Decorator Pattern

514770-1
Fall 2020
10/6/2020
Kyoung Shin Park
Computer Engineering
Dankook University

Decorator Pattern

- "Attach **additional responsibilities** to an object **dynamically**. Decorators provide a flexible **alternative to subclassing for extending functionality**."
- This pattern creates a decorator class which **wraps the original class** and provides **additional functionality** keeping class methods signature intact.
- Also known as "**Wrapper**"
- Also this pattern is really useful and commonly faced java interview question on design patterns.
- All subclasses of java.io.InputStream, OutputStream, Reader and Writer have constructors that accept objects of their own type.

Problem

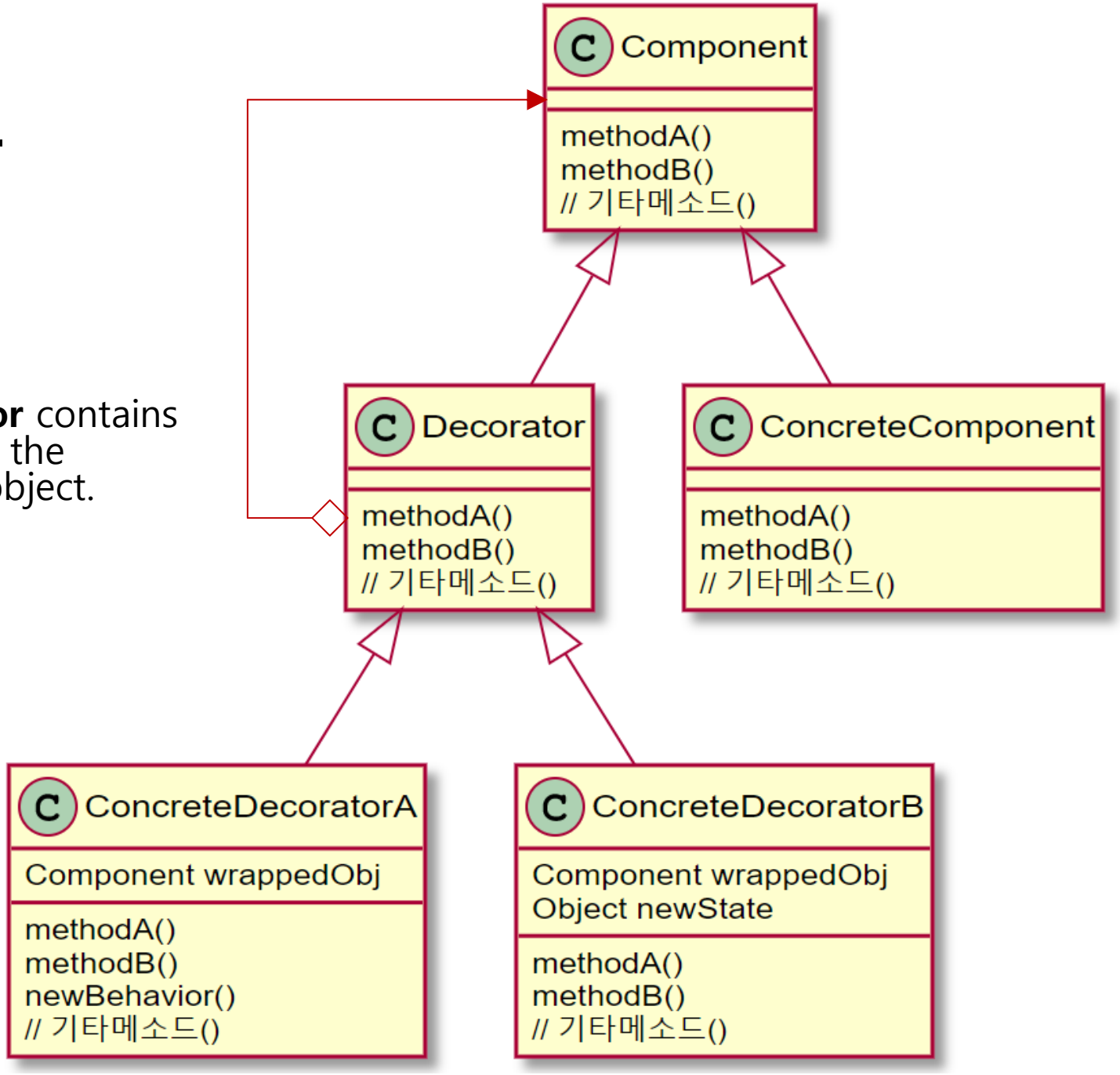
- Suppose you want to add additional features or behaviors to an existing object.
  - **Inheritance is not feasible** because it is static and applies to an entire class.
  - In fact, it has been shown that extending objects using inheritance often results in an **exploding class hierarchy**.
  - Can you add new features without using inheritance?

