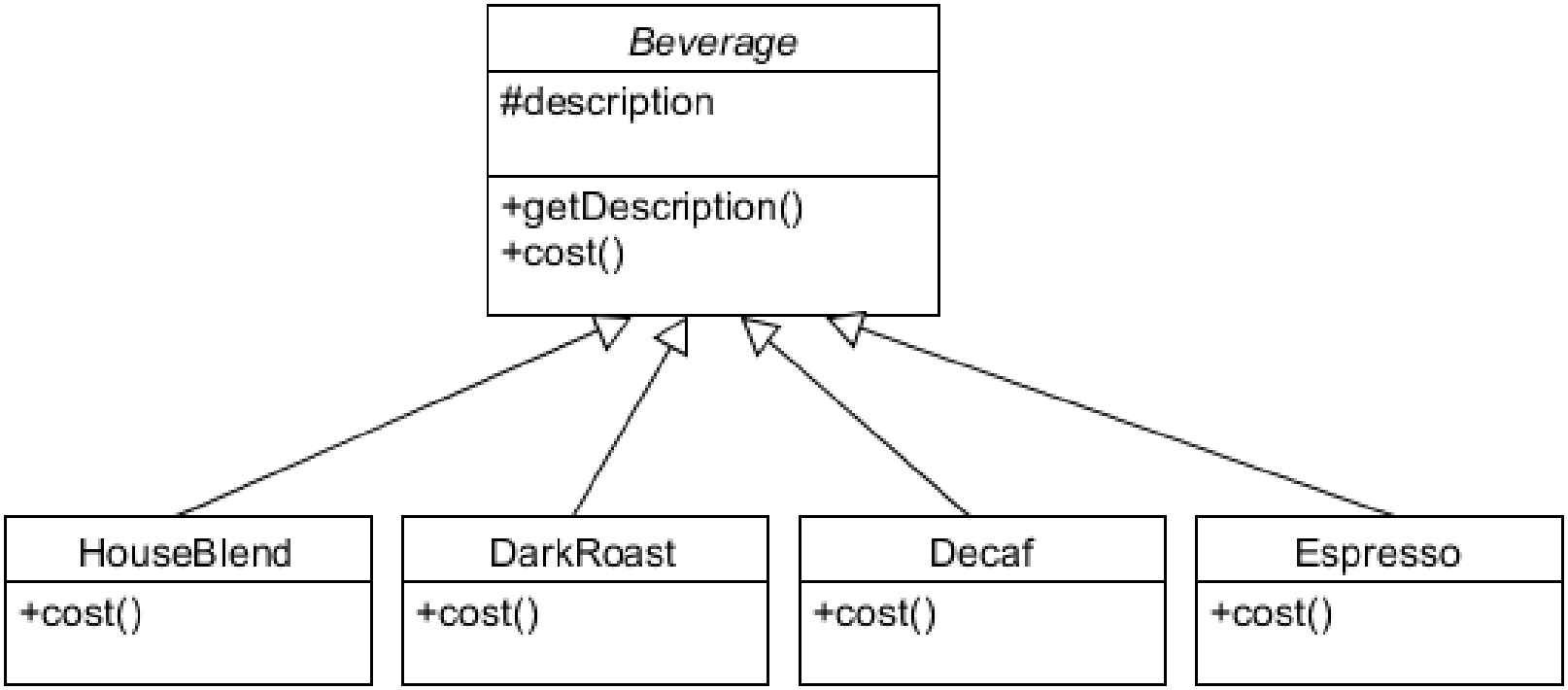
Decorator Pattern

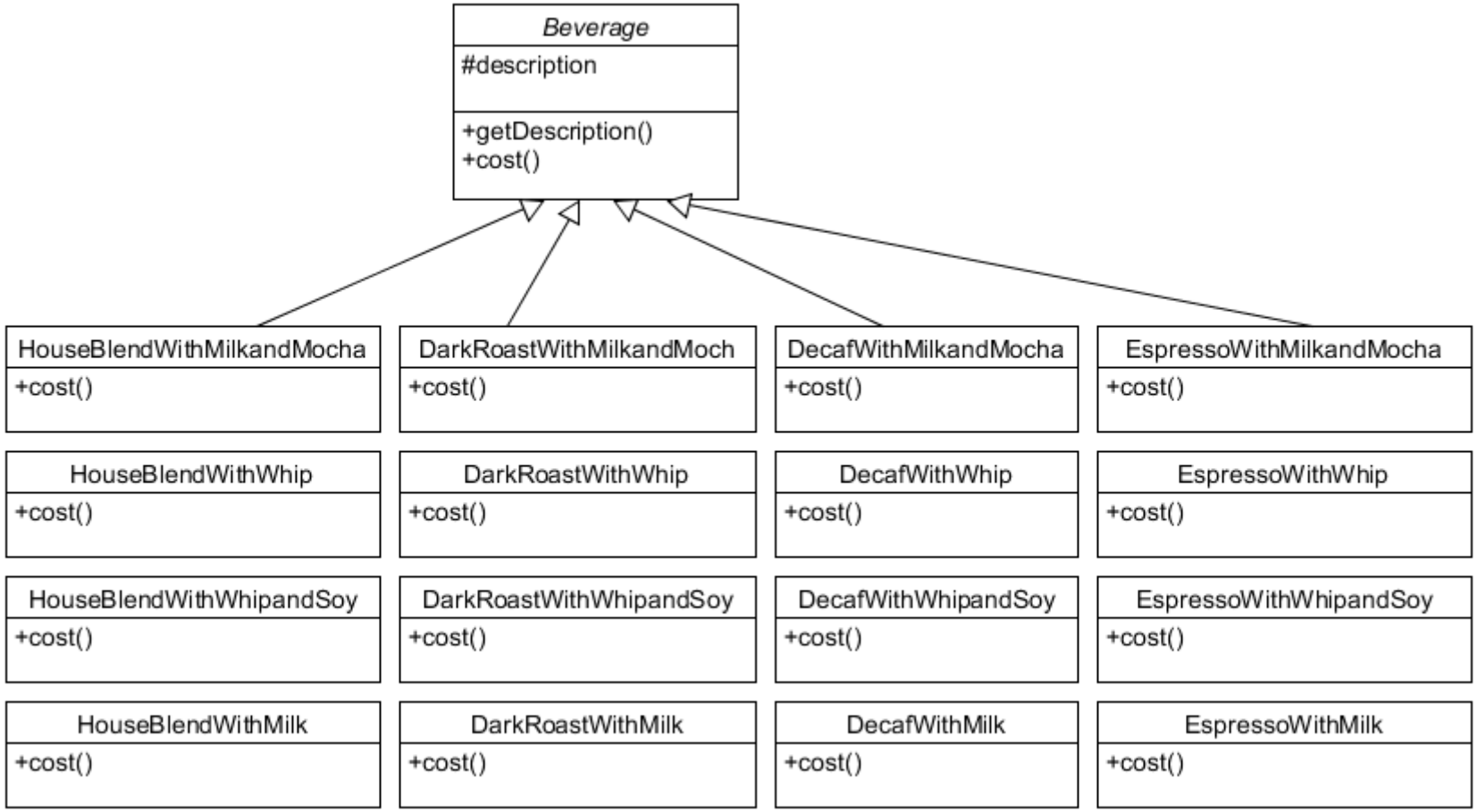
Contents

- Intro. to the problem
- Discovery of Decorator pattern
- Structure and Run-time mechanism of Decorator pattern
- Decorator in Java I/O

