

데코레이터 패턴

이관우

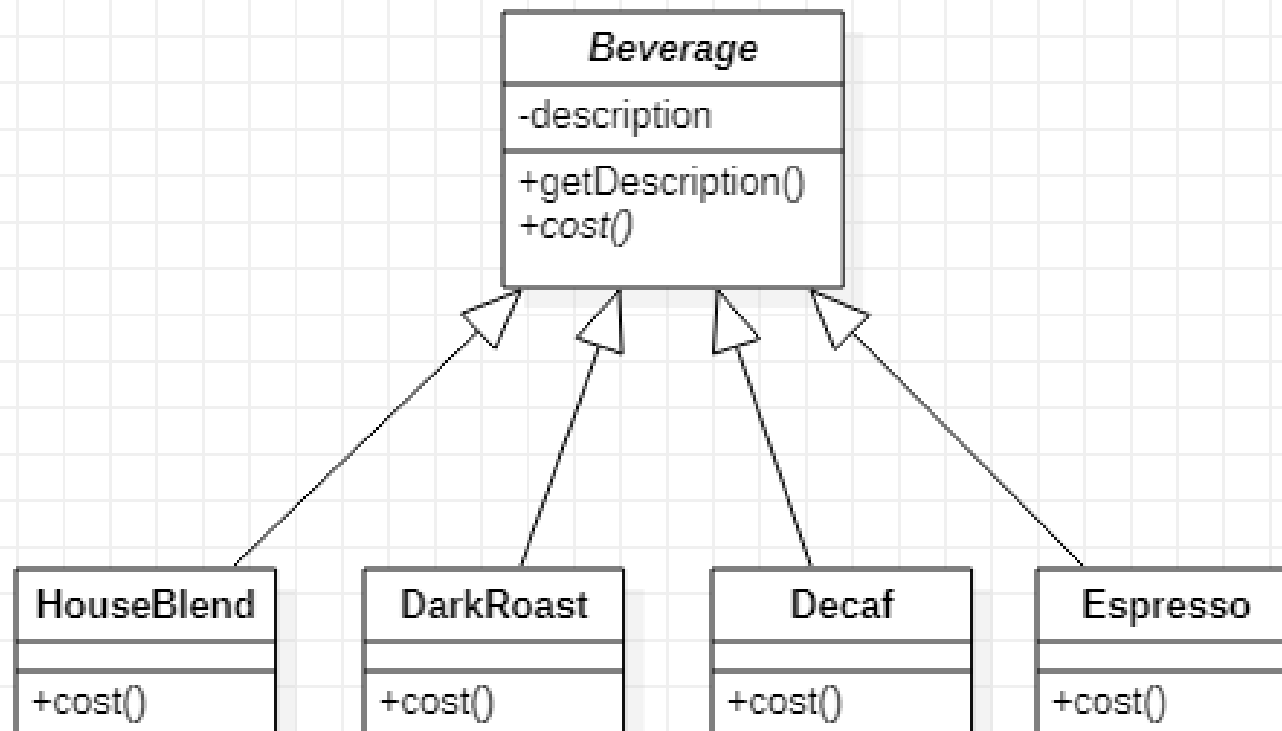
kwlee@hansung.ac.kr

학습 목표

- (문제 상황) 데코레이터 패턴이 필요한 상황을 이해한다.
- (해결 방안) 데코레이터 패턴의 작동 메커니즘을 이해한다.
- (구현) 데코레이터 패턴을 직접 구현해 본다.
- (적용사례) 데코레이터 패턴의 다양한 적용 예를 이해한다.

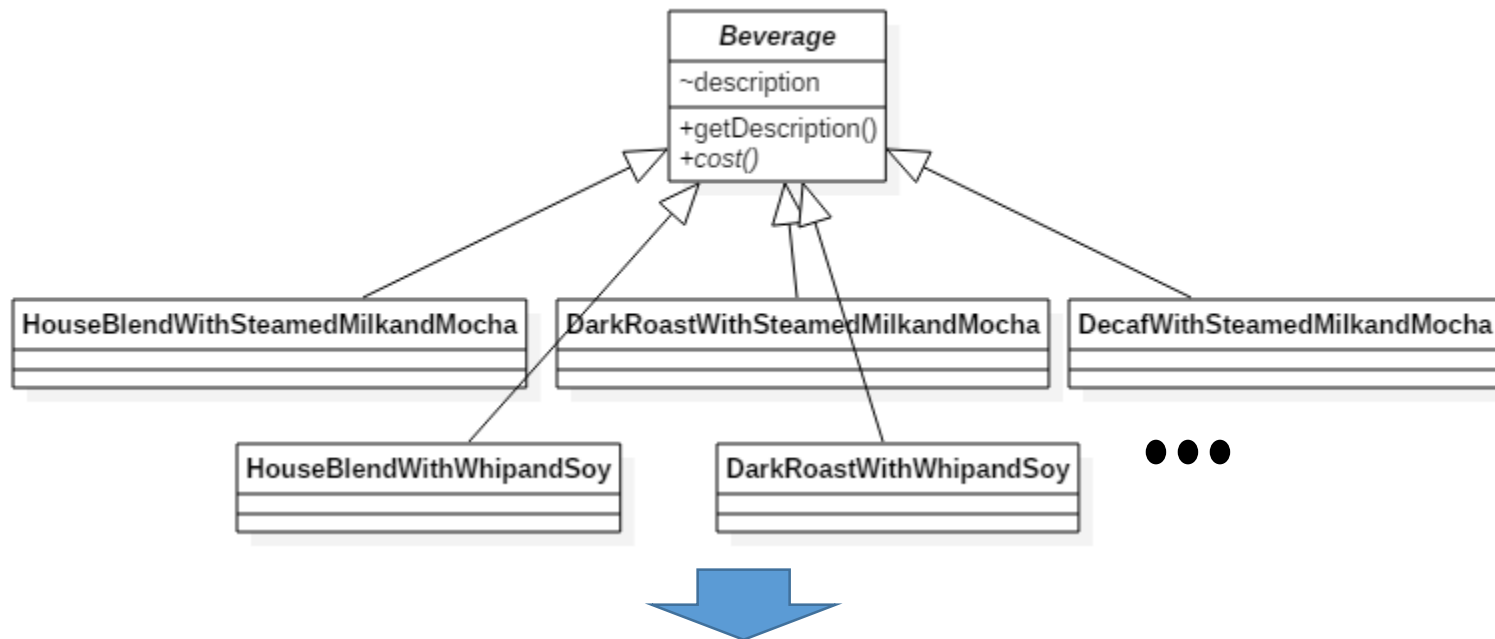
스타버즈 음료 주문 시스템

- 다양한 음료들을 모두 포괄하는 주문 시스템 (초기 설계)



