고급객체지향 프로그래밍
강의노트 #06

Factory Pattern

조용주
ycho@smu.ac.kr

팩토리 메소드 패턴 (Factory Method Pattern)

- 목적
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - 객체 생성용 인터페이스 정의. 하지만 서브클래스가 어떤 클래스를 인스턴스화(객체 생성)할 지 결정할 수 있도록 함. 팩토리 메소드는 객체 생성을 서브 클래스에서 할 수 있도록 미룰 수 있도록 만들어줌

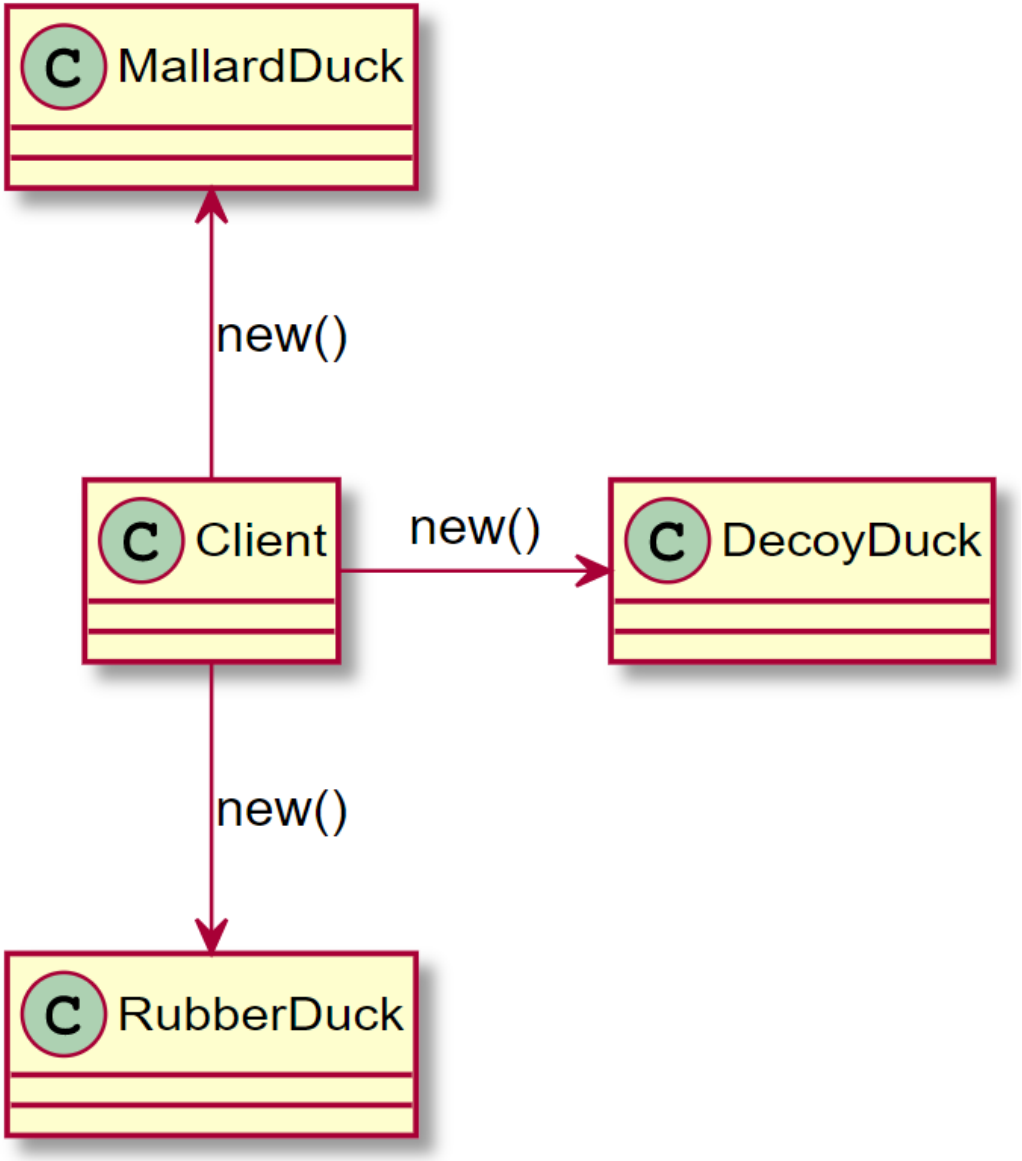
추상 팩토리 패턴 (Abstract Factory Pattern)

- 목적
  - Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
  - 구체적인 클래스를 명시하지 않고 관련된 혹은 의존적인 클래스들을 생성할 수 있는 인터페이스 제공

문제

- 객체를 생성하는 'new' 의 문제
  - new 는 인터페이스가 아니라 실제 클래스 (concrete class) 를 생성
    - 예 : new 를 호출해서 DecoyDuck, MallardDuck, RubberDuck 등을 생성
    - OCP 에 어긋남 (not closed for modification)
      - 생성할 객체가 늘어나면 코드 수정 필요
      - 클래스가 많아지거나 변경되면 클라이언트 측 변경이 많아짐

문제



디자인 패턴 요소

요소	설명
이름	팩토리 메소드 (Factory Method), 추상 팩토리 (Abstract Factory)
문제	실제로 구현되는 클래스의 객체를 생성할 때 객체의 종류가 달라지면 클라이언트 코드를 수정해야 하는 것이 너무 많음
해결방안	생성을 분리해서 캡슐화 시킴
결과	사용할 객체가 많거나 객체를 생성하는 방법이 변경되어도 연쇄적인 수정이 적어짐

사례 1 – 피자 가게

- 피자 가게를 운영하고 있다고 가정
- 피자 주문을 위해 아래 코드를 작성함

```
void prepareToBoxing(Pizza pizza) {
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
}

Pizza orderPizza() {
  Pizza pizza = new Pizza();

  prepareToBoxing(pizza);
  return pizza;
}
```

사례 1 – 피자 가게

- 피자 종류가 여러 가지 있으면 코드를 수정해야 함

```
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();
  }

  prepareToBoxing(pizza);
  return pizza;
}
```

사례 1 – 피자 가게

- 피자 종류가 여러 가지 있으므로 코드를 수정함

```
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();
  } else if (type.equals("clam") {
    pizza = new ClamPizza();
  } else if (type.equals("veggie") {
    pizza = new VeggiePizza();
  }
  prepareToBoxing(pizza);
  return pizza;
}
```

바뀌는 부분

바뀌지 않는 부분

사례 1 – 피자 가게

- 객체 생성 부분을 캡슐화

```
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;
  }
}
```

