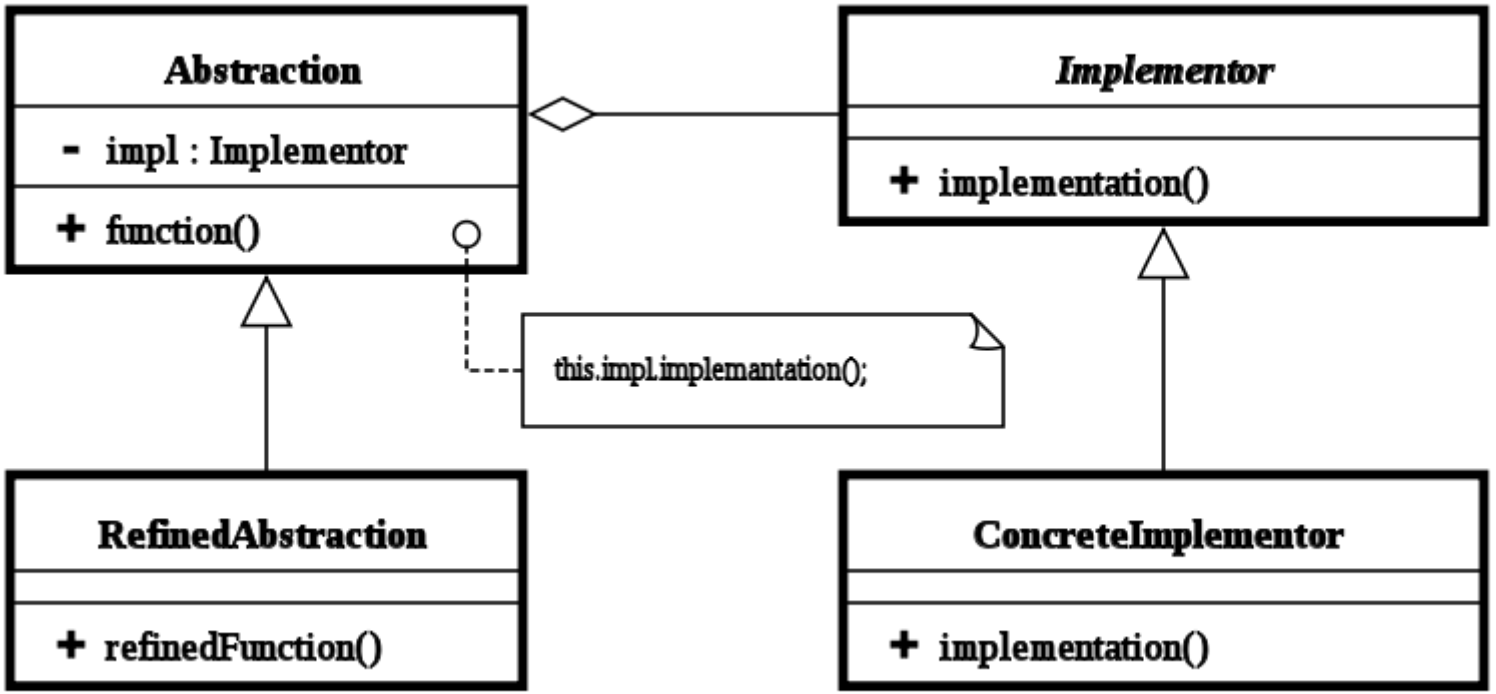
Bridge Pattern

“Used to decouple an abstraction from its implementation so that the two can vary independently”

What that means

- Abstraction and implementation are not related in the sense that the implementation is a concrete version of the abstraction
- What a class can do is the abstraction, what a class itself can be thought of is the implementation
- The Bridge Pattern allows both to change independently (thus loose coupling is in place)
- Think of this pattern as **two** layers of abstraction

Class Diagram



Discussion of Class Diagram

- Abstraction contains a reference to Implementor
- Abstraction can (typically does) have sub-classes that represent the change in the Abstraction (what a class can do)
- Implementor typically has ConcreteImplementors which allow changes on that side
- The Abstraction and the Implementor are the two items that make the Bridge: they are what allow the independent change

Code Example From Wikipedia

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
   }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
   }
}
```

Code Example Continued

```
/** "Abstraction" */
interface Shape {
   public void draw();                              // low-level
   public void resizeByPercentage(double pct);     // high-level
}

/** "Refined Abstraction" */
class CircleShape implements Shape {
   private double x, y, radius;
   private DrawingAPI drawingAPI;
   public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
       this.x = x;  this.y = y;  this.radius = radius;
       this.drawingAPI = drawingAPI;
   }

   // low-level i.e. Implementation specific
   public void draw() {
        drawingAPI.drawCircle(x, y, radius);
   }
   // high-level i.e. Abstraction specific
   public void resizeByPercentage(double pct) {
        radius *= pct;
   }
}
```

Code Example Continued

```
/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        };

