Decorator Pattern

Problem

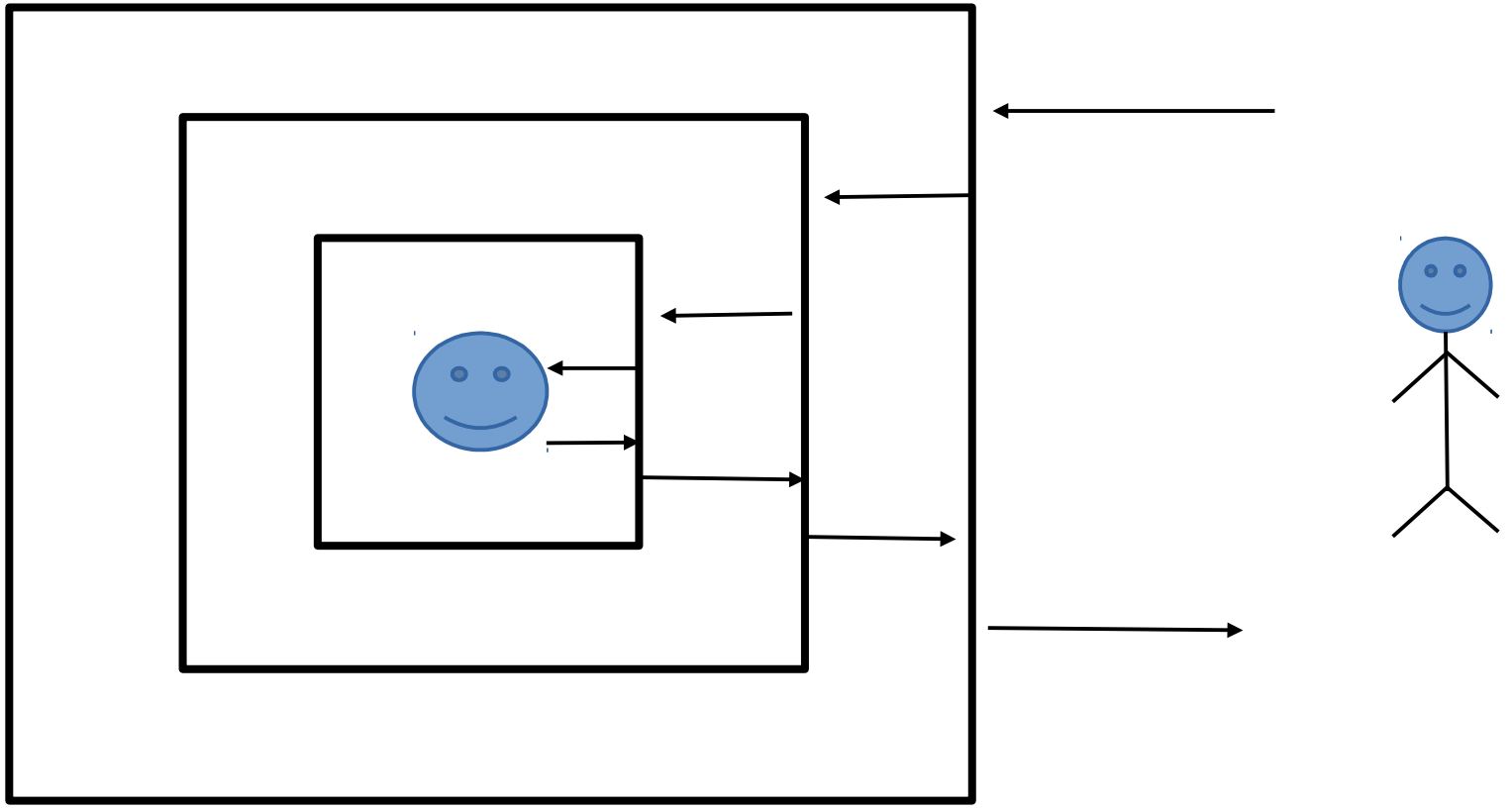
- We want to attach (or add or plugin) **additional responsibilities to individual objects**, but do not add them to an entire class of objects.

- We want to be able to add or remove responsibilities to the object at **run time**.

- If we use **subclassing** (inheritance) we will produce **huge number of classes** and we can not add new responsibilities at run time.

Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



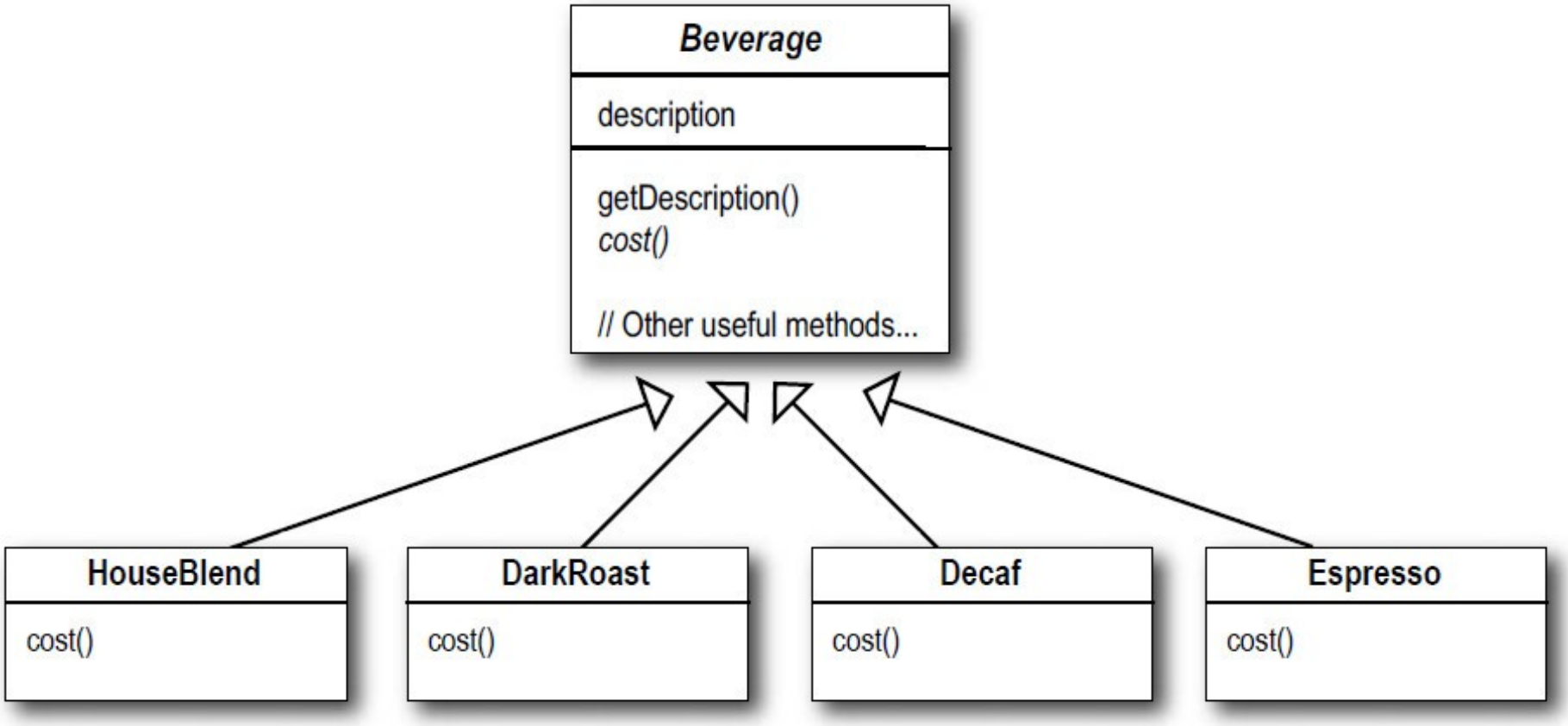
Combination of **isA** and **hasA**

Example: Coffee Machine

- We want to calculate the costs for each beverage based on the adding of condiments like milk or sugar
- We have
  - lots of **different types** of beverages
  - lots of **different condiments**
- We want to **add or remove condiments.**
- We want to **calculate the final cost** of the beverage.

Problem

- You can have a superclass and then extend it in lots of different types.



