

고급객체지향 프로그래밍 강의노트 #05

Decorator Pattern

조용주

ycho@smu.ac.kr

데코레이터 패턴 (Decorator Pattern)

□ 목적

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- 객체에 추가적인 책임을 동적으로 부여함 . 데코레이터는 서브클래싱 (상속) 을 사용하지 않아도 유연하고 융통성 있는 기능 확장을 가능하게 함

문제

- 조금씩 기능을 추가하기 위해 새로운 클래스를 생성하는 경우
 - 상속으로 문제를 풀면 너무 많은 상속 관계가 발생할 수 있음
 - 상속을 사용하지 않고 새로운 기능을 추가할 수 있나?

디자인 패턴 요소

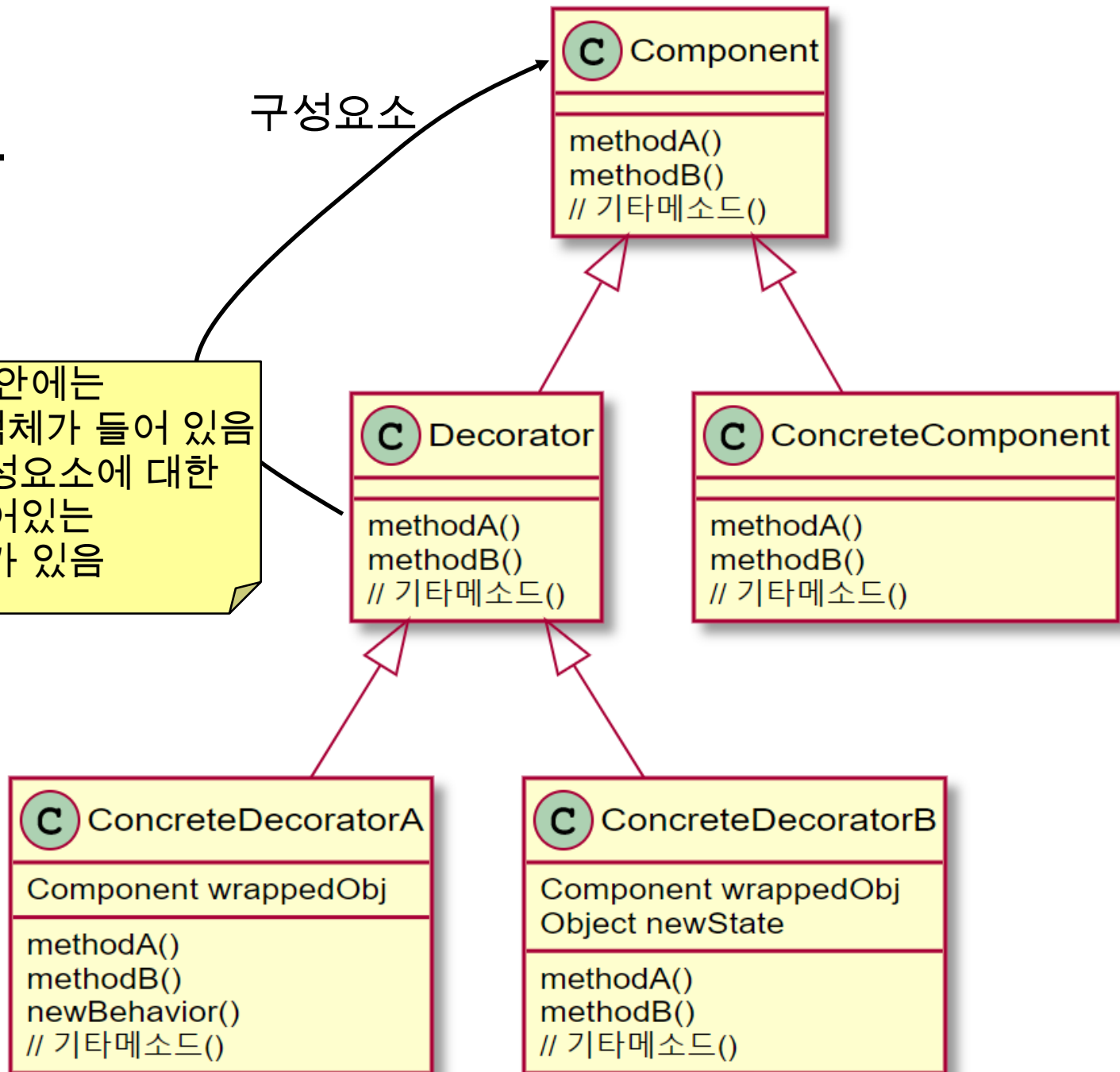
요소	설명
이름	데코레이터 (Decorator)
문제	조금씩 다른 다양한 종류 . 늘어날수록 확장 어려움
해결방안	상속을 사용하지 않고 연관으로 필요한 기능 추가 . 실행시점 확장 Extension at runtime (not compile time)
결과	확장성