        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
```

Thoughts Regarding the Bridge Pattern

- Keep in mind when using this pattern that there is an abstraction part and an implementation part
- The implementation's interface should be designed considering the different derivations of the abstract class that it will have to support: don't overdo this – only design for the derivations that are being built

Key Features of Bridge Pattern

- Intent: Decouple a set of implementations from the set of objects using them
- Problem: The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes
- Solution: Define an interface for all the implementations to use and have the derivations of the abstract class use that
- Participators and Collaborators
  - Abstraction defines the interface for the objects being implemented
  - Implementor defines the interface for the specific implementation classes. Classes derived from Abstraction use classes derived from Implementor without knowing which particular ConcreteImplementor is in use
- Consequences: The decoupling of the implementations from the objects that use them increases the extensibility. Client objects are not aware of implementation issues
- Implementation
  - Encapsulate the implementations in an abstract class
  - Contain a handle to it in the base class of the abstraction being implemented

How is Extensible is Bridge Pattern?

- With regards to a new implementation in the system, you just need to add a new ConcreteImplementation class – nothing else changes :-)
- With a new concrete example of the abstraction, more work is involved
  - Suppose we get a new Shape that requires a new drawing function: the implementations will need modified to reflect this
  - Fortunately, there is a well-defined process for the modification: modify the interface first, then the derivatives of that interface accordingly. This approach localizes the impact of change and lowers the risk of unwanted side effects

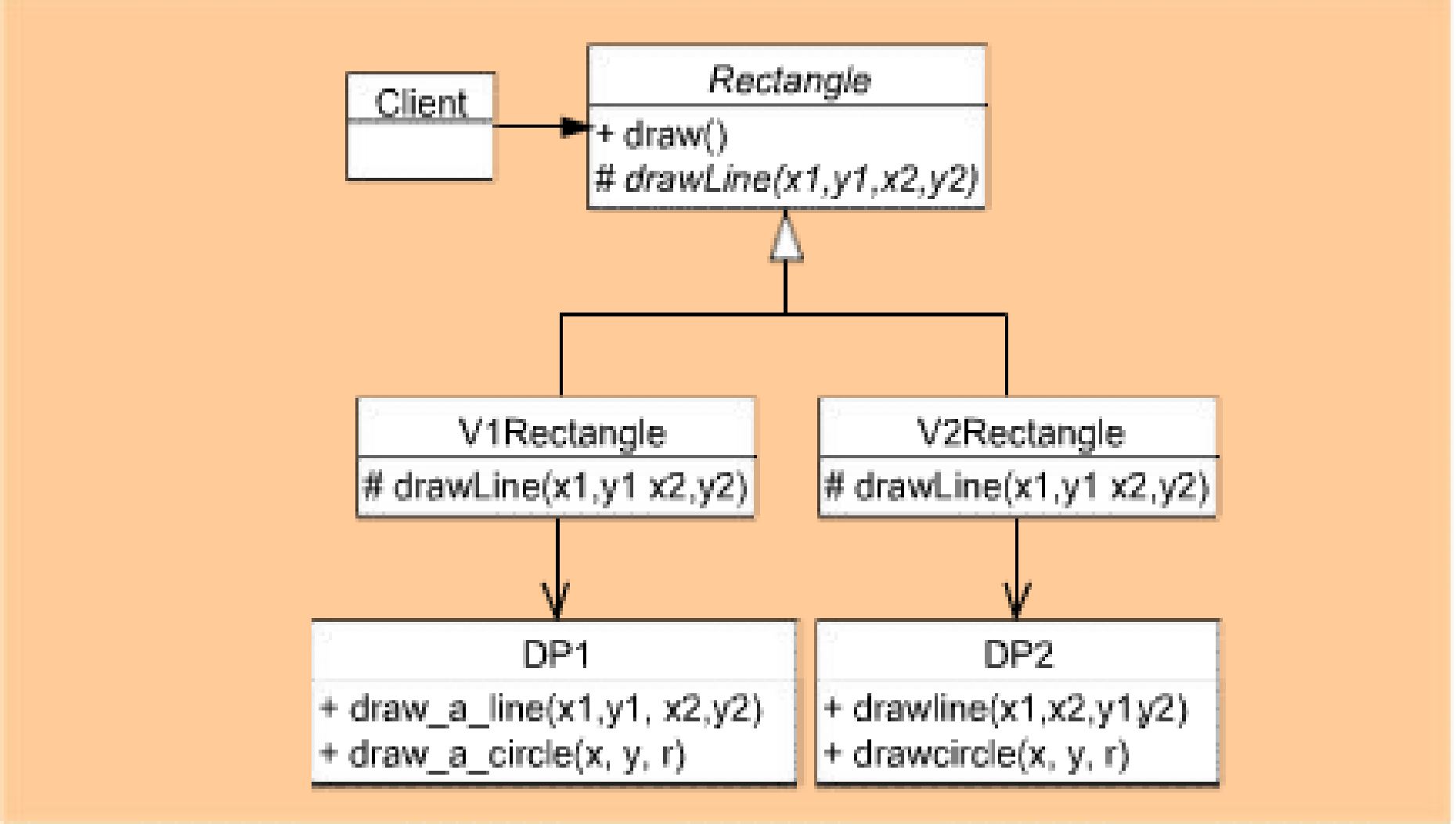
OO Principle

- The Bridge Pattern applies the “one rule, one place” principle
- A rule represents specifically how a single thing should be done. A single method in a single class should be used to represent this rule
- This leads to code with a greater number of small methods, but avoids duplication
- Refactoring code is all about applying “one rule, one place” (among other things)

Refactoring

- It is not strictly an object-oriented concept
- It is modifying code to improve its structure without adding function
- See Martin Fowler's “Refactoring: Improving the Design of Existing Code”

Bridge Pattern Applied (from DPE)



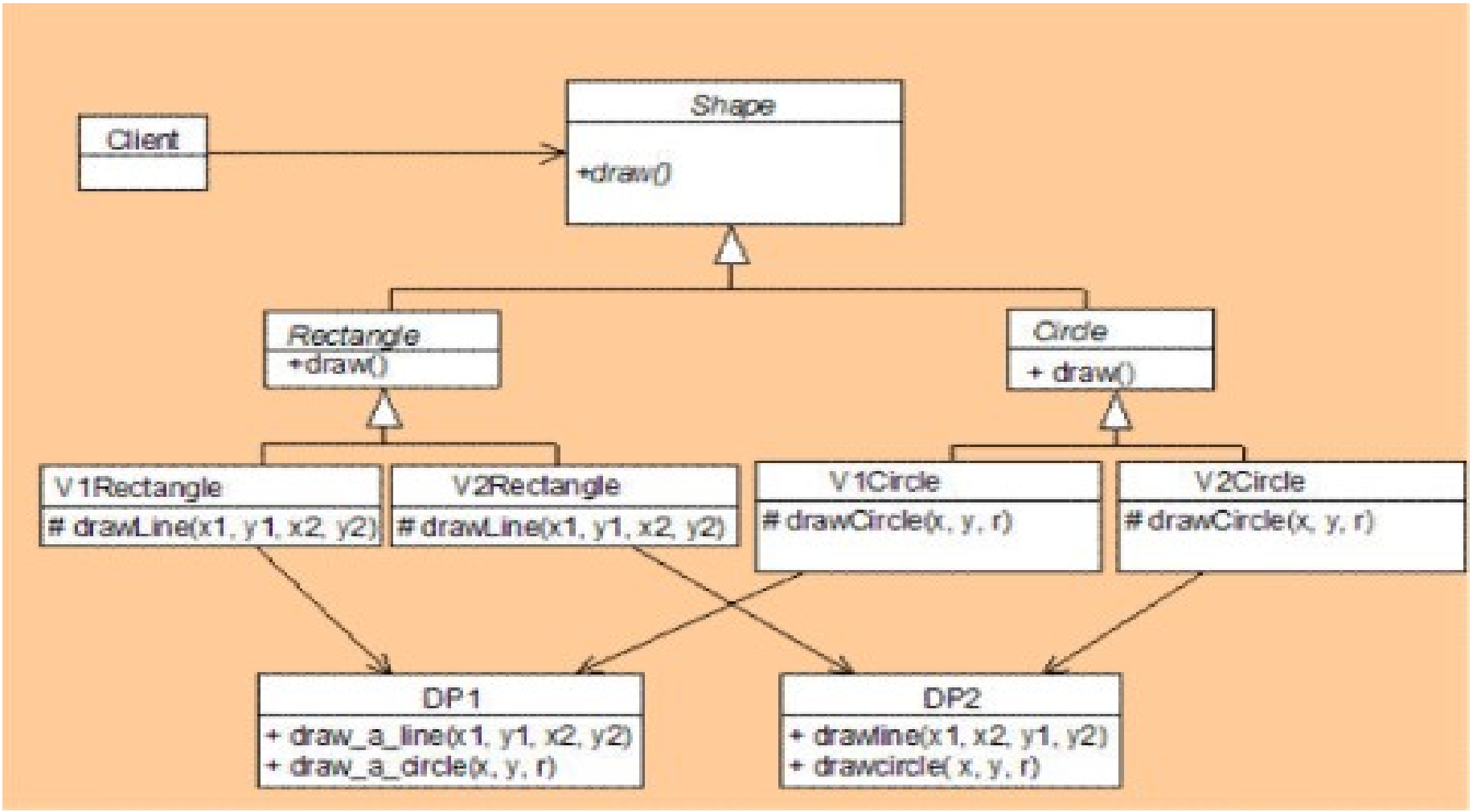
Bridge Pattern Applied: Code

