オブジェクトシステム 2002 ＃3

久野　靖*

2003.1.29

1　はじめに

出席メールを送ってくれるように言ったつもりなのですが、3 人しかくれませんでした。ちゃんとお願いしますよ～

でアンケートを取りましたので簡単にまとめます。

駒走さん 継承とコンポジションの違いを理解できた (多分)

コンポジションの方が設計する上で優れたメカニズムであると感じた。

以前ビジネスゲームで組んだゲームプログラムは継承の連続であったと気が付いた。組んだ後で新たな機能を加えようとしたところ全体が動いてしまい断念したのを思い出したからである。

コンポジションでの設計には経験とセンスが必要ではないかと感じた次第である。

久代さん なるほどと納得することの多い講義でした。

本を読んでた時には、コンポジションというものイメージできませんでしたが、プログラム実例の提示していただいたことで概念がよく理解できました。振り返ってみればコンポジションという名前は知りませんでしたが、実装経験の少ない私でも結構使っていますね。

オブジェクト指向分析を、きっちりと実施すると、コンポジションという実装方法は、その機構として必然となるようにも思えてきました。

吉武さん 今回の章は、結局私にとっては「継承は正しく使いましょう」とのメッセージしか感じられませんでした。本来継承が用いられるべきクラス間の関係とコンポジションにより表現されるべきクラス間の関係は、概念モデル上はある程度はっきり区別がつくもののように思われ、私が「継承マニア」と化していたときでも、今回の章にあるような不適切な継承を作って困ったことはあまりなかったように思います。

むしろ問題は継承が (酷な言い方をすれば) 柔軟性の低いコードの再利用と結び付いているがために、クラス間の関係としては概念上概ね正しく継承を用いた場合でも、後の修正/変更に耐えられない場合がある、というところにあり、これがまさにプログラミング言語上の課題であり、言語に応じてプログラマが留意しなくてはいけないところなわけだと思います。本書も「Java による」とわざわざ言っているからにはそういうところに触れて欲しかったです。(今回の章に書いてあるのは実装する言語にあまりよらない「まちがったデザイン」の例)

ちなみに、デザイン上の継承とコードの継承の関係について言うと、差分プログラミングの概念は重要とは思いますが、コードの再利用に関しては開発環境の方に細粒度のリポジトリをもたせる等、他の方法によっても有効に行えるように思われ、概念上の継承と結び付けるのは結果的にはあまりよくなかったように思われますね。

*筑波大学大学院経営システム科学専攻

結局、人は実装している対象に対して全てを知りそれを盛り込んでデザインすることは出来ないので、正しく継承を設計することはできないし、逆に継承によるコードの再利用のような便利な機能があれば (元々の言語の設計思想に関係なく) 禁止されない限りはいかなる使い方でもする、ということでしょうか?

後者は創造の源でもあるように思われますから、あながち悪いことではないのでしょうが、言語を設計する人達にとっては頭の痛いところかも知れませんね。ま、一旦言語を作ってしまえば「使い方はユーザーの自由」と身を委ねるのも一つの (たぶん正しい) 姿かとは思いますが。

2 本の担当箇所紹介 (3 章)

3 Java でのインタフェースの使い道

前回例題の紹介が好評だったようなので、「コード本」とは別に、私が Java を教えるところでインタフェースを導入する場面について紹介しましょう。

私が Java を教える時は、継承より先にインタフェースを教えています。というのは、私が一番教えたいのは動的分配 (dynamic dispatch) なので、コードとか変数とか余計なものがくっついてくるより、純粋に動的分配だけがあるインタフェースについて説明する方が自然だと思うからです。

