# 팩토리 패턴

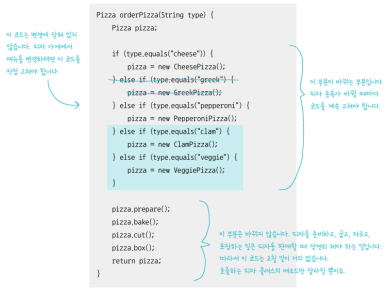


# 팩토리

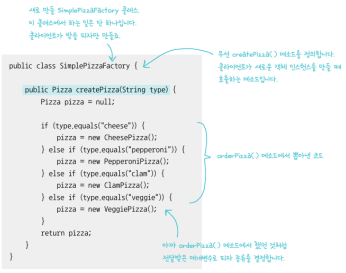
객체의 인스턴스를 만드는 작업이 항상 공개되어야 하는 것은 아니며, 오히려 모든 것을 공개했다가는 결합 문제가 생길 수 있다.

팩토리 패턴으로 불필요한 의존성을 없애서 결합 문제를 해결하는 방법을 알아보자.

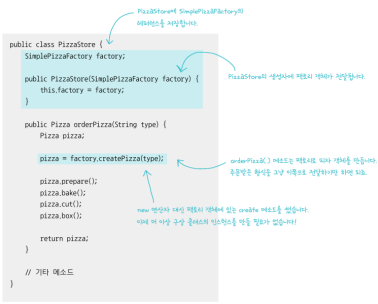
태홍대리님이 말씀하셨던 enum에 공통적으로 들어가는 메서드가 많아지게 되면 팩토리로 뺀다. 했었는데, 이게 굳이 enum 에 있지 않아도 공통적으로 관리할 수 있기 때문인건가?? 이번장 정리하고, 다시 여쭤보기!!



문제 파악



팩토리 생성



클라이언트 코드 수정

어떤 부분이 바뀌고, 어떤 부분이 바뀌지 않는지 파악했으니 캡슐화!

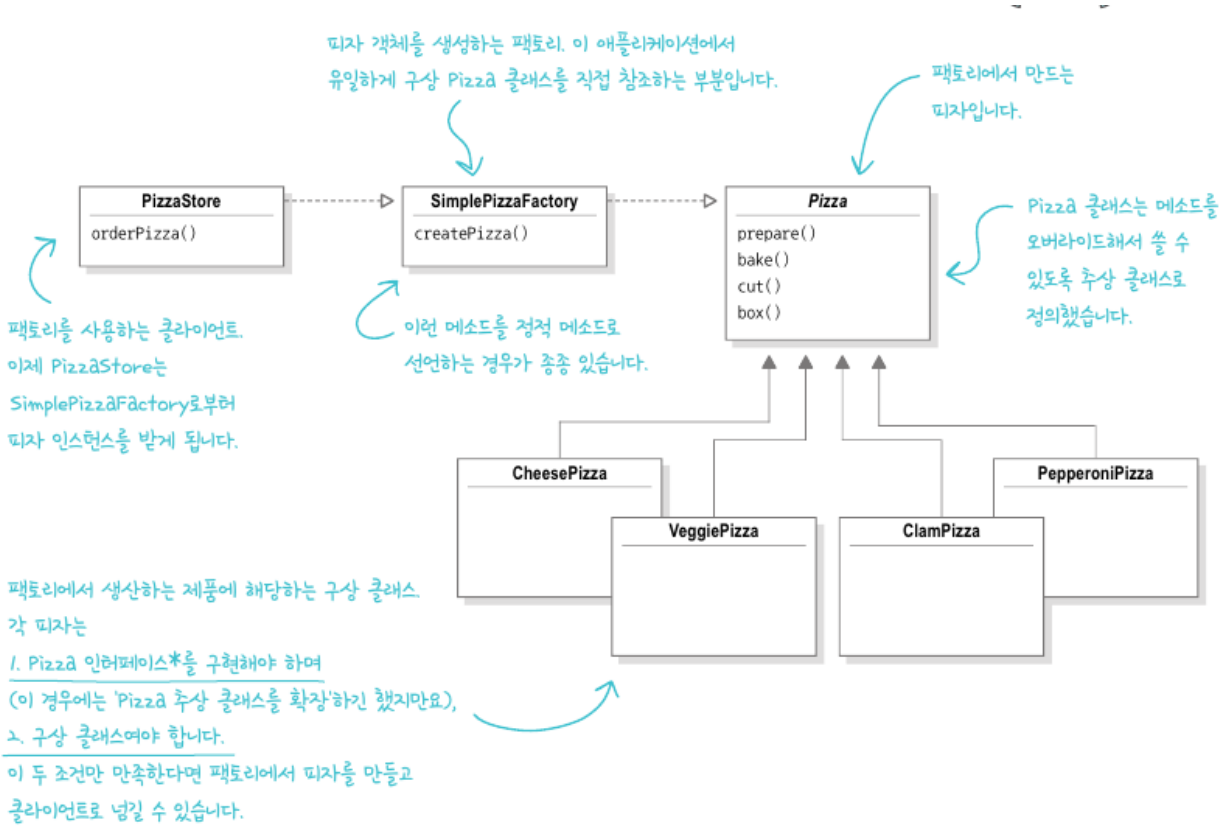
Q&A

1. 캡슐화하면 무슨 장점? 그냥 다른 객체로 넘겨 버린 것 같은데?
    a. SimplePizzaFactory를 사용하는 클라이언트가 매우 많을 수 있다. 주문 뿐만 아니라 피자 설명하거나 가격을 알려주는 PizzaShopMenu 클래스에서도 사용할 수 있고, 조금 다른 방식으로 피자 주문을 처리하는 HomeDelivery 클래스에도 이 팩토리를 사용할 수 있다.

       이런 상황에서 피자 객체 생성 작업을 팩토리 클래스로 캡슐화해놓으면 구현을 변경할 때 여기저기 고칠 필요 없이 팩토리 클래스 하나만 고치면 된다. 그리고 클라이언트 코드에서 구상 클래스의 인스턴스를 만드는 코드를 전부 없앨 수 있다.
2. 팩토리 정적 메소드로 선언한 디자인과 어떤 차이점?
    a. 간단한 팩토리를 정적 메소드로 정의하는 기법도 많이 쓰인다. 정적 팩토리라고 부르기도 함. 정적 메소드를 쓰면 객체 생성 메소드를 실행하려고 객체의 인스턴스를 만들지 않아도 된다. 하지만 서브 클래스를 만들어서 객체 생성 메소드의 행동을 변경할 수 없다는 단점이 있다.

## 팩토리 정의

간단한 팩토리는 디자인 패턴이라기 보다는 프로그래밍에서 자주 쓰이는 관용구에 가깝다. 정확히 패턴은 아님.




# 팩토리 메서드 패턴

## 프랜차이즈 사업 - 객체 동적으로 바꾸기

객체 구성을 활용하면 행동을 실행할 때 구현된 객체를 동적으로 바꿀 수 있다.

피자 프랜차이즈 사업으로 각 지점마다 지역의 특성과 입맛을 반영한 다양한 스타일의 피자를 만들어야 함.