```
abstract class Rectangle {
  void public draw () {
     drawLine(_x1,_y1,_x2,_y1);
     drawLine(_x2,_y1,_x2,_y2);
     drawLine(_x2,_y2,_x1,_y2);
     drawLine(_x2,_y1,_x1,_y1);
   }
  abstract void drawLine( double x1, double y1,
                          double x2, double y2);
}

class V1Rectangle extends Rectangle {
  drawLine( double x1, double y1, double x2, double y2) {
    DP1.draw_a_line( x1,y1,x2,y2);
  }
}

class V2Rectangle extends Rectangle {
  drawLine(double x1, double y1, double x2, double y2) {
    DP2.drawline( x1,x2,y1,y2);
  }
}
```

New Requirements

- Told we must now handle Circles
- We decide to handle it in a similar way
- Make a V1cirlce class and a V2Circle class
- Decouple the client from the specific shapes by deriving them from a common base class

Class Diagram Incorporating New Requirements



Code

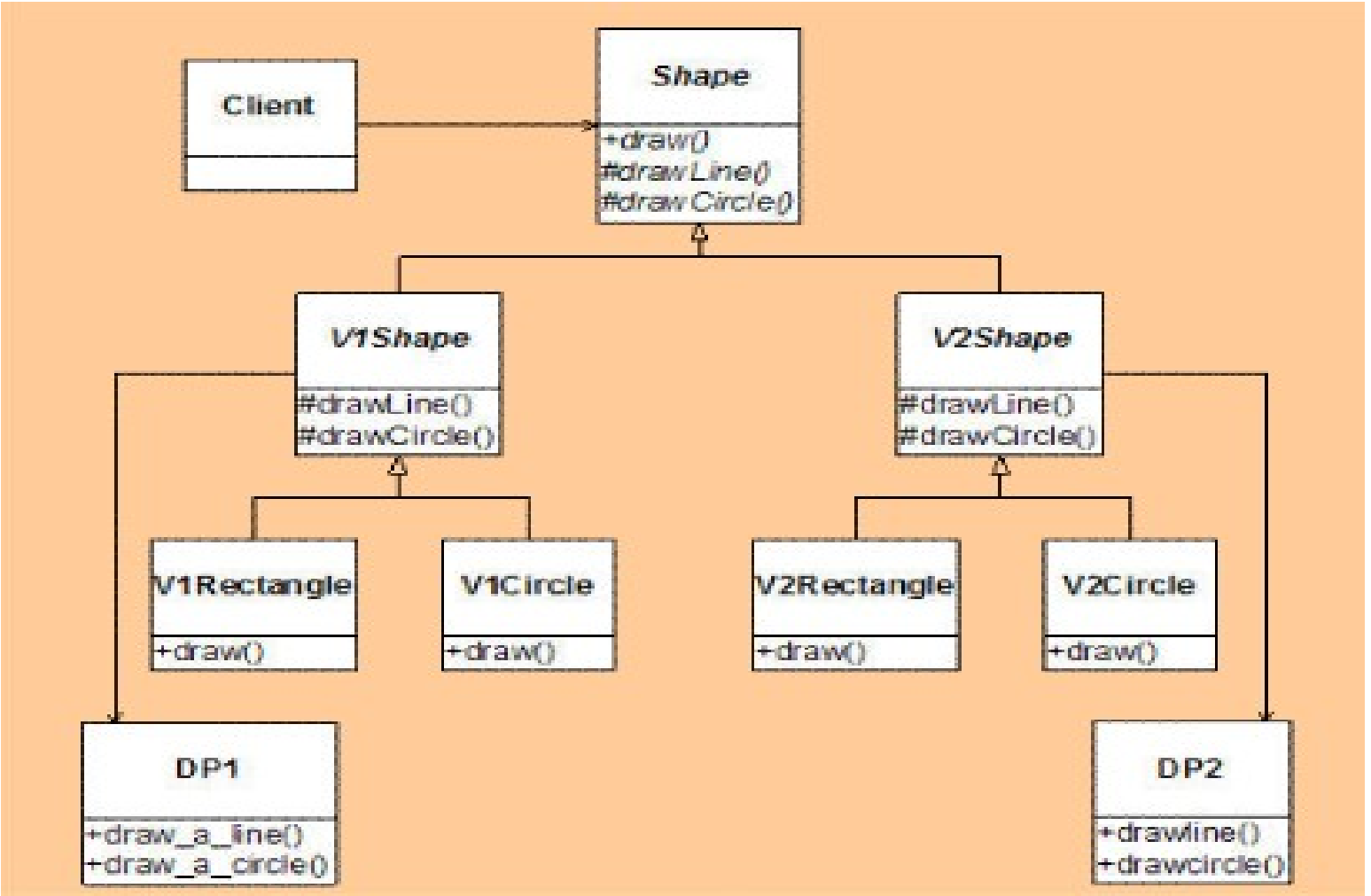
```
abstract class Shape {
  public abstract void draw();
}

abstract class Rectangle extends
Shape {
  public void draw() {
    drawline(_x1, _y1, _x2, _y1);
    drawline(_x2, _y1, _x2, _y1);
    drawline(_x2, _y2, _x1, _y2);
    drawline(_x1, _y2, _x1, _y1);
  }
  abstract void drawline(
      double x1, double y1,
      double x2, double y2);
}

class V1Rectangle extends
Rectangle {
  public void drawline(
      double x1, double y1,
      double x2, double y2){
    DP1.draw_a_line(x1, y1,
                    x2, y2);
```