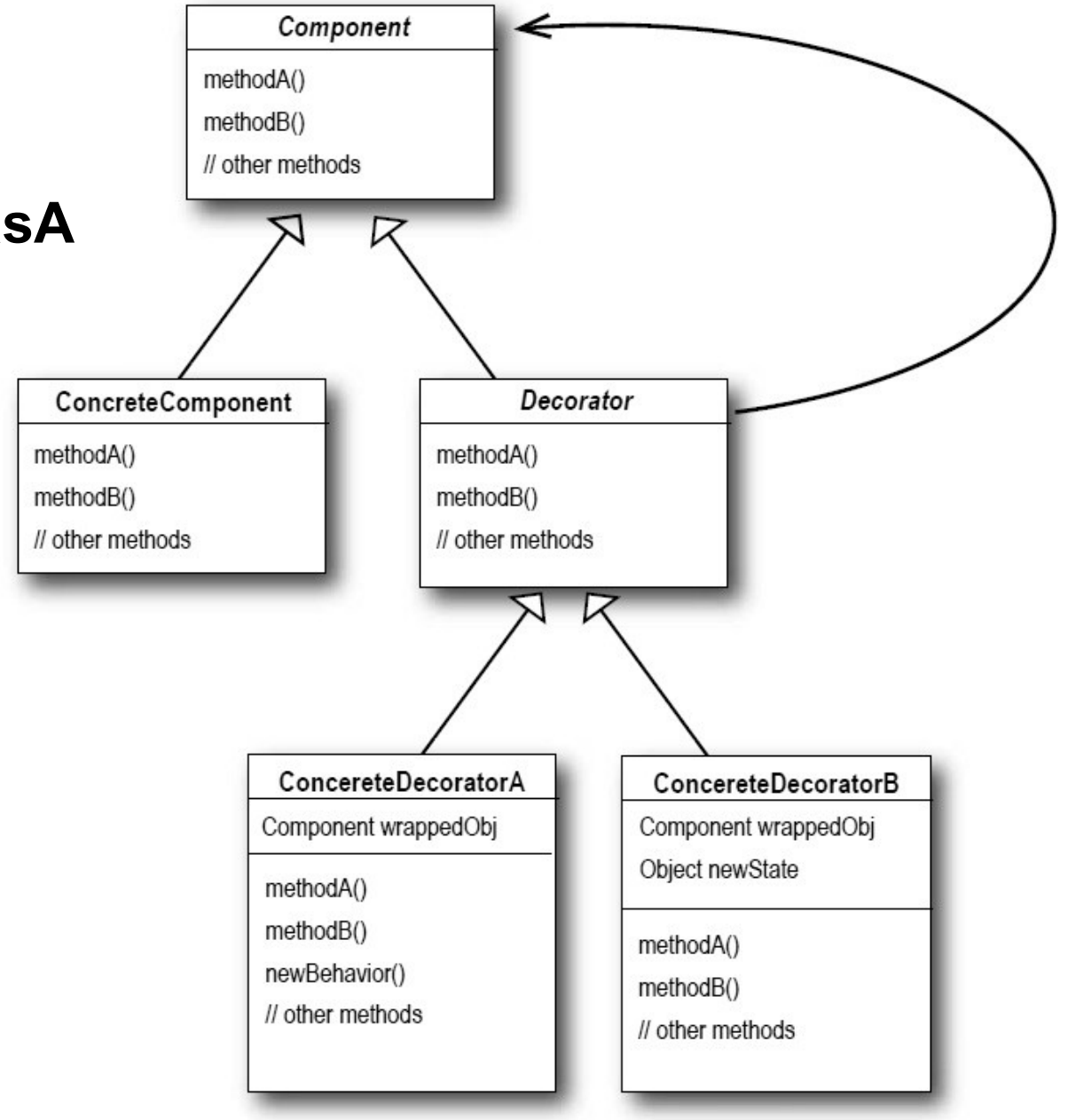
Example Source: HeadFirst Book

What are our choices?

- Passing references to a the cost method
- Iterate over the collections (Iterator Pattern) – will come later in class

