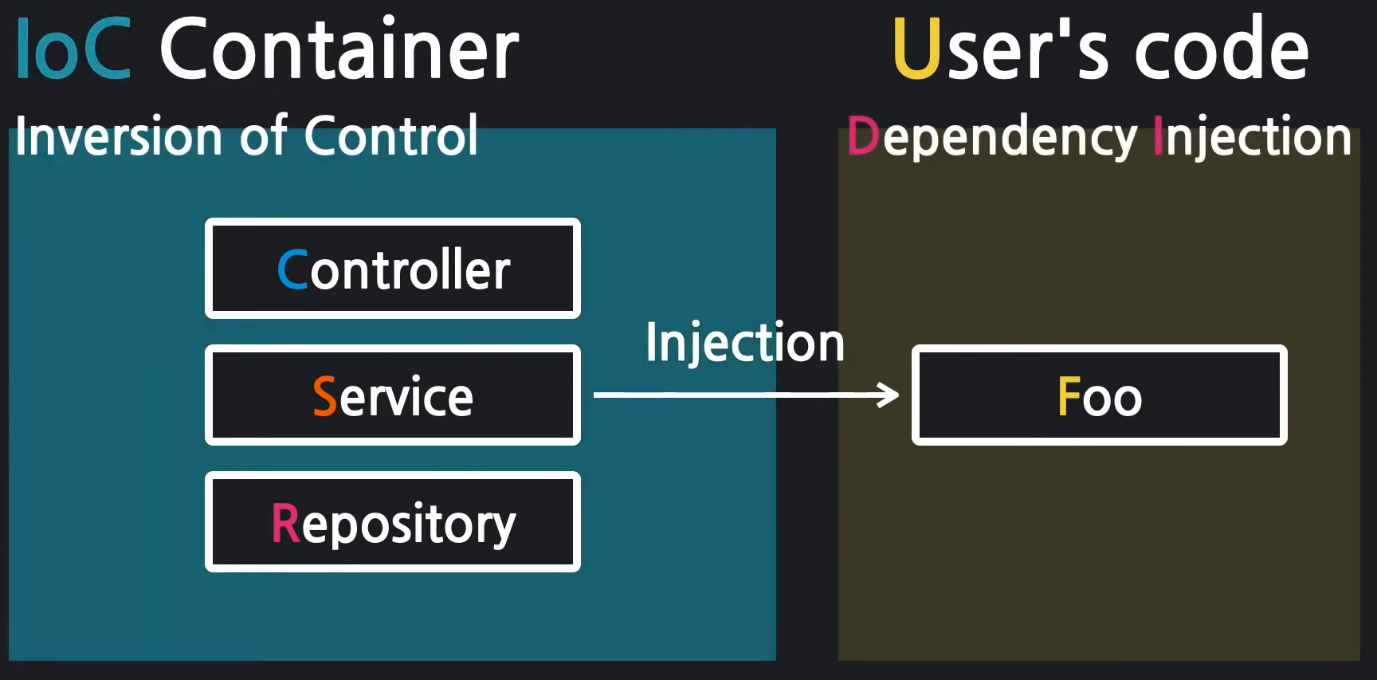
IoC(제어의역전)와 DI(의존성주입)



IoC와 DI는 모두 객체 지향 프로그래밍에서 객체 간의 의존성을 관리하는 방법을 설명하는 용어입니다.

* IoC (Inversion of Control):  
  객체의 생성과 생명주기를 외부 컨테이너에 위임하는 개념으로, 객체를 생성하고 관리하는 주체가 바뀌는 것을 의미합니다. 이는 애플리케이션의 컴포넌트들이 서로 협업하는 방식을 바꾸어, 애플리케이션의 확장성과 유지보수성을 높이는 데 도움을 줍니다. IoC는 객체의 제어를 역전시키는 것이기 때문에 "제어의 역전(Inversion of Control)"이라는 용어로도 불립니다.
* DI (Dependency Injection):  
  객체 간의 의존성을 외부에서 주입해주는 방식으로, 객체가 생성될 때 외부에서 필요한 의존 객체를 전달해주는 것을 의미합니다. DI는 코드의 결합도를 낮추고 코드 재사용성과 유지보수성을 향상시키는 데 도움을 줍니다.

