CS342: Software Design

Oct 24, 2017

Today's topic

Mid-term Question 2

A question from a student about class design

Decorator pattern

Mid-term question 4

```java
public class JunitAnnotation {
    @BeforeClass
    public static void beforeClass() {
        System.out.println("in before class");
    }

    @AfterClass
    public static void  afterClass() {
        System.out.println("in after class");
    }

    @Before
    public void before() {
        System.out.println("in before");
    }

    @After
    public void after() {
        System.out.println("in after");
    }
```

```java
    @After
    public void after() {
        System.out.println("in after");
    }

    @Test
    public void test() {
        System.out.println("in test");
    }

    @Test
    public void testAgain() {
        System.out.println("in testAgain");
    }

    @Ignore
    public void ignoreTest() {
        System.out.println("in ignore test");
    }
}
```

Execution order

First of all, the beforeClass() method executes only once.

The afterClass() method executes only once.

The before() method executes for each test case, but before executing the test case.

The after() method executes for each test case, but after the execution of test case.

In between before() and after(), each test case executes.

@ignore means the test case will be skipped

in before class -> in before -> in test -> in after -> in before -> in test again -> in after -> in after class

What kind of Pizza would cut itself?

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    abstract Pizza createPizza(String type);
}
```

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();
    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < toppings.size(); i++) {
            System.out.println(" " + toppings.get(i));
        }
    }
    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }
    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }
    public String getName() {
        return name;
    }
}
```

Alternative approach

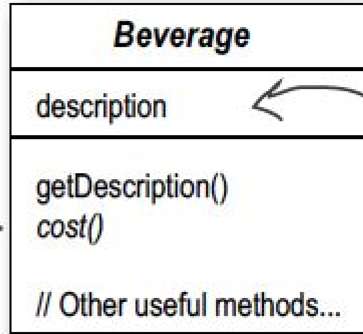
```
public abstract class PizzaStore {
    private void prepare(pizza) {
        System.out.println("Preparing " + pizza.name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < pizza.toppings.size(); i++) {
            System.out.println(" " + pizza.toppings.get(i));
        }
    }
    private void bake(pizza) {
        System.out.println("Bake for 25 minutes at 350");
    }
    private void cut(pizza) {
        System.out.println("Cutting the pizza into diagonal slices");
    }
    private void box(pizza) {
        System.out.println("Place pizza in official PizzaStore box");
    }
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        prepare(pizza);
        bake(pizza);
        cut(pizza);
        box(pizza);
        return pizza;
    }
    abstract Pizza createPizza(String type);
}
```

