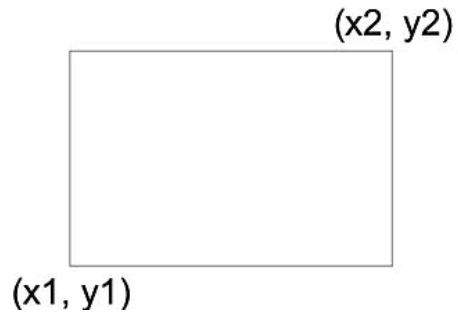
Tecniche di Progettazione: Design Patterns

GoF: Bridge

The problem

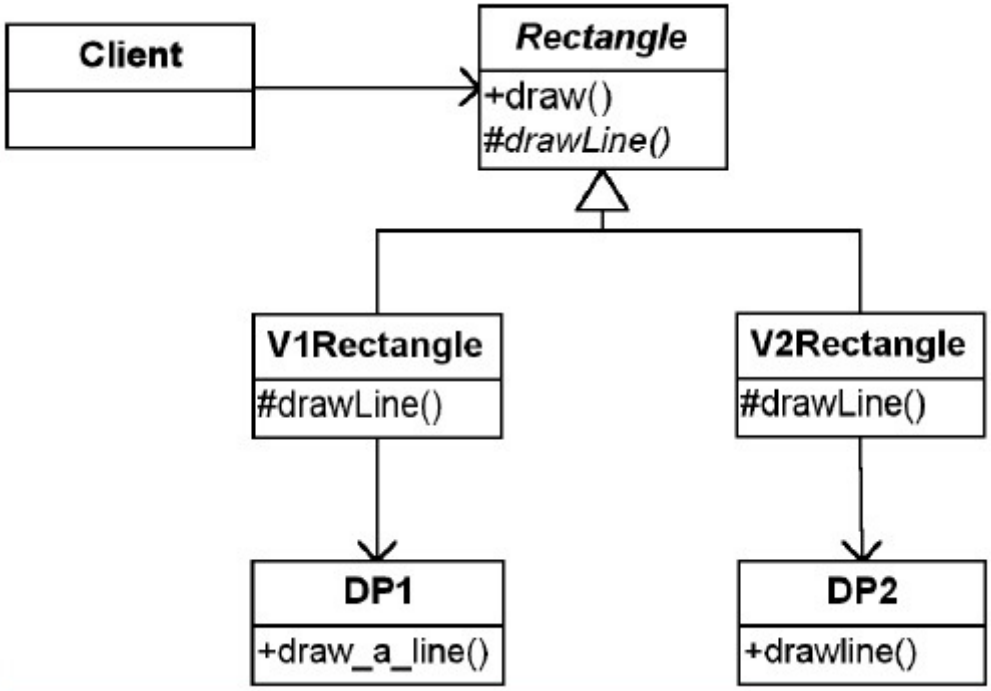
- Task:
  - writing a program that will draw rectangles with either of two drawing programs.
- The rectangles are defined as two pairs of points



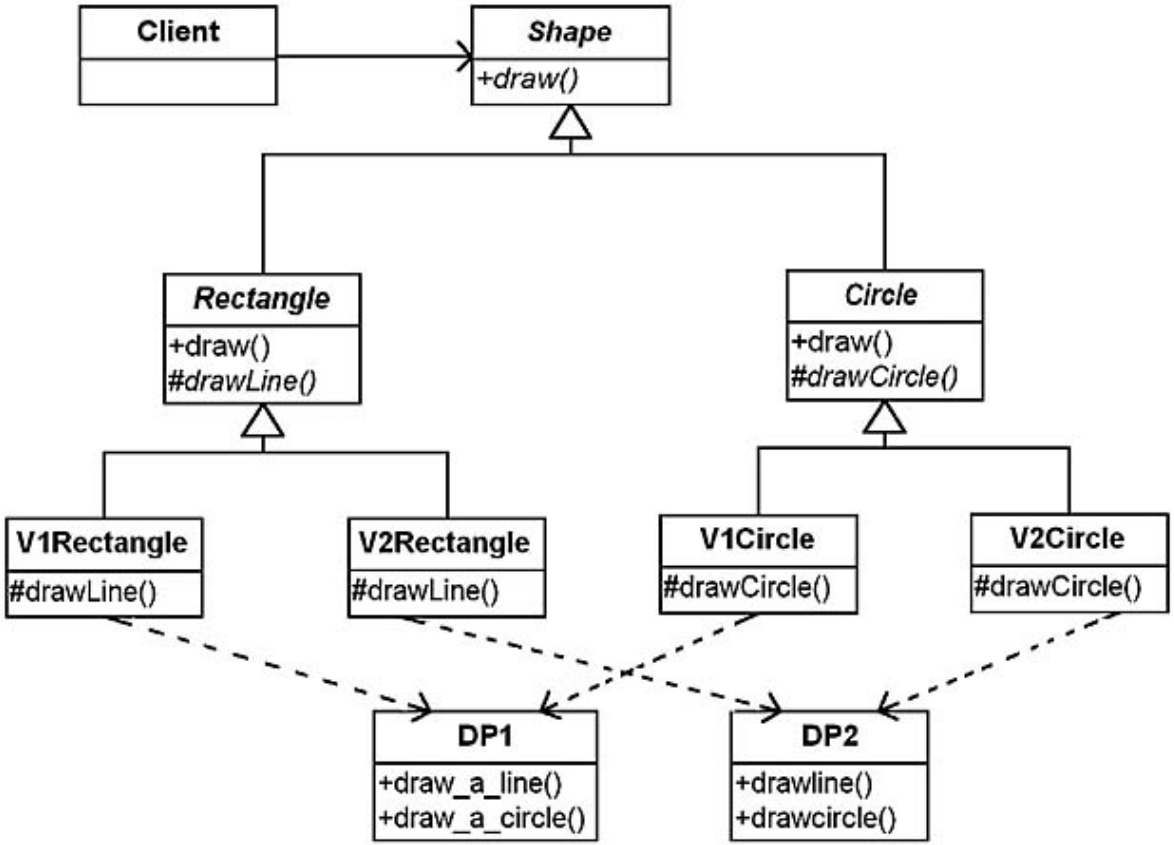
- A rectangle can be drawn using
  - drawing program 1 (DP1)
  - or drawing program 2 (DP2).