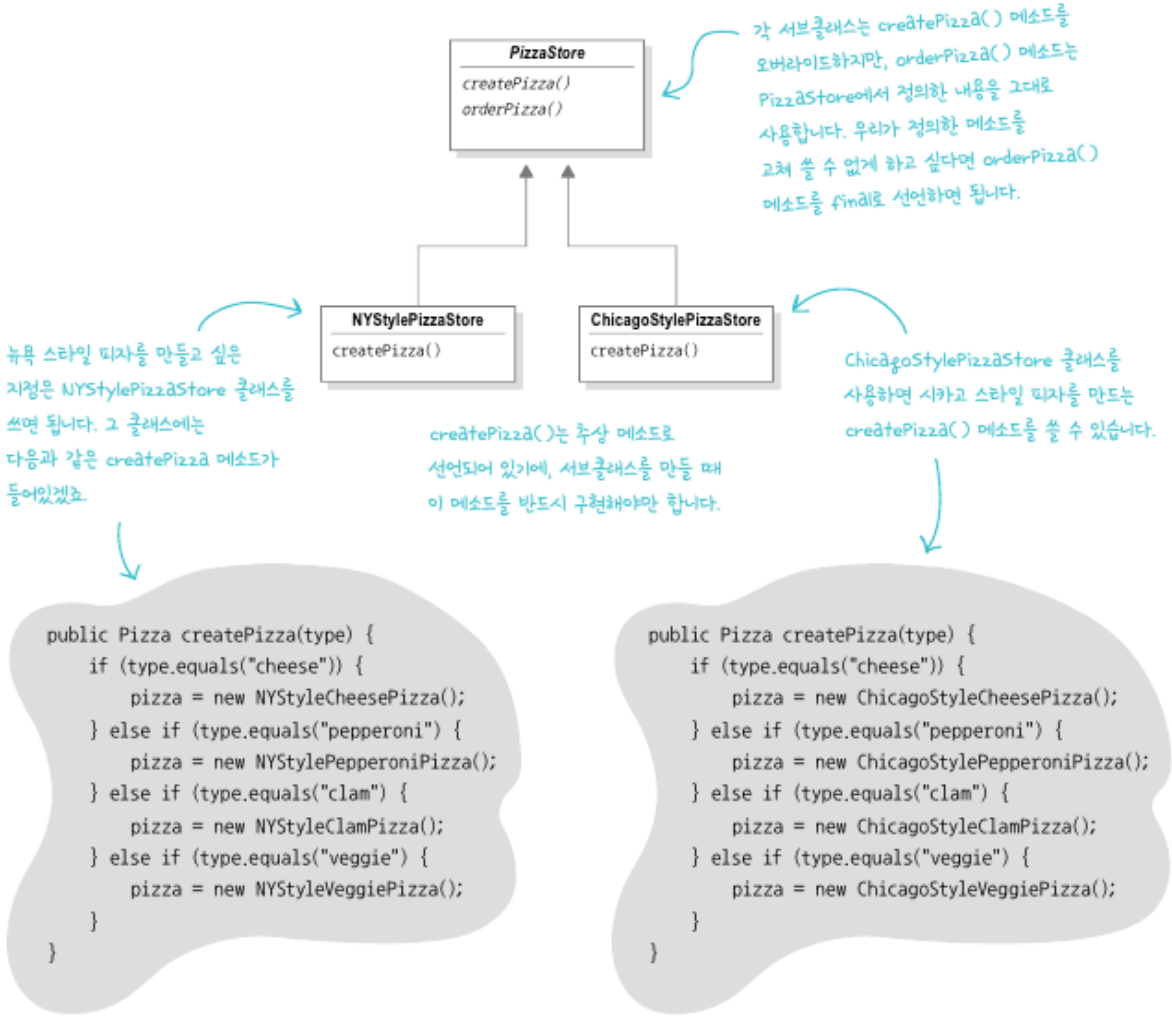
+

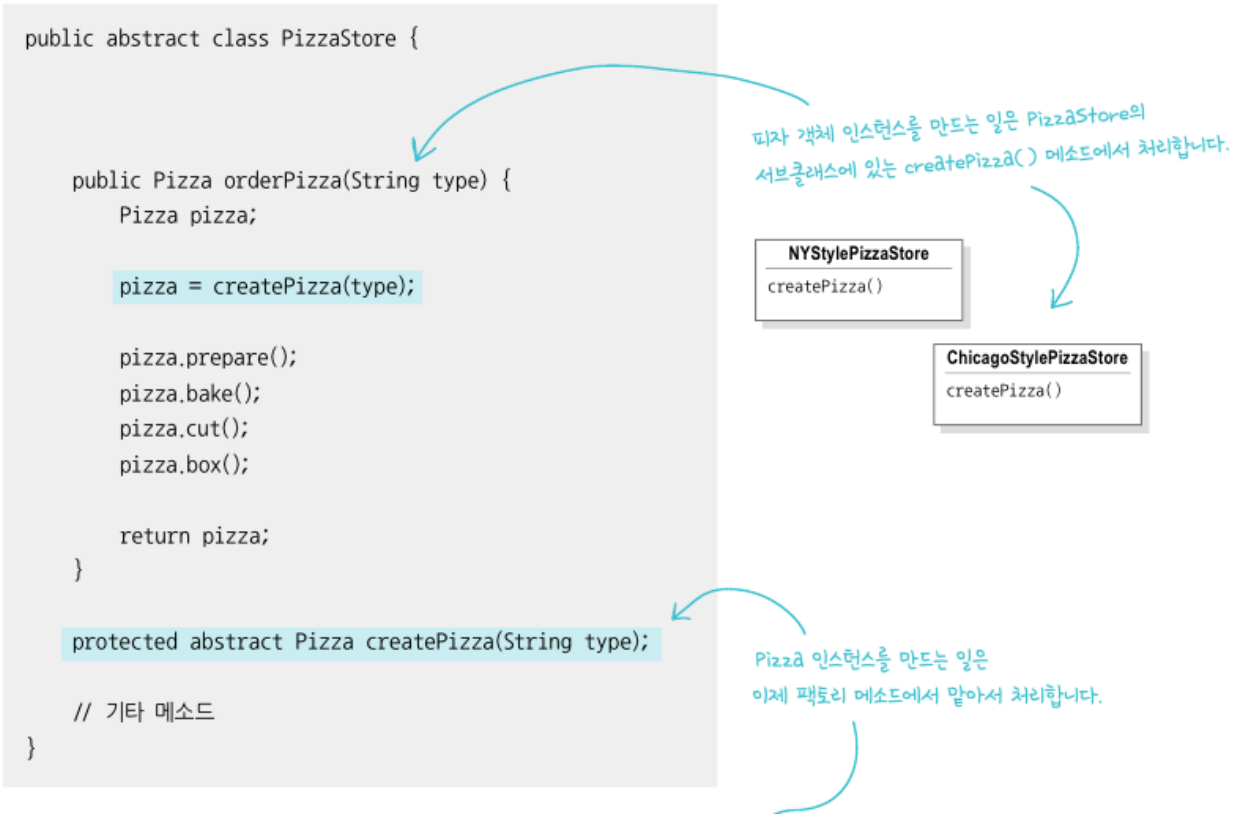
피자를 만드는 코드와 Pizza 객체를가지고 준비하고, 굽고, 자르고 포장하는 작업이 분리되어 있었음. 유연성 x. 피자 가게와 피자 만드는 과정을 하나로 묶어보자.


모든 지점에서 orderPizza() 주문 시스템을 따라 주문이 진행되어야 한다. 지점마다 달라질 수 있는 것은 피자 스타일 뿐!

createPizza() 에 피자 스타일을 넣고 해당 스타일의 피자를 만들도록 할 거임. 그러니 PizzaStore 의 서브클래스에서 createPizza()를 구현하도록 하면 됨.

→ PizzaStore 프레임워크에 충실하면서도 각각의 스타일을 제대로 구현할 수 있는 orderPizza() 메소드를 PizzaStore 서브클래스에 구비할 수 있다.


PizzaStore
createPizza()
orderPizza()
각 서브클래스는 createPizza() 메소드를 오버라이드하지만, orderPizza() 메소드는 PizzaStore에서 정의한 내용을 그대로 사용합니다. 우리가 정의한 메소드를 고쳐 쓸 수 없게 하고 싶다면 orderPizza() 메소드를 final로 선언하면 됩니다.
NYStylePizzaStore
createPizza()
ChicagoStylePizzaStore
createPizza()
뉴욕 스타일 피자를 만들고 싶은 지점은 NYStylePizzaStore 클래스를 쓰면 됩니다. 그 클래스에는 다음과 같은 createPizza 메소드가 들어있겠죠.
createPizza()는 추상 메소드로 선언되어 있기에, 서브클래스를 만들 때 이 메소드를 반드시 구현해야만 합니다.
ChicagoStylePizzaStore 클래스를 사용하면 시카고 스타일 피자를 만드는 createPizza() 메소드를 쓸 수 있습니다.
public Pizza createPizza(type) {
if (type.equals("cheese")) {
pizza = new NYStyleCheesePizza();
} else if (type.equals("pepperoni") {
pizza = new NYStylePepperoniPizza();
} else if (type.equals("clam") {
pizza = new NYStyleClamPizza();
} else if (type.equals("veggie") {
pizza = new NYStyleVeggiePizza();
}
}
public Pizza createPizza(type) {
if (type.equals("cheese")) {
pizza = new ChicagoStyleCheesePizza();
} else if (type.equals("pepperoni") {
pizza = new ChicagoStylePepperoniPizza();
} else if (type.equals("clam") {
pizza = new ChicagoStyleClamPizza();
} else if (type.equals("veggie") {
pizza = new ChicagoStyleVeggiePizza();
}
}




Pizza 인스턴스를 만드는 일은 이제 팩토리 메소드에서 맡아서 처리한다.

팩토리 메소드는 객체 생성을 서브클래스에 캡슐화할 수 있다. 그러면 슈퍼클래스에 있는 클라이언트 코드와 서브클래스에 있는 객체 생성 코드를 분리할 수 있다.