변경 사항 들

- 커피 주문시, 스팀우유나 두유, 모카 추가하기도 하고, 그 위에 휘핑크림을 얹기도 합니다.
 - 각각을 추가할 때마다 커피 가격에 반영 되어야 합니다.



클래스 수의 폭발?

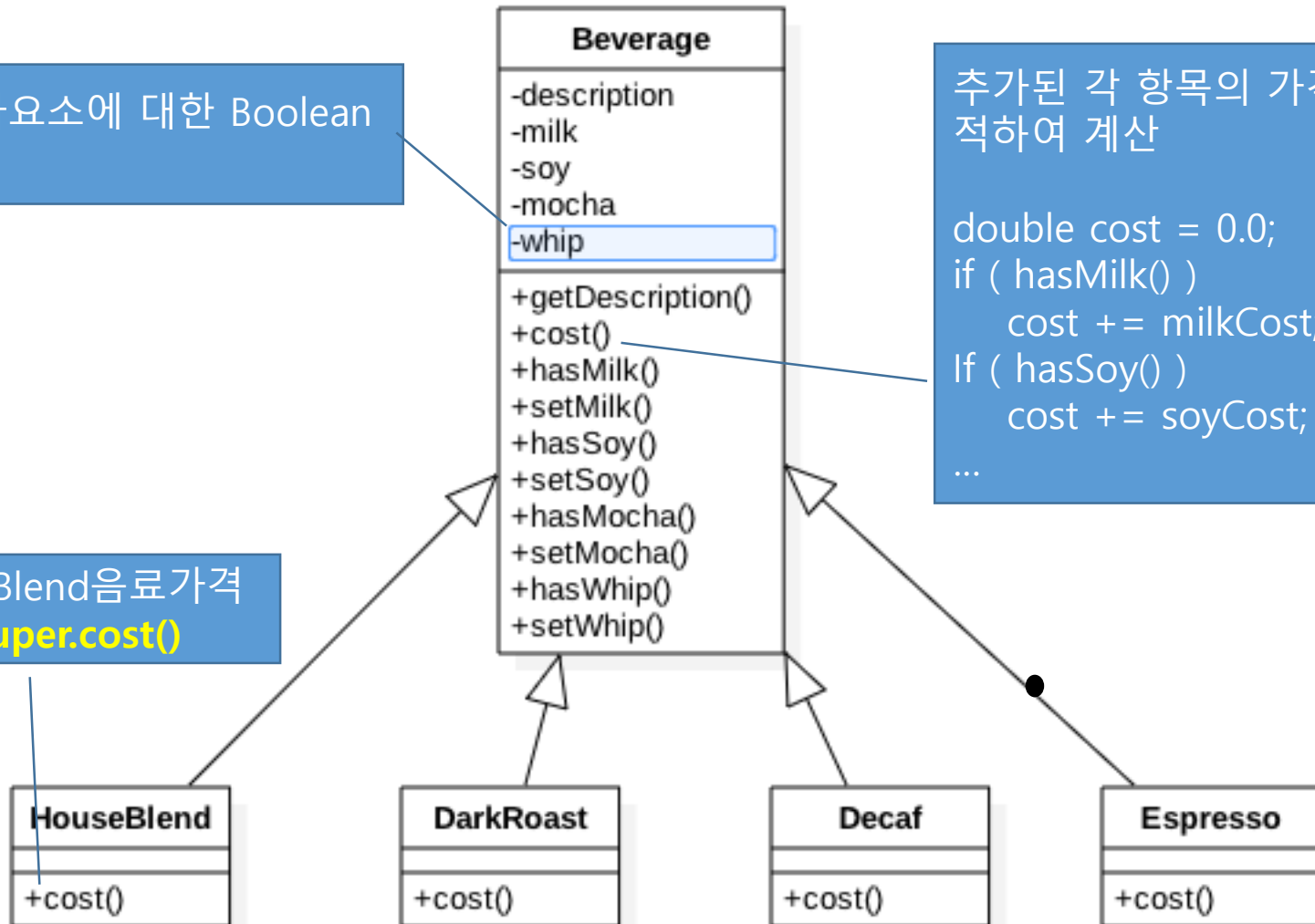
다른 방안?

각 추가요소에 대한 Boolean 변수

추가된 각 항목의 가격을 누적하여 계산

```
double cost = 0.0;
if ( hasMilk() )
    cost += milkCost;
if ( hasSoy() )
    cost += soyCost;
...
```

HouseBlend 음료 가격
+ **super.cost()**



코드 예제

예제 코드 프로젝트 링크

<https://github.com/kwanulee/DesignPattern/tree/master/decorator/StarbuzzAlternative>

문제점은?

- 첨가물 가격이 바뀔 때마다 기존 코드(Beverage 클래스) 수정
- 첨가물의 종류가 추가될 때마다 Beverage 클래스에서 새로운 메소드 (`hasXXX()`, `setXXX()`) 추가, `cost()` 메소드 수정
- 새로운 음료가 특정 첨가물을 제한하는 경우는 문제가 있음
- 손님이 더블 모카를 주문한다면?
- ???