Decorator Pattern

	Description
Pattern	Decorator
Problem	There are various kinds that are slightly different. The more kinds increase, the more difficult to expand.
Solution	Add object functionality using composition rather than inheritance. Extension at runtime (not at compile time)
Result	Open Closed Principle, Extendibility

- The decorator pattern allows a user to add new functionality to an existing object without altering its structure.

Each **decorator** contains a reference to the **Component** object.



Define Decorator Pattern

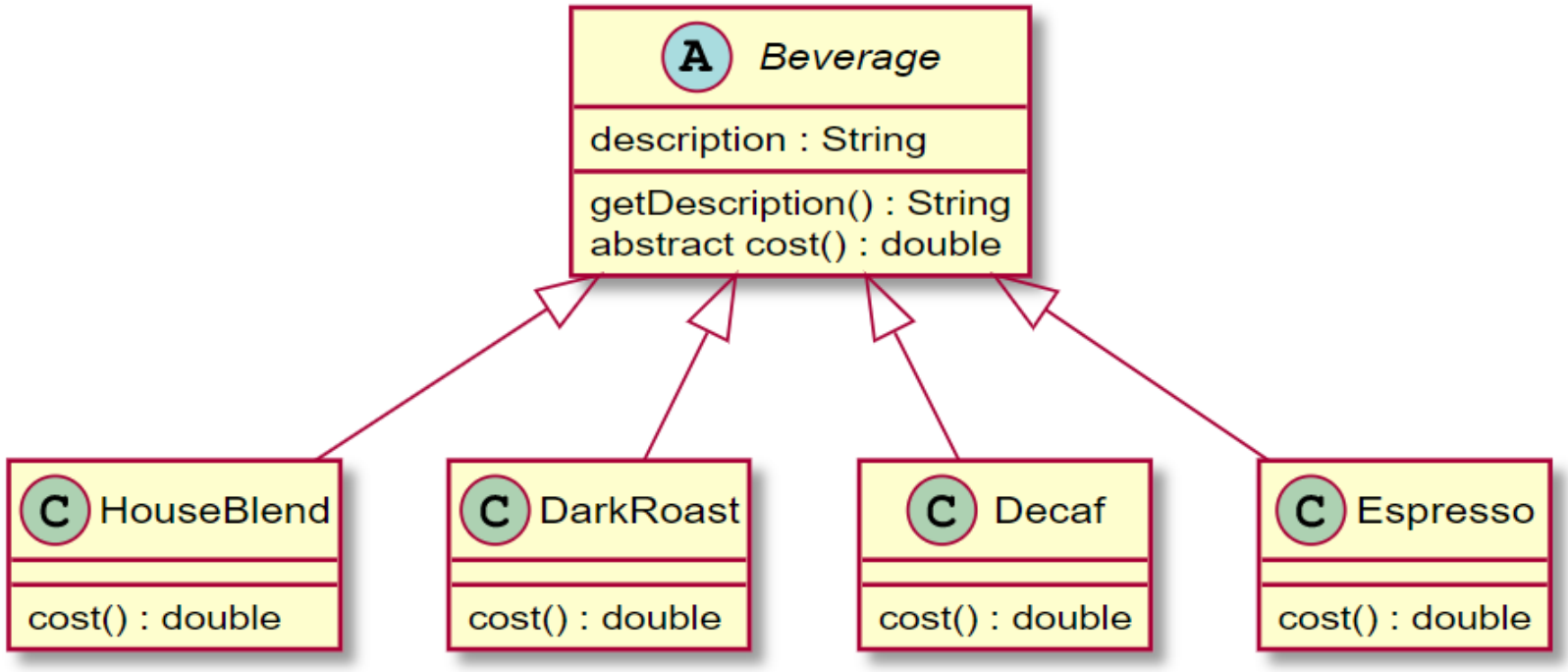
- Component
  - This is the wrapper which can have additional responsibilities associated with it at runtime
- ConcreteComponent
  - This is the original object to which the additional responsibilities are added in program.
- Decorator
  - This is an abstract class which contains a reference to the component object and also implements the component interface.
- ConcreteDecorator
  - They extend the decorator and build additional functionality on top of the Component class.
- Decorator can extend the state of Component.