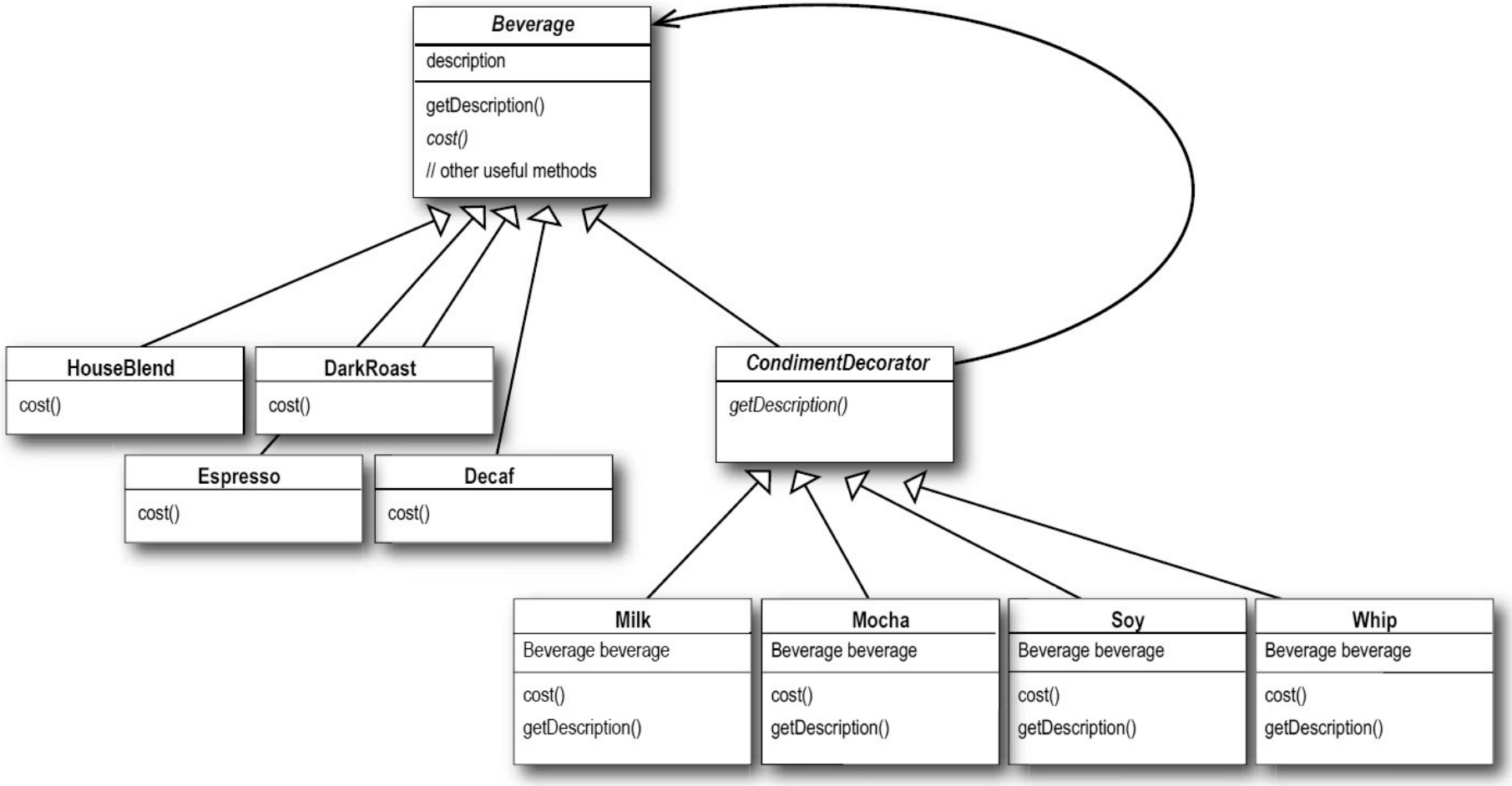
Solution

- Decorator Pattern
- **Combine isA with hasA**



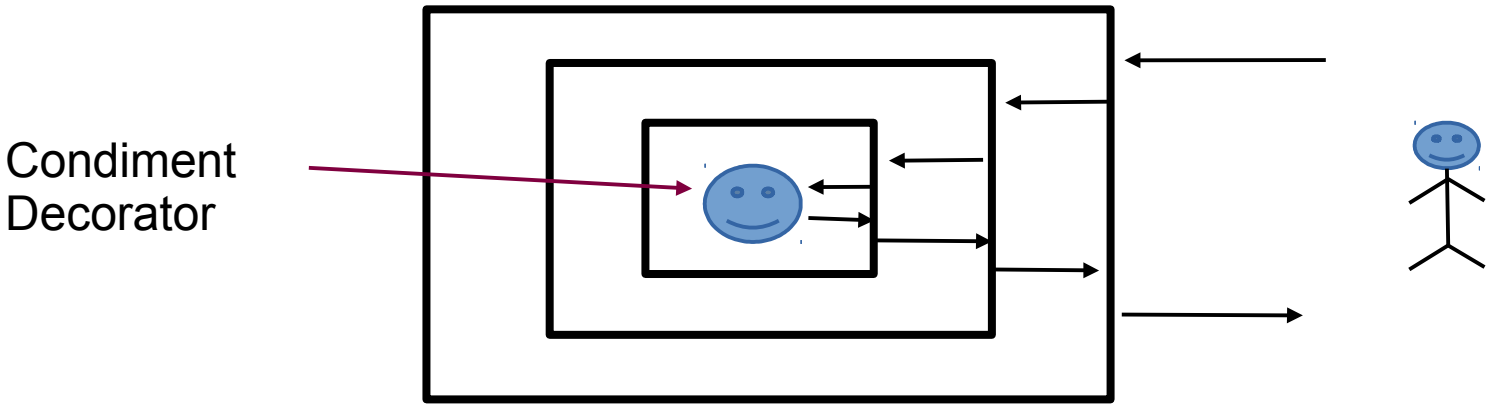
Solution

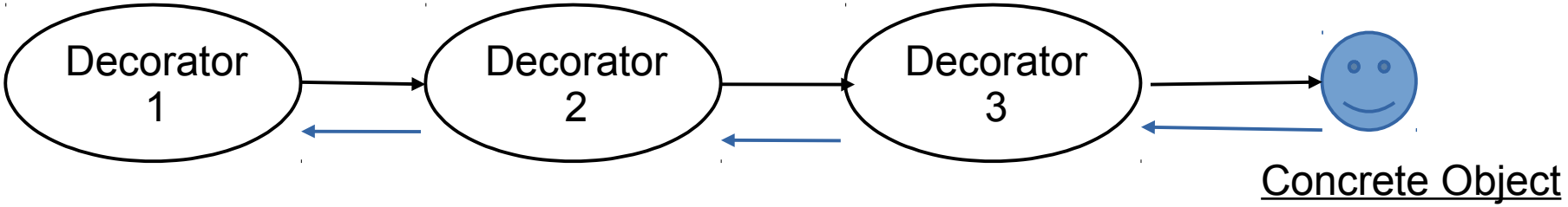
- Add a wrapper class that **isA** Beverage and **hasA** Beverage
