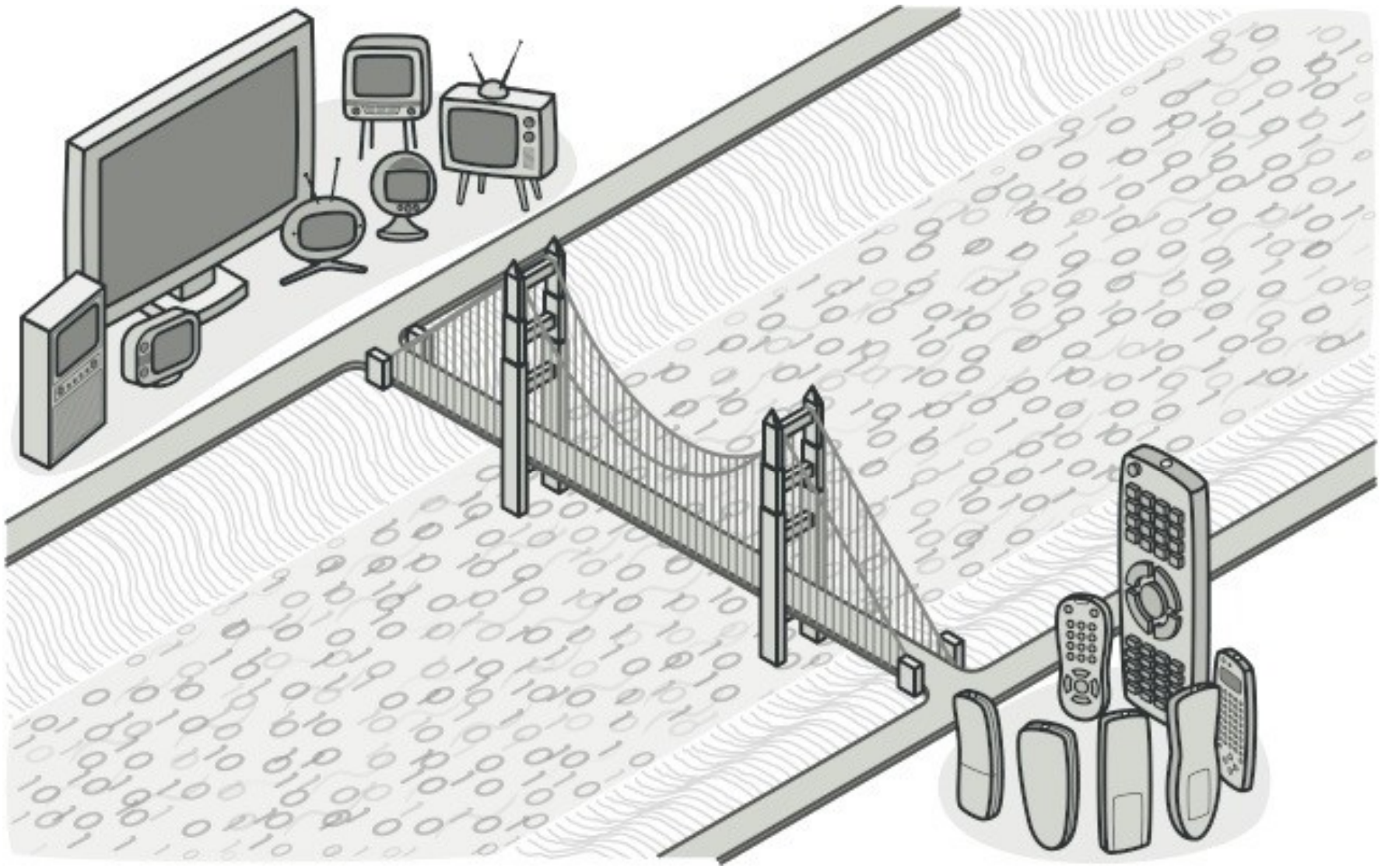
Bridge

Lecture 20

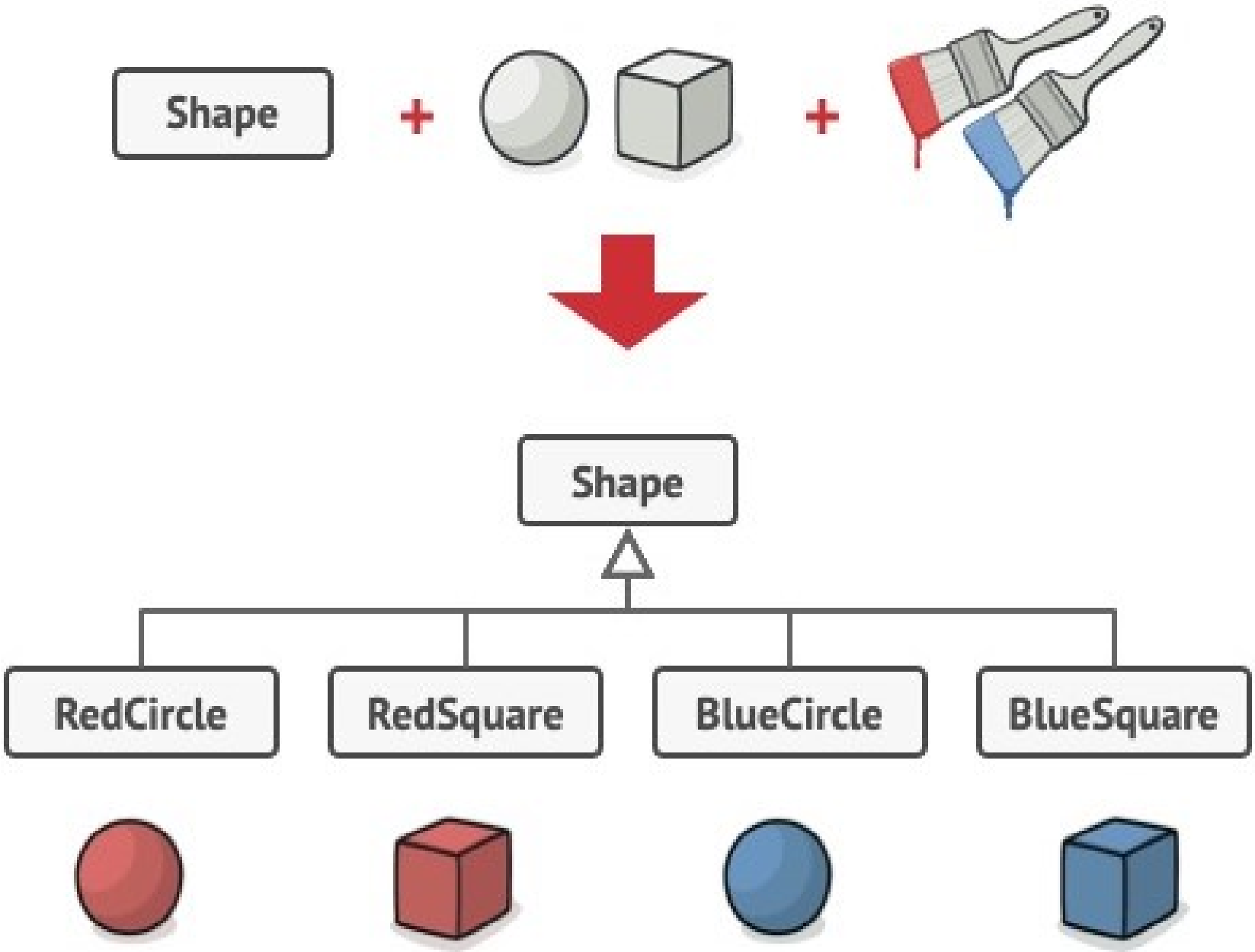
Intent

- **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