간단한 예시로, 서비스 클래스 A가 데이터베이스 클래스 B에 의존한다고 가정해보겠습니다. 일반적으로 A 클래스는 B 클래스를 직접 생성하고 사용합니다. 하지만 이를 DI로 변경하면, 외부에서 B 클래스를 생성하고 A 클래스의 생성자나 메서드에 인자로 전달하여 A 클래스와 B 클래스 간의 의존성을 완화시킬 수 있습니다.

DI를 사용하는 프레임워크로는 Spring Framework가 있으며, Spring은 IoC 컨테이너와 DI를 구현한 다양한 방법을 제공합니다.

**실습**

**돈가스 요리하기**

ioc 패키지 생성.

**Chef 클래스 생성.**

package com.example.firstproject.ioc;

public class Chef {

public String cook(String menu) {

// 요리 재료 준비

Pork pork = new Pork("한돈 등심");

// 요리 반환

return pork.getName() + "으로 만든 " + menu;

}

}

**Pork 클래스 생성**

package com.example.firstproject.ioc;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

@Getter

@NoArgsConstructor

@AllArgsConstructor

public class Pork {

private String name;

}

**ChefTest 클래스 생성**

package com.example.firstproject.ioc;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

class ChefTest {

@Test

void 돈가스\_요리하기() {

// 준비

Chef chef = new Chef();

String menu = "돈가스";

// 수행

String food = chef.cook(menu);

// 예상

String expected = "한돈 등심으로 만든 돈가스";

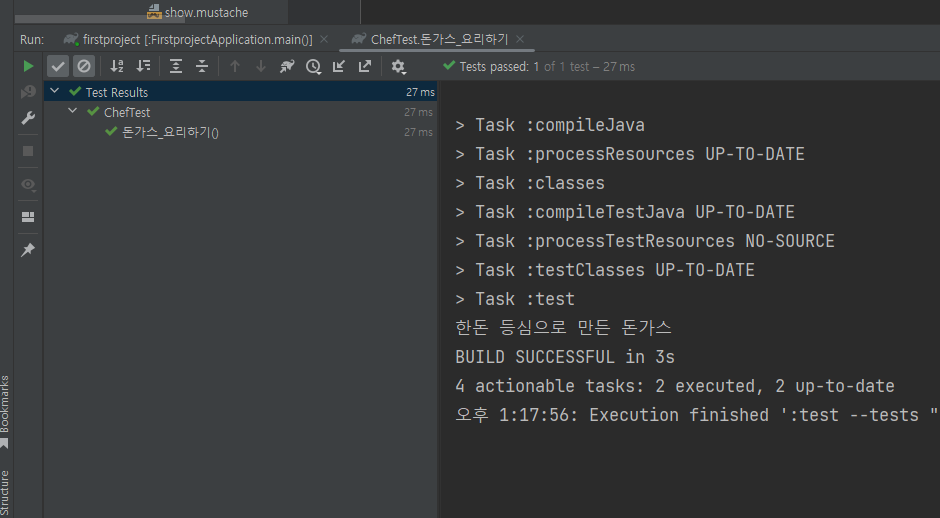
// 검증

assertEquals(expected, food);

System.out.println(food);

}

}

👌 test 성공 ✨  


**스테이크 요리하기**

ChefTest 클래스에 스테이크\_요리하기()메소드 추가

@Test

void 스테이크\_요리하기() {

// 준비

Chef chef = new Chef();

String menu = "스테이크";

// 수행

String food = chef.cook(menu);

