Decorator Pattern

Welcome to Starbuzz Coffee!

- Starbuzz Coffee has made a name for itself as the fastest growing coffee shop.
- Because they have grown so quickly, they are scrambling to update their ordering system to match their beverage offerings….



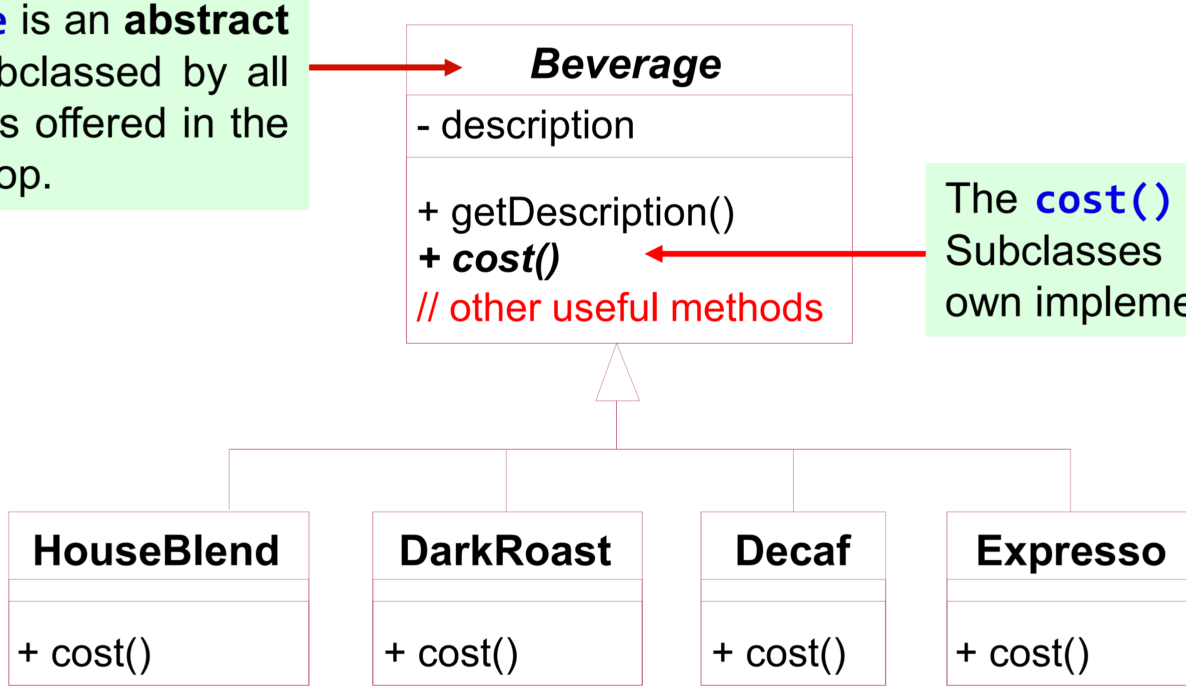


Various types of BCOFFEE

- Beverage price
  - HouseBlend: $ 0.89
  - Decaf: $ 1.05
  - Espresso: $ 1.989
  - DarkRoast: $ 0.99
- Condiment price
  - Milk: $ 0.10
  - Soy: $ 0.15
  - Mocha: $ 0.20
  - Whip: $ 0.10

The First Design of the Coffee Shop

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



The `cost()` method is **abstract**. Subclasses need to define their own implementations.