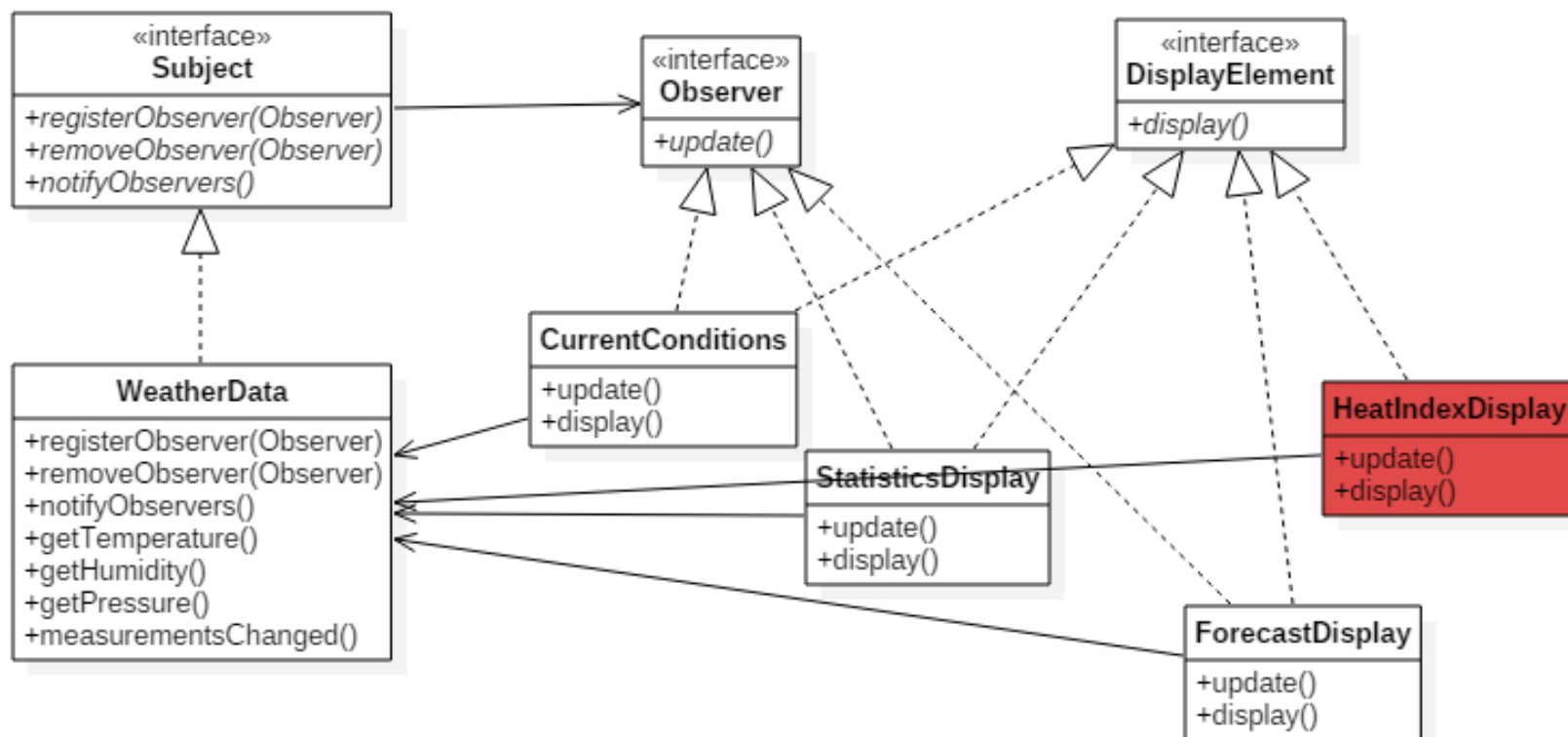
디자인 원칙-OCP (Open-Closed Principle)