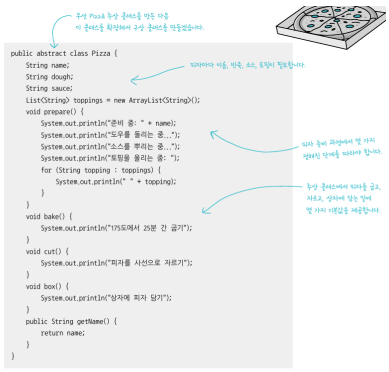
- 팩토리 메소드를 추상 메소드로 선언해서 서브클래스가 객체 생성을 책임지도록 한다.
- 팩토리 메소드는 특정 객체를 리턴하며, 그 객체는 보통 슈퍼클래스가 정의한 메소드 내에서 쓰인다.
- 팩토리 메소드는 클라이언트(슈퍼 클래스에 있는 orderPizza() 같은 코드)에서 실제로 생성되는 구상객체가 무엇인지 알 수 없게 하는 역할도 한다.
- 팩토리 메소드를 만들 때 매개변수로 만들 객체 종류를 선택할 수도 있다.
    - 매개변수 팩토리 메소드(parameterized factory method)
    - 팩토리에서 매개변수를 쓰지 않고 그냥 한 가지 객체만 만드는 경우도 많다. 어떤 방법을 쓰든 팩토리 메소드 패턴을 사용한다는 데는 변함 없다.
    - 매개 변수 팩토리 메소드에서 형식 안정성(type-safety)지장이 있을 수 있다. String 매개변수 전달하면서 오타 발생할 수 있음 → 매개변수 형식을 나타내는 객체 생성 or 정

적 상수 or enum 활용하여 형식 안정성을 보장하여 형식 오류를 컴파일 시에 잡아낼 수 있다.

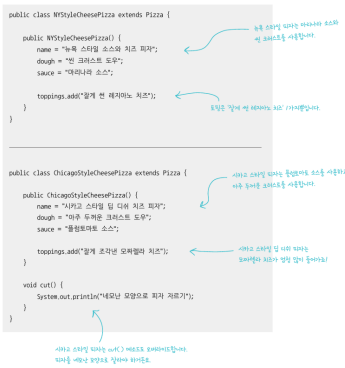
# 피자 주문하기 - 클라이언트 실행

1. 조엘은 시카고 스타일, 에단은 뉴욕 스타일 피자를 선호함
    a. 조엘은 ChicagoPizzaStore, 에단은 NYPizzaStore 인스턴스를 만들어야 함
2. PizzaStore가 만들어지면 각각 orderPizza()를 호출
    a. 이때 인자를 써서 원하는 피자 메뉴( 치즈, 야채 피자 등)를 알려줘야 된다.
3. 피자를 만들 때는 createPizza() 메소드가 호출
    a. 이 메소드는 PizzaStore 서브클래스인 NYPizzaStore와 ChicagoPizzaStore에 정의되어 있다. 각각 피자 인스턴스를 만듦. 어떤 서브클래스를 쓰든지 Pizza 객체가 orderPizza() 메소드로 리턴됨
4. orderPizza() 메소드는 어떤 스타일의 피자가 만들어졌는지 전혀 알지 못함. 하지만 피자라는 것은 알고 있어서 그 피자를 준비하고, 굽고, 자르고, 포장하는 작업을 완료함

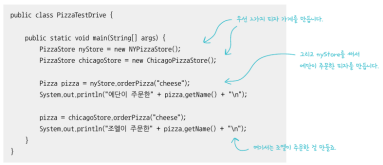
## 피자 클래스 만들고 테스트



```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList<String> toppings = new ArrayList<String>();
    void prepare() {
        System.out.println("준비 중: " + name);
        System.out.println("도우를 돌리는 중...");
        System.out.println("소스를 뿌리는 중...");
        System.out.println("토핑을 올리는 중: ");
        for (String topping : toppings) {
            System.out.println(" " + topping);
        }
    }
    void bake() {
        System.out.println("175도에서 25분 간 굽기");
    }
    void cut() {
        System.out.println("피자를 사선으로 자르기");
    }
    void box() {
        System.out.println("상자에 피자 담기");
    }
    public String getName() {
        return name;
    }
}
```



```
public class NYStyleCheesePizza extends Pizza {

    public NYStyleCheesePizza() {
        name = "뉴욕 스타일 소스와 치즈 피자";
        dough = "씬 크러스트 도우";
        sauce = "마리나라 소스";

        toppings.add("잘게 썬 레지아노 치즈");
    }
}

public class ChicagoStyleCheesePizza extends Pizza {

    public ChicagoStyleCheesePizza() {
        name = "시카고 스타일 딥 디쉬 치즈 피자";
        dough = "아주 두꺼운 크러스트 도우";
        sauce = "플럼토마토 소스";

        toppings.add("잘게 조각난 모짜렐라 치즈");
    }

    void cut() {
        System.out.println("네모난 모양으로 피자 자르기");
    }
}
```



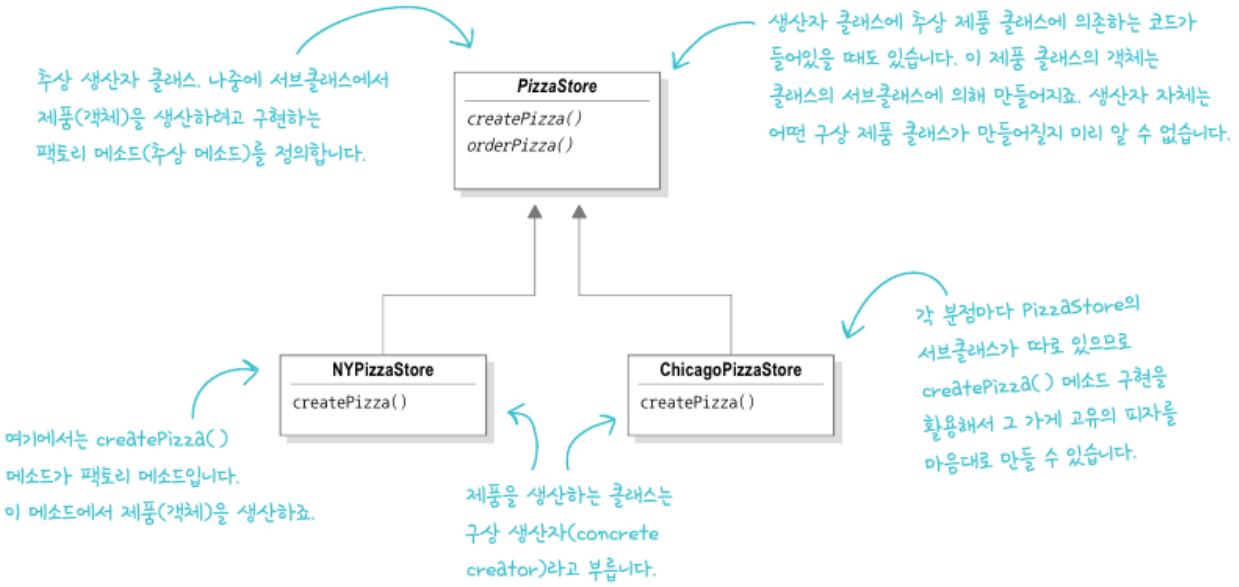
```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("에단이 주문한" + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("조엘이 주문한" + pizza.getName() + "\n");
    }
}
```

# 팩토리 메서드 패턴 정리

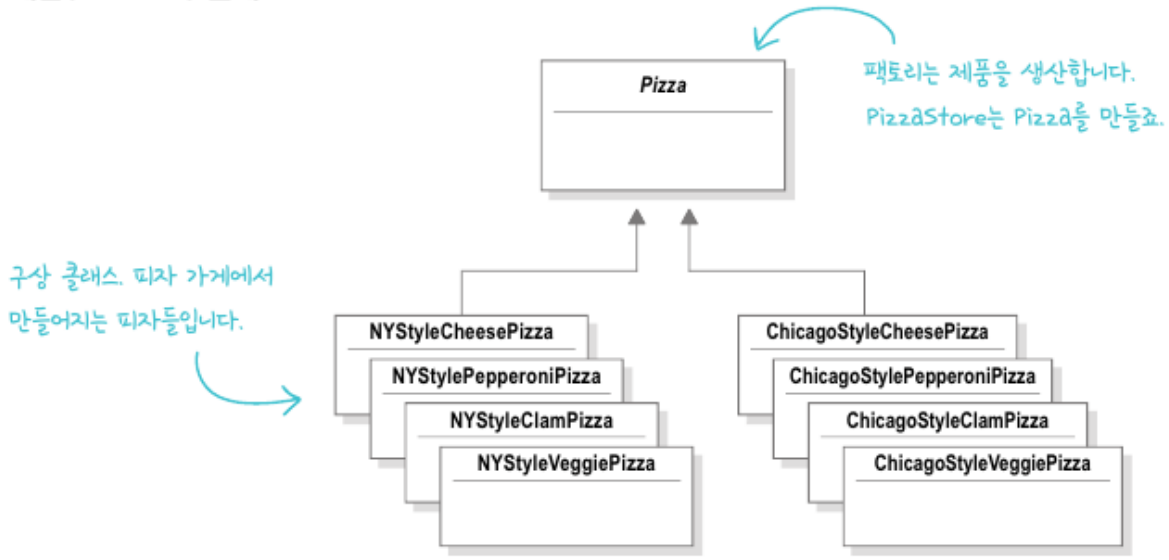
모든 팩토리 패턴은 객체 생성을 캡슐화한다. 팩토리 메소드 패턴은 서브클래스에서 어떤 클래스를 만들지 결정함으로써 객체 생성을 캡슐화한다.

