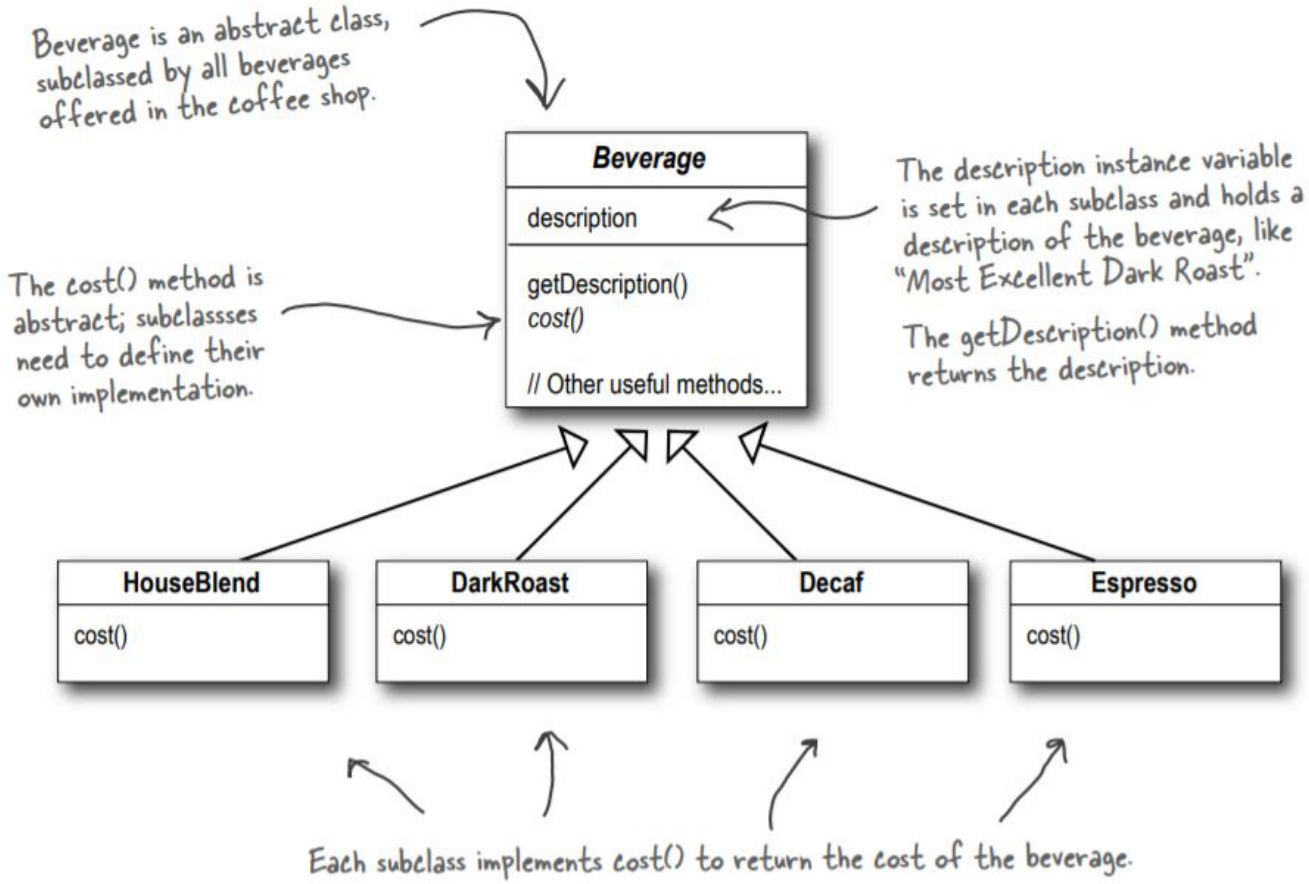
Design Patterns

Decorator Pattern

Starbuzz Coffee Shop
Example: Ordering System