- It separates the code and delegate the responsibilities



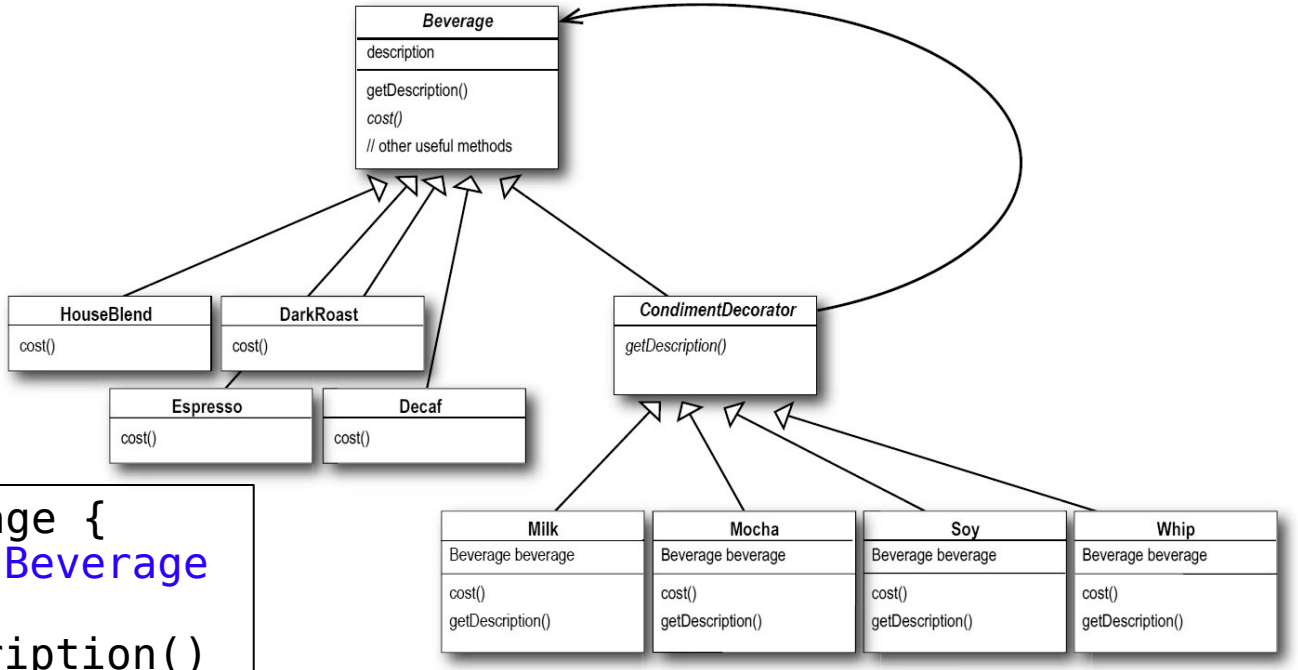
Decorator Pattern

- The Decorator Pattern attaches ***additional responsibilities* to an object dynamically**. Decorators provide a flexible alternative to subclassing for extending functionality.





