팩토리 패턴

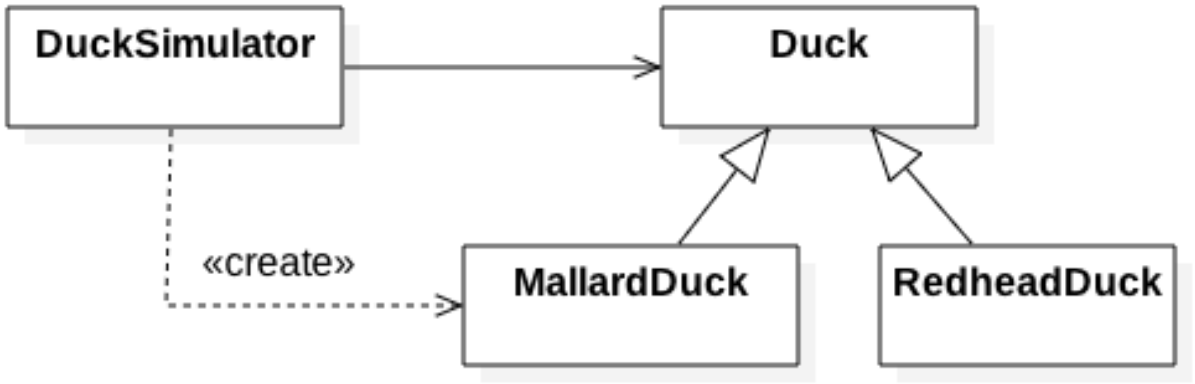
이관우

kwlee@hansung.ac.kr

학습 목표

- 객체 생성과 관련된 문제를 이해한다.
- 간단한 팩토리를 이해한다.
- 팩토리 메소드 패턴을 이해한다.
- 추상 팩토리 패턴을 이해한다.
- 간단한 팩토리, 팩토리 메소드 패턴, 추상 팩토리 패턴의 차이를 이해한다.

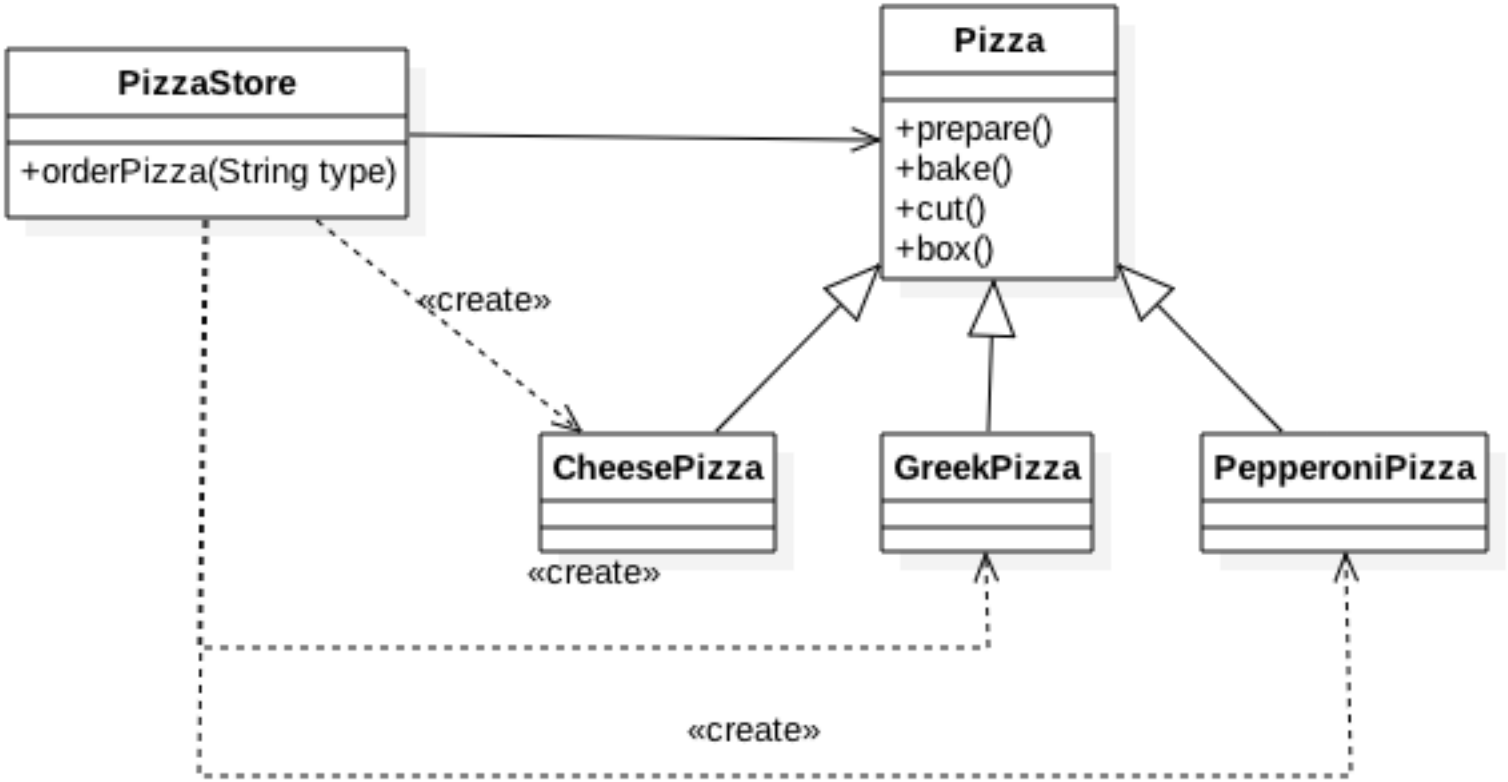
객체 생성 문제




```
Duck duck = new MallardDuck();
```

- 위 코드는 디자인 원칙 중에 하나인 "**구현이 아닌 인터페이스에 맞춰서 프로그래밍하라**" 를 준수하고 있나요?

피자 가게 (초기 설계)


PizzaStore
+orderPizza(String type)
Pizza
+prepare()
+bake()
+cut()
+box()
«create»
CheesePizza
GreekPizza
PepperoniPizza
«create»
«create»

객체 생성

```
Pizza orderPizza (String type) {
        Pizza pizza;

        if (type.equals("cheese")) {
                pizza = new CheesePizza();
        } else if (type.equals("greek")) {
                pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
                pizza = new PepperoniPizza();
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
}
```

변경 사항

- 신제품(Clam Pizza, Veggie Pizza) 출시, 구제품(Greek Pizza) 제외

```
Pizza orderPizza (String type) {
        ...
        if (type.equals("cheese")) {
                pizza = new CheesePizza();
        } else if (type.equals("greek")) {
                pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
                pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
                pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
                pizza = new VeggiePizza();
        }
        ...
        return pizza;
}
```

퀴즈

- 이전 코드에서 지켜지지 않은 디자인 원칙은 또 무엇이 있나요? 모두 골라보세요.

