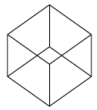
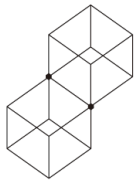
12. 팩토리 메서드 패턴



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



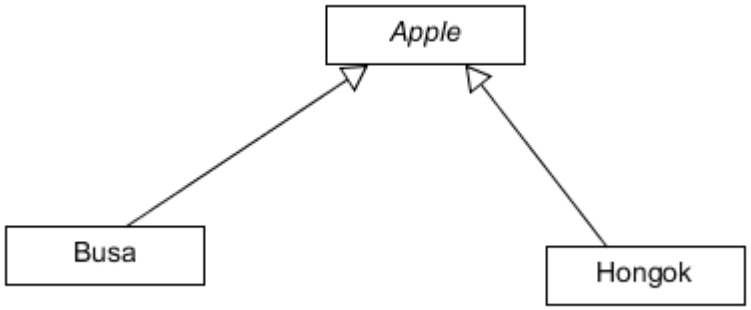
학습목표

학습목표

- 적합한 클래스의 객체를 생성하는 코드의 캡슐화 방법 이해하기
- 팩토리 메서드 패턴을 이용한 객체 생성 방법 이해하기
- 사례 연구를 통한 팩토리 메서드 패턴의 핵심 특징 이해하기

사과를 디저트로 제공하는 식당 클래스 만들기

```
public class Restaurant {
    public Apple servingApple() {
        Apple apple = new Busa();
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```



Apple

```java
public abstract class Apple {
    public abstract void wash();
    public abstract void peel();
    public abstract void slice();
}
```

```java
public class Busa extends Apple {
    @Override
    public void wash() {
        System.out.println("부사: 물로 씻기");
    }

    @Override
    public void peel() {
        System.out.println("부사: 껍질 벗기기");
    }

    @Override
    public void slice() {
        System.out.println("부사: 자르기");
    }
}
```

```java
public class Hongok extends Apple {
    @Override
    public void wash() {
        System.out.println("홍옥: 물로 씻기");
    }

    @Override
    public void peel() {
        System.out.println("홍옥: 껍질 벗기기");
    }

    @Override
    public void slice() {
        System.out.println("홍옥: 자르기");
    }
}
```

사과를 아침으로 먹는 Home 클래스 만들기

```
public class Home {
    public Apple getAppleForBreakFast() {
        Apple apple = new Hongok();
        apple.wash();
        return apple;
    }
}
```

문제점

- 사과의 종류가 변경되거나 추가된다.

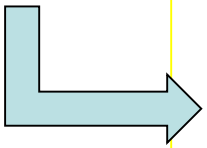
```
public class Restaurant {
    public Apple servingApple() {
        Apple apple = new Busa();
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```

```
public class Home {
    public Apple getAppleForBreakFast() {
        Apple apple = new Hongok();
        apple.wash();
        return apple;
    }
}
```

사과의 종류 변경

```
public class Restaurant {
    public Apple servingApple() {
        Apple apple = new Busa();
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```

사과 인스턴스를 생성하는 모든 코드에서
이러한 작업을 반복할 필요가 있다.

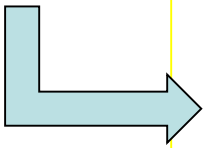


```
public class Restaurant {
    public Apple servingApple(String kind) {
        Apple apple = null;
        if (kind.equals("busa")) apple = new Busa();
        else if (kind.equals("hongok")) apple = new Hongok();
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```

사과의 종류 변경

```
public class Home {
    public Apple getAppleForBreakFast() {
        Apple apple = new Hongro();
        apple.wash();
        return apple;
    }
}
```

사과 인스턴스를 생성하는 모든 코드에서
이러한 작업을 반복할 필요가 있다.

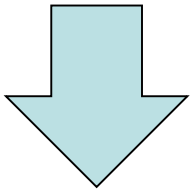


```
public class Home {
    public Apple Apple getAppleForBreakFast(String kind) {
        Apple apple = null;
        if (kind.equals("busa")) apple = new Busa();
        else if (kind.equals("hongok")) apple = new Hongok();
        apple.wash();
        return apple;
    }
}
```

제공하는 사과의 종류가 변경

```
public class Restaurant {
    public Apple servingApple(String kind) {
        Apple apple = null;
        if (kind.equals("busa")) apple = new Busa();
        else if (kind.equals("hongok")) apple = new Hongok();
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```

제공하는 사과의 종류가 변경된다.



변경이 자주 일어나는 부분은 분리하여 클래스로 캡슐화할 필요가 있다.

클래스로 캡슐화

Factory method

```
public class AppleFactory {
    public static Apple getApple(String kind) {
        Apple apple = null;
        if(kind.equals("busa"))apple =new Busa();
        else if(kind.equals("hongok"))apple =new Hongok();
        else if (kind.equals("hongro"))apple =new Hongro();
        return apple;
    }
}
```

