CMPSCI 220 Programming Methodology

07: Decorator, Singleton, & Factory Pattern

Objectives

Decorator Pattern

Learn how objects can gain new responsibilities at runtime.
Apply the decorator pattern in Scala.

Singleton Pattern

Learn how objects can be represented as singleton objects. We know how to do this in Scala. What does it look like in Java?

Factory Pattern

We have been introduced to the factory pattern already, but we didn't make it obvious. We will see how Scala handles this and what it looks like in Java.

i-clicker Question!

What design principle is applied to objects with the observer pattern?

A. Objects are tightly coupled to improve flexibility
B. Objects are loosely coupled to improve flexibility
C. Objects are combined to improve flexibility
D. Objects are eliminated to reduce complexity
E. I am not sure

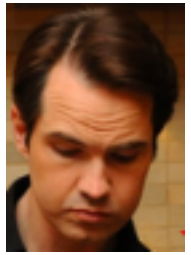
Welcome to Starbuzz Coffee!

*This is **Milton Waddams**. He didn't take CMPSCI 220 at UMass Amherst. He thinks he is awesome, but he is not. He is the mastermind behind the poorly designed ordering system at Starbuzz. He can't code himself out of a paper bag. He really likes staplers.*

*This is **Peter Gibbons (aka Jimmy)**. He is a sad man because he is upset with the developer of Starbuzz's ordering system. Every time there is a new beverage it takes **forever** for the beverage to become available in their system. Not to mention the system is sooo slow!*

Starbuzz Ordering System Design

"Oh Yeah! This is a good design!"



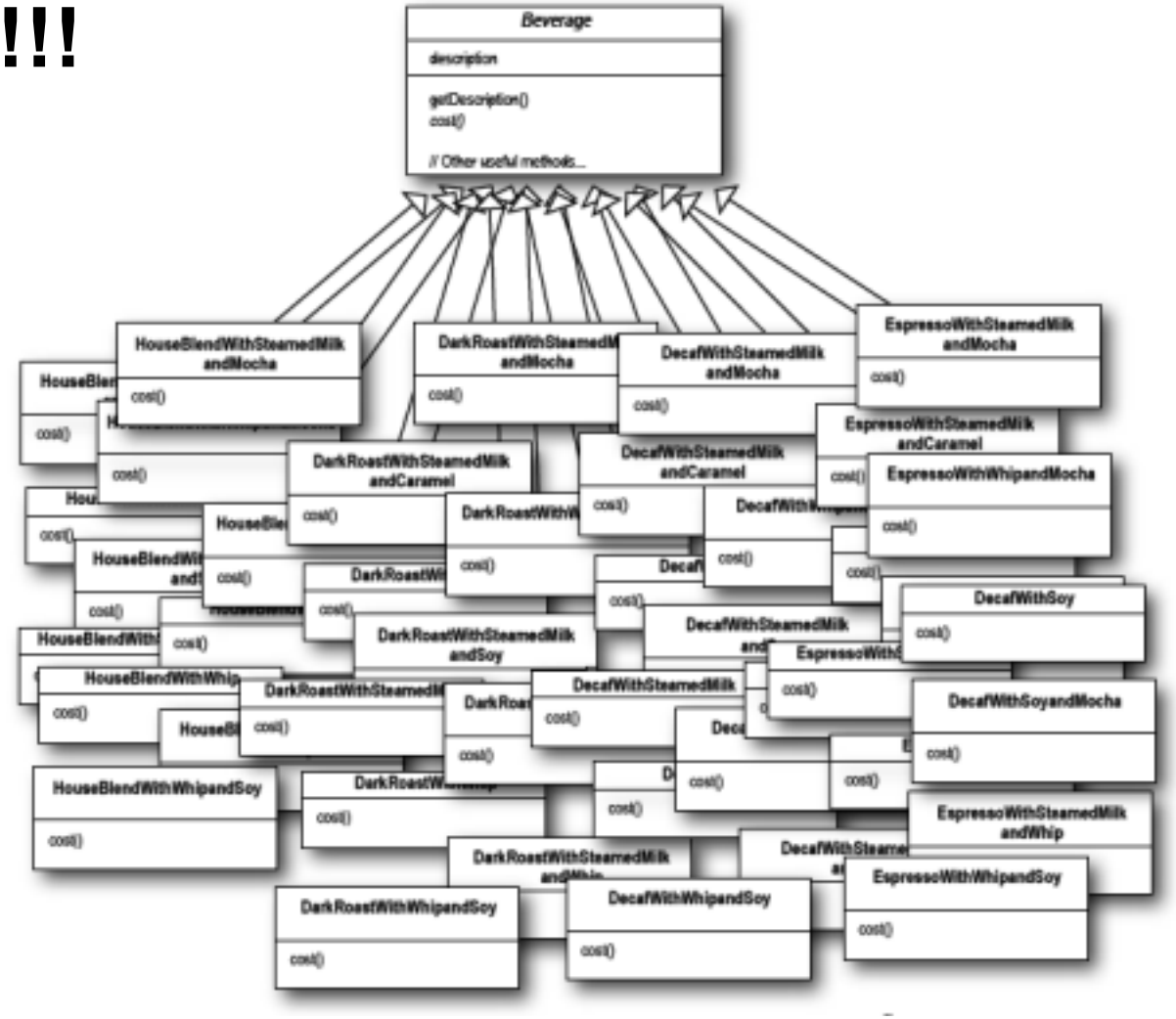
Milton, you SUCK!