1. 변화하는 부분과 변하지 않는 부분을 분리하라
2. 서로 상호작용하는 객체 사이에서는 가능하면 느슨하게 결합하는 디자인을 사용해야 한다
3. 클래스는 확장에 대해서는 열려 있어야 하지만, 코드 변경에 대해서는 닫혀 있어야 한다

객체 생성 부분의 캡슐화

```
Pizza orderPizza (String type) {
        Pizza pizza;

        객체 생성 부분

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
}
```

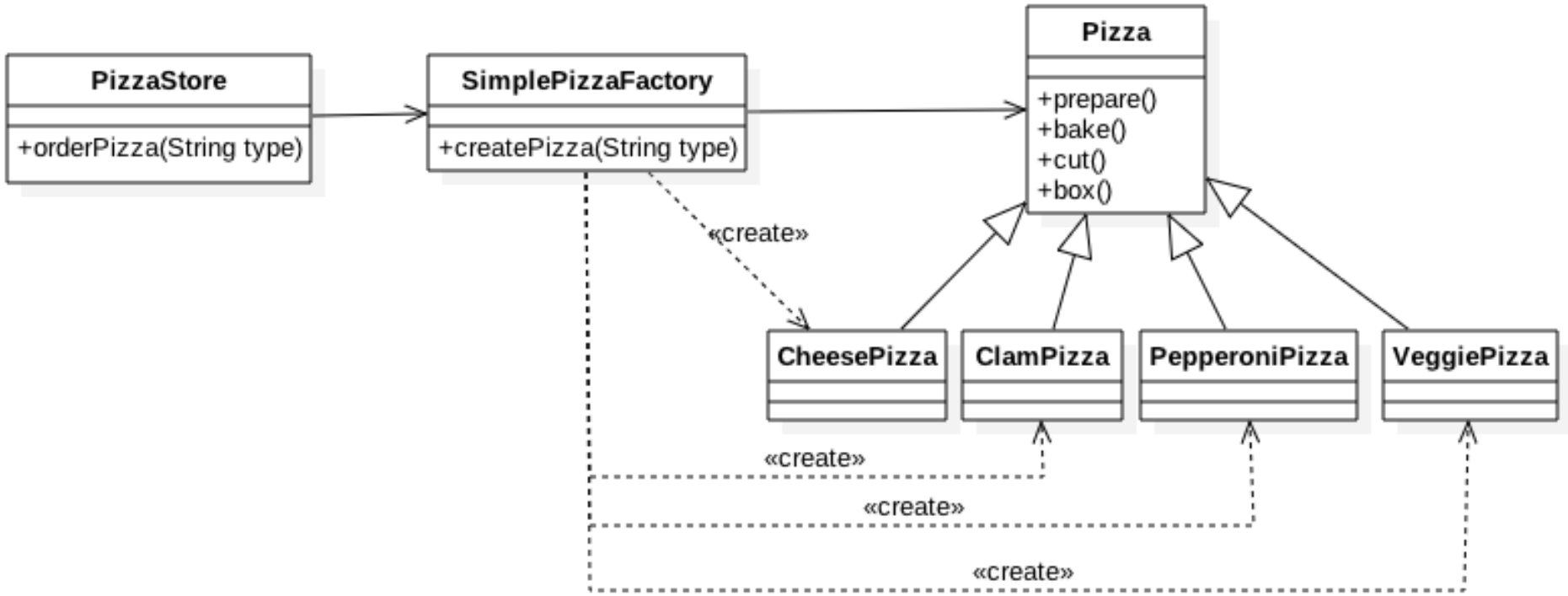
분리

```
if (type.equals("cheese")) {
    pizza = new CheesePizza();
} else if (type.equals("pepperoni")) {
    pizza = new PepperoniPizza();
} else if (type.equals("clam")) {
    pizza = new ClamPizza();
} else if (type.equals("veggie")) {
    pizza = new VeggiePizza();
}
```

캡슐화

SimplePizzaFactory

SimplePizzaFactory



SimplePizzaFactory

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }

        return pizza;
    }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzas/src/hansung/designpatterns/factory/pizzas/PizzaStore.java
https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzas/src/hansung/designpatterns/factory/pizzas/SimplePizzaFactory.java

SimplePizzaFactory 실행

- 다운로드 SimplePizzaFactory 예제 링크

  https://github.com/kwanulee/DesignPattern/tree/master/factory/pizzas

- 실행 결과

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza pizza = store.orderPizza( type: "cheese");
        System.out.println("We ordered a " + pizza + "\n");
        //System.out.println(pizza);

        pizza = store.orderPizza( type: "veggie");
        System.out.println("We ordered a " + pizza + "\n");
        //System.out.println(pizza);
    }
}
```

```
Preparing Cheese Pizza
Baking Cheese Pizza
Cutting Cheese Pizza
Boxing Cheese Pizza
We ordered a ---- Cheese Pizza ----
    Regular Crust
    Marinara Pizza Sauce
    Fresh Mozzarella
    Parmesan


Preparing Veggie Pizza
Baking Veggie Pizza
Cutting Veggie Pizza
Boxing Veggie Pizza
We ordered a ---- Veggie Pizza ----
    Crust
    Marinara sauce
    Shredded mozzarella
    Grated parmesan
    Diced onion
    Sliced mushrooms
    Sliced red pepper
    Sliced black olives
```

간단한 팩토리

- 디자인 패턴이라기 보다는 프로그래밍을 하는 데 있어서 자주 쓰이는 관용구에 가까움.

- 정적 메소드로 선언하는 경우가 많음

```
public class SimplePizzaFactory {
    public static Pizza createPizza(String type) {
     ...
    }
}
```

- 장점은?
  - 간단한 팩토리를 사용하는 클래스가 많은 경우, 구현을 변경해야 한다면 변경 부분을 이 클래스로 한정시킬 수 있음.

피자 프렌차이즈 사업의 확장

- 지역별로 피자 프렌차이즈 지점의 확장
- 기존 코드를 다른 지점에서도 사용하길 희망
- 지역별 특성 반영
  - 뉴욕 스타일: 빵은 얇고 소스로 주로 맛을 내고 치즈는 조금 적게 사용
  - 시카고 스타일: 두꺼운 빵에 풍부한 소스와 많은 치즈 사용

- 피자를 만드는 활동은 지점 간에 차이가 없지만, 지점별로 고유한 피자를 만들도록 하기 위해서는 어떻게 설계해야 할까요?

피자 가게 프레임워크

프레임워크 - 유사한 여러 애플리케이션 개발에 쉽게 재사용될 수 있고, 쉽게 확장될 수 있도록 만든 소프트웨어 구조

- 피자를 만드는 공통 활동은 PizzaStore 클래스로..
- 분점마다 고유한 스타일은 PizzaStore의 서브 클래스에..

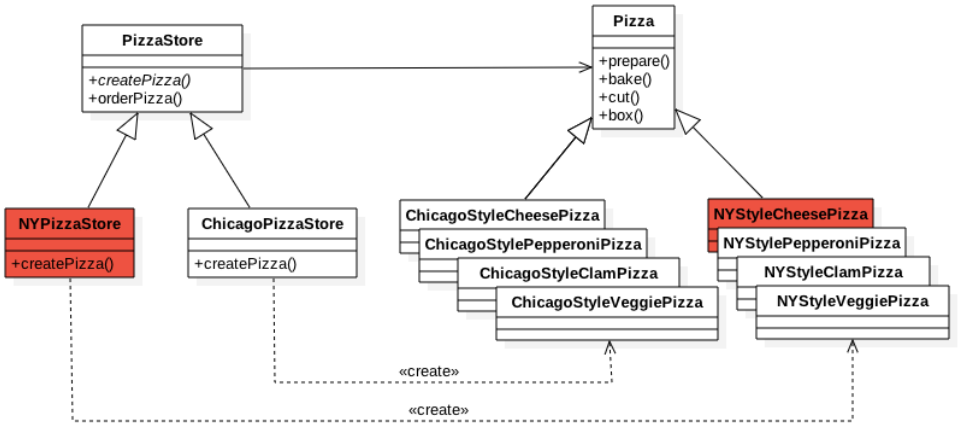
```
public abstract class PizzaStore {

        abstract Pizza createPizza(String item);

        public Pizza orderPizza(String type) {

                Pizza pizza = createPizza(type);