```
class V2Rectangle extends Rectangle {
  public void drawline(
         double x1, double y1,
         double x2, double y2){
    DP2.drawline(x1, x2,
                 y1, y2);
  }
}

abstract class Circle extends Shape {
}

class V1Circle extends Circle {
  public void draw(){
    DP1.draw_a_circle(_x, _y, _r);
  }
}

class V2Circle extends Circle {
  public void draw(){
    DP2.drawcircle(_x, _y, _r);
  }
}
```

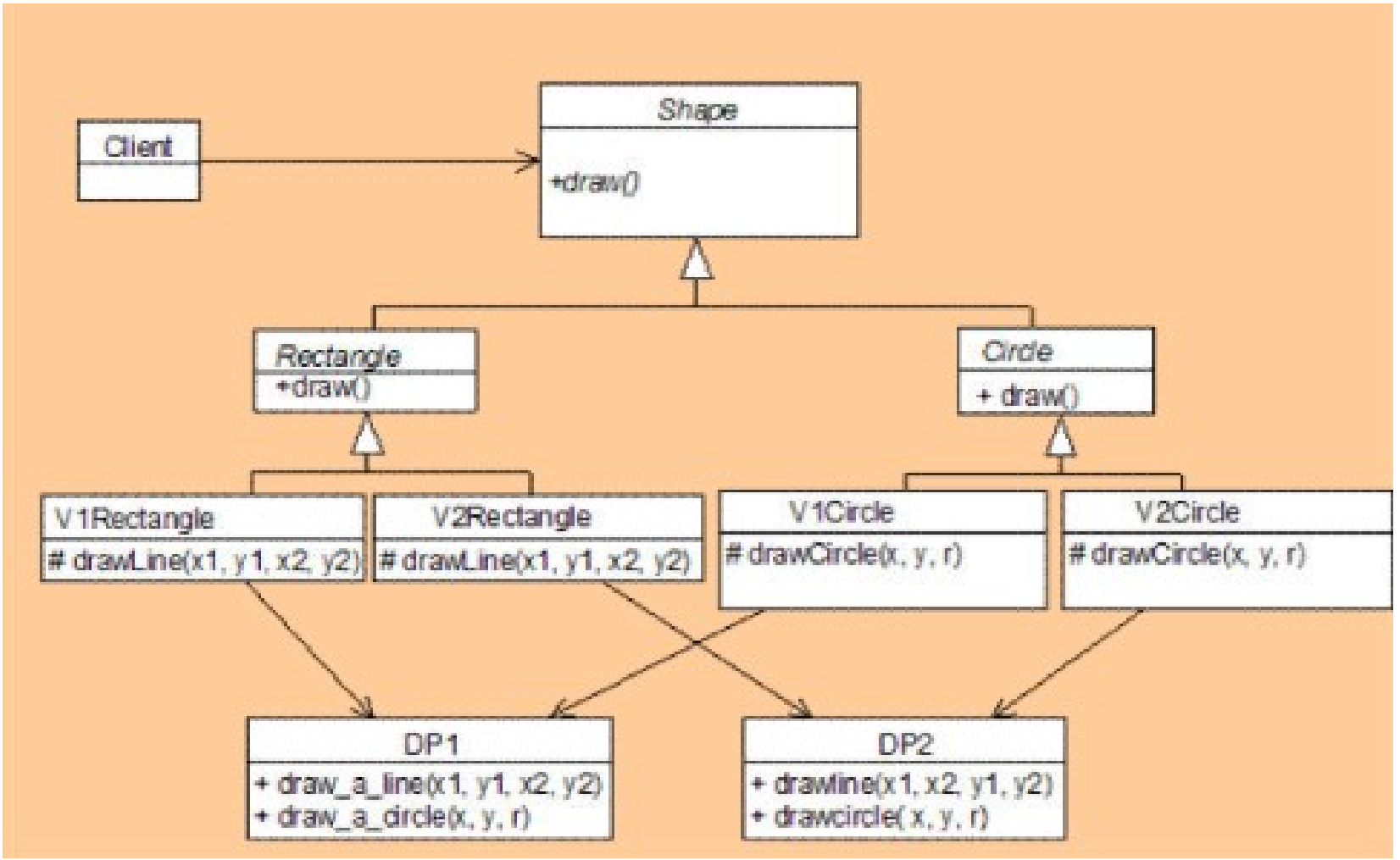
Problems With Approach

- Doesn't scale well
  - If we add another type of Shape, we have 6 implementations
  - The number of implementations will equal the # of types of Shapes **times** the # of drawing programs!
- Redundant