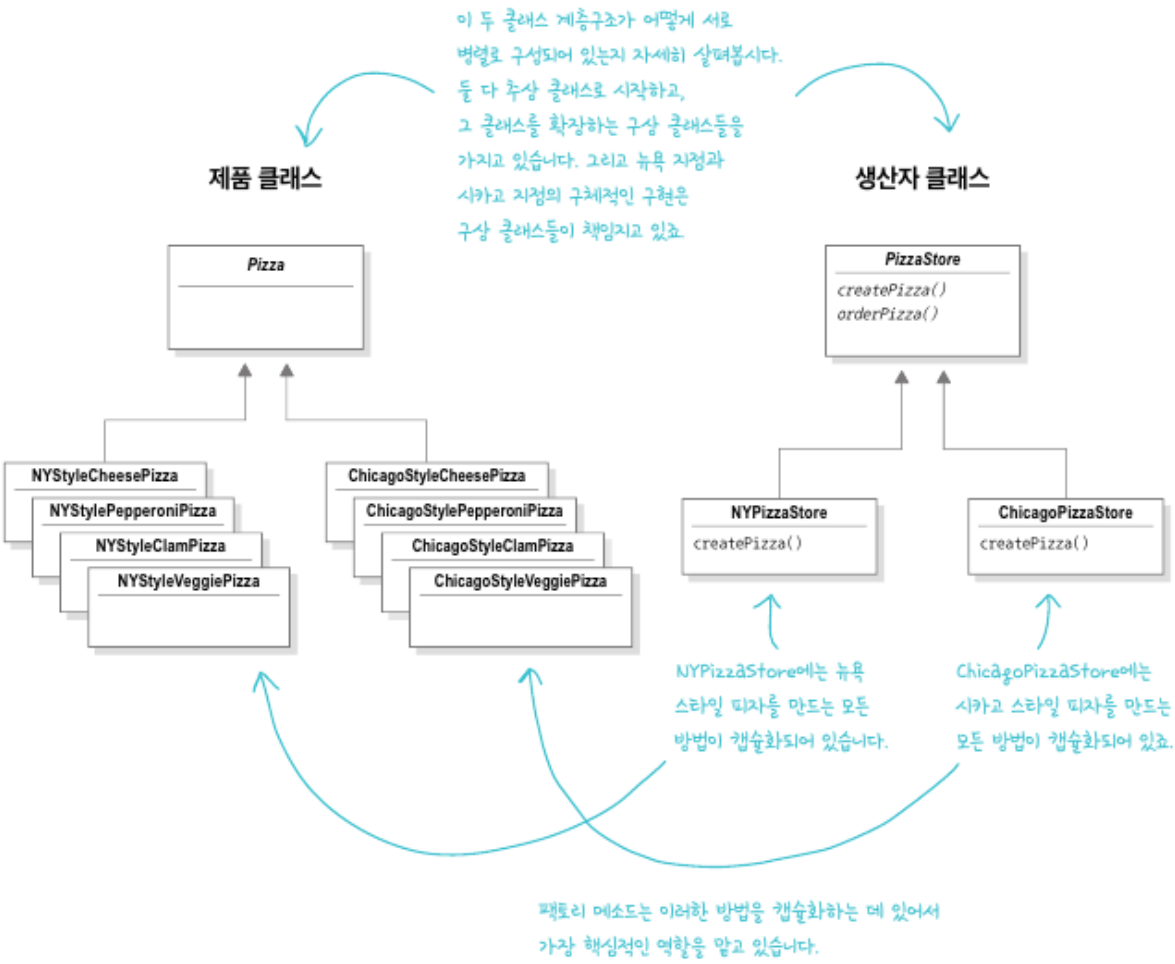
## 생산자(Creator) 클래스

조하려고 "생산자"로 옮겼습니다.

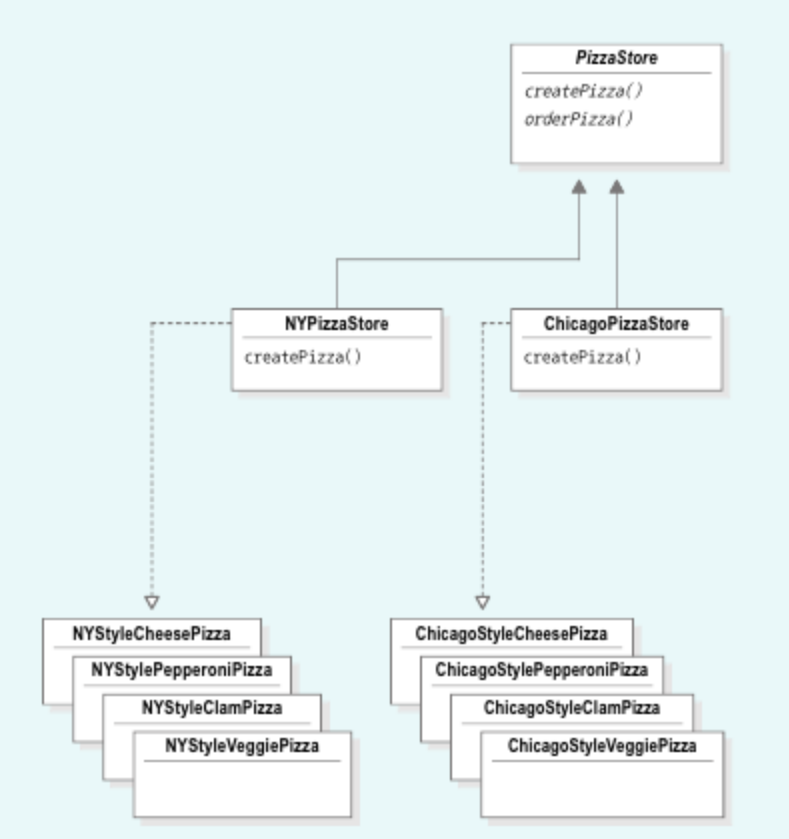



## 제품(Product) 클래스




이 두 클래스 계층구조가 어떻게 서로
병렬로 구성되어 있는지 자세히 살펴봅시다.
둘 다 추상 클래스로 시작하고,
그 클래스를 확장하는 구상 클래스들을
가지고 있습니다. 그리고 뉴욕 지점과
시카고 지점의 구체적인 구현은
구상 클래스들이 책임지고 있죠.
제품 클래스
생산자 클래스
Pizza
PizzaStore
createPizza()
orderPizza()
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
NYPizzaStore에는 뉴욕
스타일 피자를 만드는 모든
방법이 캡슐화되어 있습니다.
ChicagoPizzaStore에는
시카고 스타일 피자를 만드는
모든 방법이 캡슐화되어 있죠.
팩토리 메소드는 이러한 방법을 캡슐화하는 데 있어서
가장 핵심적인 역할을 맡고 있습니다.