// 예상

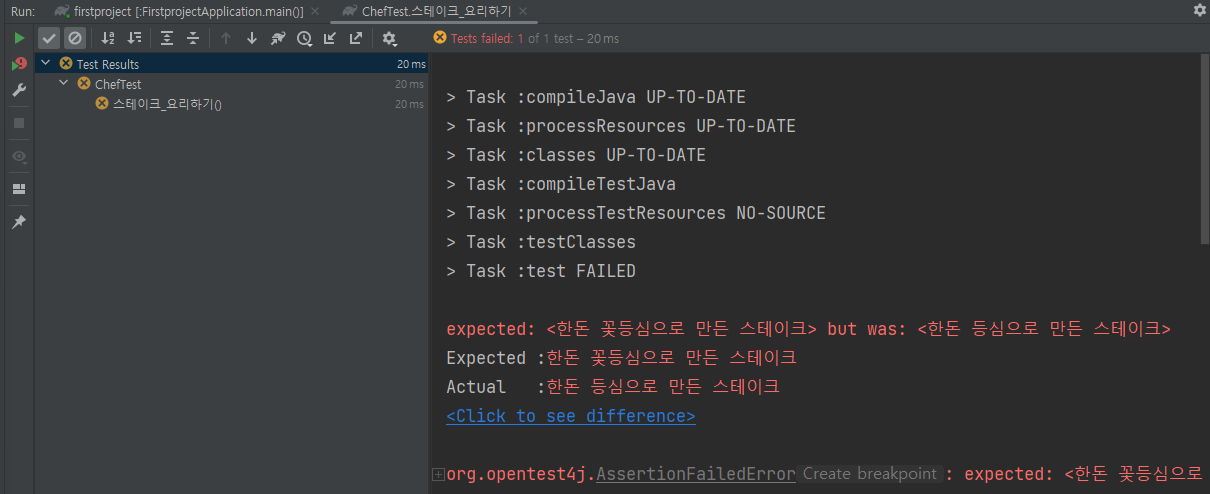
String expected = "한우 꽃등심으로 만든 스테이크";

// 검증

assertEquals(expected, food);

System.out.println(food);

}

Test 해보면 에러가 난다.  


Chef 클래스에서 한돈 등심을 준비하고 있기때문이다.

그러면 Beef클래스를 만들고

package com.example.firstproject.ioc;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

@Getter

@AllArgsConstructor

@NoArgsConstructor

public class Beef {

private String name;

}

Chef 클래스에 Beef를 추가한다.

package com.example.firstproject.ioc;

public class Chef {

public String cook(String menu) {

// 요리 재료 준비

Pork pork = new Pork("한돈 등심");

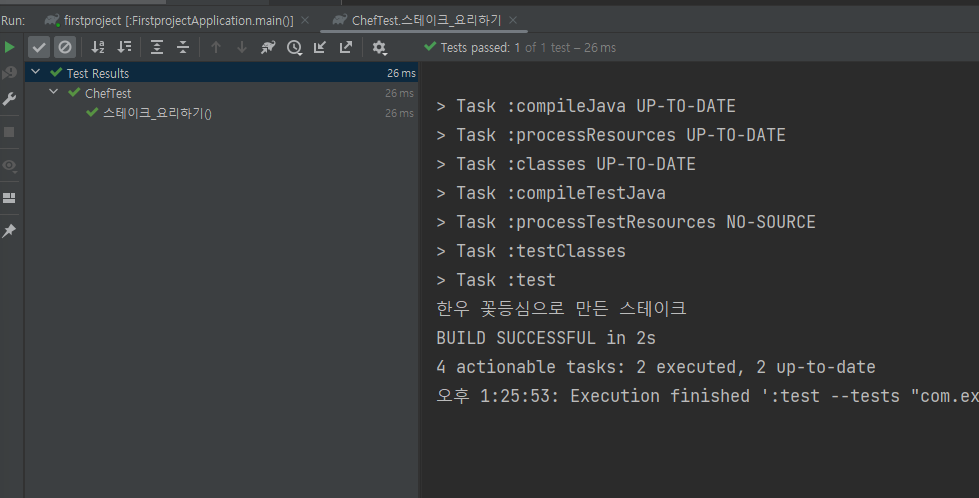
Beef beef = new Beef("한우 꽃등심");

// 요리 반환

return beef.getName() + "으로 만든 " + menu;