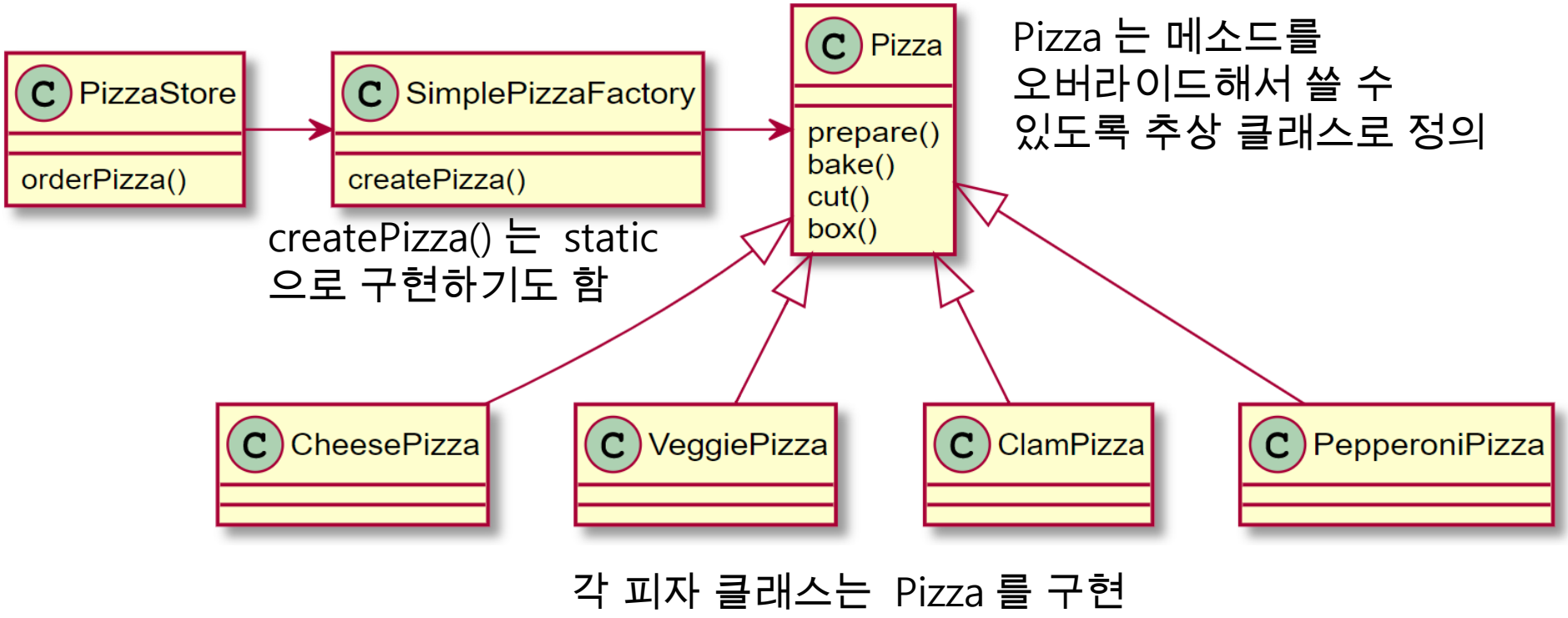
사례 1 – 피자 가게

- SimplePizzaFactory 사용

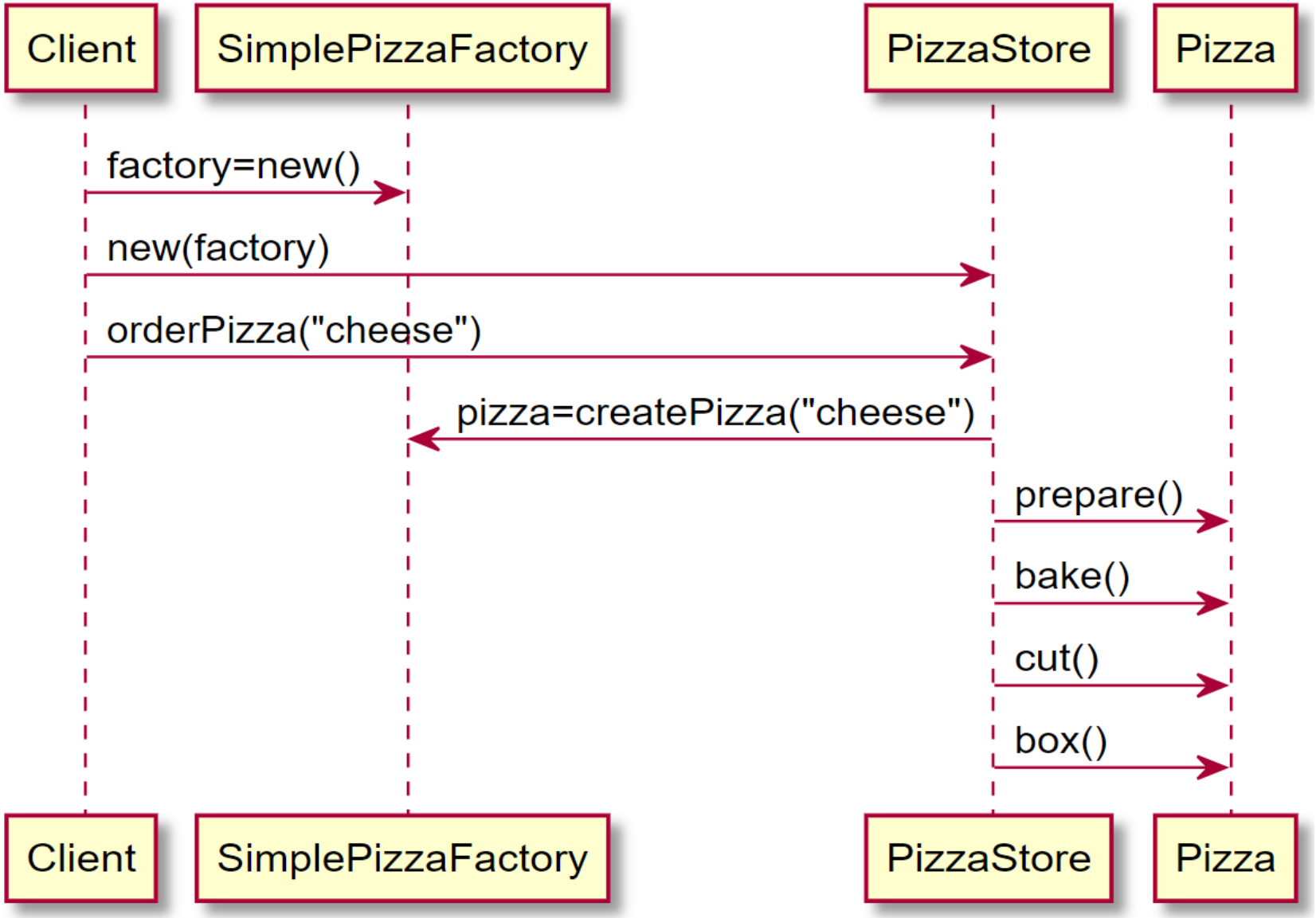
```
public class PizzaStore {
  SimplePizzaFactory factory;
  public PizzaStore(SimplePizzaFactory factory) {
    this.factory = factory;
  }
  public Pizza orderPizza(String type) {
    Pizza pizza = null;
    pizza = factory.createPizza(type);
    prepareToBoxing(pizza);
    return pizza;
  }
  void prepareToBoxing(Pizza pizza) {
    … // 기존 코드
  }
}
```