  - The use of DP1 by both V1Rectangle and V1Circle is a redundant relationship
  - There may also be many redundancies between V1Rectangle and V2Rectangle, etc...
- Static
  - Must make new objects when any issue varies at runtime
- Confusing and Complex
  - Poor cohesion makes classes hard to follow

Maybe Change the Hierarchy?



Don't Trade One Redundancy for Another...



Perhaps the Intent Makes Sense Now

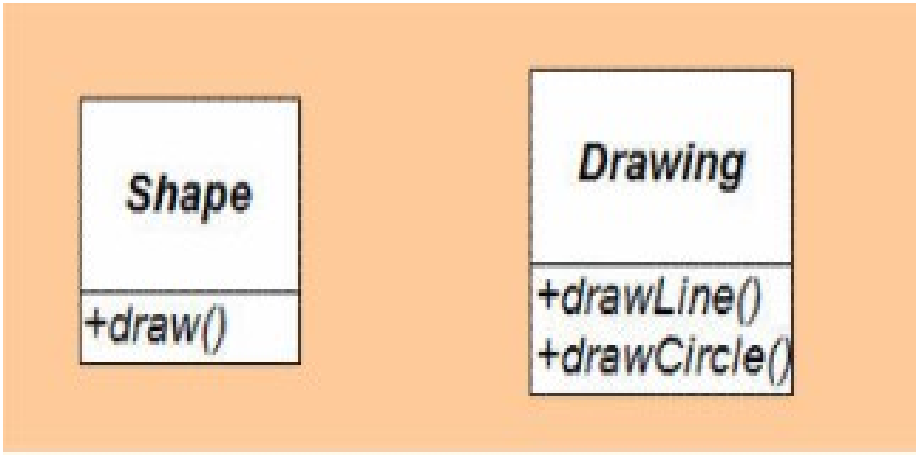
- GoF Intent: Decouple an abstraction from its implementation so that the two can vary independently
- Abstraction: The main responsible entity that the client is aware of. In this case – Shape.
- Implementation: The way the main responsible entity fulfills one aspect of its responsibilities. In this case, the drawing aspect – Drawing.