具体的な方法としては、一番最初は「画面に描けるもの」として円とか正方形をクラスで定義します。それでクラスについて学んだ後、「こういうものが沢山あるとき、いちいちこれは円、これは正方形とか言わずに何でも同じ『描けるもの』として扱いたいよねえ」とか言うわけです。で、動かないのもつまらないのでここでは「アニメーションするもの」の例を示します (前回の例のスケールダウン版)。

```
import java.awt.*;
import java.awt.event.*;

public class Sample30 extends Frame {
  boolean go;
  double time;
  Animation[] a = new Animation[20];
  int count = 0;
  public Sample30() {
    a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 30, -40);
    a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 40, 50);
    a[count++] = new RandomSquare(Color.green, 100, 100, 20, 10);
  }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
  public void start() { go = true; (new Thread(new MyRun())).start(); }
  public void stop() { go = false; }
  class MyRun implements Runnable {
    public void run() {
      time = 0.001 * System.currentTimeMillis();
      while(go) {
        try { Thread.sleep(100); } catch(Exception e) { }
        double dt = 0.001 * System.currentTimeMillis() - time;
        for(int i = 0; i < count; ++i) a[i].addTime(dt);
        time += dt; repaint();
      }
```

```
    }
  }
  public static void main(String[] args) {
    Sample30 app = new Sample30();
    app.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    app.setSize(300, 300); app.setVisible(true); app.start();
  }
  interface Animation {
    public void addTime(double dt);
    public void draw(Graphics g);
  }
  static class FlyingCircle implements Animation {
    Color col;
    double gx, gy, rad, vx, vy;
    public FlyingCircle(Color c, double x, double y, double r,
                        double vx1, double vy1) {
      col = c; gx = x; gy = y; rad = r; vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      gx += vx*dt; gy += vy*dt;
      if(gx < 0 && vx < 0 || gx > 300 && vx > 0) vx = -vx;
      if(gy < 0 && vy < 0 || gy > 300 && vy > 0) vy = -vy;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval((int)(gx-rad), (int)(gy-rad), (int)rad*2, (int)rad*2);
    }
  }
  static class RandomSquare implements Animation {
    Color col;
    double gx, gy, len, vel;
    public RandomSquare(Color c, double x, double y, double l, double v) {
      col = c; gx = x; gy = y; len = l; vel = v;
    }
    public void addTime(double dt) {
      gx += vel*(Math.random()-0.5); gy += vel*(Math.random()-0.5);
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect((int)(gx-len/2), (int)(gy-len/2), (int)len, (int)len);
    }
  }
}
```

演習 1 これは/u1a/kuno/work/Sample30.javaに入っている。もってきて動かした後、何でもいいから「別の動くもの」を追加してみよ。

4　モデルとユーザインタフェースの分離

MVC フレームワークのように、「計算の中身と外見を分離する」というのはインタフェースのよい用法。そのデモンストレーションの例題を作ってみた。

```
import java.awt.*;
import java.awt.event.*;

public class Sample31 extends Frame {
  MyApp app;
  Label l0 = new Label();
  Button b0 = new Button();
  TextField t0 = new TextField();
  public Sample31(MyApp a) {
    app = a;
    l0.setText(app.getMessage());
    b0.setLabel(app.getName());
    setLayout(null); setSize(200, 250);
    add(l0); l0.setBounds(10, 40, 180, 40);
    add(t0); t0.setBounds(10, 100, 180, 40);
    add(b0); b0.setBounds(10, 160, 90, 40);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        app.setData(t0.getText()); t0.setText("");
        l0.setText(app.getMessage());
      }
    });
  }
  public static void main(String[] args)
      throws Exception {
//  Frame f = new Sample31(new App1());
//  Frame f = new Sample31(new App2());
    Frame f = new Sample31((MyApp)
      (Class.forName(args[0])).newInstance());
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
  }
}

interface MyApp {
  public String getName();
  public String getMessage();
  public void setData(String s);
}
```

```
class App1 implements MyApp {
  String mesg = "F to C converter";
  public String getName() { return "F to C"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      float x = (new Float(s)).floatValue();
      float y = (9f/5f)*(x - 32f);
      mesg = "Temp in C: " + y;
    } catch(Exception e) { mesg = e.toString(); }
  }
}

class App2 implements MyApp {
  double sum = 0.0;
  String mesg = "Calculate Sum";
  public String getName() { return "Sum"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      sum += (new Double(s)).doubleValue();
      mesg = "Sum is: " + sum;
    } catch(Exception e) { mesg = e.toString(); }
  }
}
```

このプログラムは同一のユーザインタフェースで 2 つのアプリケーションを切り替えられる。アプリケーションの「切り口」がインタフェース MyApp で定義されている。

演習 2 これは/u1a/kuno/work/Sample31.javaに入っている。もってきて動かした後、何でもいいから「別のアプリケーション」を追加してみよ。