■ 데코레이터 패턴 정의

- 데코레이터 패턴에서는 객체에 추가적인 요건을 동적으로 첨가
 - 서브클래스를 만드는 것을 통해 기능을 유연하게 확장할 수 있는 방법을 제공

각 데코레이터 안에는
Component 객체가 들어 있음
데코레이터 구성요소에 대한
레퍼런스가 들어있는
인스턴스 변수가 있음

구성요소



데코레이터 패턴 정의

□ Component

- 각 구성요소는 직접 쓰일 수도 있고 데코레이터로 감싸져서 쓰일 수도 있음 (클래스 또는 인터페이스)

□ ConcreteComponent

- 새로운 행동을 동적으로 추가

□ Decorator 는 자신이 장식할 구성요소와 같은 인터페이스 또는 추상 클래스를 구현

□ ConcreteDecorator 에는 그 객체가 장식하고 있는 것 (데코레이터가 감싸고 있는 Component 객체) 을 위한 인스턴스 변수가 있음

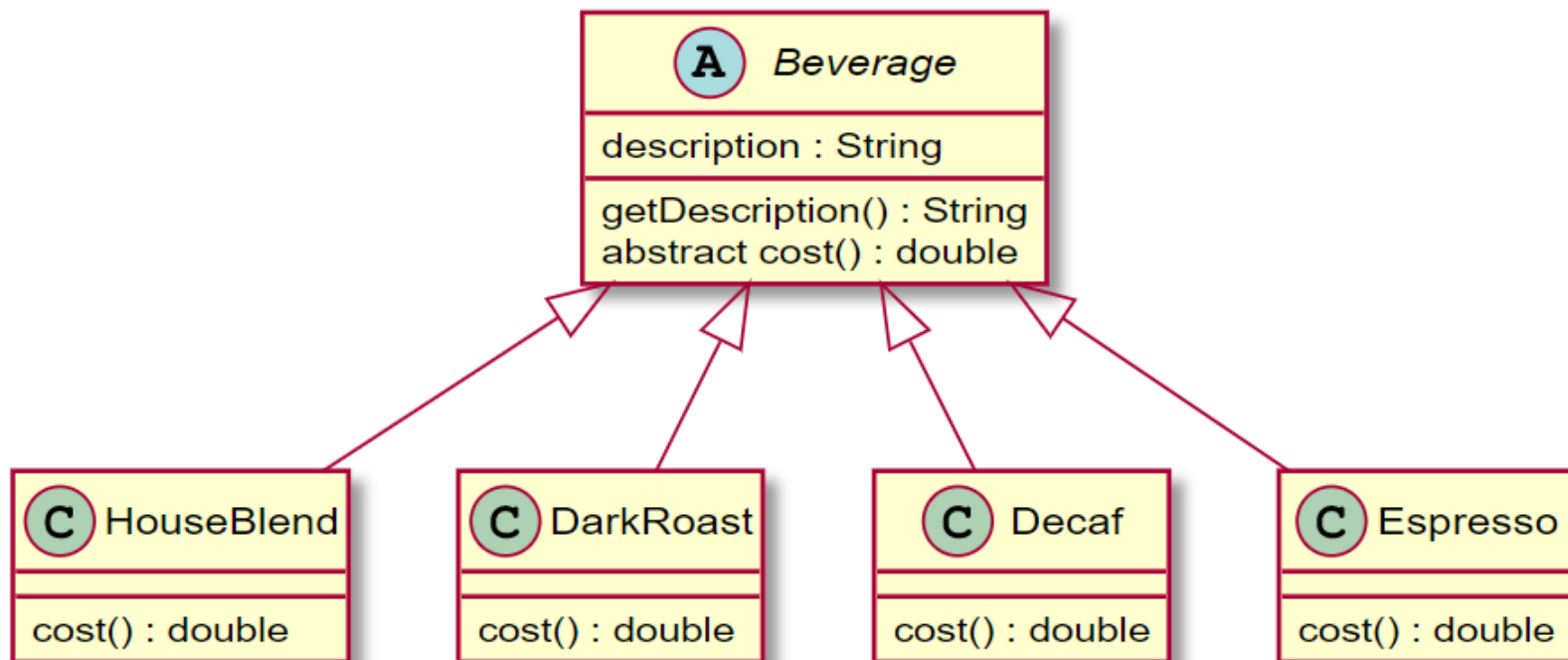
□ Decorator 는 Component 의 상태를 확장할 수 있음

데코레이터 패턴 정의

- 데코레이터에서 새로운 메소드를 추가할 수 있음 .
하지만 일반적으로 새로운 메소드를 추가하는 대신
Component 에 원래 있던 메소드를 호출하기 전 ,
또는 후에 별도의 작업을 처리하는 방식으로 새로운
기능을 추가
- 데코레이터 패턴에서의 상속은 기능 (행동) 을 물려
받기 위해서가 아니라 형식을 맞추기 위해서임
- Component 는 추상 클래스 또는 인터페이스로
구현 가능