```
public class Restaurant {
    public Apple servingApple(String kind) {
        Apple apple = null;
        apple = AppleFactory.getApple(kind);
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }
}
```

```
public class Home {
    public Apple getAppleForBreakFast(String kind) {
        Apple apple = null;
        apple = AppleFactory.getApple(kind);
        apple.wash();
        return apple;
    }
}
```

새로운 사과 추가

```
public class AppleFactory {
    public static Apple getApple(String kind) {
        Apple apple = null;
        if(kind.equals("busa"))apple =new Busa();
        else if(kind.equals("hongok"))apple =new Hongok();
        else if (kind.equals("hongro"))apple =new Hongro();
        return apple;
    }
}
```

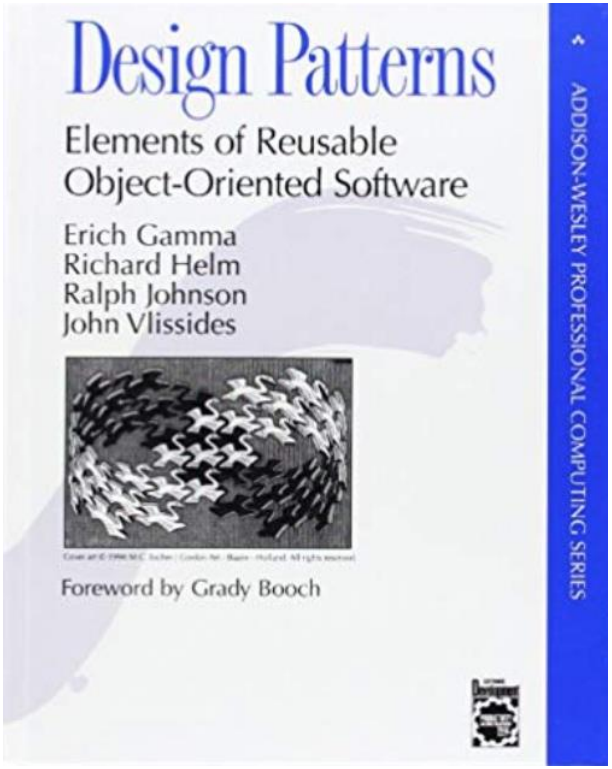
새로운 사과 종류가 추가되어도
Factory method만 변경할 필요가 있고 실제 사과를
사용하는 클래스는 영향 받지 않음.

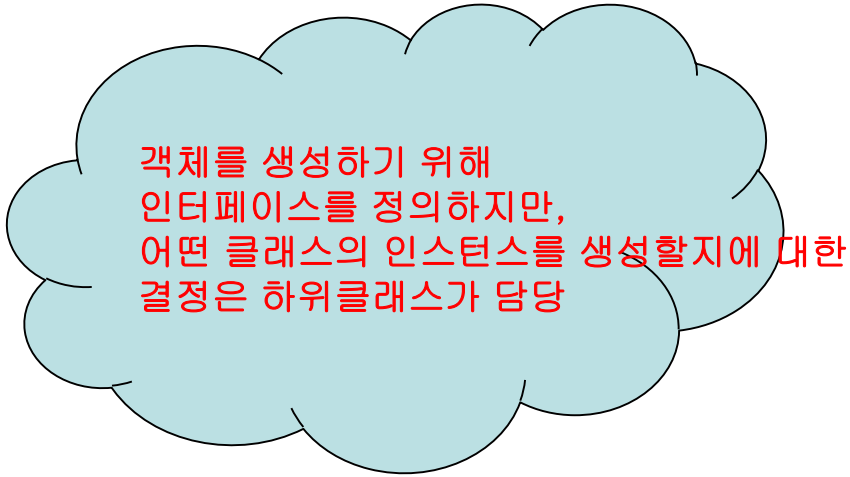
Singleton Pattern 적용

- AppleFactory 클래스에 싱글턴 패턴을 조굥하여 다시 작성해보시오

GoF의 Factory Method 패턴

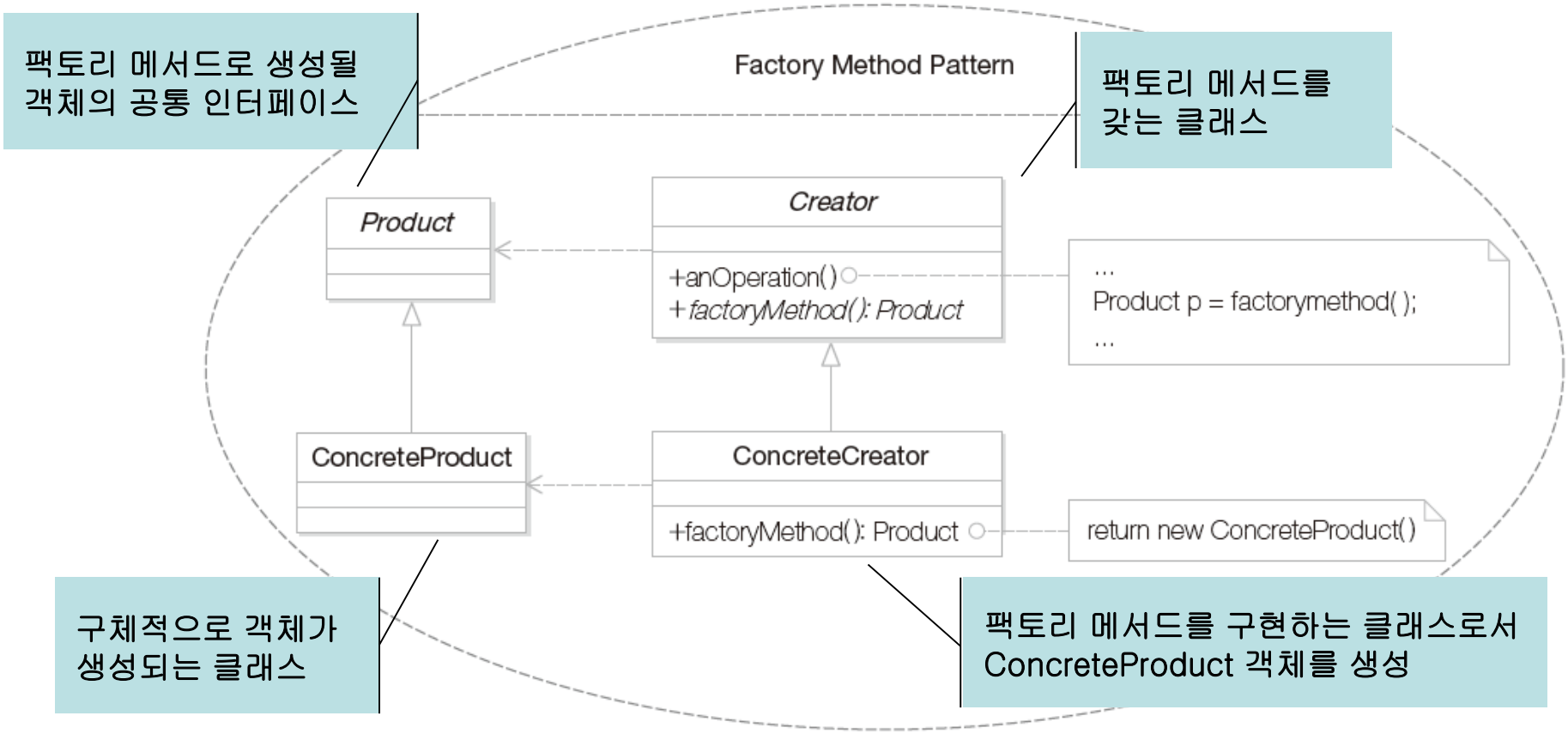
- 에리히 감마(Erich Gamma), 리처드 헬름(Richard Helm), 랄프 존슨(Ralph Johnson), 존 블리시데스(John Vlissides)





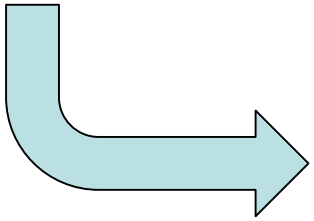
GoF 팩토리 메서드 패턴

그림 12-7 팩토리 메서드 패턴의 컬레보레이션



GoF Factory Method Pattern