클래스는 확장에 대해서는 열려 있어야 하지만, 코드 변경에 대해서는 닫혀 있어야 한다

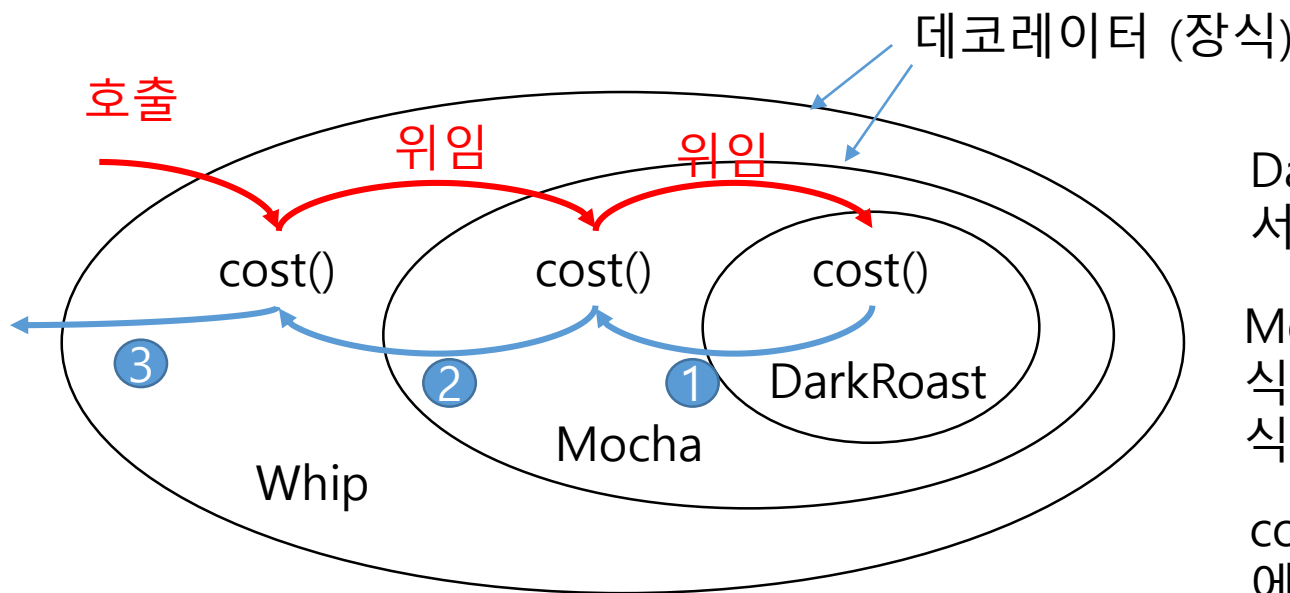


기존 코드를 건드리지 않고 확장을 통해 새로운 행동을 추가

예: 코드 변경 없이 확장하기



데코레이터 패턴 동작 원리 (장식, 위임)



DarkRoast는 Beverage의 서브 클래스

Mocha, Whip 모두 같은 형식(Beverage)의 객체를 장식함

`cost()` 계산은 장식된 객체에 일부 위임

① 99 센트 리턴

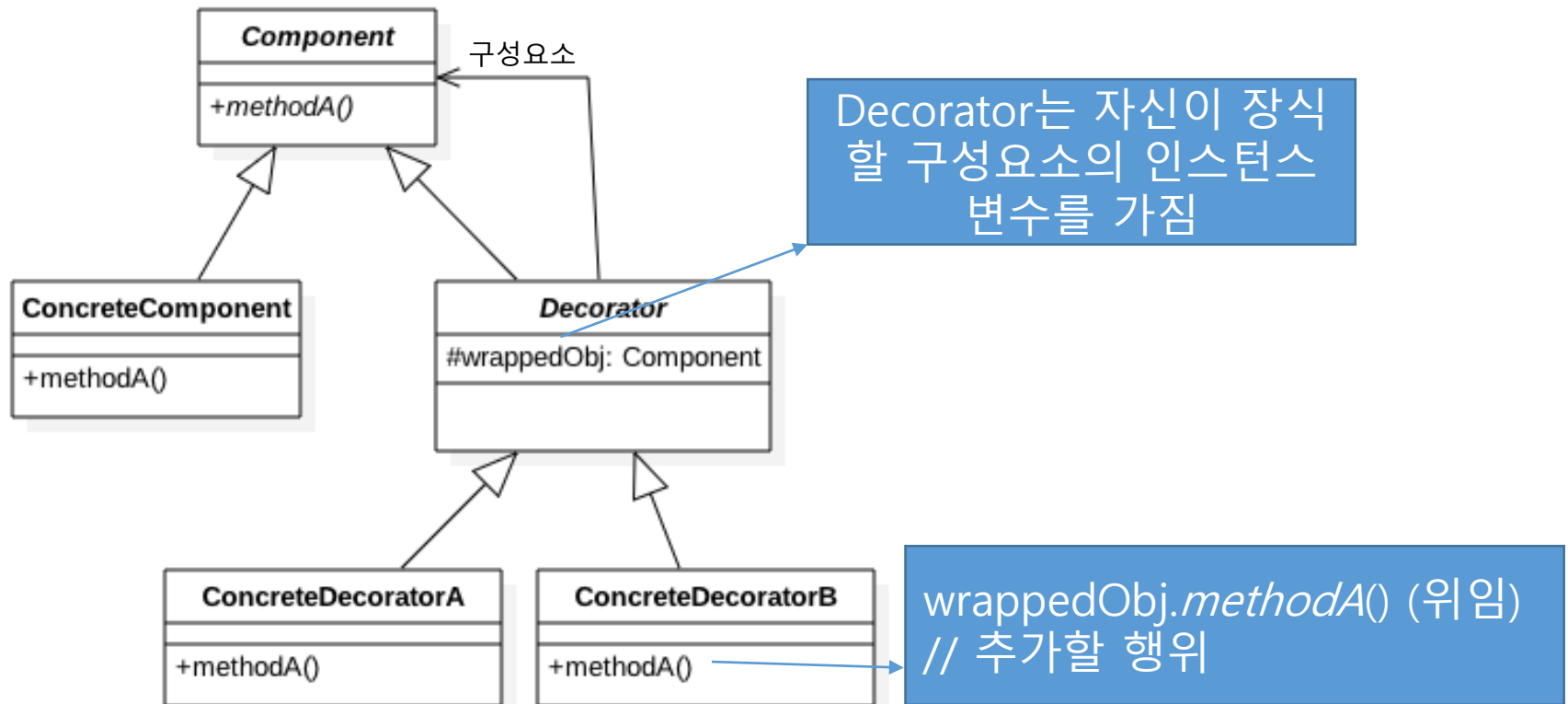
② DarkRoast 리턴 + Mocha 값 (20센트)