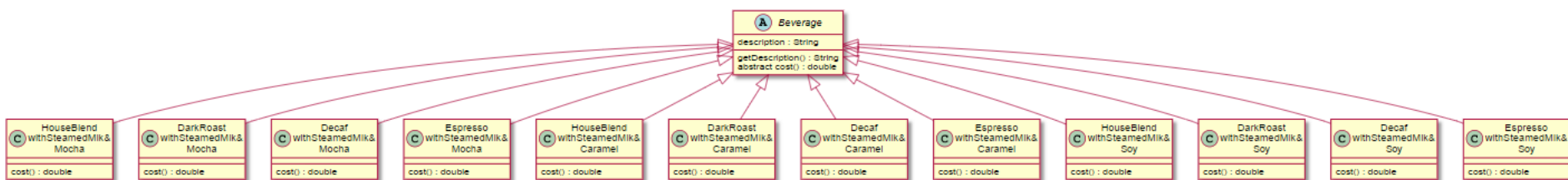
사례 1: 스타버즈 커피

- 스타버즈 커피는 급속도로 성장한 초대형 커피 전문점
 - 너무 빠르게 성장하다 보니까 다양한 음료들을 모두 포괄하는 주문 시스템을 갖추려고 함
 - 초기 시스템은 아래와 같은 형태로 구성됨



스타버즈 커피

- 커피를 주문할 때에는 스팀 우유나 두유, 모카 (초콜릿) 을 추가하기도 하고 휘핑크림을 얹기도 함 (이때마다 가격이 추가됨)



■ 문제

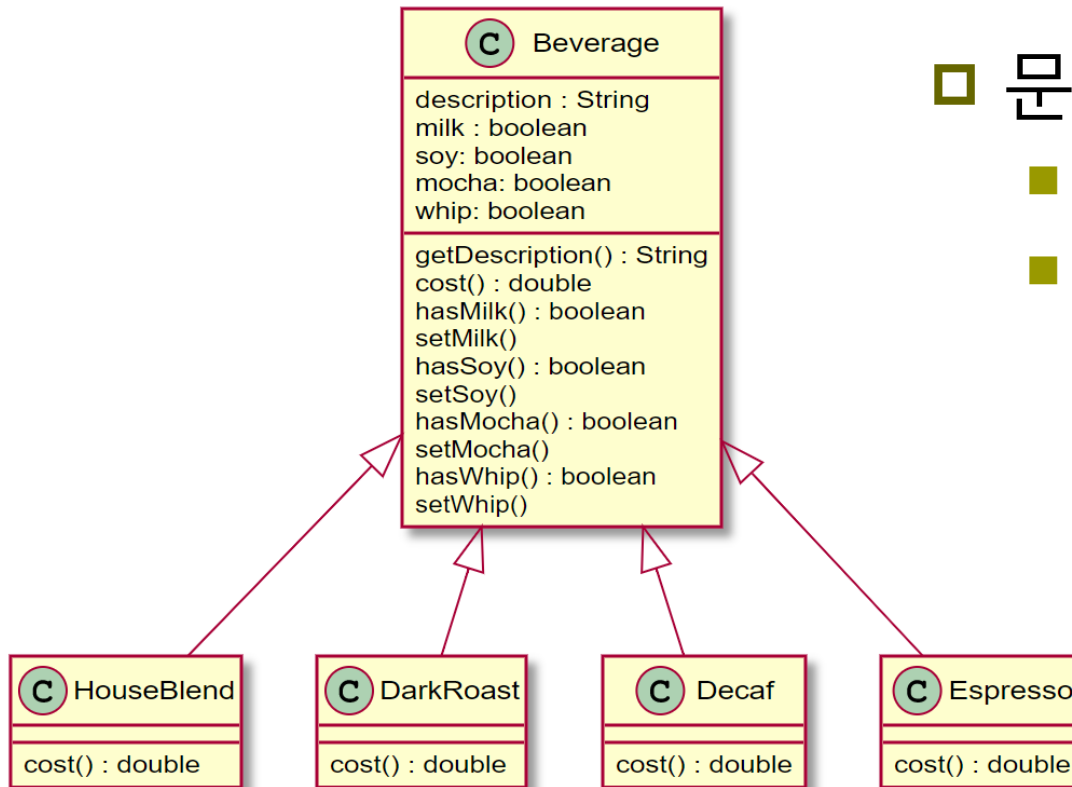
■ 관리의 어려움

- 새로운 토핑이 추가된다면 ?
- 기존에 들어가는 재료 (우유, 크림 등) 의 가격이 인상된다면 ?

수정된 설계

□ 멤버 변수를 사용하고 상속 받는다면 ?