                System.out.println("--- Making a " + pizza.getName() + " ---

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

분점 고유의 스타일은 서브 클래스의 createPizza()메소드에서재정의

분점마다 동일하게 진행되는 공통활동

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzafm/src/hansung/designpatterns/factory/pizzafm/PizzaStore.java

PizzaStore 서브 클래스




```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

```
public class ChicagoPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzafm/src/hansung/designpatterns/factory/pizzafm/NYPizzaStore.java

Pizza 클래스

```
public abstract class Pizza {
        String name;
        String dough;
        String sauce;
        ArrayList<String> toppings = new ArrayList<String>();

        void prepare() {
                System.out.println("Prepare " + name);
        }

        void bake() {
                System.out.println("Bake for 25 minutes at 350");
        }

        void cut() {
                System.out.println("Cut the pizza into diagonal slices");
        }

        void box() {
                System.out.println("Place pizza in official PizzaStore box");
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzafm/src/hansung/designpatterns/factory/pizzafm/Pizza.java

Pizza 서브 클래스 (뉴욕풍 치즈 피자)

```
public class NYStyleCheesePizza extends Pizza {

        public NYStyleCheesePizza() {
                name = "NY Style Sauce and Cheese Pizza";
                dough = "Thin Crust Dough";
                sauce = "Marinara Sauce";

                toppings.add("Grated Reggiano Cheese");
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzafm/src/hansung/designpatterns/factory/pizzafm/NYStyleCheesePizza.java

테스트 코드

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        1  PizzaStore nyStore = new NYPizzaStore();
        2  Pizza pizza = nyStore.orderPizza("cheese");

        System.out.println("Ethan ordered a " + pizza + "₩n");
        ...
```

```
public abstract class PizzaStore {

    abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
        3  Pizza pizza = createPizza(type);

        4  System.out.println("--- Making a " + pizza.getName() + " ---");
           pizza.prepare();
           pizza.bake();
           pizza.cut();
           pizza.box();
           return pizza;
    }
}
```

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzafm/src/hansung/designpatterns/factory/pizzafm/PizzaTestDrive.java

FactoryMethodPattern 예제 실행

- 다운로드 FactoryMethodPattern 예제 링크

  https://github.com/kwanulee/DesignPattern/tree/master/factory/pizzafm

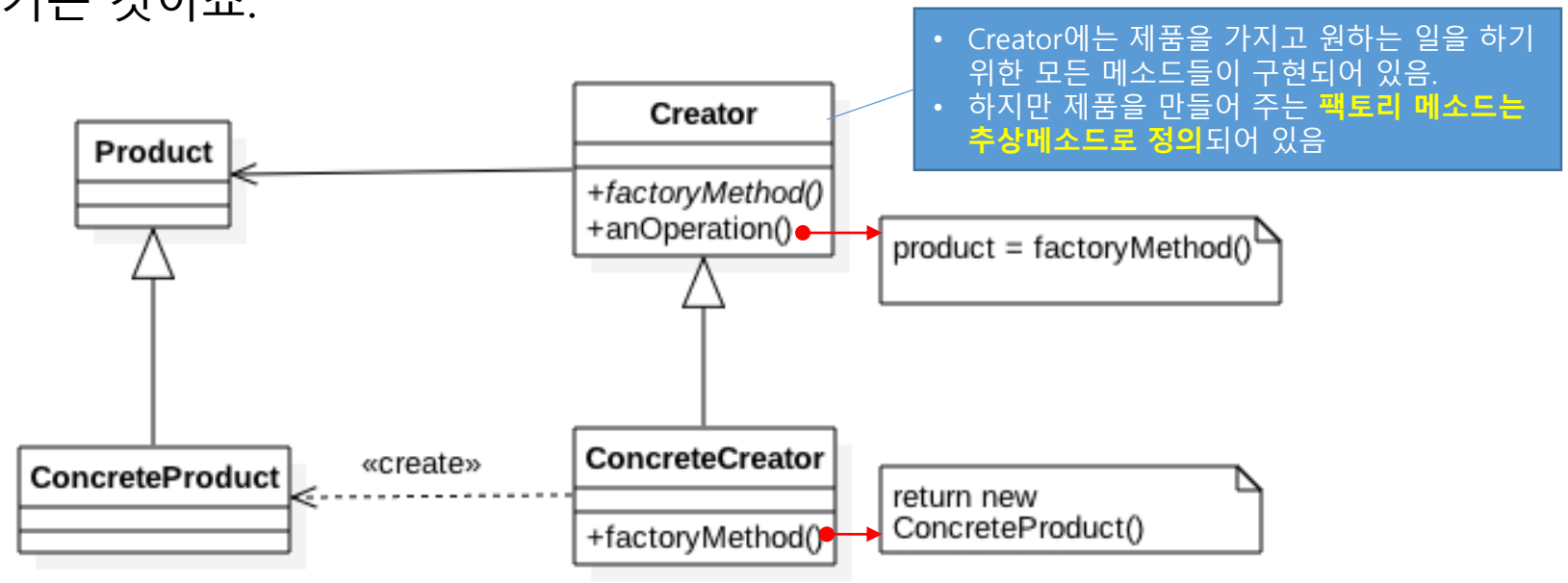
- 실행 결과

```java
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("Ethan ordered a " + pizza + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza + "\n");
```

```
--- Making a NY Style Sauce and Cheese Pizza ---
Prepare NY Style Sauce and Cheese Pizza
Bake for 25 minutes at 350
Cut the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a ---- NY Style Sauce and Cheese Pizza ----
        Thin Crust Dough
        Marinara Sauce
        Grated Reggiano Cheese