さらにこれに、別のユーザインタフェースをつけることもできる。たとえばコマンド行インタフェースをつけてみた。

```
import java.io.*;

public class Sample32 {
  public static void main(String[] args)
      throws Exception {
//  MyApp app = new App1());
//  MyApp app = new App2());
    MyApp app = (MyApp)
      (Class.forName(args[0])).newInstance();
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    while(true) {
      System.out.println(app.getMessage());
      System.out.print("> ");
```

```
      String line = in.readLine();
      if(line.equals("")) break;
      app.setData(line);
    }
  }
}

interface MyApp {
  public String getName();
  public String getMessage();
  public void setData(String s);
}

class App1 implements MyApp {
  String mesg = "F to C converter";
  public String getName() { return "F to C"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      float x = (new Float(s)).floatValue();
      float y = (9f/5f)*(x - 32f);
      mesg = "Temp in C: " + y;
    } catch(Exception e) { mesg = e.toString(); }
  }
}

class App2 implements MyApp {
  double sum = 0.0;
  String mesg = "Calculate Sum";
  public String getName() { return "Sum"; }
  public String getMessage() { return mesg; }
  public void setData(String s) {
    try {
      sum += (new Double(s)).doubleValue();
      mesg = "Sum is: " + sum;
    } catch(Exception e) { mesg = e.toString(); }
  }
}
```

演習 3 これは/u1a/kuno/work/Sample32.javaに入っている。もってきて動かした後、演習 2 で追加した「別のアプリケーション」がこれでも動くことを確認せよ。

5 Swing におけるモデルの分離

実は Java の標準 API でもインタフェースによるモデルと UI の分離は行われている。たとえば Swing の部品 JTree や JTable でも「モデルとビューの分離」がなされている。つまり、画面に現れるツリーや表は内部のデータを外部に見える形で示しているだけで、内部のデータはツリーや表でなくても構わない (図 1)。

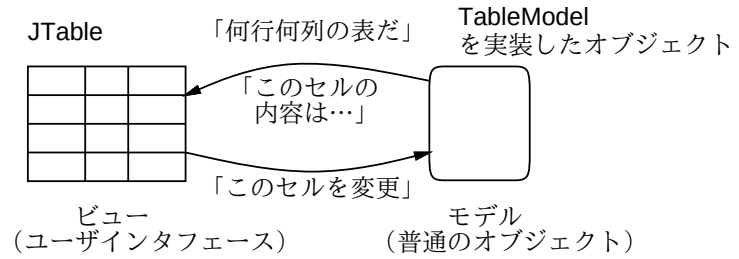

図 1: 表のビューとモデル

たとえば、表の中身は TableModel というインタフェースを実装するオブジェクトとして表すことができる。そのオブジェクトにおいては、たとえば次のようなメソッドを定義することができる。

- int getRowCount() — 表の行数を返す
- int getColumnCount() — 表の列数を返す
- String getColumName(i) — i 列目の列名を文字列として返す
- boolean isCellEditable(r,c) — r 行 c 列目のセルをユーザが編集してもよいか否かを返す
- getValueAt(r,c) — r 行 c 列目のセルに対応するオブジェクトを返す
- setValueAt(Object,r,c) — r 行 c 列目のセルの値として設定したいオブジェクトを渡して呼び出される

そのほかにも多数のメソッドがあって全部用意するのは大変なので、おなじみ MouseAdapter 同様、AbstractTableModel というクラスが用意されていて、そのサブクラスを作るようにすれば自分が再定義したいメソッドだけを書けば済む (getRowCount()、getColumnCount()、getValueAt() の 3 つは抽象メソッドなので必ず書く必要があるが)。

以上をまとめると、表を使いたい時は AbstractTableModel のサブクラスとして自分の好きな情報を保持するクラスを作り、それを JTable のコンストラクタに渡して表を作ることで、データが表の形で表示され、またユーザが表に記入してデータを更新することができる。具体例を見てみよう。

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;

public class R12Sample2 extends JFrame {
  public R12Sample2() {
    getContentPane().add(new JScrollPane(new JTable(new MyTableModel())));
  }
  public static void main(String[] args) {
    R12Sample2 app = new R12Sample2();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setSize(300, 300); app.setVisible(true);
  }
  class MyTableModel extends AbstractTableModel {
    double start = 1.0, step = 0.1;
    public int getRowCount() { return 25; }
    public int getColumnCount() { return 3; }
```

