

템플릿 메소드 패턴

이관우

kwlee@hansung.ac.kr

학습 목표

- 템플릿 메소드 패턴이 다루는 문제를 이해한다.
- 템플릿 메소드 패턴과 관련 디자인 원칙을 이해한다.
- 템플릿 메소드 패턴이 적용 사례

스타버즈 커피 바리스타 훈련용 메뉴얼

바리스타 여러분, 스타버즈 음료를 준비할 때는 아래에 있는 만드는 방법 그대로 해 주세요

스타버즈 커피 만드는 법

1. 물을 끓인다.
2. 끓는 물에 커피를 우려낸다
3. 커피를 컵에 따른다
4. 설탕과 우유를 추가한다.

스타버즈 홍차 만드는 법

1. 물을 끓인다.
2. 끓는 물에 차를 우려낸다
3. 차를 컵에 따른다
4. 레몬을 추가한다.



커피와 차를 만드는 방법이 매우 유사함

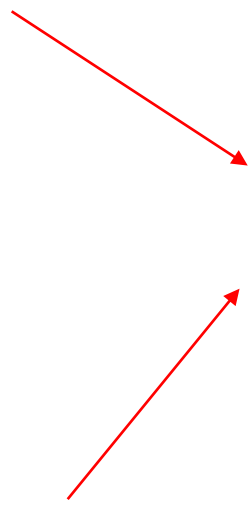
커피 및 홍차 클래스 만들기 (자바)

```
public class Coffee {  
    void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
    public void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
    public void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

커피 및 홍차 클래스 만들기 (자바)

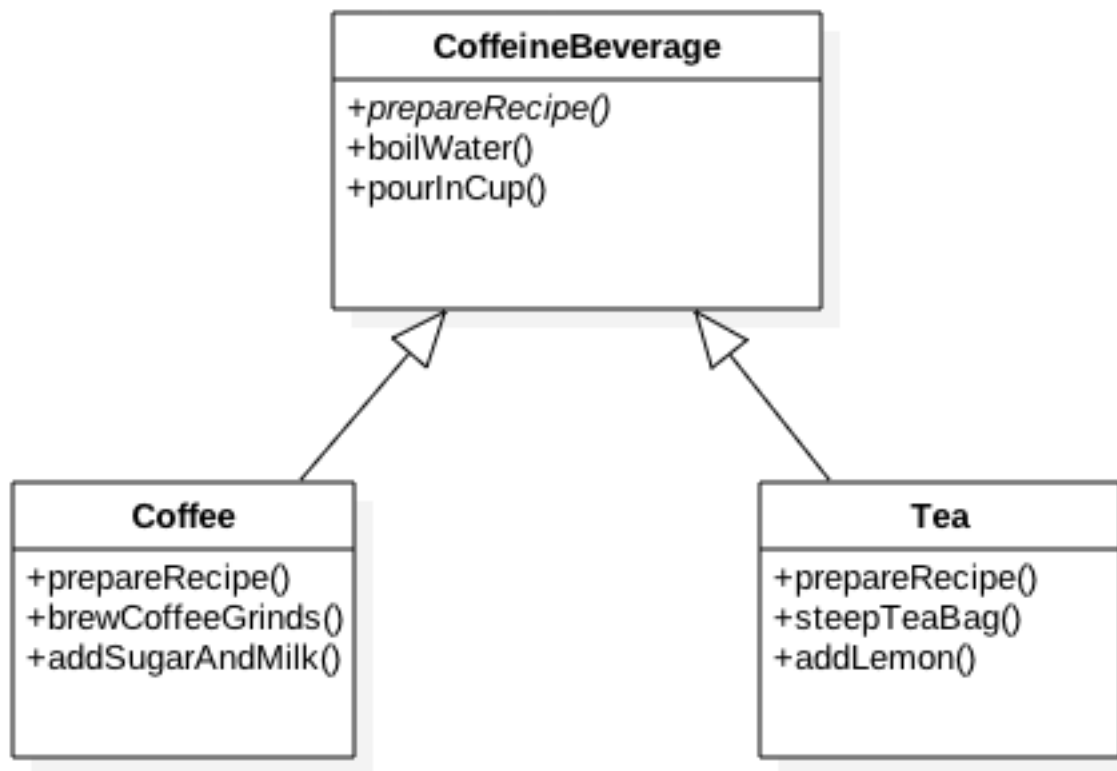
```
public class Tea {  
    void prepareRecipe() {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea");  
    }  
    public void addLemon() {  
        System.out.println("Adding Lemon");  
    }  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Coffee에 있는 메소드와 중복



코드 중복을 어떻게 없앨까요?