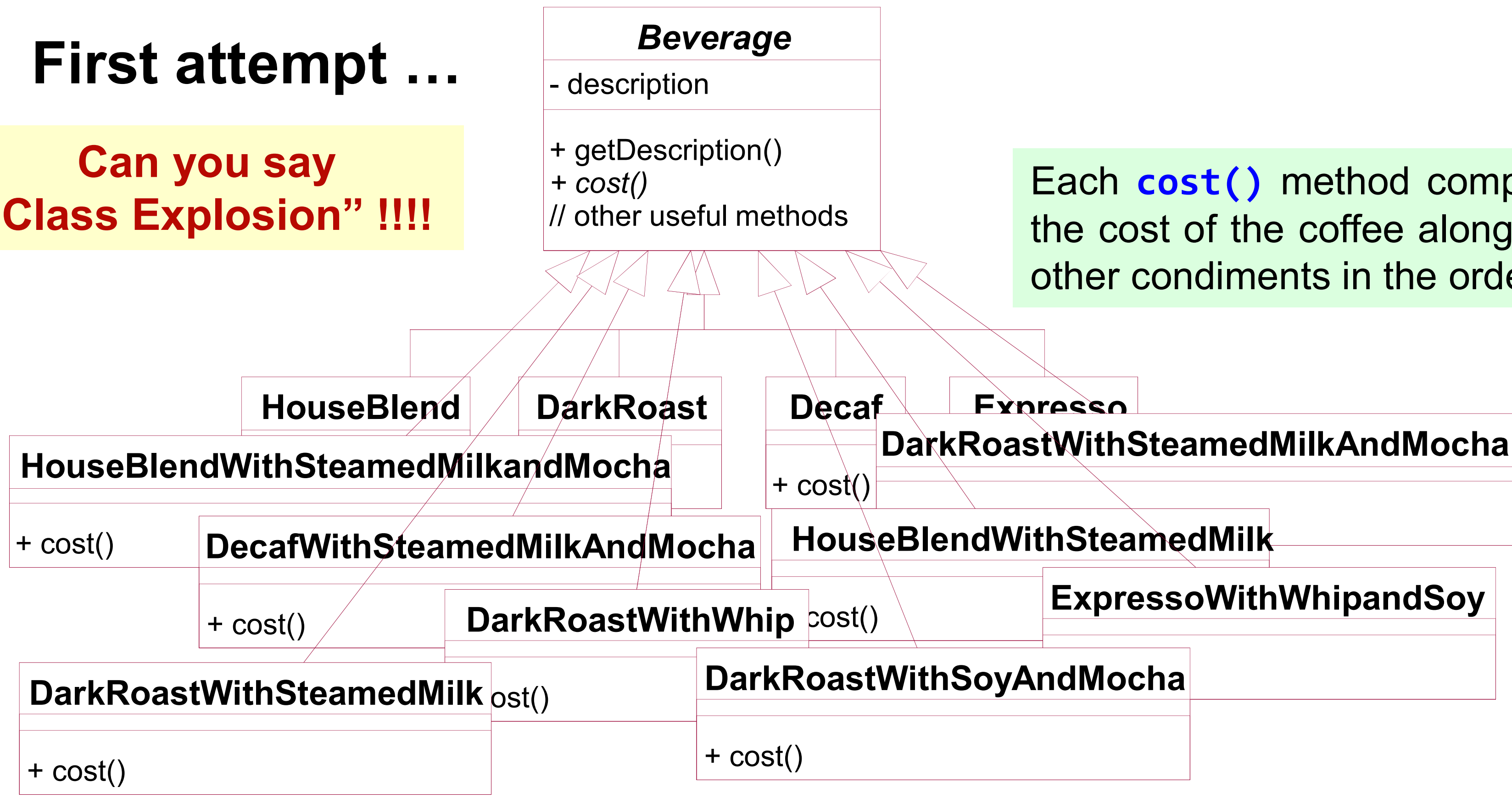
Each subclass implements `cost()` to return the cost of the beverage

Adding on …

In addition to your coffee you can also ask for several condiments like steamed milk, soy, mocha, …
Starbuzz charges a bit for each of these so they really need to get them built into the order system.

First attempt …

Can you say "Class Explosion" !!!!

