<epam> Delivering Excellence in Software Engineering

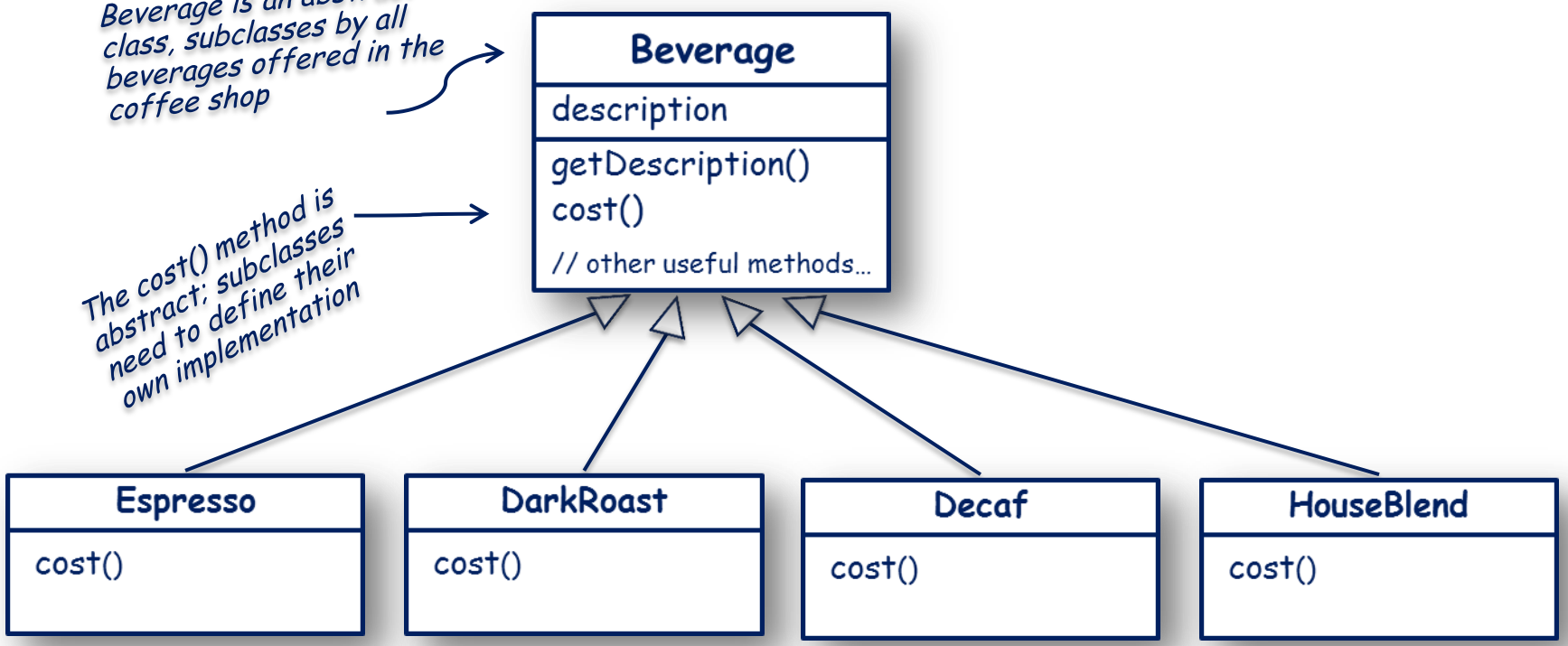
The Decorator Pattern

Vik cafe

Vik[cafe] : coffee business

Beverage is an abstract class, subclasses by all beverages offered in the coffee shop