- In other words, how can we set it up to add new versions of our Shapes OR new Drawing Programs easily, each without effecting the other?
- NOTE: we can (hopefully) see that the Bridge Pattern potentially applies without knowing anything about how it is implemented :-)

Deriving the Bridge Pattern

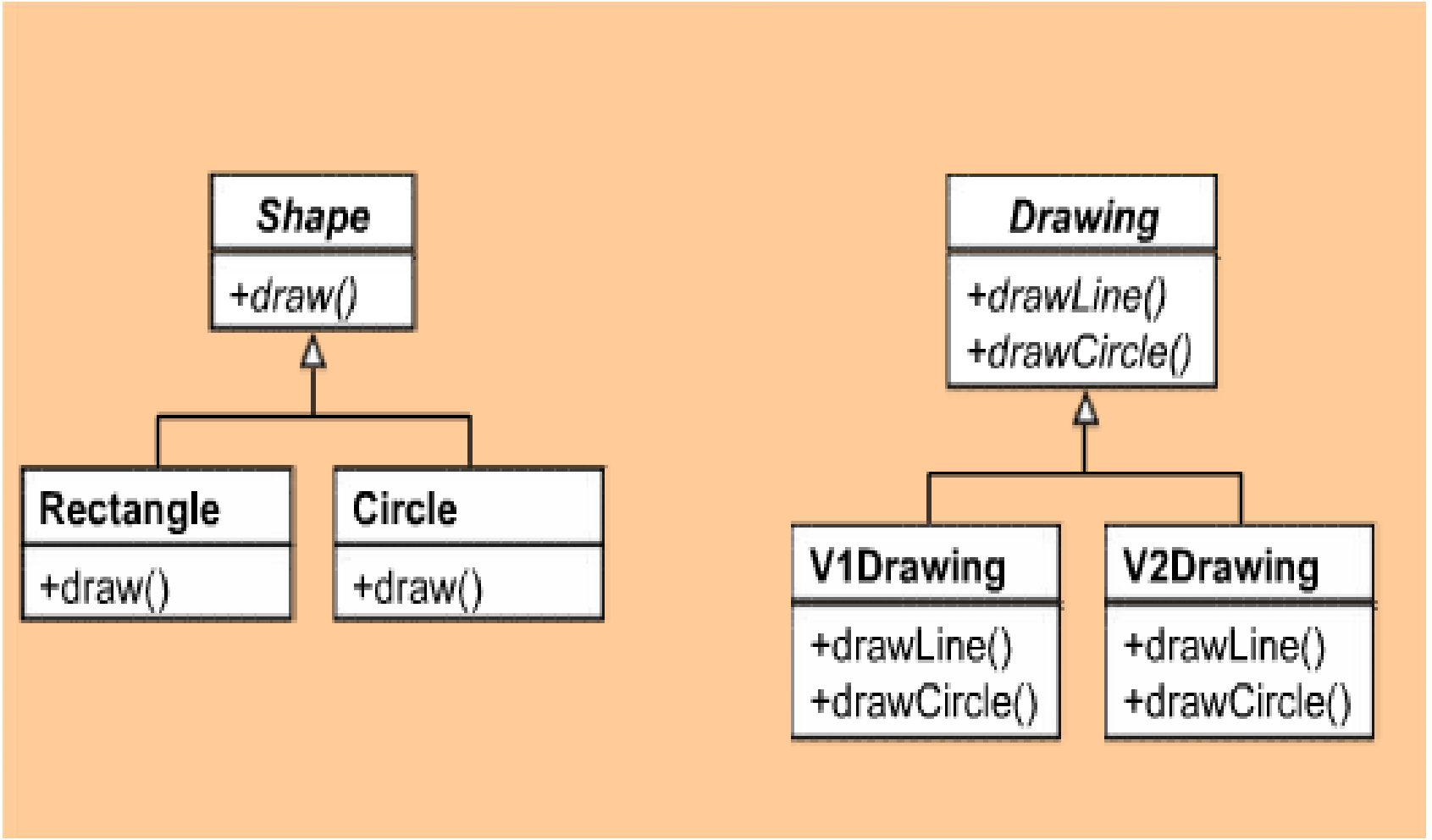
- We can use the following approach to derive the Bridge Pattern:
  - Find what is common in the problem domain
  - Find which of the commonalities vary, and how they vary
  - Identify the concept of what is varying – that is, what concept can incorporate all of the variation
  - Define an abstract class to represent each common concept