③ Mocha 리턴 + Whip 값 (10센트) = \$1.29

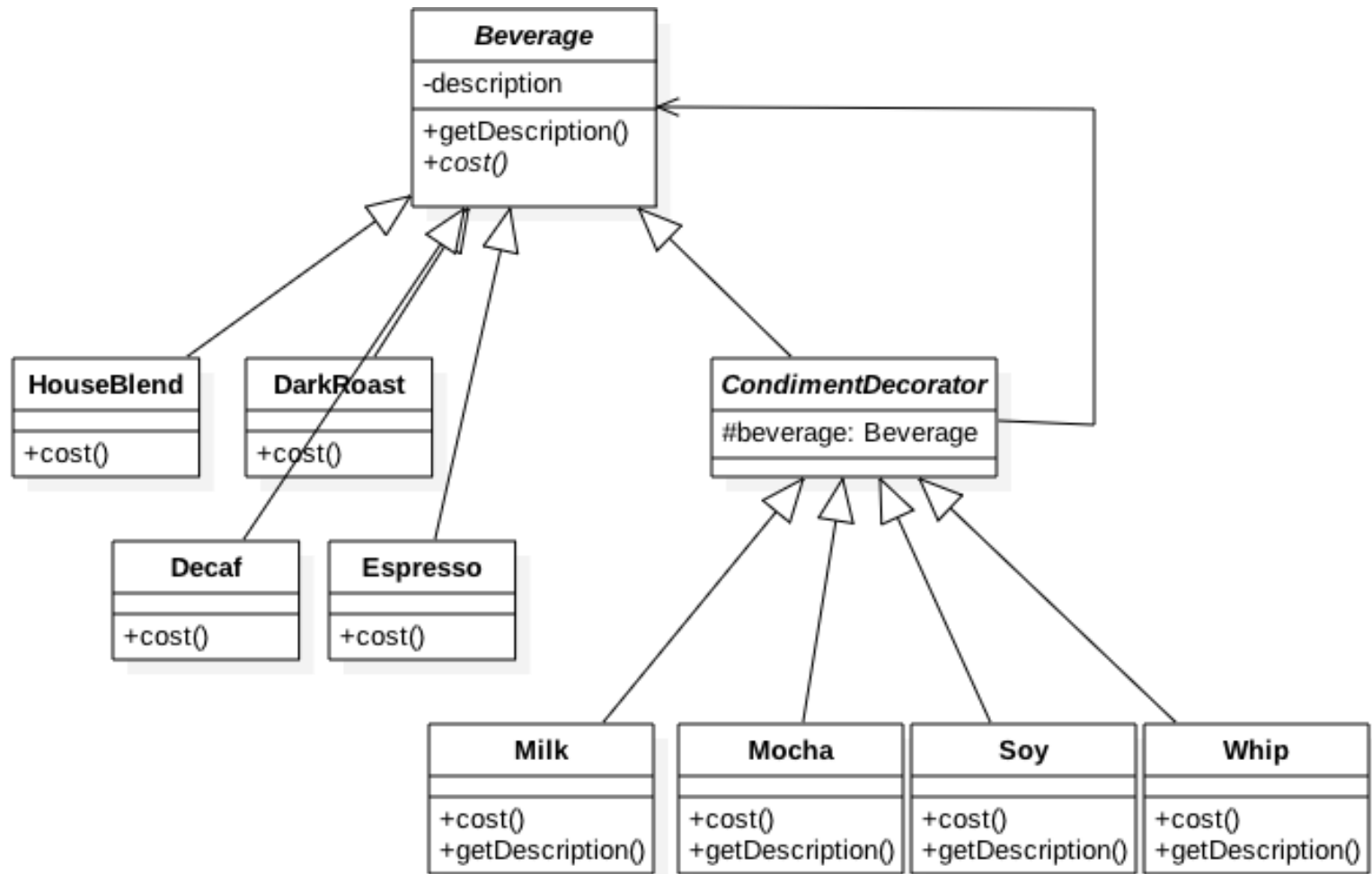
데코레이터 패턴

• 정의

- 데코레이터 패턴에서는 객체에 **추가적인 요건을 동적으로 첨가**
- 데코레이터는 **서브클래스를 만드는 것을 통해서 기능을 유연하게 확장**할 수 있는 방법을 제공



스타버즈에 데코레이터 패턴 적용



스타버즈 구현

```
public abstract class Beverage {  
    String description = "Unknown Beverage";  
  
    public String getDescription() {  
        return description;  
    }  
  
    public abstract double cost();  
}  
  
public abstract class CondimentDecorator extends Beverage {  
    protected Beverage beverage;  
  
    public CondimentDecorator(Beverage beverage) {  
        this.beverage = beverage;  
    }  
}
```

스타버즈 구현

```
public class HouseBlend extends Beverage {  
    public HouseBlend() {  
        description = "House Blend Coffee";  
    }  
  
    public double cost() {  
        return .89;  
    }  
}
```

```
public class Espresso extends Beverage {  
    public Espresso() {  
        description = "Espresso";  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

스타버즈 구현

```
public class Mocha extends CondimentDecorator {  
  
    public Mocha(Beverage beverage) {  
        super(beverage);  
    }  
  
    public String getDescription() {  
        return beverage.getDescription() + ", Mocha";  
    }  
  
    public double cost() {  
        return .20 + beverage.cost();  
    }  
}
```

스타버즈 구현 (테스트 코드)

```
public class StarbuzzCoffee {
```

```
    public static void main(String args[]) {
```

```
        Beverage beverage = new Espresso();  
        System.out.println(beverage.getDescription()  
                             + " $" + beverage.cost());
```

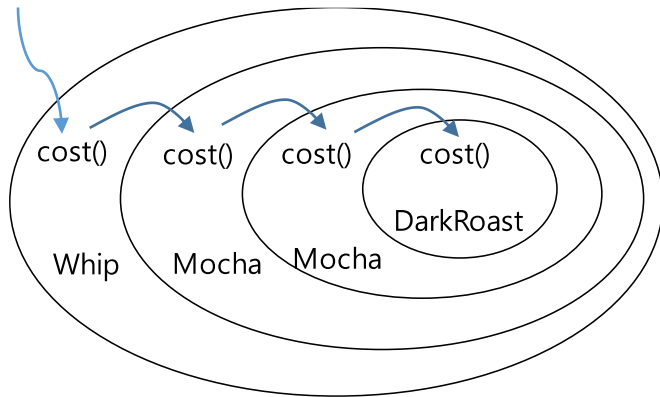
```
        Beverage beverage2 = new DarkRoast();  
        beverage2 = new Mocha(beverage2);  
        beverage2 = new Mocha(beverage2);  
        beverage2 = new Whip(beverage2);  
        System.out.println(beverage2.getDescription()  
                             + " $" + beverage2.cost());
```