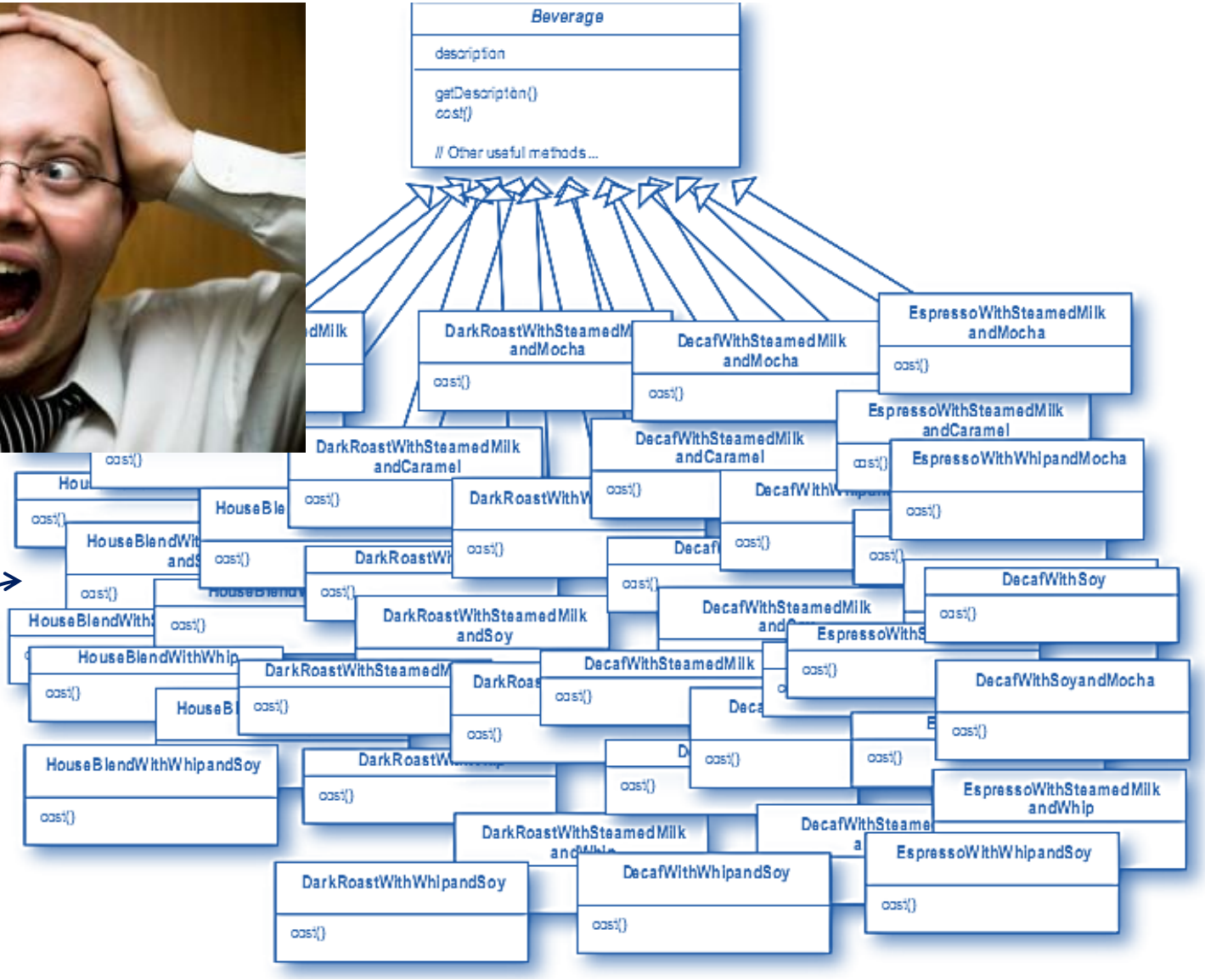
The cost() method is abstract; subclasses need to define their own implementation



<epam>

Vik[cafe] : the additions to coffee (condiments)

Can we say "class explosion"?