- Beverage 의 cost() 에서는 추가되는 토핑 가격의 합을 계산
- 다시 클래스의 cost() 에서는 음료 가격 추가



□ 문제

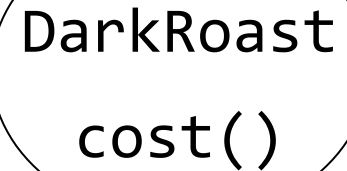
- 새로운 토핑의 추가 ?
- 불필요한 정보 포함

데코레이터 패턴

- 특정 음료에서 시작해서 첨가물로 그 음료를 장식 (decorate)
- 예 : 모카와 휘핑 크림을 추가한 다크 로스트 커피 주문
 - DarkRoast 객체를 가져옴
 - Mocha 객체로 장식
 - Whip 객체로 장식
 - cost() 메소드 호출 (첨가물의 가격을 계산하는 일은 해당 객체들에 위임됨)
 - Mocha, Whip 등은 래퍼 클래스 (Wrapper class) 임

데코레이터 패턴

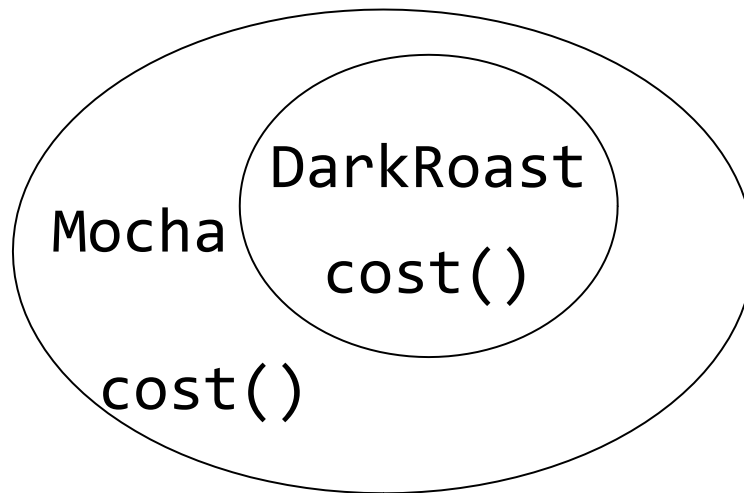
- 데코레이터를 써서 음료 주문을 완성하는 방법
 - DarkRoast 객체에서 시작
 - DarkRoast 는 Beverage 로부터 상속 받음



DarkRoast
cost()

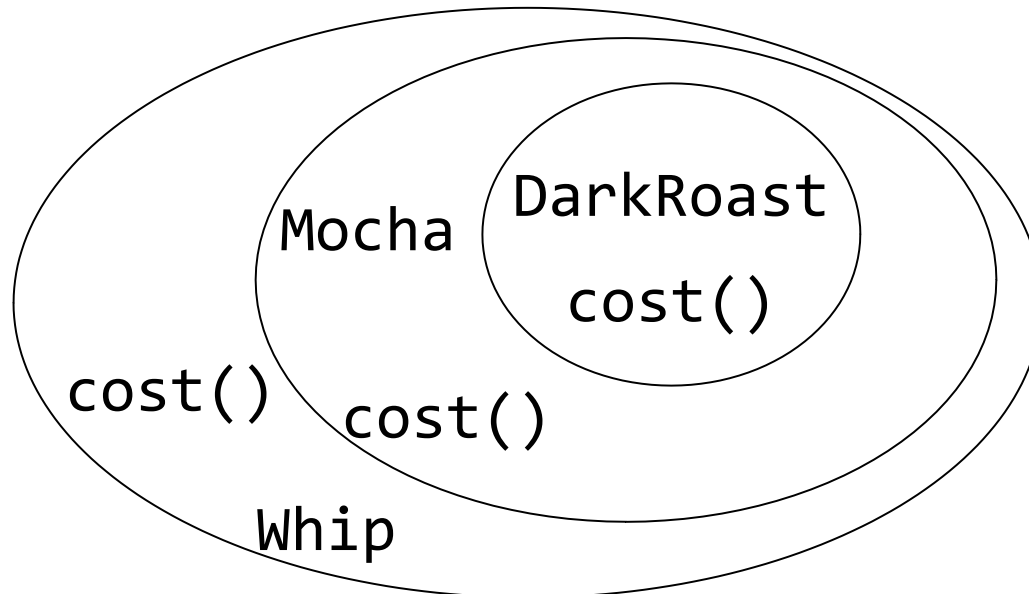
데코레이터 패턴

- 손님이 모카를 주문했으므로, Mocha 객체를 만들고 그 객체로 DarkRoast 를 감싸도록 함 (wrap up)
- Mocha 객체는 데코레이터. 이 객체의 형식은 이 객체가 장식하고 있는 객체를 반영하는데, 여기서는 Beverage
- Mocha 도 Beverage 에서 상속 받은 형태이기 때문에 Beverage 객체로 간주할 수 있음



