No.2(Structural Pattern)

Structural Pattern(構造的パターン：その１)

Facade	サブシステム（あるいはクラス群）にアクセスするインタフェース（メソッドの利用方法）の集合を、統一したインタフェースとして一つのクラスにまとめて構成するパターンであり、概念的に高度なレベルのインタフェースを提供することにより、複雑なサブシステムの機能を分かりやすく再定義するのに用いられます。
Flyweight	類似した大量のオブジェクトを効率的に共有する機能を提供するのに用いられるパターンです。
Bridge	二つのクラスの内容が副作用が無く独立して編集できる様に、それらの概念（機能定義）と実装方法とを分離するために用いられるパターンです。
Decorator	同じインタフェースをクラスに動的に保持しながら、追加の機能を継承クラスに導入するパターンです。

Facade

Facade は、複数のクラスから移譲されたメソッド群を一つのクラス内に含み、それらにアクセスするインタフェースを提供するものです。

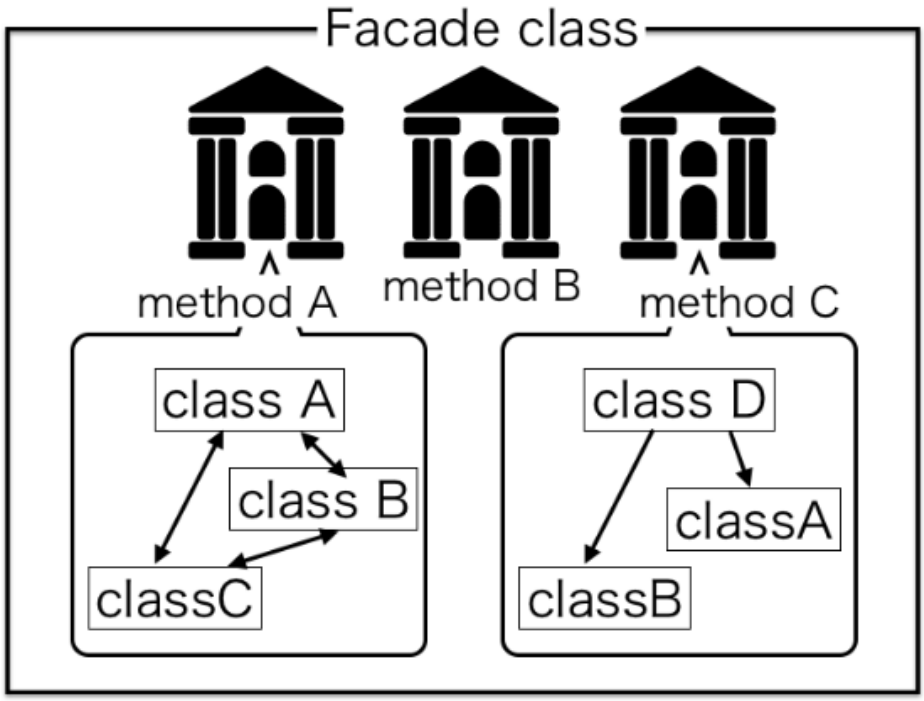
このパターンは、オブジェクトのメソッドを作成する上で最も頻繁に用いられるパターンであり、機能を単純化して分かりやすくする、いわゆる**モジュール化**の基本的な考え方です。

このパターンを用いることにより、巨大で複雑なクラス群を利用する際に、低レベル（詳細レベル）なインタフェースを意識せずに、**必要とする機能だけを備えた高レベルなインタフェースが使用できる**ようになります。