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();
    public String getName() {
        return name;
    }
}
```

After pizza, now it's time for some coffee

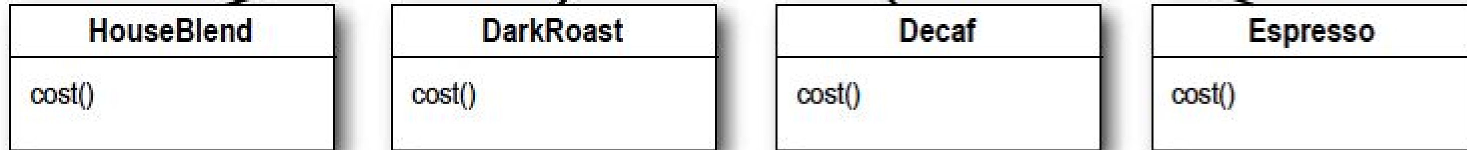
Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

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

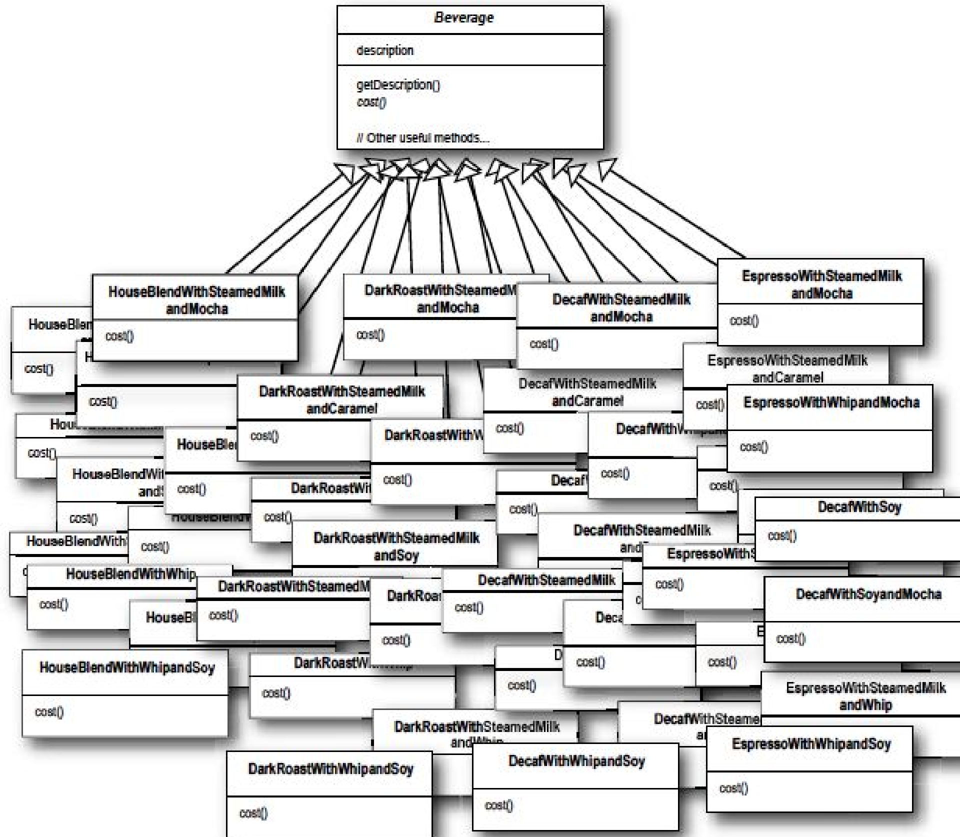


The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.



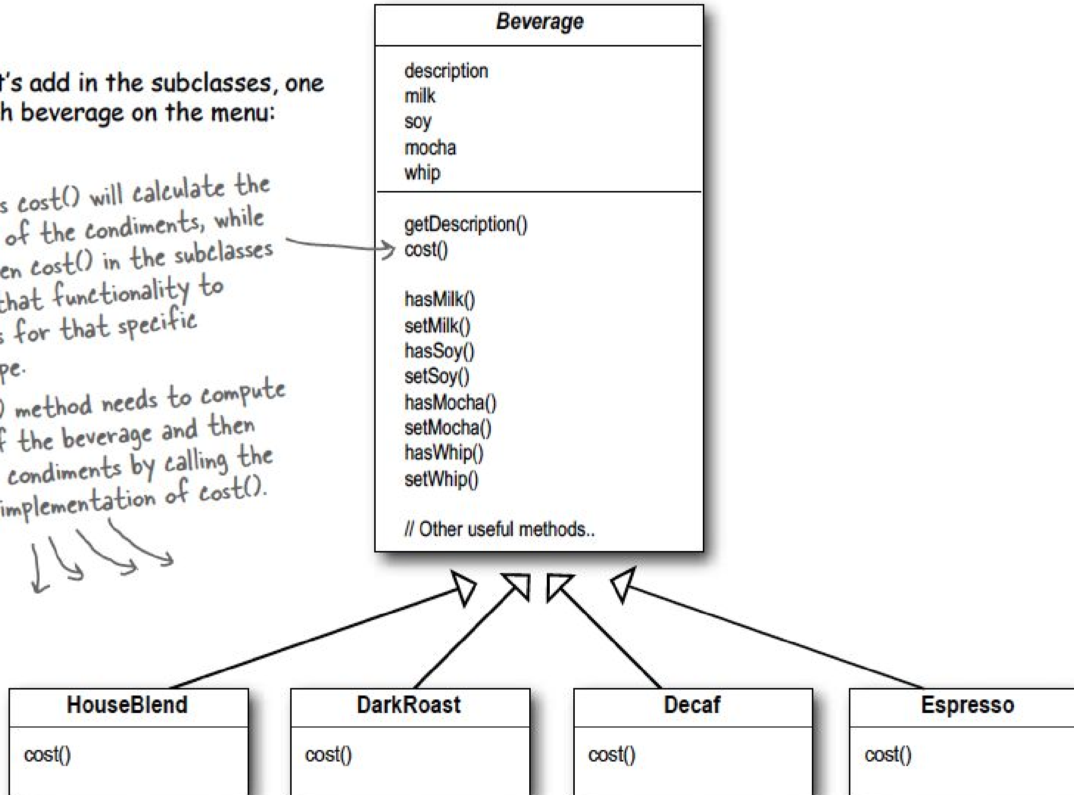
Things could get complicated with combinations of condiments



There are various of condiments

- Steamed milk, soy, mocha, whipped milk