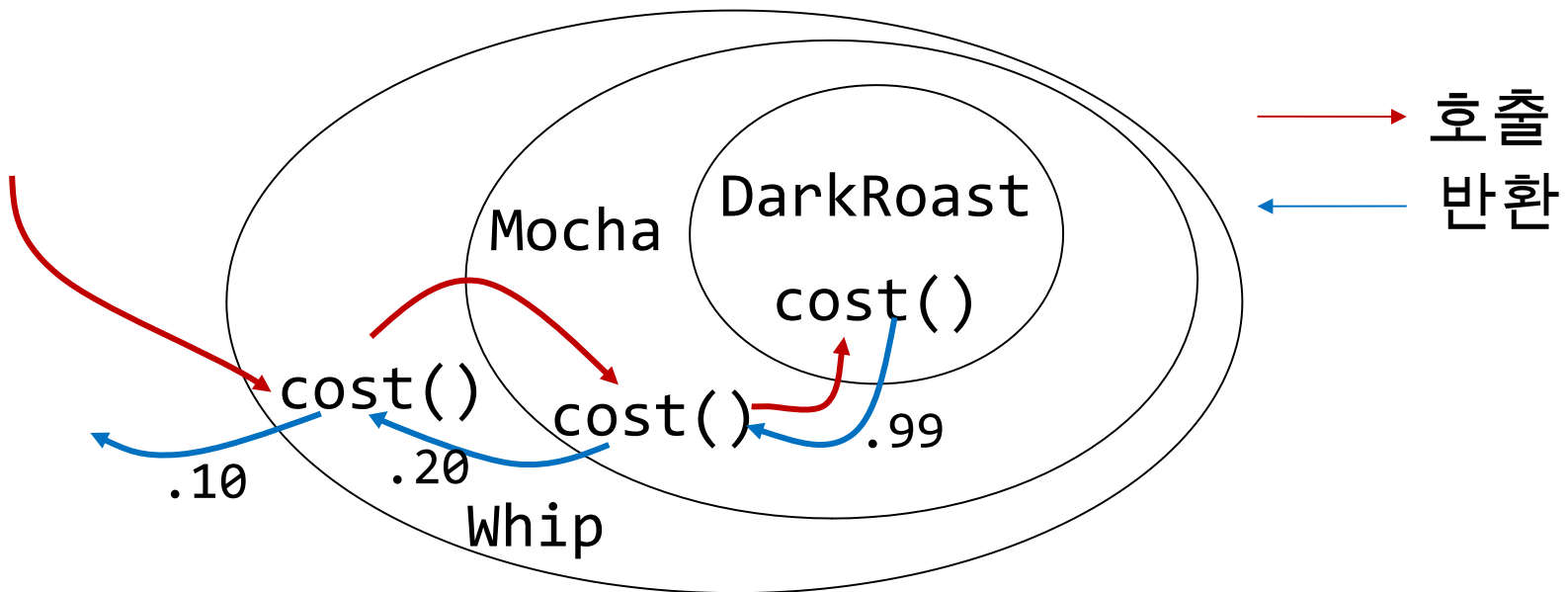
데코레이터 패턴

- 손님이 휘핑 크림도 주문했기 때문에 Whip 데코레이터를 만들고 그 객체로 Mocha 를 감싼다
- Whip 도 데코레이터 이므로 , DarkRoast 의 형식을 반영 (Whip 역시 Beverage 에서 상속됨)
- Mocha 와 Whip 으로 싸여있는 DarkRoast 는 여전히 Beverage 객체이므로 , cost() 메소드 호출을 비롯하여 , DarkRoast 에 대해 할 수 있는 것은 무엇이든 할 수 있음