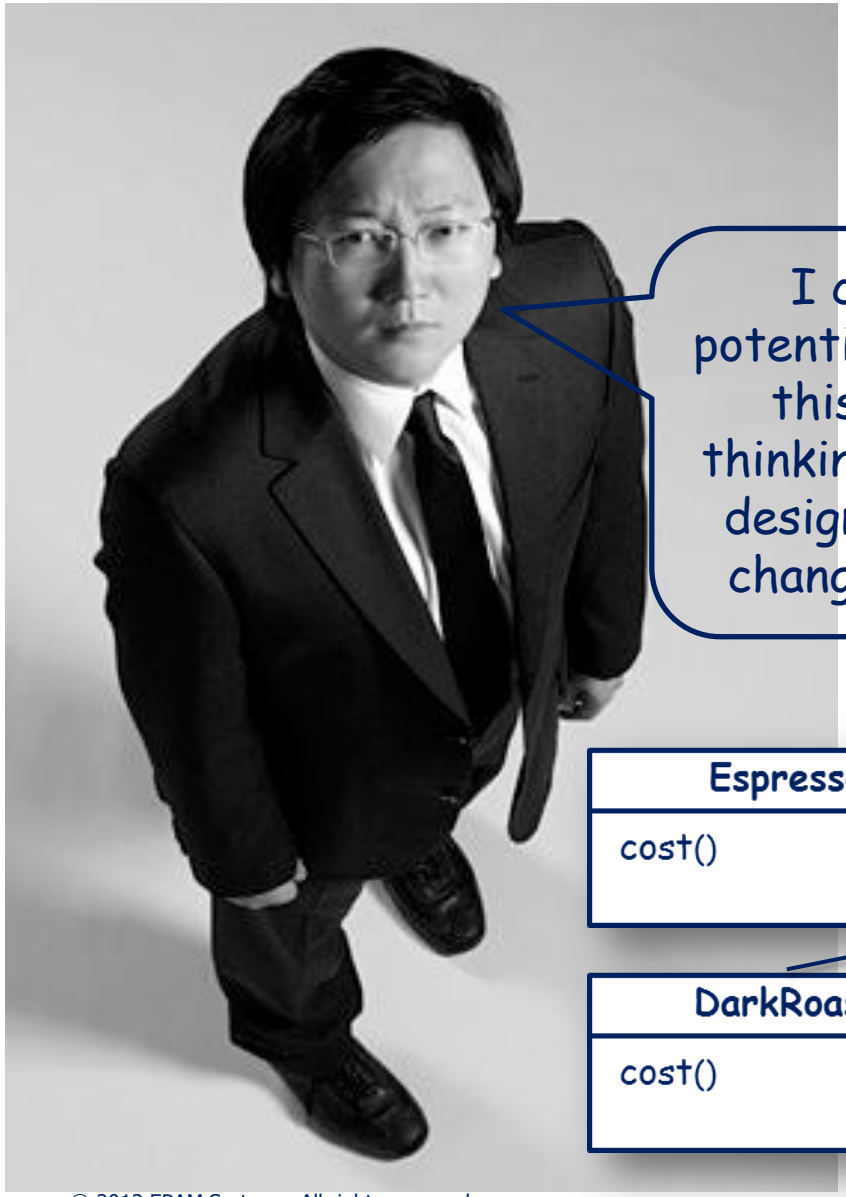


Each cost() method compute the cost of the coffee along with the other condiments in the order