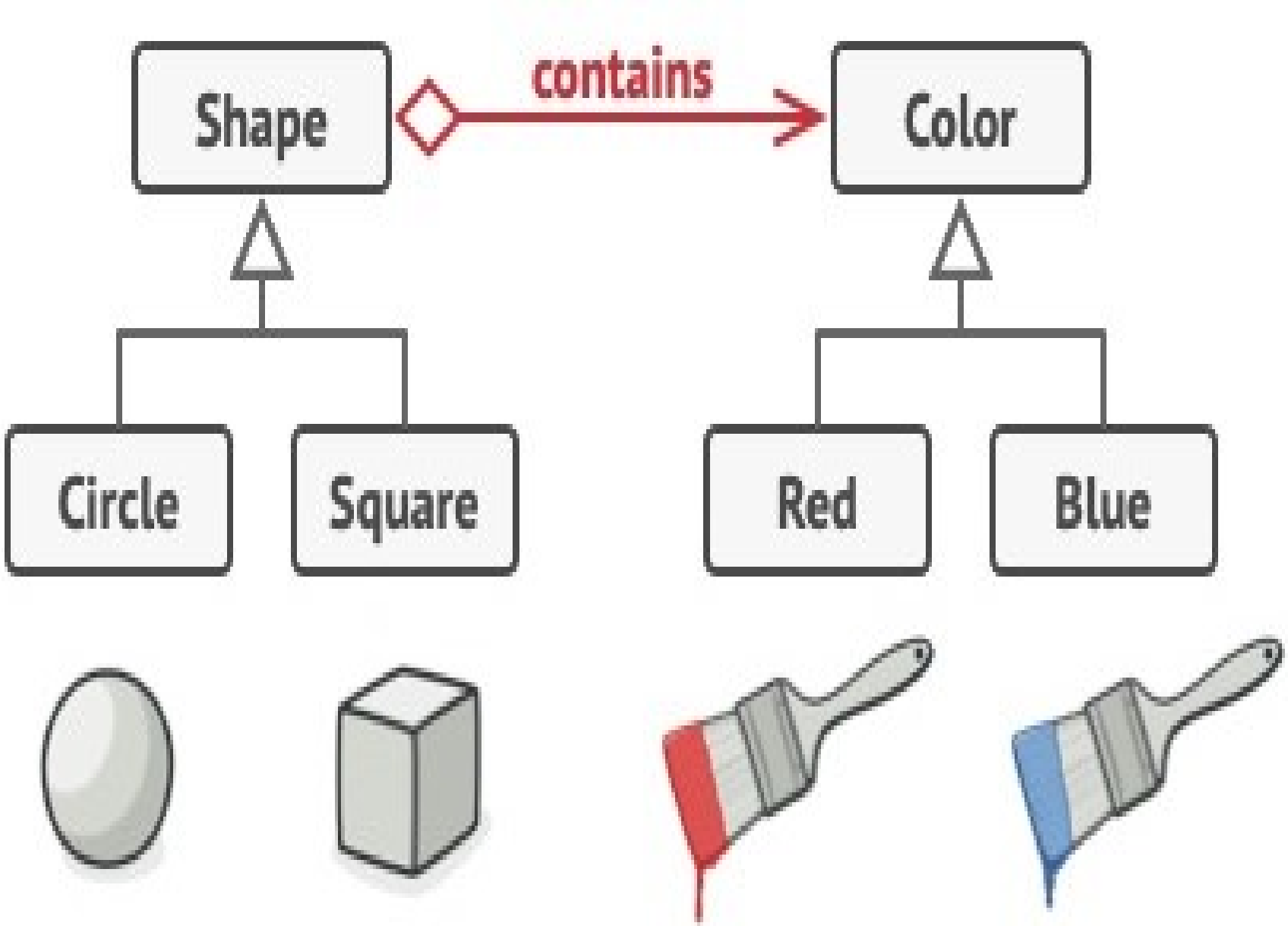
Problem

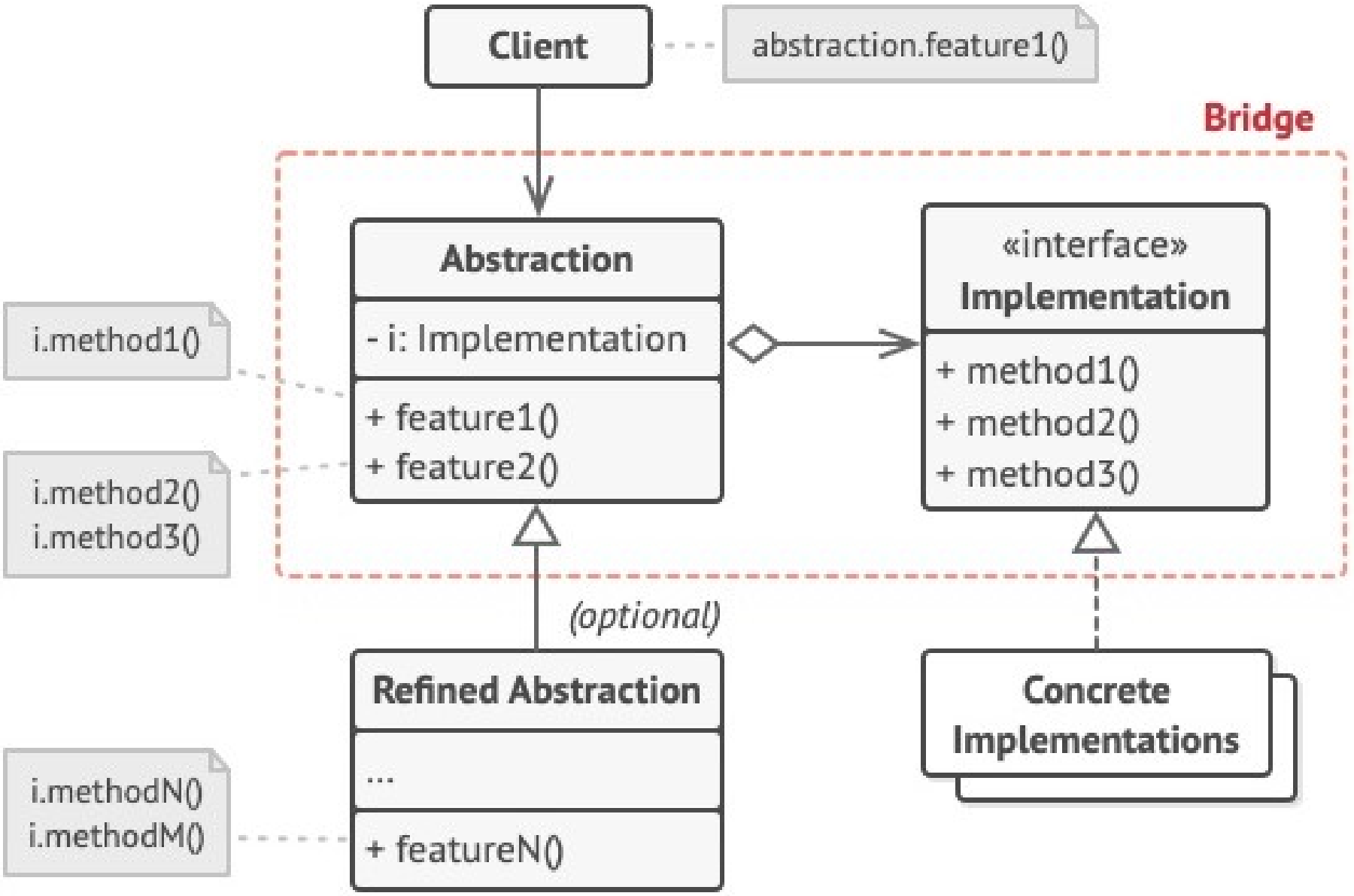
- Say you have a geometric Shape class with a pair of subclasses: Circle and Square.
- You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.
- However, since you already have two subclasses, you'll need to create four class combinations
- Adding new shape types and colors to the hierarchy will grow it exponentially.