Decorator Pattern

- Decorator pattern is designed in a way that **multiple decorators** can be stacked on top of each other, each **adding new functionality**.
- In contrast to inheritance, a decorator can operate on any implementation of a given interface, which eliminates the need to subclass an entire class hierarchy.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- **In the decorator pattern, inheritance is not to inherit a behavior, but to conform to the form.**
- **Component can be an abstract class or interface.**

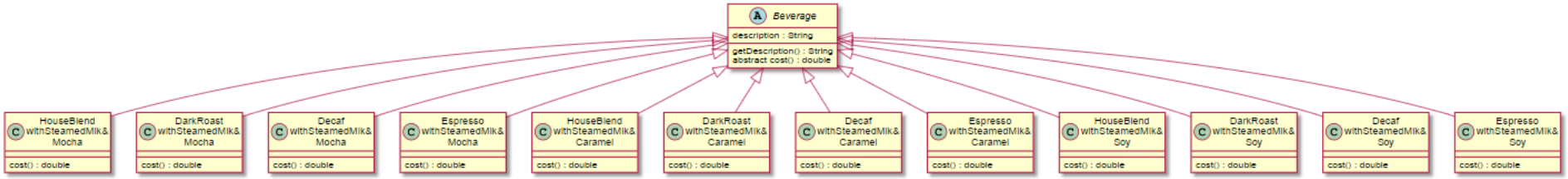
Starbuzz Coffee (HFDP Ch. 3)

- Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around.
  - Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.
  - When they first went into business, they designed their classes like this..



Starbuzz Coffee (HFDP Ch. 3)

- In addition to your coffee, you can also ask for several **condiments** like steamed milk, soy, mocha(chocolate), and whipped cream. Starbuzz **charges** a bit for each of these, so they really need to get them built into their order system. Here's their first attempt.....

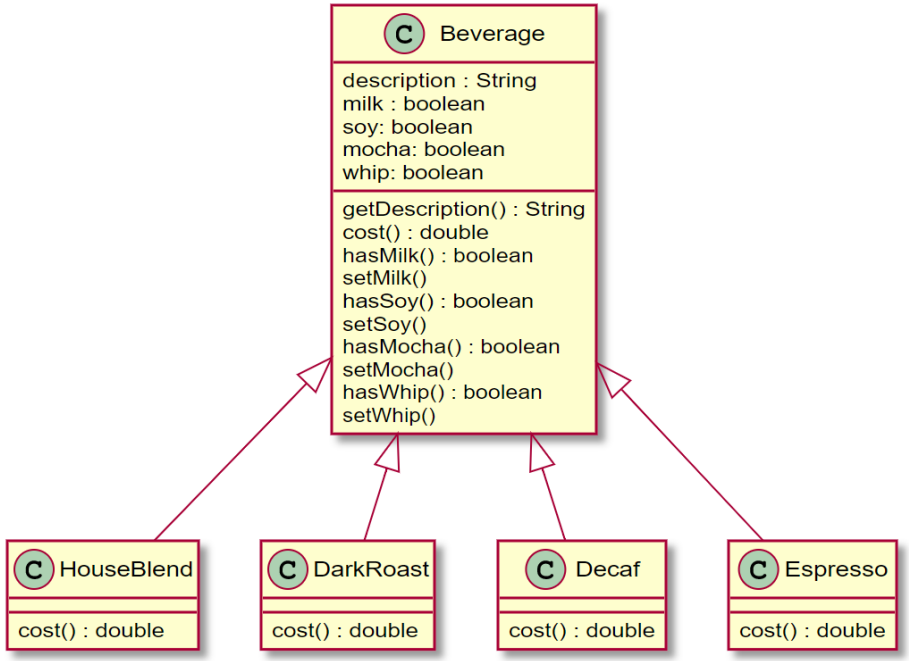


Class explosion

- Problems with maintenance
  - What if a new topping is added?
  - What if the prices of existing ingredients (milk, cream, etc) increase?

Starbuzz Coffee (HFDP Ch. 3)

- **Beverage base class** add **instance variables** to represent whether or not each beverage has milk, mocha, whip…
  - **The superclass cost() will calculate the costs for all of the condiments.**
  - The overridden cost() will extend that functionality to include costs for that specific beverage type.