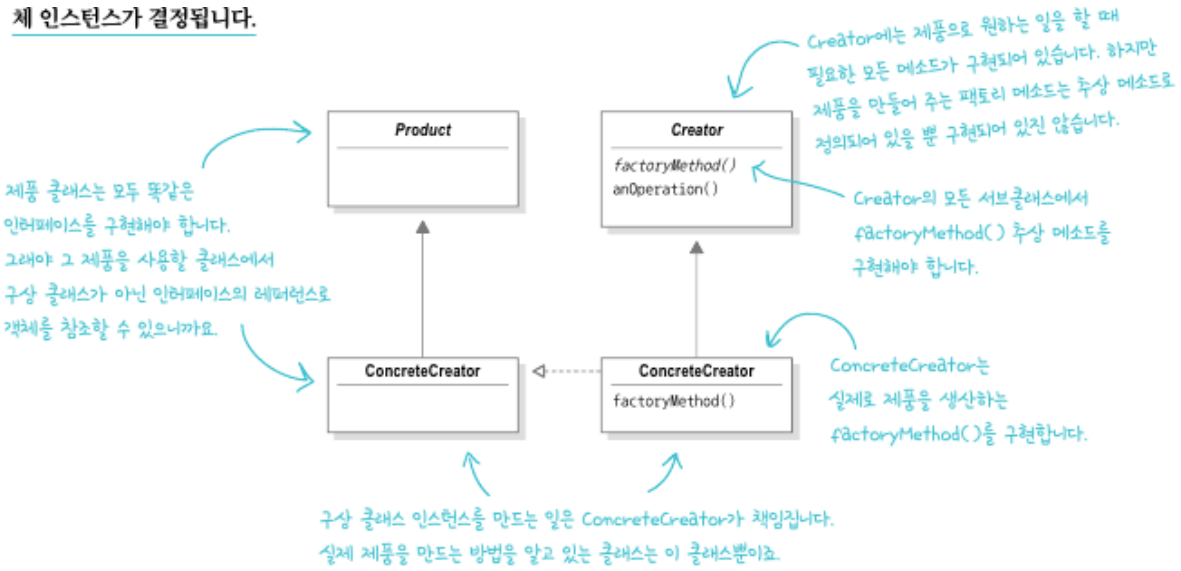
병렬 클래스 계층 구조




## 팩토리 메서드 패턴의 정의

팩토리 메소드 패턴(Factory Method Pattern)에서는 객체를 생성할 때 필요한 인터페이스를 만든다. 어떤 클래스의 인스턴스를 만들지는 서브클래스에서 결정한다. 팩토리 메소드 패턴을 사용하면 **클래스 인스턴스 만드는 일을 서브클래스에게 맡기게 된다**.

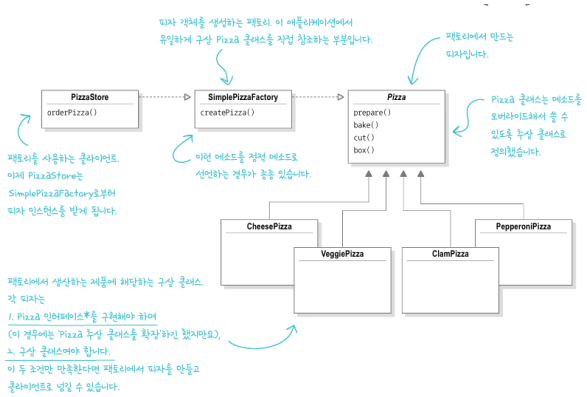
**사용하는 서브클래스에 따라 생상되는 객체 인스턴스가 결정된다**.



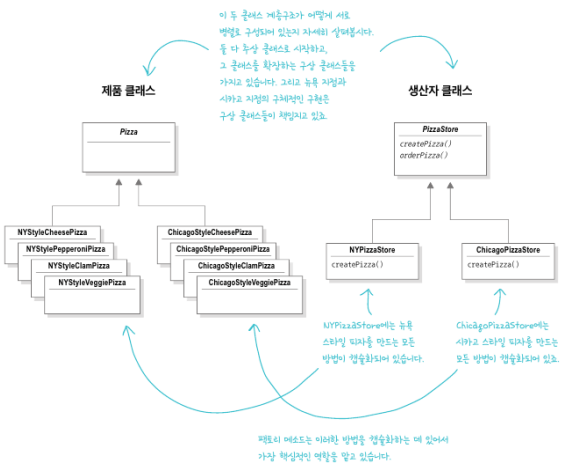

## 간단한 팩토리 vs 팩토리 메소드 패턴

팩토리 메소드 패턴에서 피자를 리턴하는 클래스가 서브클래스에 있다라는 점을 빼면 거의 같다고 생각할 수 있음!

- 상당히 비슷하다. 하지만 간단한 팩토리는 일회용 처방에 불과한 반면, 팩토리 메소드 패턴은 여러 번 재사용이 가능한 프레임워크를 만들 수 있다.
- 예를 들어, 팩토리 메소드 패턴의 orderPizza() 메소드는 피자를 만드는 일반적인 프레임워크를 제공한다. 그 프레임워크는 팩토리 메소드 피자 생성 구상 클래스를 만들었음. PizzaStore 클래스의 서브클래스를 만들 때, 어떤 구상 제품 클래스에서 리턴할 피자를만들지를 결정함.
- 이 프레임워크를 간단한 팩토리와 한번 비교해보면 간단한 팩토리는 **객체 생성을 캡슐화하는 방법을 사용하긴 하지만 팩토리 메소드만큼 유연하지 않다. 생성하는 제품을 마음대로 변경할 수 없기 때문**임.

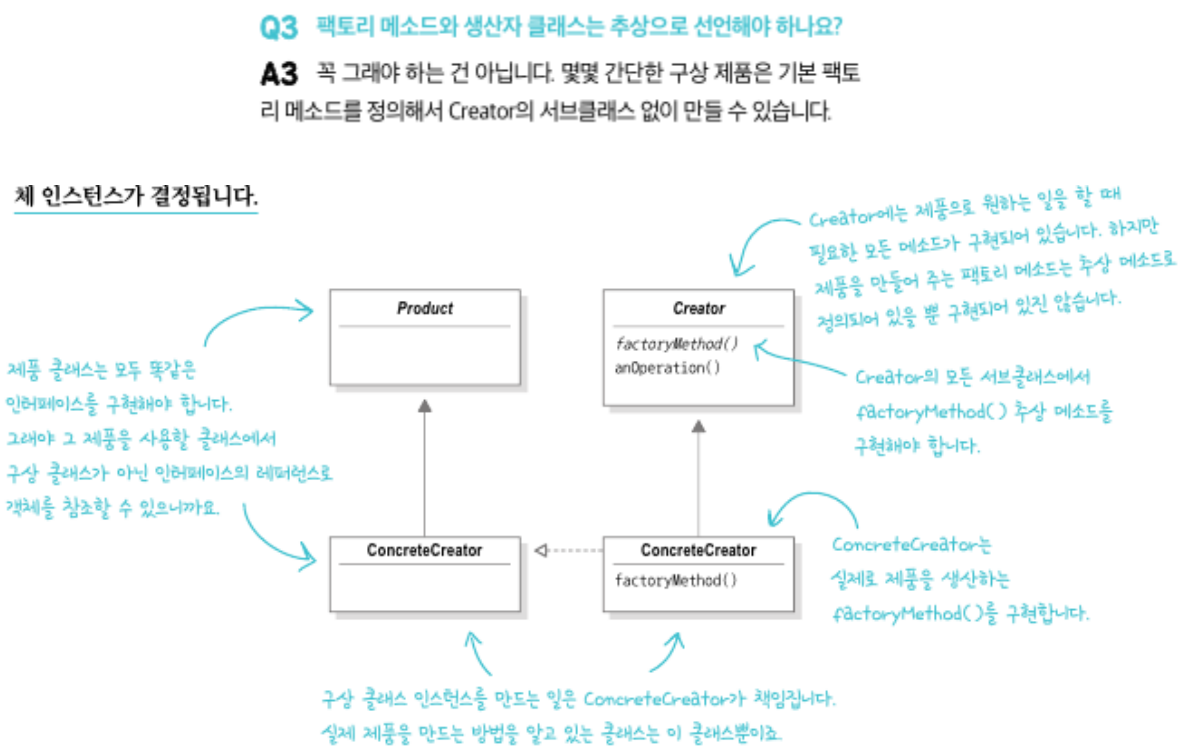



간단한 팩토리




팩토리 메소드

## 궁금한 점!!




몇몇 간단한 구상 제품은 기본 팩토리 메소드를 정의해서 Creator 서브클래스 없이 만들 수 있다고 함.

그래도 팩토리 메소드 패턴을 사용하는거니까 간단한 팩토리 보다 유연해야 하는데 간단한 팩토리와 똑같아지는 것 아닌가?? 어떤 점에서 유연해진다는건지 이해가 안됨.

굳이 추상으로 선언하지 않는다면

PizzaStore(order, create) → Pizza 바로 가서 간단한 팩토리랑 동일하지 않는가?

간단한 팩토리는 PizzaStore(order) → PizzaFactory(create) → Pizza 로 가니까 다른건가?

객체를 서브클래스에서 결정되는게 아니니까 팩토리 메소드 패턴이 아니게 되는데 팩토리 메서드 패턴이라고 부를 수 있는지..;;

으흠 간단한 팩토리에서는 Store 만들 때 Factory 를 주입받아야 함.