- One Approach



Software Design Patterns

```
public abstract class CaffeineBeverage {  
    abstract void prepareRecipe();  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

```
public class Coffee extends CaffeineBeverage{  
    void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
    void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
    void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

Coffee와 Tee사이에 또 다른 공통점은?

스타버즈 커피 만드는 법

1. 물을 끓인다.
2. 끓는 물에 커피를 우려낸다
3. 커피를 컵에 따른다
4. 설탕과 우유를 추가한다.

스타버즈 홍차 만드는 법

1. 물을 끓인다.
2. 끓는 물에 차를 우려낸다
3. 차를 컵에 따른다
4. 레몬을 추가한다.

Coffee와 Tee사이에 또 다른 공통점은?

스타버즈 커피 만드는 법

1. 물을 끓인다.
2. 끓는 물에 커피를 우려낸다
3. 커피를 컵에 따른다
4. 설탕과 우유를 추가한다.

스타버즈 홍차 만드는 법

1. 물을 끓인다.
2. 끓는 물에 차를 우려낸다
3. 차를 컵에 따른다
4. 레몬을 추가한다.



만드는 알고리즘이 동일

1. 물을 끓인다.
2. 끓는 물에 커피 또는 차를 우려낸다
3. 만들어진 음료를 컵에 따른다
4. 각 음료에 맞는 첨가물을 추가한다.

prepareRecipe() 추상화하기

Coffee

```
void prepareRecipe() {  
    boilWater();  
    brewCoffeeGrinds();  
    pourInCup();  
    addSugarAndMilk();  
}
```

Tee

```
void prepareRecipe() {  
    boilWater();  
    steepTeaBag();  
    pourInCup();  
    addLemon();  
}
```



```
void prepareRecipe() {  
    boilWater();  
    brew ();  
    pourInCup();  
    addCondiments();  
}
```

Software Design Patterns

```
public abstract class CaffeineBeverage {  
    final void prepareRecipe() {  
        boilWater();  
        brew ();  
        pourInCup();  
        addCondiments();  
    }  
    abstract void brew();  
    abstract void addCondiments();  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

```
public class Coffee extends CaffeineBeverage{  
    void brew () {  
        System.out.println("Dripping Coffee through filter");  
    }  
    void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

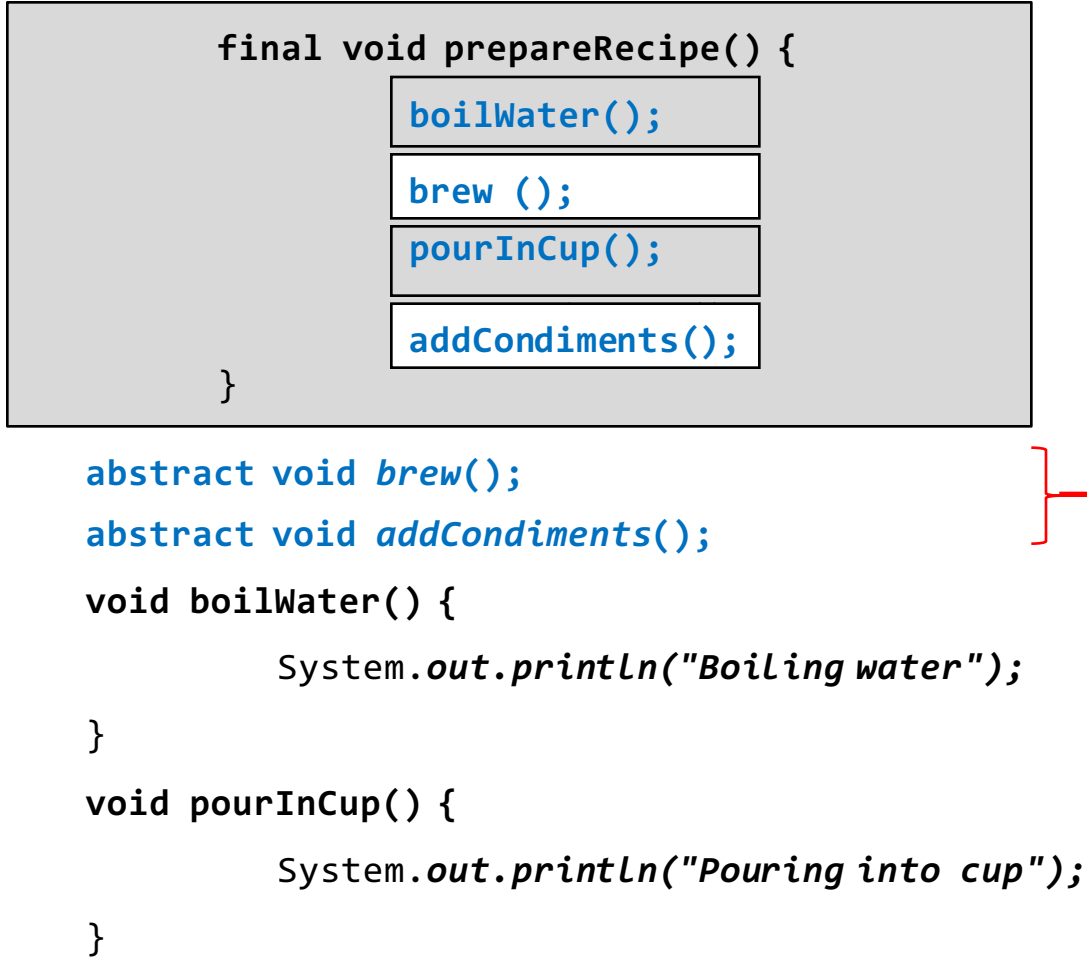
정리...