--- Making a Chicago Style Deep Dish Cheese Pizza ---
Prepare Chicago Style Deep Dish Cheese Pizza
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a ---- Chicago Style Deep Dish Cheese Pizza ----
        Extra Thick Crust Dough
        Plum Tomato Sauce
```

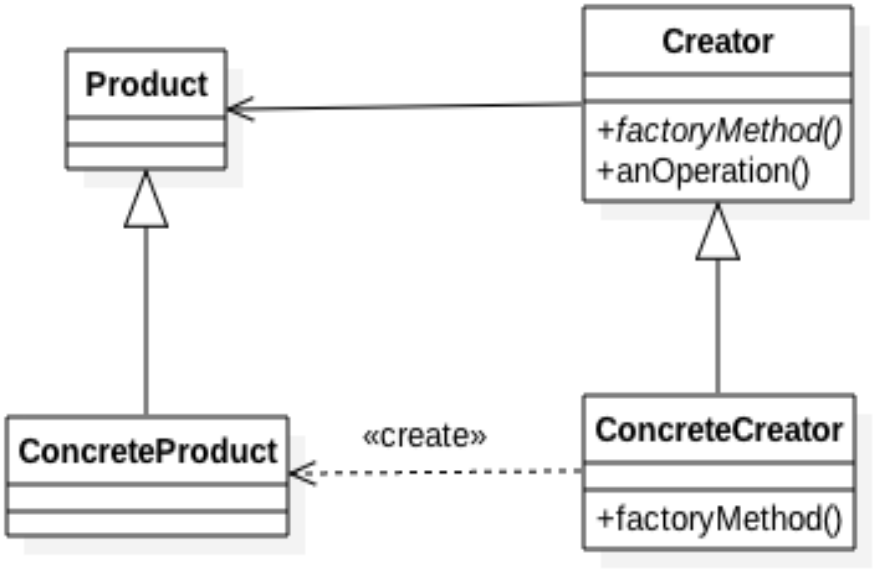
팩토리 메소드 패턴

- 정의
  - 팩토리 메소드 패턴에서는 객체를 생성하기 위한 인터페이스를 정의하는데, 어떤 클래스의 인스턴스를 만들지는 서브클래스에서 결정하게 만듭니다.
  - 팩토리 메소드(factoryMethod())는 객체 생성을 처리하며, 팩토리 메소드를 이용하면 클래스의 인스턴스를 만드는 일을 서브클래스에게 맡기는 것이죠.



퀴즈

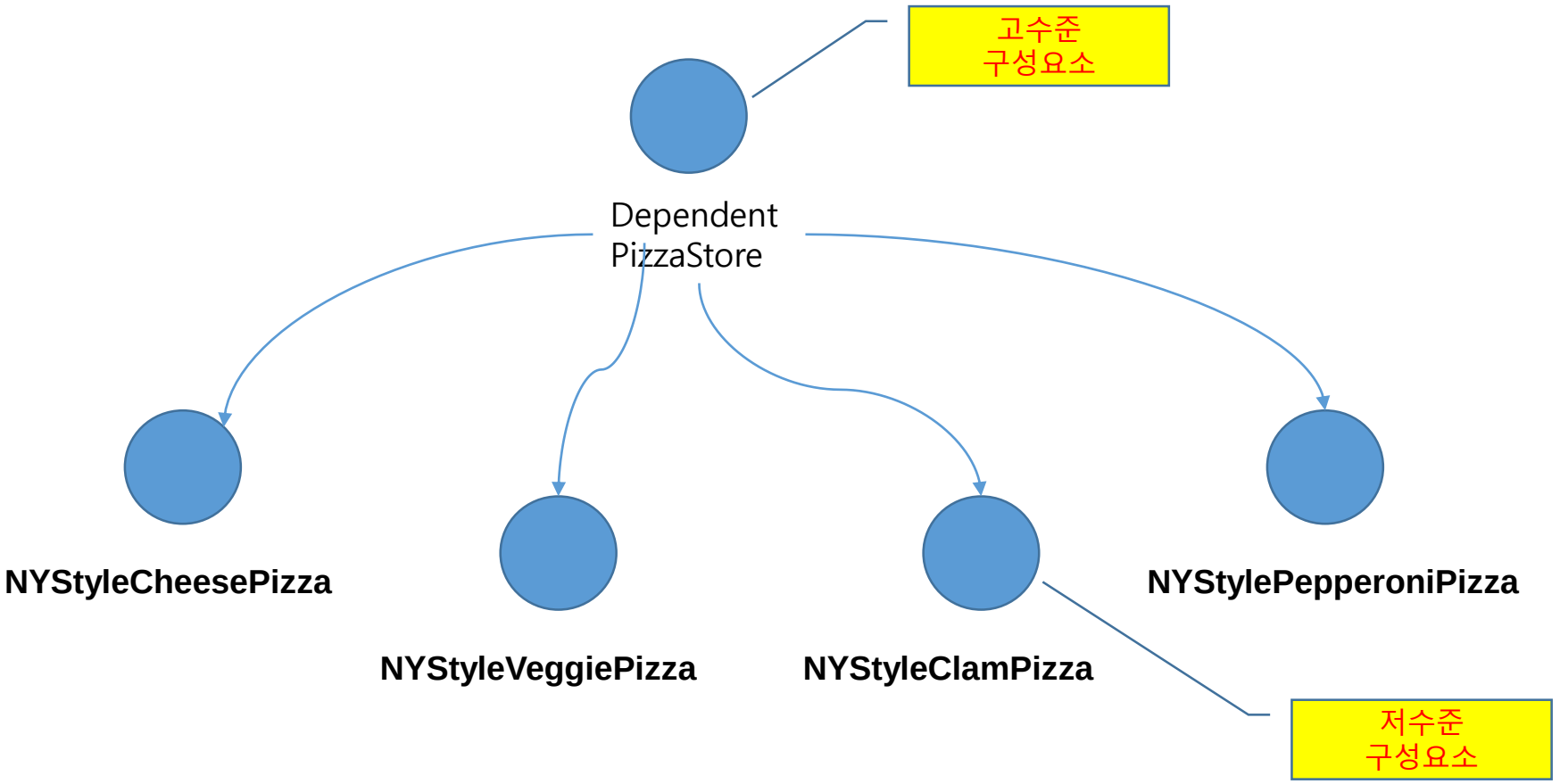
- 다음은 팩토리 메소드 패턴을 나타낸 것이다. 피자가게 예제에서 Creator, ConcreteCreator, Product, ConcreteProduct의 예를 각각 들어라.



객체 의존성 살펴보기 (팩토리 미사용)

```
public class DependentPizzaStore {

        public Pizza orderPizza(String style, String type) {
                Pizza pizza = null;
                if (style.equals("NY")) {
                        if (type.equals("cheese")) {
                                pizza = new NYStyleCheesePizza();
                        } else if (type.equals("veggie")) {
                                pizza = new NYStyleVeggiePizza();
                        } else if (type.equals("clam")) {
                                pizza = new NYStyleClamPizza();
                        } else if (type.equals("pepperoni")) {
                                pizza = new NYStylePepperoniPizza();
                        }
                } else if (style.equals("Chicago")) {
                        …
                } else {
                        System.out.println("Error: invalid type of pizza");
                        return null;
                }
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

객체 의존성 살펴보기 (앞 슬라이드)