사례 1 – 피자 가게

- 피자 가게 프로그램의 클래스 다이어그램



사례 1 – 피자 가게

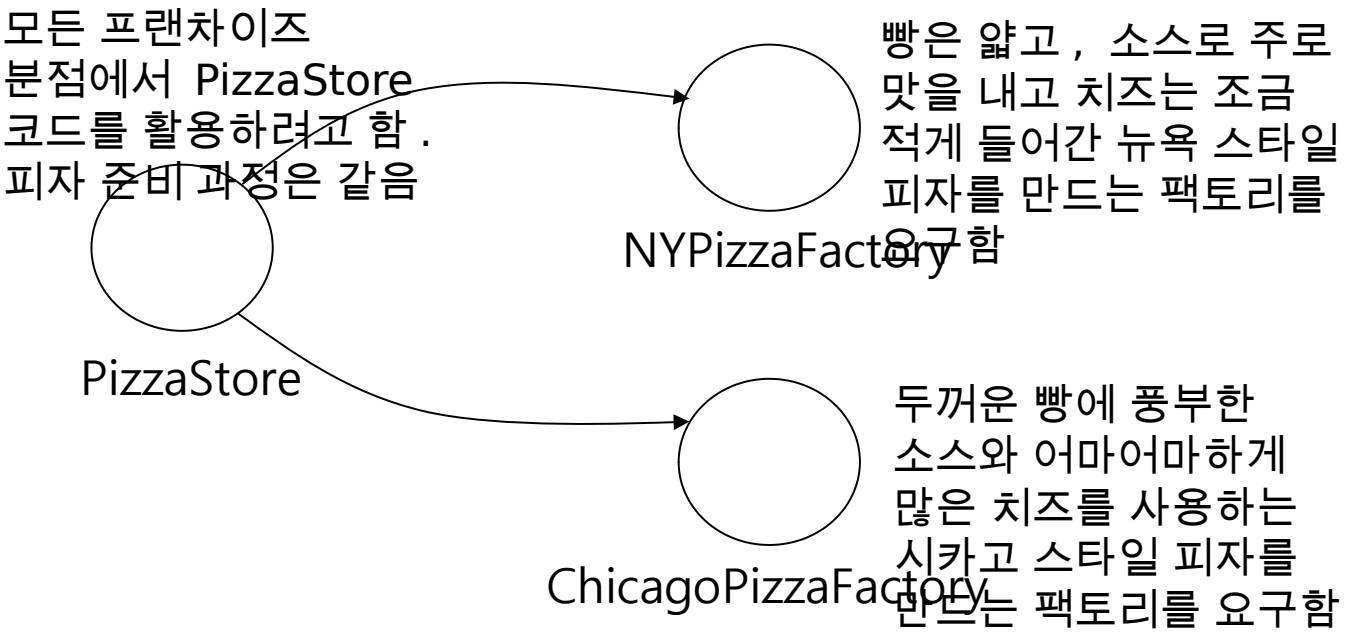

Client
SimplePizzaFactory
PizzaStore
Pizza
factory=new()
new(factory)
orderPizza("cheese")
pizza=createPizza("cheese")
prepare()
bake()
cut()
box()
Client
SimplePizzaFactory
PizzaStore
Pizza

Simple Factory

- Simple Factory 가 어느 객체를 생성할지 판단하고 , 사용자 측에 맞는 객체 반환
  - 일반적으로 "if" 문에서 문자열에 따라 생성할 객체를 결정
  - 패턴이라고 볼 수는 없음

사례 2 – 피자 프랜차이즈 사업

- 프랜차이즈 사업을 하면서 각 지점마다 해당 지역의 특성과 입맛을 반영하여 다른 스타일의 피자 (뉴욕 , 시카고 , 캘리포니아 스타일 등) 를 만들려고 함
  - 어떻게 지역별 차이점을 적용시킬까 ?



사례 2 – 피자 프랜차이즈 사업

- SimplePizzaFactory 대신 세 가지 다른 팩토리를 PizzaStore 에서 사용하도록 하면 됨

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");

ChicagoPizzaFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

- 문제
  - PizzaStore 가 피자 생성 과정과 분리되어 있어, 유연성은 보장되나, 일괄적인 처리가 어려울 수 있음 (PizzaStore 클래스의 orderPizza 프로세스)
  - 피자 스토어마다 다른 처리 과정이 나타날 수 있음