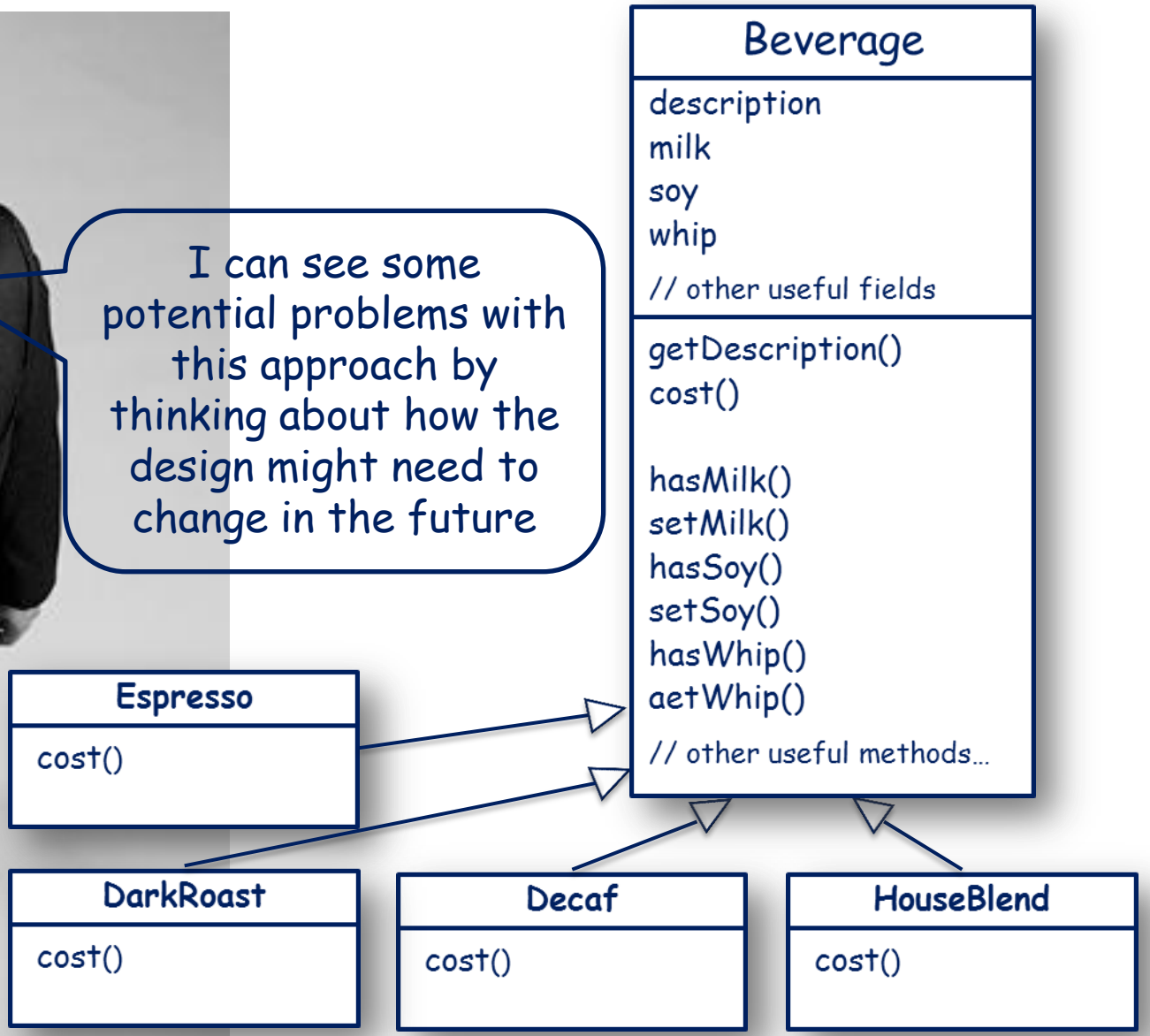
<epam>

Let's get the ball rolling with the Beverage base class

I can see the future...



I can see some potential problems with this approach by thinking about how the design might need to change in the future



Constructing a drink order with Decorators

Design Principle

Classes should be open for extension, but closed for modification.