}

}

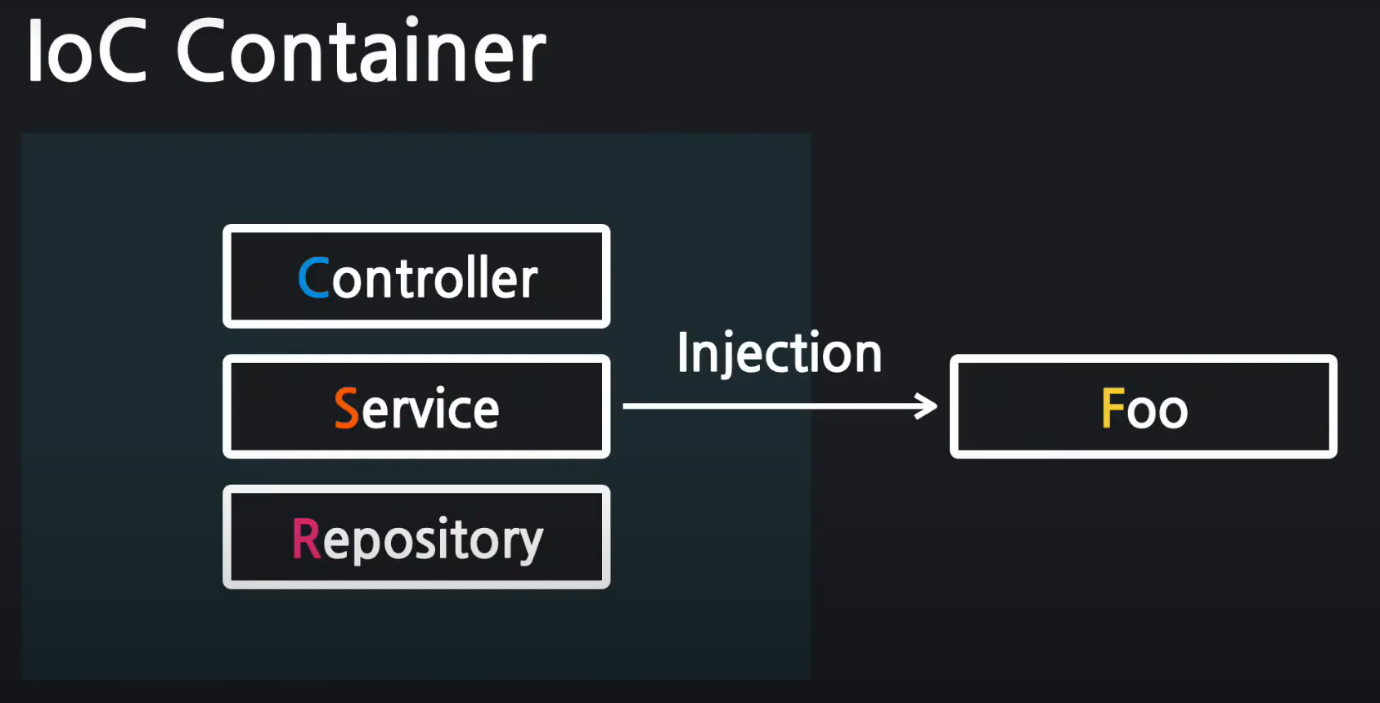
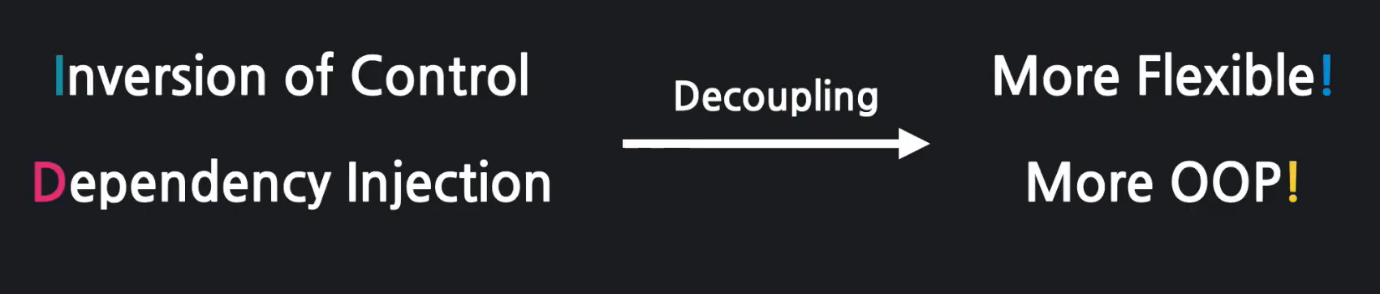
Test 성공  


**의존성이 높은 코드의 문제**

위의 코드는 문제가 있다.

다시 돈가스를 요리하려면  
Chef 클래스에 cook 메소드를 다시 바꿔야한다....😱

이렇게 다른 요청이 들어올때마다 코드를 수정해야한다..!!

그래서 DI를 사용해서 코드를 개선해야한다.  
  