DP1	**DP2**
draw_a_line(x1, y1, x2, y2)	drawline(x1, x2, y1, y2)

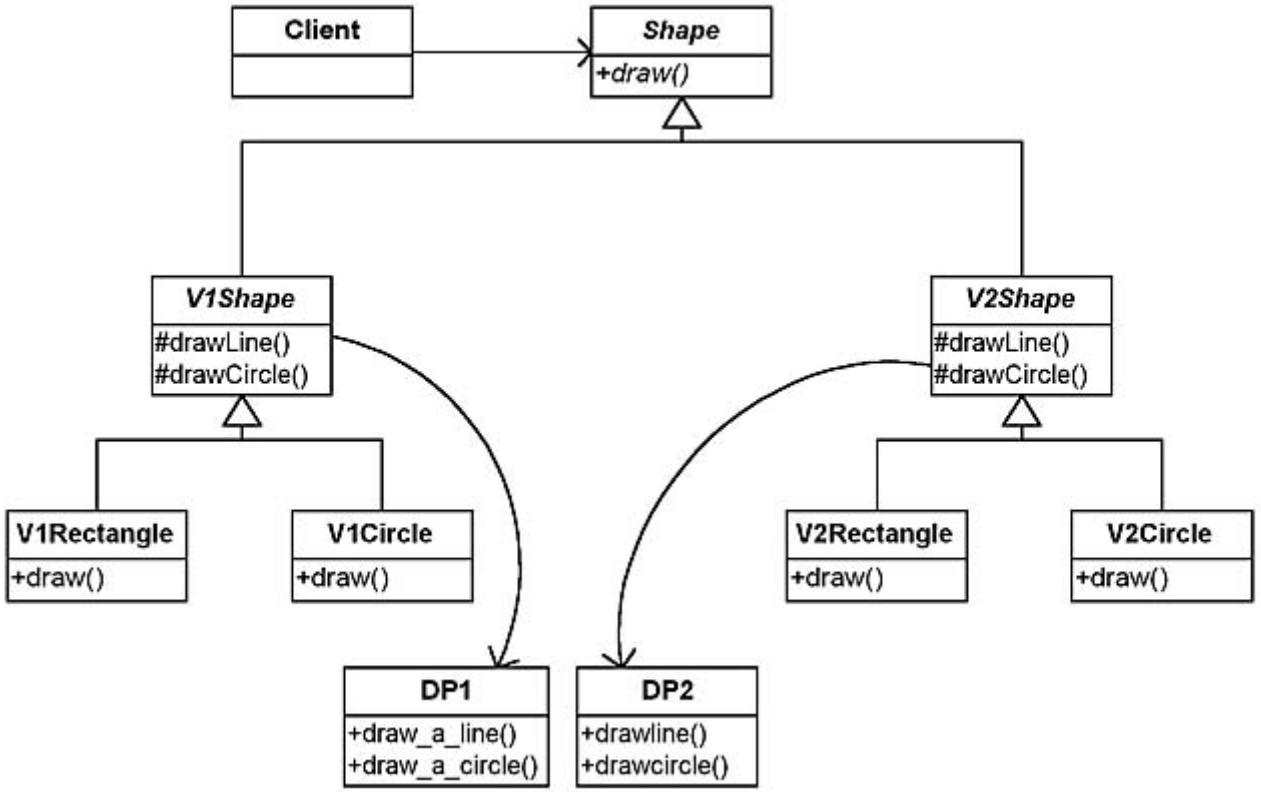
Using inheritance



Then add circles…
-> combinatorial explosion

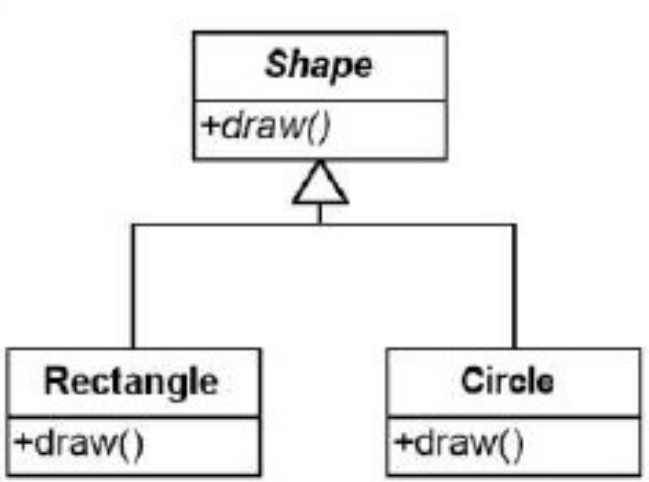


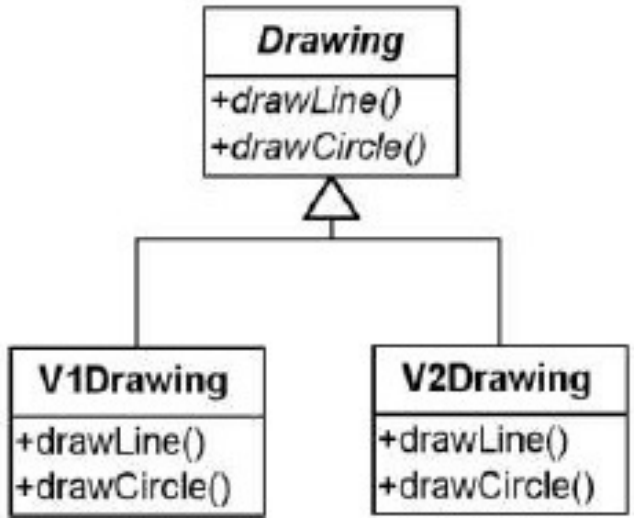
Another solution, but same problem



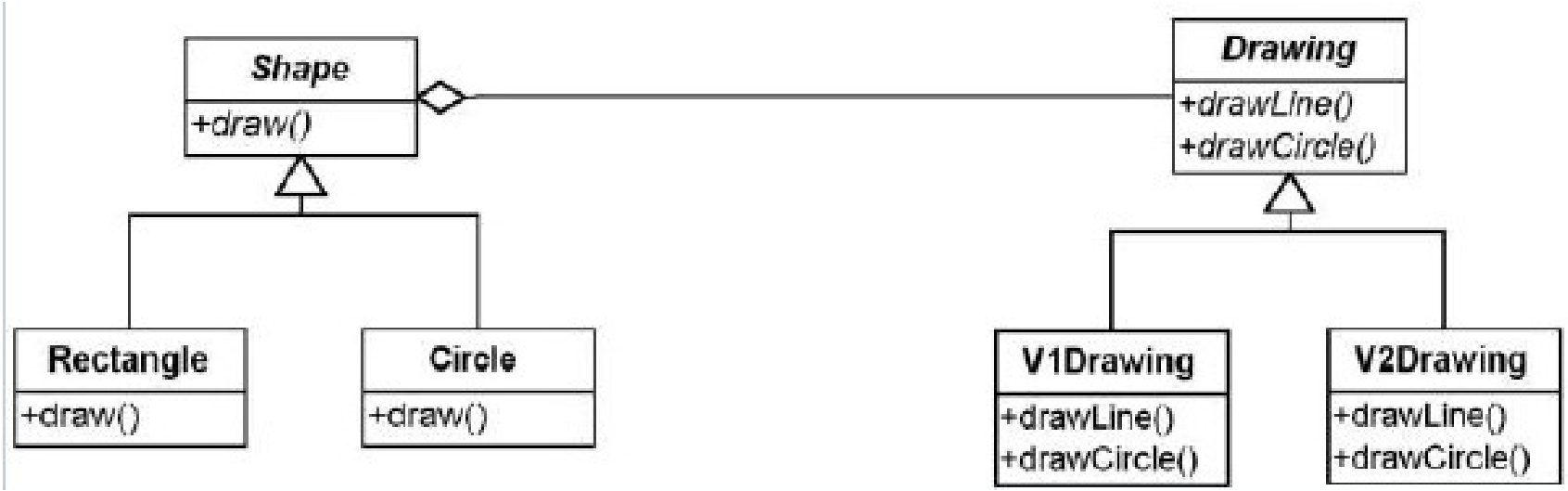
Step 1: Identify what varies

Step 2: Represent the variations

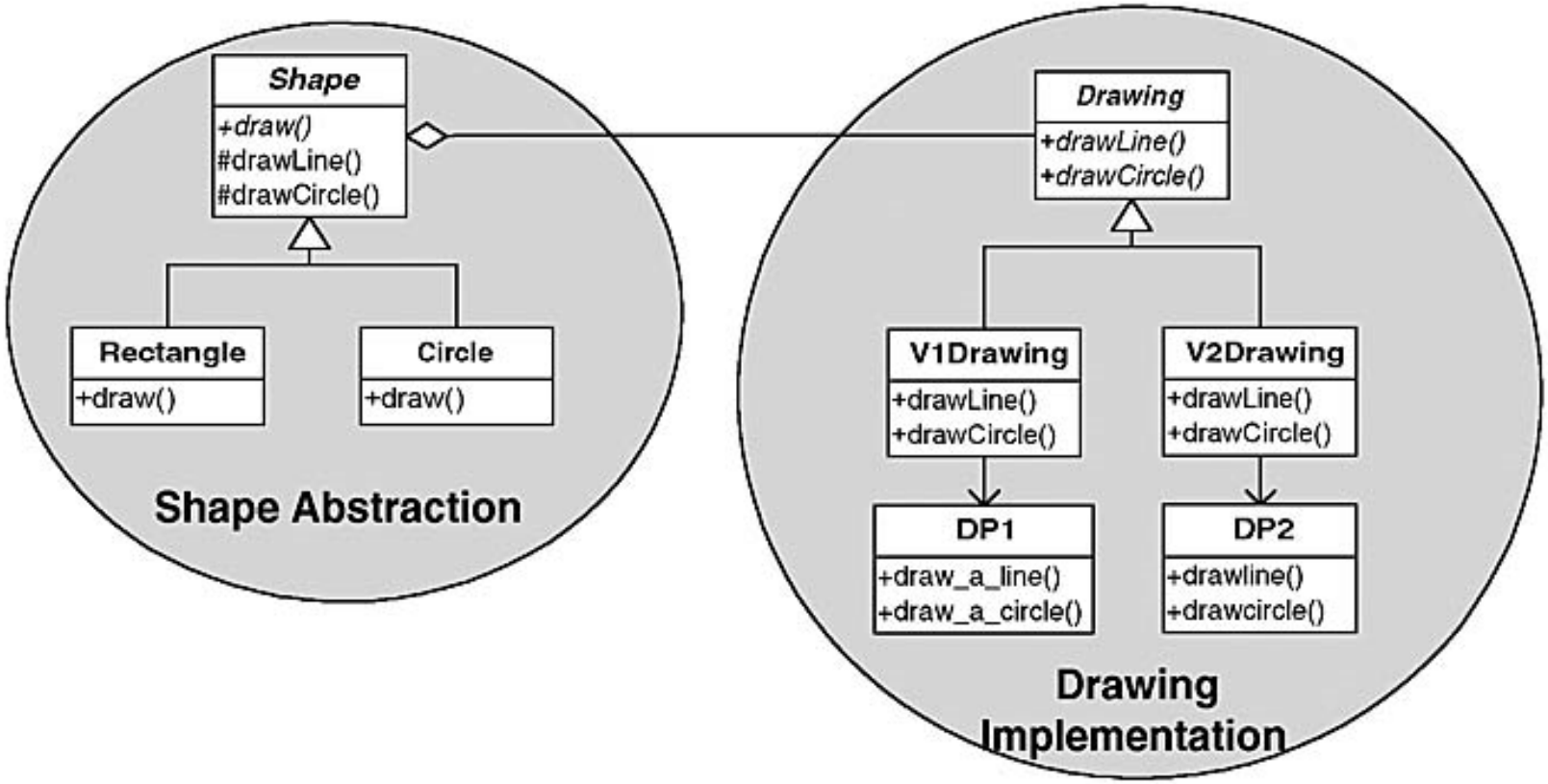




Step 2: Build the bridge to tie hierarchies together



Complete the solution



Shape

```
abstract public class Shape {
    protected Drawing myDrawing;
    abstract public void draw();
    Shape (Drawing drawing) {
        myDrawing= drawing;
    }
    protected void drawLine (double x1,double y1,double x2,double y2) {
        myDrawing.drawLine(x1,y1,x2,y2);
    }
    protected void drawCircle (double x,double y,double r) {
        myDrawing.drawCircle(x,y,r);
    }
}
```

Rectangle

```
public class Rectangle extends Shape {
    private double _x1,_y1,_x2,_y2;
    public Rectangle (Drawing dp,double x1,double y1,double x2,double y2) {
        super( dp);
        x1= x1;_y1= y1;_x2= x2;_y2= y2;
    }
    public void draw() {
        drawLine(_x1,_y1,_x2,_y1);
        drawLine(_x2,_y1,_x2,_y2);
        drawLine(_x2,_y2,_x1,_y2);
        drawLine(_x1,_y2,_x1,_y1);
    }
    protected void drawLine(double x1,double y1,double x2,double y2) {
        myDrawing.drawLine(x1,y1,x2,y2);
    }
}
```

Circle

```
public class Circle extends Shape {
    private double _x,_y,_r;
    public Circle (Drawing dp,double x,double y,double r) {
        super(dp);
        x= x;_y= y;_r= r;
    }
    public void draw() {
        myDrawing.drawCircle(_x,_y,_r);
    }
}
```