- Coffee와 Tee의 공통 메소드 추출
 - `boilWater()`, `pourInCup()` → `CaffeinBeverage`
- Coffee와 Tee의 만드는 방법을 일반화
 - `prepareRecipe()` → `CaffeinBeverage`
 - `brew()`, `addCondiment()` 추상 메소드 정의
- Coffee 고유의 메소드는 Coffee 클래스 (`CaffeinBeverage` 상속)에서 구현
 - `brew()` → Coffee 클래스에서 재정의
 - `addCondiments()` → Coffee 클래스에서 재정의
- Tea 고유의 메소드는 Tea 서브 클래스에서 구현 Tea 클래스 (`CaffeinBeverage` 상속)에서 구현
 - `brew()` → Tea 클래스에서 재정의
 - `addCondiments()` → Tea 클래스에서 재정의

템플릿 메소드 패턴

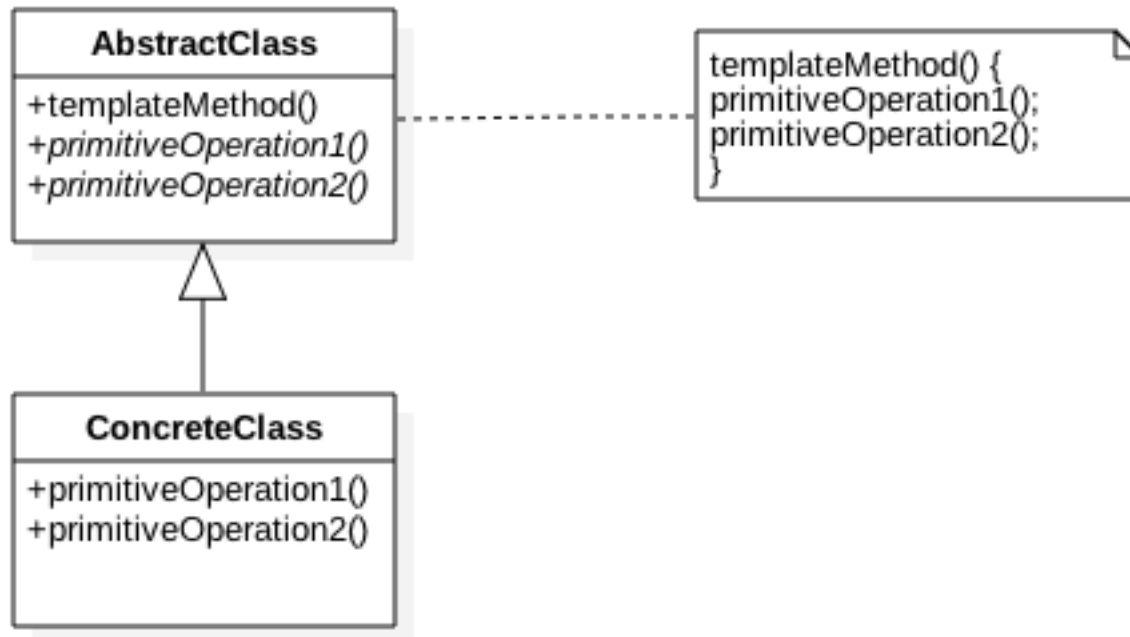
- 템플릿 메소드:
- 알고리즘에 대한 템플릿

```
public abstract class CaffeineBeverage {  
    final void prepareRecipe() {  
        boilWater();  
        brew ();  
        pourInCup();  
        addCondiments();  
    }  
  
    abstract void brew();  
    abstract void addCondiments();  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```