【GoFのデザインパターン】「Facade(ファザード)」ってどんなパターン？

既存のクラスを組み合わせて使う手順を外出しすることができます。Laravelなどでも使われているデザインパターンになります。 【Laravel】「Facade(ファサード)」について 「建物の正面」、「窓口」という意味になります。 ここで使っている「インタ

http://www.code-magagine.com/?p=2751



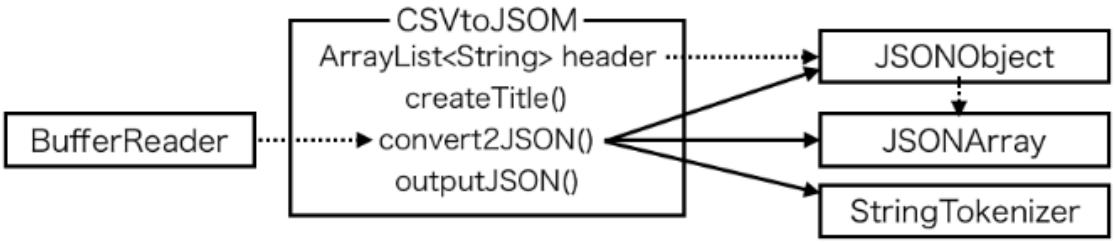
Facade の概念図

Facade は、日本語では建物の正面や、外見、体裁等に訳されますが、上記の概念図に示される様に、異なる複数のクラスへのアクセスやデータのやりとりを、クラスのメソッド（methodA,B,C）として分かりやすく提供する正面（外見、体裁）を提供するものです。その一例として、ソフトウェア演習 II の文書モデルで紹介した、CSV のデータ形式を JSON 形式に変換する CSVtoJSON の内容を再び見てみましょう。

```
public class CSVtoJSON {
    private ArrayList<String> colname;

    public void createTitle (BufferedReader br) throws IOException {
        colname = new ArrayList<String>();
        String line = br.readLine(); // read the top row
        StringTokenizer stt = new StringTokenizer(line, ",");
        while (stt.hasMoreTokens())
            colname.add(stt.nextToken()); // register title of each items
    }

    public JSONObject convert2JSON (BufferedReader br) throws JSONException, IOException {
        JSONObject root = new JSONObject(); // generate JSON object
        JSONArray carList = new JSONArray (); // generate JSON array
        root.put("CarList", carList); // add the carList element
        while ((String line = br.readLine()) != null) { // read one line
            StringTokenizer std = new StringTokenizer(line, ",");
            JSONObject car = new JSONObject(); // create car element
            for (int i = 0; i < colname.size(); i++)
                car.put(colname.get(i), std.nextToken()); // create a Map
            carList.put(car); // add the car element as child
        }
        return root;
    }

    public void outputJSON (String outFileName, JSONObject jobj) throws IOException{
        OutputStream out = new FileOutputStream(outFileName);
        Writer outWriter = new OutputStreamWriter(out, "UTF-8");
```

```
            BufferedWriter writer = new BufferedWriter(outWriter);
            writer.write(jobj.toString());
            writer.close();
        }
    }
```

ここでは、`CSVtoJSON`クラスが Facade パターンで設計されていると見なせ、そのメソッド`createTitle, convert2JSON, outputJSON`は、その内部で JSON の要素に関するクラス（`JSONObject, JSONArray`）や、出力に関するクラス（`OutputStream, Writer, BufferedWriter`）、および文字列処理のクラス（`StringTokenizer, ArrayList<String>`）間の処理やデータのやりとりを意識しないで済む様にカプセル化（高レベルな概念化）しており、簡単なインタフェースを提供していると解釈できます（下図参照）。