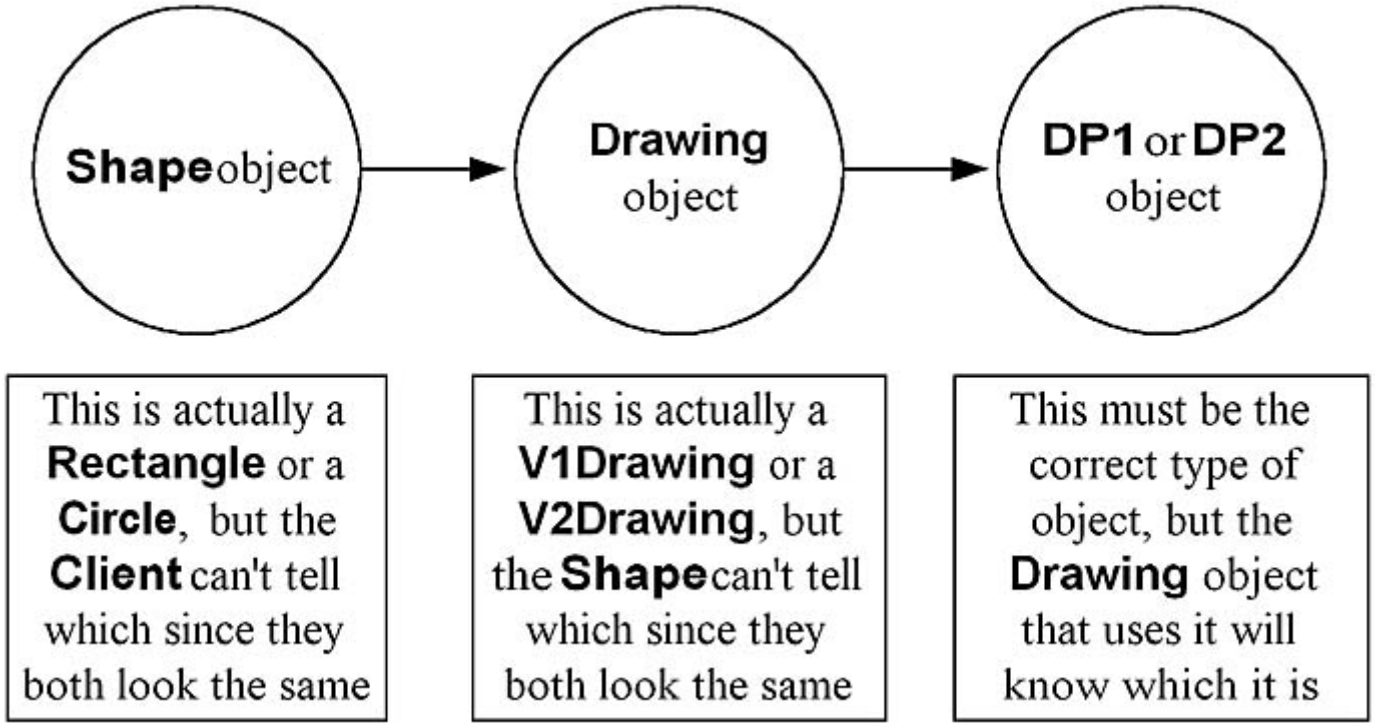
Drawing

```
abstract public class Drawing {
    abstract public void drawLine(double x1,double y1,double x2,double y2);
    abstract public void drawCircle(double x,double y,double r);
}
```

V1Drawing and V2Drawing

```
public class V1Drawing extends Drawing {
    public void drawLine (double x1,double y1,double x2,double y2) {
        DP1.draw_a_line(x1,y1,x2,y2);
    }
    public void drawCircle (double x,double y,double r) {
        DP1.draw_a_circle(x,y,r);
    }
}
public class V2Drawing extends Drawing {
    public void drawLine (double x1,double y1,double x2,double y2) {
        // arguments are different in DP2 and must be rearranged
        DP2.drawLine(x1,x2,y1,y2);
    }
    public void drawCircle (double x, double y,double r) {
        DP2.drawCircle(x,y,r);
    }
}
```

Another picture



The Bridge Pattern

- The Bridge Pattern permits to vary the implementation and abstraction by placing the two in seperate hierarchies.

- Decouple an abstraction or interface from its implementation so that the two can vary independently.

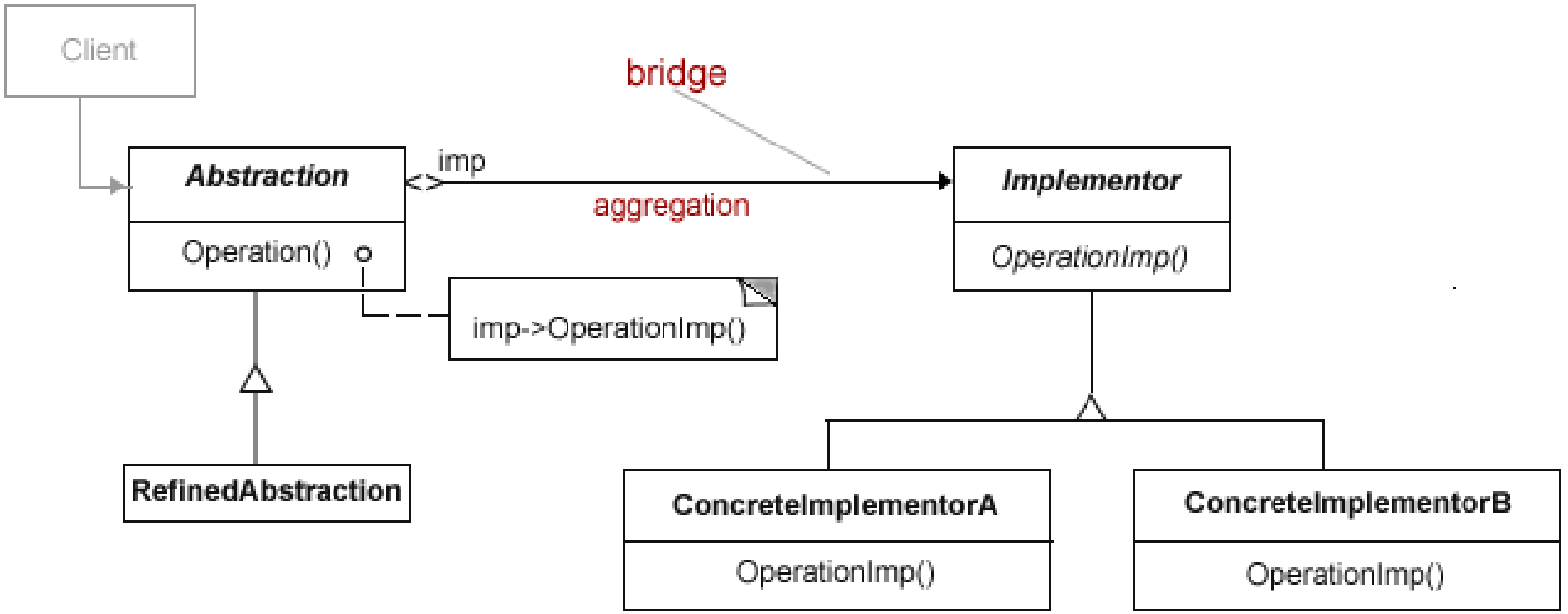
- The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

What does it means????