Implementation of Use Case – 1




```
public abstract class Beverage {
	String description = "Beverage
";
	public String getDescription()
{
		return description;
	}
	public abstract double cost();
}
```

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

Implementation of Use Case – 1

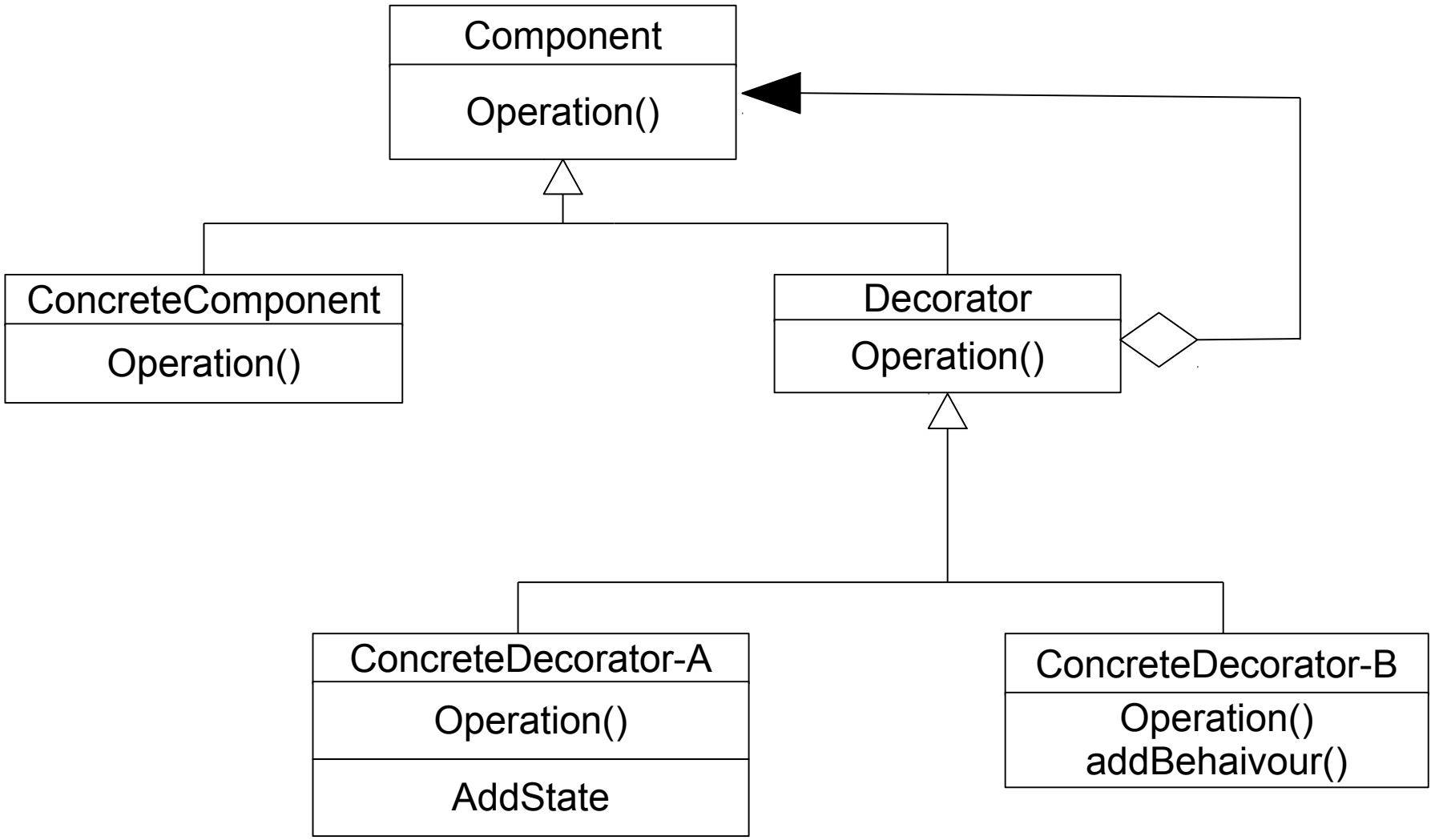
```
public abstract class CondimentDecorator extends Beverage
{
     public abstract String getDescription();
}
```