- You charge a bit for each additional condiment
- Use subclass to handle each possibility of customer choice?

Use one class to handle them all?



If "hasSoy", add $0.10

- What if you want to provide new condiments
- What if you change cost of a condiment
- What if you have a beverage that some condiments don't apply
- What if customer what double or triple condiment?

The 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
- Make it resilient to change and flexible to take new functionality to meeting changing requirements

There are multiple OOP techniques to implement the principle.

- Observer pattern: add/remove observers

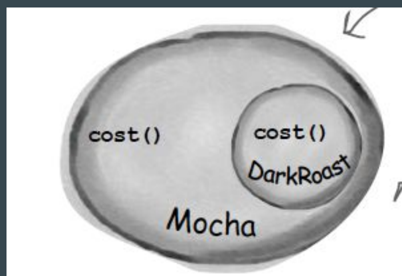
Decorator pattern

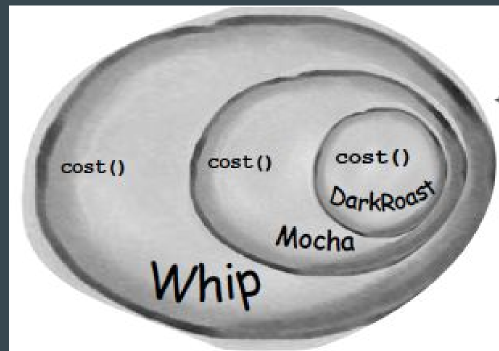
Back to the coffee condiment problem

- Start with beverage and “decorate” it with condiments at runtime

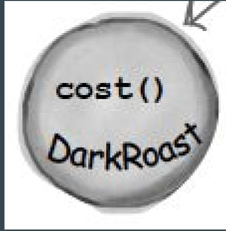
Dark Roast with Mocha and whip

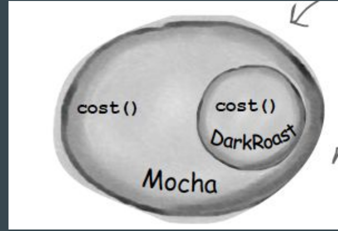
- Create DarkRoast object
- Decorate it with Mocha object
- Decorate it with Whip object
- Call cost()

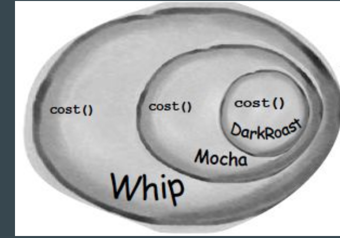


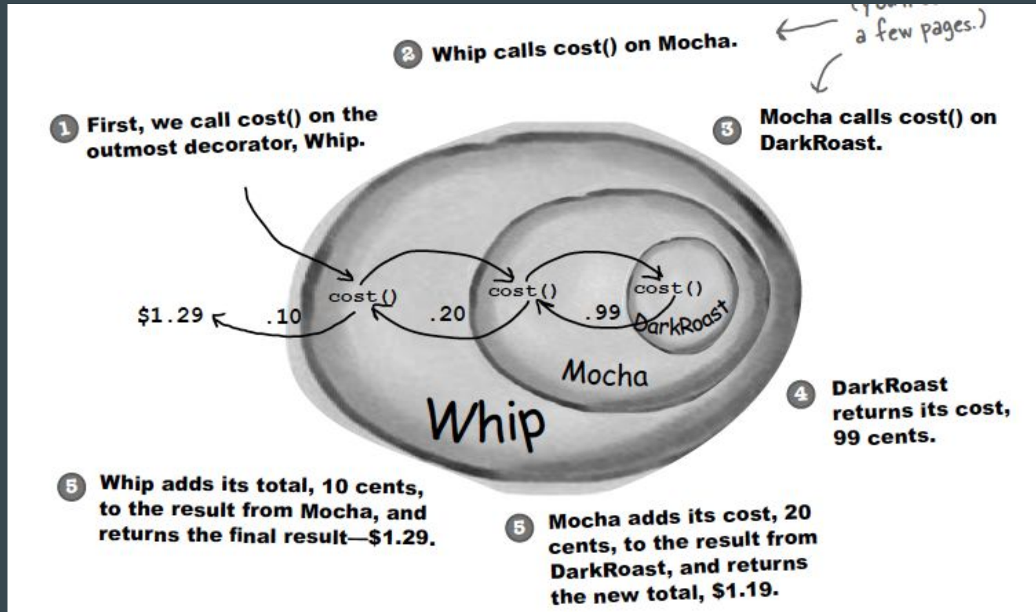


How does it work?