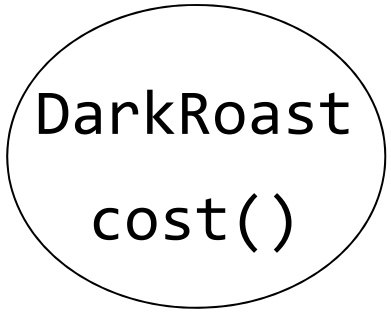


Starbuzz Coffee (HFDP Ch. 3)

- We've seen that representing our beverage plus condiment pricing with inheritance has not worked out very well – class explosions, rigid designs, or we add functionality to the base class that isn't appropriate for some of the subclasses.
- We'll start with a beverage and "decorate" it with the condiments at runtime.
- If a customer wants a **DarkRoast** with **Mocha** and **Whip**
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Decorate it with a Whip object
  - Call the cost() method and rely on delegate to add on the condiment costs
    - Mocha, Whip are decorator objects as "wrappers".

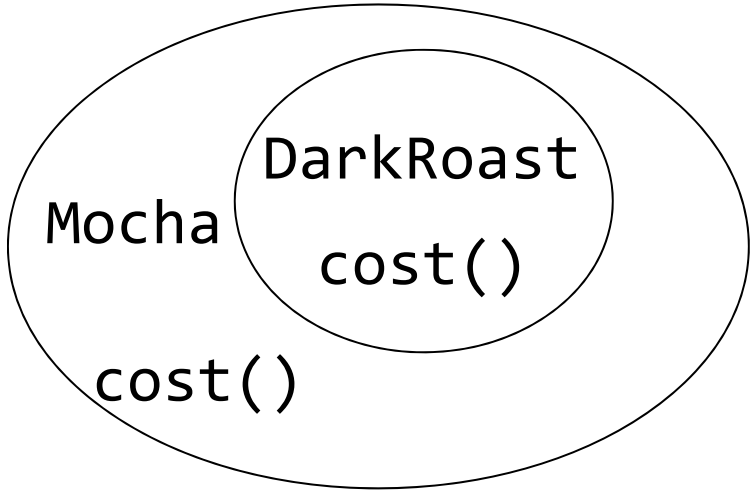
Starbuzz Coffee (HFDP Ch. 3)

- Constructing a drink order with **Decorators**
  - Start with **DarkRoast** object
    - DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.



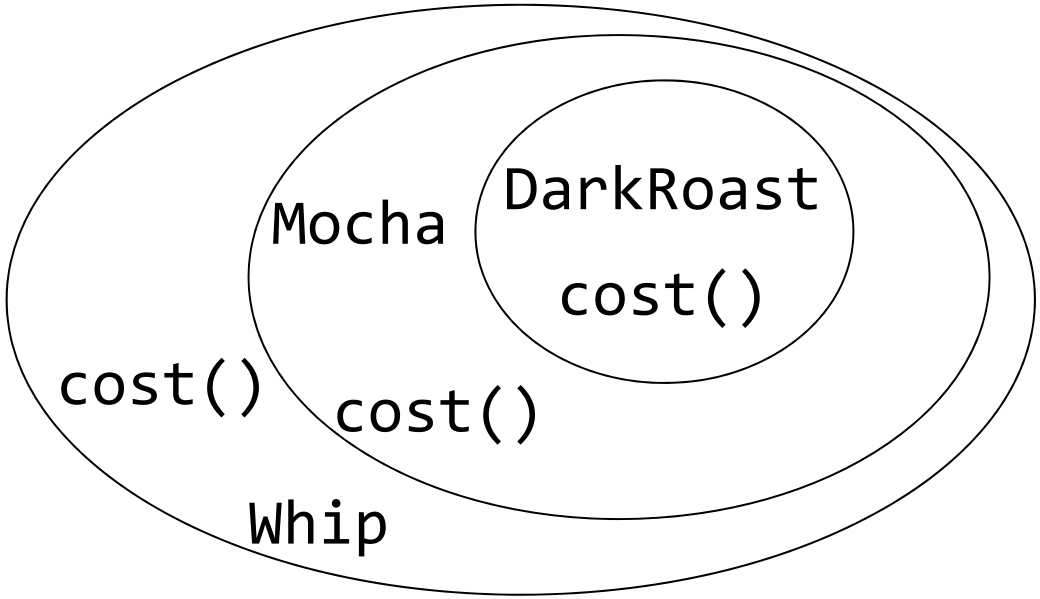
Starbuzz Coffee (HFDP Ch. 3)