In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha, and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them build into their ordering system.

How might this be done in this design?

Class Explosion!!!

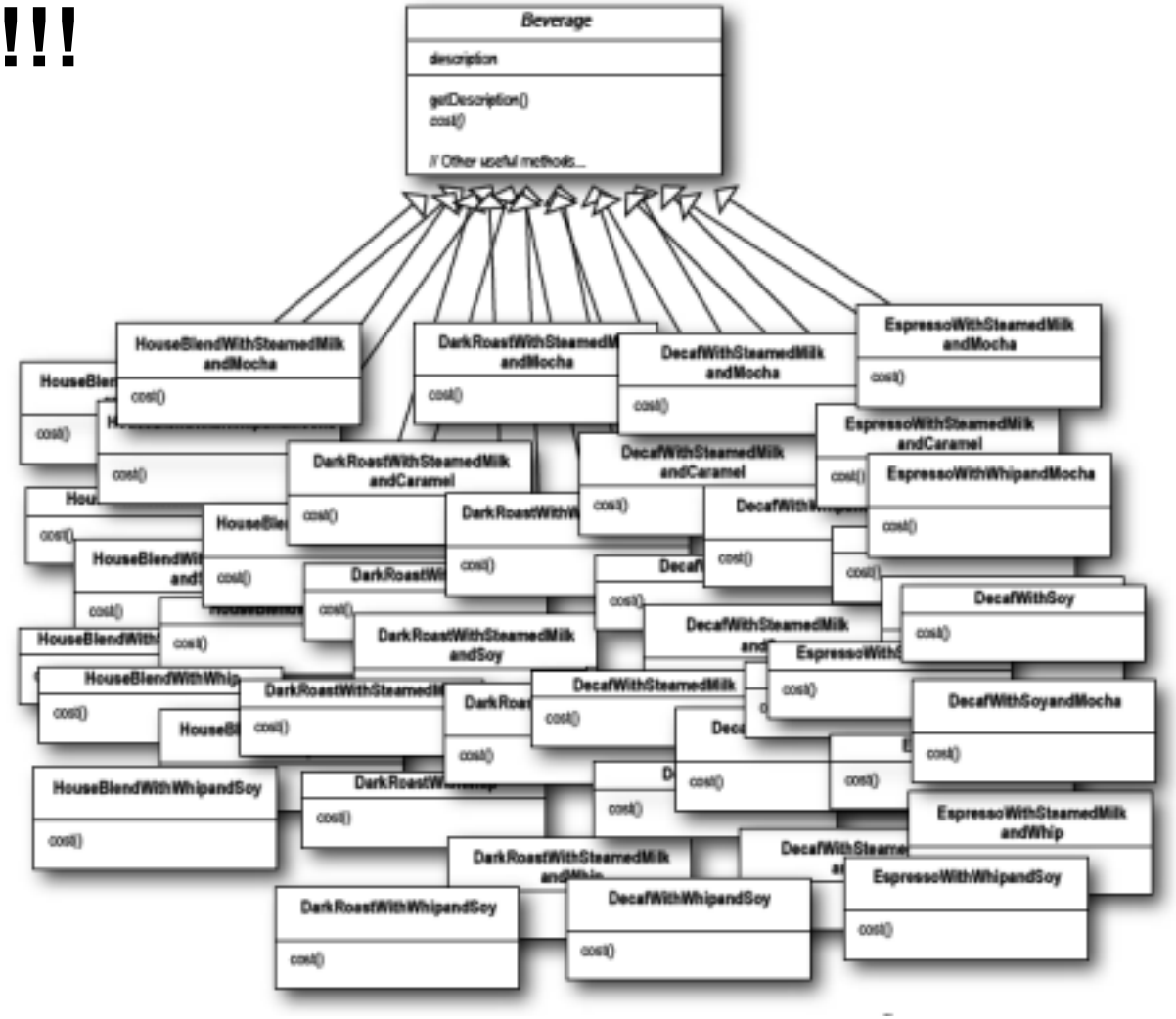
"So, what is your point..."



Class Explosion!!!

"You are FIRED! And leave the stapler!"