Shape
Shape
RedCircle
RedSquare
BlueCircle
BlueSquare

Solution

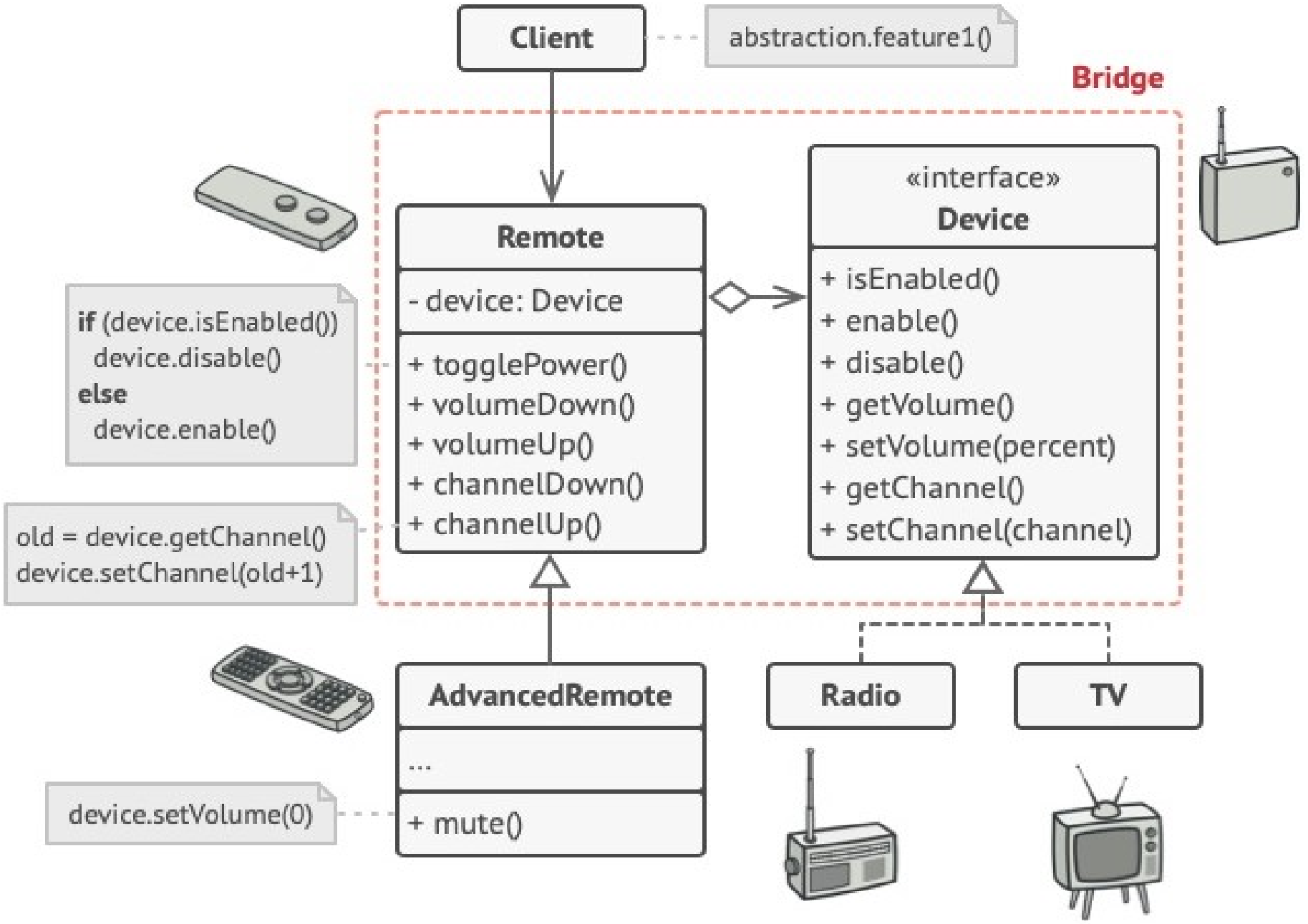
- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.


Shape
contains
Color
Circle
Square
Red
Blue


Client
abstraction.feature1()
Bridge
Abstraction
- i: Implementation
+ feature1()
+ feature2()
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(optional)
Refined Abstraction
...
+ featureN()
Concrete
Implementations
i.methodN()
i.methodM()

Example

- This example illustrates how the **Bridge** pattern can help divide the monolithic code of an app that manages devices and their remote controls.
  The Device classes act as the implementation, whereas the Remotes act as the abstraction.