- The customer wants Mocha, so we create a **Mocha** object and **wrap it around** the DarkRoast.
  - The **Mocha** object is a decorator. Its type "*mirrors*" (mean it is the same type) the object it is decorating, in this case, a Beverage.
  - So, Mocha has a cost() method, and through polymorphism we can treat any **Beverage wrapped in Mocha** as a Beverage, too (because Mocha is a subtype of Beverage).



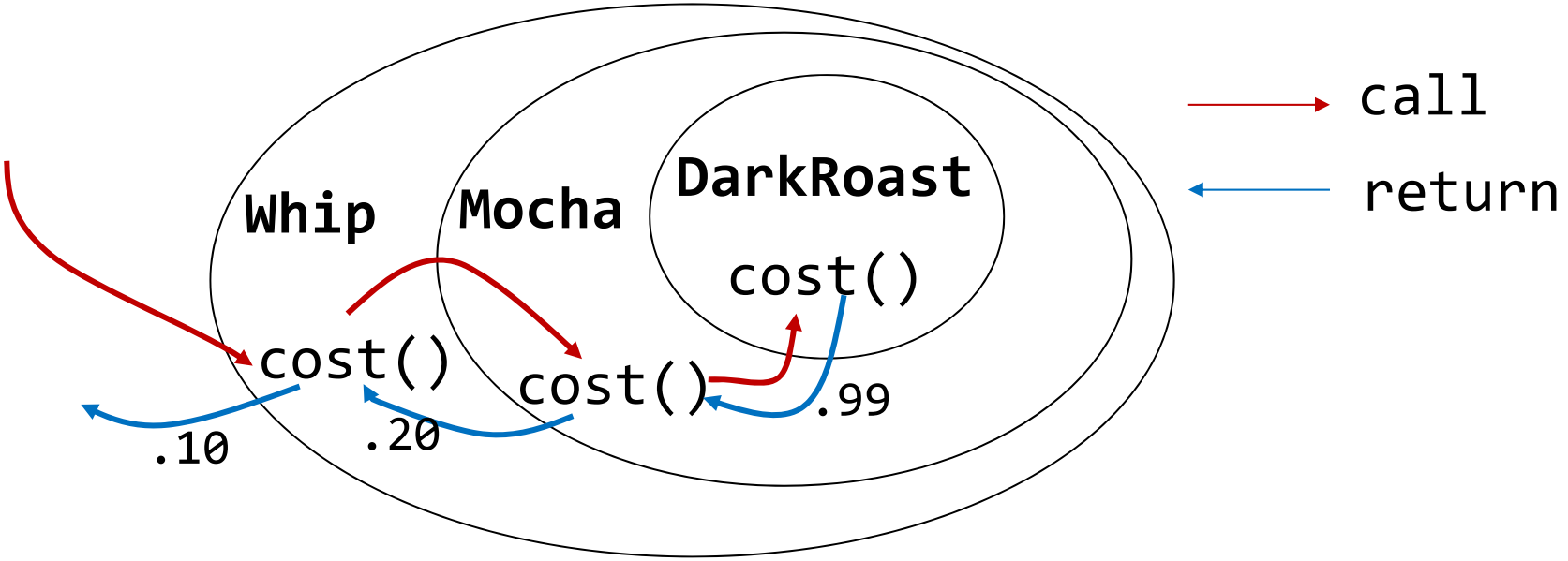
Starbuzz Coffee (HFDP Ch. 3)

- The customer also wants Whip, so we create a **Whip** object and **wrap Mocha with it**.
  - **Whip** is a decorator, so it also *mirrors* DarkRoast's type (Whip is a subtype of Beverage).
  - So, a **DarkRoast wrapped in Mocha and Whip** is still a Beverage, and we can do anything with it we can do with a DarkRoast, including call its cost() method.