Facade パターンでのデータの流れ図

（自分メモ）

いろんなクラスのメソッドの利用法をまとめたクラス的な

これ一つ見ればプログラム全体の使用がなんとなくつかめる

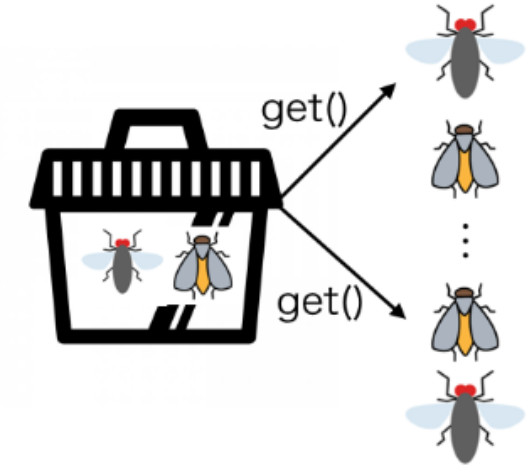
Facadeパターン設計法まとめ

- 他のクラスの生成やそれらのデータ間の処理をメソッドとして記述し、他のクラスに対するアクセスをわかりやすく簡略化（モジュール化）できるクラスを作成する。

Flyweight

Flyweight は、全体の処理の効率化のために、一度生成したオブジェクトを繰り返し再利用するのに用いるパターンです。すなわち、ハエ（Fly）の様に軽量（Flyweight）なオブジェクトの再利用法です。

このパターンは、類似のインスタンスを多数利用するときに、**インスタンスを共有して使いまわします**が、 共有するインスタンスは全く同一である必要はありません。共有できる情報は共有インスタンスに持たせ、共有できない情報はインスタンスを利用する際に渡すことにより、効率の良い再利用を行います。



Flyweight の概念図

ここで、描画可能な幾何形状の概念クラス（`Drawable`）を継承する楕円体のクラス（`Ellipse`）を考えます。ただし、以下のコードの様にその２軸の長さは描画時に引数で与えられるものとし、描画の色情報を指定するフィールド値がコンストラクタで与えられるものとします。

```
public interface Drawable {
   void draw(int width, int height);
}

public class Ellipse implements Drawable {
   private Color color;

   public Ellipse(Color color){
      this.color = color;
   }

   @Override
   public void draw(int width, int height) {
       // draw ellipse
   }
}
```

次に、この Ellipse クラスのオブジェクトを生成する工場のクラス（`ShapeFactory`）を、以下の様に構成します。

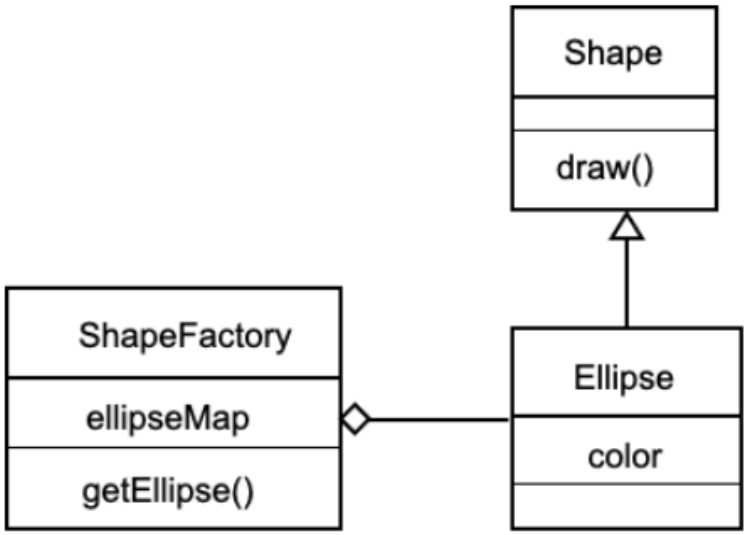
```
public class ShapeFactory {
   private static final HashMap<Color, Drawable> ellipseMap = new HashMap<Color, Drawable>();

   public static Drawable getEllipse(Color color) {
      Ellipse ellipse = (Ellipse) ellipseMap.get(color); // generate object only if the color is new one !
      if(ellipse == null) {
         ellipse = new Ellipse(color);
         ellipseMap.put(color, ellipse);
      }
      return ellipse;
   }
}
```

