局面指向プログラミングの批判的考察

玉 井 哲 雄†

発展的なプログラミングに大きなヒントを与える局面指向プログラミングの事例を分析し，その記述に AOP という機構を用いなくても，従来のオブジェクト指向の枠組の中で，モジュール構造を美しく表現し，かつ効率も悪化しないような設計が可能であることを示す．そのねらいは，局面にはどのようなものが存在し，どのような書き方が有効であるか，また統一的な局面指向プログラミングの手法はどのような形であるべきか，といった議論のきっかけとすることにある．

A Critical View on Aspect Oriented Programming

Tetsuo Tamai†

1. はじめに

Xerox PARC の研究者たちが提唱している局面指向プログラミング（Aspect Oriented Programming，以下で AOP と略記）が一部で話題になっている．まだきちんとした論文は出版されていないようだが，WWW ではかなり詳しいレポートが公表されている[1)]．AOP の主張は，プログラムの設計に際し，問題の構造に応じた自然できれいなモジュール化をしただけでは，別の要求，たとえば性能の要求に答えられない場合に，もとのモジュール構造を壊さないでその別の局面 (aspect) に対処するプログラムを独立に記述し，それともとのプログラムとをを編み合わせる手段を提供するというものである．ただ，局面ごとのプログラミングの記述方法や編み合わせの手法として統一的なものがあるわけではなく，いくつかの例題を挙げて，それぞれについて局面指向プログラミングはこうすればよいという個別の方法を提示するという形態を取っているようである．このような局面指向の考え方は，発展的なプログラムの開発に有効であろう．われわれも別の構想に基づいて，種々の異なる状況を環境として独立に記述し，計算主体たる個々のオブジェクトはそれらの環境に所属することによりその場の協調計算に参加できるというような発展型の計算モデルを提案している[2)]．

本稿では，AOP で取り上げられている事例を分析し，その記述に AOP という機構を用いなくても，従来のオブジェクト指向の枠組の中で，モジュール構造を美しく表現し，かつ効率も悪化しないような設計が可能であることを示す．このように批判的に AOP を見ることは，AOP の考え方自体を否定するものではなく，局面にはどのようなものが存在し，どのような書き方が有効であるか，また統一的な局面指向プログラミングの手法はどのような形であるべきか，といった議論のきっかけとすることがねらいである．

† 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, University of Tokyo

2. AOP の事例

AOP の紹介で最初に取り上げられている例は，画像処理プログラムである．画像処理において，画面全体の画素ごとにビットとしての和や積をとったり否定をとったりする操作を定義し，それらの組合せでより高度な操作を実現するという方法をとると，個々の操作が分かりやすくなるだけでなく，それらの組み合わせによる高度な操作の意味も理解しやすい．しかし，それにより，基本演算が呼び出されるたびに，画像全体にわたる画素情報を中間的に作り出す必要があり，データ領域に無駄が生じるだけでなく，計算速度にも大きな悪影響が及ぶ．

基本操作の概要を，以下に示す．原論文では LISP 風の表記で記述されているが，ここでは Java 風に書く．まず，画像を表現したデータを Image というクラスで表す．その仕様は次のようである．

```
class Image
  Image(int w, int h) :
    幅 (画素数)w，高さ (画素数)h の画像データを
```

生成する．初期値はすべて白（ビット 0）．

`int width()` : 幅を返す.

`int height()` : 高さを返す.

`Pixel getPixel(int i, int j)` :
座標 (i,j) の画素の値を返す.

`void setPixel(int i, int j, Pixel p)` :
座標 (i,j) の画素の値を p とする.