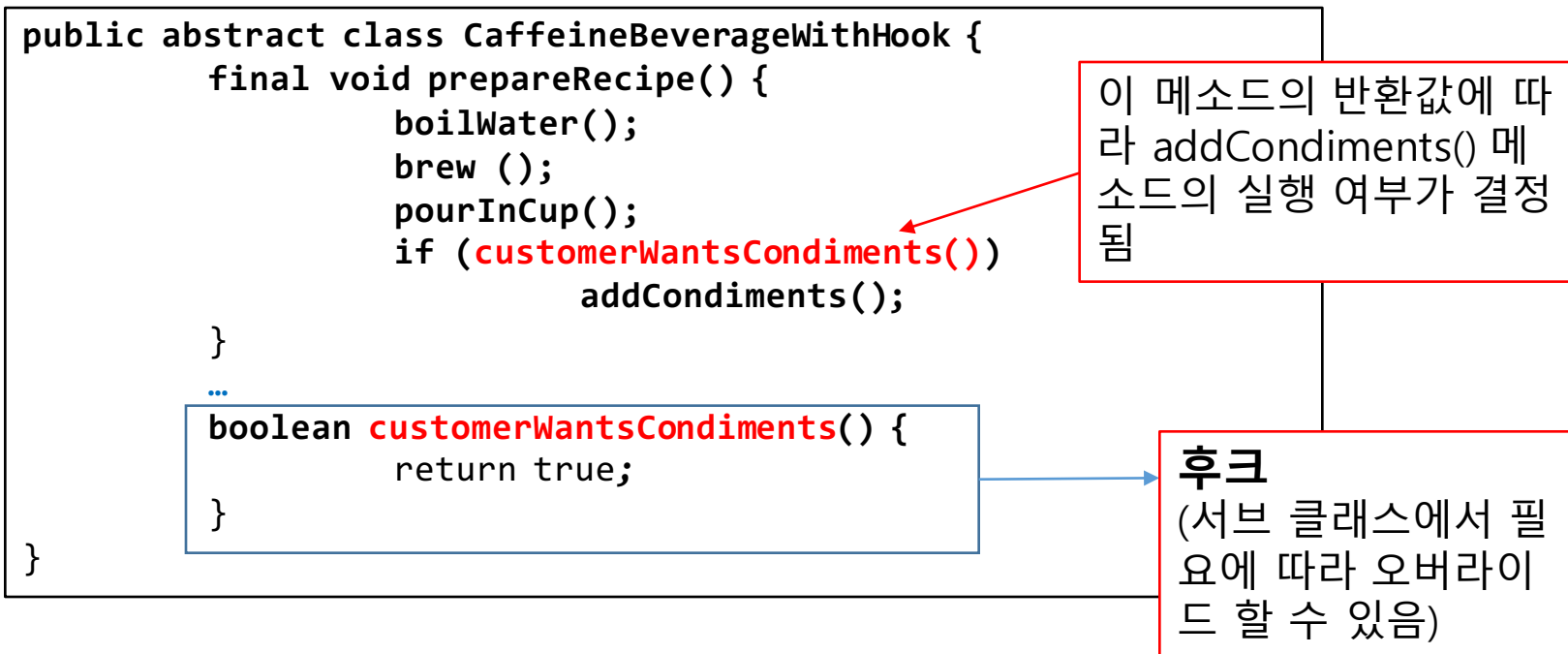
템플릿 메소드 패턴의 정의

- 템플릿 메소드 패턴에서는 메소드에서 알고리즘의 골격을 정의합니다. 알고리즘의 여러 단계 중 일부는 서브클래스에서 구현할 수 있습니다. 템플릿 메소드를 이용하면 알고리즘의 구조는 그대로 유지하면서 서브 클래스에서 특정 단계를 재정의할 수 있습니다.



템플릿 메소드와 후크

- 후크 (hook)
 - 추상 클래스내에서 선언된 메소드지만, 기본적인 내용만 구현되어 있거나 아무 코드도 들어 있지 않은 메소드
 - 서브 클래스 입장에서는 다양한 위치에서 알고리즘에 끼어들 수 있다.



후크 활용

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {  
  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
  
    // 후크를 재정의하여 원하는 기능을 구현  
    public boolean customerWantsCondiments() {  
  
        String answer = getUserInput();  
  
        if (answer.toLowerCase().startsWith("y")) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

사용자의 입력을 받음

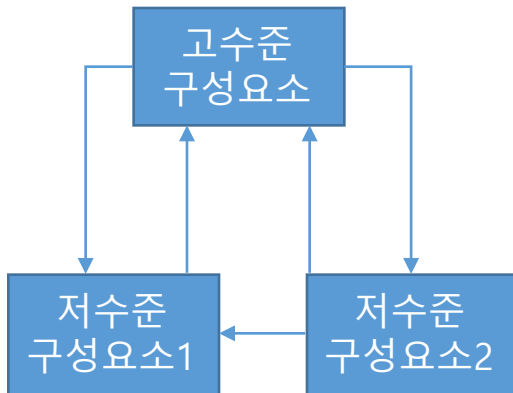
후크 활용