**Decoupling**

Decoupling은 객체 간의 결합도를 낮추는 것을 의미합니다. 객체 간의 결합도가 높을 경우, 한 객체의 변경이 다른 객체에 영향을 미치는 경우가 많아져 유지보수가 어려워집니다. 또한, 객체 간의 결합도가 높을 경우 재사용성도 떨어집니다.

Decoupling을 위해서는 객체 간의 인터페이스를 추상화하여 구현부와 인터페이스를 분리해야 합니다. 이를 통해 객체 간의 결합도를 낮출 수 있으며, 한 객체의 변경이 다른 객체에 영향을 미치지 않도록 할 수 있습니다.

Decoupling은 소프트웨어 아키텍처에서 매우 중요한 원칙 중 하나이며, 유지보수성, 재사용성, 확장성 등을 향상시키는 데 큰 역할을 합니다.

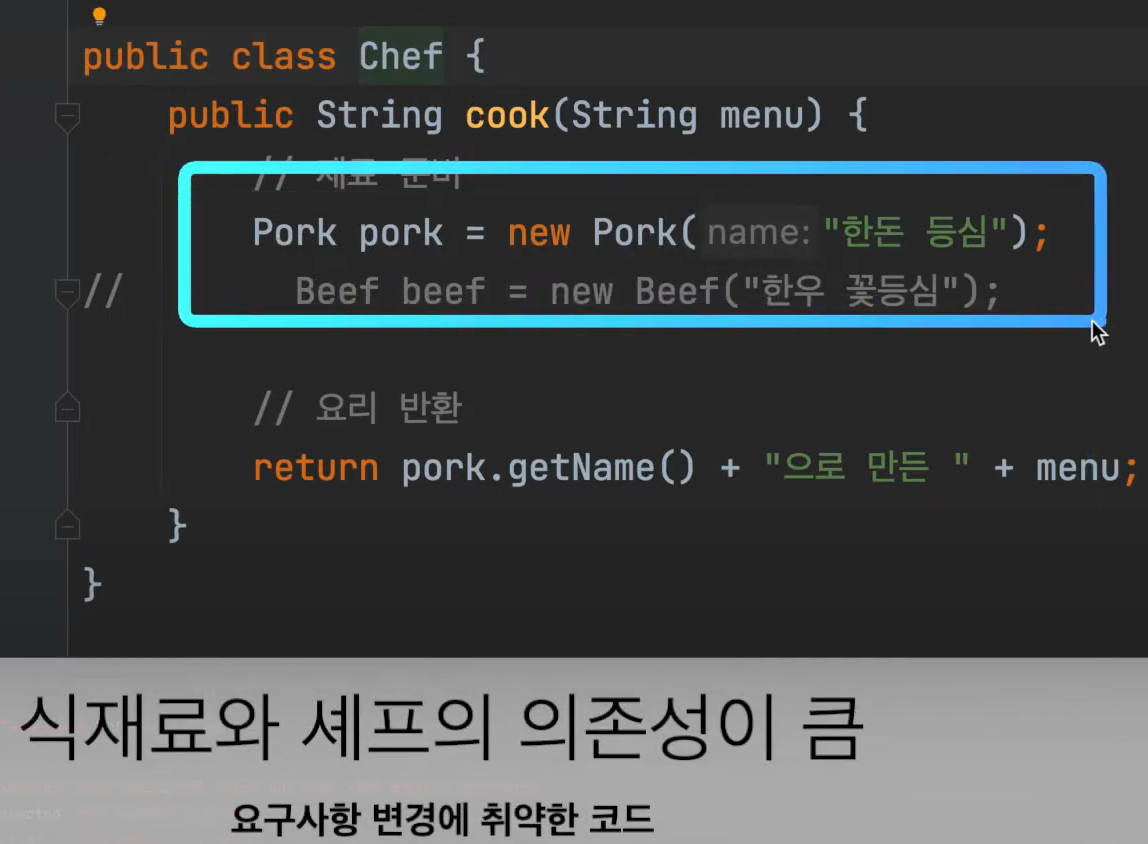
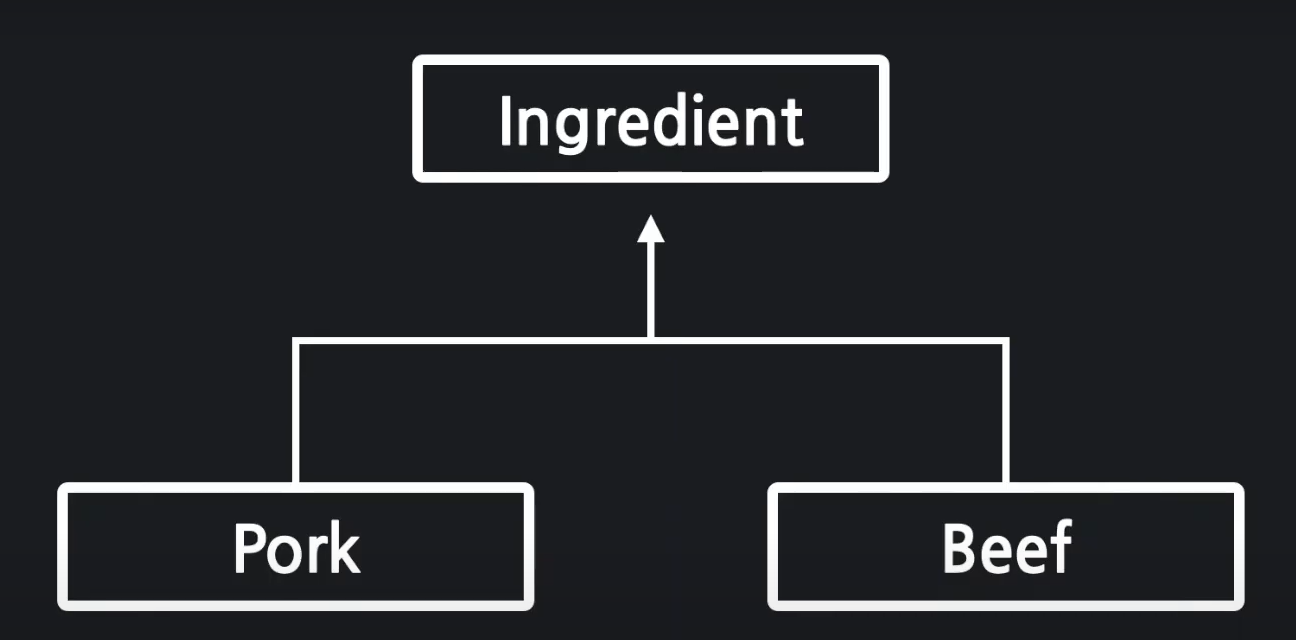
**Flexible**