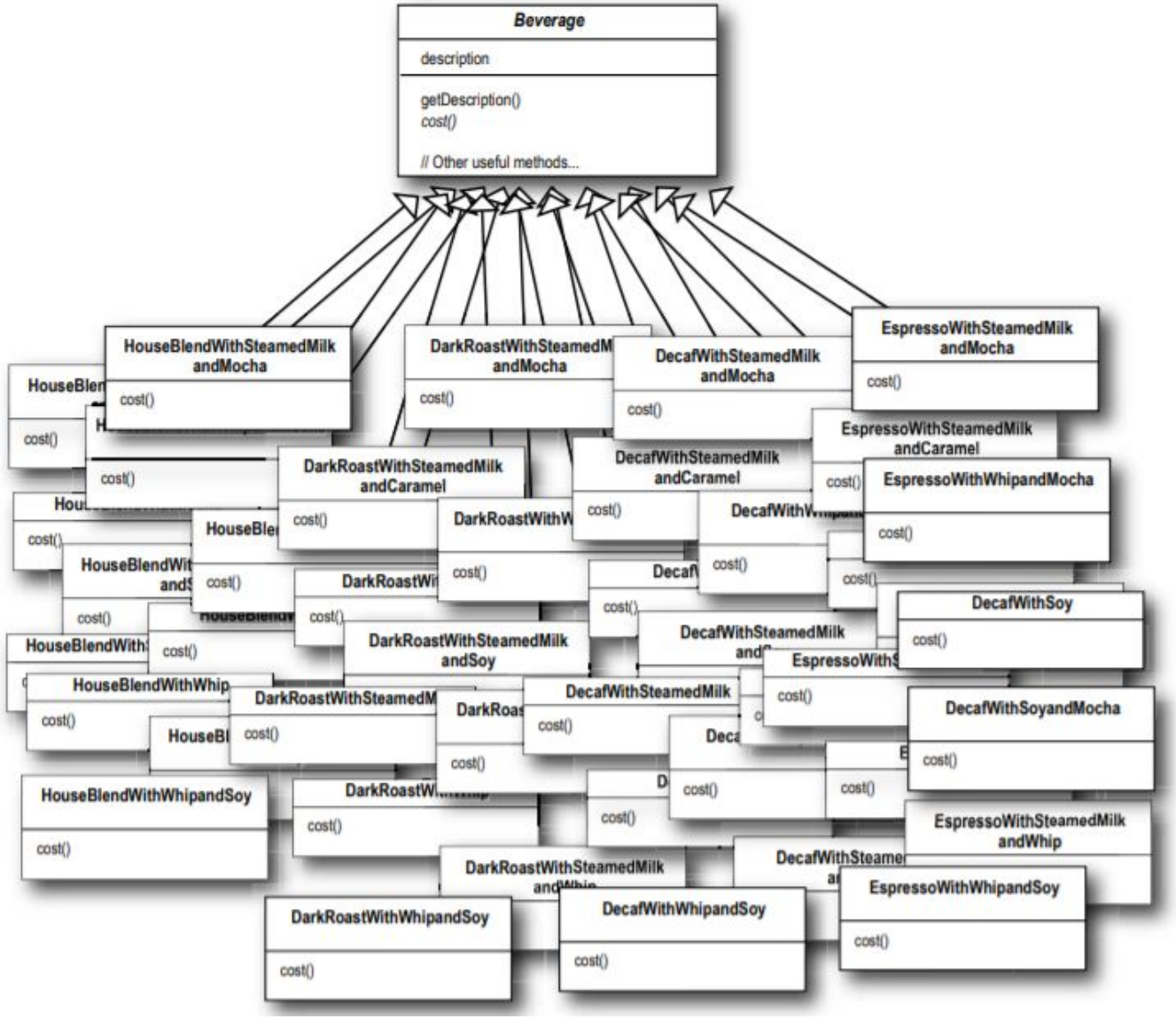


Example: Ordering System Solution#1

You can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.