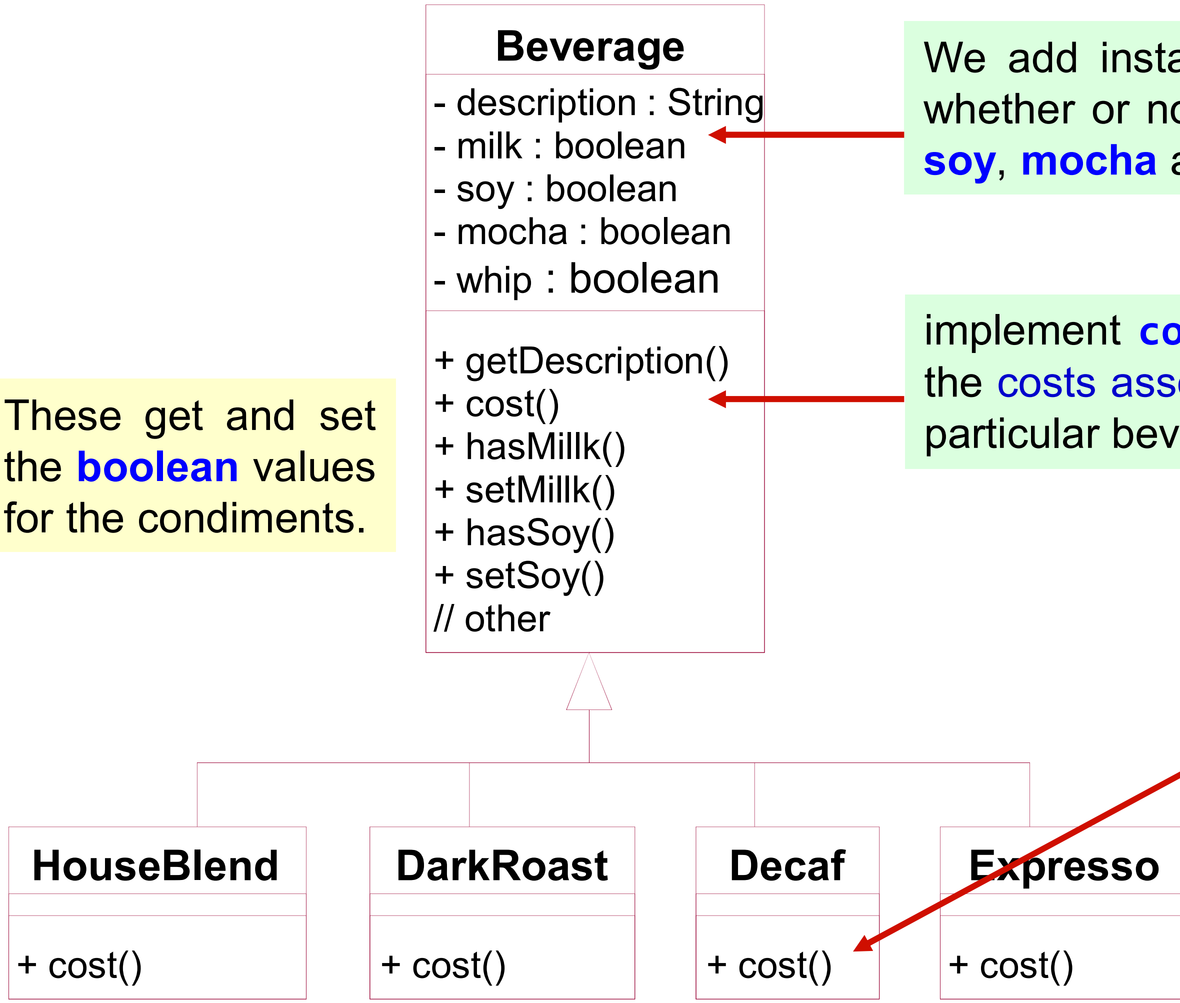


Each `cost()` method computes the cost of the coffee along with other condiments in the order.

Question

- It is pretty obvious that Starbuzz has created a maintenance nightmare for themselves.
- What happens when the price of milk goes up? Or when they add a new caramel topping?
- What OO design principle(s) are they violating here?
  - Encapsulate what varies
  - Program through an Interface not to an Implementation
  - Favor Composition over Inheritance

Alternatives to the Design?



We add instance variables to represent whether or not each beverage has **milk**, **soy**, **mocha** and **whip** ...

implement `cost()` in **Beverage** that calculate the costs associated with the condiments for a particular beverage instance.

These get and set the **boolean** values for the condiments.

Subclasses will still override `cost()`, but they will also invoke the `super` **version** so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

Sharpen your pencil

- Write **cost()** method for the following classes:

```java
public class Beverage {
  ...
  public double cost() {

    double sum = 0;
    if hasMilk() sum += 0.1;
    if hasSoy() sum += 0.15;
    if hasMocha() sum += 0.2;
    if hasWhip() sum += 0.1;

    return sum;




  }
}
```

```java
public class DarkRoast
      extends Beverage {
  public DarkRoast() {
    description =
      "More excellent Dark Roast";
  }

  public double cost() {
      return 0.99 + super.cost();





  }
}
```

Is this ok?

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

1) ***Price changes for condiments*** will force us to alter the existing code
2) ***New condiments*** will force us to add new methods and alter the `cost()` method in the **superclass**.
3) ***New beverages*** like iced tea. The iced tee class will still inherit the methods like `hasWhip()`.
4) What if a customer wants a ***double mocha***?

- **What else?**

The Open-Closed Principle

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

- Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.
- What do we get if we accomplish this?
  - Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

Meet the Decorator Pattern

- Decorating Coffee: 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:
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Decorate it with a Whip object
  - Call the cost() method and rely on delegation to add on the condiment costs.

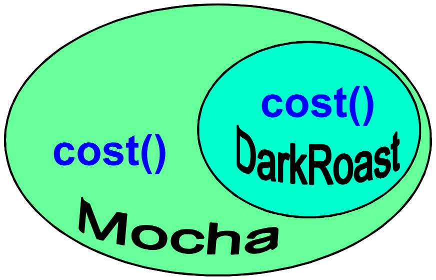
How do you ***"decorate"***
and how does delegation come into this?

Constructing a drink order with Decorators

1. Start with the `DarkRoast` object

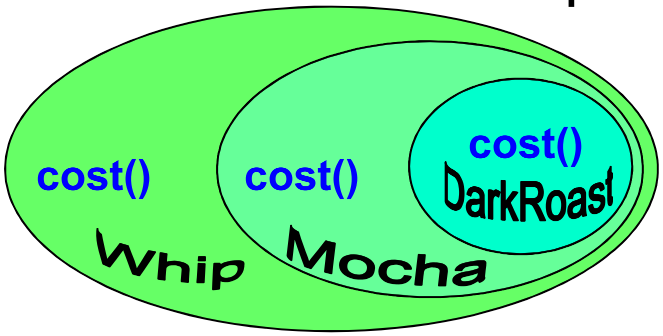
`DarkRoast` inherits from **`Beverage`** and has a **`cost()`** method that computes the cost of the drink.

2. 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`**.

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.

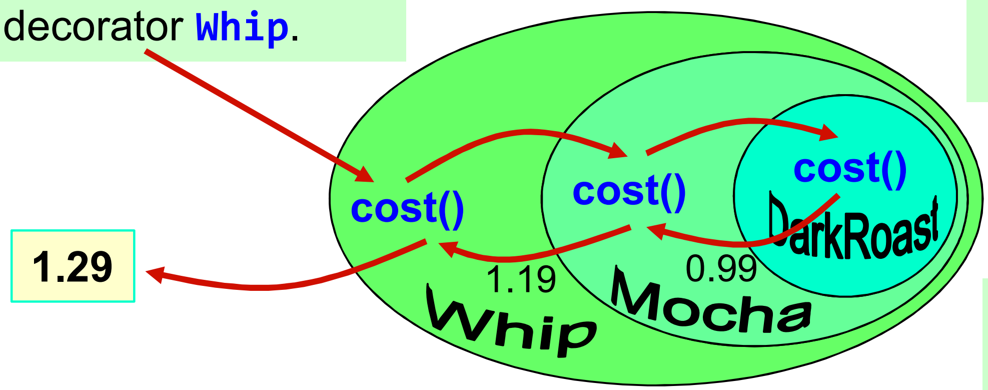
Constructing a drink order with Decorators

4. 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`**.