소프트웨어에서 "Flexible"한 코드는 변경에 대한 대처가 뛰어나며, 다양한 요구사항에 대처할 수 있는 능력이 있습니다. 이러한 유연성은 소프트웨어의 재사용성, 유지보수성, 확장성 등을 향상시키는 데 큰 역할을 합니다.

소프트웨어에서 유연성을 높이기 위해서는 코드의 의존성을 최소화하고, 코드를 모듈화하고, 추상화를 적절히 사용해야 합니다. 또한, 코드를 유닛 테스트 가능하도록 작성하여 변경에 대한 영향을 최소화하고, Design Pattern 등의 개념을 적극적으로 활용하여 유연성을 높일 수 있습니다.

유연한 코드는 시간이 지나도 변화에 대처할 수 있으며, 다양한 요구사항에 대응할 수 있는 능력이 있습니다. 이를 통해 개발자는 유지보수성이 높은 소프트웨어를 개발할 수 있으며, 비즈니스 요구사항의 변화에 대처할 수 있습니다.

**의존성 주입하기(DI)**

  
현재는 Chef가 직접 재료를 준비하기 때문에  
의존성이 크다.  
셰프와 식재료 사이에 조달 공장을 두어 의존성을 낮춰보자.  


Chef가 요리를 할때 재료를 바로 준비하는 것이아니라  
식재료 공장에서 받아와서 사용한다.(DI)

// 재료공장

IngredientFactory ingredientFactory = new IngredientFactory();

Chef chef = new Chef(ingredientFactory);

String menu = "돈가스";

**Chef**

package com.example.firstproject.ioc;

public class Chef {

// 셰프는 식재료 공장을 알고있어야하기 때문에 변수를 가지고있어야함.

private IngredientFactory ingredientFactory;

// 셰프가 식재료 공장과 협업하기 위한 DI

// 외부에서 공장(객체)에 대한 정보를 받아옴.

public Chef(IngredientFactory ingredientFactory) {

this.ingredientFactory = ingredientFactory;