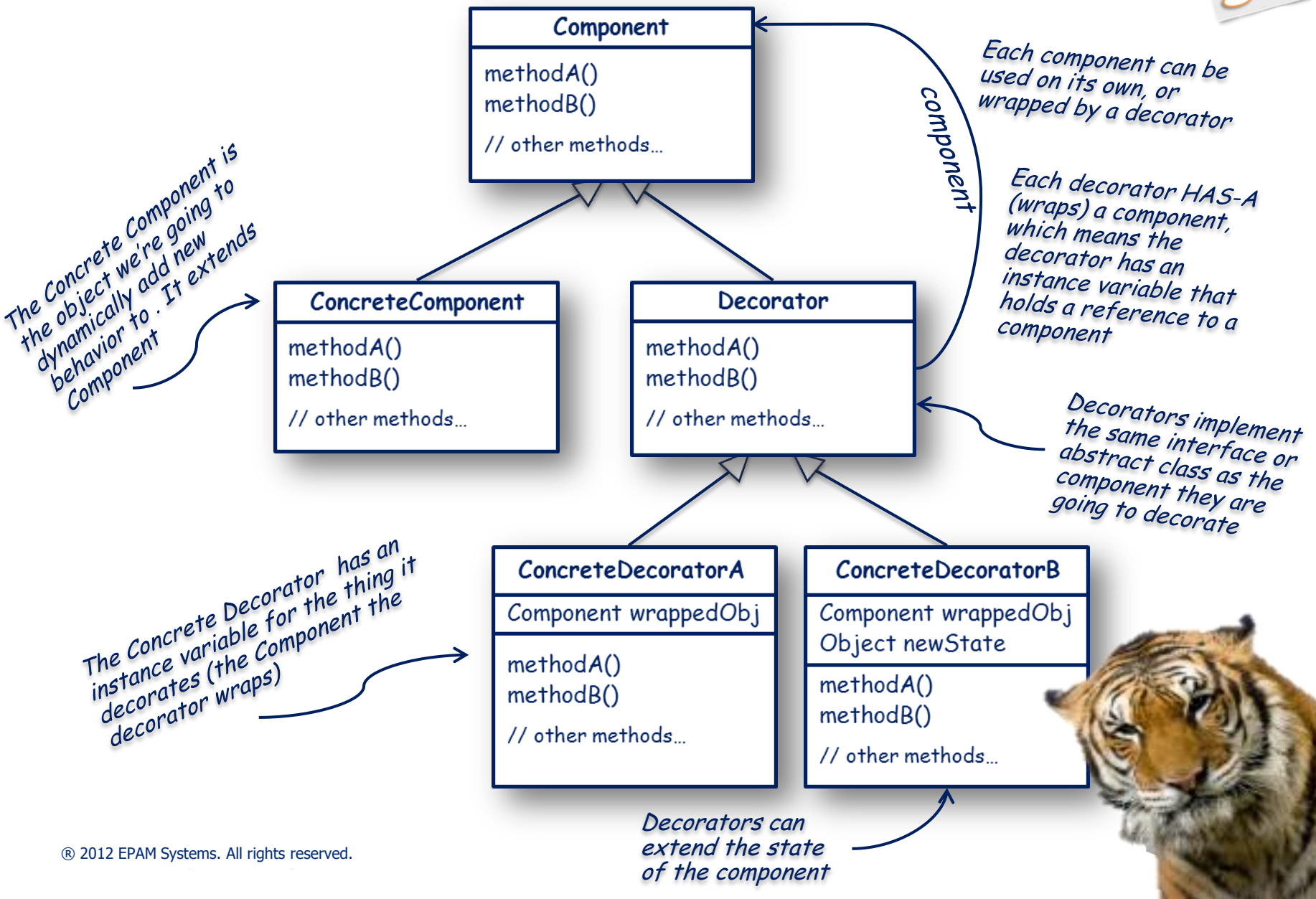
So, here's what we'll do instead: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:

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

The Decorator Pattern defined

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

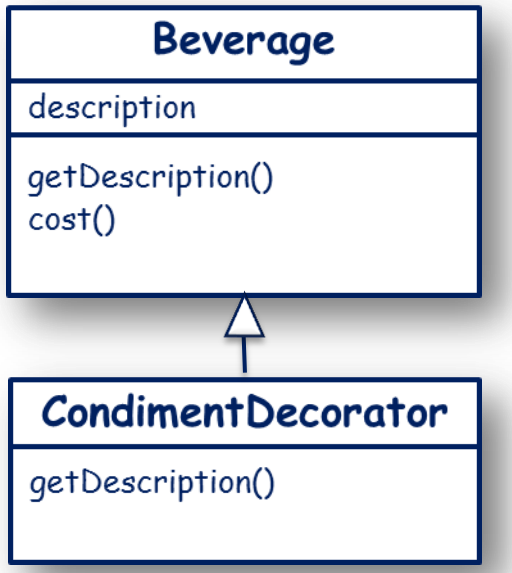
Pattern Decorator: the class diagram



Vik cafe : decorating our Beverages

Writing the abstract classes

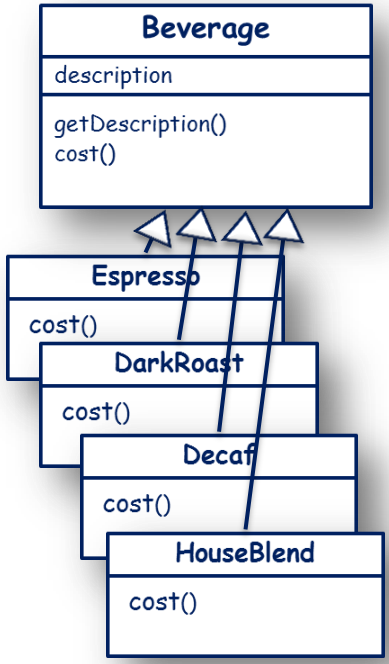
```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription(){
        return description;
    }

    public abstract double cost();
}
```