```
        ...
```

```
    }
```

```
}
```

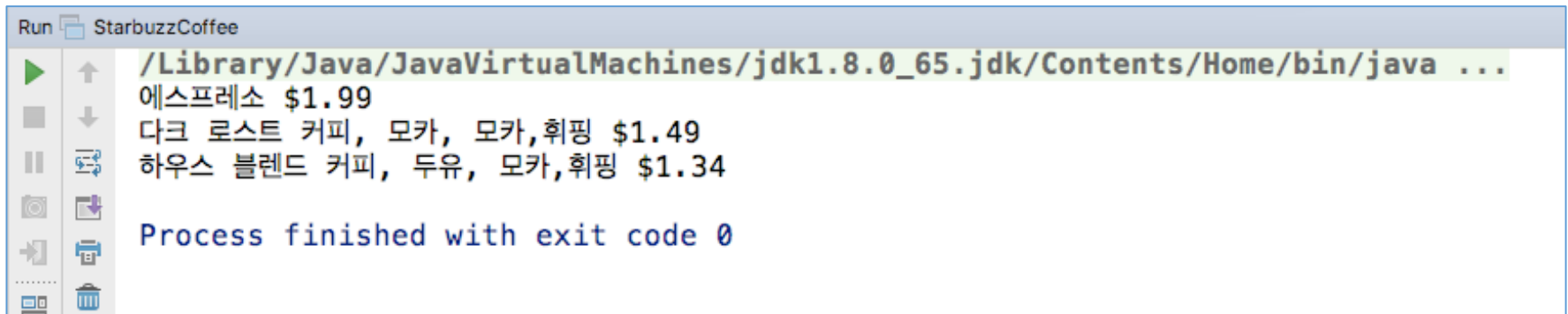
beverage2.cost()



스타버즈 프로젝트 테스트

- 예제 코드 프로젝트 링크

<https://github.com/kwanulee/DesignPattern/tree/master/decorator/starbuzz>



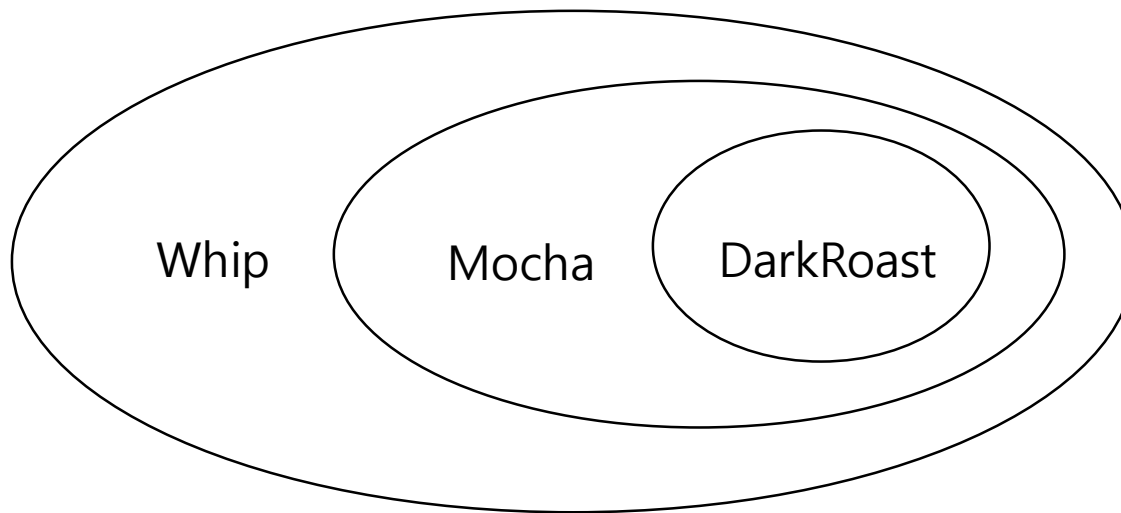
```
Run StarbuzzCoffee
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java ...
에스프레소 $1.99
다크 로스트 커피, 모카, 모카, 휘핑 $1.49
하우스 블렌드 커피, 두유, 모카, 휘핑 $1.34
Process finished with exit code 0
```

데코레이터 패턴의 정리

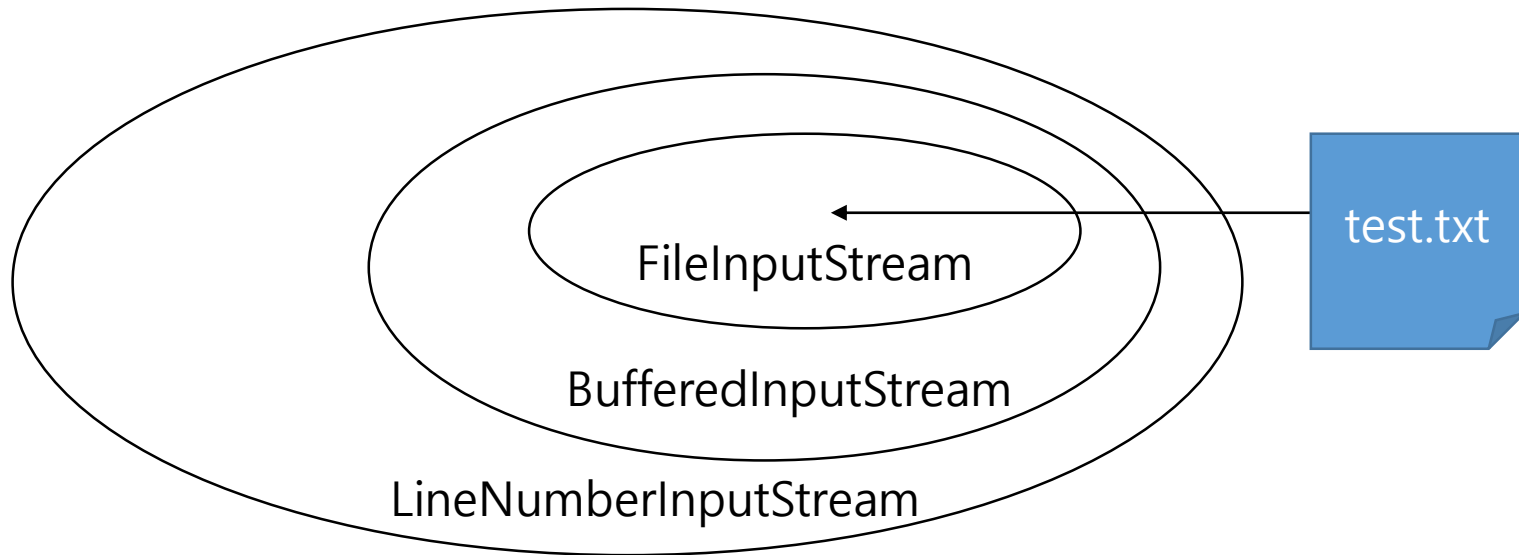
- 데코레이터의 수퍼클래스는 자신이 장식하고 있는 객체의 수퍼클래스와 같다.