ここでは、`getEllipse`メソッドによって、指定された色のオブジェクトが生成されますが、同色の Ellipse オブジェクトは内部状態が同じなので、共通のオブジェクトを使い回せる様にしています。すなわち、`HashMap ellipseMap`のフィールドに、指定色で生成されたオブジェクトを登録し、既に登録されたオブジェクトと同色のもの生成が要求された際には、`ellipseMap`からその登録された値を返す様にしています。

以下のコードは、Flyweight のデザインパターンで構成された上記の `ShapeFactory`の使用例です。ここでは、`getColor`は色の値を無作為に生成して返すメソッドとし、サイズの値（`width, height`）も乱数で与えられています。

```
// getColor() randomly returns the value of pre-defined java.awt.Color
 for(int i=0; i < 10000; ++i) {
    Ellipse ellipse = (Ellipse) ShapeFactory.getEllipse(getColor());
    int width = (int) (10.0 * Math.random()) + 1;
    int height = (int) (10.0 * Math.random()) + 1;
    ellipse.draw(width, height);
 }
```

上記のコードでは、楕円体オブジェクトが１万回生成されていますが、**実際に生成されているオブジェクトは `getColor` が生成する色の個数**（例えば、７色に限定されていれば７個）に留められます。



Flyweight を用いた ShapeFactory クラスの構成

Prototypeとの相違点

Prototype は生成型に、Flyweight は構造型に分類されている違いはありますが、どちらも軽量のオブジェクトを使い回すと言う点では類似しています。その相違点は、以下の様に明確化されます。

- **Prototype ではオブジェクトを**クローン化（インスタンスのコピー）して再利用するのに対して、**Flyweight ではオブジェクトを共有化**（インスタンスの参照）して使い回します。
- **Prototype** のオブジェクトは **mutable（変更可能）**なのに対して、**Flyweight** のオブジェクトは副作用を防止するために **immutable（変更不可）**である必要があります。

- Prototype では原型となるオブジェクトは初期化に際に全て生成しておきますが、**Flyweight ではそれが必要となった時点で生成**します。
- Prototype では一つのオブジェクトを一つの型（利用法）で生成する際に用いられ、**Flyweight では一つのオブジェクトを複数の型（利用法）で生成する**際に用いられます。
- Prototype はオブジェクトを（コピーではなく）新たに生成する際の計算／メモリコストを削減するための原型の利用方法であり、**Flyweight は**メモリ消費量を減らすために、**既に生成したオブジェクトをできるだけ使い回す（再利用する）方法**です。

Flyweightパターン設計法のまとめ

- 何度も利用するオブジェクトを保持するための（List や Map のフレームワークを用いた）フィールドを有するクラスを作成し、それらのオブジェクトはこのクラスを介して生成する。
- オブジェクトの生成が要求された際には、登録されている再利用可能なオブジェクトを探索し、存在する場合にはそのオブジェクトを戻り値で返し、存在しない場合には新たに生成して上記のフィールドに登録した後に返す仕組みを設ける。

