- Observer Pattern
- Bridge Pattern

Observer Pattern

Observer Pattern (Behavioral Pattern):

- Intent (GOF): Define a one-to-many dependency between objects so that when one object chnages state, all its dependents are notified and updated automatically.

Also known as Publish-subscribe

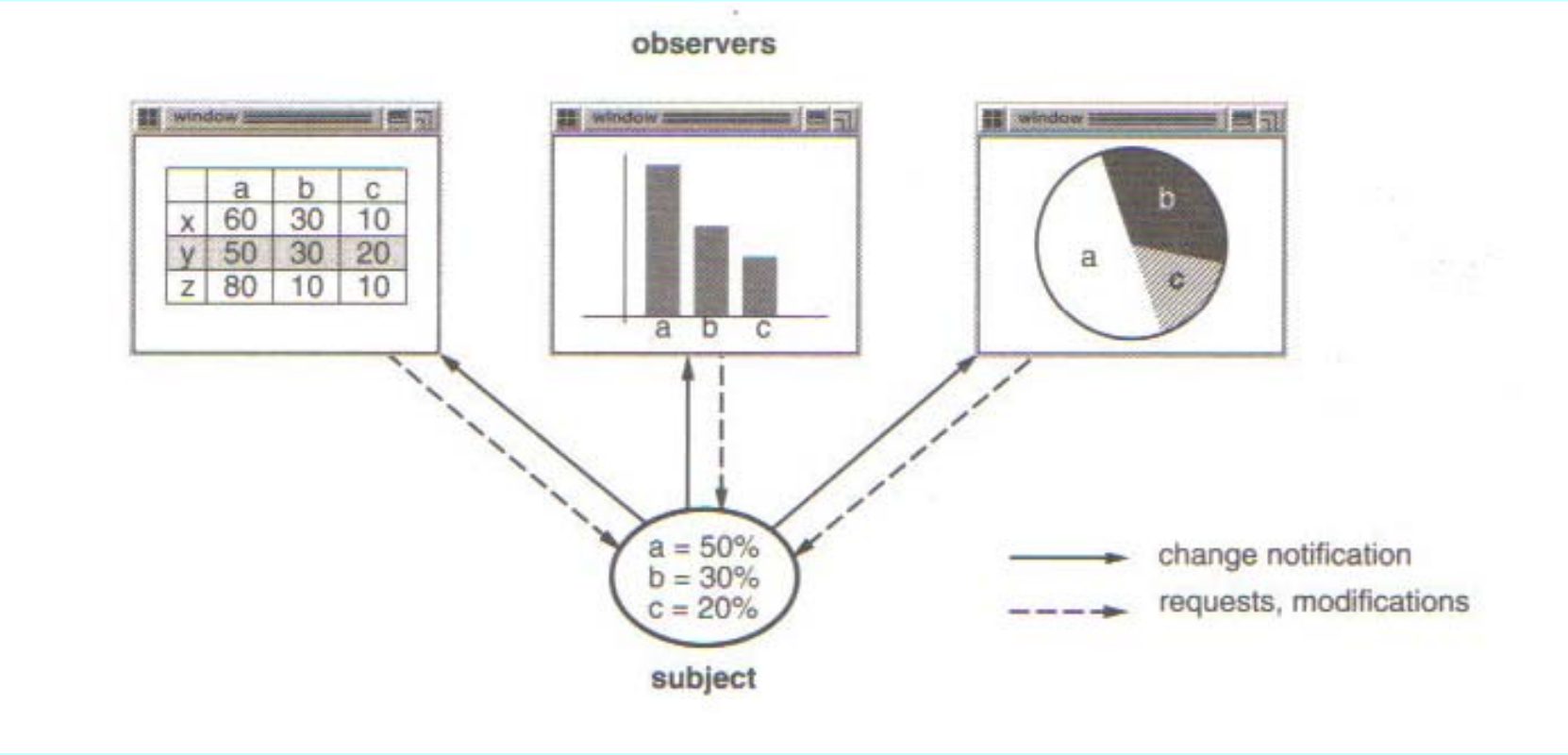
- Motivation: lets see an example..

Example:

Many GUI toolkits separate the presentational aspects of the UI from the underlying application data..

A spreadsheet object and a bar chart object can depict information in the same application data object using different presentations. They don't know about each other thereby letting you reuse only the one you need. But they *behave* as they do….

When the user changes the information in the spreadsheet, the bar chart reflects the changes..


observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications

This implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state.

There can be more number of dependent objects…

Solution: Use Observer pattern. Key objects: Subject and Observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subjects's state.

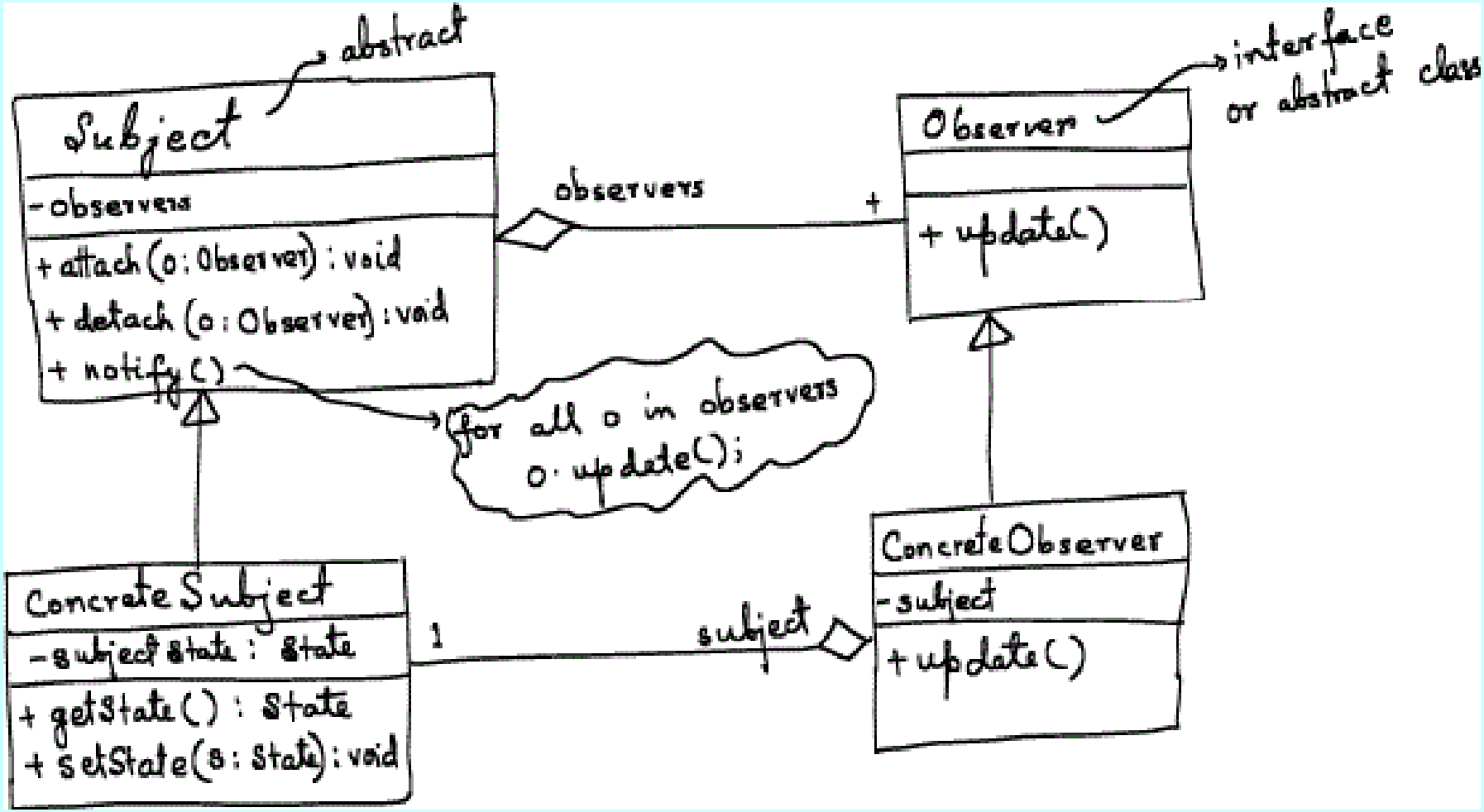
Also called publish-subscribe. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Observer Pattern (contd.):