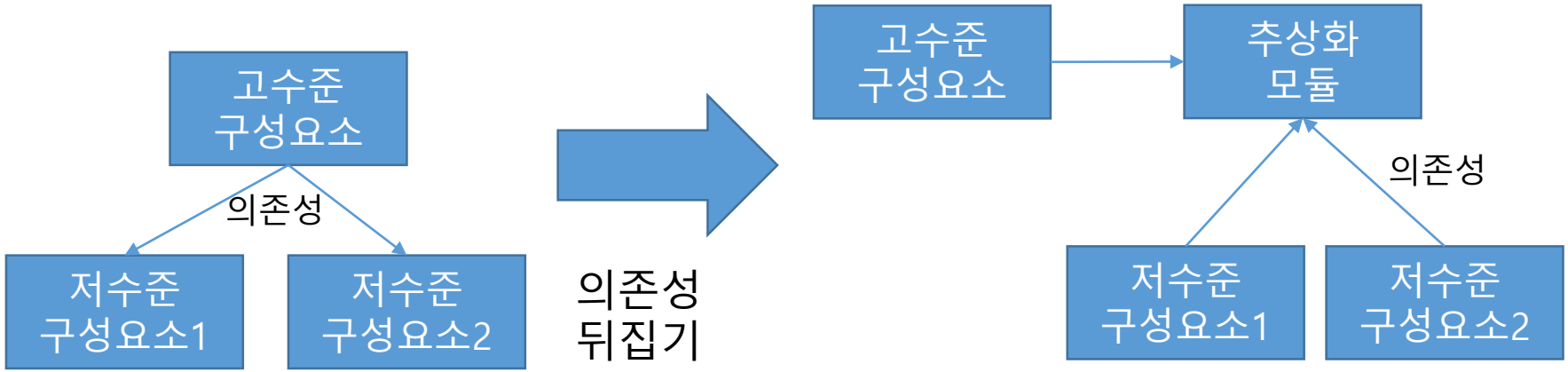


"고수준" 구성요소에서 "저수준" 구성요소로의 의존관계

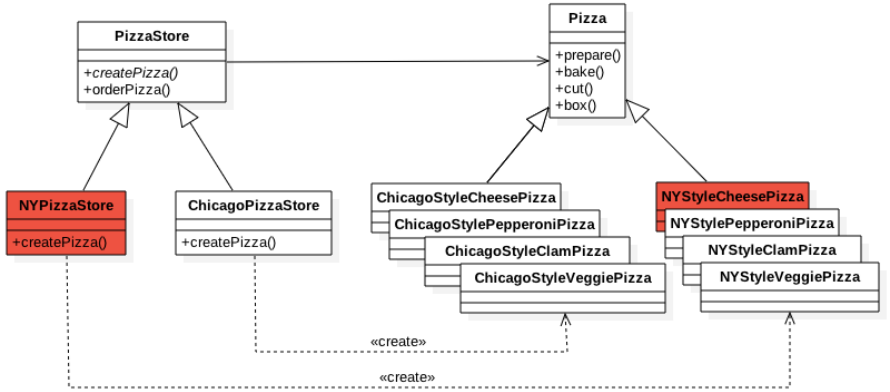
참고, "고수준" 구성요소는 다른 "저수준" 구성요소에 의해 정의되는 또는 "저수준" 구성요소를 제어하는 행동이 들어 있는 구성요소

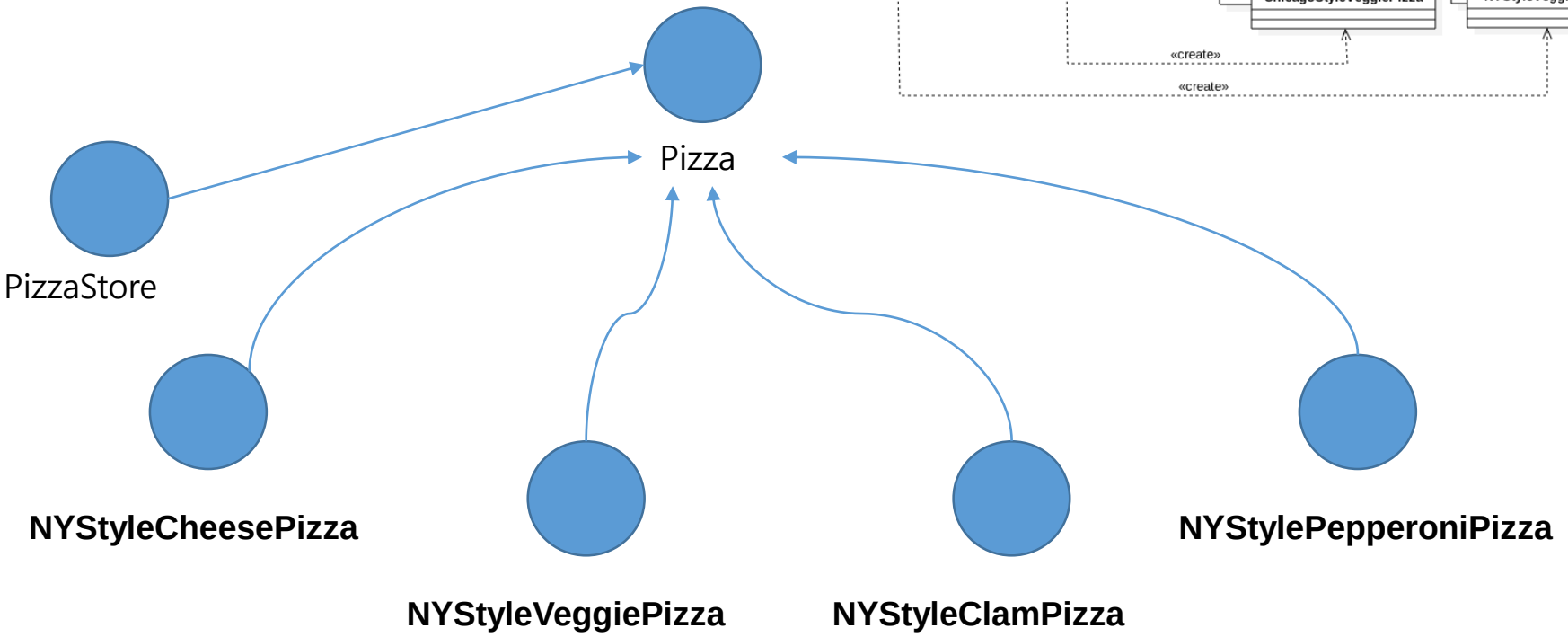
디자인 원칙 (의존성 역전: Dependency Inversion)

추상화된 것에 의존하도록 만들어라.
구상 클래스에 의존하도록 만들지 마라.
("고수준" 구성요소가 "저수준" 구성요소에 의존하면 안된다")



의존성 뒤집기(팩토리 메소드 패턴)





의존성 뒤집기(팩토리 메소드 패턴)

```
public abstract class PizzaStore {

        abstract Pizza createPizza(String item);

        public Pizza orderPizza(String type) {

                Pizza pizza = createPizza(type);

                System.out.println("--- Making a " + pizza.getName() + " ---");
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

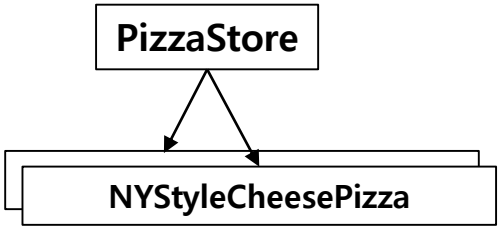
PizzaStore는 추상 클래스인 Pizza에 의존적임

토의

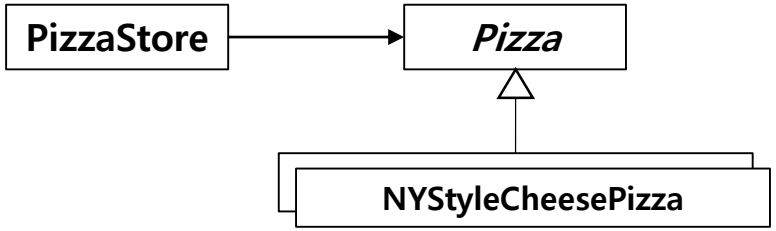
- "구현이 아닌 인터페이스에 맞춰서 프로그래밍하라"와 " 의존성 뒤집기 원칙"의 공통점과 차이점은?

의존성 뒤집기 원칙에서 뭘 뒤집는다는 거죠?

- 일반적인 디자인 절차
  - 상위 수준 모듈 (정책 결정 및 제어 모듈)에서 하위 수준 모듈 (구체적인 알고리즘이나 방법)을 호출



- 의존성 뒤집기 방식
  - 변경 가능한 하위 수준 모듈을 바탕으로 추상화된 모듈 정의
  - 추상화된 모듈을 이용하여 상위 수준 모듈 정의



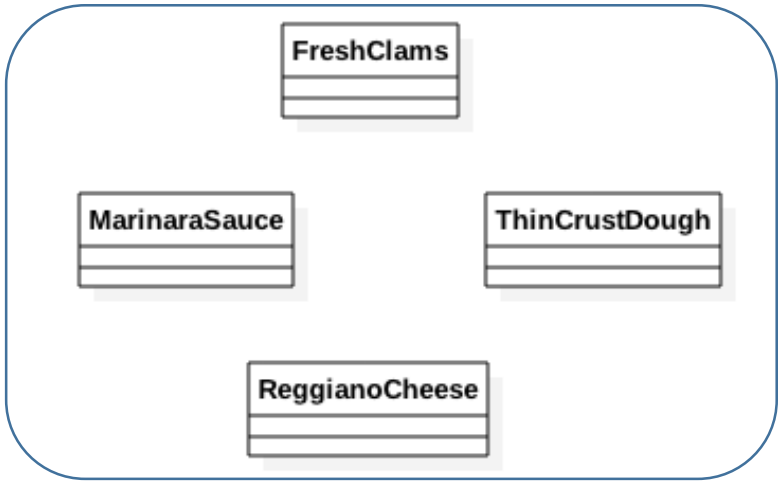
피자 가게의 새로운 요구사항

- 현 상황
  - 모든 분점에서 공통 프레임워크에 따라 정해진 절차로 피자 생산
  - 각 분점 별로 특성을 반영한 피자 생산
- 문제 상황
  - 몇몇 분점에서 자잘한 재료를 더 싼 재료로 바꿔서 원가 절감

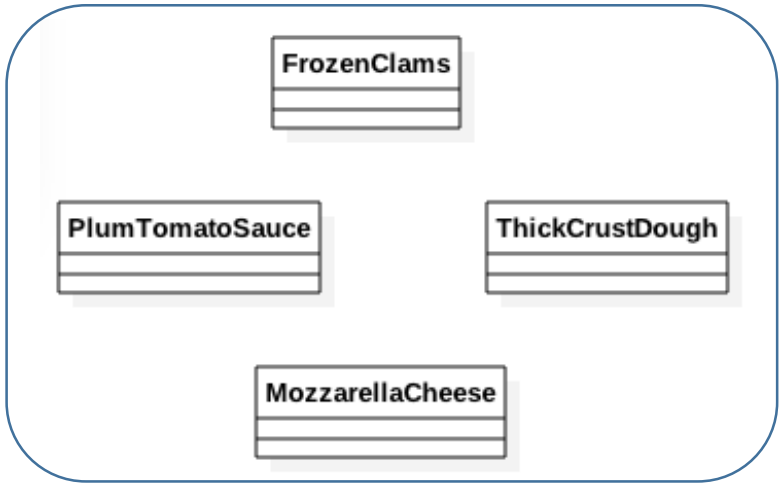
  →원재료의 품질 관리 방안 필요
- 해결 방안
  - 원재료를 생산하는 팩토리를 만들고 팩토리로부터 생산된 원재료를 분점에 배달

원재료 패밀리

- 분점 마다 사용하는 **원재료 종류(예, 소스, 치즈, 반죽 등)는 동일**하지만, 각 원재료 종류 별로 사용되는 원재료는 분점마다 상이
  - 원재료 패밀리: 특정 분점의 피자를 만드는데 필요한 원재료들의 집합



뉴욕 분점의
원재료 패밀리



시카고 분점의
원재료 패밀리

원재료 공장 만들기 (팩토리 인터페이스)