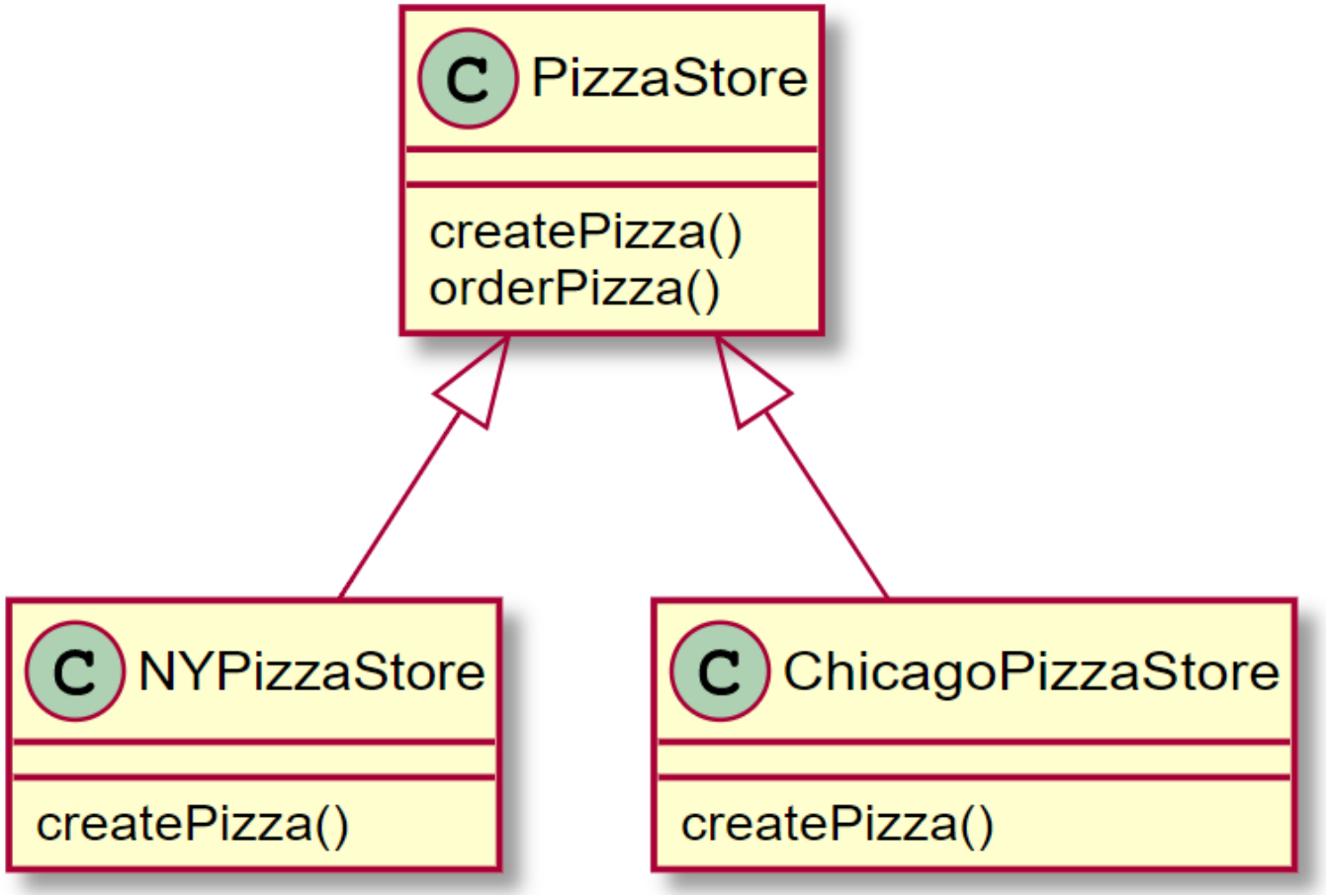
사례 2 – 피자 프랜차이즈 사업

- 피자 가게와 피자 제작 과정 전체를 하나로 묶어주는 프레임워크를 만들기로 함
  - 유연성은 지켜야 함 (SimpleFactory 이전 코드로 못돌아감)
  - createPizza() 메소드를 PizzaStore 에 다시 넣고 , 추상 메소드로 선언
- 분점마다 달라질 수 있는 것은 피자의 스타일 . 주문 시스템은 모든 분점에서 똑같이 진행됨
- orderPizza() 에서는 어떤 피자가 만들어지는지 알 수 없음

사례 2 – 피자 프랜차이즈 사업

```
public abstract class PizzaStore {
  void prepareToBoxing(Pizza pizza) {
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }

  public Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);
    prepareToBoxing(pizza);
    return pizza;
  }

  //  팩토리 메소드
  abstract Pizza createPizza(String type);
}
```

사례 2 – 피자 프랜차이즈 사업



사례 2 – 피자 프랜차이즈 사업

```
public class NYPizzaStore extends PizzaStore {
  Pizza createPizza(String 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();
    }
  }
}
```

사례 2 – 피자 프랜차이즈 사업

```
public class ChicagoPizzaStore extends PizzaStore {
  Pizza createPizza(String 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();
    }
  }
}
```

팩토리 메소드 (Factory Method)

- 팩토리 메소드는 객체 생성을 처리하며, 팩토리 메소드를 이용하면 객체를 생성하는 작업을 서브클래스에 캡슐화시킬 있음
- 수퍼클래스에 있는 클라이언트 코드와 서브클래스에 있는 객체 생성 코드를 분리시킬 수 있음

```
abstract Product factoryMethod(String type)
```

- 팩토리 메소드는 특정 제품 (객체) 를 반환
  - 해당 객체는 수퍼클래스에서 정의한 메소드 내에서 사용
- 팩토리 메소드는 클라이언트 (수퍼클래스의 orderPizza() 같은 코드) 에서 실제로 생성되는 실제 객체가 무엇인지 알 수 없게 만드는 역할

피자 클래스 구현

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

  void prepare() {
    System.out.println("Preparing " + name);
    System.out.println("Tossing dough…");
    System.out.println("Adding sauce…");
    System.out.println("Adding toppings: ");
    for (int i = 0; i < toppings.size(); i++) {
      System.out.println("  " + toppings.get(i));
    }
  }
  void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }
```

피자 클래스 구현