(1) First we call **`cost()`** on the outermost decorator **`Whip`**.

(2) **`Whip`** calls **`cost()`** on **`Mocha`**

(3) **`Mocha`** calls **`cost()`** on **`DarkRoast`**



(4) **`DarkRoast`** returns its cost, 99 cents

(5) **`Mocha`** adds its cost, 20 cents, to the result 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.

So what do we know so far?

- Decorators have the same supertype as the objects that 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 object.
- The decorator adds its own behavior either before and/or after delegating to the object its decorates to do the job.
- Objects can be decorated at any time, so we can decorate objects at runtime with as many decorators as we like.

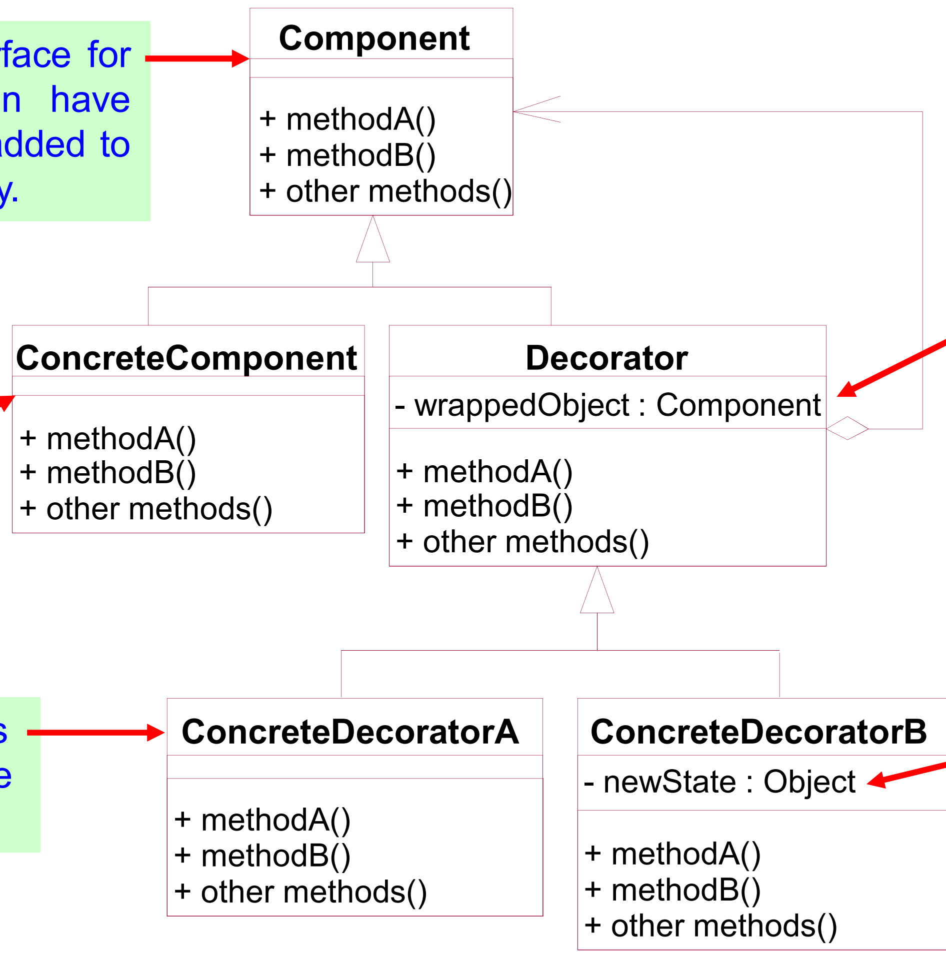
Key point!

Decorator Pattern Defined

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

defines the interface for objects that can have responsibilities added to them dynamically.

defines an object to which additional responsibilities can be attached.