```
private String getUserInput() {  
    String answer = null;  
  
    System.out.print("Would you like milk and sugar  
                     with your coffee (y/n)? ");  
  
    BufferedReader in = new BufferedReader(  
                        new InputStreamReader(System.in));  
  
    try {  
        answer = in.readLine();  
    } catch (IOException ioe) {  
        System.err.println("IO error trying to read your answer");  
    }  
    if (answer == null) {  
        return "no";  
    }  
    return answer;  
}
```

헐리우드 원칙 (제어의 반전, Inversion of Control)

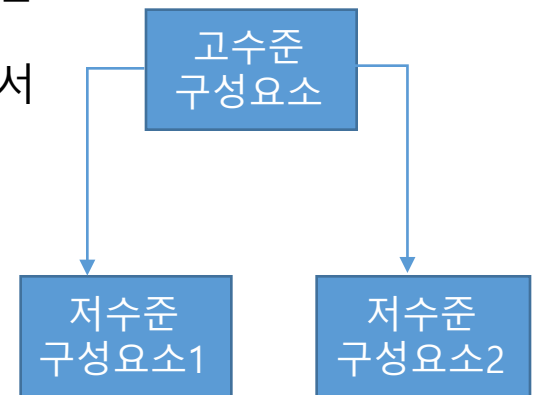
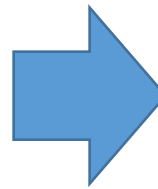
먼저 연락하지 마세요.
저희가 연락 드리겠습니다.

고수준 구성요소 : 프로그램의 전반적인 흐름을 구성하는 모듈
저수준 구성요소 : 프로그램의 일부분을 구성하는 모듈



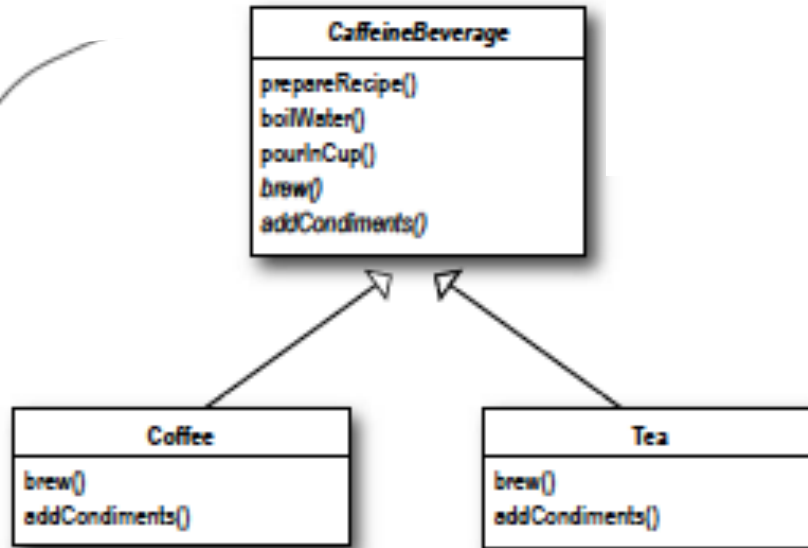
Dependency rot
(의존성 부패)

저수준 구성요소는 자기자신을 전체 시스템안으로 연결되게 만들지만, 언제 어떻게 사용되는지는 고수준 구성요소에 의해서 결정되게 함



헐리우드 원칙과 템플릿 메소드 패턴

CaffeineBeverage는 고수준 구성요소입니다. 음료를 만드는 방법에 해당하는 알고리즘을 장악하고 있고, 메소드 구현이 필요한 상황에서만 서브클래스를 불러내죠.

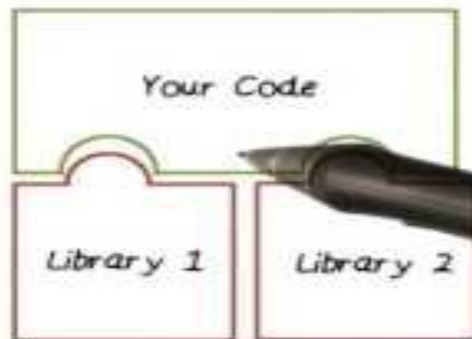


서브클래스는 알고리즘을 구성하는 일부분의 메소드 구현을 제공하기 위한 용도로만 쓰입니다.

Tea와 Coffee 클래스에서는 호출 "당하기" 전까지는 절대로 추상 클래스를 직접 호출하지 않습니다.

Framework and Inversion of Control

Frameworks and Inversion of Control



토의

- “의존성 역전 (Dependency Inversion)” 원칙과 “헐리우드 (Inversion of Control)” 원칙과의 관계는?
- 의존성 역전 (Dependency Inversion) 원칙
 - 추상화된 것에 의존하도록 만들어라.
 - 구상 클래스에 의존하도록 만들지 마라.
 - “고수준” 구성요소가 “저수준” 구성요소에 의존하면 안된다 (의존성 방향)
- 헐리우드 (Inversion of Control) 원칙
 - “먼저 연락하지 마세요. 저희가 연락 드리겠습니다.”
 - 저수준 구성요소가 언제 사용되는 지는 고수준 구성요소에 의해서 결정됨 (제어의 방향)

템플릿 메소드 적용사례- Swing Framework

```
public class MyFrame extends JFrame {  
    private static final long serialVersionUID = 2L;  
  
    public MyFrame(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        this.setSize(300,300);  
        this.setVisible(true);  
    }  
  
    public void paint(Graphics graphics) {  
        super.paint(graphics);  
        String msg = "I rule!!";  
        graphics.drawString(msg, 100, 100);  
    }  
  
    public static void main(String[] args) {  
        MyFrame myFrame = new MyFrame("Head First Design Patterns");  
    }  
}
```

Swing Framework에서
Jframe의 paint()를 호출합
니다. paint()는 후크임.

템플릿 메소드 적용사례- Swing Framework

```
public class MyFrame extends JFrame {  
    private static final long serialVersi
```

```
    public MyFrame(String title) {  
        super(title);  
        this.setDefaultCloseOperation
```

```
        this.setSize(300,300);  
        this.setVisible(true);  
    }
```

```
    public void paint(Graphics graphics)  
        super.paint(graphics);  
        String msg = "I rule!!";  
        graphics.drawString(msg, 100,  
    }
```

```
    public static void main(String[] args) {  
        MyFrame myFrame = new MyFrame("Head First Design Patterns");  
    }
```