- **Applicability:** Use it in following cases:
  - when an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - when a change to one object requires changing others and you don't know how many objects need to be changed.
  - when an object should be able to notify other objects without making assumptions about who these objects are. You don't want these objects tightly coupled.

Observer Pattern (contd.):

- **Generic Structure:**



Observer Pattern (contd.):

- Participants:

  - Subject: provides an interface for attaching and detaching Observer objects

  - Observer: defines an updating interface for objects that should be notified of changes in a subject.

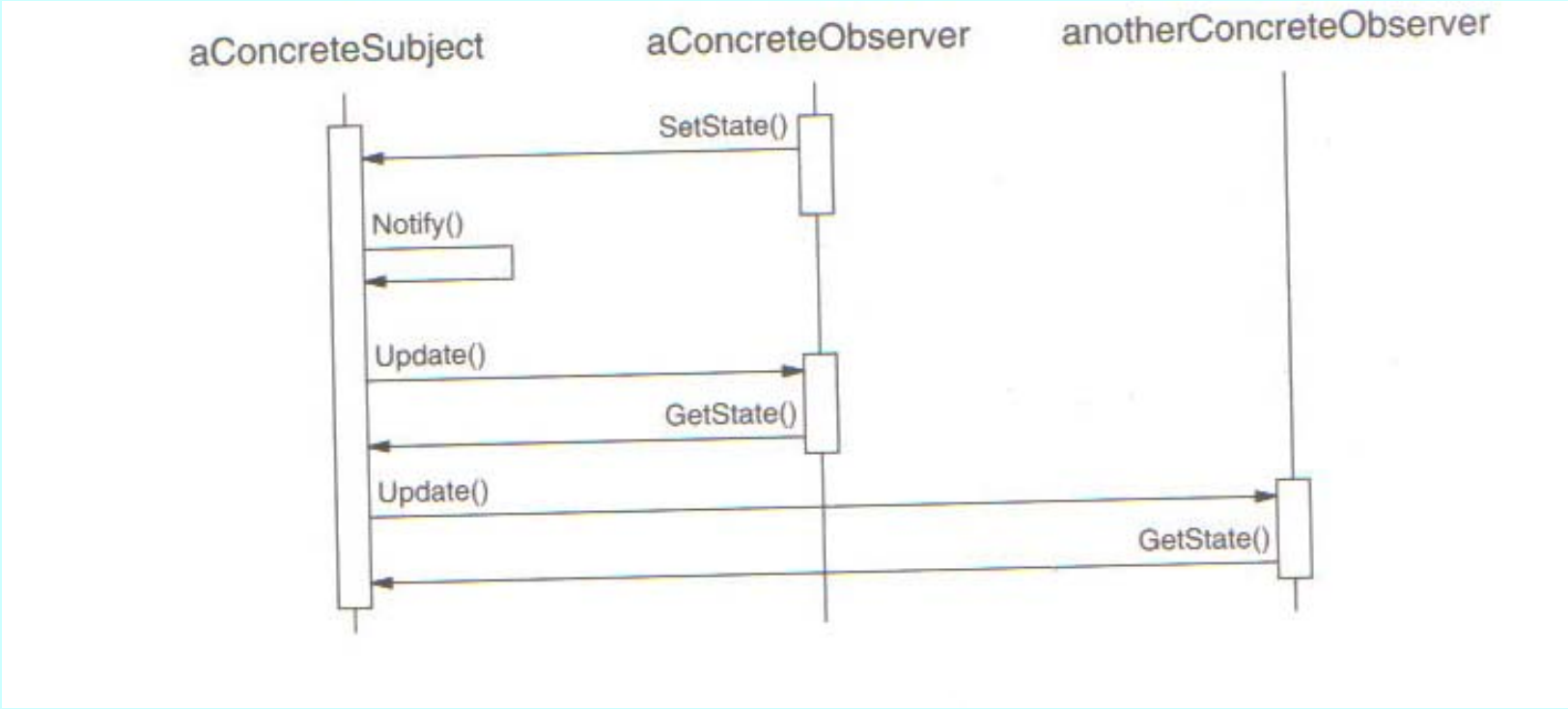
  - ConcreteSubject: stores state of interest to ConcreteObserver objects. It sends notification to its observers when its state changes.

  - ConcreteObserver: maintains a reference to a ConcreteSubject object. It stores state that should stay consistent with the subject's. It implements the Observer updating interface.

Observer Pattern (contd.):

- **Collaborations:**

  •Note how the Observer object that initiates the change request postpones its update until it gets notification from the subject.



Observer Pattern (contd.):

- **Consequences:**
  - can reuse subjects without reusing their observers, and vice versa.
  - lets you add observers without modifying the subject or other observers.
  - minimal coupling between subjects and observers.
  - support for broadcast communication
  - unexpected updates

Observer Pattern (contd.):

- **Implementation:**

  - Have Observers that want to know when an event happens attach themselves to Subject that is watching for the event to occur or that triggers the event itself.

