画像問題のアスペクト

玉井哲雄

1　はじめに

2006 年 5 月に東京で開かれた SEA フォーラム「アスペクト指向技術は実用に使えるか」で，私とともに企画を担当された小林修氏（SRA）が，パネル討論の際にアスペクト指向プログラミング（AOP) の源流と目される Gregor Kiczales の論文 [1] を取り上げて，議論の材料とされた．そこで私は，2001 年 11 月に当時増原英彦氏（この日の SEA フォーラムの第一講演者）が文部省の在外研究でカナダのブリティッシュ・コロンビア大学 (UBC) の Kiczales 氏のところに滞在していたのを訪れ，Kiczales グループに向けてこの論文の内容に対して批判的な講演を行ったことを思い出した．フォーラムの場でその 2001 年に使ったスライドが発掘できたので，会場でそれを見せてどんな話をしたのかごく手短かに紹介した．

例によって岸田編集長からフォーラムの結果をまとめよという指令が来たので，これを機会にこのときの話を文章化しようと思い立った．実際，この話は UBC や仲間内の研究会や学生とのゼミで紹介したが，論文や読み物としてはどこにも発表していない．時宜を失したともいえるが，この AOP フォーラムの報告に紛れ込ませて，SEA 会員の皆様のご批判の目に供するのも一興かと思う．

とここまで書いて，この話をどこにも出していないというのは誤りであることに気がついた．1999 年 1 月に高知で開かれたウィンターワークショップ・イン・高知（情報処理学会ソフトウェア工学研究会主催) の参加表明論文（いわゆるポジションペーパー) に「局面指向プログラミングの批判的考察」と題して，4 ページほどの小論を書いているのである [2]．しかし，書いた当人も忘れているくらいだから，ほとんど読まれていないと思う．

以下，部分的にその小論を再利用しつつ，現在の目で見た見直しを含めて，この話を書いてみよう．

2　どんな問題か

まず言っておかなければいけないのは，この時点では，AspectJ はまだ存在していなかったということである．1995 年頃から Kiczales などの Xerox PARC の研究者たちが，アスペクト指向プログラミングという概念を提唱していることはある程度知られていた．しかし，きちんとした論文はなかなか出版されず，技術レポートが Web に公開されているという状態がしばらく続いていたと思う．1999 年のワークショップで槍玉にあげた論文も，そのようにして Web から取ったものだった．実は，その内容をもとにした論文は，すでに 1997 年の ECOOP で発表されていたのだが [1]，当時はうかつにもそれに気づいていなかった．

いずれにせよ AspectJ 以前だから，アスペクトの記述方法やそれをプログラムに織り込む手段は，問題ごとにアドホックに考えるという立場である．したがって，AOP の手法自身の説明も，とりあげた例題の性質に多分に，というよりきわめて強く依存している．

その例題とは，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 and 演算は，Image のクラス・メソッドとして次のように定義される．

```
public static Image and(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.and(a.getPixel(i,j),b.getPixel(i,j)));
  return x;
}
```

ここで and 演算をクラス・メソッドとして定義しインスタンス・メソッドにしなかったのは，その方が原論文の Lisp 表現に近いからである．まったく同様に，画像の 2 項演算 or，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));
}
```

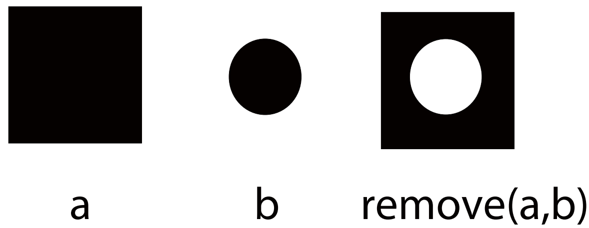

図 1: 関数 remove

```
public static Image topEdge(Image a) {
  /* a の上縁を求める. */
  return Image.remove(a, Image.down(a));
}
public static Image bottomEdge(Image a) {
  /* a の下縁を求める. */
  return Image.remove(a, 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));
}
```

図 1 に関数 remove の様子を，図 2 に関数 horizontalEdge の様子を示す.

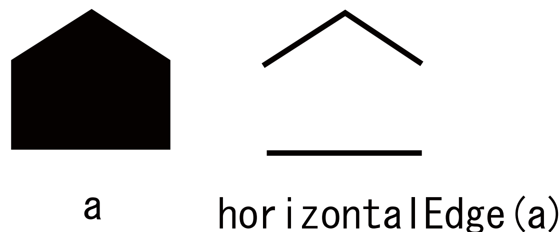

図 2: 関数 horizontalEdge

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 AOP では，ループを統合するという変換操作をアスペクト・プログラムとして別に記述する．その記述法は，図 3 のようである．細部は重要でないが，これはもとのプログラムのデータの流れがグラ