```
public interface PizzaIngredientFactory {

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

}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/PizzaIngredientFactory.java

뉴욕 원재료 공장 만들기

```
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(); // 시카고 공장과 동일
        }

        public Clams createClam() {
                return new FreshClams();  // 뉴욕은 바닷가라서 신선한…
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/NYPizzaIngredientFactory.java

피자 클래스 변경 - Pizza

- 피자 클래스에서 팩토리에서 생산한 원재료만 사용하도록 함.

```
public abstract class Pizza {
        String name;
        Dough dough;
        Sauce sauce;
        Veggies veggies[];
        ...
        PizzaIngredientFactory ingredientFactory;

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

        abstract void prepare();
        void bake() { System.out.println("Bake for 25 minutes at 350"); }
        void cut() { System.out.println("Cutting the pizza into diagonal slices"); }
        void box() { System.out.println("Place pizza in official PizzaStore box"); }
        ....
}
```

원재료 공장

피자를 만드는 데 필요한 재료들을 정돈.
모든 원재료는 원재료 팩토리에서 가져옴

공통 메소드

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/Pizza.java

피자 클래스 변경 - CheesePizza

- 팩토리 메소드 패턴에서의 NYStyle**CheesePizza**와 ChicagoStyle**CheesePizza**는 지역별로 다른 재료를 사용함.
- 지역별로 다른 점은 원재료 공장에서 커버해 줌

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

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
            super(ingredientFactory);
        }

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

ingredientFactory 멤버변수 초기화

지역별 다른점은 원재료 공장에서 커버

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/CheesePizza.java

피자 클래스 변경 - ClamPizza

- ClamPizza의 경우 CheesePizza에 비해서 Clam 원재료 준비 부분이 추가됨

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

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
            super(ingredientFactory);
        }

        void prepare() {
            System.out.println("Preparing " + name);
            dough = ingredientFactory.createDough();
            sauce = ingredientFactory.createSauce();
            cheese = ingredientFactory.createCheese();
            clam = ingredientFactory.createClam();
        }
}
```