그래서 더 유연하다는건가? 이해가 안가네;;

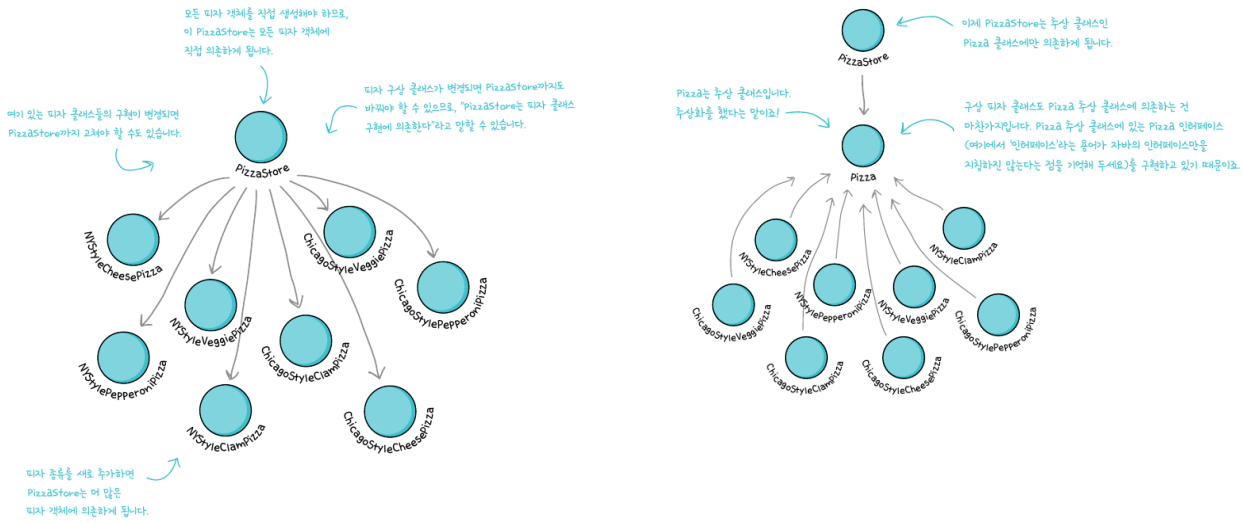
팩토리 메소드 패턴이 간단한 팩토리보다 유연하다고 말하려면 absctract 를 사용했을 때만 성립하는 것 같음. 그렇지 않다면 이해가 안됨;;;;!!!!!!!!!!! 누가 도와줘

# 객체 의존성 살펴보기

클라이언트 코드도 구체적인 클래스를 알아야 하기 때문에 변경에 닫히지 않느냐. 맞음. DI 를 통해 최소화할 수 있음.

## 의존성 뒤집기 원칙(Dependency Inversion Principle)

- **추상화**된 것에 의존하게 만들고 구상 클래스에 의존하지 않게 만든다.
- 구현보다 인터페이스에 맞춰서 프로그래밍한다. 라는 원칙보다 추상화를 더 많이 강조함.
- 고수준 구성 요소가 저수준 구성 요소에 의존하면 안 되며, 항상 추상화에 의존하게 만들어야 한다.
    - 고수준 구성요소는 다른 저수준 구성 요소에 의해 정의되는 행동이 들어있는 구성 요소를 뜻한다.
    - 예를 들어, PizzaStore 행동은 피자에 의해 정의되므로 PizzaStore 는 고수준 구성 요소라고 할 수 있다. PizzaStore는 다양한 피자 객체를 만들고, 피자를 준비하고, 굽고, 자르고 포장함. 이때 PizzaStore에서 사용하는 피자 객체는 저수준 구성 요소이다.




패토리 메소드 패턴을 적용하면 고수준 구성 요소인 PizzaStore와 저수준 구성 요소인 피자 객체가 모두 추상 클래스인 Pizza에 의존한다는 사실을 알 수 있다. 팩토리 메소드 패턴이 의존성 뒤집기 원칙을 준수하는 유일한 방법은 아니지만 적합한 방법 중 하나다.

## 근데 왜 뒤집기? 뭘 뒤집는다는 거지

객체 지향 디자인을 할 때 일반적으로 생각하는 방법과는 반대로 뒤집어서 생각해야 하기 때문임.

저수준 구성요소가 고수준 추상 클래스에 의존함. 그리고 저수준 구성 요소도 같은 추상 클래스에 연결되어 있음.

왼쪽 다이어그램에서 의존성이 위에서 아래로 내려가기만 했던 것과 반대로 뒤집어져 있다. 고수준 모듈과 저수준 모듈이 둘 다 하나의 추상 클래스에 의존하게 됨.

## 의존성 뒤집기 원칙을 지키는 방법

다음 가이드라인을 따르면 의존성 뒤집기 원칙에 위배되는 객체지향 디자인을 피하는 데 도움이 됨

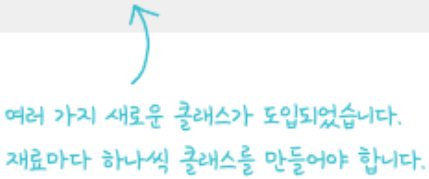
1. 변수에 구상 클래스의 레퍼런트를 저장하지 말자
    a. 팩토리를 써서 구상 클래스의 레퍼런스를 변수에 저장하는일을 미리 방지
2. 구상 클래스에서 유도된 클래스를 만들지 말자
    a. 특정 구상 클래스에 의존하게 됨. 인터페이스나 추상 클래스처럼 추상화된 것으로부터 클래스를 만들어야 한다.

3. 베이스 클래스에 이미 구현되어 있는 메소드를 오버라이드하지 말자
    a. 베이스 클래스가 제대로 추상화되지 않음. 모든 서브클래스에서 공유될 수 있는 것만 베이스 클레스에 메소드를 정의해야 함.

# 추상 팩토리 패턴

## 원재료 팩토리 만들기

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FreshClams();
    }
}
```

재료군에 들어있는 재료를
뉴욕 지점에 알맞게 만듭니다.

야채는 야채로 구성된 배열을 리턴합니다.
여기에서는 야채를 만드는 부분을 직접 하드 코딩했습니다
이 부분도 조금 더 복잡하게 만들 수 있지만,
팩토리 패턴을 배우는 과정에서는 별로 필요할 것
같지 않아서 그냥 간단하게 했습니다.

최고 품질의 슬라이스 페퍼로니. 시카고와 뉴욕에서
같은 페페로니를 씁니다. 여러분이 시카고 팩토리를 직접
만들어 볼 텐데, 그때도 이 재료를 쓰세요.

뉴욕은 바닷가에 있어서 신선한 조개를 쉽게
구할 수 있습니다. 하지만 시카고는 내륙에 있어서
어쩔 수 없이 냉동 조개를 써야 합니다.

```
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("175도에서 25분 간 굽기");
    }

    void cut() {
        System.out.println("피자를 사선으로 자르기");
    }

    void box() {
        System.out.println("상자에 피자 담기");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // 피자 이름을 출력하는 부분
    }
}
```

피자마다 준비 과정에서 사용하는 원재료들이 있습니다.

이제 prepare() 메소드를 추상 메소드로 만들었습니다.
이 부분에서 피자를 만드는 데 필요한 재료들을 가져옵니다.
물론 모든 원재료는 원재료 팩토리에서 가져옵니다.

prepare() 메소드를 제외한
다른 메소드들은 바뀌지 않습니다.

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("준비 중:" + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

피자의 원재료를 제공하는 팩토리가 필요합니다. 각 피자 클래스는 생성자로부터 팩토리를 전달받고 그 팩토리를 인스턴스 변수에 저장합니다.

팩토리의 마법이 일어나는 부분!

prepare() 메소드에서 치즈 피자를 만드는 각 단계를 처리합니다. 재료가 필요할 때마다 팩토리에 있는 메소드를 호출해서 만듭니다.

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("뉴욕 스타일 치즈 피자");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("뉴욕 스타일 야채 피자");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("뉴욕 스타일 조개 피자");

        } else if (item.equals("pepperoni")) {

            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("뉴욕 스타일 페퍼로니 피자");

        }
        return pizza;
    }
}
```

뉴욕 지점에는 뉴욕 피자 원재료 팩토리를 전달해 줘야 합니다. 뉴욕 스타일 피자를 만들 때 필요한 재료는 이 팩토리에서 공급합니다.

이제 피자에 맞는 재료를 만드는 팩토리를 피자 객체에 전달해 줍니다.

앞쪽을 보고 피자와 팩토리가 어떤 식으로 연결되는지 확실히 이해하고 넘어갑시다.

피자 형식마다 새로운 Pizza 인스턴스를 만들고 원재료를 공급받는데 필요한 팩토리를 지정해 줍니다.

