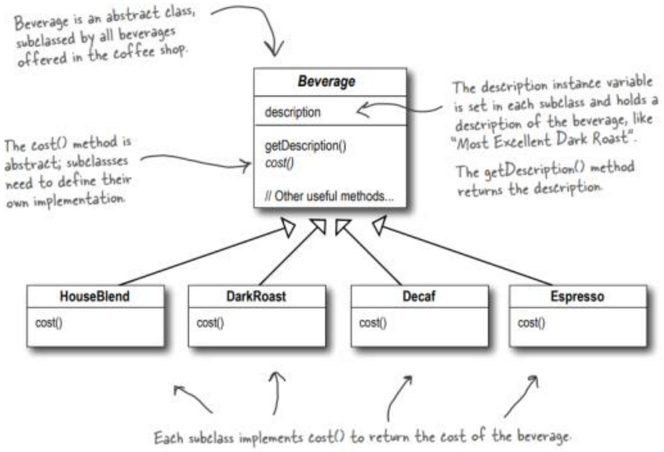
DECORATOR DESIGN PATTERN

why ?

→ when we want to rearrange classes, So as to get Objects at runtime rather than doing it at compile time. we use composition for it.

Taking Example of Starbuzz Coffee

Initial Stage



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

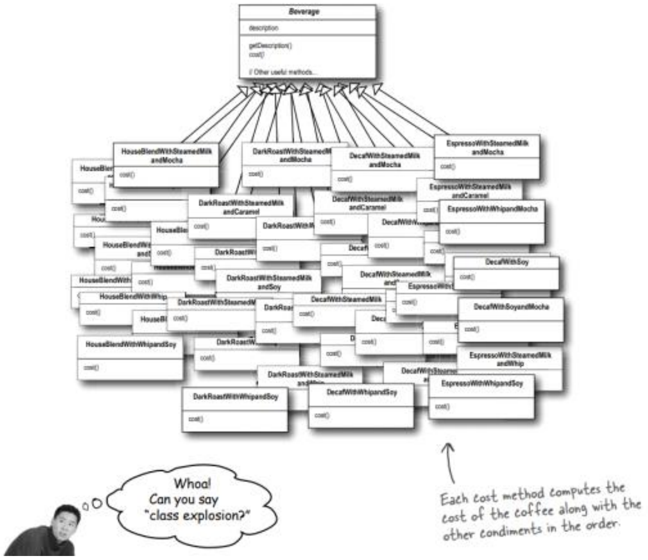
The cost() method is abstract; subclassses need to define their own implementation.

The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.

Each subclass implements cost() to return the cost of the beverage.

what if more variety of coffee coming up ? this is how it will look



Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilkandMocha
DarkRoastWithSteamedMilkandMocha
DecafWithSteamedMilkandMocha
EspressoWithSteamedMilkandMocha
DarkRoastWithSteamedMilkandCaramel
DecafWithSteamedMilkandCaramel
EspressoWithSteamedMilkandCaramel
EspressoWithWhipandMocha
DarkRoastWithSteamedMilkandSoy
DecafWithSoy
DecafWithSoyandMocha
HouseBlendWithWhipandSoy
EspressoWithSteamedMilkandWhip
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()
Whoa! Can you say "class explosion?"

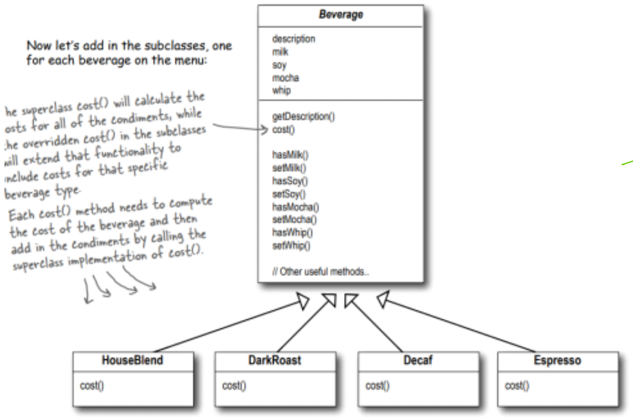
Each cost method computes the cost of the coffee along with the other condiments in the order.

It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

?

Another possible Approach ?



Now let's add in the subclasses, one for each beverage on the menu:

he superclass cost() will calculate the osts for all of the condiments, while he overridden cost() in the subclasses will extend that functionality to nclude costs for that specific beverage type.

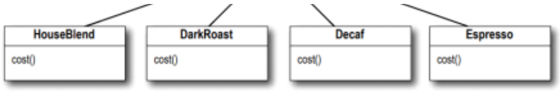
Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

→ what's problem with it?