ingredientFactory 멤버변수 초기화

지역별 다른점은 원재료 공장에서 커버

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/CheesePizza.java

피자 가게

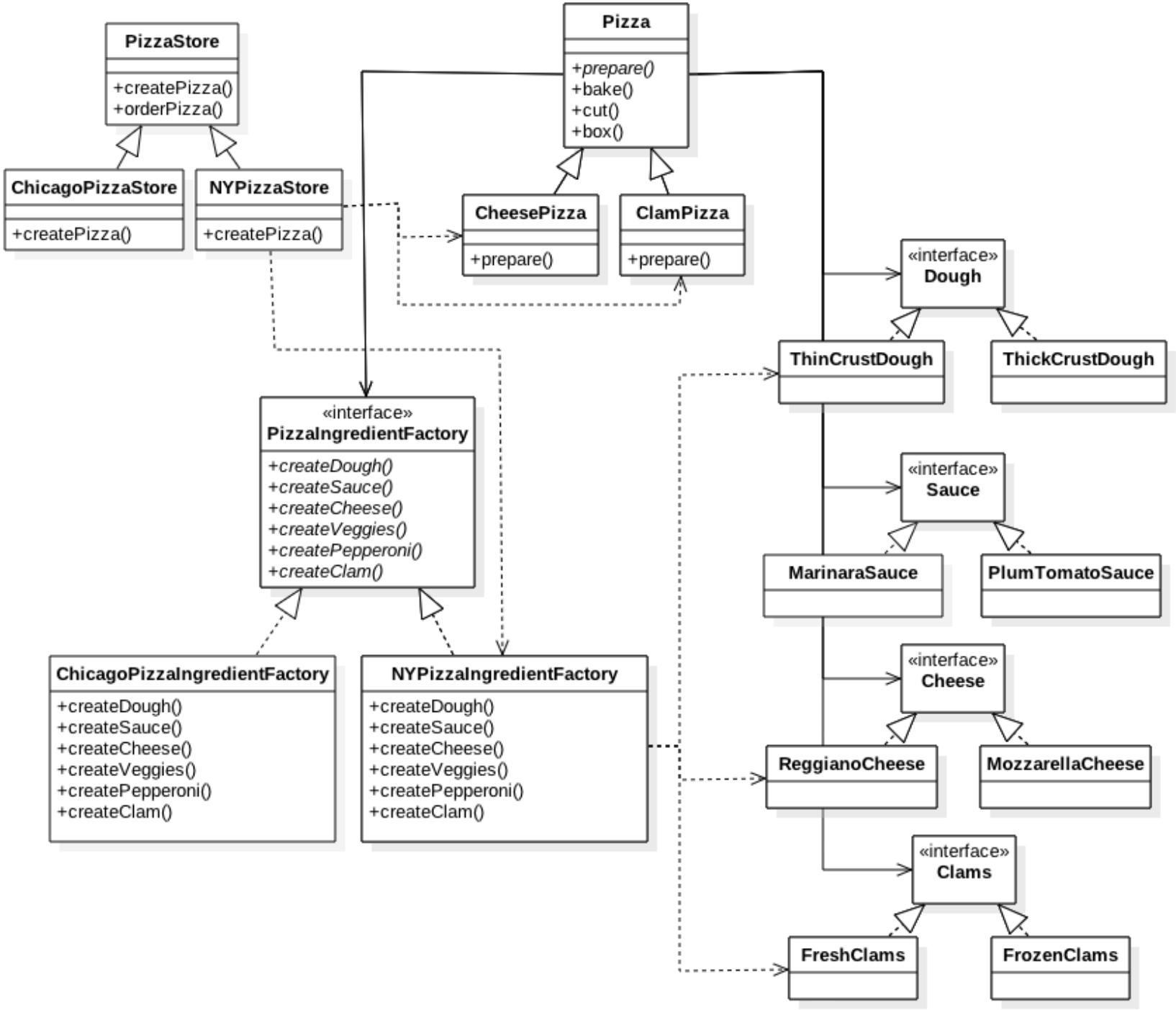
```
public abstract class PizzaStore {

        protected abstract Pizza createPizza(String item);

        public Pizza orderPizza(String type) {
                Pizza pizza = createPizza(type);
                System.out.println("--- Making a " + pizza.getName() + " ---");
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/NYPizzaStore.java

피자 가게

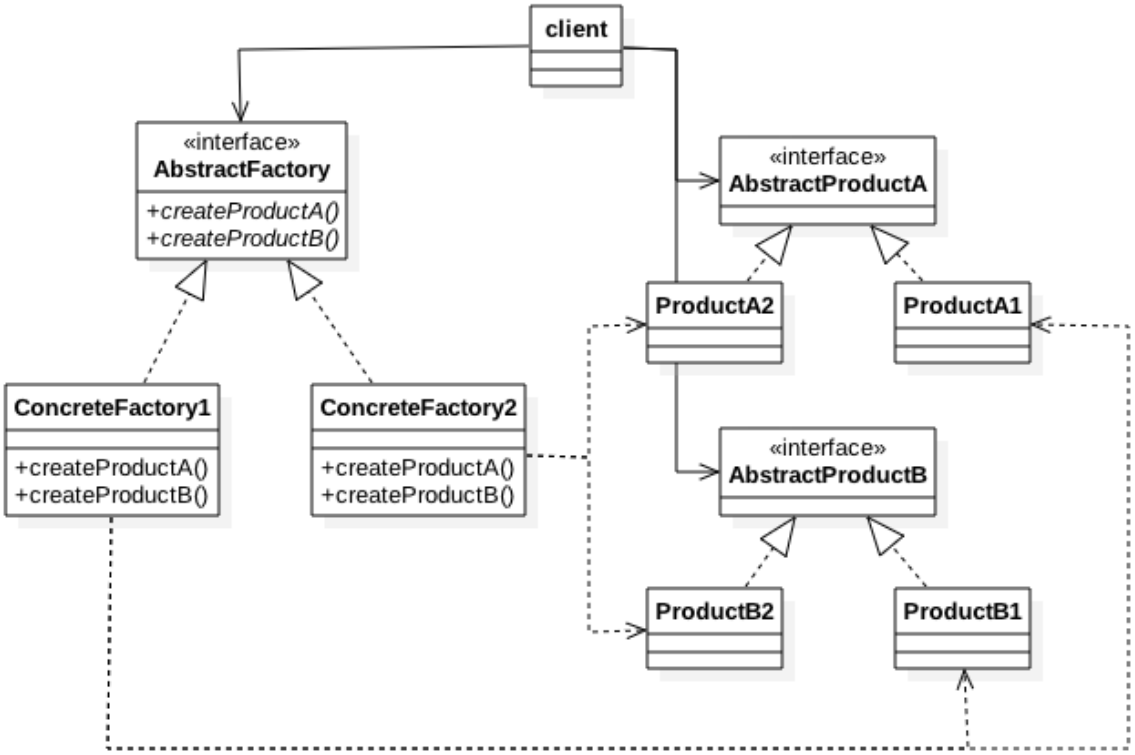
```
public class NYPizzaStore extends PizzaStore {

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

                if (item.equals("cheese")) {
                        pizza = new CheesePizza(ingredientFactory);
                        pizza.setName("New York Style Cheese Pizza");

                } else if (item.equals("veggie")) {
                        pizza = new VeggiePizza(ingredientFactory);
                        pizza.setName("New York Style Veggie Pizza");

                } else if (item.equals("clam")) {
                        pizza = new ClamPizza(ingredientFactory);
                        pizza.setName("New York Style Clam Pizza");