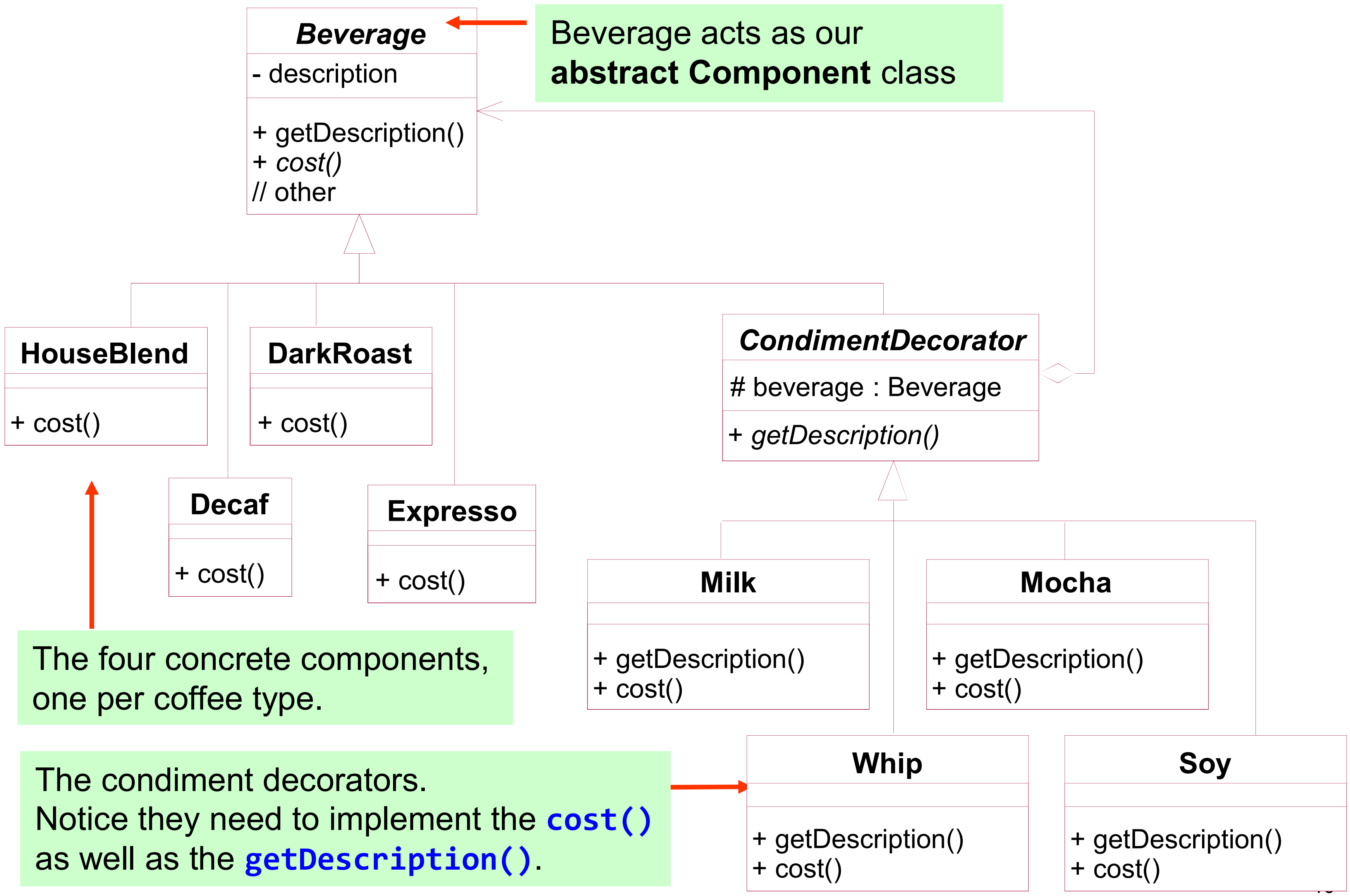


maintains a reference to a **Component** object and implement the same interface as the component they are going to decorate.

Decorators can adds responsibilities to the component.

can extend the state of the component

Decorate the Beverages!



Some Real Code!

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

   public abstract double cost();
}
```

Beverage is an **abstract** class. **getDescription()** method is already implemented, but we need to implement **cost()** method in the subclasses.

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

```
public abstract class CondimentDecorator extends Beverage {
   protected Beverage beverage;
   public CondimentDecorator (Beverage beverage) {
      this.beverage = beverage;
   }

   public abstract String getDescription();
}
```

require that the condiment decorators reimplement the **getDescription()** method.

Coding Beverages

```
public class DarkRoast extends Beverage {
   public DarkRoast() {
      description = "Dark Roast Coffee";
   }
   public double cost() {
      return .99;
   }
}
```

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

Coding Condiments

```
public class Mocha extends CondimentDecorator {

   public Mocha(Beverage beverage) {
      super(beverage);
   }

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

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

We want our description to say not only **DarkRoast** -- but to also include the item decorating each beverage for instance: **DarkRoast**, **Mocha**.

Similarly, to compute the cost of the beverage with **Mocha**, we first **delegate** to the object that is being decorated, so that we can compute its cost and then add in the cost of the **Mocha**.