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

***Problem**?*

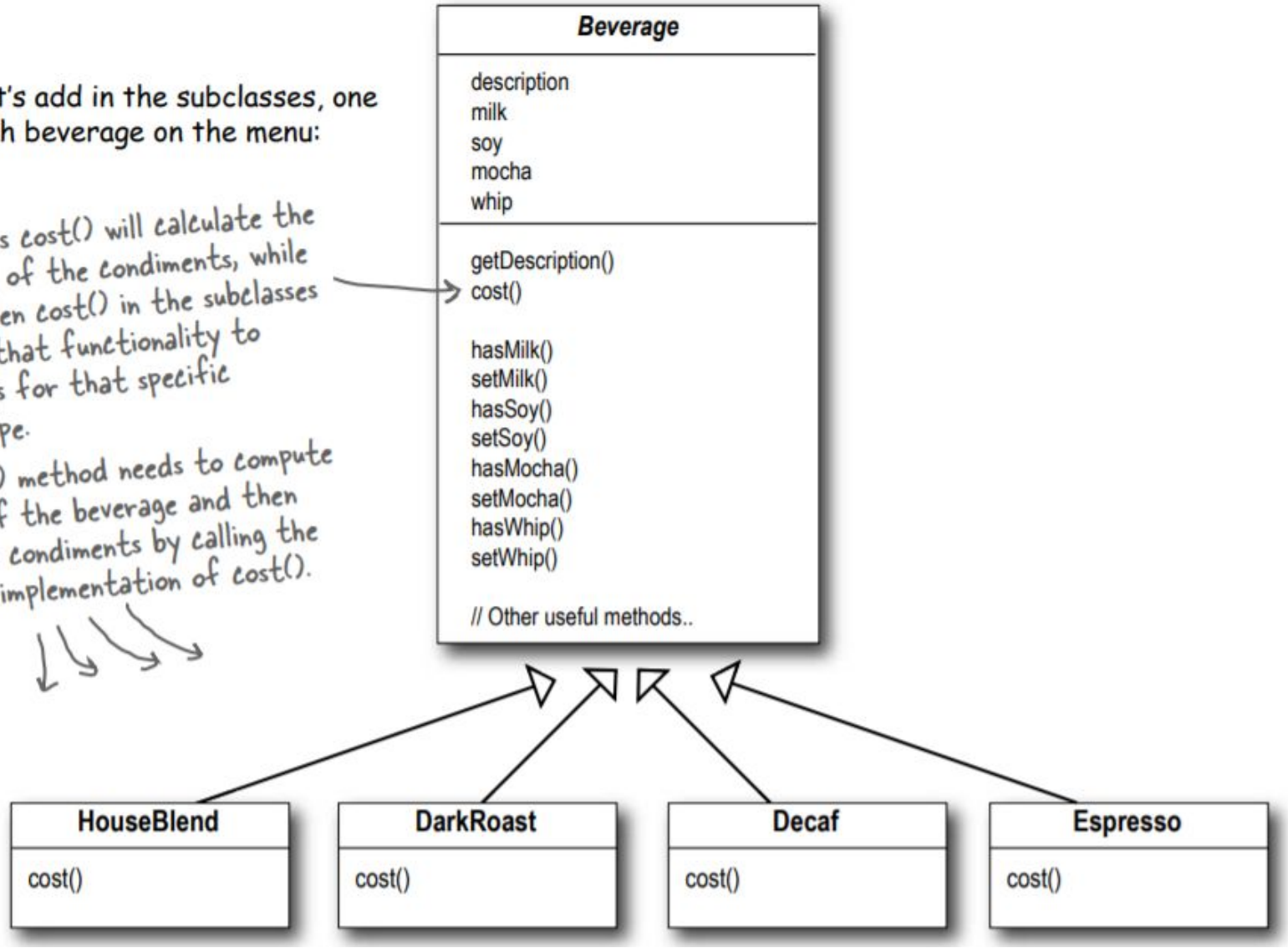


Example: Ordering System Solution#2

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

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include 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().



Example: Ordering System Solution#2

Problems with using 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?

Design Principle: The Open-Closed Principle

Design Principle: Classes should be open for extension but closed for modification.

Code should be closed (to change) like the lotus flower in the evening, yet open (to extension) like the lotus flower in the morning.

Decorator Pattern

- Decorate your classes at runtime using a form of object composition.