- Terms «abstraction» and «implementation» may be misleading.

- Think of a hierarchy of classes (that we call abstraction) that all use some operations, that can be implemented in different ways
  - A hierarchy is created to accommodate the various implementations of these operations
  - (similar ti Strategy)

Pattern structure



Participants

- **Abstraction**

  defines the abstract interface
  maintains the Implementor reference

- **Refined Abstraction**

  extends the interface defined by Abstraction

- **Implementor**

  defines the interface for implementation classes

- **ConcreteImplementor**

  implements the Implementor interface

Uses and Benefits

- Want to separate abstraction and implementation permanently
- Share an implementation among multiple objects
- Want to improve extensibility
- Hide implementation details from clients

Uses and Benefits cont'd

- Bridge might be a situation where the programmer thought it would be best to isolate the handling of the system-dependent stuff from the handling of the system-independent stuff.
- The collections class framework in the Java API provides several examples of use of the bridge pattern. Both the ArrayList and LinkedList concrete classes implement the List interface. The List interface provides common, abstract concepts, such as the abilities to add to a list and to ask for its size. The implementation details vary between ArrayList and LinkedList, mostly with respect to when memory is allocated for elements in the list.

First, we have our TV implementation interface

```
//Implementor
public interface TV {
        public void on();
        public void off();
        public void tuneChannel(int channel);
}
```

And then we create two specific implementations.

```
//Concrete Implementor
public class Sony implements TV{
    public void on(){
            //Sony specific on
    }
    public void off(){
            //Sony specific off
    }
    public void tuneChannel(int
    channel) {
            //Sony specific tuneChannel
    }
}
```

```
//Concrete Implementor
public class Philips implements TV{
    public void on(){
            // Philips specific on
    }
    public void off(){
            // Philips specific off
    }
    public void tuneChannel(int
    channel) {
            // Philips specific tuneChannel
    }
}
```

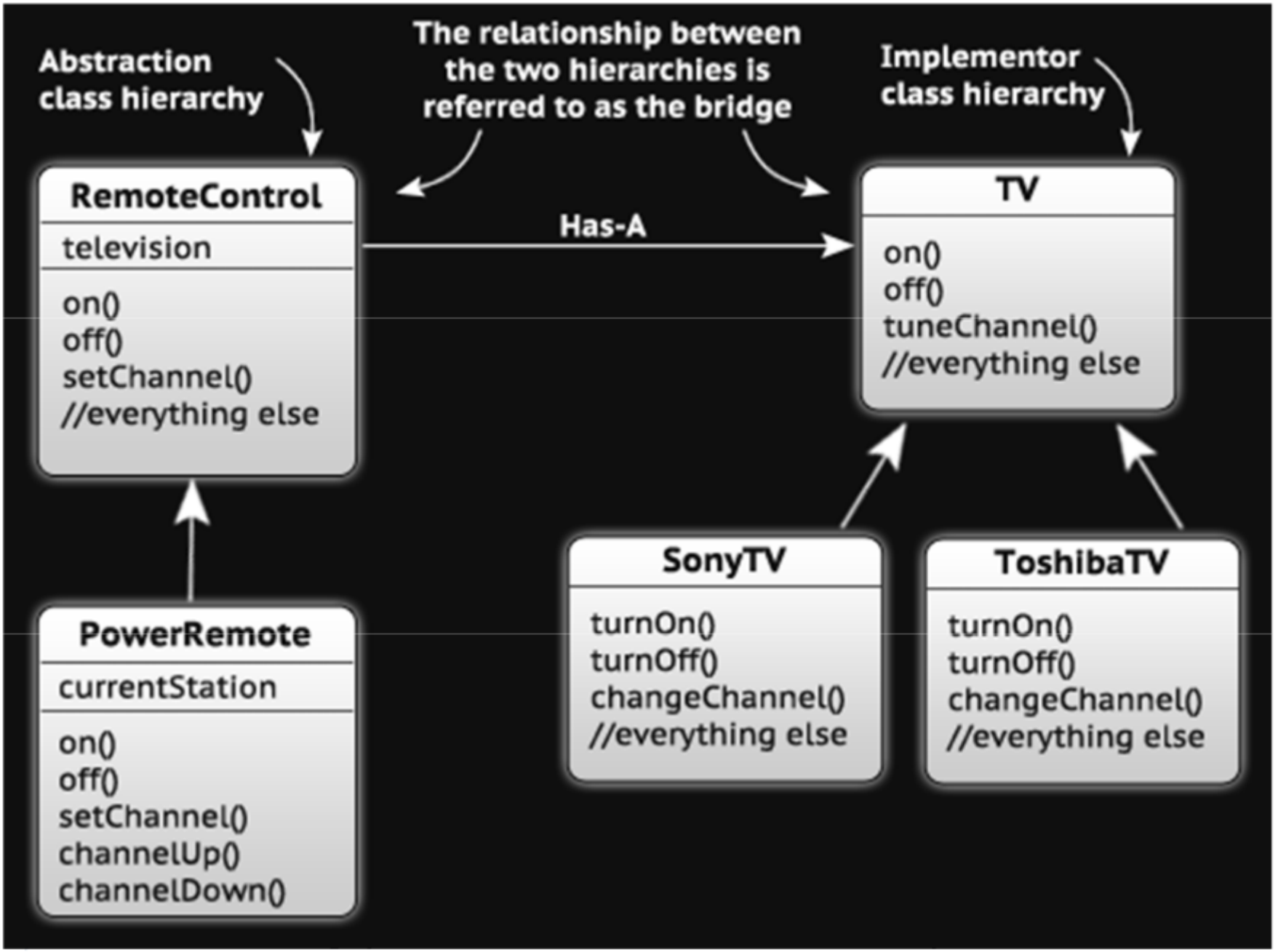
Now, we create a remote control abstraction to control the TV

```
//Abstraction
public abstract class RemoteControl {
   private TV implementor;
   public void on() { implementor.on(); }
   public void off() { implementor.off(); }
   public void setChannel(int channel) {
         implementor.tuneChannel(channel); }
}
```