Ordering Coffee

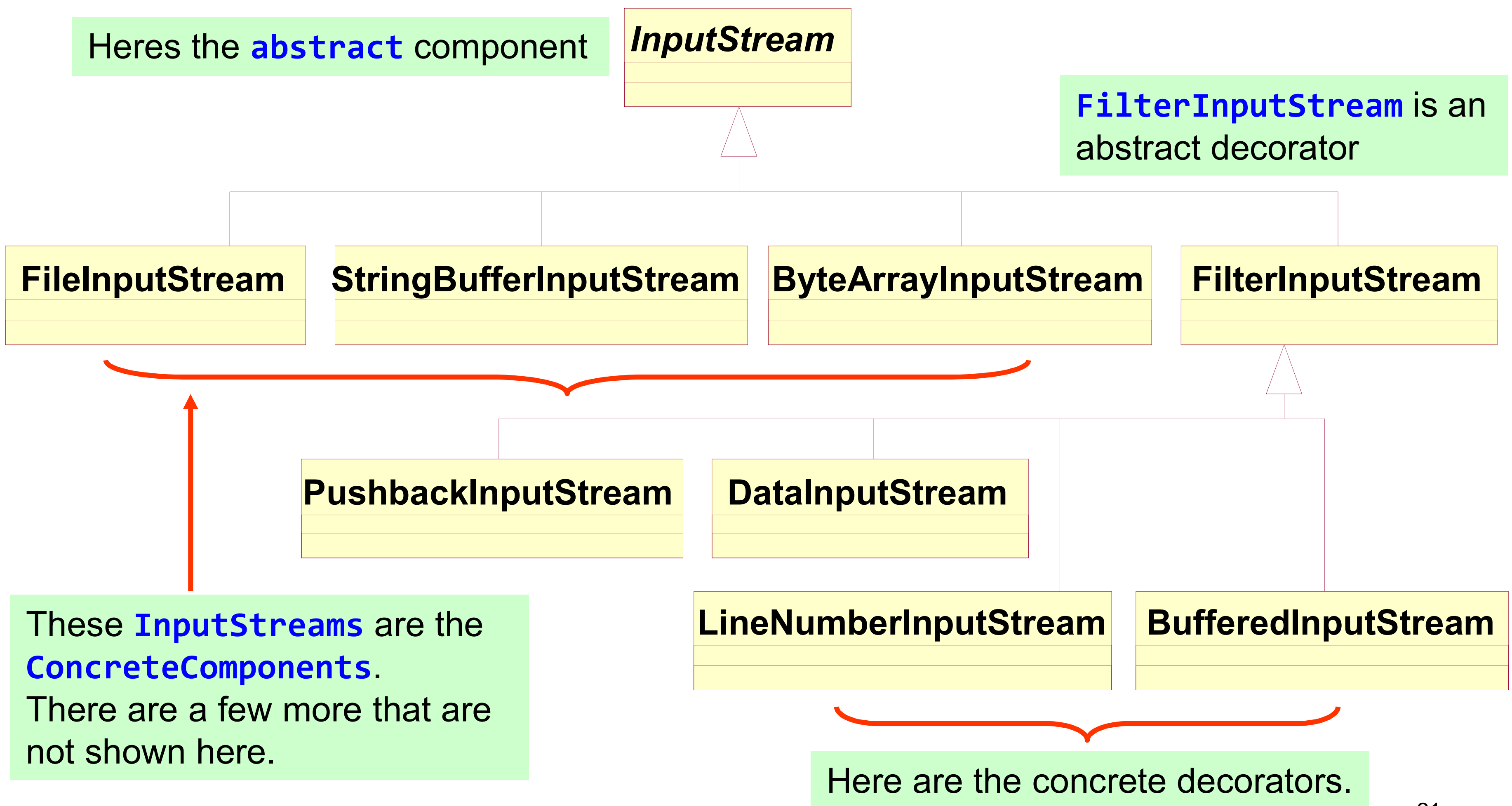
```java
public class StarbuzzCoffee {
  public static void main(String args[]) {
    Beverage beverage1 = new Espresso();
    System.out.println(beverage.getDescription()
                       + " $" + beverage.cost());
    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
                       + " $" + beverage2.cost());
  }
}
```

Output:

```
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
```

Real World Decorators

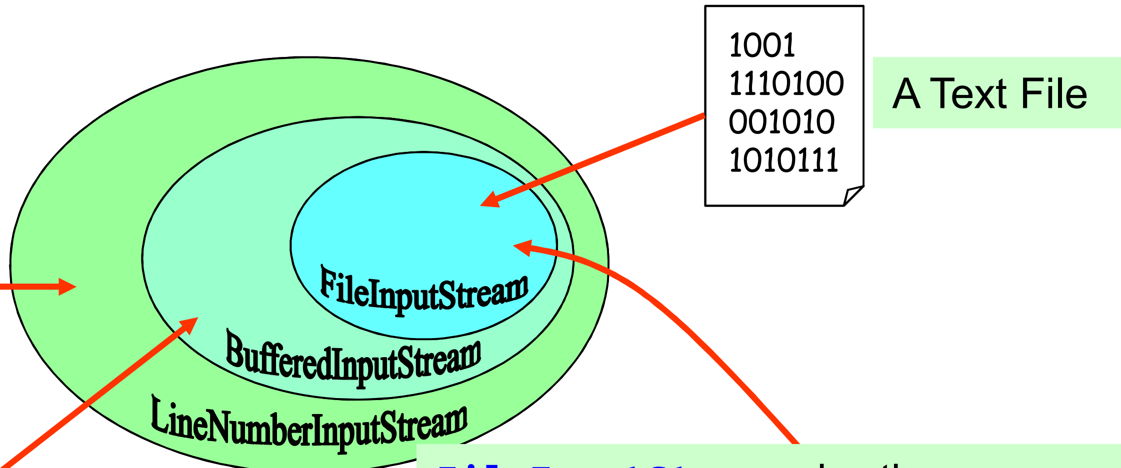
- The **java.io** package uses the Decorator pattern!



The java.io Package (contd.)

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

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



`BufferedInputStream` is a <u>concrete</u> *<u>Decorator</u>*. **`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 *<u>component</u>* 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.