                …
                return pizza;
        }
}
```

https://github.com/kwanulee/DesignPattern/blob/master/factory/pizzaf/src/hansung/designpatterns/factory/pizzaaf/NYPizzaStore.java


PizzaStore
+createPizza()
+orderPizza()
Pizza
+prepare()
+bake()
+cut()
+box()
ChicagoPizzaStore
+createPizza()
NYPizzaStore
+createPizza()
CheesePizza
+prepare()
ClamPizza
+prepare()
«interface»
Dough
ThinCrustDough
ThickCrustDough
«interface»
PizzaIngredientFactory
+createDough()
+createSauce()
+createCheese()
+createVeggies()
+createPepperoni()
+createClam()
«interface»
Sauce
MarinaraSauce
PlumTomatoSauce
ChicagoPizzaIngredientFactory
+createDough()
+createSauce()
+createCheese()
+createVeggies()
+createPepperoni()
+createClam()
NYPizzaIngredientFactory
+createDough()
+createSauce()
+createCheese()
+createVeggies()
+createPepperoni()
+createClam()
«interface»
Cheese
ReggianoCheese
MozzarellaCheese
«interface»
Clams
FreshClams
FrozenClams

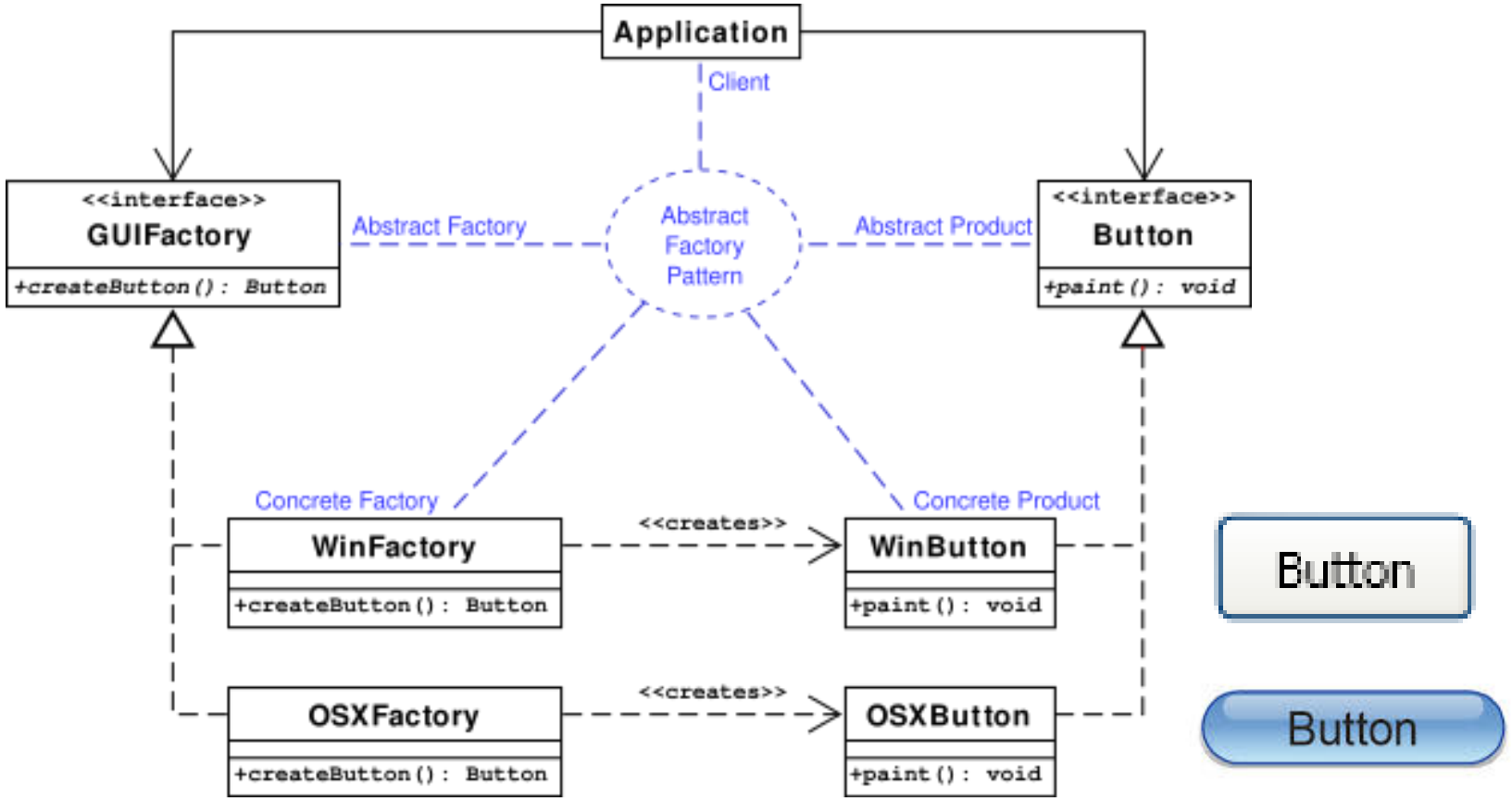
추상 팩토리 패턴

- 정의
  - 추상 팩토리 패턴에서는 인터페이스를 이용하여 서로 연관된, 또는 의존하는 객체들의 패밀리를 구상 클래스를 명시하지 않고도 생성할 수 있습니다.



추상 팩토리 패턴의 활용 예

- GUIFactory (https://en.wikipedia.org/wiki/Abstract_factory_pattern)



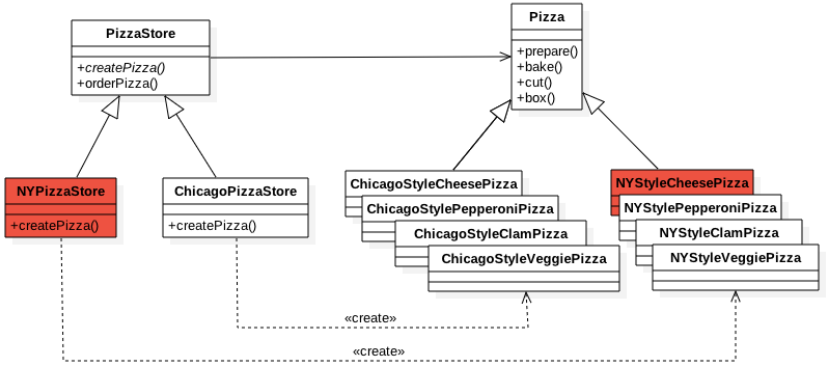
추상 팩토리 패턴의 장단점

- 장점
  - 많은 수의 연관된 제품들의 생성을 상황 변화(예, 지역 변화, 운영체제 변화 등)에 따라 한번에 교체할 수 있도록 있다.

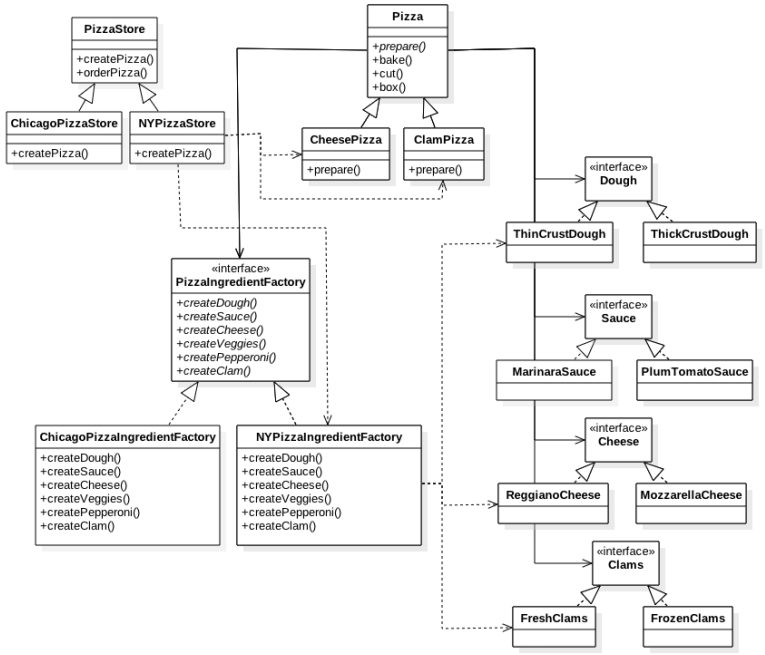
- 단점
  - 생성시킬 제품(product)이 추가되는 경우에는 AbstractFactory의 인터페이스가 추가되어야 하므로, 많은 ConcreteFactory 클래스에 영향을 준다.

팩토리 메소드 패턴 vs. 추상 팩토리 패턴

- 팩토리 메소드 패턴
  - 클래스 상속 통해 객체 생성을 서브 클래스에 위임



- 추상 팩토리 패턴
  - 추상 팩토리 인터페이스를 구현한 팩토리 객체 인스턴스를 구성하고, 객체 생성은 팩토리 객체 인스턴스에 위임.



핵심 정리

- 팩토리를 쓰면 객체 생성을 캡슐화 할 수 있습니다.
- 간단한 팩토리는 엄밀하게 말해서 디자인 패턴은 아니지만, 클라이언트와 구상 클래스를 분리시키기 위한 간단한 기법으로 활용할 수 있습니다.
- 팩토리 메소드 패턴에서는 상속을 활용합니다. 객체 생성이 서브클래스에게 위임되죠. 서브클래스에서는 팩토리 메소드를 구현하여 객체를 생산합니다.
- 추상 팩토리 패턴에서는 객체 구성을 활용합니다. 객체 생성이 팩토리 인터페이스에서 선언한 메소드들에서 구현되죠.

핵심 정리

- 모든 팩토리 패턴에서는 애플리케이션의 구상 클래스에 대한 의존성을 줄여줌으로서 느슨한 결합을 도와줍니다
- 추상 팩토리 패턴은 구상 클래스에 직접 의존하지 않고도 서로 관련된 객체들로 이루어진 제품 패밀리를 만들기 위한 용도로 쓰입니다.
- 의존성 역전 원칙을 따르면 구상 형식에 대한 의존을 피하고 추상화를 지향할 수 있습니다.
- 팩토리는 구상 클래스가 아닌 추상 클래스/인터페이스에 맞춰서 코딩할 수 있게 해 주는 강력한 기법입니다.