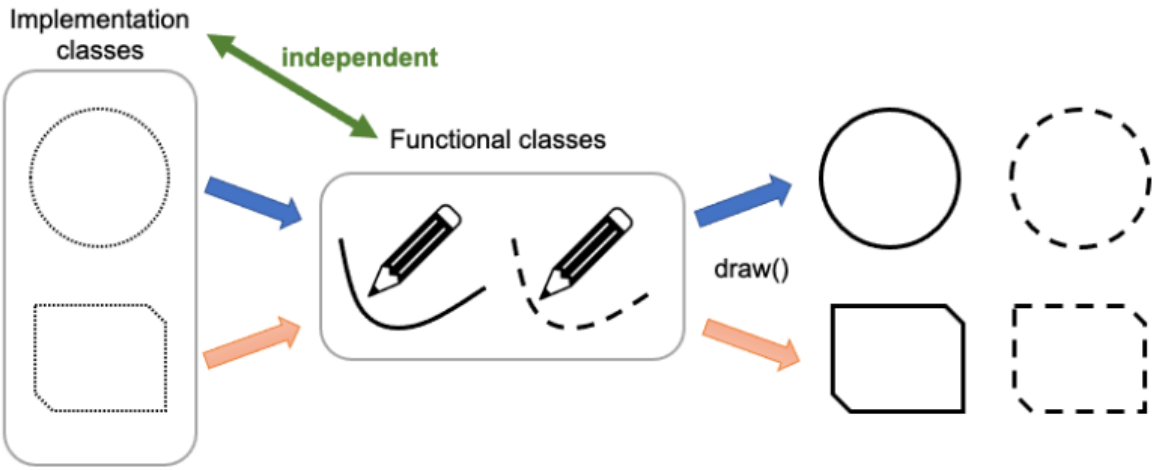
Bridge

Bridge とは、共通機能を実現する「実装」のクラス階層と、それを使った拡張機能を実現する「機能」のクラス階層を明示的に分離して、それらを橋渡しするデザインパターンです。「橋渡し（Bridge）」のクラスを導入することにより、クラスを複数の独立な方向に拡張させることを可能にします。

このパターンを用いることにより、**「機能」の追加に対して既存の全ての「実装」が対応でき、「実装」の追加に対しても既存の全ての「機能」が対応できる**という利点が得られます。このように「実装」と「機能」のクラス階層を分離できると、機能の組み合わせによるクラス数の不用意な増加を抑えることができます。

これまでは、形状のクラスには描画用のメソッド `draw` が有りましたが、ここでは**形状の定義（型や縦横のサイズ）と描画機能を分離して扱える**様に、以下の様な形状一般の描画を扱えるクラス `DrawAPI` を定義します。すなわち、形状のクラスは「実装のクラス階層」であり、描画のクラスは「機能のクラス階層」に対応し、これらを独立に改変できる様な仕組みにします。

この例では、クラス階層の分離を行わなくても必要とされるクラスの数は同じ（＝４種類）ですが、形状クラスが N(>2) 種類、描画クラスが M(>2) 種類に増加した場合を考えると、Bridge による実装と機能の分離がなされていれば必要となるクラスは N+M ですが、その様な分離がなされていない場合には全ての組み合わせ数である N*M のクラスを作成する必要が生じます。ゆえに、形状や描画の種類の数（N と M の値）が増大するに伴い、その効率化の効果は絶大なものとなります。



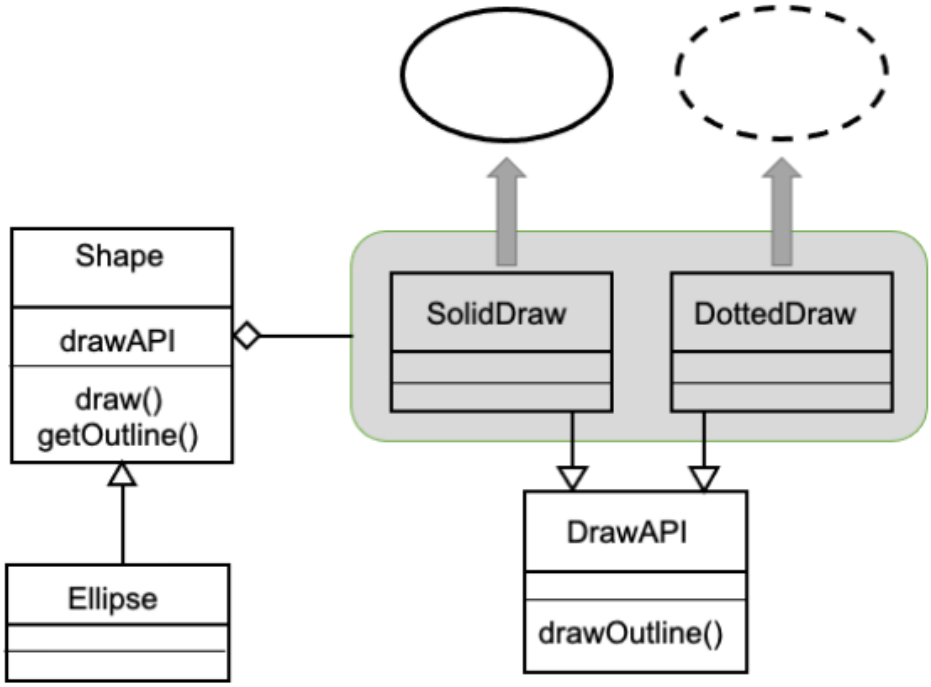
Bridge の概念図

```
public interface DrawAPI {
   // draw shape with given style
   public void drawOutline(Shape s, int w, int h);
}

public class SolidDraw implements DrawAPI {
   @Override
   public void drawOutline(Shape shp, int w, int h) {
      ArrayList<Vector2D> outlineVectors = shp.getOutline();
       // draw solid-type polylines using outlineVectors
   }
}

public class DottedDraw implements DrawAPI {
   @Override
   public void drawOutline(Shape shp, int w, int h) {
      ArrayList<Vector2D> outlineVectors = shp.getOutline();
      // draw dotted-type polylines using outlineVectors
   }
}
```

