Design Patterns

GoF Patterns

- ✓ Creational Patterns
- Structural Patterns
  - concerned with how classes and objects are composed to form large structure.
- Behavioral Patterns

Structural Patterns

- Adapter
- Decorator

Adapter

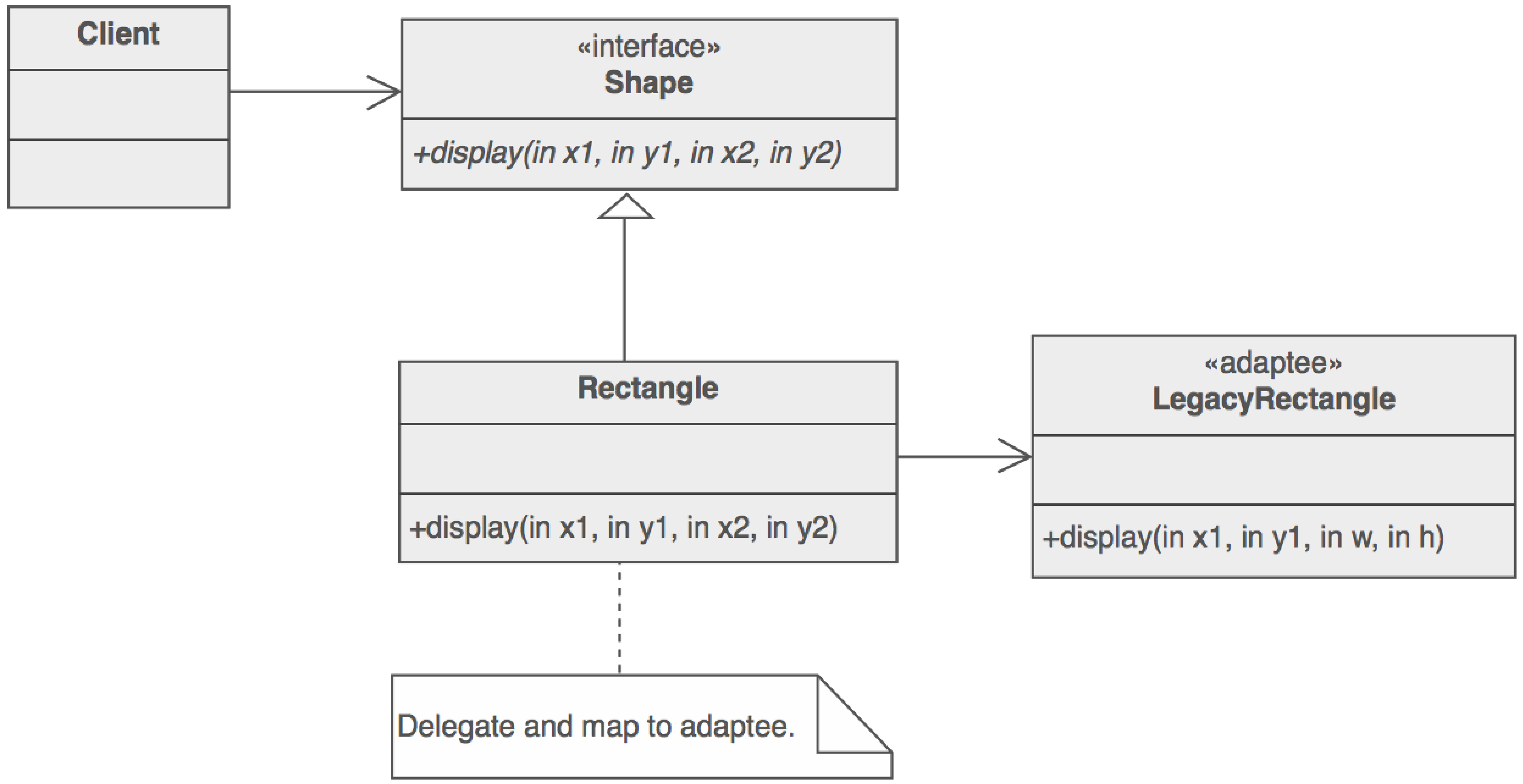
- **Intent**: Convert the interface of a class into another that clients expect

- Adapter pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

Adapter (Cont')

- **Example:** a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

Adapter (Cont’)


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.

Adapter (Cont')

We will define the components of our system. They include:

- **Adaptee (Legacy Rectangle):** Defines an existing interface that needs adapting; it represents the component with which the client wants to interact with the.

- **Target (Shape):** Defines the domain-specific interface that the client uses; it basically represents the interface of the adapter that helps the client interact with the adaptee.