  - When the event occurs, the Subject tells the Observers that it has occurred.

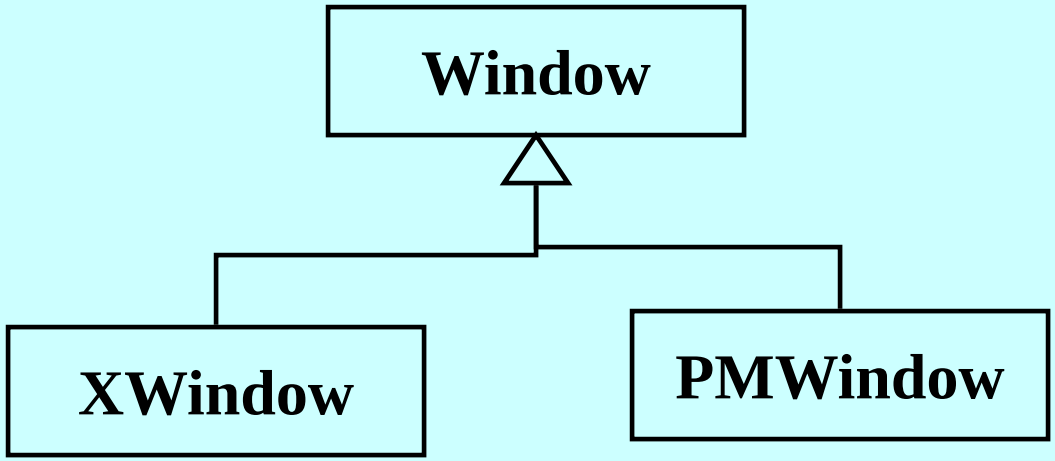
Bridge Pattern

Bridge Pattern (Structural Pattern):

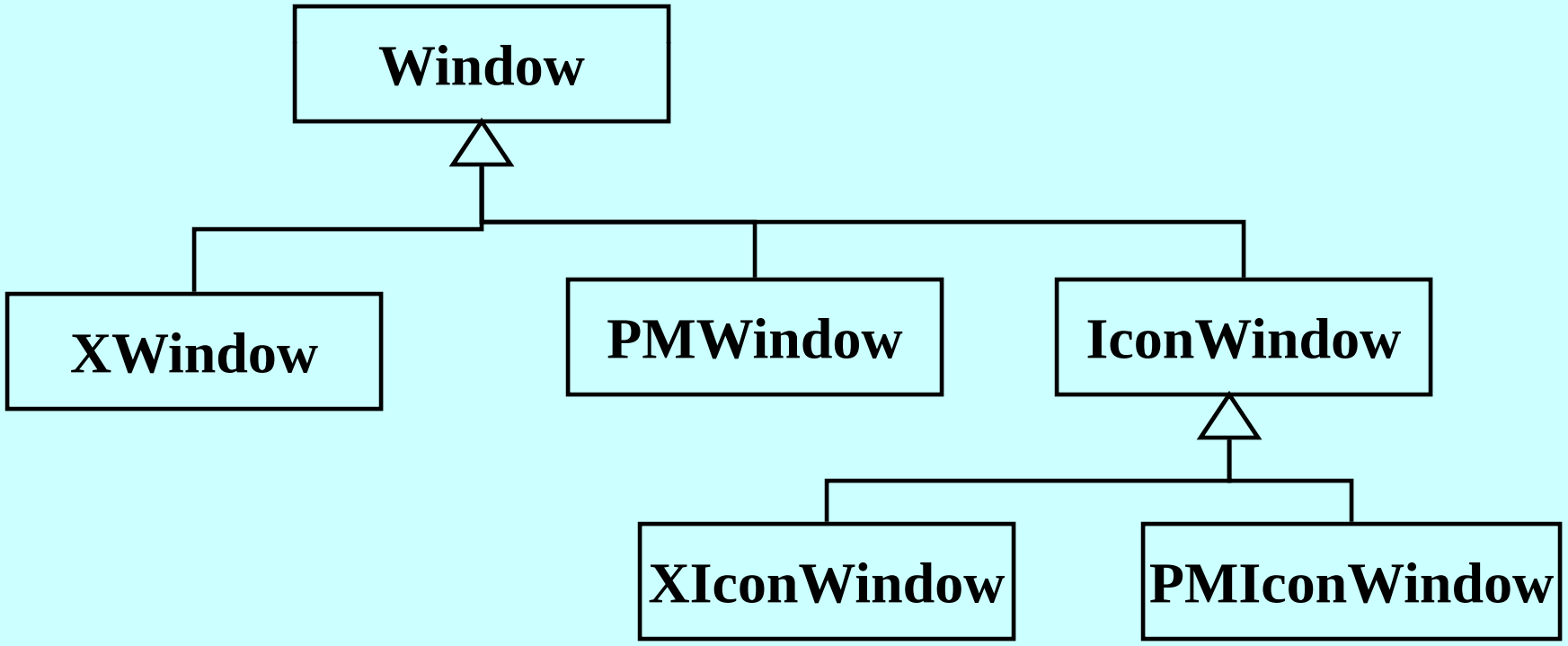
- **Intent:** Decouple an abstraction from its implementation so that the two can vary independently.

- **Motivation:** lets see an example.

Example: **Consider the implementation of a portable Window abstraction in a UI toolkit. This abstraction should enable us to write applications that work on both the XWindow system and Presentation Manager(PM).**



Drawback: inconvenient to extend the Window abstraction to cover different kinds of Windows, e.g. IconWindow that specializes the Window abstraction for icons. Here, we have to define two classes to *every* kind of window. Supporting a third platform…..



Drawback: Makes client code platform-independent. Whenever a client creates a window, it instantiates a concrete class that has a specific implementation. Makes it harder to port client code to other platforms.