ここでは、描画メソッド `draw(Shape shp, int w, int h)` は、形状のオブジェクトを第一引数に持つことに注意してください。また、形状の概念クラス（`Shape`）は、輪郭形状の２次元ベクトルのリストを取得するためのメソッド `getOutline` を実装すべき機能として宣言しており、`draw` メソッドはこの `getOutline` から得られる情報を元に描画を実行します。

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

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw(int w, int h);
   public abstract ArrayList<Vector2D> getOutline(int w, int h);
}

public class EllipseOutline extends Shape {

   public EllipseOutline(DrawAPI drawAPI) {
      super(drawAPI);
   }
```

```
    public void draw(int w, int h) {
       drawAPI.drawOutline(this, w, h);
    }
    public ArrayList<Vector2D> getOutline(int w, int h) {
       // returns the ArrayList of 2D vectors of the outline
    }
}
```



Bridge パターンを用いた Ellipse クラスの構成

以上の様な構成により、**形状のクラス（`Shape` インタフェースを継承したクラス）と、描画のクラス（`DrawAPI` インタフェースを継承したクラス）の独立性が保たれ**ており、形状の種類（Ellipse, Rectangle, Triangle 等）ごとに個別の描画のクラス（Rectangle+SolidDraw, Triangle+DottedDraw 等）を作成する際の、重複する無駄なコードを避けることができます。

これらのクラスを用いた実行例は、以下の様になります。