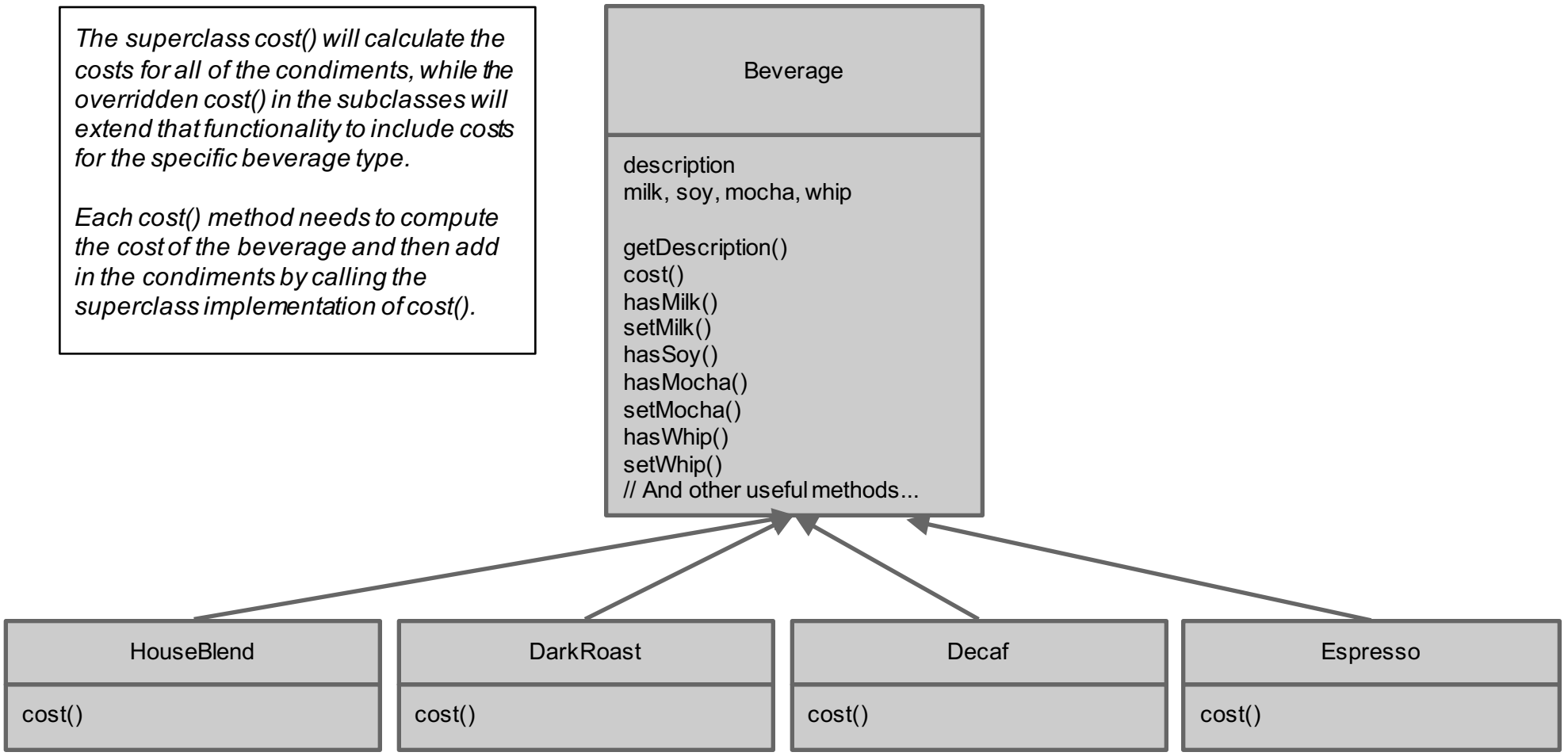
How can we fix this?



Starbuzz Ordering System Design Take 2

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 the 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().



Your Turn!

Write the cost() methods for the two classes below (pseudo-Scala is okay)

```
class Beverage {
  def cost: Double = {




  }
}
```

```
class DarkRoast extends Beverage {
  val description = "Most Excellent Dark Roast"
  def cost: Double = {




  }
}
```

What Might 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 of 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 or espresso?

The Open-Closed Principle

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

How does this relate to our existing ordering system design?