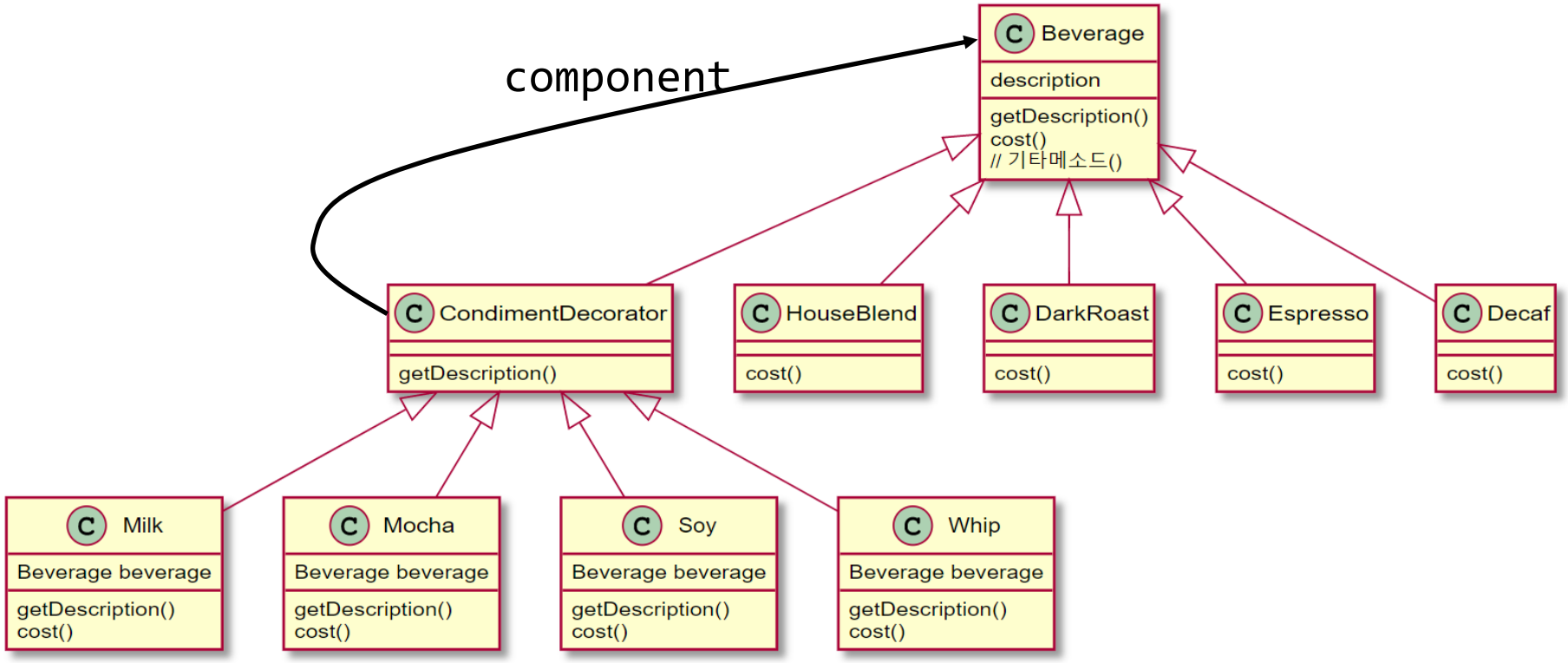


Starbuzz Coffee (HFDP Ch. 3)

- It's time to compute the cost for the customer.
  - We do this by calling **cost()** on the **outermost decorator, Whip.**
  - Whip is going to delegate computing the cost to the objects it decorates. – Whip calls cost() on Mocha. Mocha calls cost() on DarkRoast. DarkRoast returns its cost, 99 cents. Mocha adds its cost, 20 cents, and returns the new total, $1.19.
  - Once it gets a cost, it will add on the cost of the Whip. - Whip adds its cost 10 cents, and returns the final total, $1.29.



Starbuzz Coffee (HFDP Ch. 3)



- Beverage acts as our abstract Component class.

Starbuzz Coffee (HFDP Ch. 3)

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

    public String getDescription() {
        return description;
    }
    // the different types of beverages
    // will have different cost
    public abstract double cost();
}

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

Starbuzz Coffee (HFDP Ch. 3)

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

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

Starbuzz Coffee (HFDP Ch. 3)

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

Starbuzz Coffee (HFDP Ch. 3)

```
public class Whip extends CondimentDecorator {
    Beverage beverage;
    public Whip(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + "+ Whip
";
    }
    public double cost() {
        return beverage.cost() + .10;
    }
}
```