```
EllipseOutline solidEllipse = new EllipseOutline(new SolidDraw());
EllipseOutline dottedEllipse = new EllipseOutline(new DottedDraw());
solidEllipse.draw(20, 10);
dottedEllipse.draw(20, 10);
```

Bridgeパターン設計法のまとめ

- 特定の機能を表すインタフェースクラスを作成し、各々の具体的な機能はそのクラスを継承（implements）して作成する。
- 機能の適用対象となる実装オブジェクトには、上記の各機能を有するオブジェクトをフィールドに保持する。

- 実装オブジェクトの各機能を呼び出すメソッドは、フィールドに保持した対応する機能オブジェクトのメソッドを介して実行される。

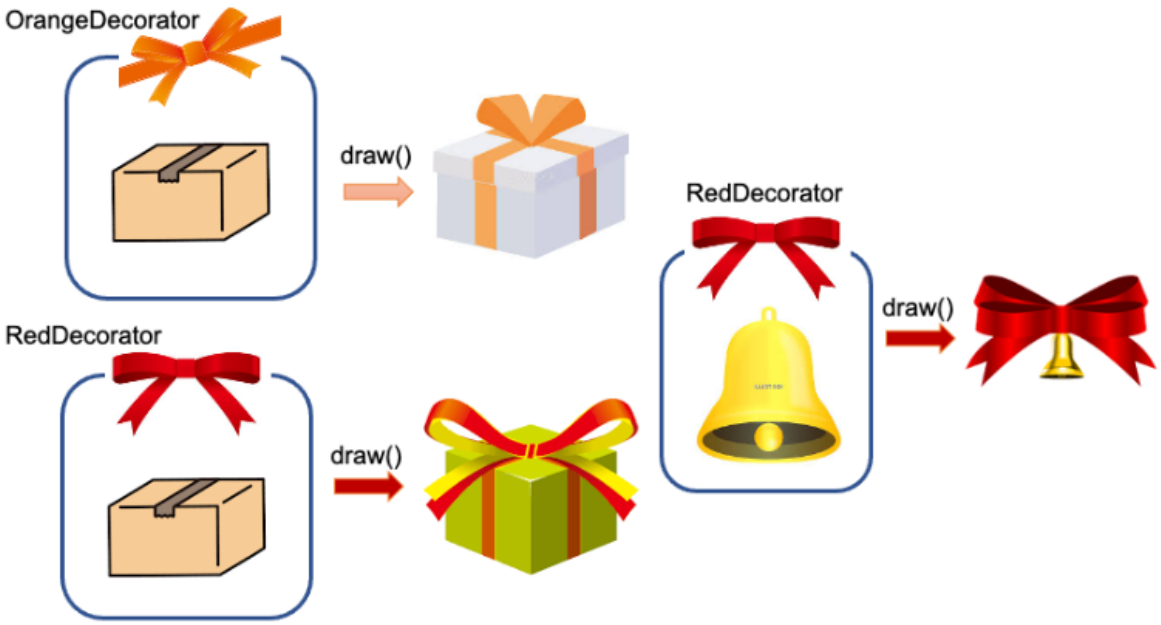
Decorator

Decorator は、既存のオブジェクトに対してその構造を変えることなく新たな機能を追加するために、**クラスをラップして装飾（decorate）する**パターンです。

これは、**継承を用いずに委譲（他のクラスに実行を委任する）を使って機能を拡張**します。その拡張方法が機能を装飾していくように見えるので Decorator（装飾者）と呼ばれます。

【GoFのデザインパターン】「Decorator(デコレータ)」ってどんなパターン？
【GoFのデザインパターン】「Decorator(デコレータ)」ってどんなパターン？ | プログラミングマガジン

http://www.code-magagine.com/?p=2725



Decorator パターンの概念図

ここでは、形状クラス（EllipseDI, RectangleDI）に対して、赤色で描画するという装飾を加えるクラス（RedDecorator）を構成します。

まず、インタフェースDrawItemを以下の様に宣言します。

```
public interface DrawItem {
    void draw();
}
```

次に、このインタフェースを継承する形状クラスを作成します。

```
public class EllipseDI implements DrawItem {
    private int width, height;
    public EllipseDI(int w, int h) {
        width = w;
        height = h;
    }

    @Override
    public void draw() {
        // Draw the shape of ellipse based on DrawItem interface
    }
}
```

次に、`DrawItem`インタフェースを継承し、形状の描画方法を拡張するという概念のクラス`ShapeDecorator`を宣言します。

```
public abstract class ShapeDecorator implements DrawItem {
   protected DrawItem decoratedShape;

   public ShapeDecorator(DrawItem decorShape) {
      this.decoratedShape = decorShape;
   }
   public abstract void draw(); // should be implemented
}
```

ここで、`ShapeDecorator`はそのフィールドに具体的な形状オブジェクトである`decoratedShape`を保持し、コンストラクタも`DrawItem`を継承したクラスのオブジェクトを引数としている点に注意して下さい。

次に、この概念クラス（`ShapeDecorator`）を継承し、`RedDecorator`クラスを実装します。