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

Implementation of Use Case – 1

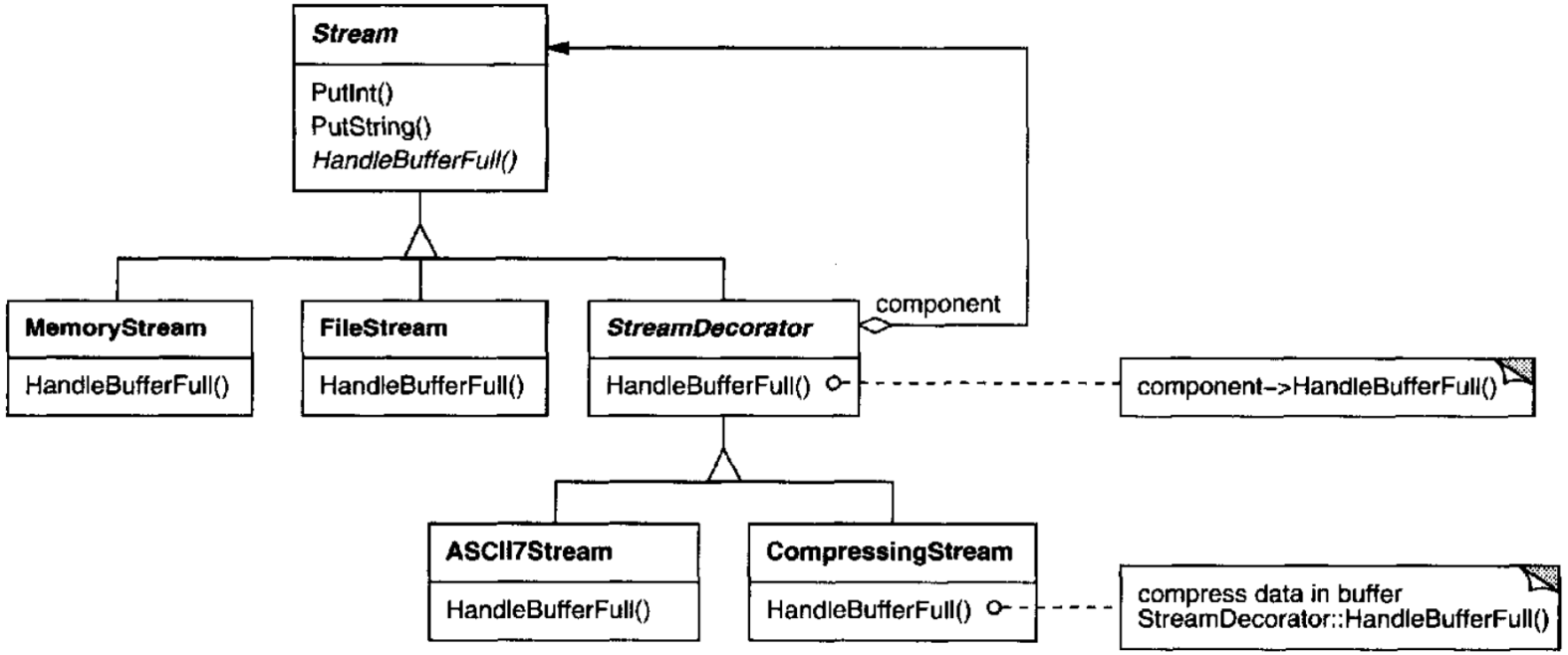
```
public class CoffeeMakerMain {

      public static void main(String args[]) {

            Beverage beverage = new Espresso();

            System.out.println(beverage.getDescription() + " $" +
beverage.cost());

            Beverage beverage2 = new MediumRoast();
            beverage2 = new Milk(beverage2);
            beverage2 = new Milk(beverage2);
            beverage2 = new Suger(beverage2);

            System.out.println(beverage2.getDescription() + " $" +
beverage2.cost());

            Beverage beverage3 = new HouseBlend();
            beverage3 = new Soy(beverage3);
            beverage3 = new Mocha(beverage3);
            beverage3 = new Whip(beverage3);

            System.out.println(beverage3.getDescription() + " $" +
beverage3.cost());
      }
}
```