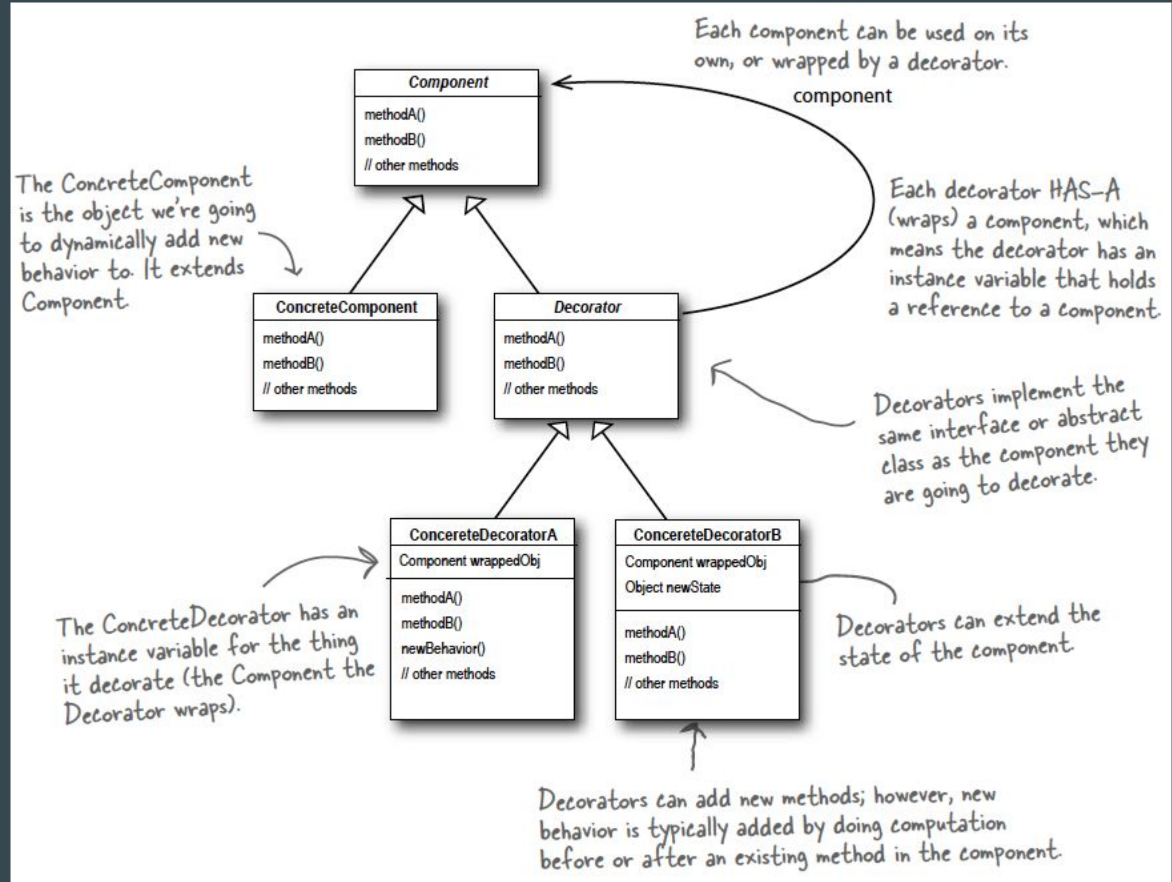


Decorators

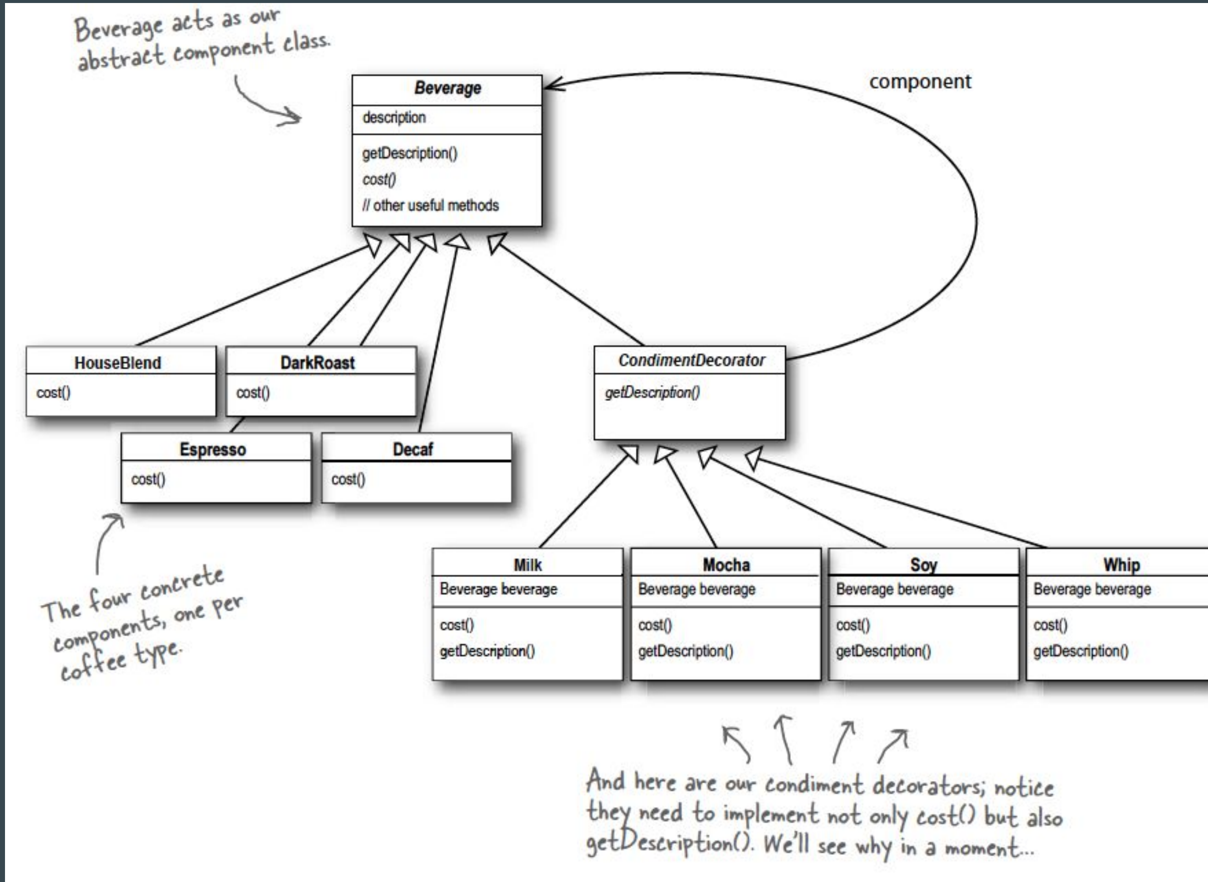
- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds 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.

Official definition

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



Decorating the beverage



How do i decide which is “component” to be decorated, and which is decorator?

Abstract Component, Abstract Decorator, Concrete component

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

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

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