ここで，画素 (クラス Pixel) は白黒の 2 値データとし，ビットと同一視する．画素に対し単項演算 not，2 項演算 and と or が通常のビット演算として定義されている．Image は要するに，ビットの配列と思えばよい．画面全体に対する or 演算は，Image のスタティック・メソッドとして次のように定義される．

```
public static Image or(Image a, Image b) {
  Image x = new Image(a.width(),a.height());
  for (int i=0; i<a.height(); i++)
    for (int j=0; j<a.width(); j++) {
      x.setPixel(i,j,
                 Pixel.or(a.getPixel(i,j),
                          b.getPixel(i,j)));
  return x;
}
```

まったく同様に，画像の 2 項演算 and，単項演算 not が定義される．さらに，画像全体を 1 行上に上げる演算 up を次のように定める．

```
public static Image up(Image a) {
  Image x = new Image(a.width(), a.height());
  for (int i=0; i<a.height()-1; i++)
    for (int j=0; j<a.width(); j++)
      x.setPixel(i,j,a.getPixel(i+1,j);
  return x;
}
```

同様に，全体を 1 行下に下げる演算 down も定義される．

さて，これらの基本演算 (原論文ではこれらをコンポーネントと呼んでいる) を組み合わせて，次のような複合的な演算を定義することを考える．

```
public static Image remove(Image a,Image b) {
  /* a から b の像を取り除く．*/
  return Image.and(a, Image.not(b));}
public static Image topEdge(Image a) {
  /* a の上縁を求める．*/
  return Image.remove(a,
                      Image.not(Image.down(a)));}
public static Image bottomEdge(Image a) {
  /* a の下縁を求める．*/
  return Image.remove(a,
                      Image.not(Image.up(a)));}
public static Image horizontalEdge(Image a) {
  /* a の上下の縁を求める．*/
  return Image.or(Image.topEdge(a),
                  Image.bottomEdge(a));}
```

これらの定義は分かりやすいが，確かに効率は悪い．ループ計算が何重にも起こり，その度に，中間的な配列構造が作られる．しかし，効率を良くするようなコードを埋め込むと，モジュール構造が見にくくなる．そこで AOL では，ループを統合するという変換操作を局面プログラムとして別に記述する．これを元のプログラムに編み込むと，効率的なプログラムが生成されるという算段である．しかし，その局面プログラムは決して分かりやすくないばかりか，and, or, not というタイプのループの合成には成功しているが，ループの形が異なる up, down の合成には成功していない．

3. 抽象クラスを利用した解法

もっとうまい方法はないだろうか．上に出てきたメソッドは，いずれも Image というデータを返すという形をしている．一般に，ある型のオブジェクトを返す関数（メソッド）は，通常その型のオブジェクトを作成して，それへのポインターを返すことになる．しかし，外から見たオブジェクトはそのサービス（メソッド）の集合だから,「データ」としてのオブジェクトを返す必要はなく，適用可能なメソッドを公開してやればよい．たとえば，論文の中の or という関数を次のように書いてみる．

```
public Image or(Image a, Image b) {
  public Pixel getPixel(int i, int j) {
    or(a.getPixel(i,j), b.getPixel(i,j));
  }
}
```

それでこれらを使った horizontalEdge は，単に次のように書けば良い．

```
public Image horizontalEdge(Image a) {
  public Pixel getPixel(int i, int j) {
    or(topEdge(a), bottomEdge(a)).getPixel(i,j);
  }
}
```

残念ながら，Java のような通常のオブジェクト指向言語では，このような書き方はできない．しかし，抽象クラスと継承関係を利用すれば，同じような効果が実現できる．すなわち，以下に示すように Image という名の抽象クラスを作り，not, or, and, up, down

その他の演算を，すべてその子クラスとして，そのgetPixel というメソッドを変えるという形で実現する．なお，記述を簡単にするため（効率への配慮もあるが），これまでメソッドとして書いてきたwidth() とheight() を public な変数に変えている．

```
abstract class Image {
  public int width, height;
  abstract Pixel getPixel(int i,int j);
}

class NotImage extends Image {
  Image im;
  NotImage(Image x) {
    im = x;
    width = im.width;
    height = im.height;
  }
  public Pixel getPixel(int i, int j) {
    return Pixel.not(im.getPixel(i,j));
  }
}

class AndImage extends Image {
  Image im1, im2;
  AndImage(Image x, Image y) {
    im1 = x; im2 = y;
    width = im1.width;
    height = im1.height;
  }
  public Pixel getPixel(int i, int j) {
    return Pixel.and(im1.getPixel(i,j),
                     im2.getPixel(i,j));
  }
}
```