```
  void cut() {
    System.out.println("Cutting the pizza into
diagonal slices");
  }
  void box() {
    System.out.println("Place pizza in official
PizzaStore box");
  }
  public String getName() {
    return name;
  }
}
```

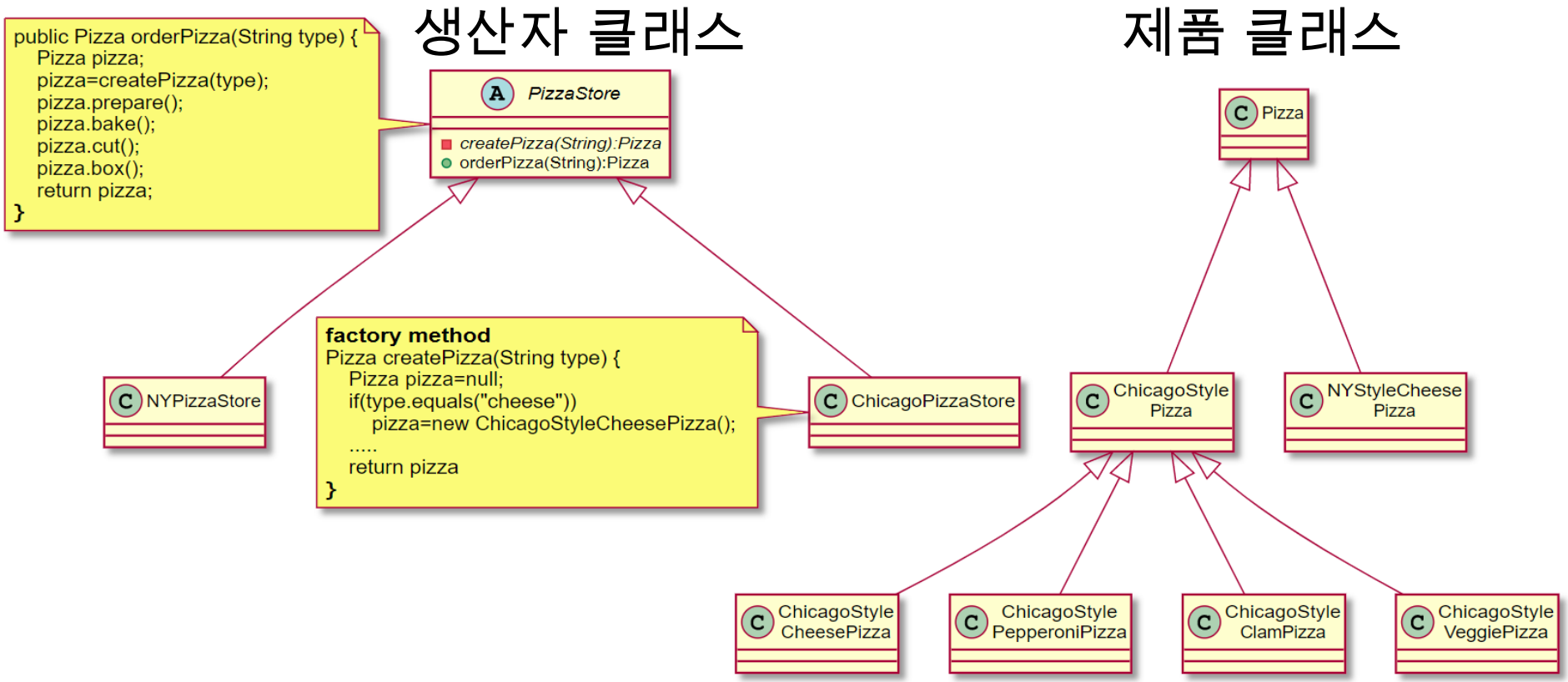
```
public class NYStyleCheesePizza extends Pizza {
  public NYStyleCheesePizza() {
    name = "NY Style Sauce and Cheese Pizza";
    dough = "Thin Crust Dough";
    sauce = "Marinara Sauce";
    toppings.add("Grated Reggiano Cheese");
  }
}

public class ChicagoStyleCheesePizza extends Pizza
{
  public ChicagoStyleCheesePizza () {
    name = "Chicago Style Deep Dish Cheese Pizza";
    dough = "Extra Thick Crust Dough";
    sauce = "Plum Tomato Sauce";
    toppings.add("Shredded Mozzarella Cheese");
  }
  void cut() {
    System.out.println("Cutting the pizza into
square slices");
  }
}
```

main 함수 구현

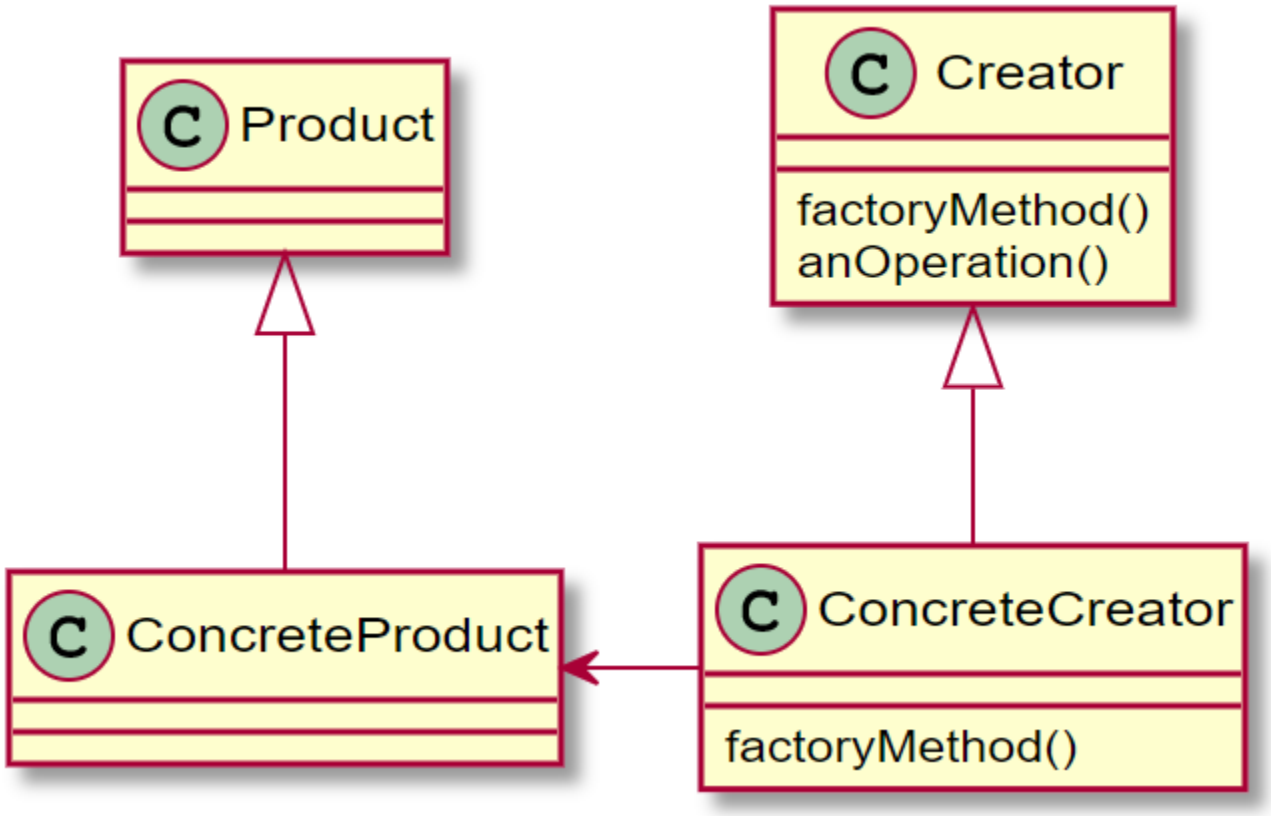
```
public class PizzaTestDrive {
  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();
    Pizza pizza = nyStore.orderPizza("cheese");
    System.out.println("Ethan ordered a "
                          + pizza.getName() + "\n");
    pizza = chicagoStore.orderPizza("cheese");
    System.out.println("Joel ordered a "
                            + pizza.getName + "\n");
  }
}
```

팩토리 메소드 패턴

- 병렬 클래스 계층 구조



팩토리 메소드 패턴

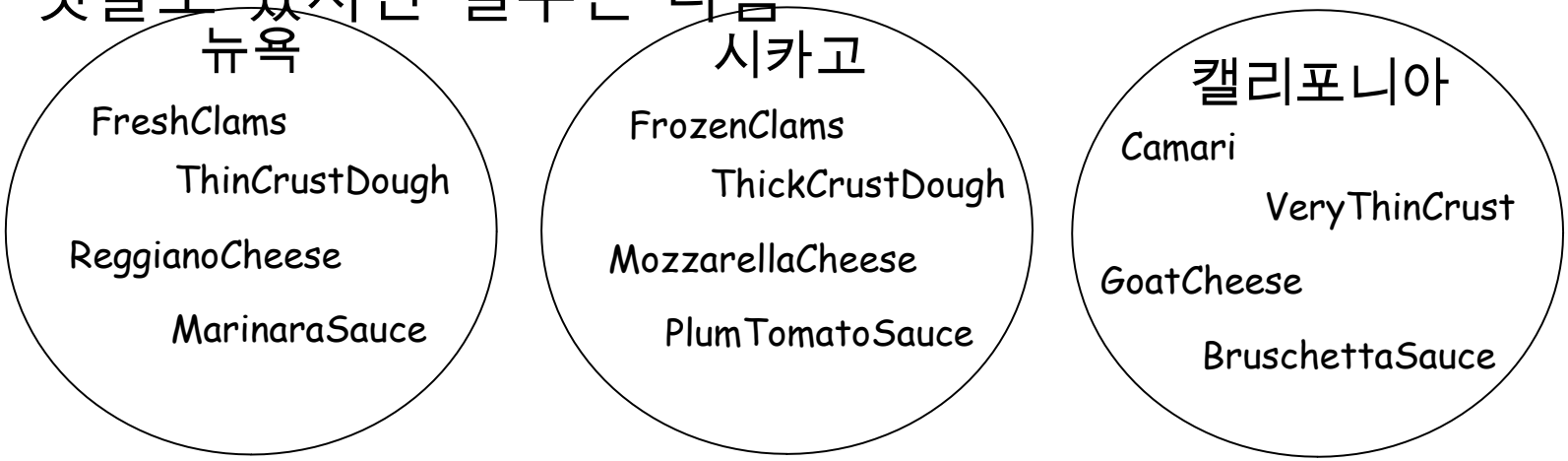

Product
Creator
factoryMethod()
anOperation()
ConcreteProduct
ConcreteCreator
factoryMethod()

만약 팩토리 메소드 패턴을 사용하지 않는다면?