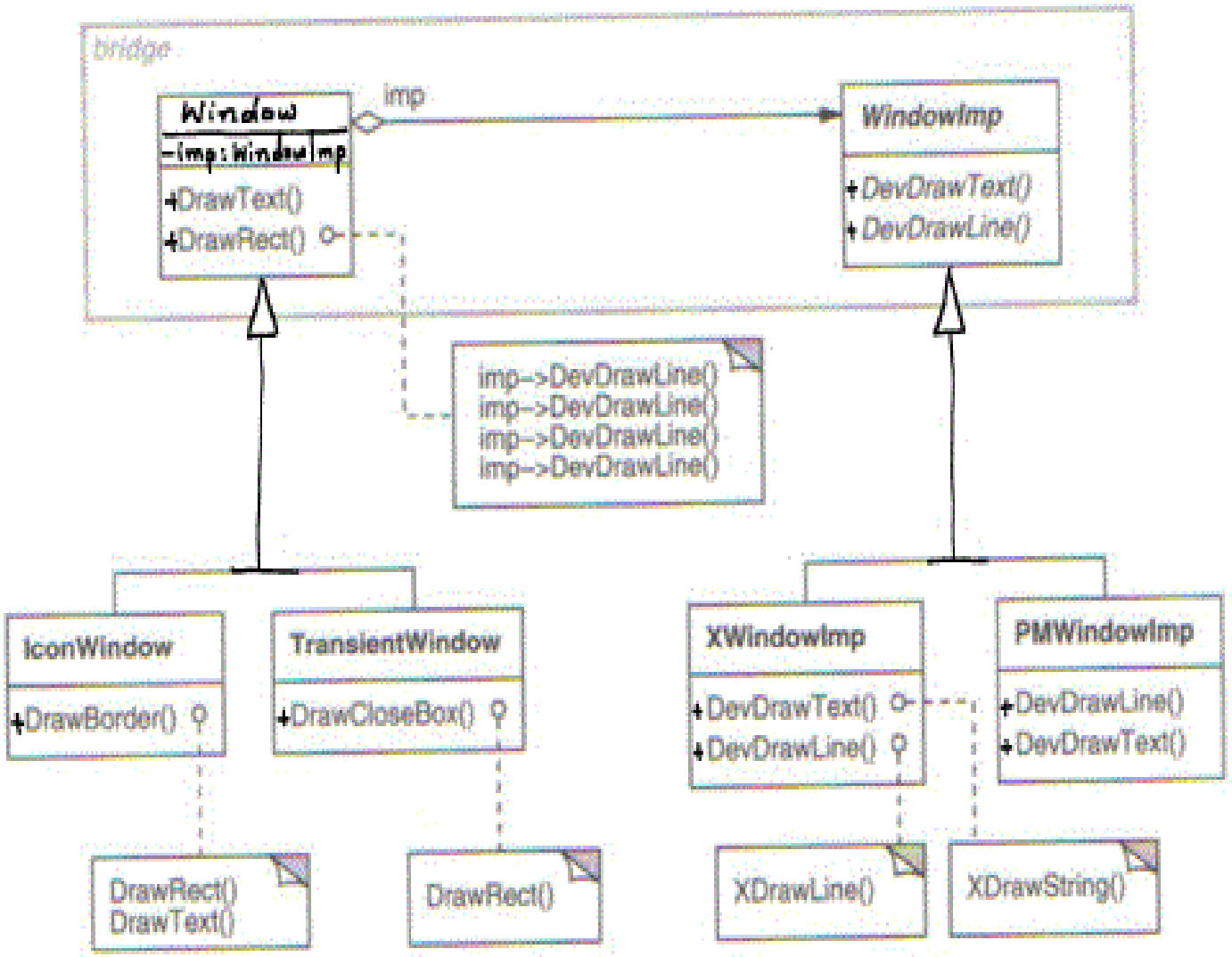
Ideally, Client should be able to create a window without committing to concrete implementation.

Solution: Use Bridge pattern.

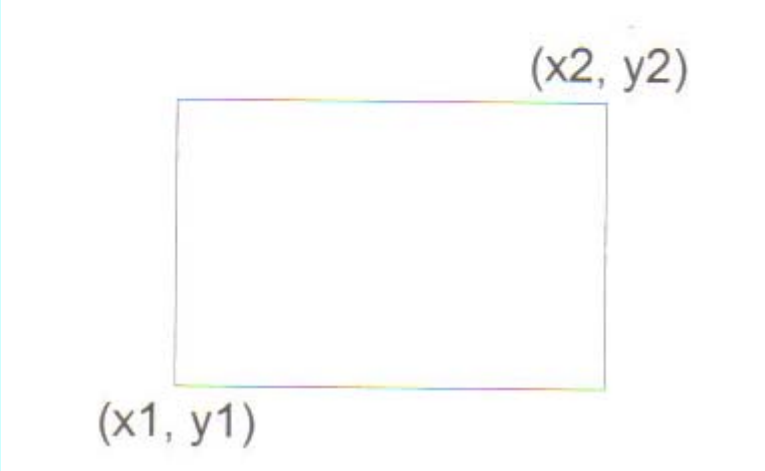
It puts the Window abstraction and its implementation in separate class hierarchies. There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations (WindowImp, XWindowImp, PMWindowImp).

All operations on *Window* subclasses are implemented in terms of abstract operations from *WindowImp* interface. This decouples the window abstractions from various platform-specific implementations.

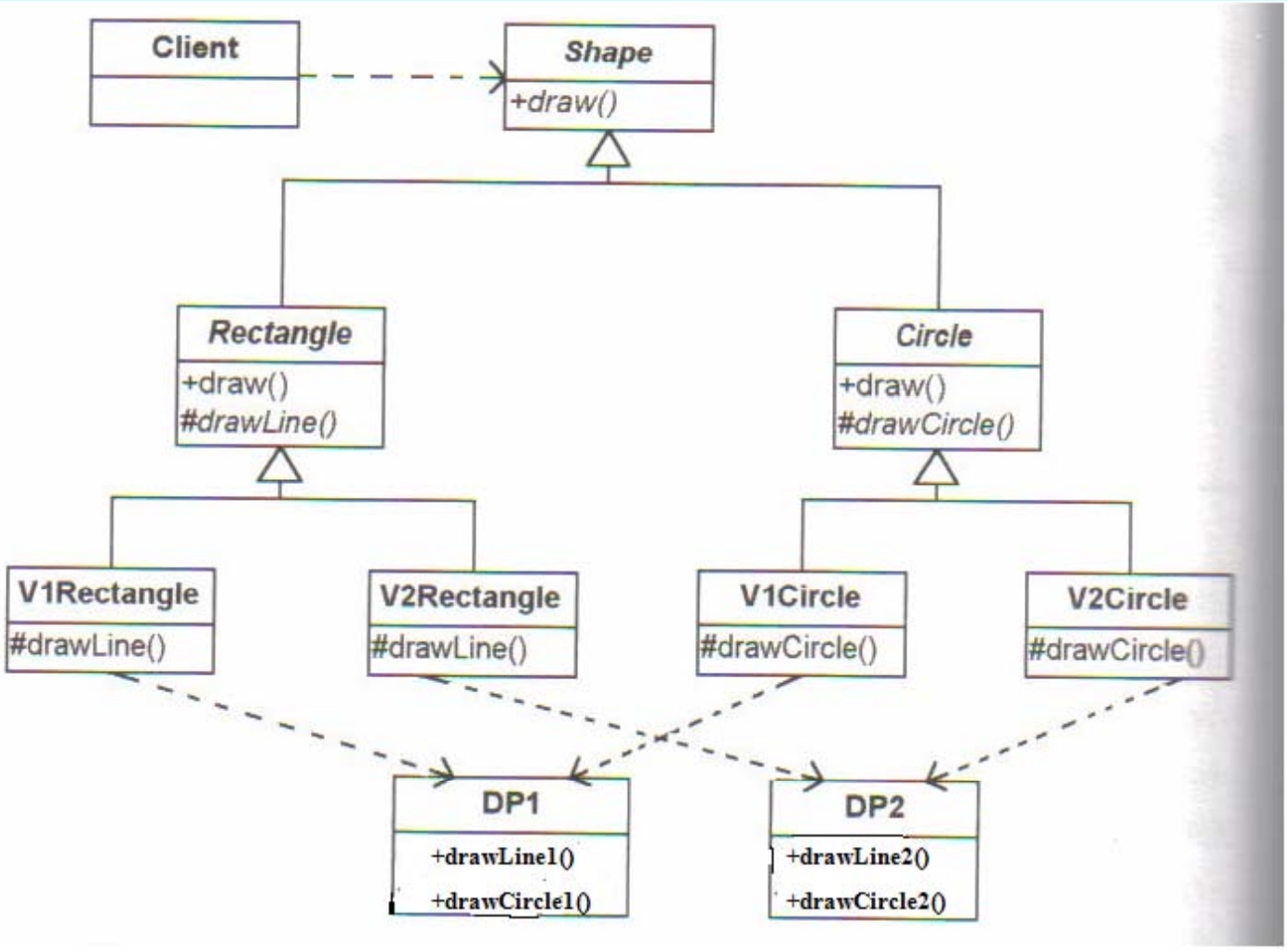
Relationship between Window and WindowImp: bridge (bridges the abstraction and implementation)