```
(cond ((and (eq (loop-shape node) 'pointwise)
            (eq (loop-shape input) 'pointwise))
       (fuse loop input 'pointwise
             :inputs (splice …)
             :loop-vars (splice …)
             :body (subst …))))
```

図 3: アスペクト・プログラム例

フとして表現されていることを前提として，そのグラフで結合関係にある 2 つの節点 (node) が，同じよう

なループを持っている場合に，それを融合するという操作を表している．このような操作が可能なようにデータフロー・グラフを作っておく必要があるが，実はそのためにもとのプログラムにも新しい記述方法を導入してある種の変更を加えている．

このやり方が，AspectJ 以前という時代性を表している．つまり，ここで用いられるアスペクト記述は，この問題に特化した特別な記述法なのである．また，そのアスペクトを元のプログラムに編み込む方法も，この問題用に特別に作られたものである．それで初めて，効率的なプログラムが生成される仕組みが得られるわけである．

しかし，図 3 のアスペクト・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，そのオブジェクトに適用できるメソッドが実質的に与えられればよい．具体的には，getPixel(int,int) というメソッドが定義できればよい．

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, and, or, 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．例として and の場合を示す．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 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));
  }
}
```

ただ，このように 1 つの演算ごとに 1 つのクラスを定義するのはむだが多いので，Java の無名クラスを利用して以下のように簡素化する．

```
class ImageOp {
  public static Image not(final Image x) {
    return new Image(x.width, x.height) {
      public Pixel getPixel(int i, int j) {
```

```
        return Pixel.not(x.getPixel(i,j));
      }
    }
  }
  public static Image and(final Image x, final Image y) {
    return new Image(x.width, x.height) {
      public Pixel getPixel(int i, int j) {
        return Pixel.and(x.getPixel(i,j), y.getPixel(i,j));
      }
    }
  }
//以下 or, up, down の定義は同様.
  ...
  public static Image remove(final Image x, final Image y) {
    return new Image(x.width, x.height) {
      public Pixel getPixel(int i, int j) {
        return and(x, not(y)).getPixel(i,j);
      }
    }
  }
  public static Image topEdge(final Image x) {
    return new Image(x.width, x.height) {
      public Pixel getPixel(int i, int j) {
        return remove(x, down(x)).getPixel(i,j);
      }
    }
  }
  //bottomEdge も同様.
  ...
  //それで最終的に
  public static Image horizontalEdge(final Image x) {
    return new Image(x.width, x.height) {
      public Pixel getPixel(int i, int j) {
        return or(topEdge(x), bottomEdge(x)).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，何よりアスペクトのプログラミング部分がまったくいらなくなる．

4 関連する手法

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．また，設計パターンではコマンド・パターンと見なすこともできる．ただ，GOF によるコマンド・パターンはこのような性能についての考慮をしたものではないようだ．

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 つ目の構成子が，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．

5 Kiczales グループの反応とその後

この話を 2001 年に Kiczales のグループにしたときの反応が面白かった．グループでも若い学生は，そもそも AOP の原典ともいうべきこの論文を知らない．それよりも，これとは別に話したわれわれの役割に基

づく協調モデル Epsilon に興味を持ってくれた．一方，Kiczales を初めとする教員や年長の研究者・学生はこの画像処理例題の話を大変熱心に聴いてくれた．とくに Kiczales は大変口惜しがり，筆者の方法の粗を探そうと努めたのが面白かった．Kiczales によれば，そもそもこの問題は，Xerox PARC の画像処理グループが実際に困っていたのを Kiczales たちが相談に乗ったというもので，現実の要求に基づくものだという．それで当時の資料を引っ張り出してきて，隠れた要求があり，それに応じるには筆者の方法だとまずいところがあるのではないか，というような議論があったが，最終的にはわれわれのやり方のよさを認めざるをえなかった．

それにしても，今の時点で見ると，AOP にとって AspectJ が作られたことはいかに大きなことだったかがつくづく思われる．もし，この時代のようにアスペクトの記述言語と織込みの処理系を問題ごとに一つ一つ作るということが続いていたとしたら，アスペクト指向プログラミングがこれほどもてはやされるようにはならなかったろう．

しかし，この画像処理の例題を現在の AspectJ で扱おうとしてもうまくいかない．AspectJ で処理速度の効率化をはかるような例がまったく扱えないわけではないが，この例で必要なループの融合のようなプログラム変換はできそうにない．AspectJ は一つの新しいモジュール化の方法を提供したが，モジュール化の可能性は多様であり，アスペクトだけが道ではないというメッセージを伝えたいというのが本稿の意図である．

謝辞　同僚の増原英彦氏には，今回東京と大阪で開催したフォーラムでも大変よい話をしてもらったが，本稿の元となった議論を 2001 年に Kiczales のグループする機会を作っていただいた．改めて感謝したい．

参考文献

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming(ECOOP), Finland*. Springer-Verlag, June 1997.

[2] 玉井哲雄．局面指向プログラミングの批判的考察．In **ウインターワークショップ・イン・高知 論文集**, pages 65–68, 高知, 1 月 1999. (社) 情報処理学会.