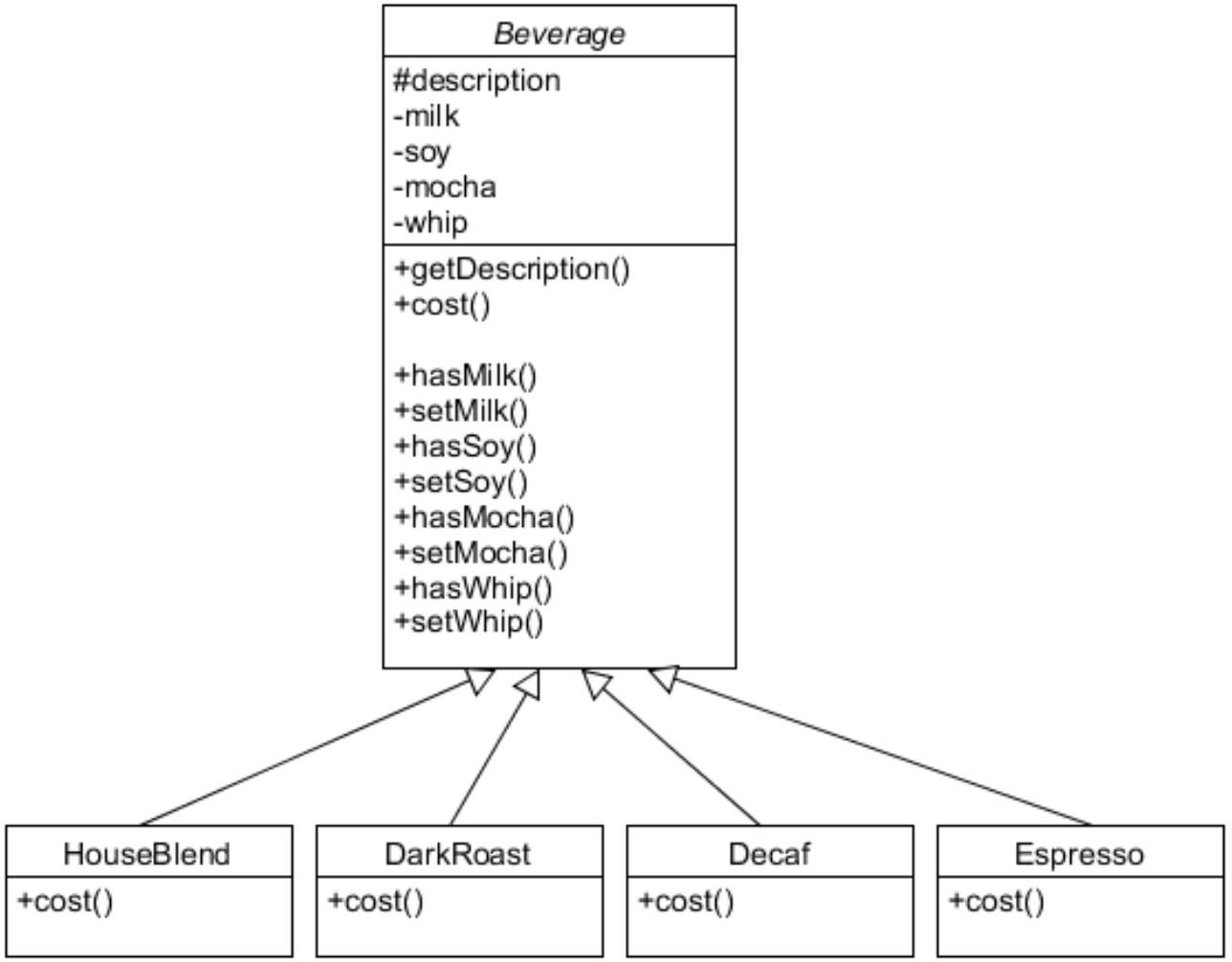
Original Design



Adding Condiments



Adding Boolean Variables



Writing cost() method

```
public class Beverage {
    protected String description;
    boolean milk, soy, mocha, whip;
    public float cost () {
        float condimentCost = 0.0;
        if (hasMilk())
            condimentCost += milkCost;
        if (hasSoy())
            condimentCost += soyCost;
        if (hasMocha())
            condimentCost += mochaCost;
        if (hasWhip())
            condimentCost += whipCost;
        return condimentCost;
    }
}

public class DarkRoast extends Beverage {
    public DarkRoast () {
        description = "Excellent Dark Roast";
    }
    public float cost () {
        return 1.99 + super.cost();
    }
}
```