→ difficult to maintain if new changes comes up.

→ what changes? let's discuss.

rinciple.



violating O of solid principle.

What requirements or other factors might change that will impact this design?

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 (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

What if a customer wants a double mocha?

possible issue.

The Open-Closed Principle

Design Principle

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

Meet the Decorator Pattern

Okay, enough of the "Object Oriented Design Club." We have real problems here! Remember us? Starbuzz Coffee? Do you think you could use some of those design principles to actually help us?

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**

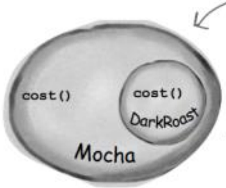
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Constructing a drink order with Decorators

1. **We start with our DarkRoast object.**

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

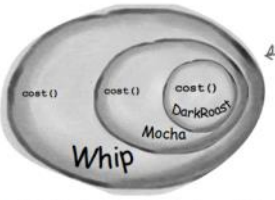
2. **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type.)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

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

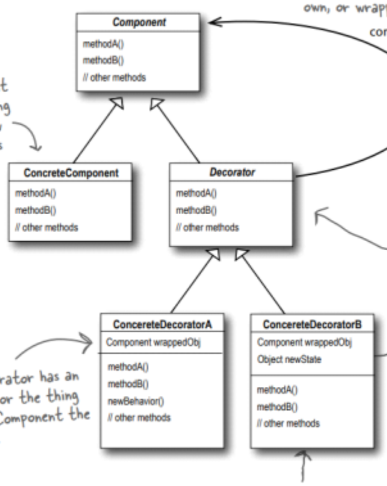


Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

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.

4. **Now it's time to compute the cost for the customer. We do this**

Class Level diagram (Generic)



Each component can be used on its own, or wrapped by a decorator.

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can extend the state of the component.

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

solution of double mocha

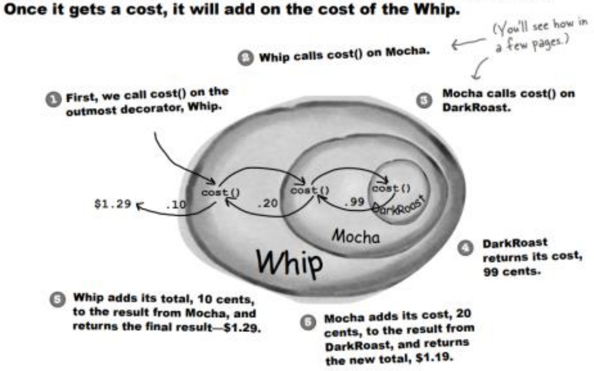
New barista training

"double mocha soy latte with whip"

1. **Whip calls cost() on Mocha**

a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

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



1 First, we call cost() on the outmost decorator, Whip.

2 Whip calls cost() on Mocha.

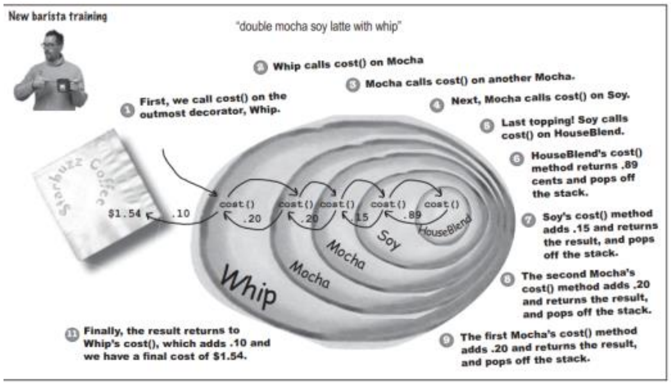
(You'll see how in a few pages.)

3 Mocha calls cost() on DarkRoast.

4 DarkRoast returns its cost, 99 cents.

5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.



New barista training

"double mocha soy latte with whip"