  - Define concrete classes for each of these abstractions that represent a particular variation
  - Observe how these abstractions relate to each other

We have the Following Commonalities

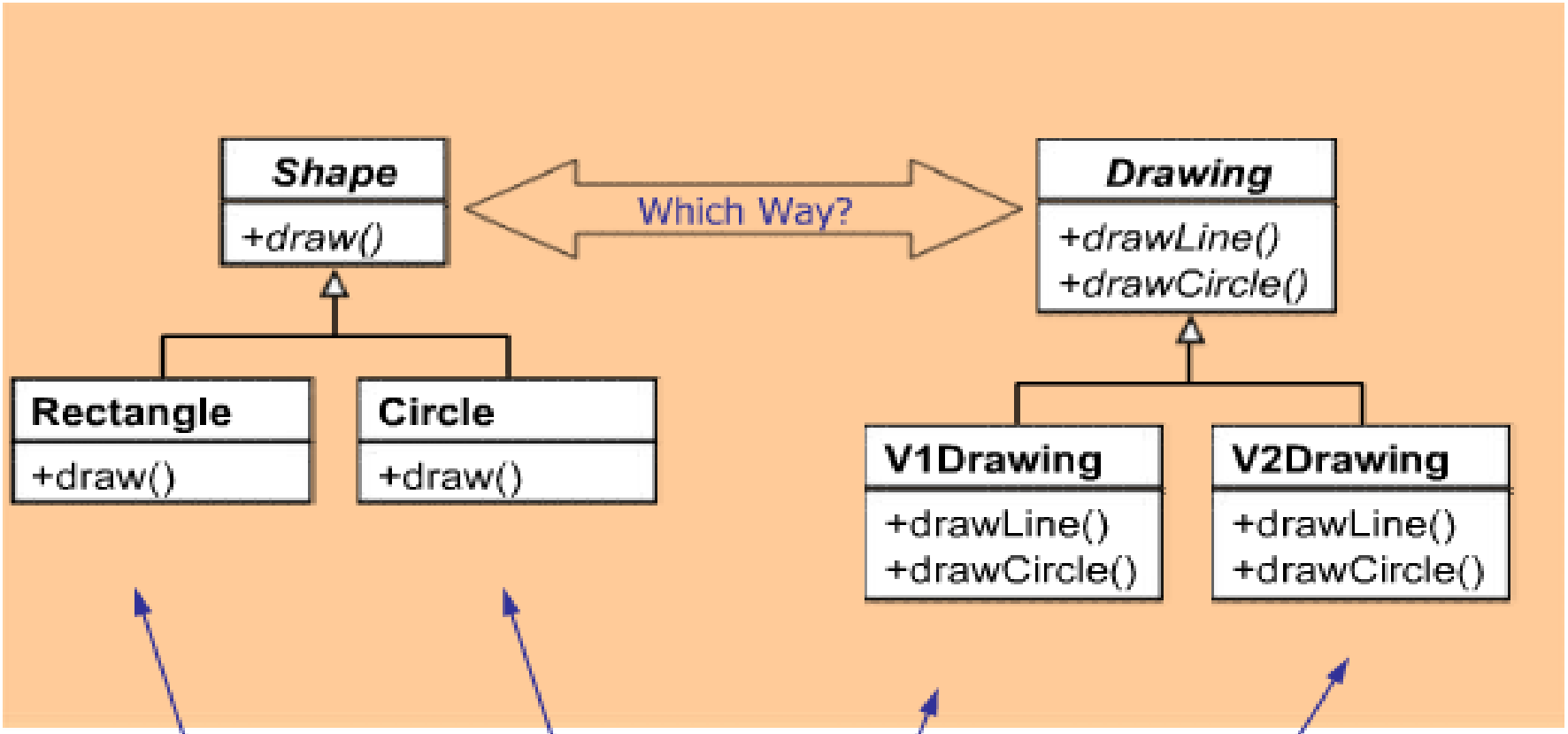
- Our Shape class represents our commonality of abstraction. In fact, this class is already abstracted and derived!
- We also have an implementation which varies:



Derive the Variations

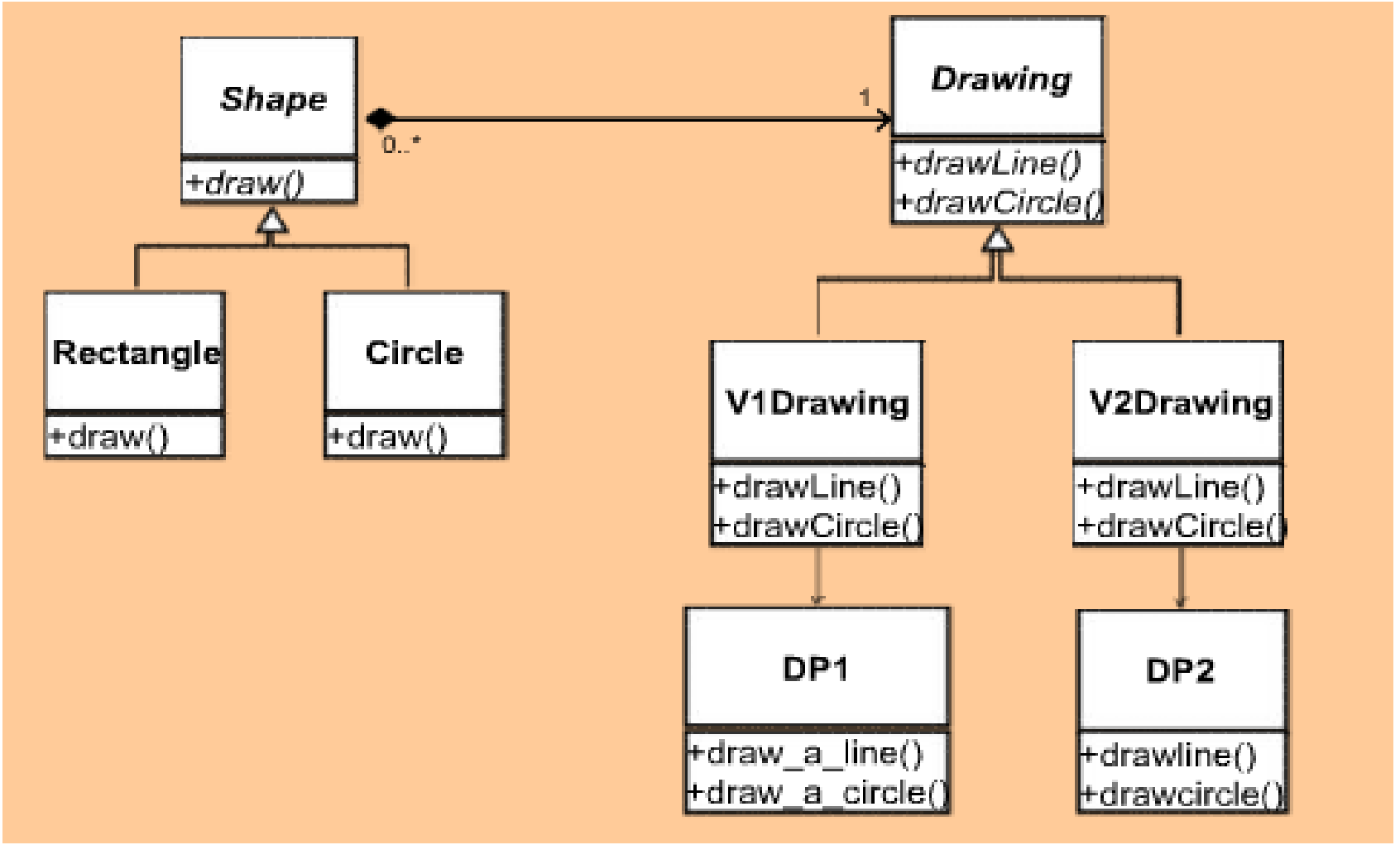


Determine the Relationship

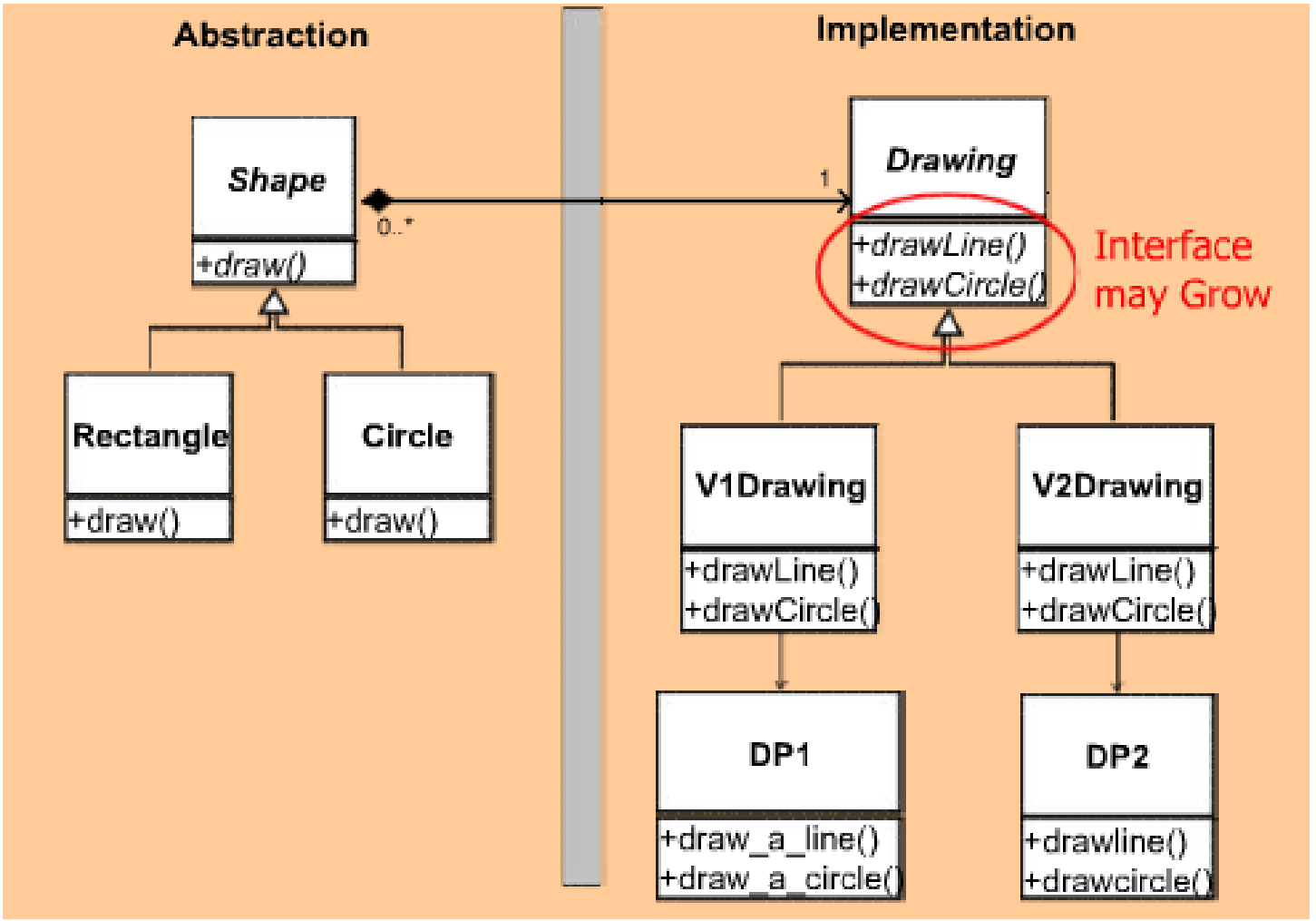


Don't want them down here: "Design to Interfaces"

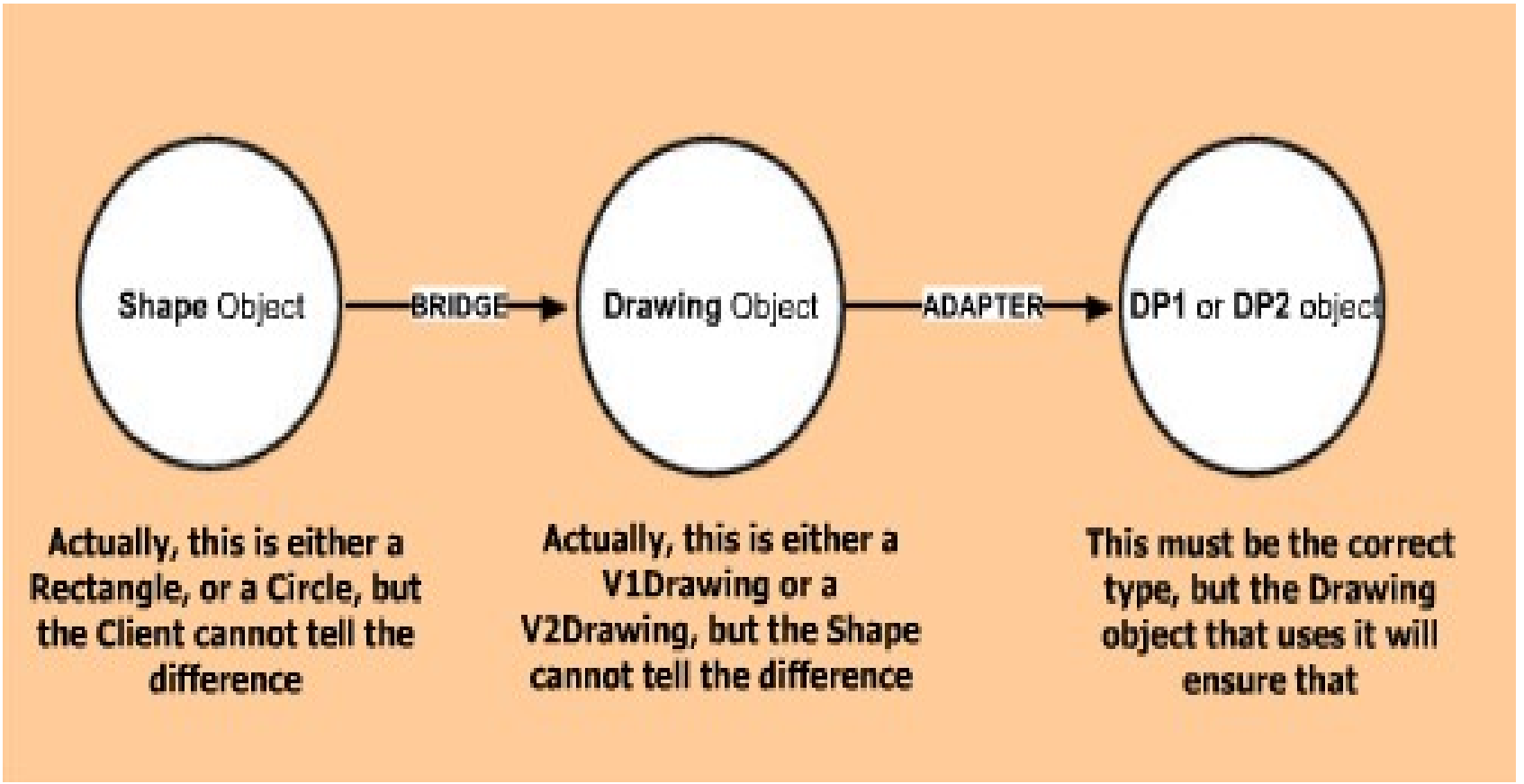
Apply the Bridge



Illustrating the Separation



Another Way to Look At It



Example Implementation