Expected Changes Impacting the Design

```
public class Beverage {
    protected String description;
    boolean milk, soy, mocha, whip;
    public float cost () {
        float condimentCost = 0.0;
        if (hasMilk())
            condimentCost += milkCost;
        if (hasSoy())
            condimentCost += soyCost;
        if (hasMocha())
            condimentCost += mochaCost;
        if (hasWhip())
            condimentCost += whipCost;
        return condimentCost;
    }
}
```

- Price changes for condiments will force us to alter existing code
- New condiments will force us to add new methods and alter the cost method in the superclass
- We may have new beverages. For some of these beverages the condiments may not be appropriate
- What if a customer wants double mocha?

Design Principle: OCP

- Open Closed Principle
  - Classes should be open for extension, but closed for modification
  - allow classes to be easily extended to incorporate new behavior without modifying existing code.
    - We get designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

- Caution: Don't try applying the Open-Closed Principle (OCP) to every single case. Keep simple designs if possible!

Meet the Decorator Pattern

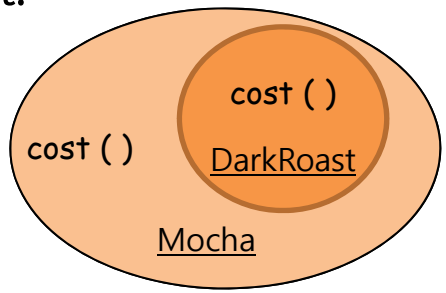
We start with a beverage and "decorate" it with the condiments at runtime. If a customer wants a Dark Roast with Mocha and Whip we do the following:

1. Take a `DarkRoast` object
2. Decorate it with a `Mocha` object
3. Decorate it with a `Whip` object
4. Call the `cost()` method and rely on delegation to add on the condiment costs.

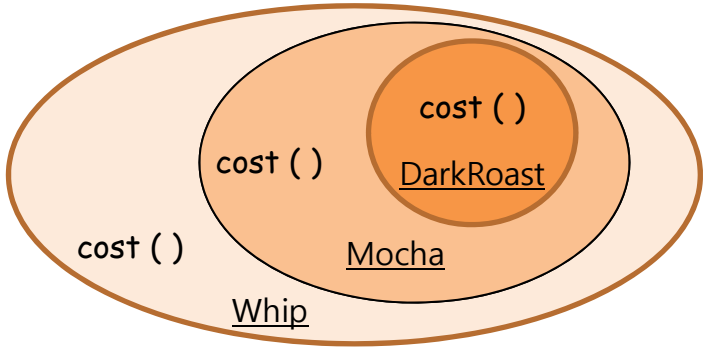
Constructing a drink order with Decorators

1. Start with the DarkRoast object

2. Customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

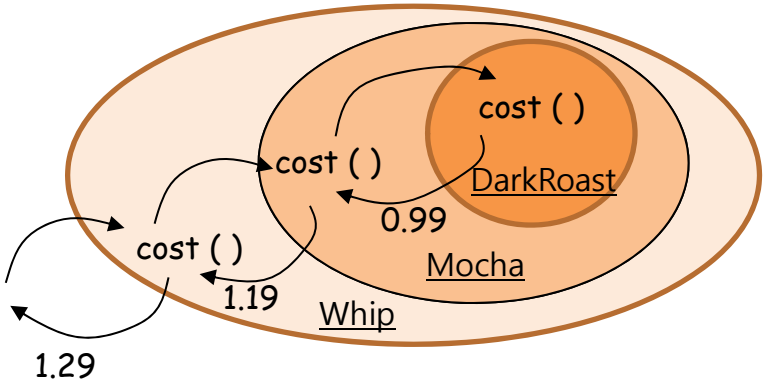


3. The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Constructing a drink order with Decorators

4. Now its time to compute the cost for the customer. Do this by calling cost( ) on the outermost decorator, Whip, and Whip is going to delegate computing cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