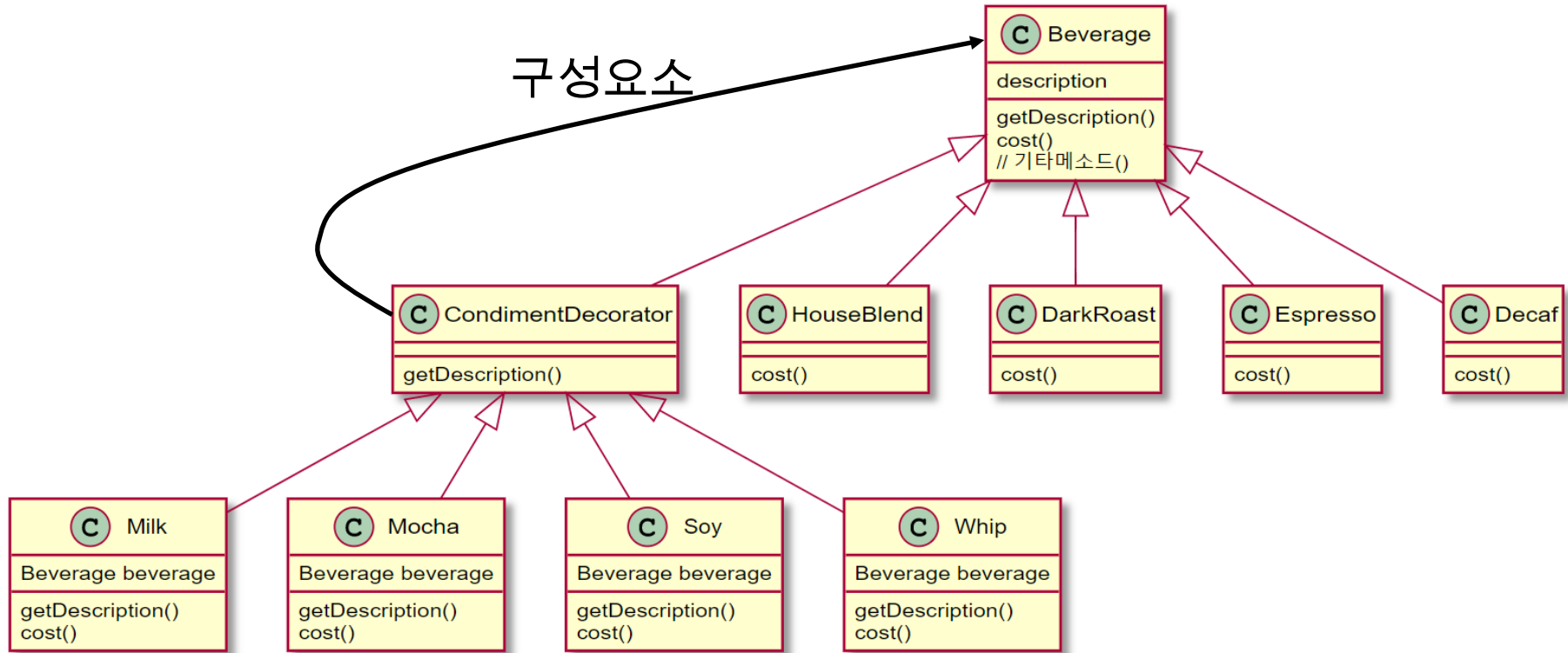


데코레이터 패턴

- 가격 계산을 할 때에는 가장 바깥쪽에 있는 데코레이터인 Whip 의 `cost()` 를 호출하면 됨
- Whip 에서는 그 객체가 장식하고 있는 객체에게 가격 계산을 위임함. 가격이 구해지고 나면, 구해진 가격에 휘핑 크림의 가격을 더한 다음 그 결과를 반환
 - Whip 에서 호출되는 Mocha 객체도 비슷하게 동작함



스타버즈 음료



- Beverage 는 구성요소를 나타내는 Component 추상 클래스 같은 개념으로 볼 수 있음

스타버즈 음료

```
public abstract class Beverage {
    String description = "제목 없음";

    public String getDescription() {
        return description;
    }
    public abstract double cost();
}

public abstract class CondimentDecorator
    extends Beverage {
    public abstract String getDescription();
}

public class Espresso extends Beverage {
    public Espresso() {
        description = "에스프레소";
    }
}
```