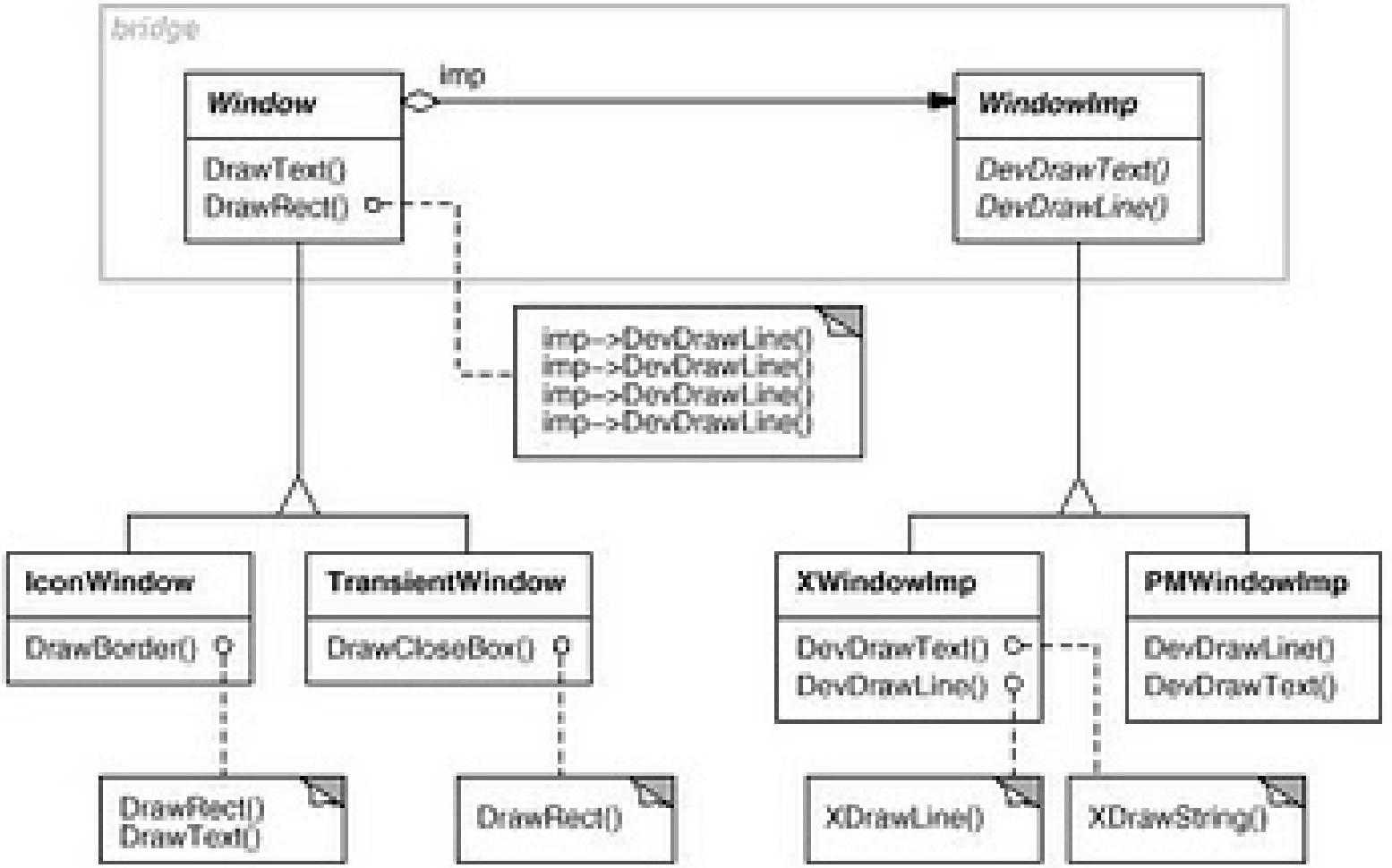
But what if we want a more specific remote control - one that has the + / - buttons for moving through the channels?

```
//Refined abstraction
public class ConcreteRemote extends RemoteControl { private
   int currentChannel;
   public void nextChannel() {
         currentChannel++;
         super.setChannel(currentChannel); }
public void prevChannel() {
         currentChannel--;
         super.setChannel(currentChannel); }
public void setChannel(int channel) {
         super.setChannel(channel);
         currentChannel=channel;}
}
```

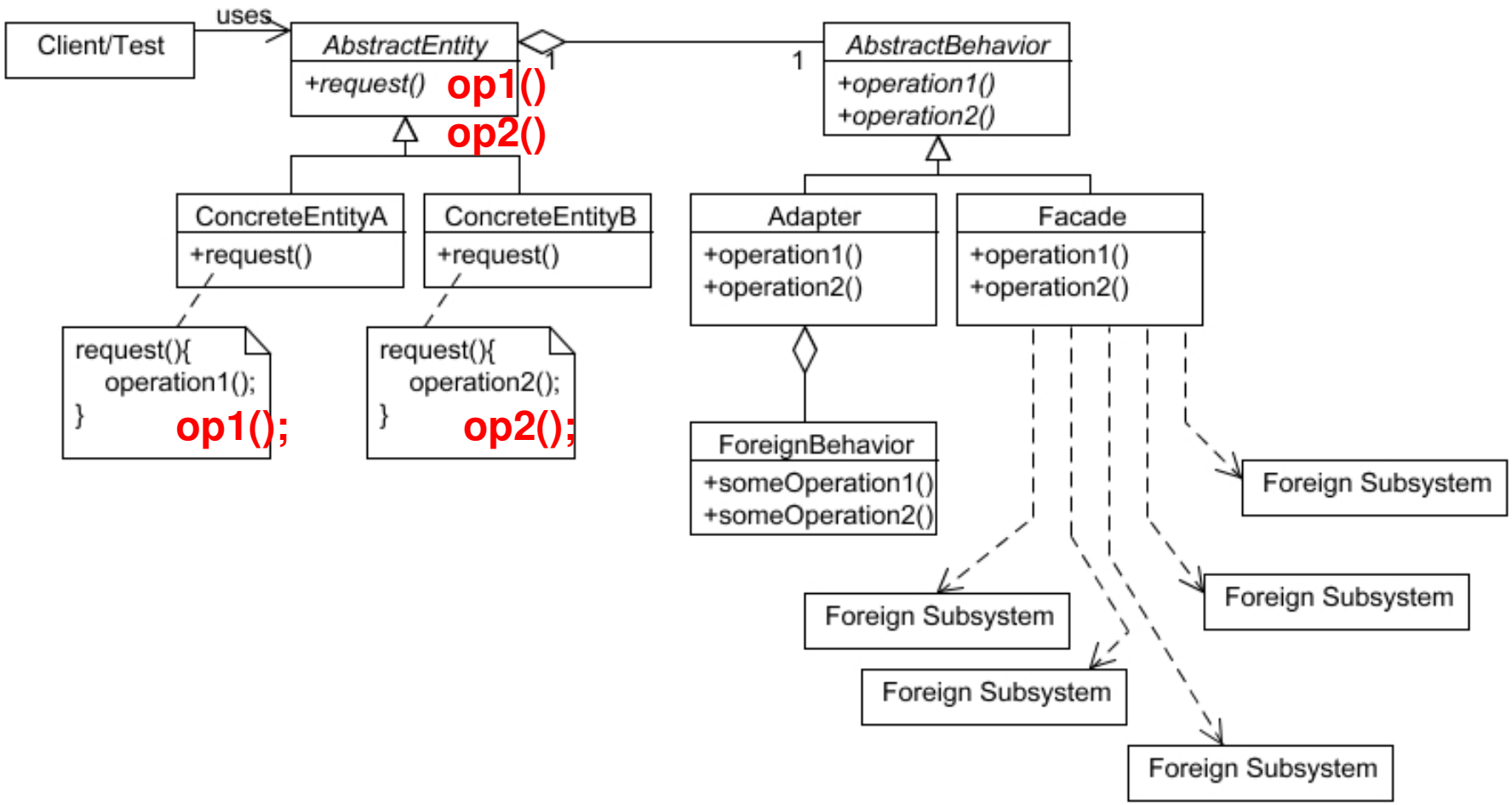
nextChannel definito chiamando metodi dell'astrazione, non dell'implementazione