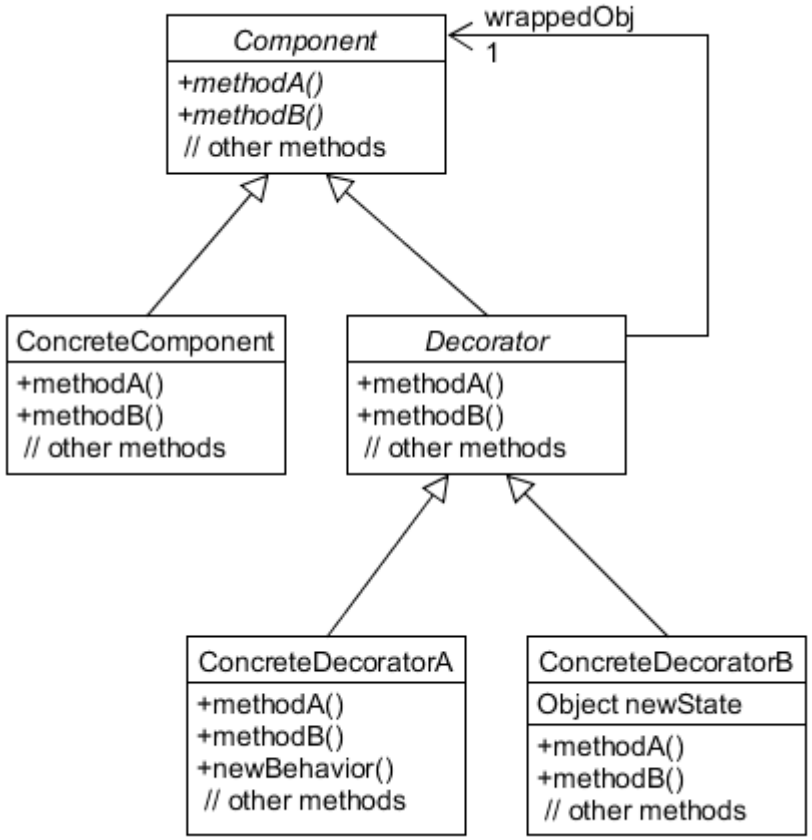
Ingredient	Price
DarkRoast	$0.99
Mocha	$0.2
Whip	$0.1