Starbuzz Coffee (HFDP Ch. 3)

```
public class StarbuzzCoffee {
    public static void main(String[] args[]) {
        Beverage b = new Espresso();
        System.out.println(b.getDescription()
          + " $" + b.cost());
        b = new DarkRoast();
        b = new Mocha(b);
        b = new Mocha(b); // add second mocha
        b = new Whip(b);
        System.out.println(b.getDescription()
          + " $" + b.cost());
        b = new HouseBlend();
        b = new Soy(b);
        b = new Mocha(b);
        b = new Whip(b);
        System.out.println(b.getDescription()
          + " $" + b.cost());
    }
}
```

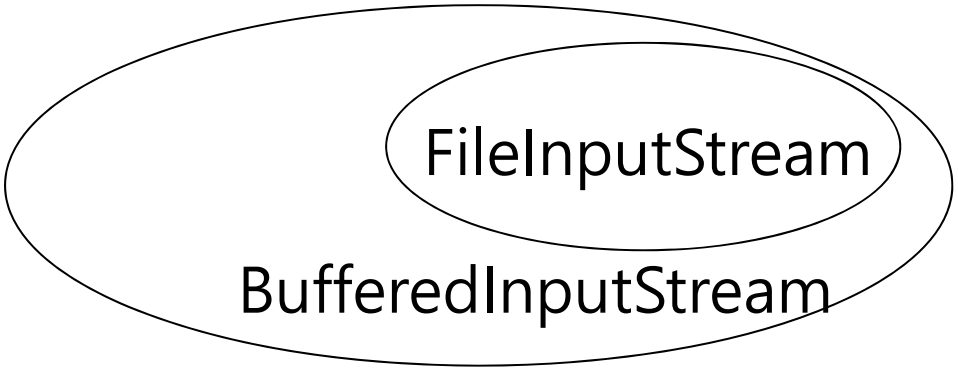
Java I/O

- Java I/O is handled by the io package, and the decorator pattern is used around the following four classes.
- The following classes are used as the **decorator** and cannot be used directly because they are abstract classes.

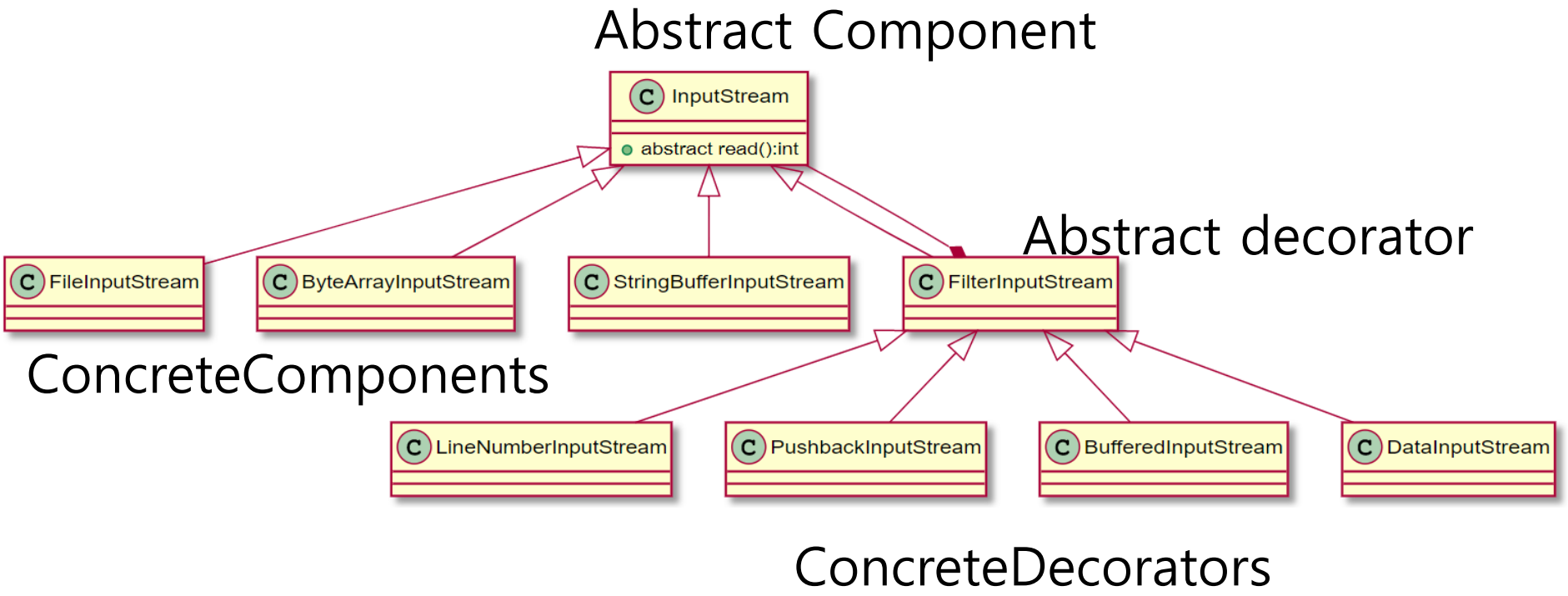
	Input	Output
Byte	InputStream	OutputStream
Text	Reader	Writer