以下 or，up，down の定義は同様．これらを用いて，

```
class TopEdgeImage extends Image {
  Image im;
  TopEdgeImage(Image x) {
    im = new RemoveImage(x, new DownImage(x));
    width = im.width; height = im.height;
  }
  public Pixel getPixel(int i, int j) {
    return im.getPixel(i,j);
  }
}
```

BottomEdgeImage も同様．

```
class HorizontalEdgeImage extends Image {
  Image im;
  HorizontalEdgeImage(Image x) {
    im = new OrImage(new TopEdgeImage(x),
                     new BottomEdgeImage(x));
    width = im.width; height = im.height;
  }
  public Pixel getPixel(int i, int j) {
    return im.getPixel(i,j);
  }
}
```

この方法は一見したところ不自然に見えるかもしれないが，画素配列というデータを保存する代わりに，そのデータ要素を取り出すための操作を保存するという常套的な方法を実現したものと考えれば，それほど不自然でもない．もとの論文ではうまく書けずにごまかしてある，up や down もまったく同じように書ける．コンポーネントの記述自身，すっきりしているし，何より局面プログラミング部分がまったくいらなくなる．

4. 議　　論

ここで取った方法は，以下のようなすでによく知られている手法と関連がある．

(1) すでに述べたように，データを保存する代わりにデータを生成する操作を保存するという手法
(2) データが必要になった時に計算するという遅延評価
(3) not, and, or などの演算は 1 種のフィルタと見なせるが，それらをパイプによって結合し無駄な中間データを省くフィルタ・パイプ手法

その意味で，過去のよい設計手法の応用といえる．近年はやりの言葉で言えば，設計パターンといってもよい．

しかし，問題もある．画像処理の場合は，1 画素に対する操作がメソッド呼び出しに比べて軽いものだから，このようにメソッド呼び出しが連鎖することによるのオーバーヘッドは，効率上大きな問題となりうる．これに対し，いくつかの解決策が考えられる．

(1) 画像上のごく一部のデータを，少ない頻度で参照する場合は，このプログラムのままでよい．
(2) メソッドの呼び出しをインライン展開する．た

だし，この例ではメソッド呼び出しが深く入れ子になっているので，コンパイラによる機械的な展開は難しいかもしれない．それでも，原論文にあるようなアドホックな局面プログラムよりははるかに一般的な方法といえる．

(3) 多くのアクセスが必要な画像は，上の形でそれを表現したオブジェクトからたとえば画素の配列で実現したオブジェクトに簡単に変換できるので，そのような変換を行う．その変換の際にも，複合化したループが 1 つに統合されるという効果がある．

この 3 番目の方法を具体的に実現するには，たとえば次のようなプログラムを書けばよい．

```
class ConcreteImage extends Image {
  Pixel[][] pixels;
  ConcreteImage(int w, int h, Pixel[][] p) {
  width = w; height = h; pixels = p;
  }
  ConcreteImage(Image im) {
    width = im.width; height = im.height;
    pixels = new Pixel[width][height];
    for (int i = 0; i < width; i++)
      for (int j = 0; j < height; j++)
        pixels[i][j] = (im.getPixel(i,j));
  }
}
```

ここで 2 つ目のコンストラクタが，抽象クラスとしての画像をもらって，配列で具体化された画像データを作り出す働きをする．

謝　　辞

この問題についてきわめて有意義な議論をしていただいた増原英彦氏に感謝します．

参 考 文 献

1) http://www.parc.xerox.com/aop
2) 鵜林尚靖，玉井哲雄：オブジェクト間協調に基づく環境適応型計算モデル，オブジェクト指向最前線―情報処理学会 OO'99 シンポジウム，朝倉書店，pp.141-149 (1998)．