}

public String cook(String menu) {

// 요리 재료 준비

// Pork pork = new Pork("한돈 등심");

// Beef beef = new Beef("한우 꽃등심");

Ingredient ingredient = ingredientFactory.get(menu);

// 요리 반환

return ingredient.getName() + "으로 만든 " + menu;

}

}

**Ingredient**

추상 클래스

package com.example.firstproject.ioc;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.NoArgsConstructor;

public abstract class Ingredient {

private String name;

public Ingredient(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**IngredientFactory**

package com.example.firstproject.ioc;

public class IngredientFactory {

public Ingredient get(String menu) {

switch (menu) {

case "돈가스":

return new Pork("한돈 등심");

case "스테이크":

return new Beef("한우 꽃등심");

default:

return null;

}

}

}

이제 Beef와 Pork는 Ingredient를 상속받아서 사용한다.

public class Pork extends Ingredient{

public Pork(String name) {

super(name);

}

}

public class Beef extends Ingredient{

public Beef(String name) {

super(name);

}

}

**요구사항 변경에 유연한 확장**

**크리스피 치킨 요리하기 테스트**

@Test

void 크리스피\_치킨\_요리하기() {

// 준비

// 재료공장

IngredientFactory ingredientFactory = new IngredientFactory();

Chef chef = new Chef(ingredientFactory);

String menu = "크리스피 치킨";

// 수행

String food = chef.cook(menu);

// 예상

String expected = "국내산 10호 닭으로 만든 크리스피 치킨";

// 검증

assertEquals(expected, food);

System.out.println(food);

}

IngredientFactory에 case 추가

switch (menu) {

case "돈가스":

return new Pork("한돈 등심");

case "스테이크":

return new Beef("한우 꽃등심");

case "크리스피 치킨":

return new Chicken("국내산 10호 닭");

default:

return null;

}

Chicken 클래스 생성

public class Chicken extends Ingredient {

public Chicken(String name) {

super(name);

}

}

간단한 코드추가, 클래스 생성으로 유연하게 메뉴변경에 대처할 수 있다.

**Ioc컨테이너와 DI**

이제 IngredientFactory, Chef를 Ioc 컨테이너에 등록해서 직접 생성하지 않고 가져와서 사용하는 방식으로 바꿔보자.

ChefTest에  
@SpringBootTest 와  
@Autowired  
IngredientFactory ingredientFactory; 추가.