```
public class RedDecorator extends ShapeDecorator {
   public RedDecorator(DrawItem decoratedShape) {
      super(decoratedShape);
   }
   @Override
   public void draw() {
      setRedBorder(decoratedShape);
      decoratedShape.draw();
   }
   private void setRedBorder(DrawItem decoratedShape){
      // set red color to the border
   }
}
```

ここでは、描画関数`draw`は、輪郭線を赤色に設定するための前処理である`setRedBorder`を実行したのちに、フィールドとして保持している形状オブジェクトの描画関数`decoratedShape.draw(w, h)`を呼び出しています。ここでは、（赤色で描画するという）機能の拡張が、新たなクラス（`RedDecorator`）のメソッド（`setRedBorder`）に移譲されています。また、`RedDecorator`は、`DrawItem`インタフェースを継承した全形状を扱える様に設計されています。

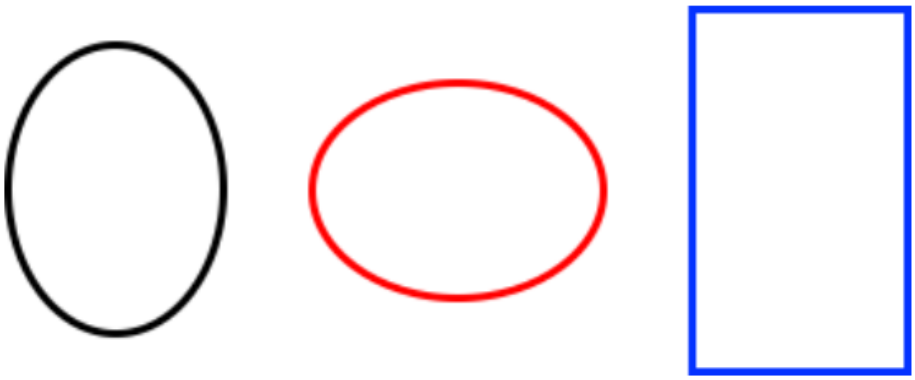
また、輪郭線を青色で描画するクラス（`BlueDecorator`）も同様に構築できることは容易に理解できるでしょう。

これらのクラスは、以下の様に実行されます。

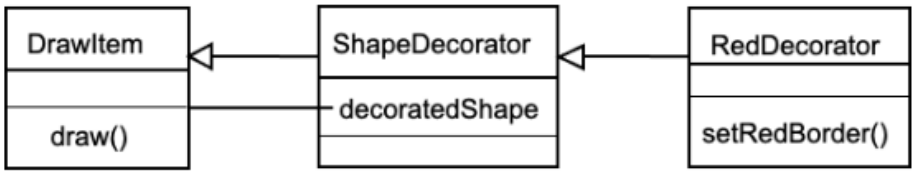
```
DrawItem ellipse = new EllipseDI(15, 20);
ellipse.draw();

DrawItem redEllipse = new RedDecorator(new EllipseDI(20, 15));
redEllipse.draw();

DrawItem blueRectangle = new BlueDecorator(new RectangleDI(15, 25));
blueRectangle.draw();
```

ここで、最初に生成されるellipseは、によってラッピング（修飾）されていないので、デフォルトの黒色で描画されます。

また、ShpaeDecoratorを継承したクラスは楕円体だけでなく、他の形状クラスのオブジェクト（RectangleDI等）に対しても適用可能です。



上記コードの実行結果



Decorator パターンを用いた RedDecorator クラスの構成

Decoratorパターン設計法のまとめ

- オブジェクトを操作するメソッドを宣言したインタフェースクラスを作成する。
- 上記のインタフェースを継承した（概念）クラスを作成し、必要であれば操作対象のオブジェクトをフィールドに保持する。
- 上記のクラスに対して（または、概念クラスを継承して）、新たな装飾（機能）を施すメソッドを作成する。