- 한 객체를 여러 개의 데코레이터로 감쌀 수 있다.
- 데코레이터는 자신이 감싸고 있는 객체와 같은 수퍼클래스를 가지고 있기 때문에 원래 객체(싸여져 있는 객체)가 들어갈 자리에 데코레이터 객체를 집어 넣어도 상관없다.
- 데코레이터는 자신이 장식하고 있는 객체에게 어떤 행동을 위임하는 것 외에 원하는 추가적인 작업을 수행할 수 있다.
- 객체는 언제든지 감쌀 수 있기 때문에 실행 중에 필요한 데코레이터를 마음대로 적용할 수 있다.

데코레이터 패턴의 정리

- 데코레이터는 그것이 감싸는 구성요소 들 중에 무엇이 구체적으로 있는지 알 수 없다. 따라서, 그것이 감싸는 특정한 구성요소에 따라서 다른 행동을 할수는 없다.
 - 예를 들면, "DarkRoast 커피에 추가된 휘핑크림은 50% 할인"과 같은 행동은 데코레이터에서는 구조적으로 불가능하다.

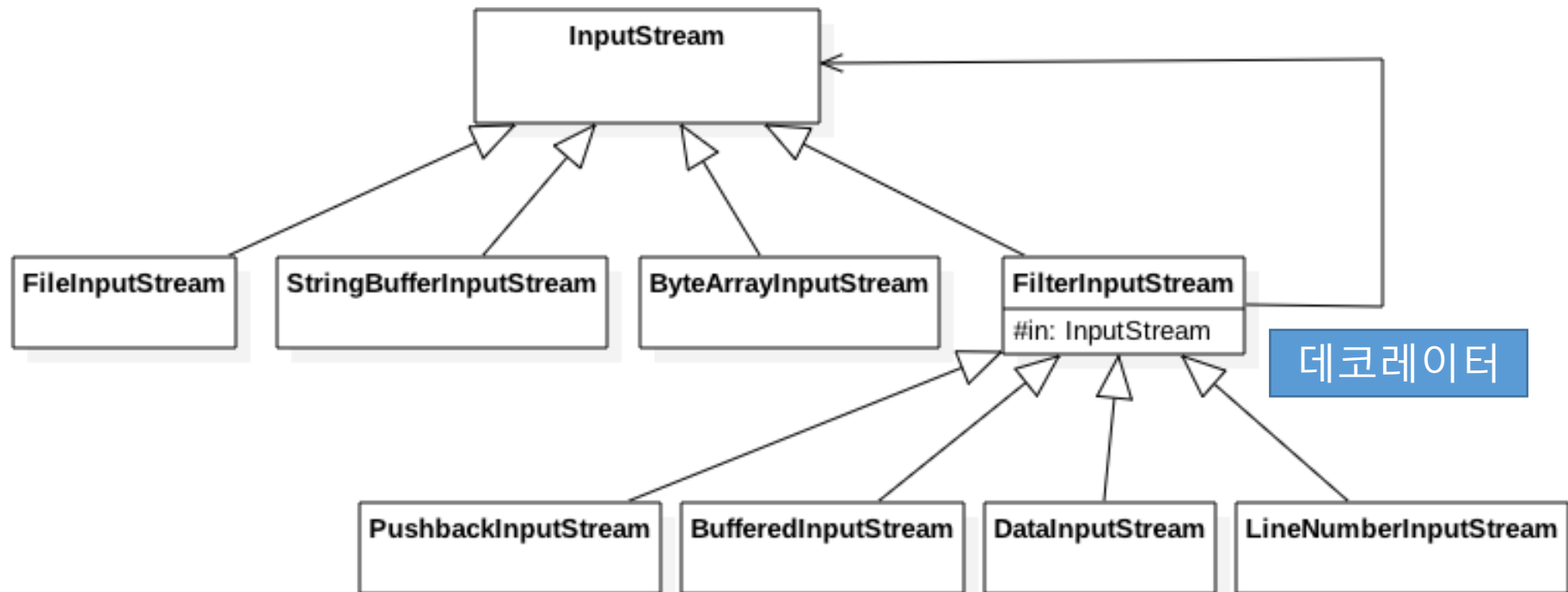


데코레이터가 적용된 예: 자바 I/O

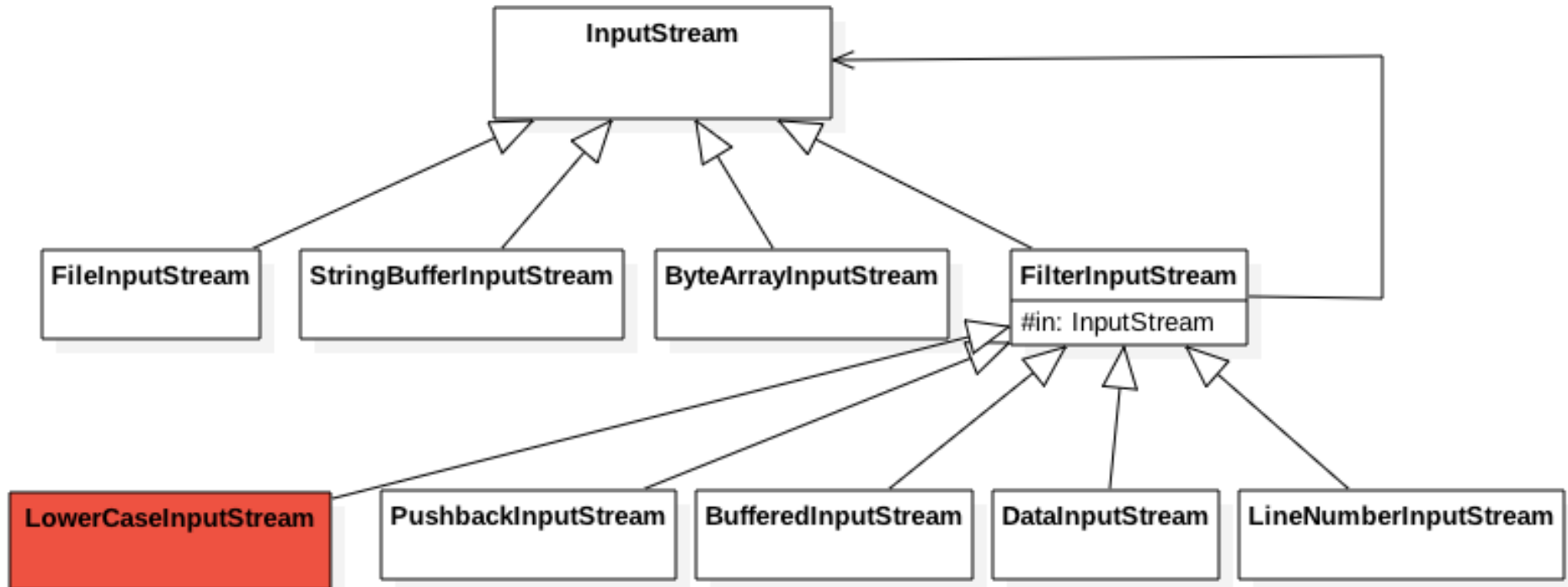