```
public class DependentPizzaStore {
  public Pizza createPizza(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 (style.equals("Chicago")) {
      if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
      } else if (type.equals("veggie")) {
        pizza = new ChicagoStyleVeggiePizza();
      }
      …
    }
    …
}
```

사례 3 – 피자 원재료 품질 관리

- 분점에서 좋은 재료를 사용하도록 관리할 수 있을까 ?
  - 원재료를 생산하는 공장 (팩토리) 를 만들고 분점까지 재료를 제공
  - 문제는 분점이 떨어져 있고 , 지점마다 재료들이 같은 것들도 있지만 일부는 다름



사례 3 – 피자 원재료 품질 관리

- 원재료를 생산할 팩토리 인터페이스 정의

```
public interface PizzaIngredientFactory {
   public Dough createDough();
   public Sauce createSauce();
   public Cheese createCheese();
   public Veggies[] createVeggies();
   public Pepperoni createPepperoni();
   public Clams createClam();
}
```

사례 3 – 피자 원재료 품질 관리

- 뉴욕 원재료 공장

```
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;
  }
```

사례 3 – 피자 원재료 품질 관리

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

사례 3 – 피자 원재료 품질 관리

- 새로운 피자 클래스

```
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("Bake for 25 minutes at 350");
  }
  void cut() {
    System.out.println("Cutting the pizza into diago-
nal slices");
```

사례 3 – 피자 원재료 품질 관리

```
  void cut() {
    System.out.println("Cutting the pizza into diago-
nal slices");
  }
  void box() {
    System.out.println("Place pizza in official Piz-
zaStore box");
  }
  void setName(String name) {
    this.name = name;
  }
  String getName() {
    return name;
  }
  public String toString() {
    //  피자 이름 출력
  }
}
```

사례 3 – 피자 원재료 품질 관리

- 팩토리 메소드 패턴을 이용한 코드에서 NYCheesePizza 와 ChicagoCheesePizza 클래스는 지역별로 다른 재료를 사용한다는 것만 빼면 같음
  - 재료만 다를 뿐 결국 준비 단계는 같음
  - 따라서 피자마다 지역별로 따로 만들 필요가 없음
  - 지역별로 다른 재료들은 원재료 공장에서 만들어줌

사례 3 – 피자 원재료 품질 관리

- 치즈 피자 클래스

```
public class CheesePizza extends Pizza {
  PizzaIngredientFactory ingredientFactory;
  public CheesePizza(PizzaIngredientFactory
                                    ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
  }
  void prepare() {
    System.out.println("Preparing " + name);
    dough = ingredientFactory.createDough();
    sauce = ingredientFactory.createSauce();
    cheese = ingredientFactory.createCheese();
  }
}
```

사례 3 – 피자 원재료 품질 관리

- 조개 피자 클래스

