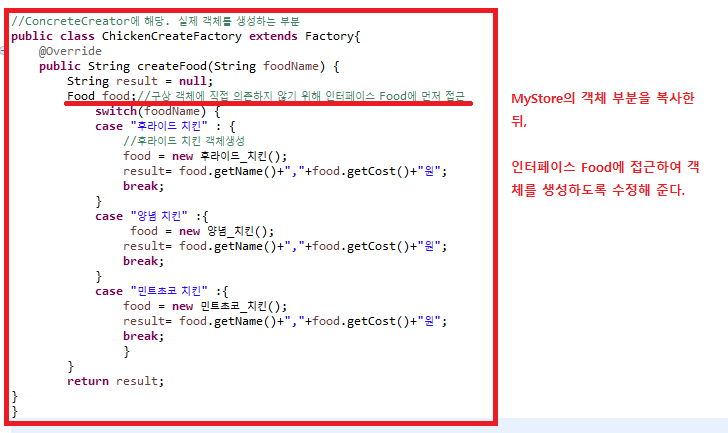
Product : 인터페이스. 구상 클래스가 가져야 할 필수 메소드를 지정해 둔다.

이제 상기의 코드를 수정하여 UML을 따라 팩토리 메소드 디자인을 구현해보자.

-Creator 구현 : Factory.java



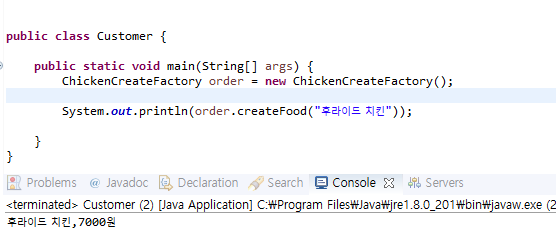
-Concrete Creator 구현 : ChickenCreateFactory.java



-Product, ConcreteProduct : Food.java

치킨 객체 자체는 이전과 동일. 추가로 메소드를 정의한 인터페이스 Food를 생성해 둔다.

-결과 : Customer.java

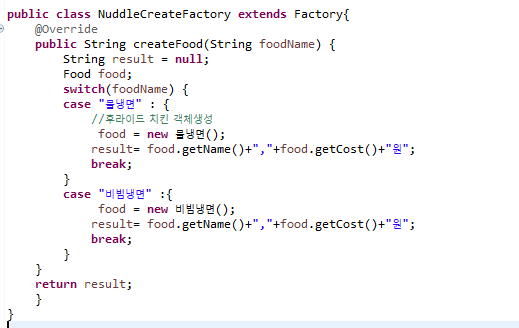


결과 자체는 변경 전과 다르지 않다. 그러나 새로운 객체의 기능을 추가할 경우 기존 코드를 수정하지 않아도 된다. 팩토리 메소드 디자인 패턴을 적용한 채로 냉면을 만드는 객체를 추가해 보자. Interface Food를 상속받아 냉면 객체를 만들고, Creator에 해당하는 추상메소드 Factory를 상속받아 새로운 Concrete Creator를 만든다. 냉면 객체를 생성하는 코드는 여기에 입력한다.

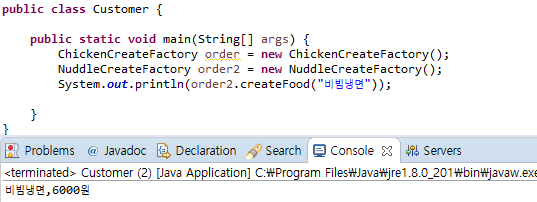
-추가된 ConcreteProduct : 물냉면.java, 비빔냉면.java

형태는 치킨 객체와 동일

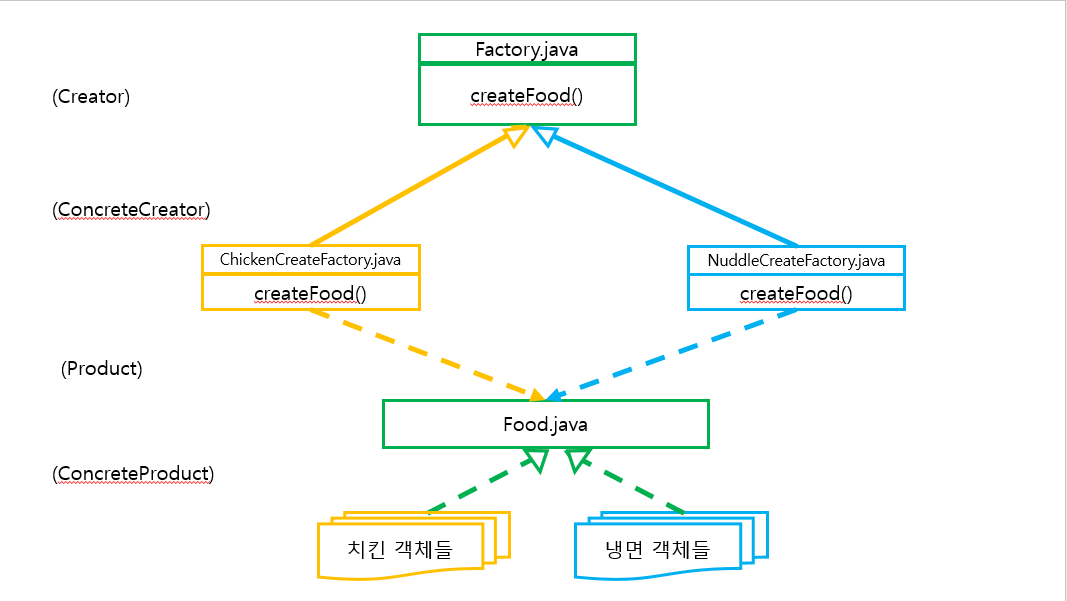
-냉면 객체를 생성하는 ConcreteFactory : NuddleCreateFactory.java