```
}
```



Head First Design Patterns

I rule!!

핵심 정리

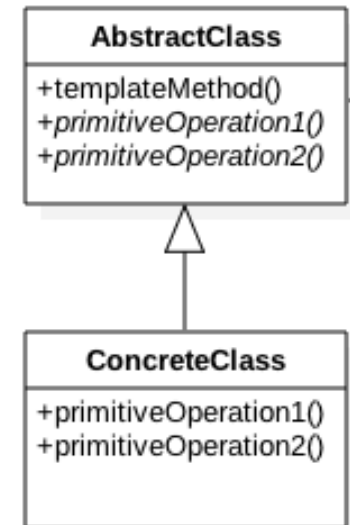
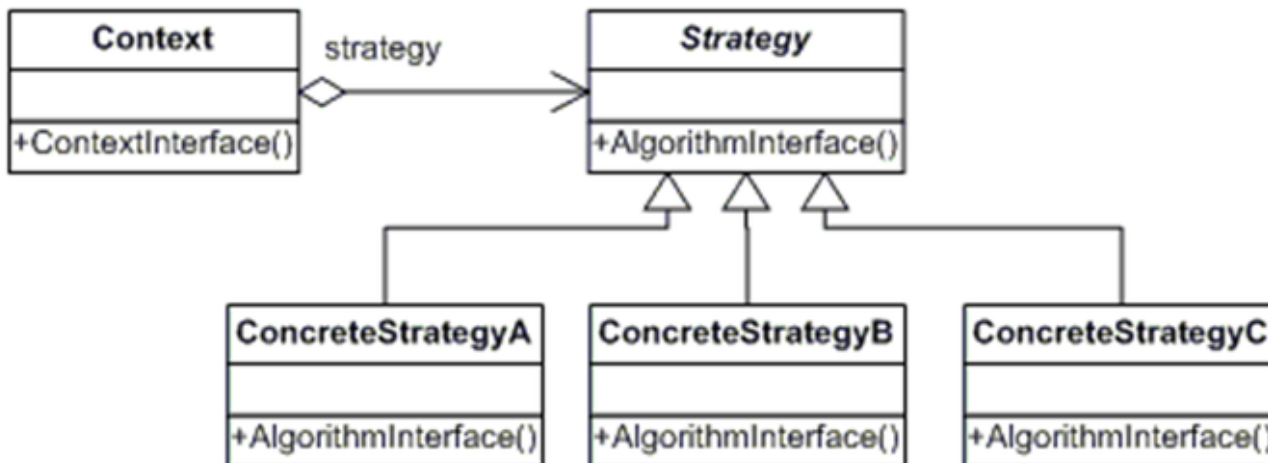
- “템플릿 메소드”에서는 알고리즘의 단계들을 정의하는데, 일부 단계는 서브클래스에서 구현하도록 할 수 있습니다.
- 템플릿 메소드 패턴은 코드 재사용에 크게 도움이 됩니다.
- 템플릿 메소드가 들어 있는 추상클래스에서는 구상 메소드, 추상 메소드, 후크를 정의할 수 있습니다.
- 추상 메소드는 서브클래스에서 구현합니다.
- 후크(hook)는 추상 클래스에 들어 있는, 아무일도 하지 않거나 기본 행동을 정의하는 메소드로, 서브클래스에서 재정의할 수 있습니다.

핵심 정리

- 서브 클래스에서 템플릿 메소드에 들어 있는 알고리즘을 함부로 바꾸지 못하게 하고 싶다면, 템플릿 메소드를 final로 선언하면 됩니다.
- 헐리우드 원칙에 의하면, 저수준 모듈을 언제 어떻게 호출할지는 고수준 모듈에서 결정하는 것이 좋습니다.