```
public class ClamPizza extends Pizza {
  PizzaIngredientFactory ingredientFactory;
  public ClamPizza(PizzaIngredientFactory
                                  ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
  }
  void prepare() {
    System.out.println("Preparing " + name);
    dough = ingredientFactory.createDough();
    sauce = ingredientFactory.createSauce();
    cheese = ingredientFactory.createCheese();
    clam = ingredientFactory.createClam();
  }
}
```

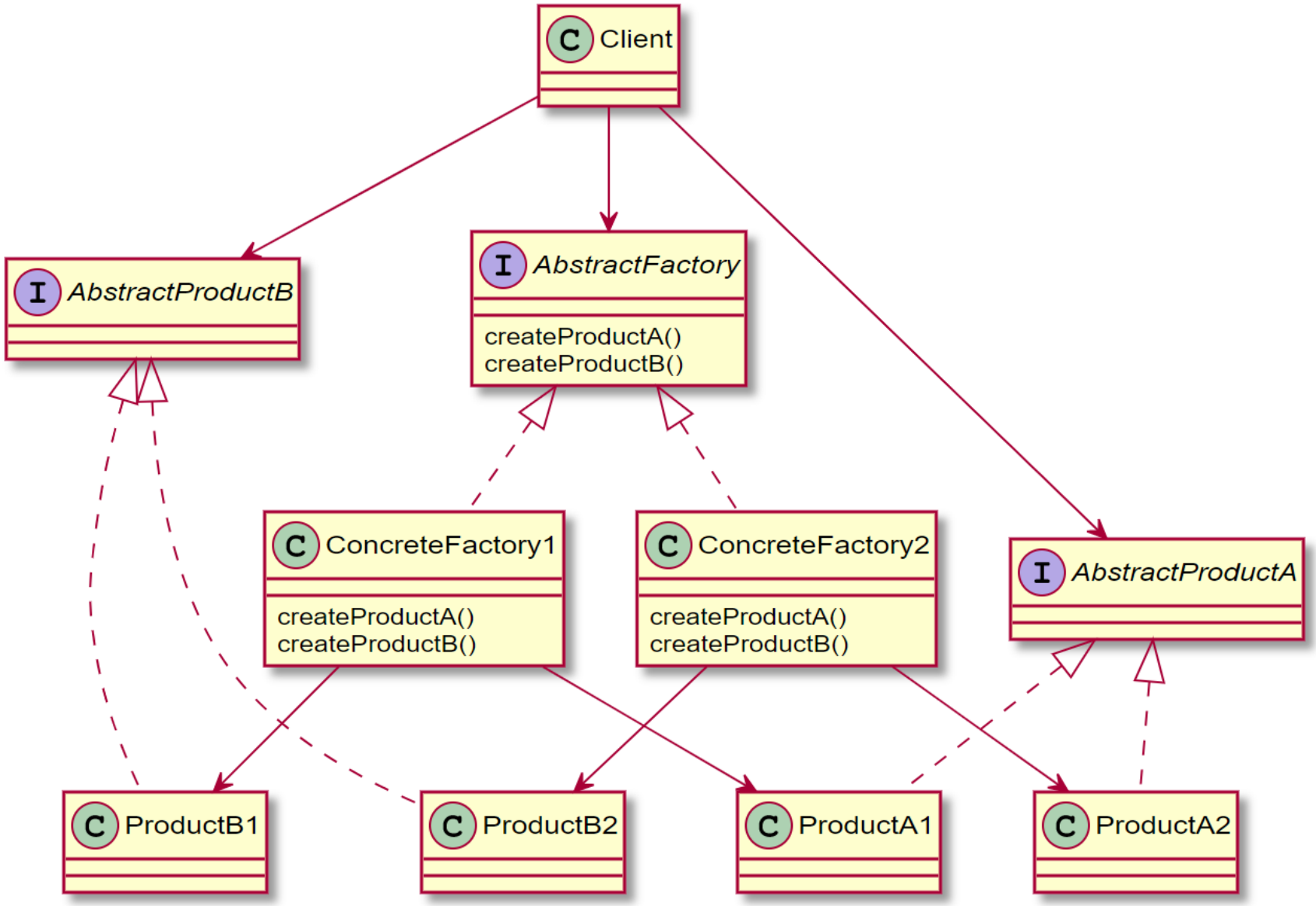
사례 3 – 피자 원재료 품질 관리