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

1. Take a DarkRoast object.

2. Decorate it with a Mocha object.

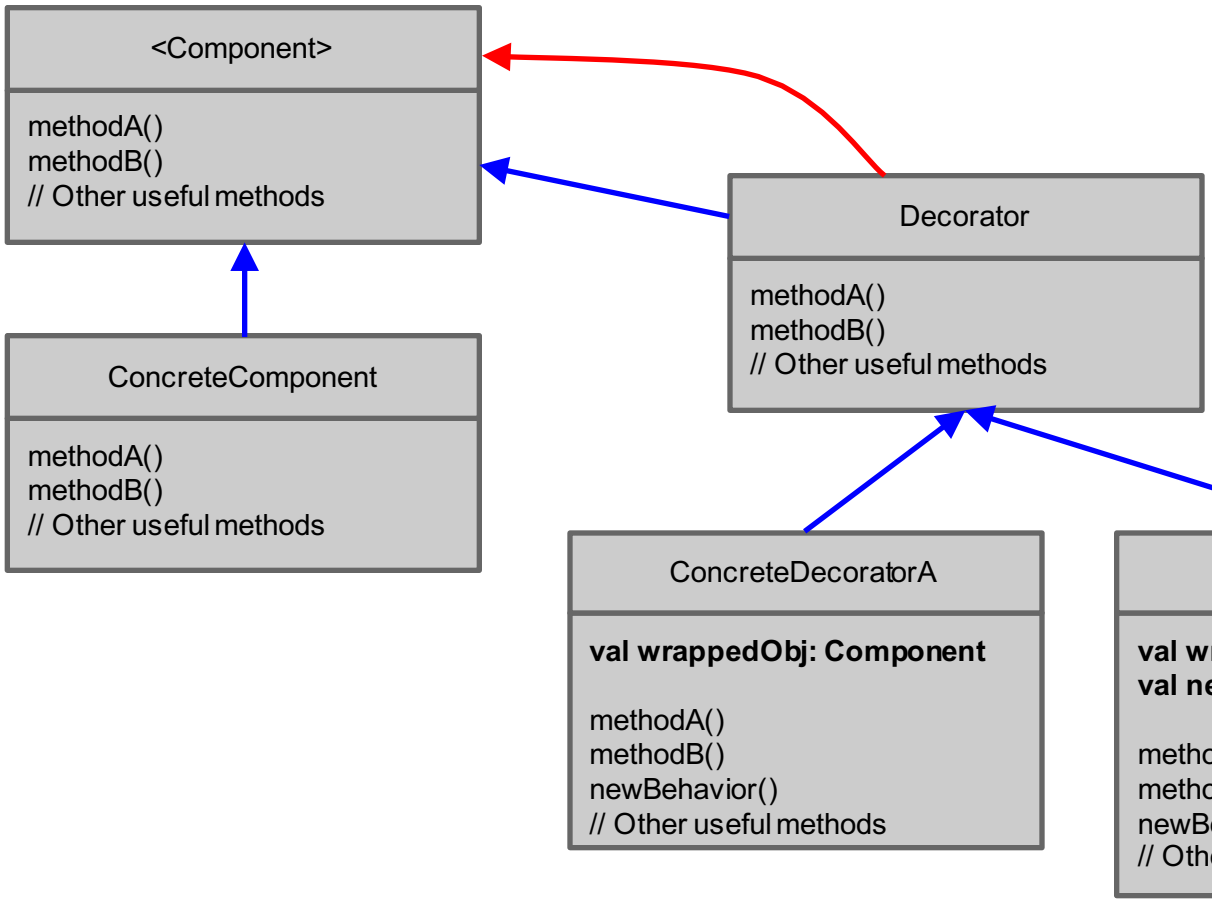
3. Decorate it with a Whip object.

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.

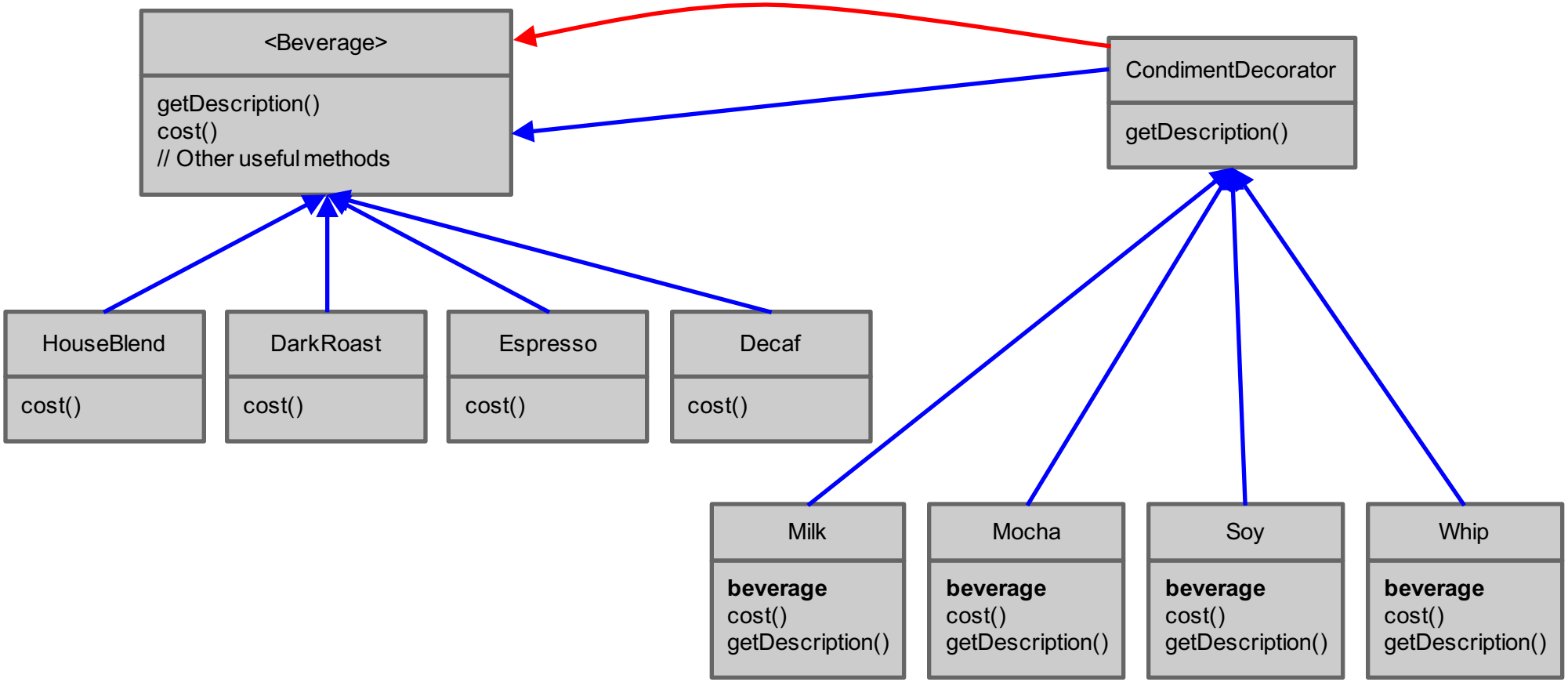
Decorator Class Diagram



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

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

Starbuzz's New Design



Implementing the Starbuzz Application

src/main/scala/cs220/Decorator.scala

Singleton Pattern

It is often the case that we desire only a single instance of a class. This single instance is referred to as a *singleton object*.