- **Adapter (Recatangle):** Adapts the interface Adaptee to the Target interface; in other words, it implements the Target interface, defined above and connects the adaptee, with the client, using the target interface implementation

Adapter (Cont’)

```
class LegacyRectangle {
   public void draw( int x, int y, int w, int h )
    {
          System.out.println( "rectangle at (" + x + ',' + y + ") with width " + w + " and height " + h );
    }
}
class LegacyLine {
   public void draw( int x1, int y1, int x2, int y2 )
    {
              System.out.println( "line from (" + x1 + ',' + y1 + ") to (" + x2 + ',' + y2 + ')' );
   }
}
```

Adapter (Cont')

```
interface Shape {
   void draw( int x1, int y1, int x2, int y2 );
}

class Rectangle implements Shape {
   private LegacyRectangle adaptee = new LegacyRectangle();
   public void draw( int x1, int y1, int x2, int y2 ) {
      adaptee.draw( Math.min(x1,x2), Math.min(y1,y2),  Math.abs(x2-x1), Math.abs(y2-y1) );
   }
}

class Line implements Shape {
   private LegacyLine adaptee = new LegacyLine();
   public void draw( int x1, int y1, int x2, int y2 ) {
      adaptee.draw( x1, y1, x2, y2 );
   }
}
```

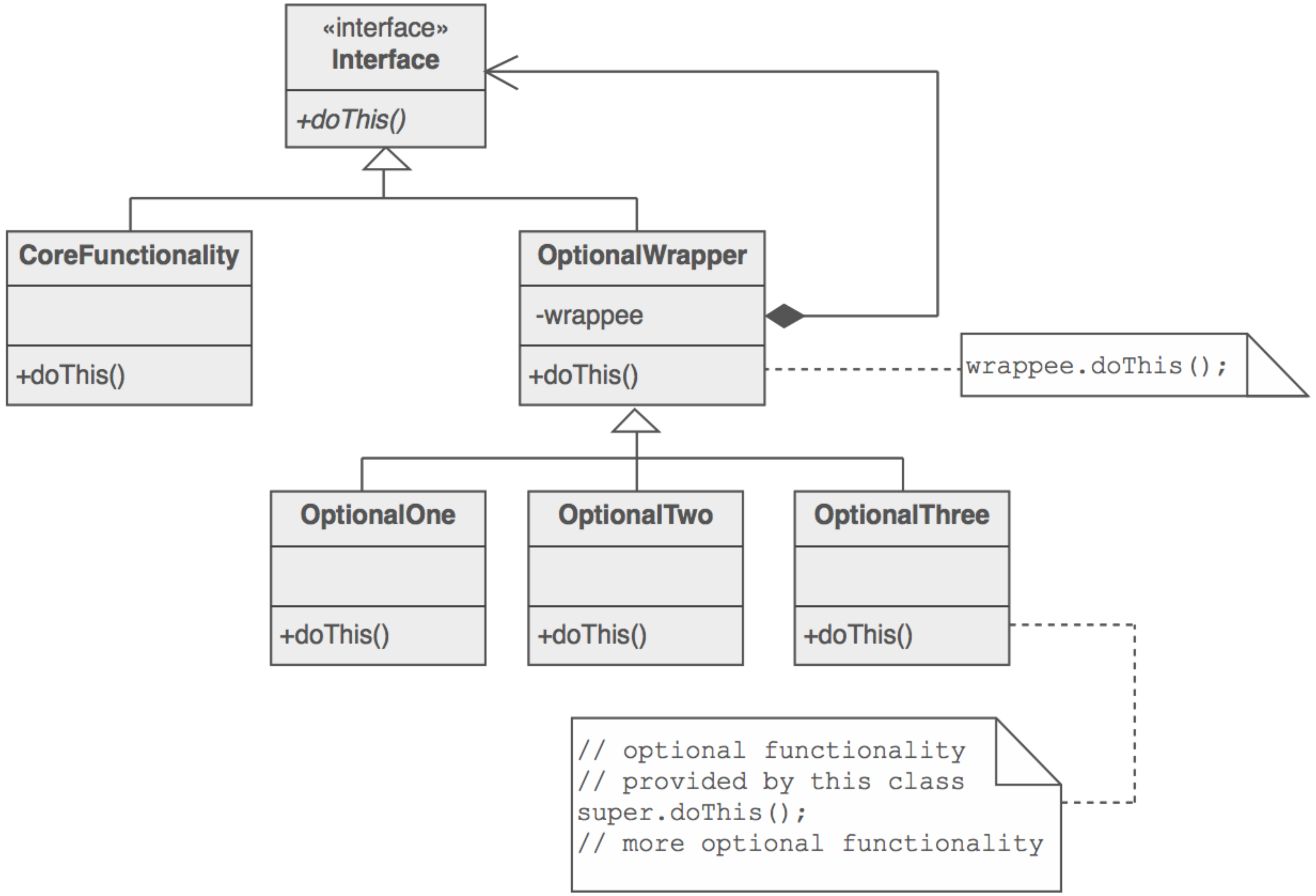
Adapter (Cont')

```
public class AdapterDemo {
   public static void main( String[] args ) {
      Shape[] shapes = { new Line(), new Rectangle() };
      // A begin and end point from a graphical editor
      int x1 = 10, y1 = 20;
      int x2 = 30, y2 = 60;
      for (int i=0; i < shapes.length; ++i)
         shapes[i].draw( x1, y1, x2, y2 );
   }
}
```