Review of Decorator Idea

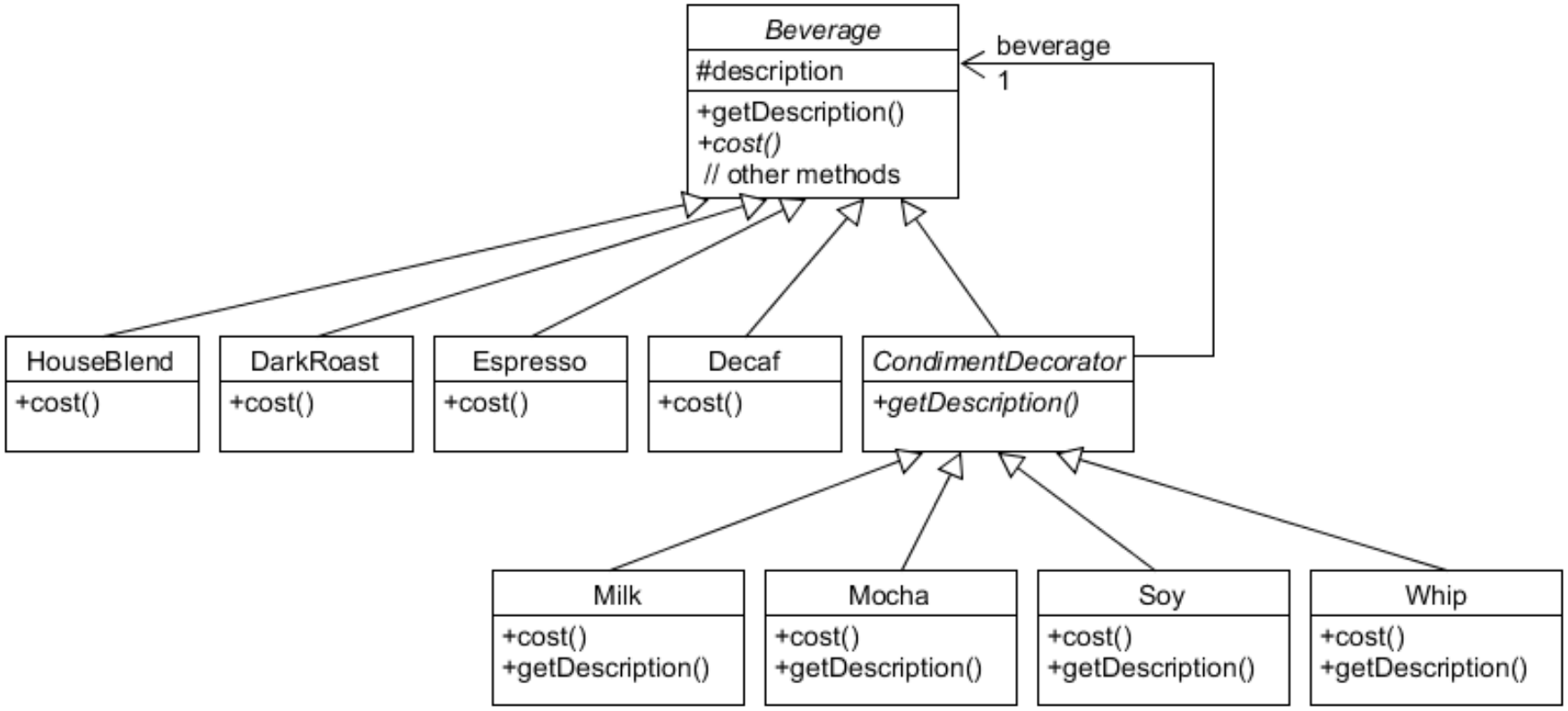
- The decorator adds its own behavior.
- You can use one or more decorators to wrap an object.
- We can pass around a decorated object in place of the original (wrapped) object.
- Decorators have the same super type as the objects they decorate.
- We can decorate objects dynamically at runtime with as many decorators as we want.

The Definition of the Decorator

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



Applying to Our Example



Beverage class and Concrete Beverage class

```
public abstract class Beverage {
   protected String description = “Unknown 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;
   }
}
```

Coding Condiments

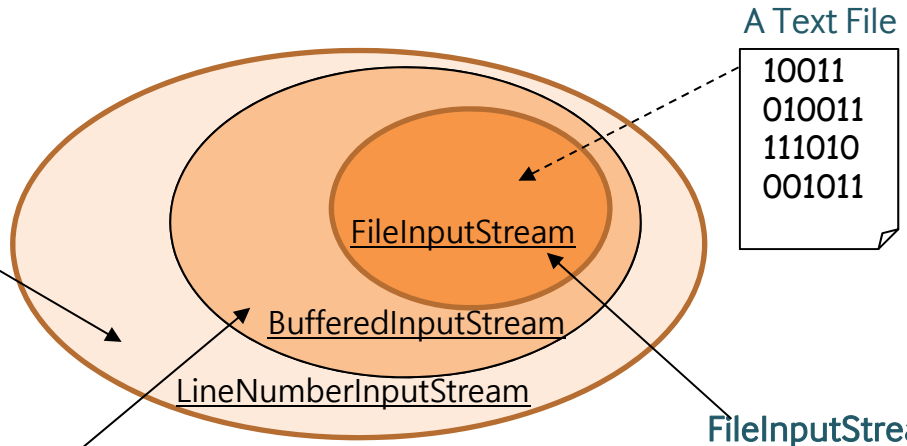
```
public abstract class CondimentDecorator extends Beverage {
   protected Beverage beverage;
   public abstract String getDescription();
}


public class Mocha extends CondimentDecorator {
   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }

   public String getDescription() {
      return beverage.getDescription() + ", Mocha";
   }

   public double cost() {
      return .20 + beverage.cost();
   }
}
```

Decorator Test Drive

```
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());

      Beverage beverage3 = new HouseBlend();
      beverage3 = new Soy(beverage3);
      beverage3 = new Mocha(beverage3);
      beverage3 = new Whip(beverage3);
      System.out.println(beverage3.getDescription()
       + " $" + beverage3.cost());
   }
}
```

Using Decorator Pattern in Java I/O

What is the typical set of objects that use decorators to add functionality to reading data from a file?