- 템플릿 메소드 패턴은 실전에서도 꽤 자주 쓰이지만, 반드시 “교과서적인” 방식으로 적용되진 않습니다.
- 스트래티지 패턴과 템플릿 메소드 패턴은 모두 알고리즘을 캡슐화하는 패턴이지만 전자에서는 구성을, 후자에서는 상속을 이용합니다.
- 팩토리 메소드 패턴은 특화된 템플릿 메소드 패턴입니다.

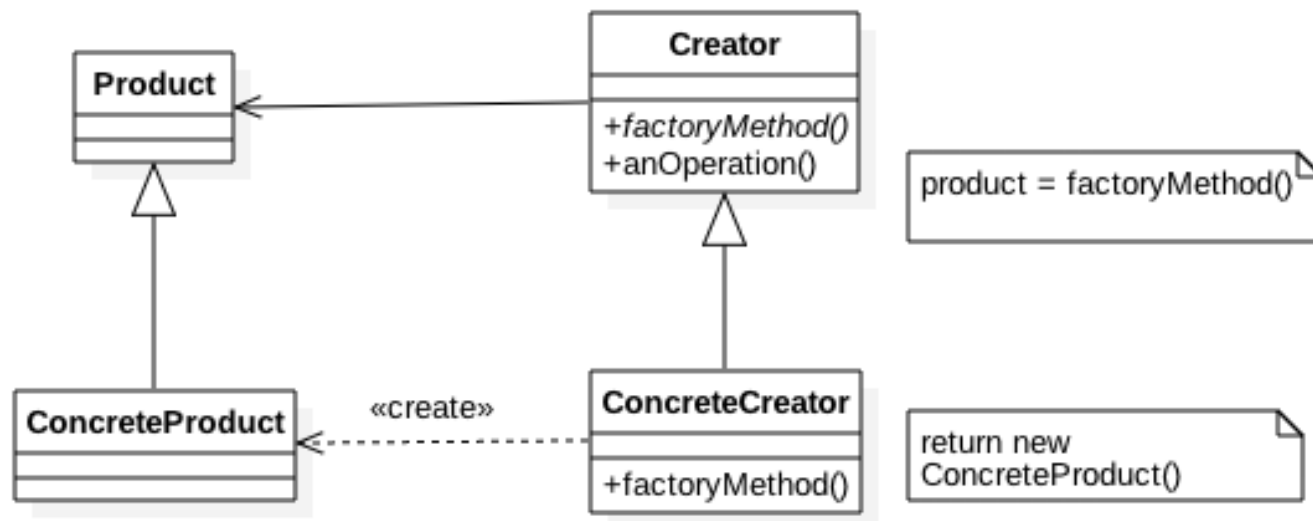
다음 패턴 간의 공통점과 차이점은?

- 스트래티지 패턴 vs. 템플릿 메소드 패턴
 - 공통점
 - 알고리즘의 캡슐화
 - 차이점
 - 스트래티지 패턴: 객체 구성 활용
 - 템플릿 메소드 패턴: 클래스 상속 사용



다음 패턴 간의 공통점과 차이점은?

- 팩토리 메소드 패턴 vs. 템플릿 메소드 패턴
 - 공통점
 - 알고리즘의 구조 정의
 - 차이점
 - 팩토리 메소드 패턴은 객체 생성을 담당하는 특화된 템플릿 메소드 패턴임.



피자 가게 프레임워크 (revisited)

- 피자를 만드는 **공통 활동**은 PizzaStore 클래스로..
- 분점마다 **고유한 스타일**은 PizzaStore의 서브 클래스에..