Structure of Decorator pattern



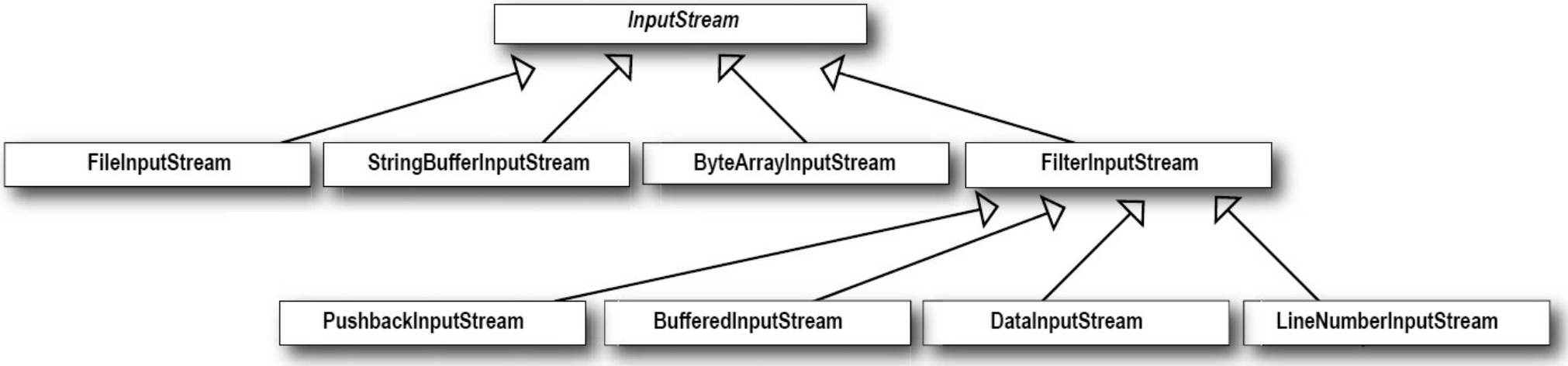
Another Example

- Streaming in programing languages like C++ or Java
- A MemoryStream **isA** Stream
- A CompressingStream **isA** Stream and **hasA** Stream



Example Source: GangOfFour Book

Input / Output Stream in Java

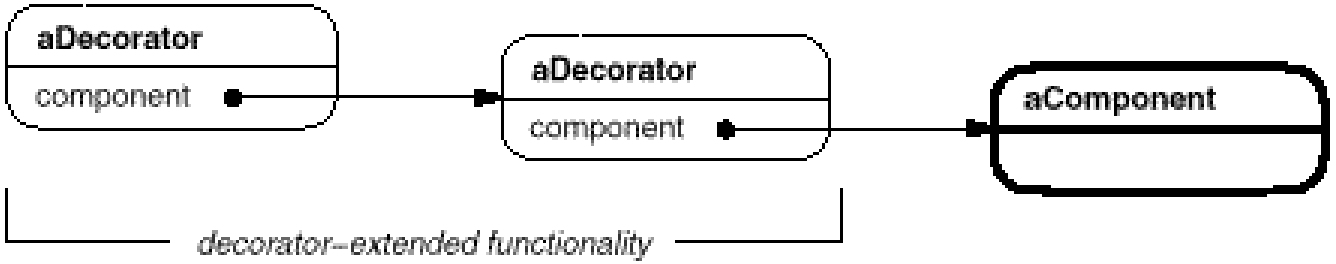


You might have seen codes like this before:

```
FileInputStream fis = new FileInputStream("/objects.gz");
// We want speed, so let's buffer it in memory:
BufferedInputStream bis = new BufferedInputStream(fis);
// The file is gzipped, so we need to ungzip it:
GzipInputStream gis = new GzipInputStream(bis);
// We need to unserialize those Java objects
ObjectInputStream ois = new ObjectInputStream(gis);
```

Strategy pattern vs. Decorator pattern

- Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators. Decorators are transparent to the component.



- With strategies, the component knows about possible extensions and it has to reference and maintain the corresponding strategies.

When use the Decorator Pattern

Use Decorator pattern when:

- **to add responsibilities** to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by **sub-classing is impractical**.

  A large number of independent extensions are. possible, but would produce an explosion of subclasses to support every combination.

Consequences

+ More flexibility than static inheritance. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.

+ Avoids feature-laden classes high up in the hierarchy.

Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of supporting all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Consequences

- A decorator and its component aren't identical.

A decorator acts as a transparent enclosure, but it has a different object identity, and is not identical to the component itself.

You shouldn't rely on object identity when you use decorators.

- Lots of little objects.

It generate lots of little objects that all look similar and differ only in the way they are interconnected, not in their class or in the value of their variables.

Such systems are easy to customize by those who understand them, they can be hard to learn and debug.

Summary

- Decorator pattern decorates your classes at runtime using a form of object composition so that **additional responsibilities can be added or removed at run-time**.

- The decorator pattern let us create a chain of objects that starts with the decorator (the objects the objects responsible for the new function) and ends with the original object. In this way, new **responsibilities can be added or removed to the original object by wrapping inside decorating objects**.

Example – 3 : Implement This

- **E-Mail Collector System**

  In a company, we have a email sender application that collects all of the company emails, and before they go out checks the emails, adds text, … (does somethings with them) and then sends them out.

  The email sender applications receives the **email text and header,** and should be able to

  - **add a text** to the end of each email (a **disclaimer**)
  - **virus inspect** the emails
  - **send a bcc** to the manager

- We should be able to add or remove some of these behaviors or responsibilities at run time