Abstraction class hierarchy
The relationship between the two hierarchies is referred to as the bridge
Implementor class hierarchy
RemoteControl
television
on()
off()
setChannel()
//everything else
Has-A
TV
on()
off()
tuneChannel()
//everything else
PowerRemote
currentStation
on()
off()
setChannel()
channelUp()
channelDown()
SonyTV
turnOn()
turnOff()
changeChannel()
//everything else
ToshibaTV
turnOn()
turnOff()
changeChannel()
//everything else

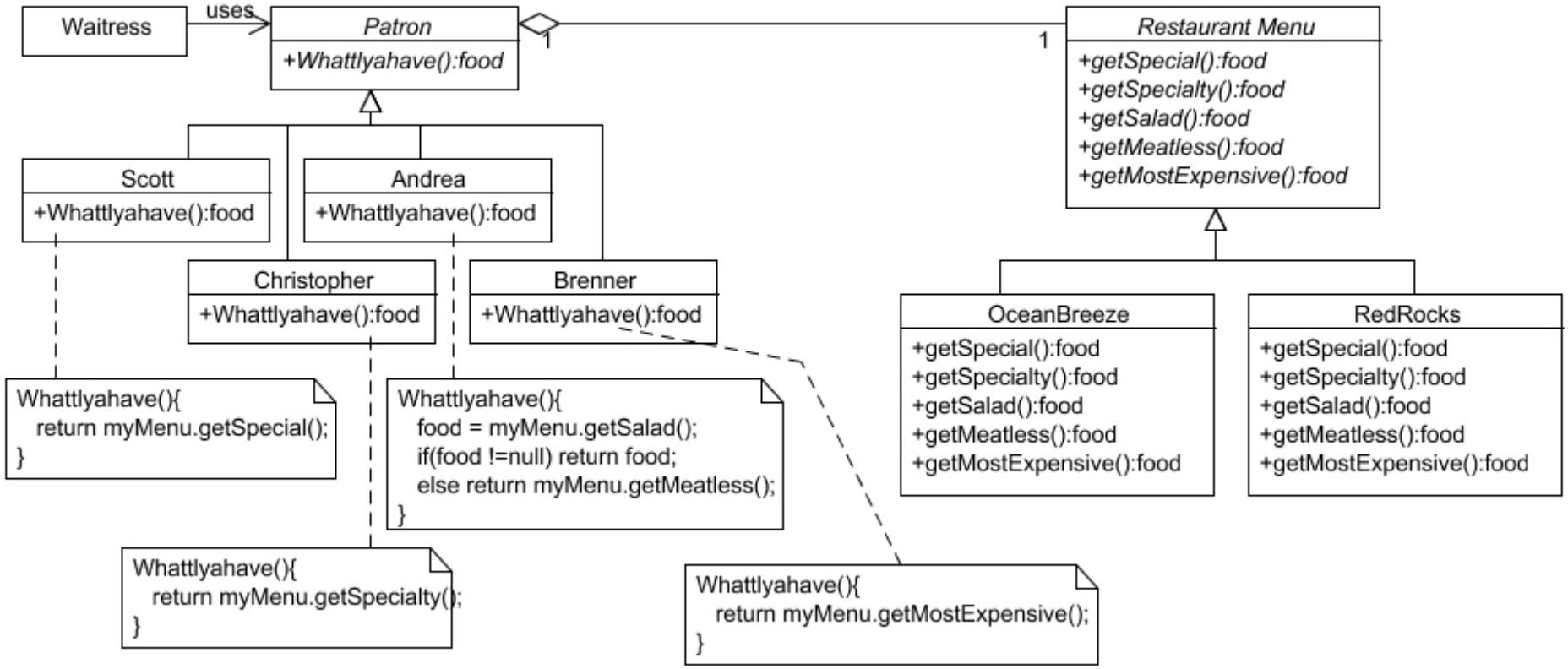
GoF example


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

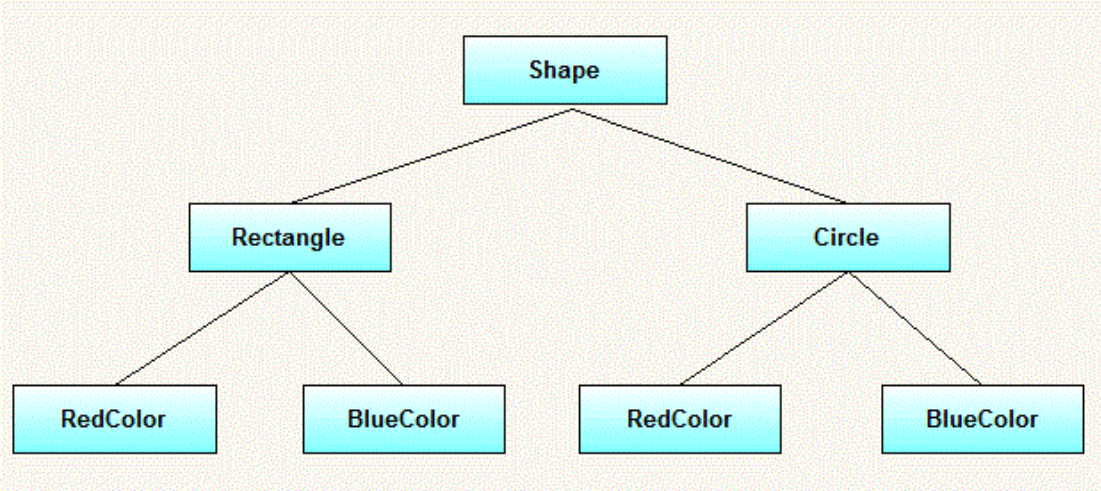
Example: right or wrong? Just misleading -> better as in red