Condiments: Decorator class

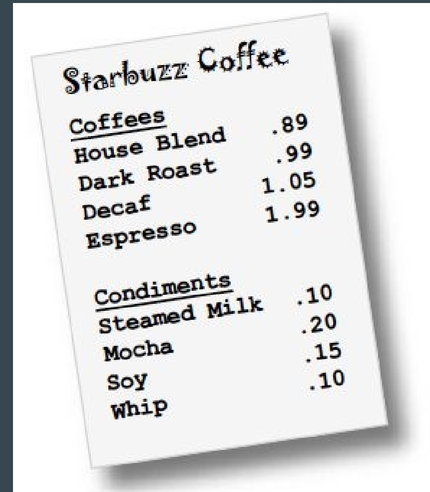
```
public class Mocha extends CondimentDecorator {
    Beverage beverage;
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
```

- Has an object of Beverage, named "beverage", which is the object to be "decorated"
- Addition 20c on top of beverage's cost
- Delegate the call to the decorated object "beverage", to compute the cost, then add the mocha cost

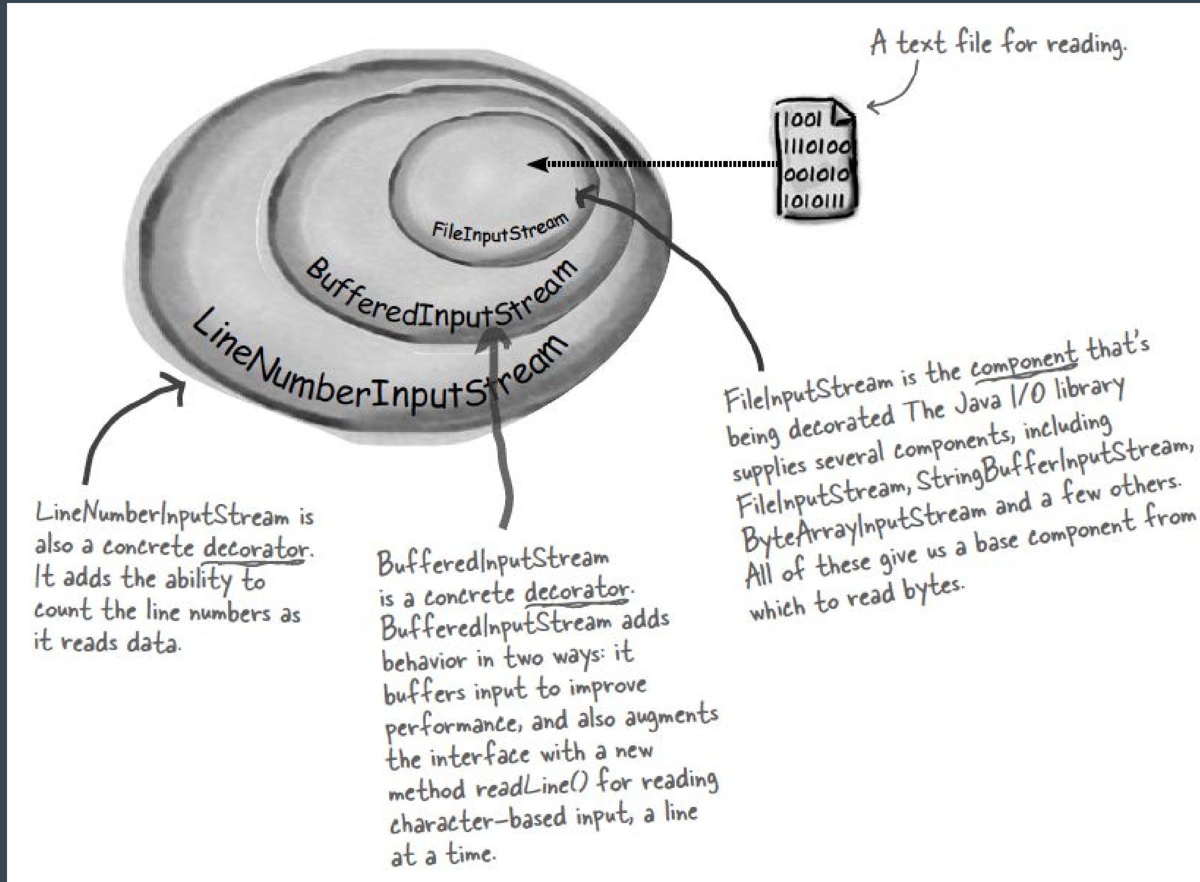
Order the coffee you want

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

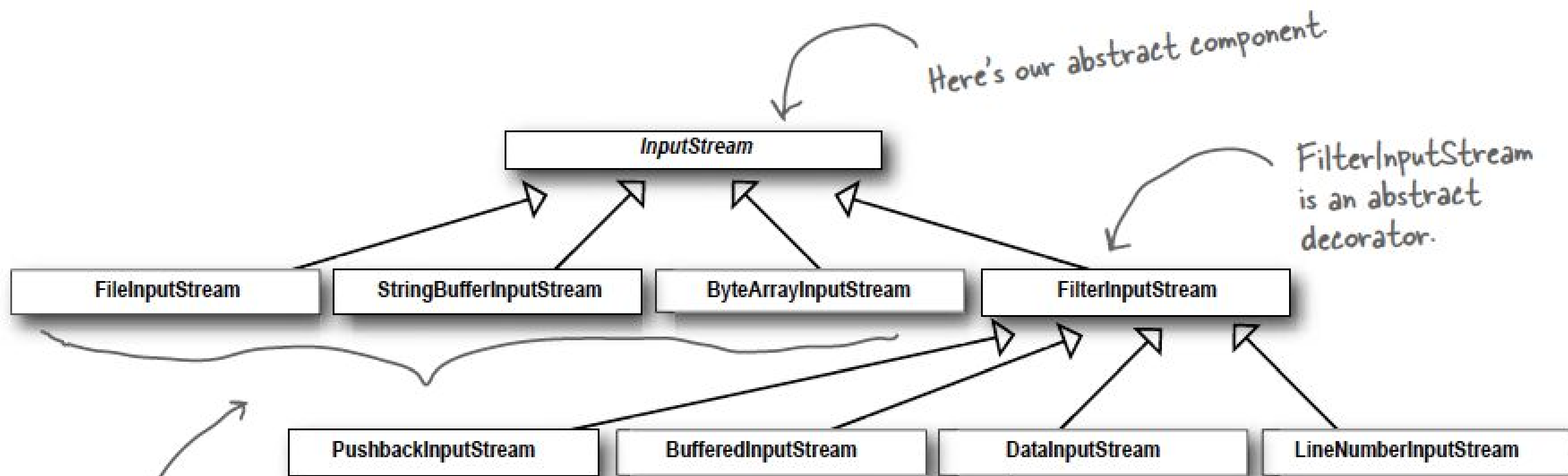
- beverage: Espresso.
- beverage2: dark roast, double mocha, whip
- Beverage3: house blend, soy, mocha, whip



Decorators are widely used in Java I/O packages



Decorating the java.io classes



I want all uppercase chars converted to lowercase from input stream

```
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;
    }
}
```

- What does super(in) do?

Test your 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();
        }
    }
}
```

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

File Edit Window Help DecoratorsRule

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