Java I/O

- Component - FileInputStream
- Decorator - BufferedInputStream
  - Read bytes of data into the buffer
  - Provide the readLine() method to read input line by line
- Decorator - LineNumberInputStream
  - Provide a extra functionality of keeping track of the current line number.



Java I/O


Abstract Component
InputStream
abstract read():int
Abstract decorator
FileInputStream
ByteArrayInputStream
StringBufferInputStream
FilterInputStream
ConcreteComponents
LineNumberInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
ConcreteDecorators

Java I/O – File I/O

- FileInputStream

```
import java.io.FileInputStream;

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

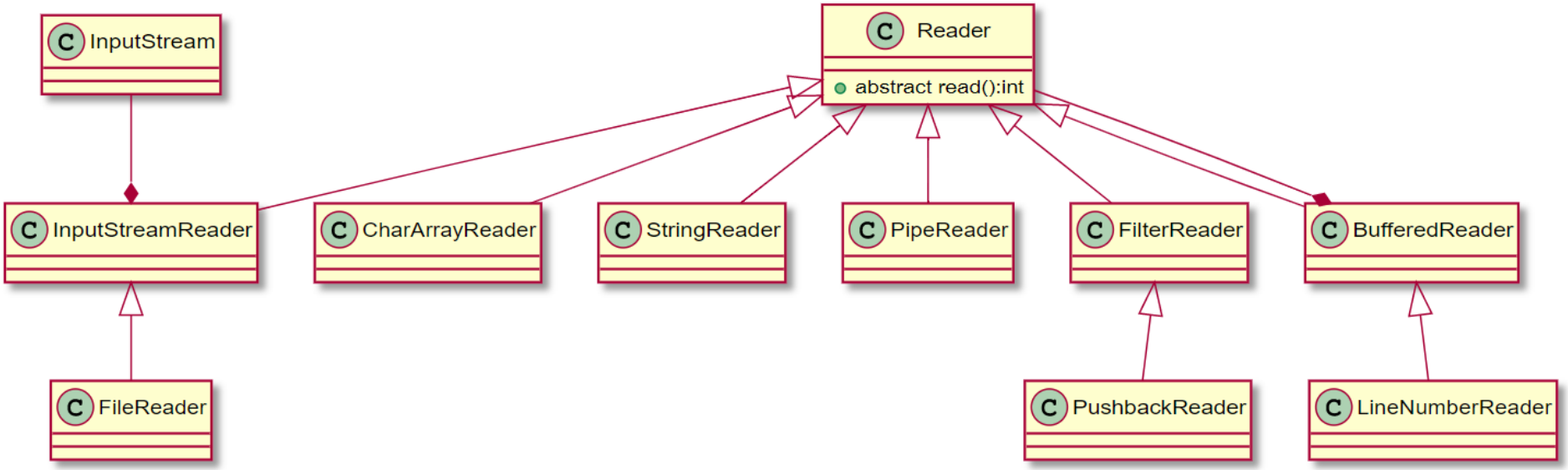
Java I/O – File I/O

- BufferedInputStream

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

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

Java I/O – File I/O



Java I/O – File I/O

- FileReader

```
import java.io.FileReader;

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

Java I/O – File I/O

- BufferedReader

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

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

Java I/O – File I/O

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

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

Lower Case Decorator (HFDP Ch. 3)

- LowerCaseInputStream decorator converts all uppercase letters in the input stream to lowercase letters.

```
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));
  }
```

Lower Case Decorator (HFDP Ch. 3)

```
  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]);
    }
  }
} // end of LowerCaseInputStream
```

Lower Case Decorator (HFDP Ch. 3)

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