스타버즈 음료

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

public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "하우스 블렌드 커피 ";
    }
    public double cost() {
        return .89;
    }
}
```

스타버즈 음료

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

스타버즈 음료

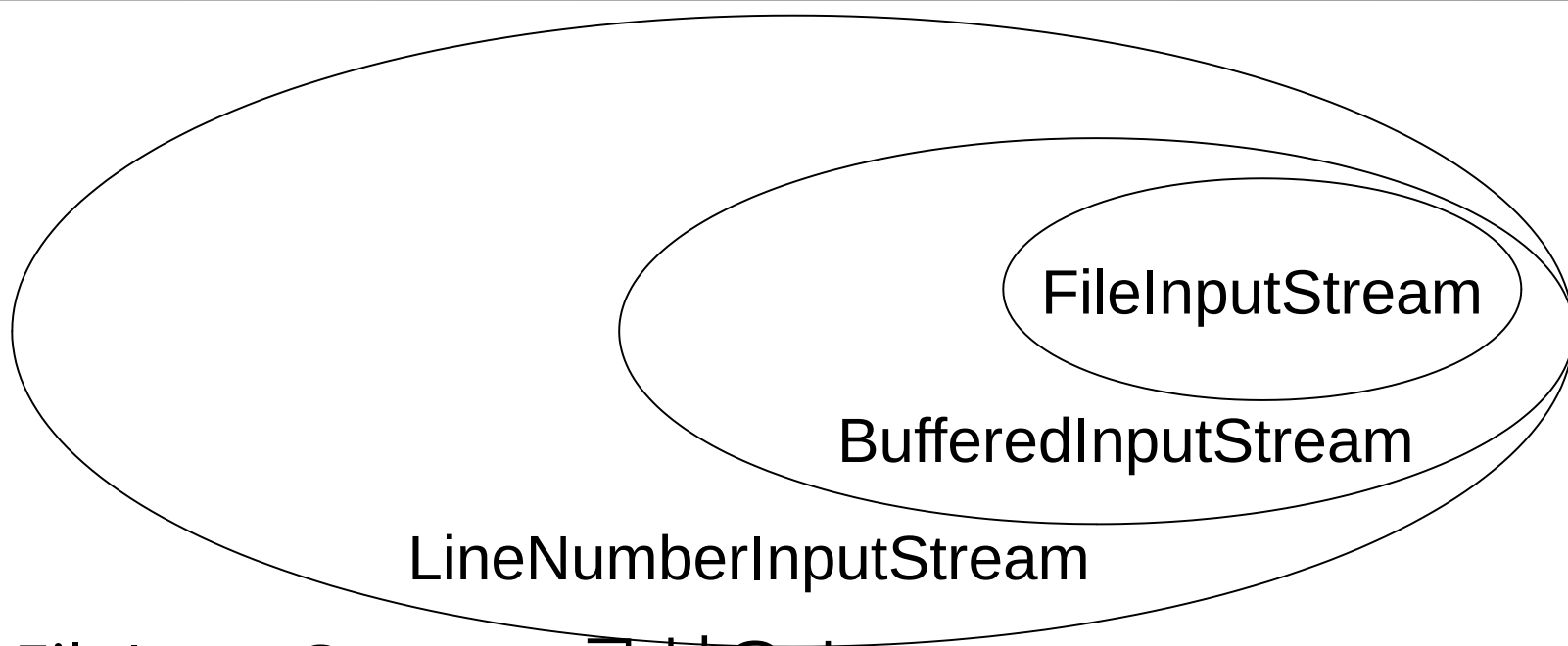
```
public class StarbuzzCoffee {  
    public static void main(String[] args[]) {  
        Beverage b = new Espresso();  
        System.out.println(b.getDescription()  
            + " $" + b.cost());  
        Beverage b2 = new DarkRoast();  
        b2 = new Mocha(b2);  
        b2 = new Mocha(b2); // 모카 한 개 더 추가  
        b2 = new Whip(b2);  
        System.out.println(b2.getDescription()  
            + " $" + b2.cost());  
        Beverage b3 = new HouseBlend();  
        b3 = new Soy(b3);  
        b3 = new Mocha(b3);  
        b3 = new Whip(b3);  
        System.out.println(b3.getDescription()  
            + " $" + b3.cost());  
    }  
}
```

사례 2: 자바 I/O

- 자바의 입출력은 io 패키지에서 처리하고, 다음 4개의 클래스를 중심으로 데코레이터 패턴을 사용
- 아래 클래스들은 데코레이터의 구상요소로 쓰이며, 추상 클래스라서 직접 사용할 수 없음

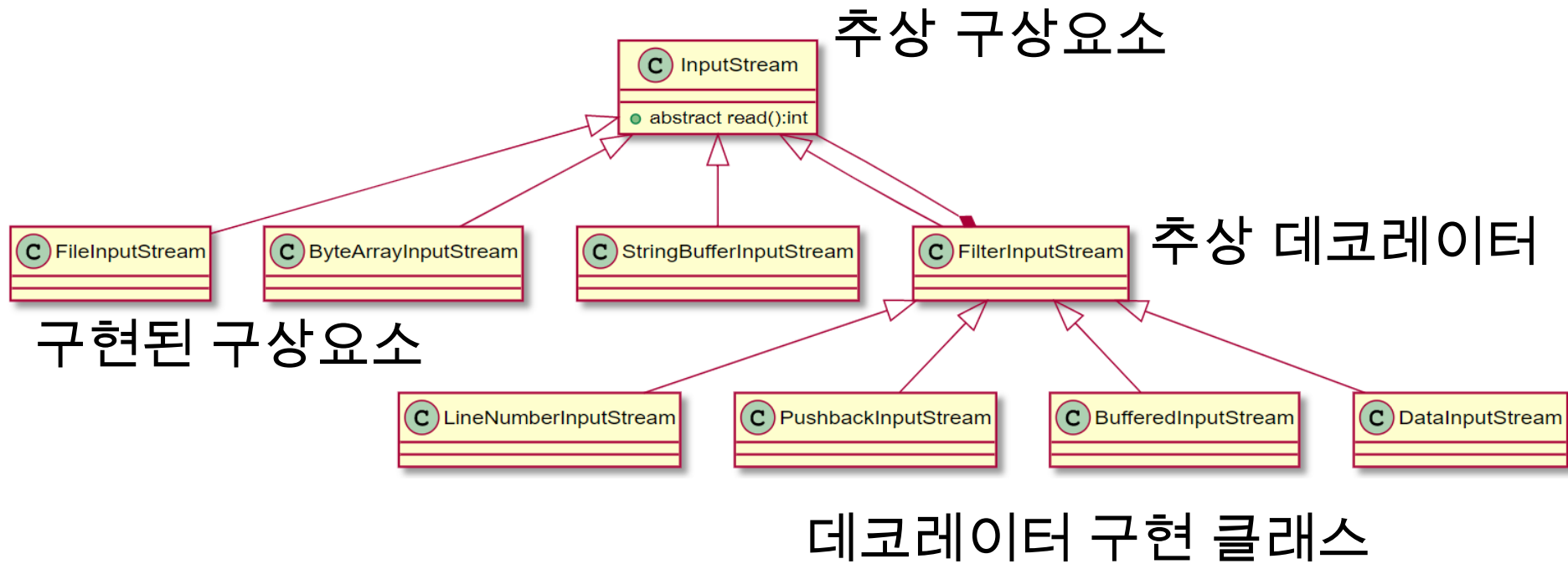
구분	입력	출력
바이트	InputStream	OutputStream
문자	Reader	Writer

사례 2: 자바 I/O



- FileInputStream 구성요소
- BufferedInputStream 데코레이터
 - 입력된 내용을 버퍼에 저장
 - 입력 내용을 한 줄씩 읽을 수 있게 readLine() 제공
- LineNumberInputStream 데코레이터
 - 줄 번호를 붙여줌

자바 I/O



파일에서 읽기

▣ 파일 입력 예제