```
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")) {
    …
    }
    return pizza;
  }
}
```

추상 팩토리 패턴 (Abstract Factory Pattern)

- 추상 팩토리를 통해서 제품군을 생성하기 위한 인터페이스를 제공할 수 있음
  - 인터페이스를 이용하는 코드를 만들면 코드를 제품을 생산하는 실제 팩토리와 분리시킬 수 있음
  - 이렇게 함으로써 지역, 운영체제, 룩앤필 (Look & Feel) 등 서로 다른 상황별로 적당한 제품을 생산할 수 있는 다양한 팩토리 구현 가능
  - 코드가 실제 제품과 분리되어 있으므로 다른 공장을 사용하면 다른 결과를 얻을 수 있음
    - (플럼 토마토 소스 대신 마리나라 소스를 쓰는 식)

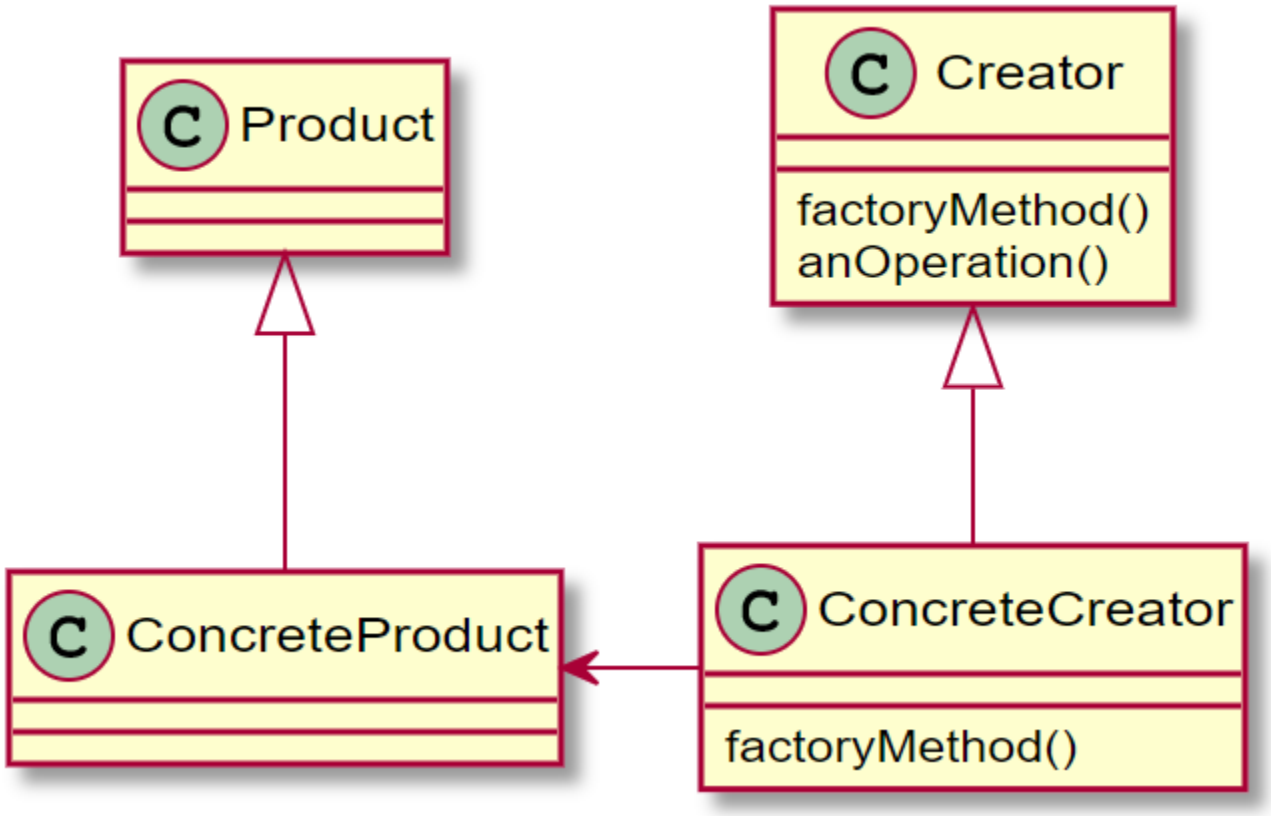
추상 팩토리 패턴


Client
AbstractProductB
AbstractFactory
createProductA()
createProductB()
ConcreteFactory1
createProductA()
createProductB()
ConcreteFactory2
createProductA()
createProductB()
AbstractProductA
ProductB1
ProductB2
ProductA1
ProductA2

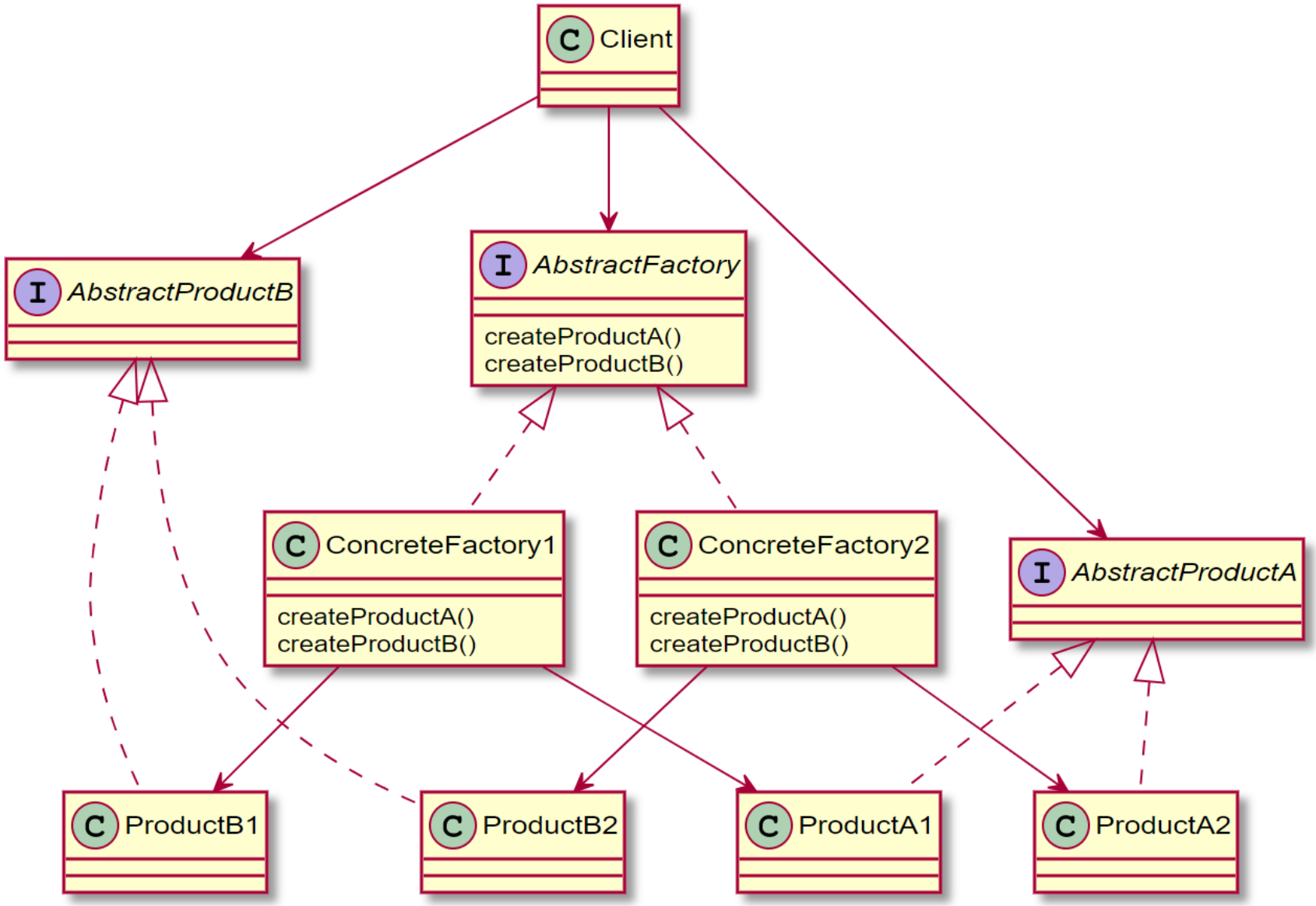
디자인 패턴 요소

요소	설명
이름	팩토리 메소드 (Factory Method), 추상 팩토리 (Abstract Factory)
문제	실제로 구현되는 클래스의 객체를 생성할 때 객체의 종류가 달라지면 클라이언트 코드를 수정해야 하는 것이 너무 많음
해결방안	생성을 분리해서 캡슐화 시킴
결과	사용할 객체가 많거나 객체를 생성하는 방법이 변경되어도 연쇄적인 수정이 적어짐

팩토리 메소드 패턴


Product
Creator
factoryMethod()
anOperation()
ConcreteProduct
ConcreteCreator
factoryMethod()

추상 팩토리 패턴


Client
AbstractProductB
AbstractFactory
createProductA()
createProductB()
ConcreteFactory1
createProductA()
createProductB()
ConcreteFactory2
createProductA()
createProductB()
AbstractProductA
ProductB1
ProductB2
ProductA1
ProductA2