Singleton Use Case: Databases

In many real-world applications a database is necessary for storing large amounts of data.

Typically, we are interacting with only a single database instance. That is, we do not want to create a “new instance” of a database each time we interact with it. Rather, we want to interact with the same database.

Singleton in Java: Interface

```
package java.singleton;

public interface Database {
    public String get(String key);
    public String set(String key, String value);
}
```

Singleton in Java: Implementation

```
public class MyDatabase implements Database {
    private Map<String, String> db;

    private MyDatabase() {
        db = new HashMap<>();
    }

    @Override
    public String get(String key) {
        return db.get(key);
    }

    @Override
    public String set(String key, String value) {
        db.put(key, value);
        return value;
    }
    ...
```

Singleton in Java: Implementation

```
public class MyDatabase implements Database {
   private Map<String, String> db;

   …
   …

   private static MyDatabase instance = null;

   public static MyDatabase getInstance() {
     if (instance == null) {
         instance = new MyDatabase();
     }
     return instance;
   }
}
```

Implementing the Singleton Pattern

src/main/java/j/singleton/*.java

Singleton in Scala: Trait

```
package scala.singleton

trait Database {
  def get(key: String): String
  def set(key: String, value: String): String
}
```

Singleton in Scala: Implementation

```
object MyDatabase extends Database {
  private var db = Map[String, String]()

  override def get(key: String): String = db(key)

  override def set(key: String, value: String): String = {
    db = db + (key -> value)
    value
  }
}
```

Implementing the Singleton Pattern

src/main/scala/s/singleton/*.scala

Factory Pattern

The *factory pattern* is a *creational pattern* which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created.

This is done by creating objects by calling a factory method that creates the object and returns it as an abstract class or interface type.

This allows for easily trading in different implementations without affecting client code that calls the factory method.

Factory Pattern Use Case

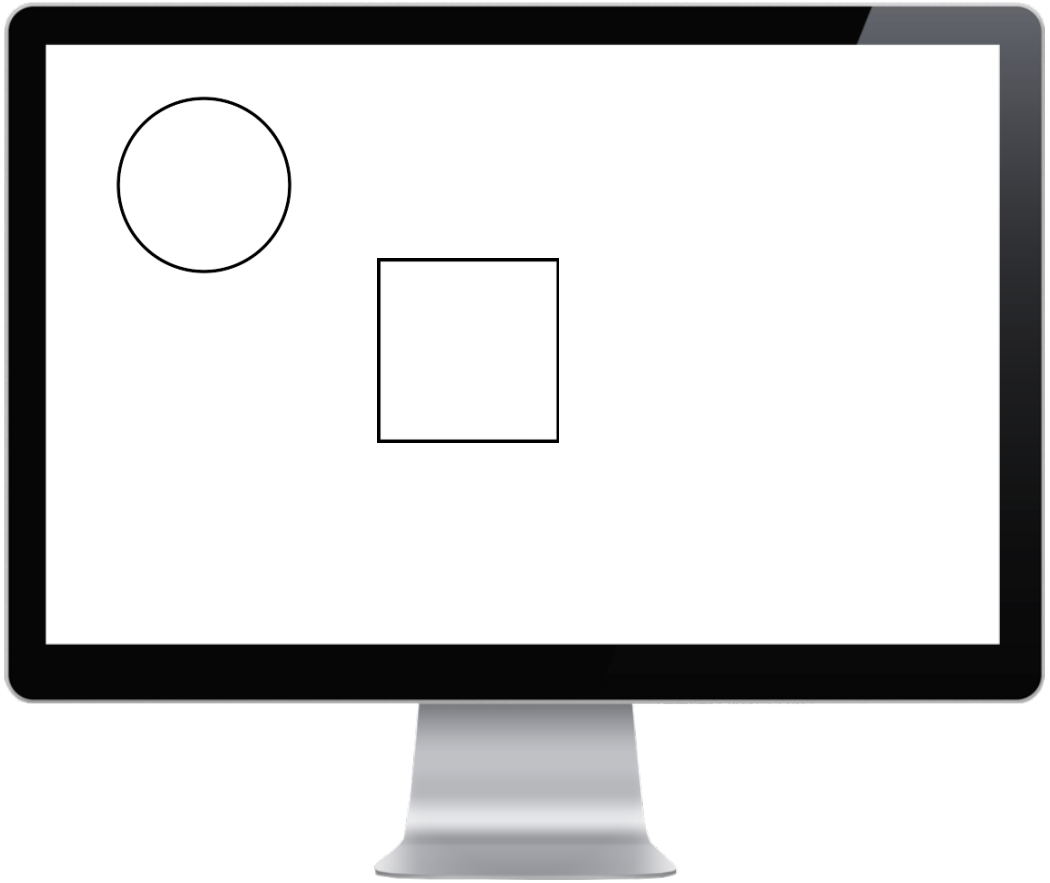
Imagine we want to create an application that draws shapes to the screen.