**뇌 단련**

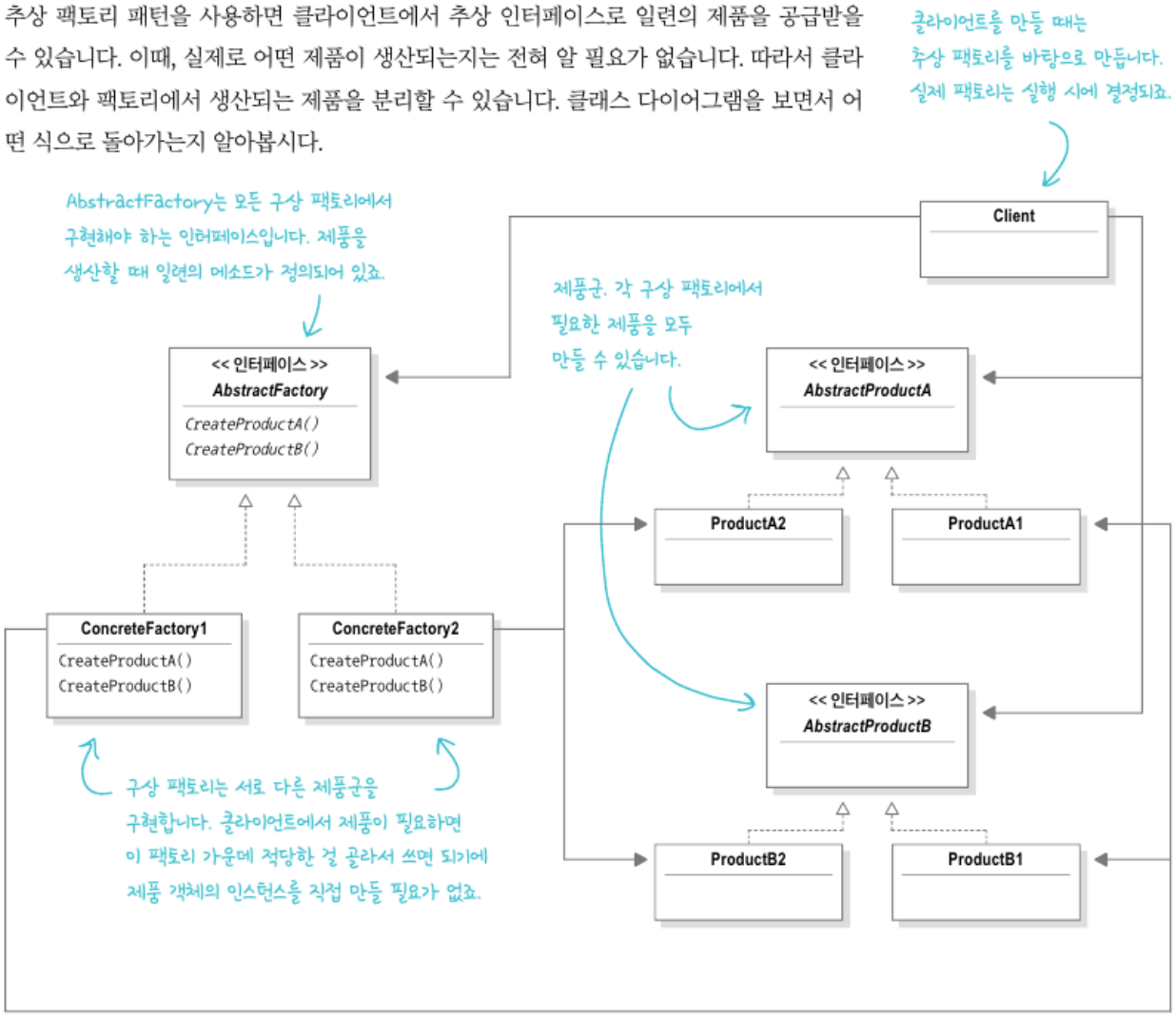
여기에 있는 createPizza( ) 메소드와 앞

# 추상 팩토리 메서드 패턴 정의

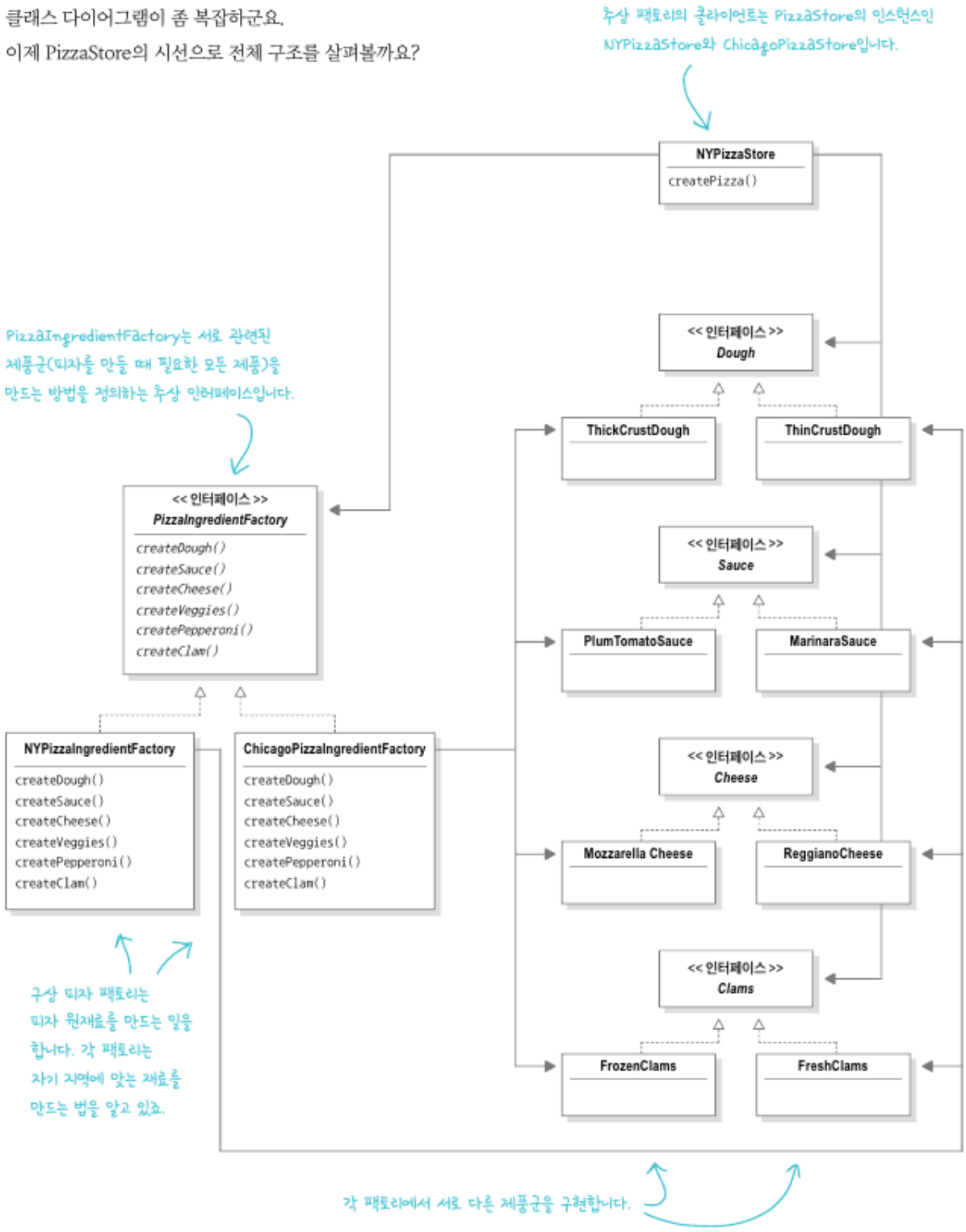
제품군을 만들 때 쓸 수 있는 패턴.

추상 팩토리 패턴(Abstract Factory Pattern)은 구상 클래스에 의존하지 않고도 서로 연관되거나 의존적인 객체로 이루어진 제품군을 생성하는 인터페이스를 제공한다. 구상 클래스를 서브클래스에서 만든다.

추상 팩토리 패턴을 사용하면 클라이언트에서 추상 인터페이스로 일련의 제품을 공급받을 수 있습니다. 이때, 실제로 어떤 제품이 생산되는지는 전혀 알 필요가 없습니다. 따라서 클라이언트와 팩토리에서 생산되는 제품을 분리할 수 있습니다. 클래스 다이어그램을 보면서 어떤 식으로 돌아가는지 알아봅시다.



클래스 다이어그램이 좀 복잡하군요.
이제 PizzaStore의 시선으로 전체 구조를 살펴볼까요?