bridge
Window
-imp:WindowImp
+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: Suppose I have been given of writing a program that will draw rectangles with either of the two drawing programs. I have been told that when I instantiate a rectangle, I will know whether I should use drawing program 1 (DP1) or drawing program 2(DP2).


(x2, y2)
(x1, y1)

Alternative-1:



```
public abstract class Shape {
  public abstract void draw();
}
public abstract class Rectangle extends Shape {
  private double x1, x2, y1, y2;
  public Rectangle(double i, double j,
                   double k, double l) {
      x1 = i; x2 = j; y1 = k; y2 = l;
  }
```

```
  public void draw() {
    drawLine(x1, y1, x2, y1);
    drawLine(x2, y1, x2, y2);
    drawLine(x2, y2, x1, y2);
    drawLine(x1, y2, x1, y1);
  }

  abstract protected void drawLine(double i,
                                   double j, double k,
                                   double l);
}
```

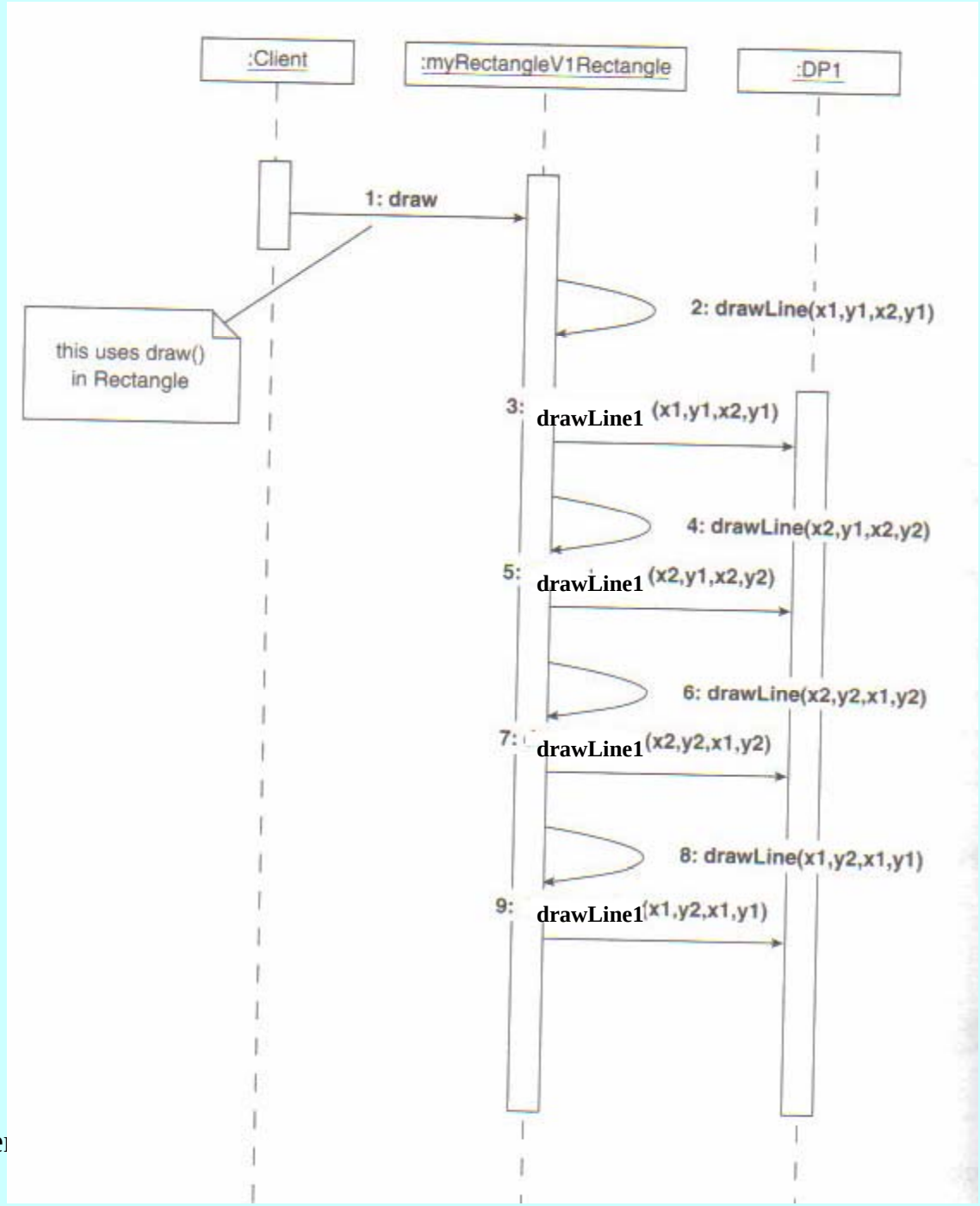
```
public class V1Rectangle extends Rectangle {

  private DP1 dp = new DP1();
  public V1Rectangle(double i, double j,
                     double k, double l) {
    super(i, j, k, l);
  }

  protected void drawLine(double i, double j,
                          double k, double l) {
    dp.drawLine1(i, j, k, l);
  }

}
```