In addition, we want to be able to add new shape implementations to our application without affecting the code that uses the shapes.

Factory Example

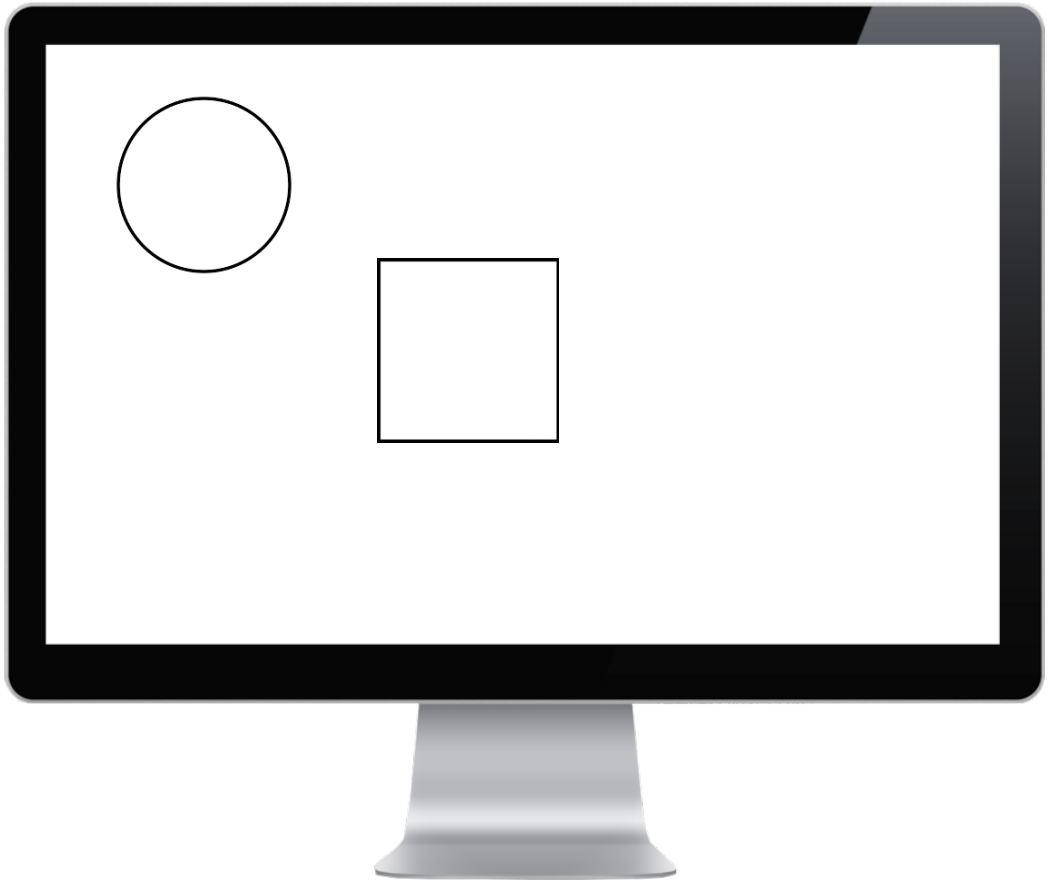
```
val s1 = new Circle
s1.draw
```

Factory Example



```
val s1 = new Circle
s1.draw
val s2 = new Square
s2.draw
```

Factory Example



```
val s1 = new Circle
s1.draw
val s2 = new Square
s2.draw
```

What if we wanted our circles to be drawn differently?

Factory Example

```
val s1 = new Circle
val s1 =
  new BrushedCircle
s1.draw
val s2 = new Square
s2.draw
```

What if we wanted our circles to be drawn differently?

Factory Example

~~val s1 = new Circle~~

```
val s1 =
  new BrushedCircle
s1.draw
val s2 = new Square
s2.draw
```

What if we wanted our circles to be drawn differently?

What is the problem here?