@SpringBootTest

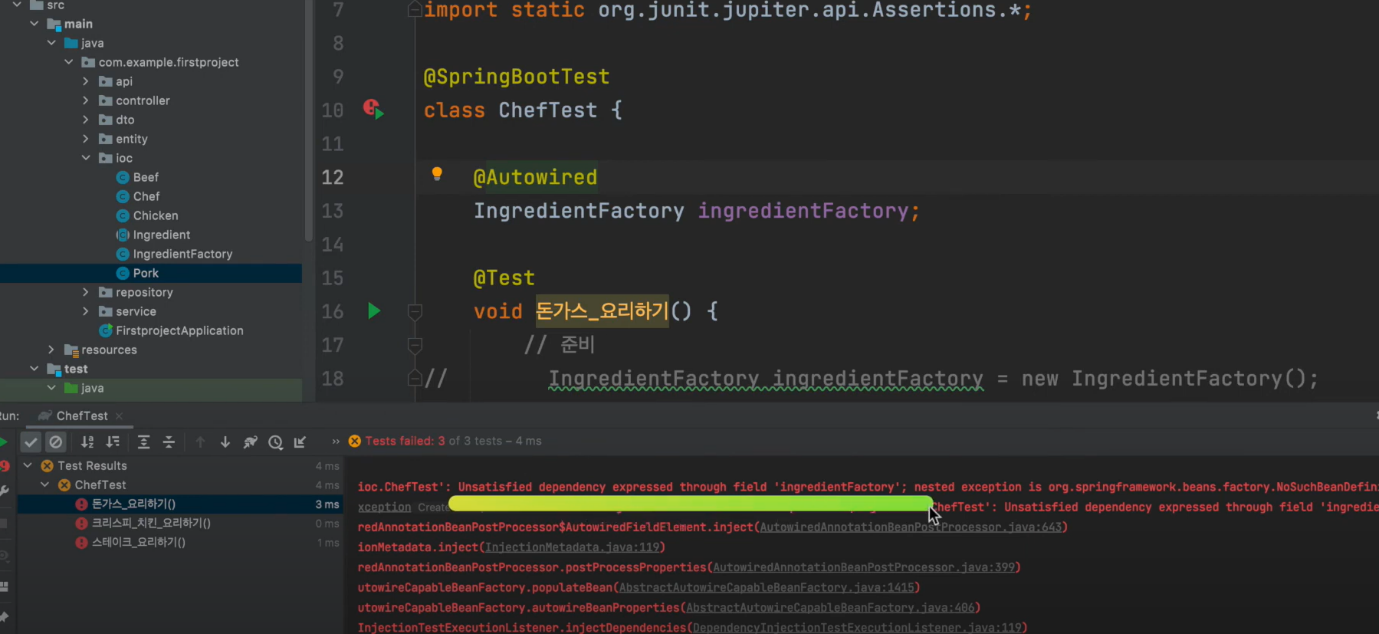
class ChefTest {

@Autowired

IngredientFactory ingredientFactory;

...

}

Test 진행해보면 IngredientFactory를 찾을 수 없다는 에러가 발생한다.  


**@Component**

해당 클래스를 객체로 만들고, 이를 Ioc 컨테이너에 등록하는 어노테이션

IngredientFactory,Chef에 @Component 어노테이션을 추가하면 Ioc에 등록이 된다!!

이렇게 하면 IngredientFactory,Chef를 DI하고  
매번 객체를 생성하지 않아도 바로바로 가져와서 사용할 수 있다.

**전체코드**

**IngredientFactory**

@Component // 해당 클래스를 객체로 만들고, 이를 Ioc 컨테이너에 등록하는 어노테이션

public class IngredientFactory {

public Ingredient get(String menu) {

switch (menu) {

case "돈가스":

return new Pork("한돈 등심");

case "스테이크":

return new Beef("한우 꽃등심");

case "크리스피 치킨":

return new Chicken("국내산 10호 닭");

default:

return null;

}

}

}

**Chef**

@Component

public class Chef {

// 셰프는 식재료 공장을 알고있어야하기 때문에 변수를 가지고있어야함.

private IngredientFactory ingredientFactory;

// 셰프가 식재료 공장과 협업하기 위한 DI

// 외부에서 공장(객체)에 대한 정보를 받아옴.

public Chef(IngredientFactory ingredientFactory) {

this.ingredientFactory = ingredientFactory;

}

public String cook(String menu) {

// 요리 재료 준비

// Pork pork = new Pork("한돈 등심");

// Beef beef = new Beef("한우 꽃등심");

Ingredient ingredient = ingredientFactory.get(menu);

// 요리 반환

return ingredient.getName() + "으로 만든 " + menu;

}

}

**Ingredient**

public abstract class Ingredient {

private String name;

public Ingredient(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**ChefTest**

@SpringBootTest

class ChefTest {

@Autowired

IngredientFactory ingredientFactory;

@Autowired

Chef chef;

@Test

void 돈가스\_요리하기() {

// 준비

// 재료공장

// IngredientFactory ingredientFactory = new IngredientFactory();

// Chef chef = new Chef(ingredientFactory);

String menu = "돈가스";

// 수행

String food = chef.cook(menu);

// 예상

String expected = "한돈 등심으로 만든 돈가스";

// 검증

assertEquals(expected, food);

System.out.println(food);

}

@Test

void 스테이크\_요리하기() {

// 준비

// 재료공장

// IngredientFactory ingredientFactory = new IngredientFactory();

// Chef chef = new Chef(ingredientFactory);

String menu = "스테이크";

// 수행

String food = chef.cook(menu);

// 예상

String expected = "한우 꽃등심으로 만든 스테이크";

// 검증

assertEquals(expected, food);

System.out.println(food);

}

@Test

void 크리스피\_치킨\_요리하기() {

// 준비

// 재료공장

// IngredientFactory ingredientFactory = new IngredientFactory();

// Chef chef = new Chef(ingredientFactory);

String menu = "크리스피 치킨";

// 수행

String food = chef.cook(menu);

// 예상

String expected = "국내산 10호 닭으로 만든 크리스피 치킨";

// 검증

assertEquals(expected, food);

System.out.println(food);

}

}