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

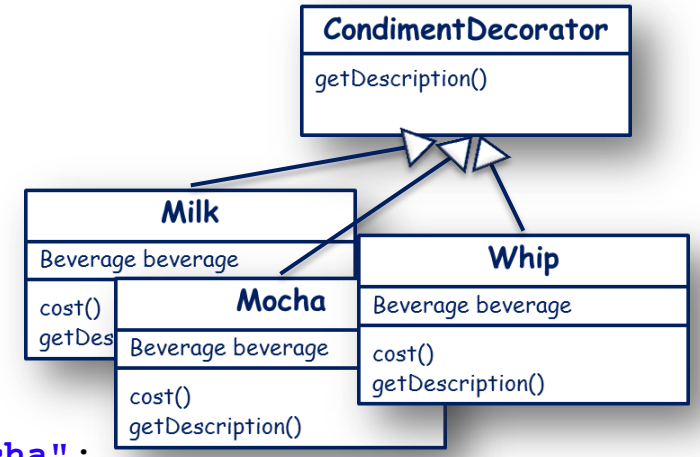
Coding beverages

```
public class Espresso extends Beverage {
    public Espresso(){ description = "Espresso"; }
    @Override
    public double cost() { return 1.99; }
}
public class DarkRoast extends Beverage {
    public DarkRoast(){ description = "Dark Roast coffee"; }
    @Override
    public double cost() { return 1.59; }
}
public class Decaf extends Beverage{
    public Decaf(){
        description = "Decaf coffee: real caffeine-free"; }
    @Override
    public double cost() { return .79; }
}
public class HouseBlend extends Beverage {
    public HouseBlend(){ description = "House Blend coffee"; }
    @Override
    public double cost() { return .99; }
}
```



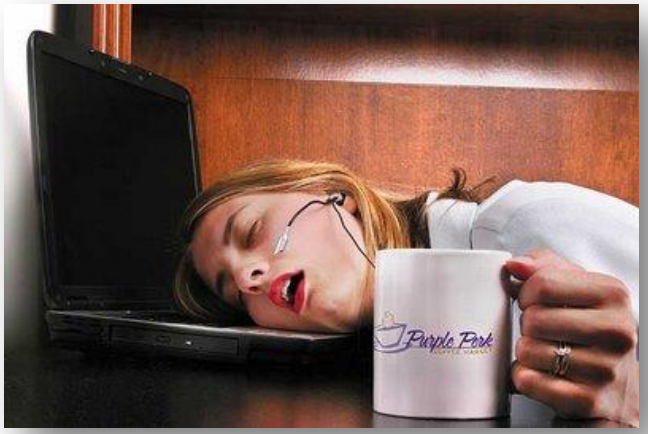
Coding condiments

```
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 .20 + beverage.cost();
    }
}
```