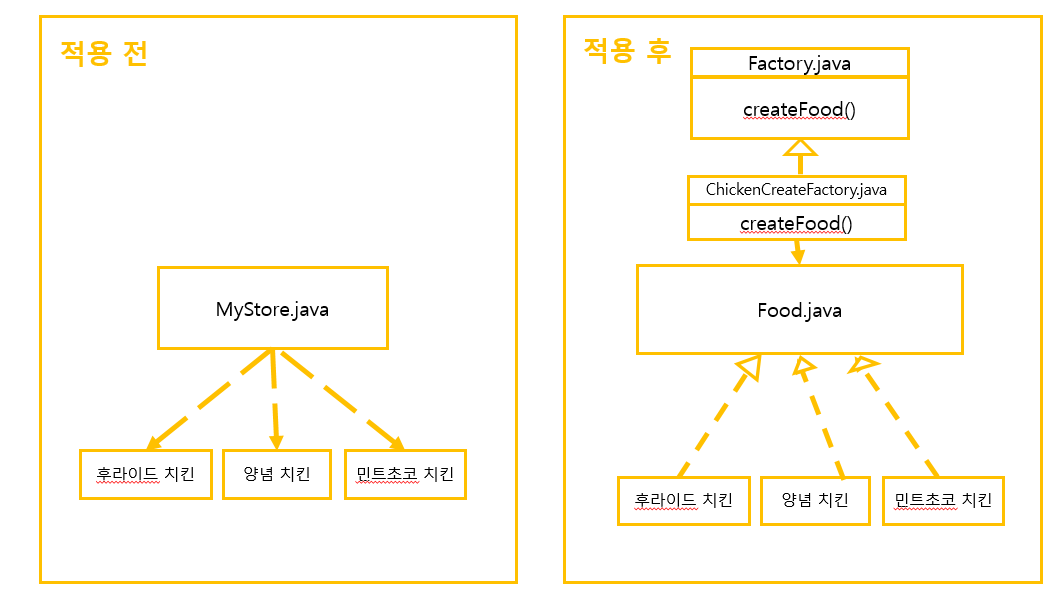
-결과 확인 :Customer.java



새로운 음식 객체 추가를 기존 코드의 추가적인 수정없이 해냈다. 이렇게 함으로써 새로운 음식 객체 생성은 얼마든지 할 수 있는(= 확장에 열려있는) 반면, 기존 코드를 변경은 하지 않아도 된다(= 변경에 닫혀 있다). OCP가 구현된 것이다. 명확한 이해를 위해 클래스 다이어그램으로 나타낼 경우 다음과 같다. 추가된 코드는 파란색 외곽선을, 공통으로 사용하는 추상 메소드는 녹색 외곽선으로 구별하였다.



그렇다면 각 클래스 간 의존성에 관련한 원칙, 즉 DIP는 지켜졌을까? 팩토리 메소드 적용 전후를 비교해 보았다.



우선, 음식 객체를 생성하는 클래스(적용 전에는 MyStore, 적용 후에는 ChickenCreateFactory가 해당)와 다른 클래스간 의존도가 낮아졌다. 디자인 적용 전에는 총 3개의 클래스와 의존관계를 갖고 있었지만, 적용 후에는 단 하나의 클래스에만 의존관계를 형성하고 있다.

무엇이 달라졌을까? 주목해야 하는 점은 의존 대상이 되는 클래스이다. 적용 전에는 구상 클래스인 치킨 객체에 의존한 반면, 적용 후에는 고수준 구성요소인 ChickenCreateFactory와 저수준 구성요소인 치킨 객체들이 모두 Food.java라는 인터페이스, 즉 추상 객체에 의존하게 되었다. 그럼으로써 위에서 아래로 내려가기만 했던 의존성(Dependency)이 그 반대 방향으로 역전(Inversion)되었음을 확인 가능하다. 의존성 역전 원칙 역시 구현된 것이다.

결과적으로, 팩토리 메소드 패턴은 추상클래스를 만들고 구체적 내용을 서브 클래스에 구현하는 방식으로 OOP적 원칙을 구현하는 디자인임을 파악할 수 있다.