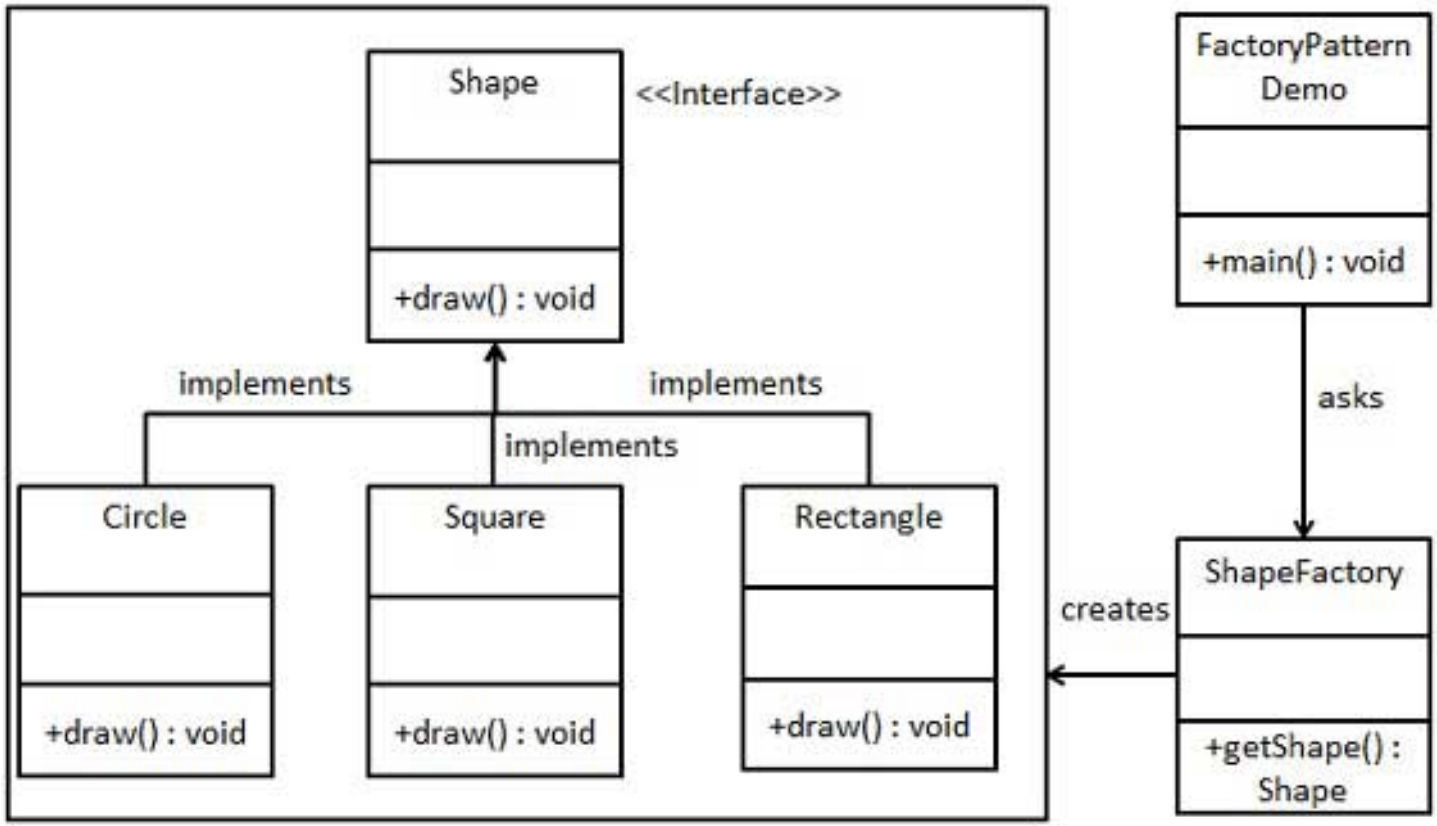
Factory Example

```
val s1 = new Circle
val s1 =
  new BrushedCircle
s1.draw
val s2 = new Square
s2.draw
```

What if we wanted our circles to be drawn differently?

Is there a way to improve on this design so we need not change the “client” code?