```
import java.io.FileInputStream;

public class ReadFile {
    public static void main(String[] args) {
        try {
            FileInputStream readme
                = new FileInputStream("readme.txt");
            int b = readme.read();
            System.out.println("b = " + b);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


파일에서 읽기

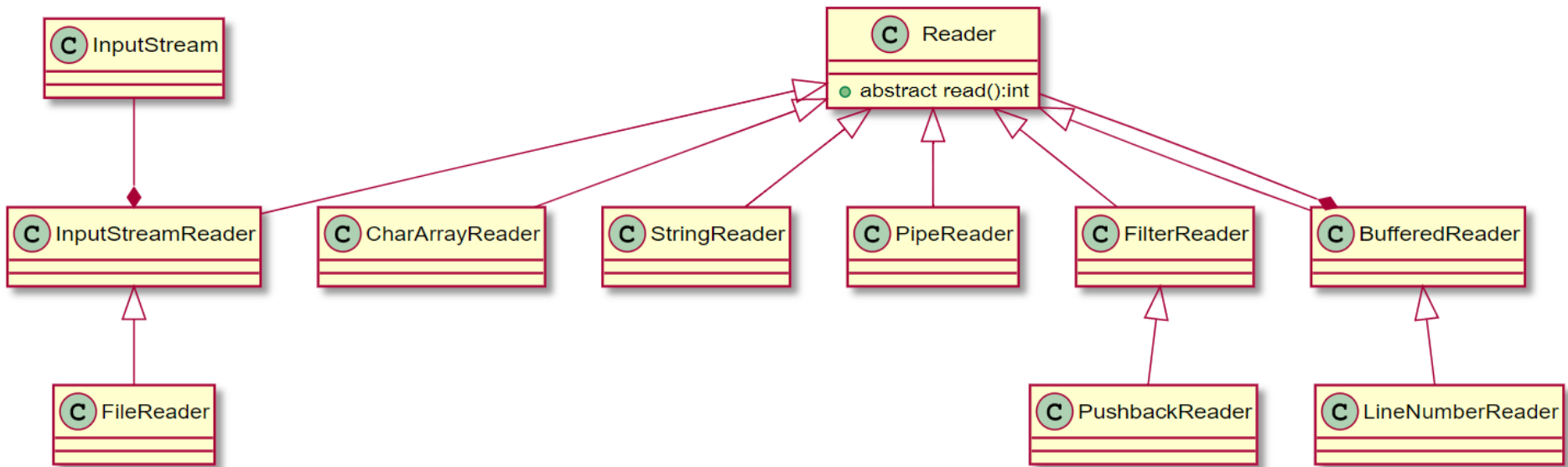
▣ 파일 입력 예제 – BufferedInputStream 사용

```
import java.io.FileInputStream;
import java.io.BufferedInputStream;

public class ReadFile {
    public static void main(String[] args) {
        try {
            BufferedInputStream readme
                = new BufferedInputStream(
                    new FileInputStream("readme.txt"));

            int b = readme.read();
            System.out.println("b = " + b);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

자바 I/O



파일에서 읽기

▣ 파일 입력 예제

```
import java.io.FileReader;

public class ReadFile {
    public static void main(String[] args) {
        try {
            FileReader readme
                = new FileReader("readme.txt");
            int b = readme.read();
            System.out.println("b = " + b);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

파일에서 읽기

▣ 파일 입력 예제 – BufferedReader 사용

```
import java.io.FileReader;
import java.io.BufferedReader;

public class ReadFile {
    public static void main(String[] args) {
        try {
            BufferedReader readme
                = new BufferedReader(
                    new FileReader("readme.txt"));
            String line = readme.readLine();
            System.out.println("line = " + line);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

파일에서 읽기

```
import java.io.FileReader;
import java.io.LineNumberReader;

public class ReadFile {
    public static void main(String[] args) {
        try {
            LineNumberReader readme
                = new LineNumberReader(
                    new FileReader("readme.txt"));
            String line = readme.readLine();
            System.out.println("line " +
                readme.getLineNumber() + " = " + line);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

소문자 데코레이터 (Lower Case Decorator)

- 입력 스트림에 있는 대문자를 전부 소문자로 바꿔주는 데코레이터

```
import java.io.FilterInputStream;

public class LowerCaseInputStream
    extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        return ((c == -1) ? c :
            Character.toLowerCase((char) c));
    }
}
```

소문자 데코레이터

```
public int read(byte[] b, int offset, int len)
    throws IOException {
    int result = super.read(b, offset, len);
    for (int i = offset; i < offset + result; i++) {
        b[i] = (byte)Character.toLowerCase((char)b[i]);
    }
}

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);
            }
        }
    }
}
```

소문자 데코레이터

```
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();
        }
    }
}
```


데코레이터 패턴

구성요소

각 데코레이터 안에는
Component 객체가 들어 있음
데코레이터 구성요소에 대한
레퍼런스가 들어있는
인스턴스 변수가 있음