Exercise your brains…

- How would you write a decorator that converts all uppercase characters to lowercase in the input stream?

Writing your own Java I/O Decorator

```java
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;
  }
}
```

extend the `FilterInputStream` abstract decorator for all InputStreams

implement two read methods. They take a byte (or an array of bytes) and convert each byte to lowercase.

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.

Code Demo

- Read a plain text file and compress it using the GZIP format GZIP.java
- Read a compress file in the GZIP format and write it to a plain text file UNGZIP.java

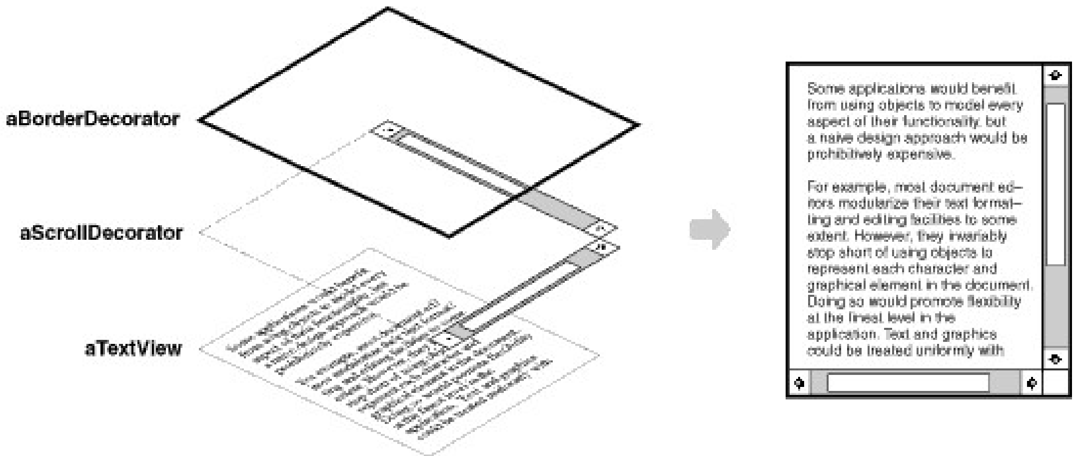
Compress text file

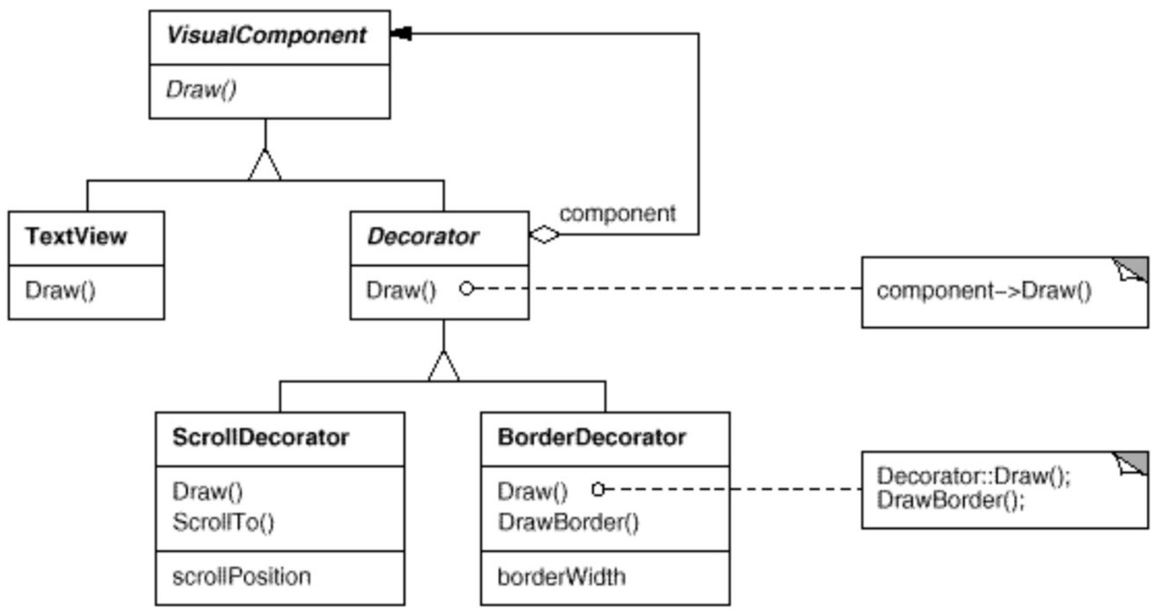
```
// Open the input file
String inFilename = "iliad10.txt";
FileInputStream input = new FileInputStream(inFilename);

// Open the output file
String outFilename = "iliad10.gz";
GZIPOutputStream out = new GZIPOutputStream(
        new FileOutputStream(outFilename));

// Transfer bytes from the output file to the compressed file
byte[] buf = new byte[1024];
int len;
while ((len = input.read(buf)) > 0) {
  out.write(buf, 0, len);
}

// Close the file and stream
input.close();
out.close();
```