Decorator

- **Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator (Cont')



```
// optional functionality
// provided by this class
super.doThis();
// more optional functionality
```

Decorator (Cont’)

- **Example Consider a case of a pizza shop. In the pizza shop they will sell few pizza varieties and they will also provide toppings in the menu. Now imagine a situation wherein if the pizza shop has to provide prices for each combination of pizza and topping. Even if there are four basic pizzas and 8 different toppings, the application would go crazy maintaining all these concrete combination of pizzas and toppings.**

Decorator (Cont’)

```
public abstract class BasePizza
{
    protected double myPrice;

    public virtual double GetPrice()
    {
        return this.myPrice;
    }
}
public abstract class ToppingsDecorator : BasePizza
{
    protected BasePizza pizza;
    public ToppingsDecorator(BasePizza pizzaToDecorate)
    {
        this.pizza = pizzaToDecorate;
    }
    public override double GetPrice()
    {
        return (this.pizza.GetPrice() + this.myPrice);
    }
}
```

Decorator (Cont')

```
public class Margherita : BasePizza
{
    public Margherita()
    {
        this.myPrice = 6.99;
    }
}

public class Gourmet : BasePizza
{
    public Gourmet()
    {
        this.myPrice = 7.49;
    }
}
```

Decorator (Cont')

```
public class ExtraCheeseTopping : ToppingsDecorator
{
    public ExtraCheeseTopping(BasePizza pizzaToDecorate)
        : base(pizzaToDecorate)
    {
        this.myPrice = 0.99;
    }
}
public class MushroomTopping : ToppingsDecorator
{
    public MushroomTopping(BasePizza pizzaToDecorate)
        : base(pizzaToDecorate)
    {
        this.myPrice = 1.49;
    }
}
```