```
class Client {

   public static void main
   (String argv[]) {
      Shape s1;
      Shape s2;
      Drawing dp;

      dp= new V1Drawing();
      s1= new Rectangle(dp,1,1,2,2);

      dp= new V2Drawing();
      s2= new Circle( dp,2,2,4);

      s1.draw();
      s2.draw();
   }
}
```

```
abstract class Shape {
  Drawing _dp;

  Shape (Drawing dp) {
      _dp= dp;
  }

  abstract void draw ();

  protected void drawLine (double
   x1,double y1,double x2,double y2) {
    _dp.drawLine( x1, y1, x2, y2);
  }

  protected void drawCircle (double
   x,double y, double radius){
    _dp.drawCircle( x, y, radius);
  }
}
```

NOTE: imports have not been included

Example Continued

```
class Rectangle extends Shape {
  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, _x2, _y1);
      drawLine(_x2, _y1, _x1, _y1);
   }
}

class Circle extends Shape {
   Circle (Drawing dp, double x,
     double y, double radius) {
      super( dp);
      _x= x; _y= y; _radius= radius;
   }
   public void draw () {
      drawCircle( _x, _y, _radius);
   }
}
```

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

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 radius) {
      DP1.draw_a_circle( x, y, radius);
   }
}

class V2Drawing extends Drawing {
   public void drawLine (double x1, double y1,
                  double x2, double y2) {
      DP2.drawline( x1, x2, y1, y2);
   }
   public void drawCircle (double x, double y,
                   double radius) {
      DP2.drawcircle( x, y, radius);
   }
}
```

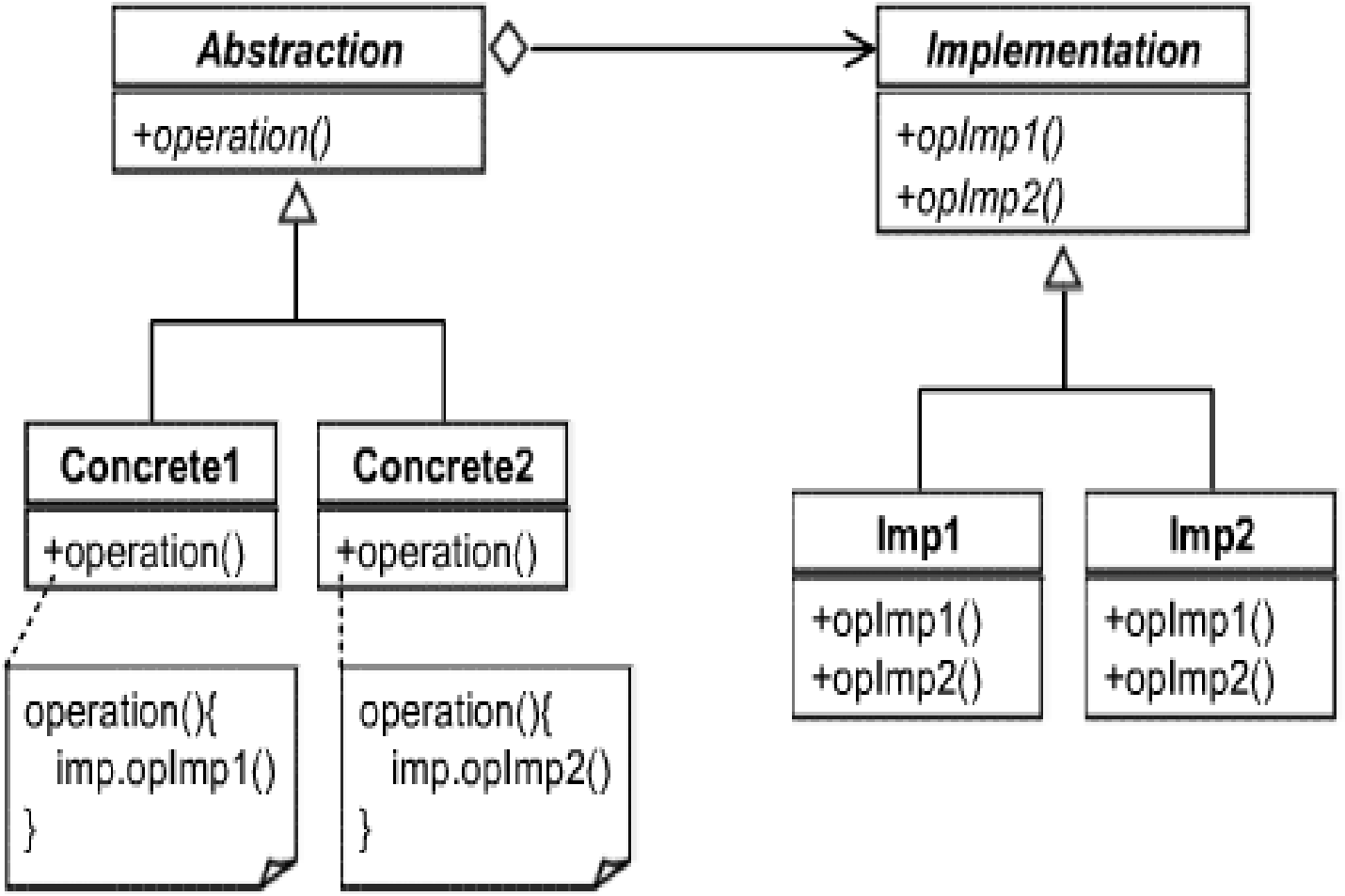
Example Continued

```
class DP1 {
   static void draw_a_line { double x1, double y1,
                             double x2, double y2} {
      // draw_a_line implementation given to us
   }
   static void draw_a_circle{ double x, double y,
                              double radius} {
      // draw_a_circle implementation given to us
   }
}

class DP2 {
   static void drawline { double x1, double y1,
                          double x2, double y2} {
      // drawline implementation given to us
   }
   static void drawcircle{ double x, double y,
                           double radius} {
      // drawcircle implementation given to us
   }
}
```

A Note on Implementing the Bridge

- If you compare the before and after implementations, you will see how having the protected methods in Shape access the DP1 drawing program greatly simplified the implementation of the Bridge Pattern
- All that was modified was:
  - The constructors
  - The protected methods
- The essence of the pattern is defining the interface for the implementations to use that satisfies the needs of the concrete classes derived from the abstract class

Almost Done: The Canonical Bridge Pattern (again)



Another Way to Think of the Bridge

- Consider the following:
  - The cost of having a common interface for the implementations
  - The saving such a common interface would provide
  - The likelihood of new implementations that would benefit from a common implementation interface
- If the resulting savings are greater than the cost of implementing the design, then implement it
- The pattern is not **just** the design
- It is also the above consideration

Bridge and Adapter

- Bridge is often confused with the Adapter pattern
- Recall that Adapter should be used when you have an existing class whose interface is incompatible with another class or set of classes. Adapter provides an interface to the class you want to use. This is done through composition/aggregation
- Clearly Bridge involves adaption between different entities, but its intent is to allow a relation between those entities where both need to change and you want things to remain loosely coupled