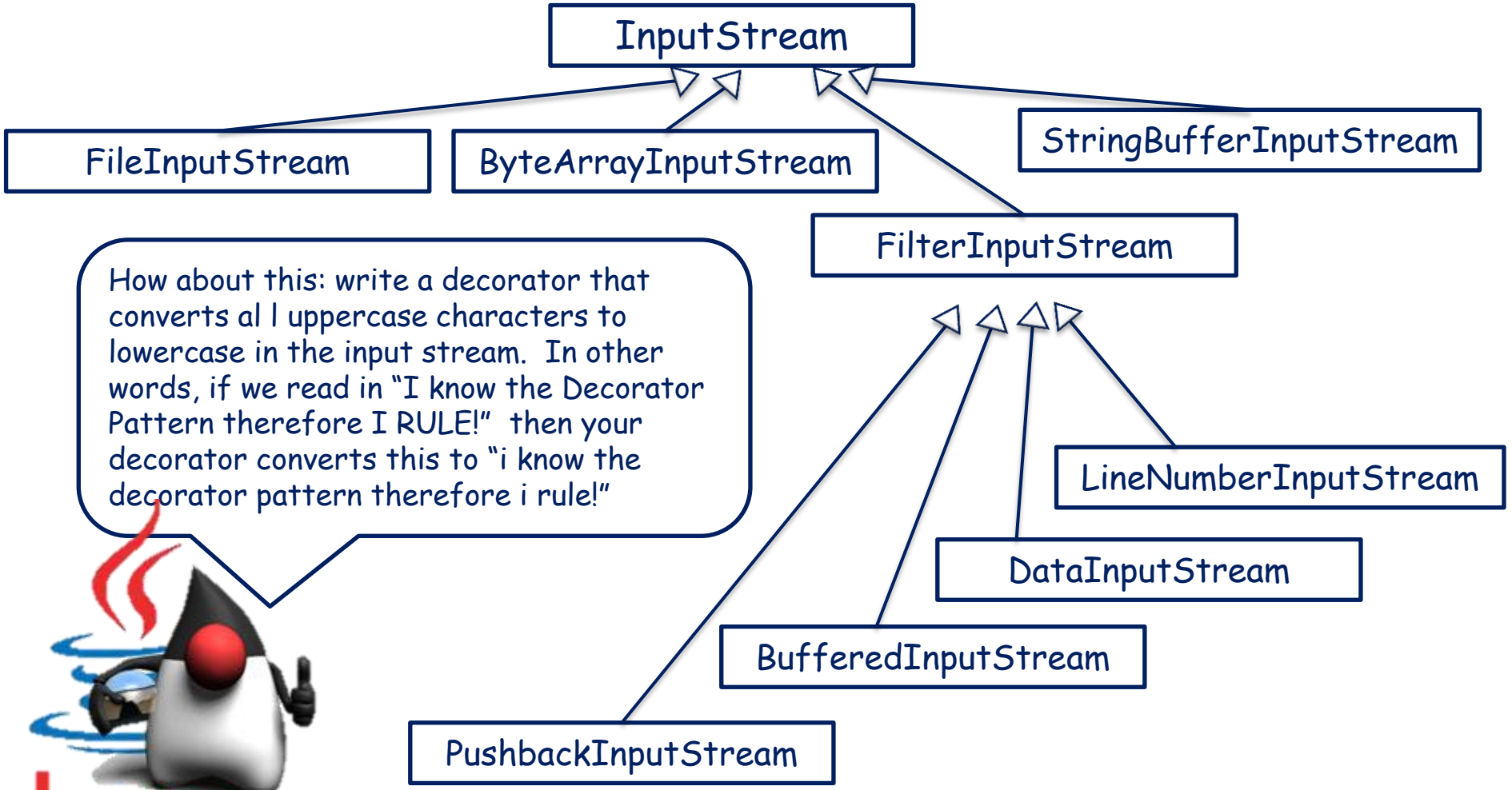
Serving some coffee

```
public class LavazzaCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
            + " Br " + beverage.cost());
```



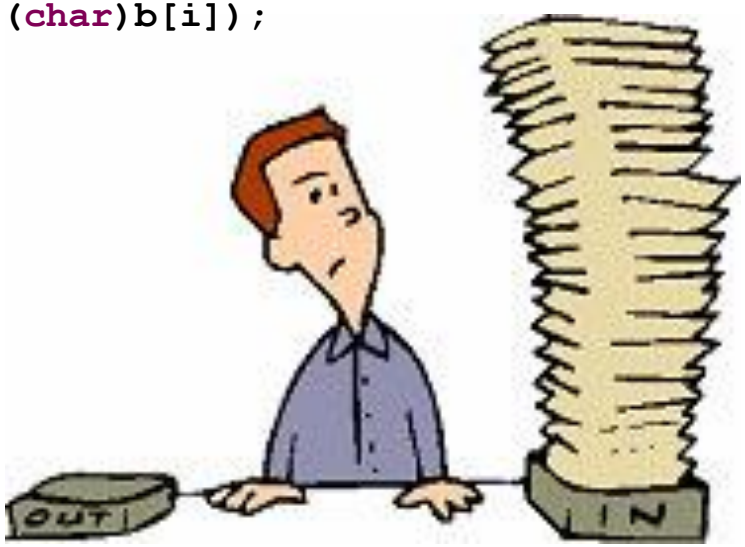
```
        Beverage beverage2 = new Mocha(new Mocha(new Whip(new DarkRoast());

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

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

Real World Decorators: java.io



Writing our own 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;
    }
}
```



Test out our new 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.print((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

<epam> Delivering Excellence in Software Engineering

The Decorator Pattern

For more information, please contact:
Victor Ivanchenko,
epam trainer
Email: ivanvikvik@gmail.com

EPAM Systems, Inc.
http://www.epam.com