Decompress file

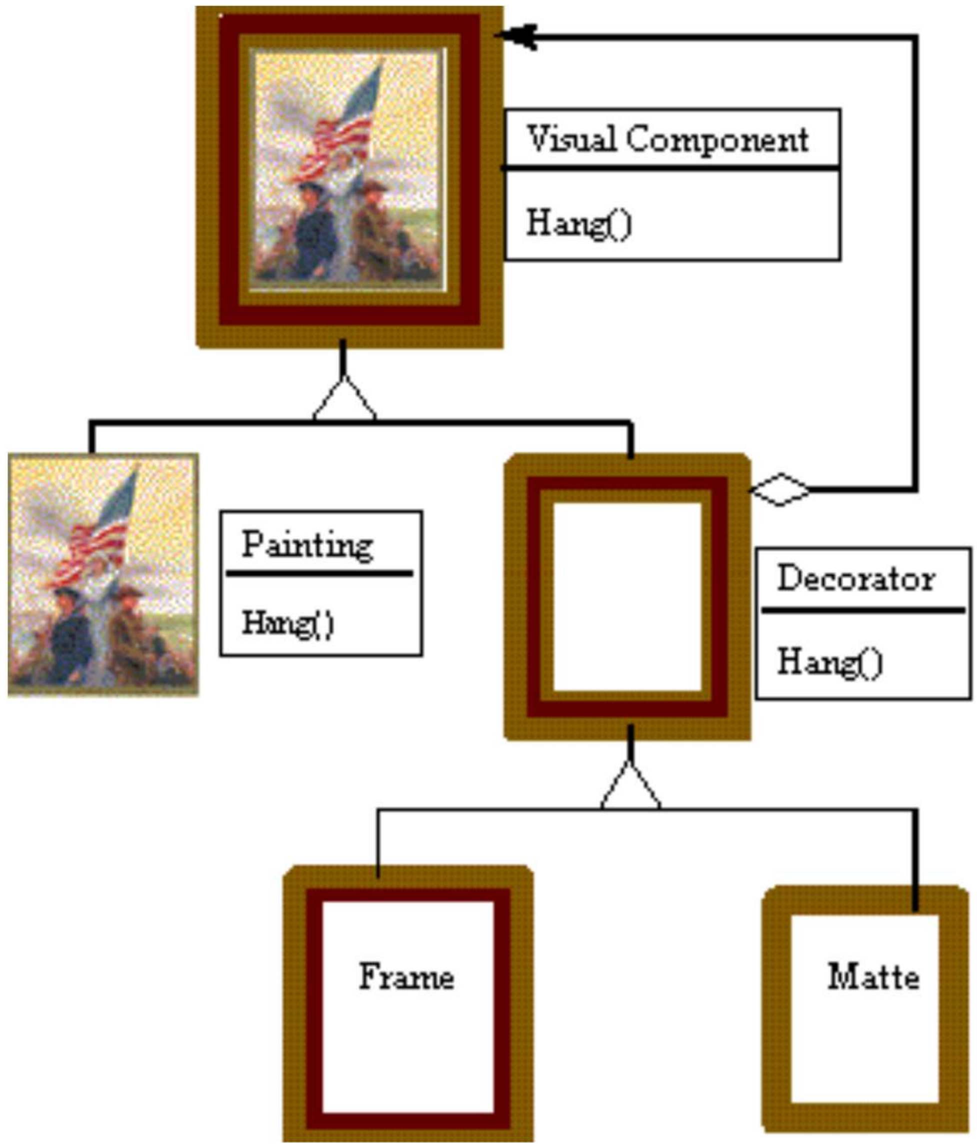
```java
// Open the gzip file
String inFilename = "iliad10.gz";
GZIPInputStream gzipInputStream =
        new GZIPInputStream(new FileInputStream(inFilename));
// Open the output file
String outFilename = "TheIliadByHomer";
OutputStream out = new FileOutputStream(outFilename);

// Transfer bytes from the compressed file to the output file
byte[] buf = new byte[1024];
int len;
while ((len = gzipInputStream.read(buf)) > 0) {
  out.write(buf, 0, len);
  for (int i = 0; i < len; i++)
    System.out.print((char) buf[i]);
  System.out.println();
}
// Close the file and stream
gzipInputStream.close();
out.close();
```

Decorating Text





Decorator – Non Software Example



The Constitution of Software Architects

- Encapsulate what varies
- Program through an interface not to an implementation
- Favor Composition over Inheritance
- **Classes should be open for extension but closed for modification**
- ?????????
- ?????????
- ...

Decorator Advantages/Disadvantages

- ++
  - Provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class
  - Allows to customize a class without creating subclasses high in the inheritance hierarchy.
- --
  - A `Decorator` and its enclosed component are not identical. Thus, tests for object types will fail.
  - `Decorators` can lead to a system with “lots of little objects” that all look alike to the programmer trying to maintain the code

Summary

- Decorator patterns are based on the open-closed principle!
  - We should allow behavior to be extended without the need to modify existing code.
- The Decorator Pattern
  - Provides an alternative to subclassing for extending behavior.
  - Involves a set of decorator classes that are used to wrap concrete components
  - Decorator classes mirror the types of the components they decorate.
  - Decorators change the behavior of their components by adding new functionality before and/or after method calls to the component.
  - You can wrap a component with any number of decorators.
  - Decorators are typically transparent to the client of the component -- unless the client is relying on the component's concrete type.
  - Decorators can result in many small objects in our design, and overuse can be complex!