```
    public boolean isCellEditable(int r, int c) { return r == 1 && c == 1; }
    public Object getValueAt(int r, int c) {
      if(c == 0) {
        return new Integer(r);
      } else if(c == 1) {
        double x = start;
        for(int i = 0; i < r; ++i) x = x + step;
        return new Double(x);
      } else {
        return "?";
      }
    }
    public void setValueAt(Object o, int r, int c) {
      if(r == 1 && c == 1) {
        step = (new Double(o.toString())).doubleValue() - 1.0;
      }
      fireTableDataChanged();
    }
  }
}
```

プログラムの全体構造はなるべく簡単にするため、「内側をスクロール可能にする部品 (JScrollPane) の内側に後で定義する MyTableModel クラスのインスタンスをデータとして持つような JTable を貼りつけているだけである。[1]

さて、結局このプログラムの「中心」は MyTableModel クラスということになる。このクラスはデータとしては step という double 型の変数 1 つだけを持っている。ということはデータというのはそれしかないわけだ。さて、ここから先は表とのやりとりのためのメソッドだが、まず表の行数は 25、列数は 3、そして「1 行 1 列」だけが変更可能としている (最初の行と列は 0 行と 0 列)。そして、0 列目のデータとしては行の番号を返して表示させる。1 列目のデータとしては i 行目に初項が 1、階差が step の等差数列の i 番目の項を表示する。それ以外の、ということは 2 列目のセルには「?」を表示させる。

データが変更された時の処理は、変更されるのは 1 行 1 列だけだが一応そのことをチェックし、渡されたものを文字列に変換し、double に変換し、初項 1 を引く。ということは、階差が求まるわけで、それを step に入れる。これでデータが変ったので表の内容は全部変化する可能性がある。その場合は AbstractTableModel から継承している自分のメソッド fireTableDataChanged() というメソッドを呼ぶと表の方を再表示してくれるようになっている。

このように、「モデルとユーザインタフェース (表示部分) を分離する」という考え方は、プログラムの構造が整理される (半分ずつに分けると作成の労力はそれぞれが元の 1/4 くらいずつになると言われている)、1 つのモデル (計算処理) でユーザインタフェースだけ取り替えることができる、などの利点があるので、機会があれば活用してみることをお勧めする。具体的には次のように考えるとよい。

- 自分がしたい計算部分を「外からアクセスする」にはどのようなメソッドがあれば十分かを考え、それを Java のインタフェースとして定義してやる。
- 計算部分はそのインタフェースを implements するクラスとして作る。
- ユーザインタフェース部分はそのインタフェースを implements したオブジェクトを最初に受けとり、そのメソッドを呼び出しながら表示や入力を行うように作る。

演習 4 これは/u1a/kuno/work/Sample34.java に入っているのでそのまま動かせ。動いたら次のように変更してみよ。

[1] なぜかメニューバーを使うと JScrollPane がうまく動かなかったという理由もあって簡単にしてある…

(a) 上の例では初項が「1.0」に固定だったが、これを (0 行 1 列のセルを編集することで) 変更できるようにしてみよ。

(b) 上の例では 1 番目の値だけが変更できたが、1 列目の他のセルも変更できるようにしてみよ (変更するとそれに応じて正しい等差数列になるように他のセルも変化するようにする)。

(c) 2 列目に等比数列を表示するようにせよ。つまり 0 行 1 列が初項 a で、1 行目は ar、2 行目は ar^2、… となるようにする。もちろん、1 行目と 2 行目はデータを打ち込めるようにする (他の行にも打ち込めるとなおよい)。

(d) 2 列目に各数が素数か否かを表示させよ。つまり行の番号が「2」や「3」なら「prime」、「4」なら「not prime」のように表示されるようにする。表の行数を 10000 にしても、実際に表示に必要な部分しか素数の計算は行われないので、遅くはないことを確認する。

(e) 住宅ローン計算プログラムを作れ。最初に借りた金額と、毎年の返済額と、年利をどこかのセルに入力させる。毎年、前年の債務残高から返済額を引き、債務残高に年利を書けた値を加えたものが翌年の債務残高になるわけ。

(f) 表を使って何か面白いプログラムを作れ。