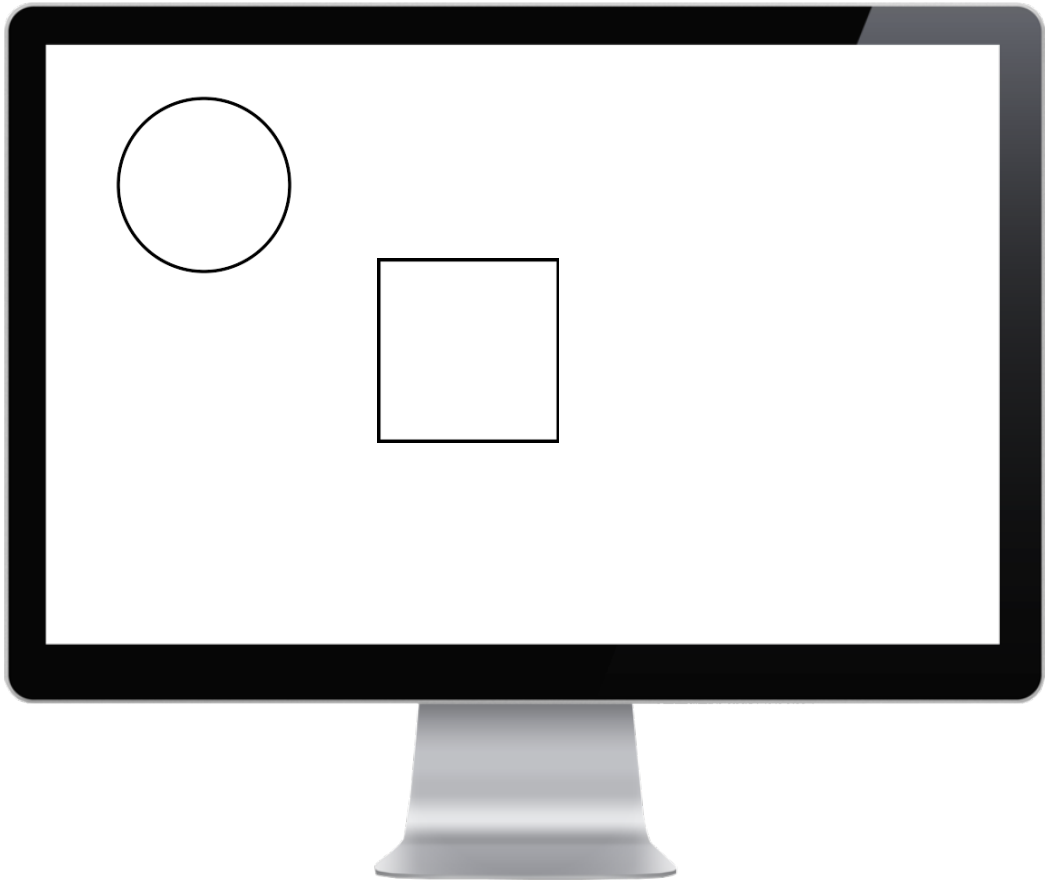
Factory UML Diagram



Factory Example

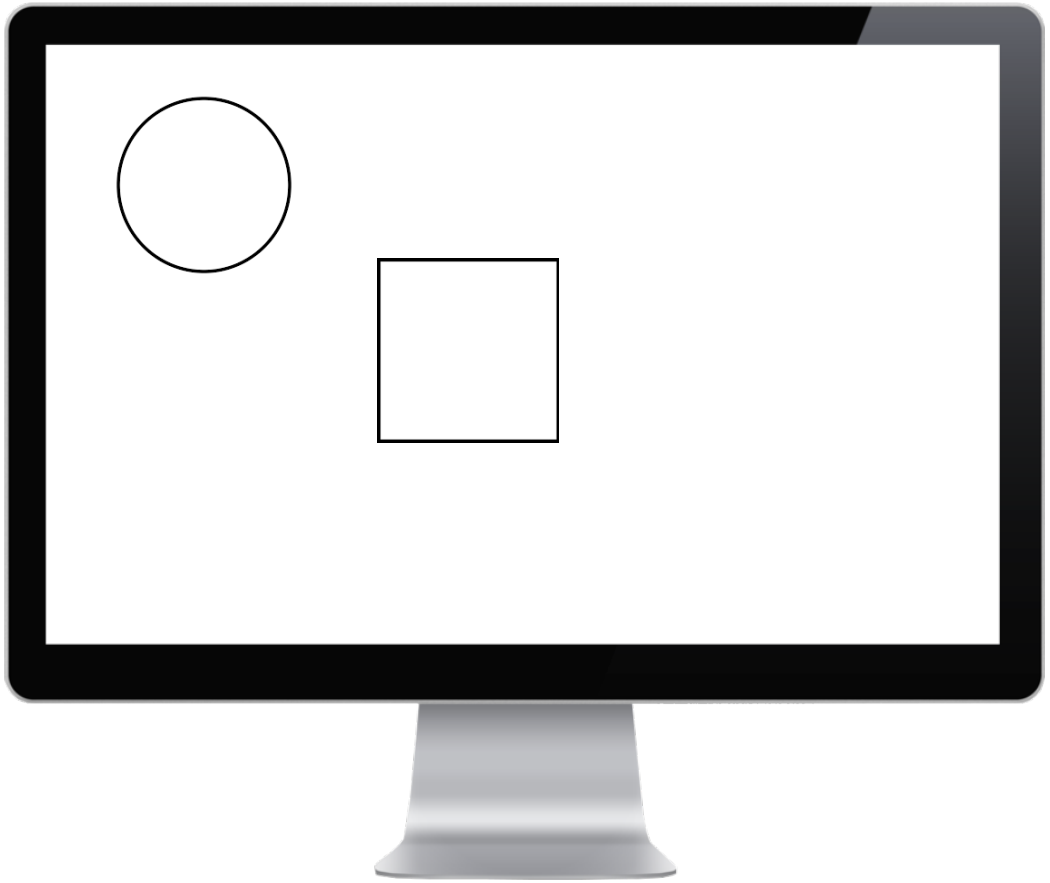
```
val f = new Factory
val s1 = f.shape(“circle”)
s1.draw
```

Factory Example



```
val f = new Factory
val s1 = f.shape(“circle”)
s1.draw
val s2 = f.shape(“square”)
s2.draw
```

Factory Example



```
val f = new Factory
val s1 = f.shape(“circle”)
s1.draw
val s2 = f.shape(“square”)
s2.draw
```

Now, imagine we change the implementation of our circle.

Factory Example

```
val f = new Factory
val s1 = f.shape(“circle”)
s1.draw
val s2 = f.shape(“square”)
s2.draw
```

Now, imagine we change the implementation of our circle.

This will not change the client code requesting shapes.

Implementing the Factory Pattern

src/main/scala/j/factory/*.java
src/main/scala/s/factory/*.scala