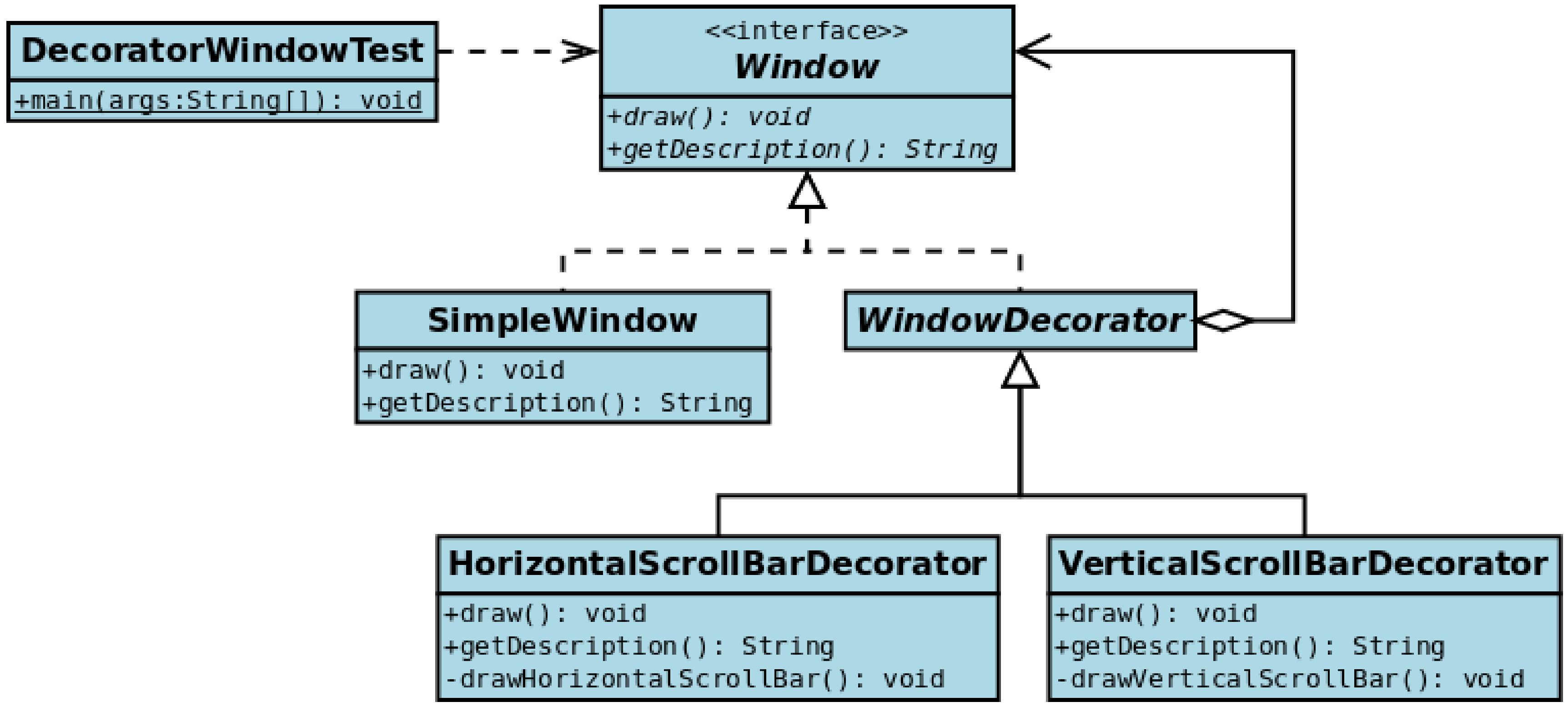
Decorator (Cont’)

```
class Program
{static void Main()
   {
       //Client-code
       Margherita pizza = new Margherita();
       Console.WriteLine("Plain Margherita: " + pizza.GetPrice().ToString());

       ExtraCheeseTopping moreCheese = new ExtraCheeseTopping(pizza);
       ExtraCheeseTopping someMoreCheese = new ExtraCheeseTopping(moreCheese);
       Console.WriteLine("Plain Margherita with double extra cheese: " +
                                    someMoreCheese.GetPrice().ToString());

       MushroomTopping moreMushroom = new MushroomTopping(someMoreCheese);
       Console.WriteLine("Plain Margherita with double extra cheese with mushroom: " +
                                    moreMushroom.GetPrice().ToString());


   }
```

Decorator (Cont’)

- **Example:** Consider a window in a windowing system. To allow scrolling of the window's contents, one may wish to add horizontal or vertical scrollbars to it, as appropriate.
  Assume windows are represented by instances of the Window class, and assume this class has no functionality for adding scrollbars. One could create a
  subclass **ScrollingWindow** that provides them, or create a **ScrollingWindowDecorator** that adds this functionality to existing Window objects. At this point, either solution would be fine.

Decorator (Cont')


DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void

Decorator (Cont’)

```
// The Window interface class
public interface Window {
    public void draw(); // Draws the Window
    public String getDescription(); // Returns a description of the Window
}

// Extension of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // Draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

Decorator (Cont’)

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window windowToBeDecorated; // the Window being decorated

    public WindowDecorator (Window windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }

    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }

    public String getDescription() {
        return windowToBeDecorated.getDescription(); //Delegation
    }
}
```

Decorator (Cont’)

```
// The first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override    public void draw() {
        super.draw();
        drawVerticalScrollBar();
     }

    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }

    @Override    public String getDescription() {
        return super.getDescription() + ", including vertical scrollbars";
     }
}
```

GoF Patterns

- ✓ Creational Patterns
- ✓ Structural Patterns
- ● Behavioral Patterns

Decorator (Cont’)