Similarly V2Rectangle…


:Client
:myRectangleV1Rectangle
:DP1
1: draw
this uses draw()
in Rectangle
2: drawLine(x1,y1,x2,y1)
3: drawLine1 (x1,y1,x2,y1)
4: drawLine(x2,y1,x2,y2)
5: drawLine1 (x2,y1,x2,y2)
6: drawLine(x2,y2,x1,y2)
7: drawLine1(x2,y2,x1,y2)
8: drawLine(x1,y2,x1,y1)
9: drawLine1(x1,y2,x1,y1)

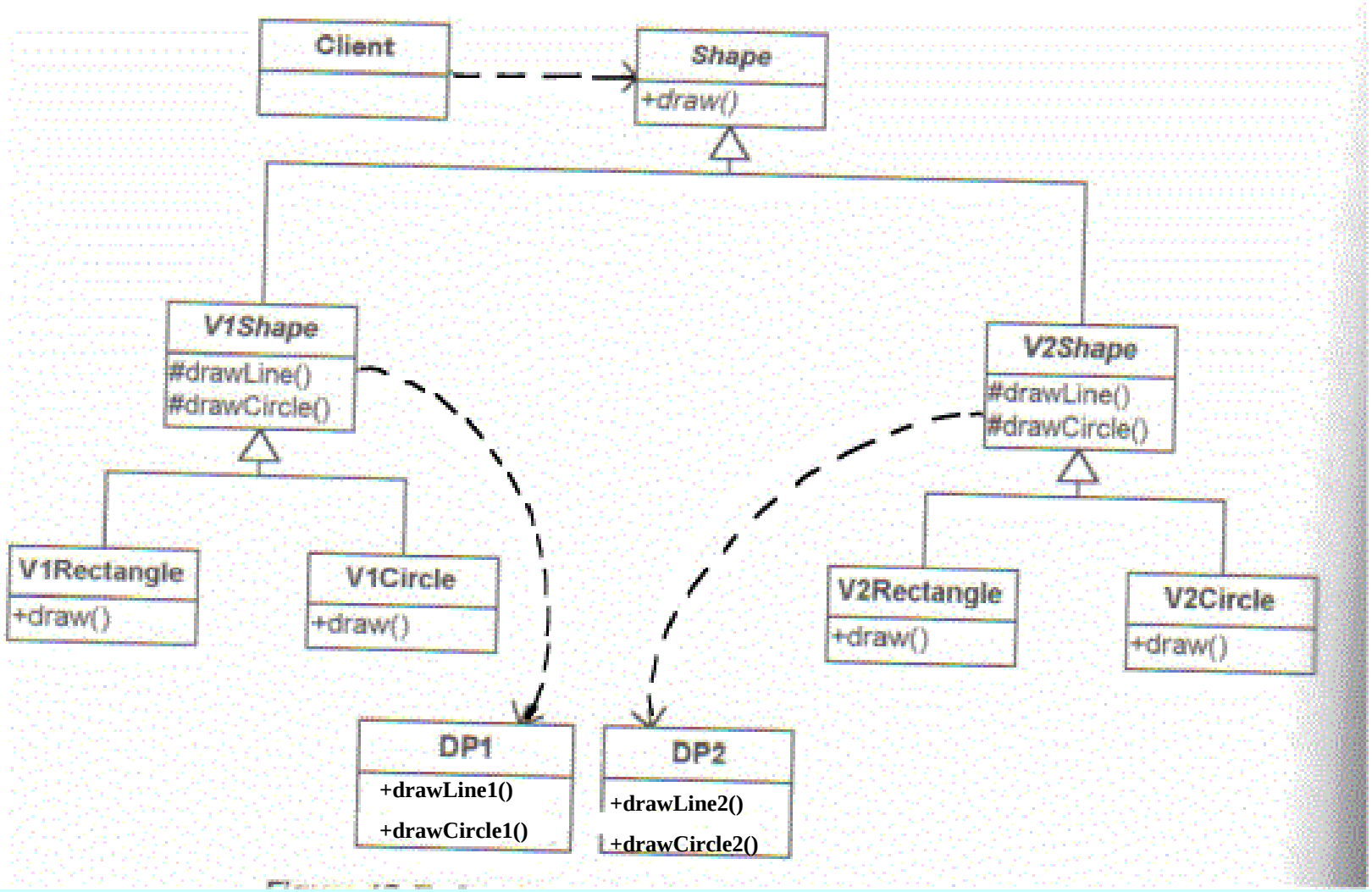
This approach has some problems:

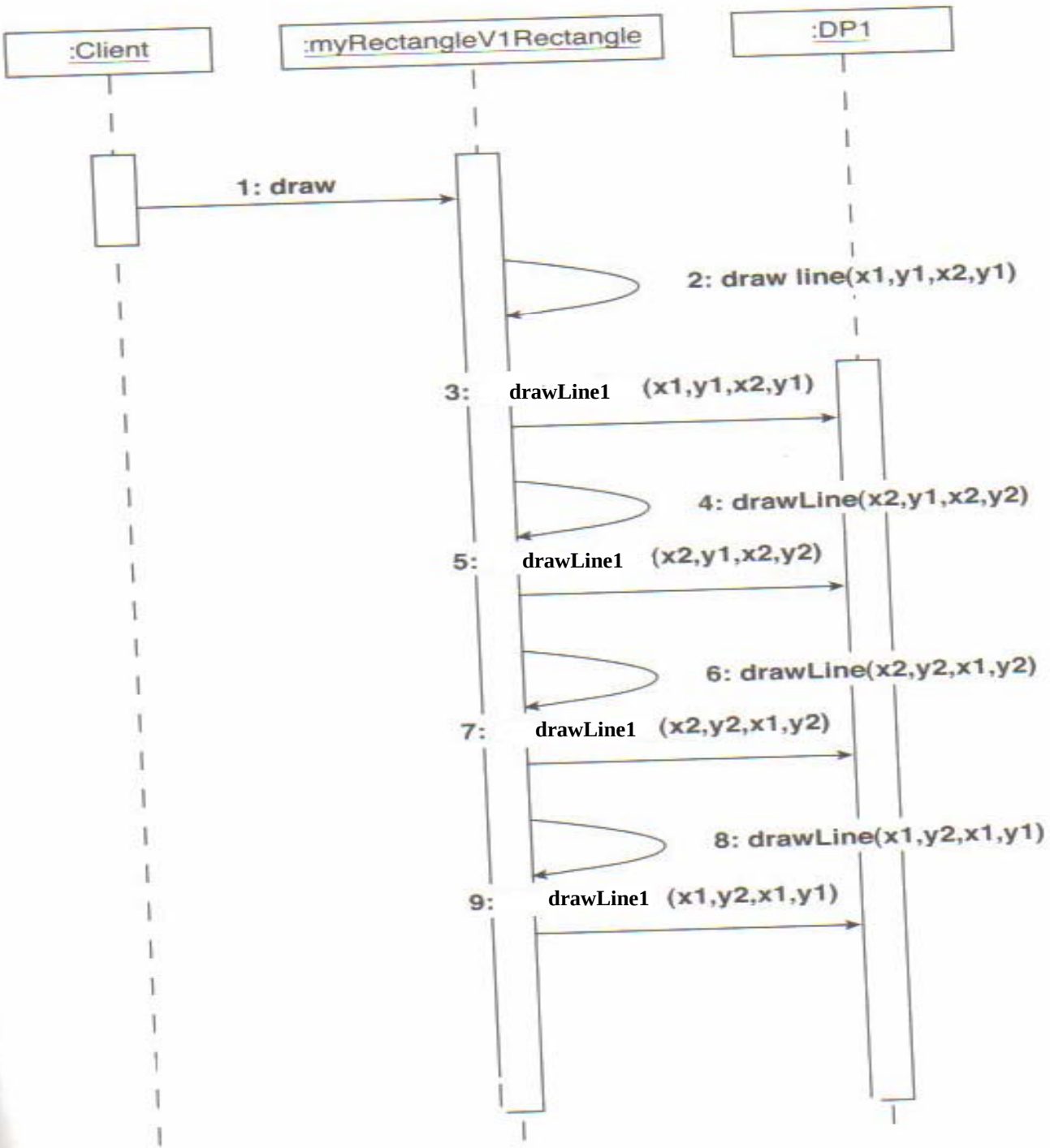
- what happens if I get another drawing program – another variation of the implementation ?
- what happens if I get another type of Shape – another variation in concept or abstraction ?

Explosion problem: arises because in this solution the abstraction (the kinds of Shape) and the implementation (the drawing programs) are tightly coupled.

Each type of Shape must know what type of drawing program it is using.

Alternative-2:

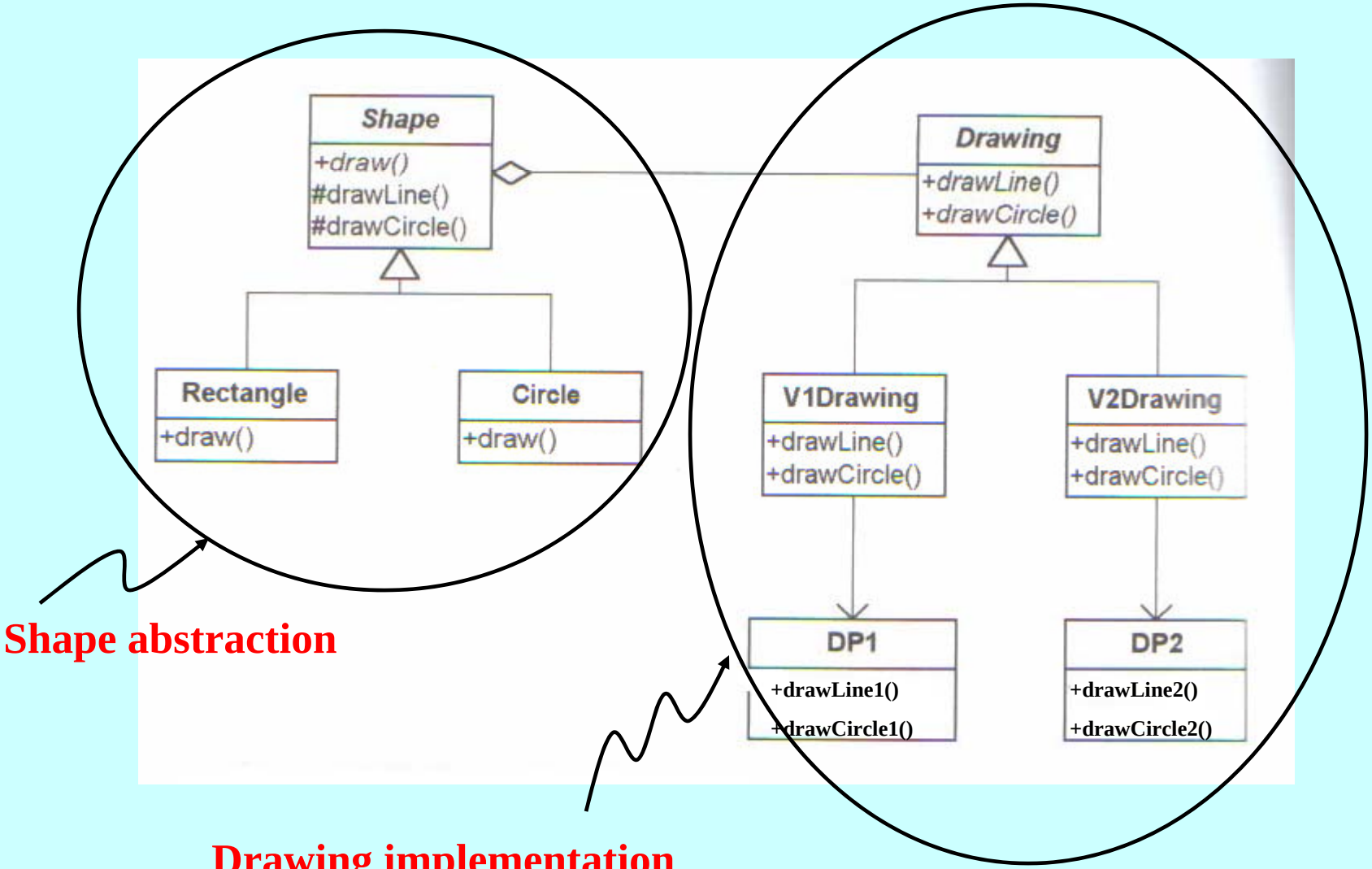



:Client
:myRectangleV1Rectangle
:DP1
1: draw
2: draw line(x1,y1,x2,y1)
3: drawLine1 (x1,y1,x2,y1)
4: drawLine(x2,y1,x2,y2)
5: drawLine1 (x2,y1,x2,y2)
6: drawLine(x2,y2,x1,y2)
7: drawLine1 (x2,y2,x1,y2)
8: drawLine(x1,y2,x1,y1)
9: drawLine1 (x1,y2,x1,y1)

This approach is improvement over Alternative-1. Still it has the problem of scaling.