1 First, we call cost() on the outmost decorator, Whip.

2 Whip calls cost() on Mocha

3 Mocha calls cost() on another Mocha.

4 Next, Mocha calls cost() on Soy.

5 Last topping! Soy calls cost() on HouseBlend.

6 HouseBlend's cost() method returns .89 cents and pops off the stack.

7 Soy's cost() method adds .15 and returns the result, and pops off the stack.

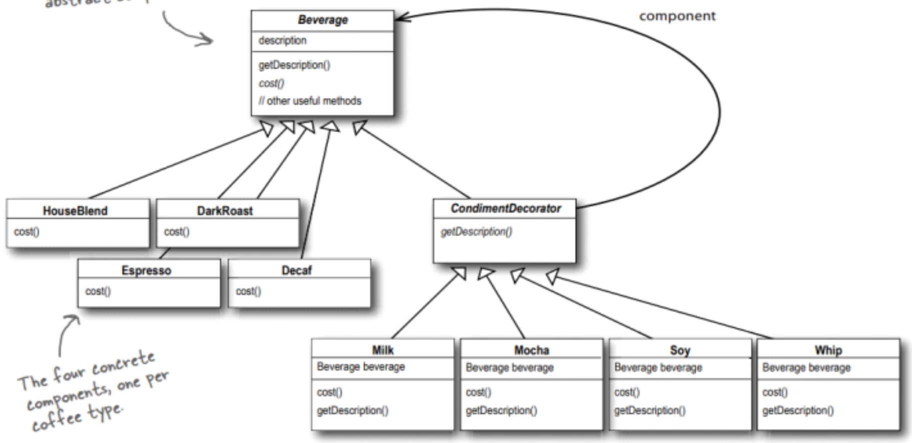
8 The second Mocha's cost() method adds .20 and returns the result, and pops off the stack.

9 The first Mocha's cost() method adds .20 and returns the result, and pops off the stack.

11 Finally, the result returns to Whip's cost(), which adds .10 and we have a final cost of $1.54.

Decorating our Beverages

Okay, let's work our Starbuzz beverages into this framework...



Beverage acts as our abstract component class.

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Writing the Starbuzz code

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

abstract decorator to be used by condiments

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

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...

Concrete components

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

First we extend the Beverage class, since this is a beverage.

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

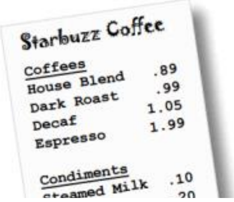
Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just

Concrete Components

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return



Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
	.20

```
    public double cost() {
        return 1.99;
    }
}
```

class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

```
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

You can create the other two Beverage classses (DarkRoast and Decaf) in exactly the same way.

Dark Ro 1.05
Decaf 1.99
Espresso

Condiments
Steamed Milk .10
Mocha .20
Soy .15
Whip .10

Concrete Contiments

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
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();
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Here's some test code*to make orders: Testing our code.

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

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

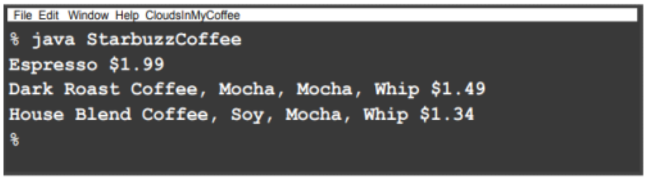
Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

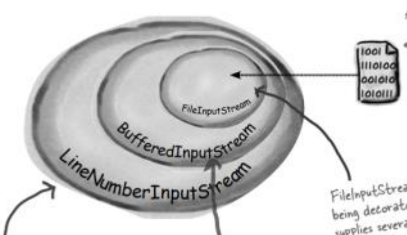
* We're going to see a much better way of creating decorated objects when we cover the Factory Pattern (and the Builder Pattern, which is covered in the appendix).



```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Real World Decorators: Java I/O

Implemented in java library.



A text file for reading.

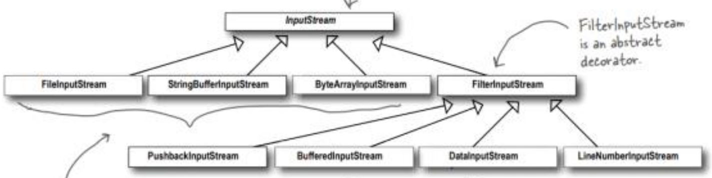
LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

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, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

BufferedInputStream and ***LineNumber***InputStream both extend ***Filter***InputStream, which acts as the abstract decorator class.

Decorating the java.io classes



Here's our abstract component

FilterInputStream is an abstract decorator.

These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.

And finally, here are all our concrete decorators.

Writing your own Java I/O Decorator

Don't forget to import java.io... (not shown)

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

```
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;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

Test out your new Java I/O Decorator

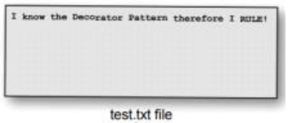
```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

Just use the stream to read characters until the end of file and print as we go.

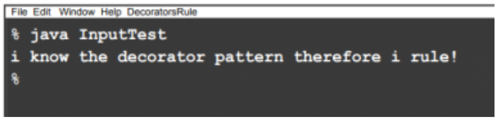


```
I know the Decorator Pattern therefore I RULE!
```

test.txt file

You need to make this file.

Give it a spin:



```
% java InputTest
i know the decorator pattern therefore i rule!
%
```