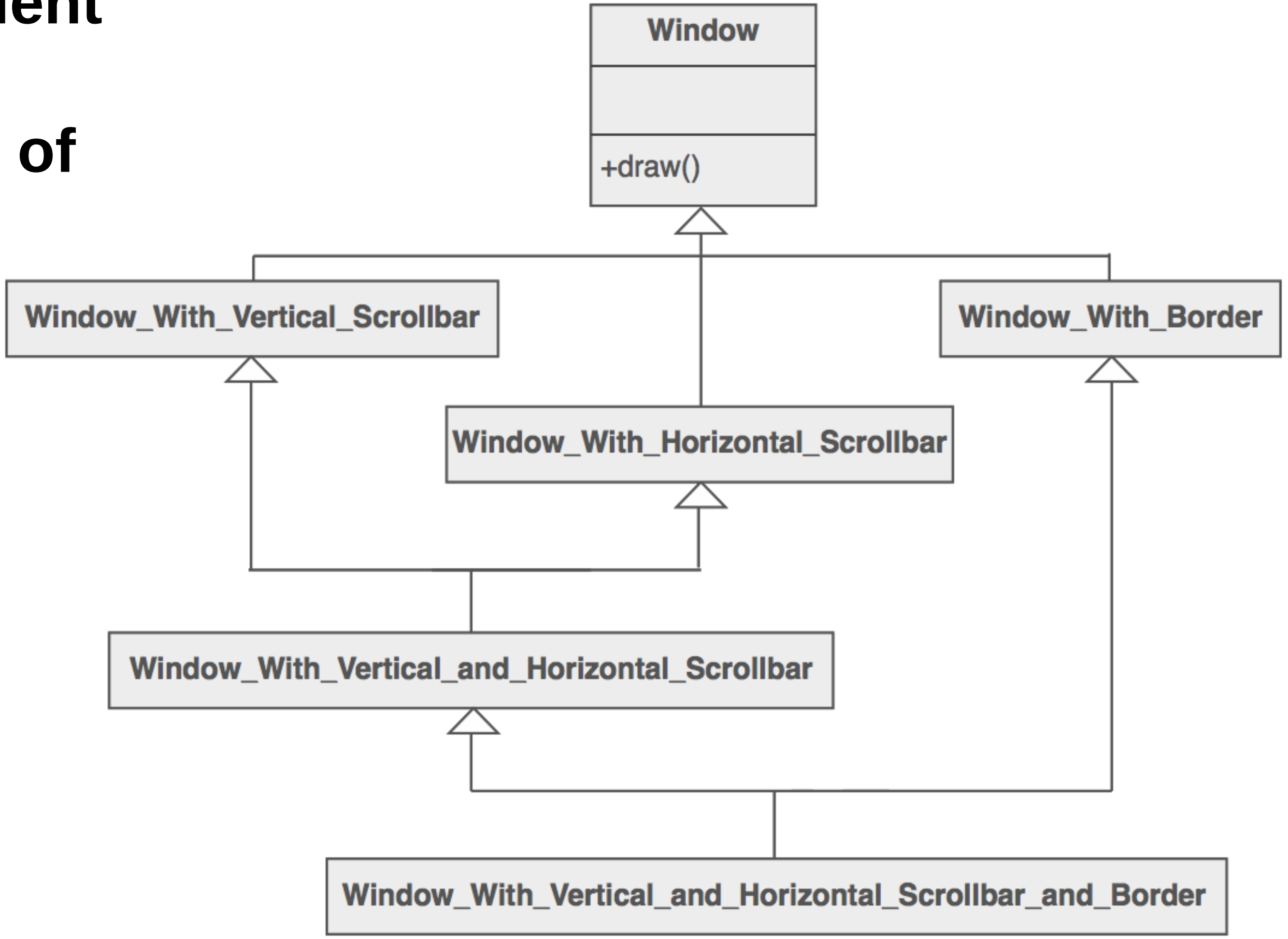
```
// The second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
     }

    private void drawHorizontalScrollBar() {
        // Draw the horizontal scrollbar
    }

    @Override    public String getDescription() {
        return super.getDescription() + ", including horizontal scrollbars";
    }
}
```

Decorator (Cont')

```
public class DecoratedWindowTest
{
    public static void main(String[] args) {
        // Create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator (new SimpleWindow()));

        // Print the Window's description
        System.out.println(decoratedWindow.getDescription());
        }
}
```

Decorator (Cont')

Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Requirement

- Assume you have a function called sumMatrix which takes two (2x2) square matrices (each is a multidimensional array) and add each two corresponding elements together so that the resulting matrix be m1.

  Void sumMatrix (int** m1, int** m2)

  We want to **use** this legacy function with two modifications:

  - Input parameters to the function be two rolled matrices each a single dimensional array.

  e.g. Void sumMatrix(int* m1, int* m2)

  - It's required to print the determinate of resulting matrix after addition.

  You are required to implement the legacy function "sumMatrix" and use the Structural design patterns to satisfy the two requirements.

Hint:

The determinate of a 2x2 matrix is $\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$

REQ2: Put this function (i.e. sumMatrices) into MatrixCalculator class and apply singleton design pattern to this class