## 추상 팩토리 패턴에서 메소드가 팩토리 메소드로 구현되는 경우도 있다?

> 추상 팩토리에 들어있는 각 메소드는 사실 팩토리 메소드(createDough(), createSauce() 등) 같다. 메소드는 추상 메소드로 선언되어 있고 서브클래스에서 오버라이드해서 객체를 만들던데, 그러면 그냥 팩토리 메소드 아닌가요?

당연히 그럴 수 있다. 추상 팩토리가 원래 일련의 제품을 만드는 데 쓰이는 인터페이스를 정의하려고 만들어 진것이기 때문이다.

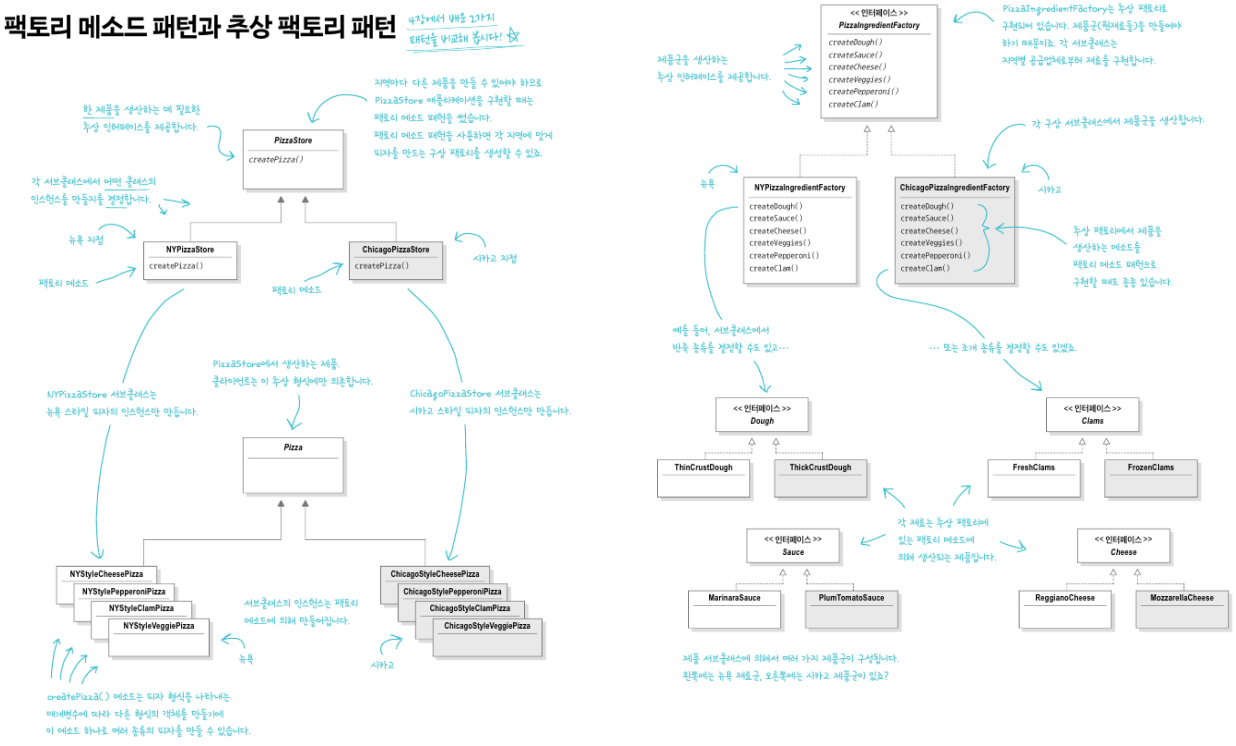
그 인터페이스에 있는 각 메소드는 구상 제품을 생산하는 일을 맡고, 추상 팩토리의 서브클래스를 만들어서 각 메소드의 구현을 제공한다. 따라서 추상 팩토리 패턴에서 제품을 생산하는 메소드를 구현하는 데 있어서 팩토리 메소드를 사용하는 것은 너무나도 자연스러운 일이라고 할 수 있다.

## 팩토리 메소드 패턴 vs 추상 팩토리 패턴

- 둘 다 객체를 만든일을 한다. 하지만
- 팩토리 메소드 패턴은
  - **상속**으로 객체를 만든다. = 클래스를 써서 제품을 만든다.
  - 팩토리를 **구현**하는 방법에 초점
  - **구체적인 객체 생성 과정을 하위 또는 구체적인 클래스로 옮기는 것이 목적**
  - 객체를 생성할 때는 클래스를 확장하고 팩토리 메소드를 오버라이드해야 함.
  - 팩토리는 객체를 만듦. 팩토리 메소드 패턴을 사용하는 이유는 서브클래스로 객체를 만드려는 거니까.
  - 즉, 클라이언트와 구상 형식을 분리하는 역할
- 추상 팩토리 패턴은
  - **객체 구성**(composition)으로 만든다. = 객체를 써서 제품을 만든다.
  - 팩토리를 **사용**하는 방법에 초점
  - **관련있는 여러 객체를 구체적인 클래스에 의존하지 않고 만들 수 있게 해주는 것이 목적**
  - 클라이언트와 구상 형식을 분리하는 역할임은 동일하지만 방식이 다름.
  - 제품군을 만드는 추상 형식을 제공함. 제품이 생산되는 방법은 서브클래스에서 정의함.

- 팩토리를 사용하고 싶으면 일단 인스턴스를 만든 다음 추상 형식을 써서 만든 코드에 전달
- 따라서 팩토리 메소드 패턴을 쓸 때와 마찬가지로 클라이언트와 실제 구상 제품이 분리됨.
- 연관된 제품을 하나로 묶을 수 있다는 장점도 있음
- 새로운 제품을 추가하려면 인터페이스를 바꿔야 한다.
  - 하지만 많은 제품이 들어가있는 제품군을 생성하기에 인터페이스도 아주 큰 편.
  - 팩토리 메소드 패턴은 한 가지 제품만 생산. 복잡한 인터페이스도 필요하지 않고, 메소드도 하나만 있으면 됨.
- 팩토리 메소드로 구상 팩토리를 구현할 수 있다. 이 경우 제품을 생산하는 용도로만 쓰긴 함.

어떤 패턴을 쓰든 객체 생성을 캡슐화해서 애플리케이션의 결합을 느슨하게 만들고, 특정 구현에 덜 의존하도록 만들 수 있다는 것은 바람직한 일이다.

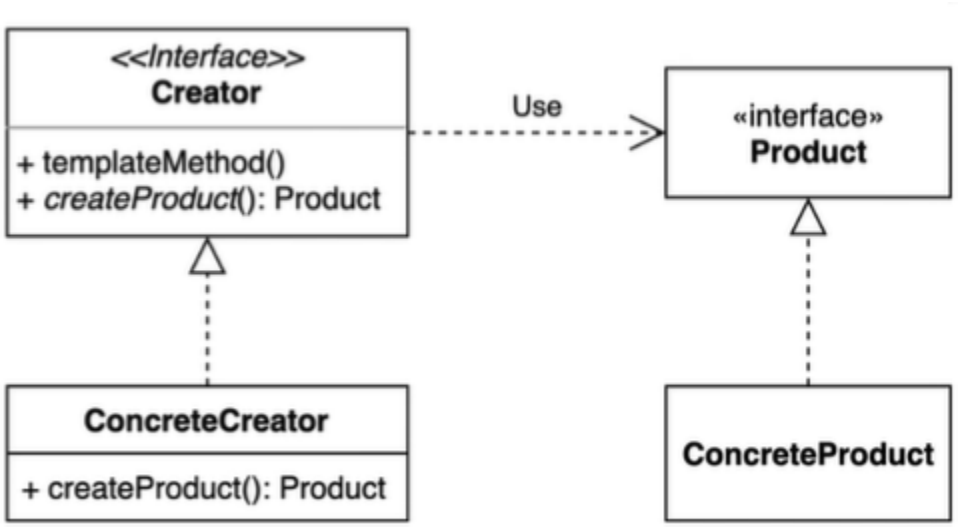


## 느낀점

상속과 인터페이스의 차이점을 팩토리 메소드 패턴과 추상 팩토리 패턴을 통해 배울 수 있었다.

상속은 말그대로 부모 자식 간 (자식은 부모에게 유전적 형질을 받죠) 상속이고, 인터페이스는 객체의 행위의 추상화

근데 팩토리 메소드 패턴도 인터페이스로 만들 수 있음. default/private 메소드로 변하지 않는 부분 공통화시키고 create 메소드만 서브 클래스에서 구현하도록.




## 궁금한 점!!

1. 추상 팩토리 패턴에서 팩토리 메소드 패턴을 사용하지 않으면 어떻게 되는건데? 단일 PizzaIngredientFactory 가 되는건가?