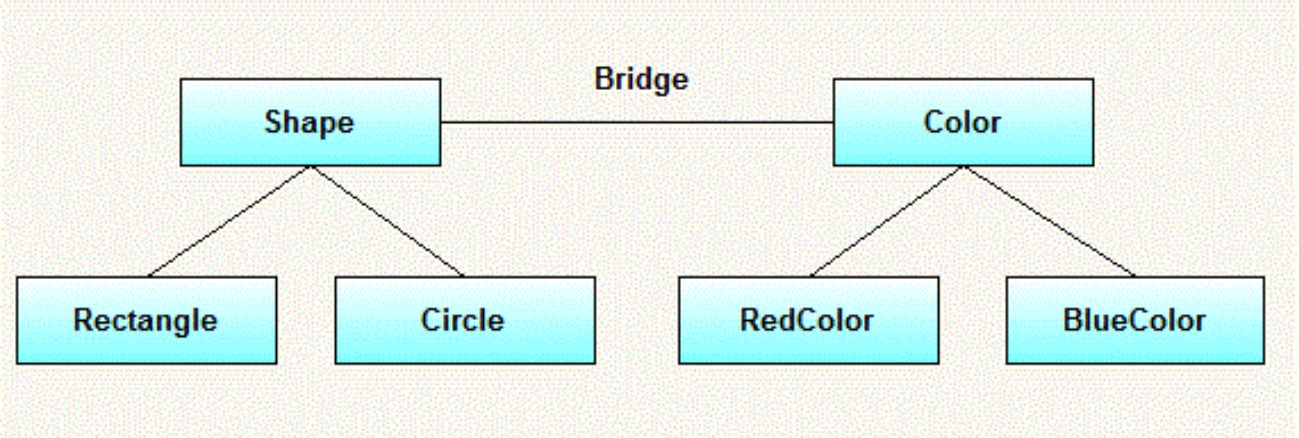


Another misleading example

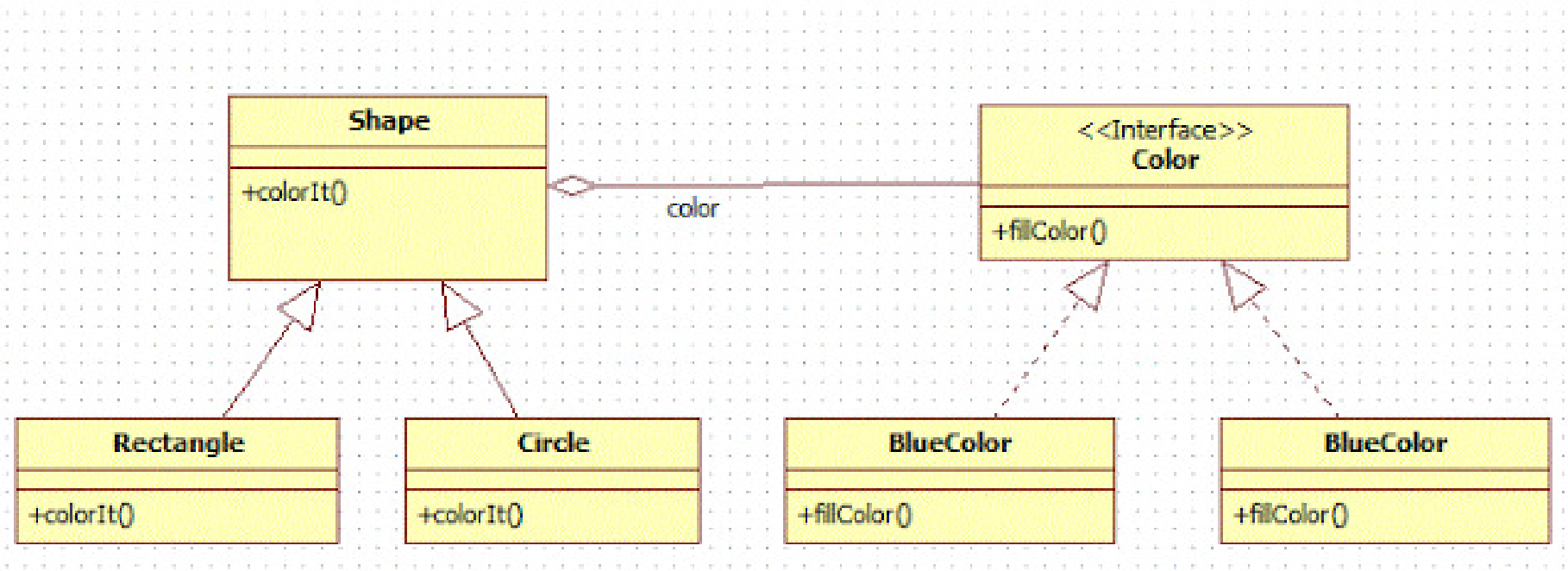


Example:
before and after applying the pattern





A colouring functionality



Shape (Abstraction)

```
abstract class Shape {

    Color color;
    Shape(Color color)
    {
        this.color=color;
    }
    abstract public void colorIt();
}
```

RefinedAbstractions

Rectangle

```
public class Rectangle extends Shape{
    Rectangle(Color color) {
            super(color);
    }
    public void colorIt() {
            System.out.print("Rectangle
filled with ");
            color.fillColor();
            }
}
```

Circle

```
public class Circle extends Shape{
   Circle color) {
            super(color);
    }
    public void colorIt() {
            System.out.print(" Circle filled
with ");
            color.fillColor();
            }
}
```

Color (Implementor)

```
public interface Color {
        public void fillColor();
}
```

Implementors

RedColor

```
public class RedColor implements Color{
        public void fillColor() {

        System.out.println("red color");
        }
}
```

BlueColor

```
public class BlueColor implements Color{
        public void fillColor() {

        System.out.println(“blue color");
        }
}
```

Now…

- Improve the colors example

Bridge vs Strategy
Bridge vs Adapter

- Often, the Strategy Pattern is confused with the Bridge Pattern. Even though, these two patterns are similar in structure, they are trying to solve two different design problems. Strategy is mainly concerned in encapsulating algorithms, whereas Bridge decouples the abstraction from the implementation, to provide different implementation for the same abstraction.
- The structure of the Adapter Pattern (object adapter) may look similar to the Bridge Pattern. However, the adapter is meant to change the interface of an existing object and is mainly intended to make unrelated classes work together.

Homework

- Use the Bridge pattern to model the software for switched devices in a home. Switches come in several varieties, such as
  - a ceiling light two-position switch (to toggle the light on and off),
  - a ceiling fan pull chain (i.e., each pull increases the fan's speed until the maximum speed is reached, where upon the next pull turns the fan off), and
  - a light dimmer switch (to permit continuous adjustment to the light's brightness).
- Use a UML class diagram to illustrate your model.