Client
abstraction.feature1()
Bridge
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

```
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)


// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)
```

```
// All devices follow the same interface.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

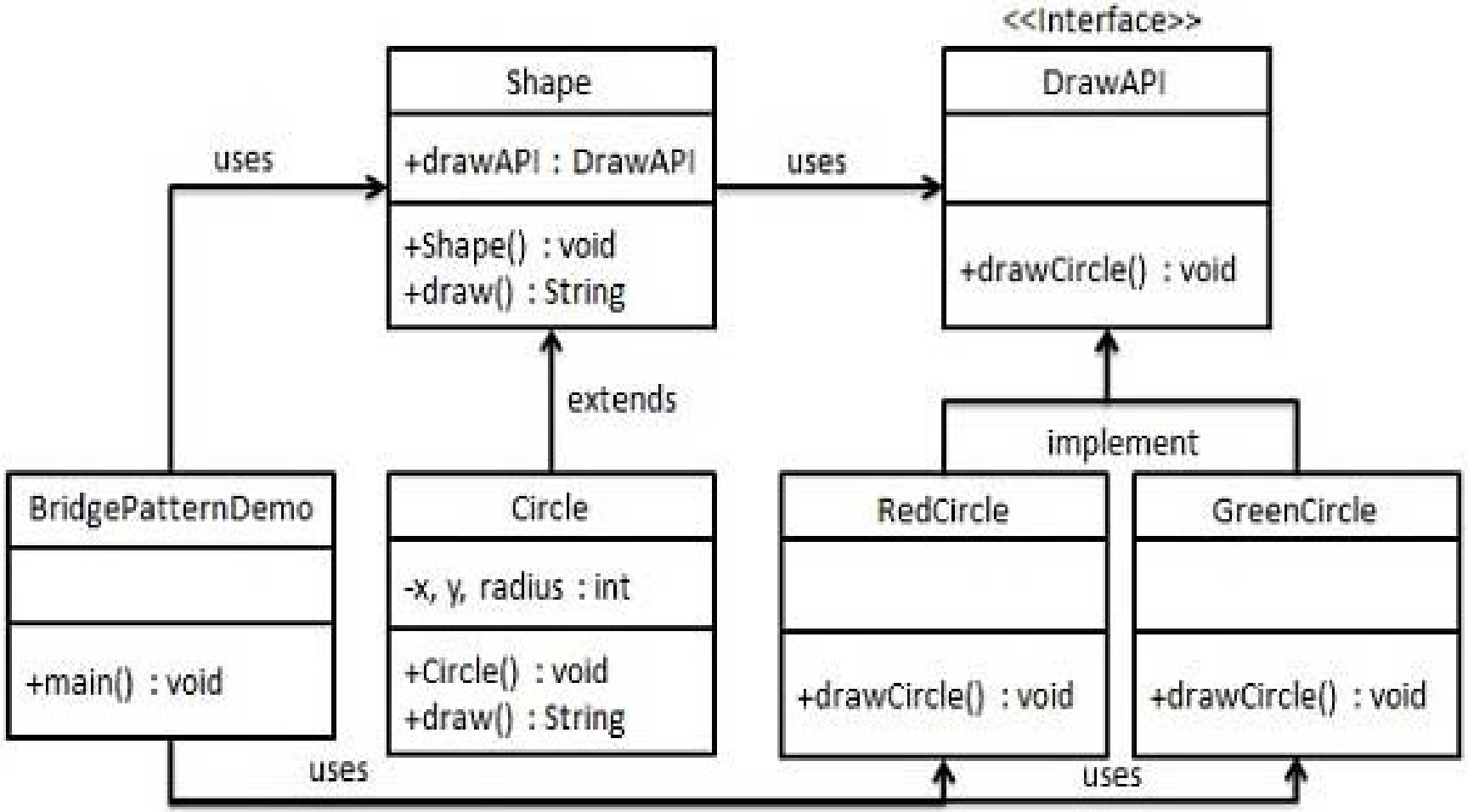
Advantages

- You can create platform-independent classes and apps.
- The client code works with high-level abstractions. It isn't exposed to the platform details.
- *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
- *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

Disadvantage

- You might make the code more complicated by applying the pattern to a highly cohesive class.

Example


<<Interface>>
Shape
DrawAPI
uses
+drawAPI : DrawAPI
uses
+Shape() : void
+draw() : String
+drawCircle() : void
extends
implement
BridgePatternDemo
Circle
RedCircle
GreenCircle
-x, y, radius : int
+main() : void
+Circle() : void
+draw() : String
+drawCircle() : void
+drawCircle() : void
uses
uses

Step 1

Create bridge implementer interface.

DrawAPI.java

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

Step 2

Create concrete bridge implementer classes implementing the *DrawAPI* interface.

RedCircle.java

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x:
   }
}
```

GreenCircle.java

```
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " + radius + ",
   }
}
```

Step 3

Create an abstract class *Shape* using the *DrawAPI* interface.

Shape.java

```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

Step 4

Step 4

Create concrete class implementing the *Shape* interface.

Circle.java

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Step 5

Use the *Shape* and *DrawAPI* classes to draw different colored circles.

BridgePatternDemo.java

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

Reference

- https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm
- https://refactoring.guru/design-patterns/bridge