- **what happens if I get another drawing program – another variation of the implementation ?**
- **what happens if I get another type of Shape – another variation in concept or abstraction ?**

Alternative 3: Using Bridge pattern



```
public class Client {
  public static void main(String[] args) {
    V1Drawing d = new V1Drawing();
    Shape s = new Rectangle(d, 0, 0, 5, 5);
    s.draw();
  }
}
```

```
public abstract class Shape {
  protected Drawing myDrawing;
  public Shape(Drawing d) {myDrawing = d;}
  public abstract void draw();
  protected void drawLine(double i, double j,
                          double k, double l) {
    myDrawing.drawLine(i, j, k, l);
  }
  protected void drawCircle(double i, double j,
                            double r) {
    myDrawing.drawCircle(i, j, r);
  }
}
```

```
public class Rectangle extends Shape {
  private double x1, x2, y1, y2;
  public Rectangle(Drawing d, double i,
                   double j,double k, double l) {
      super(d);
      x1 = i; x2 = j; y1 = k; y2 = l;
  }
  public void draw() {
    drawLine(x1, y1, x2, y1);
    drawLine(x2, y1, x2, y2);
    drawLine(x2, y2, x1, y2);
    drawLine(x1, y2, x1, y1);
  }
}
```

```
public class Circle extends Shape {
  private double x, y, r;
  public Circle(Drawing d, double i,
                double j, double k) {
      super(d);
      x = i; y = j; r = k;
  }

  public void draw() {
    drawCircle(x, y, r);
  }
}
```

```
public abstract class Drawing {
  public abstract void drawLine(double i,
                                double j,
                                double k,
                                double l);

  public abstract void drawCircle(double i,
                                  double j,
                                  double k);
}
```

```
public class V1Drawing extends Drawing {
  private DP1 dp = new DP1();

  public void drawLine(double i, double j,
                       double k, double l) {
    dp.drawLine1(i, j, k, l);
  }

  public void drawCircle(double i,
                         double j,
                         double k) {
    dp.drawCircle1(i, j, k);
  }
}
```

Similarly for V2Drawing…

Bridge Pattern (contd.):

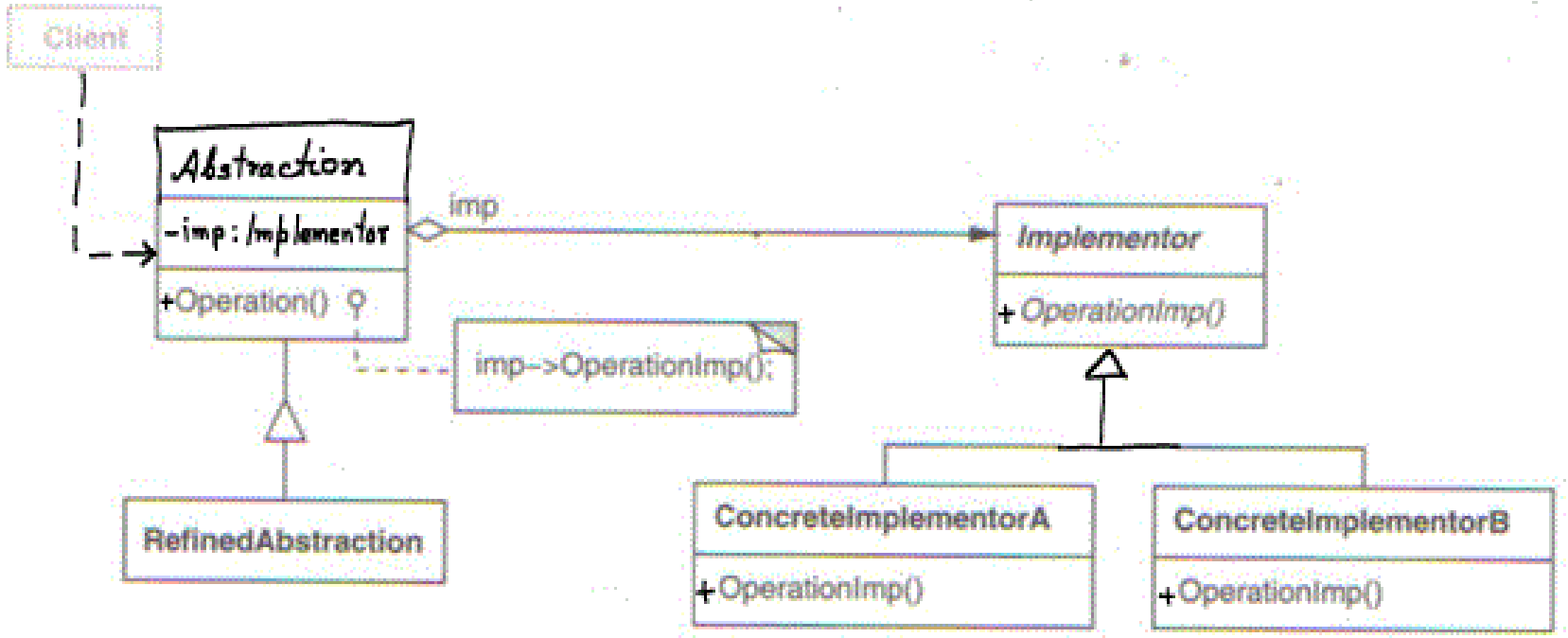
Applicability: Use it when:

- want to avoid permanent binding betwn. Abstraction and implementation

- both the abstractions and implementations should be extensible by subclassing.

Bridge Pattern (contd.):

- **Generic Structure:**



Bridge Pattern (contd.):

- Participants:
  - Abstraction: maintains a reference to an object of Implementor
  - RefinedAbstraction
  - Implementor: defines the methods for implementation classes. The methods doesn't have to correspond exactly to the methods in Abstraction.
  - ConcreteImplementor

Bridge Pattern (contd.):

- **Collaborations:**
  - **Abstraction** forwards client requests to its **Implementor** object.

Bridge Pattern (contd.):

- **Consequences:** several benefits and liabilities:
  - decoupling abstraction and implementation eliminates compile-time dependency on the implementation.
  - Improved extensibility: Abstraction and Implementor hierarchies can be extended independently.

Bridge Pattern (contd.):

- **Implementation:**
  - **Encapsulate the implementations in an abstract class**
  - **Contain a reference to its in the base class of abstraction.**