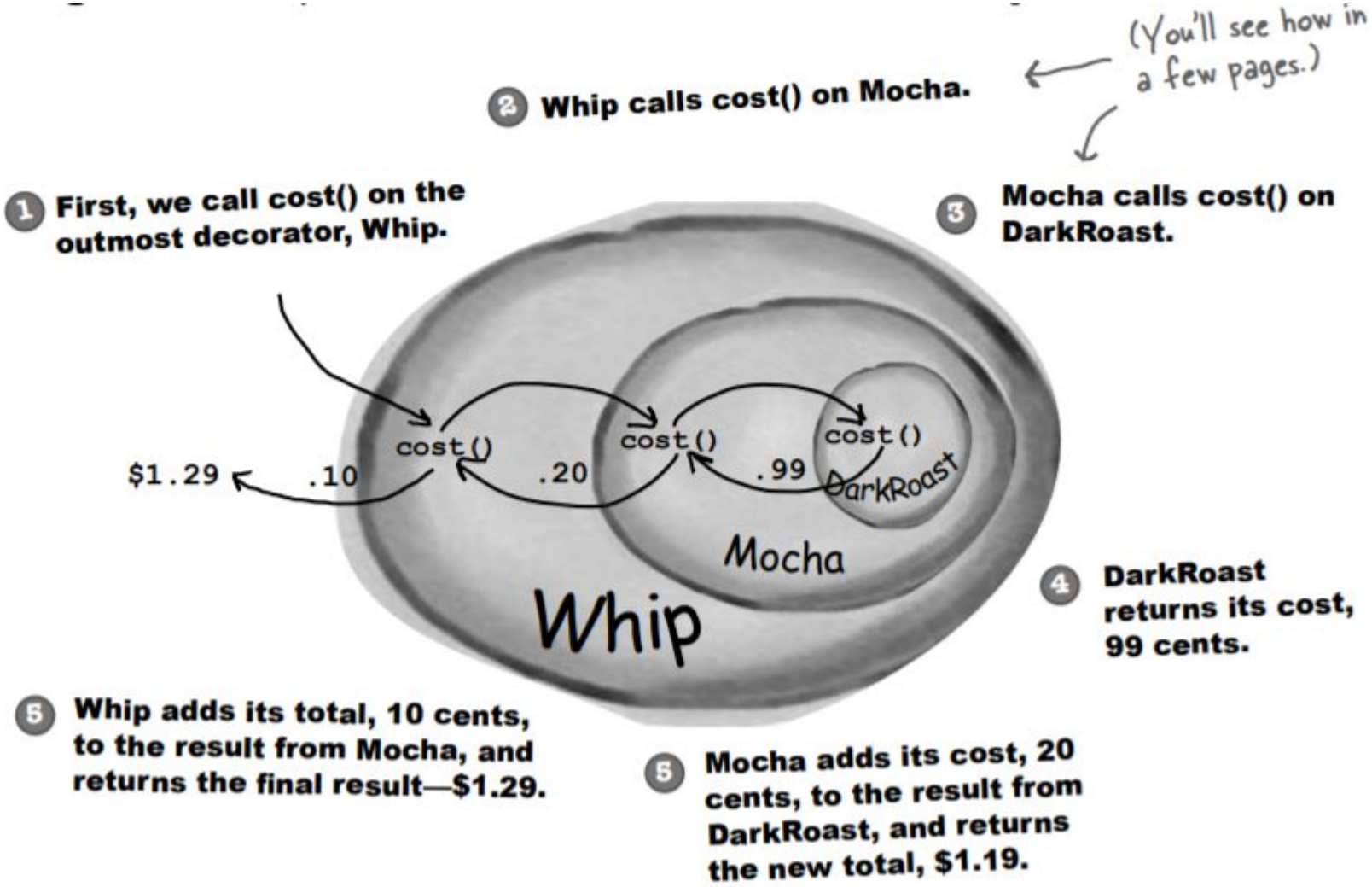
- Once we know the techniques of decorating, we will be able to give our (or someone else's) objects new responsibilities without making any code changes to the underlying classes.

- 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**

Think of decorator objects as "wrappers."