```
InputStream in = new LineNumberInputStream(  
    new BufferedInputStream(  
        new FileInputStream("test.txt")));
```

Java.io 클래스와 데코레이터 패턴



새로운 Java I/O 데코레이터 추가



새로운 Java I/O 데코레이터 추가

```
public class LowerCaseInputStream extends FilterInputStream {  
    public LowerCaseInputStream(InputStream in) {  
        super(in);  
    }  
  
    public int read() throws IOException {  
        int c = in.read();  
        return (c == -1 ? c : Character.toLowerCase((char)c));  
    }  
  
    public int read(byte[] b, int offset, int len) throws IOException {  
        int result = in.read(b, offset, len);  
        for (int i = offset; i < offset+result; i++) {  
            b[i] = (byte)Character.toLowerCase((char)b[i]);  
        }  
        return result;  
    }  
}
```

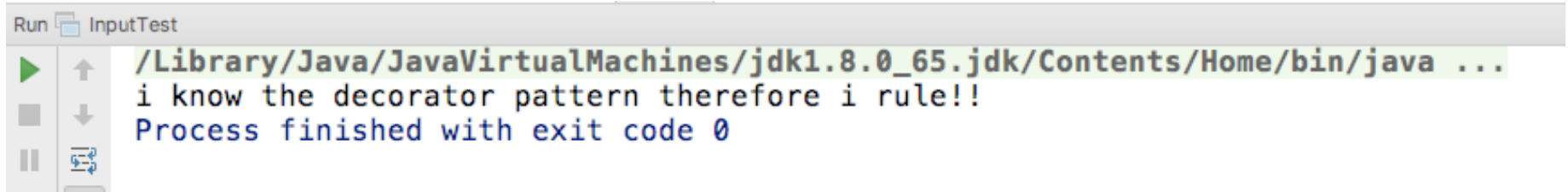
새로운 Java I/O 데코레이터 추가

```
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
  
        try {  
            InputStream in = new  
                LowerCaseInputStream(  
                    new BufferedInputStream(  
                        new FileInputStream("test.txt")));  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```


Java I/O 데코레이터 프로젝트 테스트

- 예제 코드 프로젝트 링크

<https://github.com/kwanulee/DesignPattern/tree/master/decorator/io>



The screenshot shows a Java IDE's Run console window. The title bar reads "Run InputTest". On the left, there are icons for running (a green play button), stepping through (a square), and debugging (a bug icon). The console output is as follows:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java ...  
i know the decorator pattern therefore i rule!!  
Process finished with exit code 0
```

핵심 정리

- 상속을 통해 확장을 할 수도 있지만, 디자인의 유연성 면에서 보면 별로 좋지 않음.
- 구성과 위임을 통해서 실행중에 새로운 행동을 추가할 수 있음
- 데코레이터 패턴에서는 구상 구성요소(Concrete Component)를 감싸주는 데코레이터들을 사용
- 데코레이터 클래스의 형식은 그 클래스가 감싸고 있는 클래스의 형식을 반영

핵심 정리

- 데코레이터에서는 자기가 감싸고 있는 구성요소의 메소드를 호출한 결과에 새로운 기능을 더함으로써 행동을 확장
- 구성요소를 감싸는 데코레이터의 개수에는 제한없음
- 구성요소의 클라이언트에서 구성요소의 구체적인 형식에 의존하게 되는 경우는 데코레이터 패턴을 사용할 수 없음.
- 데코레이터 패턴을 사용하면 자잘한 객체들이 매우 많이 추가될 수 있고, 데코레이터를 너무 많이 사용하면 코드가 필요 이상으로 복잡해질 수 있음.