- 식당에서 사과를 디저트로 고객에게 제공하는 것을 서울 지점과 뉴욕 지점으로 확대
- 사과를 서빙하기 위해 수행하는 단계는 어느 지점이나 공통이 되도록 한다.
  - getApple
  - Wash
  - Peel
  - Slice
- 반면에 사과의 종류는 지역(지점)마다 달리할 수 있다.



Template Method Pattern

GoF factory method

```
public abstract class Restaurant {
    public Apple servingApple(String kind) {
        Apple apple = getApple(kind);
        apple.wash();
        apple.peel();
        apple.slice();
        return apple;
    }

    public abstract Apple getApple(String kind);
}
```

factory method

Template method

Seoul Restaurant

```
public class SeoulRestaurant extends Restaurant {
    @Override
    public Apple getApple(String kind) {
        Apple apple = null;
        if(kind.equals("busa"))apple =new Busa();
        else if(kind.equals("hongok"))apple =new Hongok();
        else if (kind.equals("hongro"))apple =new Hongro();
        return apple;
    }
}
```

NewYork Restaurant

```
public class NewYorkRestaurant extends Restaurant {
    @Override
    public Apple getApple(String kind) {
        Apple apple = null;
        if(kind.equals(“koru"))apple =new Koru();
        else if(kind.equals(“crispy"))apple =new Evercrispy();
        else if (kind.equals(“pl"))apple =new Pinklady();
        return apple;
    }
}
```