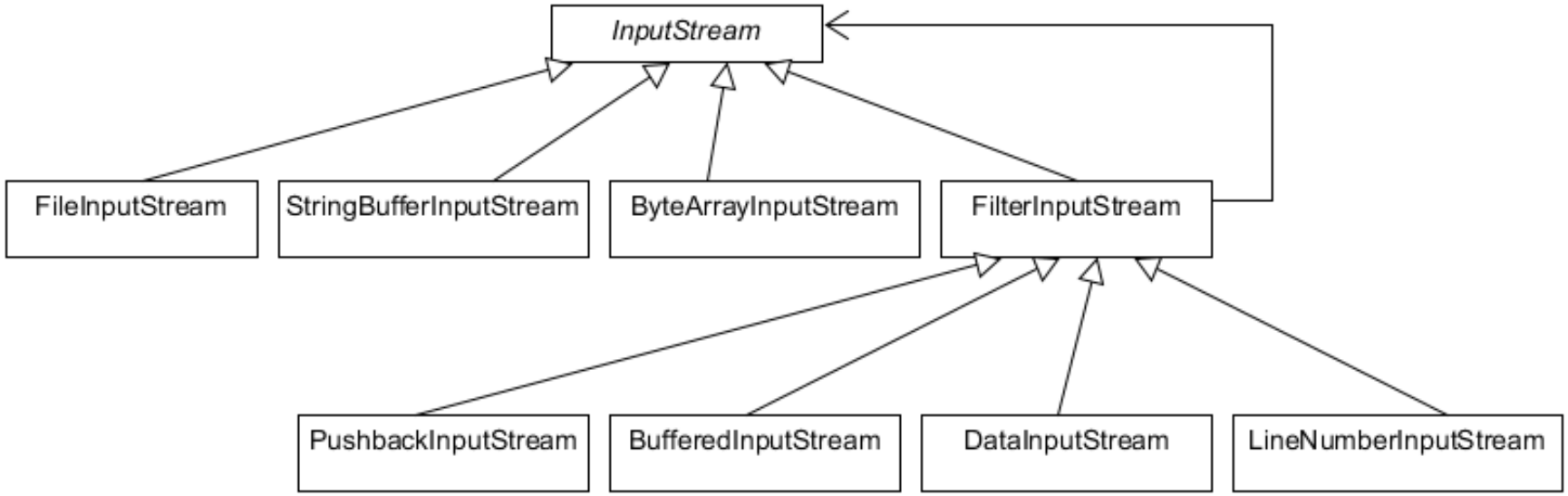


LineNumberInputStream is also a **concrete decorator**. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a **concrete Decorator** . BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine ( ) for reading character-based input, a line at a time.

FileInputStream is the **component** that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, and others. All of these give us the base component from which to read bytes.

java.io classes



A Sample of Java I/O Decorator

```
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));
   }

   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]);
      }
      return result;
   }
}
```

Run our Java I/O decorator

```
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.println((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStrackTrace();
        }
    }
}
```

Related Patterns

- Regarding the interfaces
  - Adapter provides a different interface to its subject
  - Proxy provides the same interface
  - Decorator provides an enhanced interface

- Design Principle: Open-Closed Principle (OCP)
- Mechanism: Composition and Delegation
  - Decorator Pattern attaches additional responsibilities to an object dynamically
  - Decorators provide a flexible alternative to subclassing for extending functionality
- Decorator mirrors the type of components they are decorating
  - We can wrap a component with any number of decorators

- Down-side of the decorator pattern
  - can generate a lot of small classes(e.g. Java I/O)
  - hard to understand if not familiar with the pattern