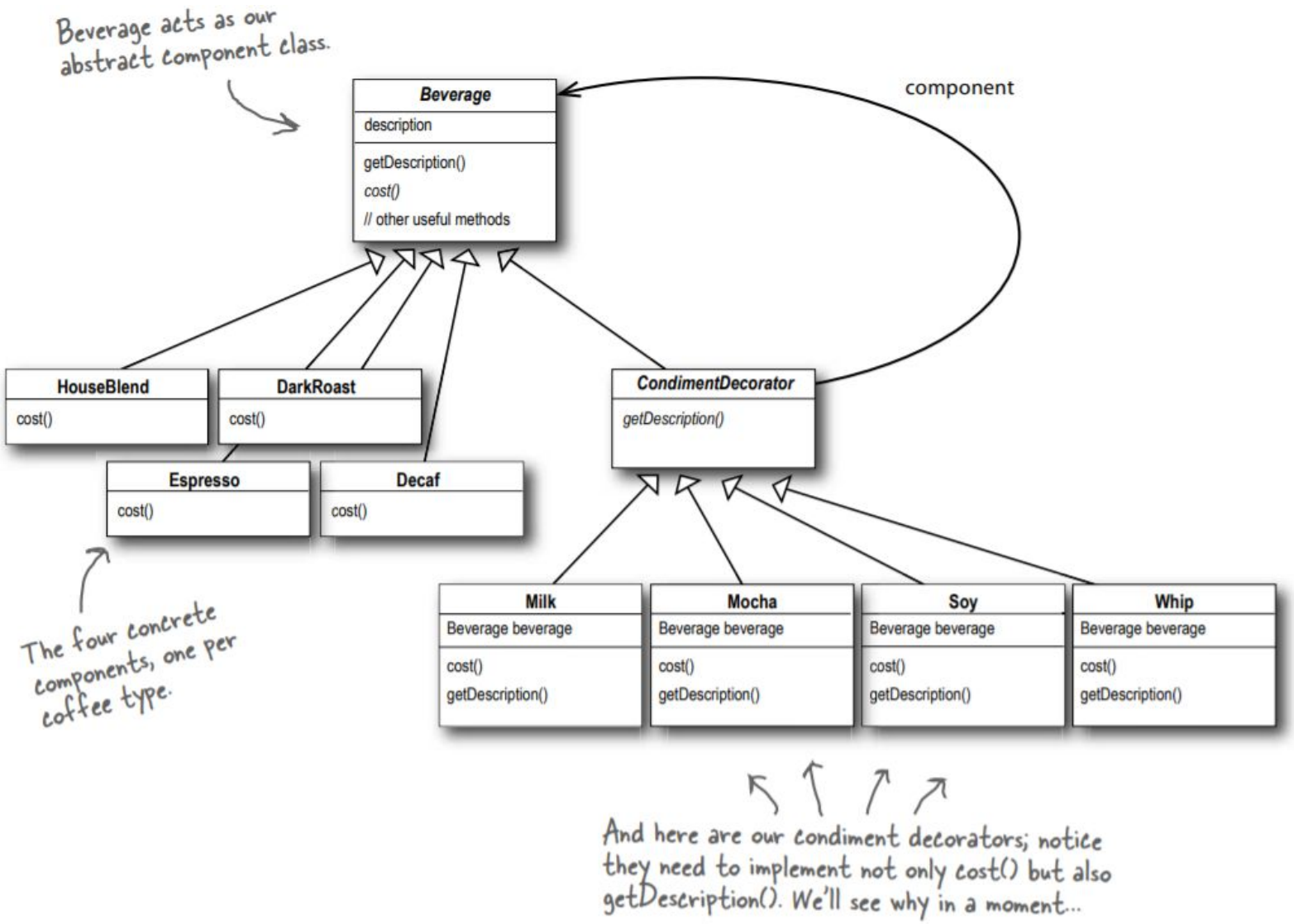
Decorator Pattern



Key Points

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Class Diagram
Decorating the Beverages



Let see some code!

Decorator Pattern - Code

Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:

```
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.

Decorator Pattern - Code

Beverage is simple enough. Let's implement the abstract class for the Condiments (Decorator) as well:

```
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...

Decorator Pattern - Code

Coding Beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and implement the cost() method.

```
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 need to return the price of an Espresso: $1.99.

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

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

Decorator Pattern - Code

Coding Condiments

If you look back at the Decorator Pattern class diagram, you'll see we've now written our abstract component (Beverage), we have our concrete components (HouseBlend), and we have our Abstract decorator (CondimentDecorator). ***Now it's time to implement the concrete decorators. Here's Mocha:***

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.

Decorator Pattern - Code

Main/Driver Class

Here's some test code to make orders:

```
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.