```
public abstract class PizzaStore {
```

```
    abstract Pizza createPizza(String item);
```

분점 고유의 스타일은 서브 클래스의 createPizza() 메소드에서 재정의

```
    public Pizza orderPizza(String type) {
```

```
        Pizza pizza = createPizza(type);
```

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

```
        pizza.prepare();
```

```
        pizza.bake();
```

```
        pizza.cut();
```

```
        pizza.box();
```

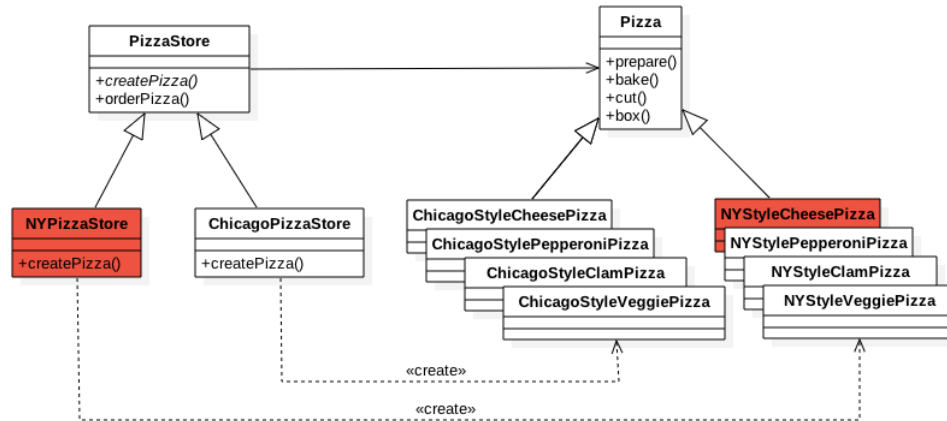
```
        return pizza;
```

```
    }
```

```
}
```

분점마다 동일하게 진행되는
공통활동

PizzaStore 서브 클래스 (revisited)



```
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;
    }
}

```