Decorator Pattern - Code

Main/Driver Class

Output

```
File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Technical Debt

Technical Debt Landscape

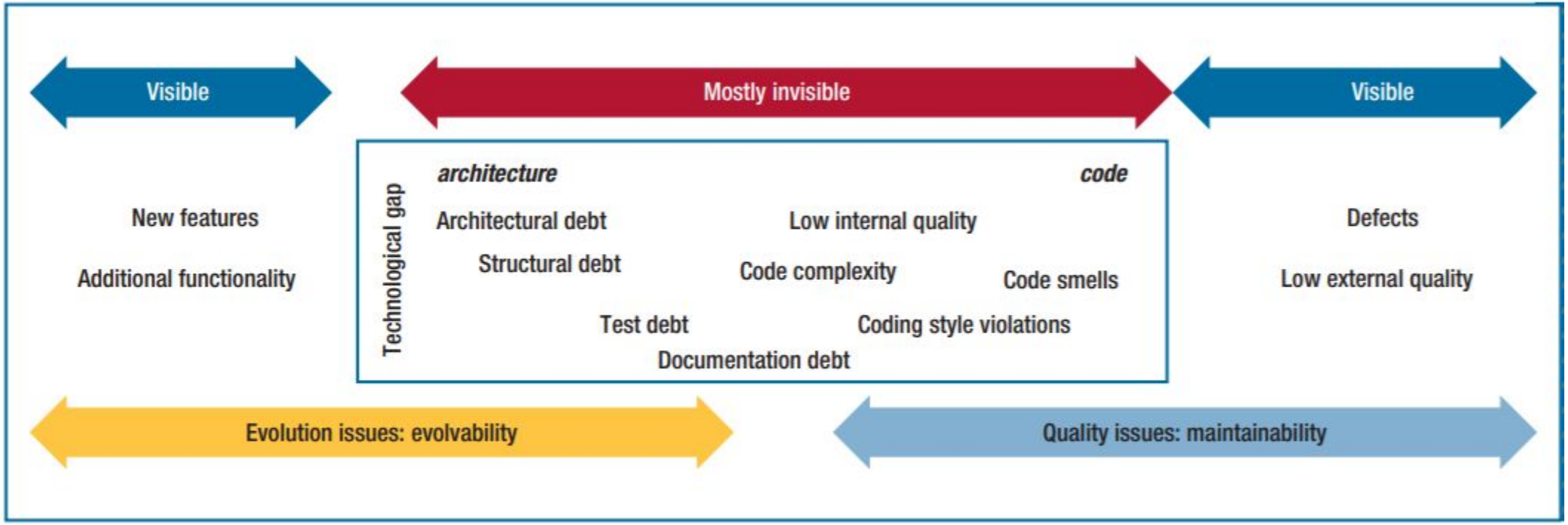


FIGURE 1. The technical debt landscape. On the left, evolution or its challenges; on the right, quality issues, both internal and external.

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. Ieee software, 29(6), 18-21.

Technical Debt Landscape

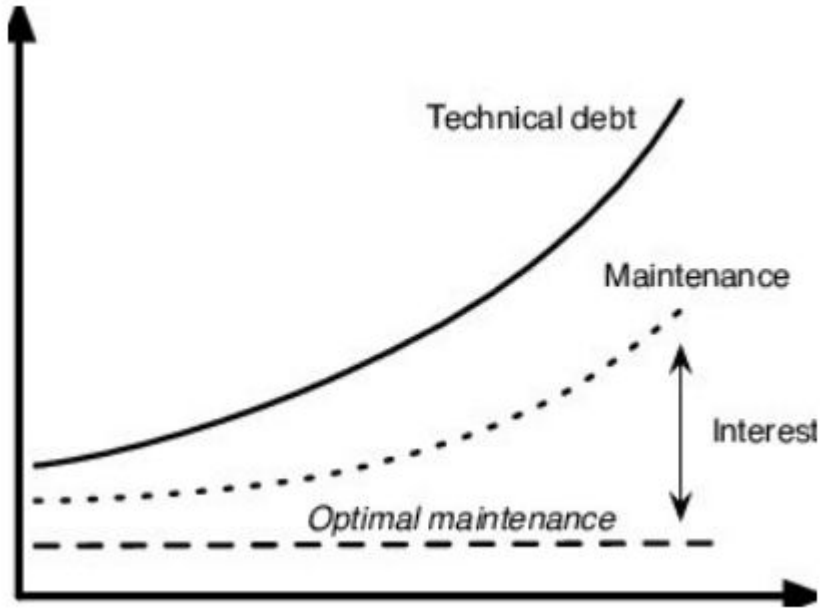


Figure: Technical debt and its interest grow over time if not resolved

https://www.researchgate.net/publication/228684782_An_Empirical_Model_of_Technical_Debt_and_Interest