オブジェクトシステム 2002 ＃2

久野　靖*

2003.1.29

1　はじめに

皆様に出席メール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 本の担当 2.1-2.3。求めるもの→ (1) オブジェクト指向プログラミングの基礎知識、(2) 営業系の人間が技術系の人間にうまく要求定義を伝えるための知識。

打矢さん 本の担当 2.4-2.6。求めるもの→ (1)OO に関する体系だった知識、(2)OO における情報の捉え方/扱い方。

久代さん 本の担当 2.7-2.10。求めるもの→ (1)UML による記述の修得、(2) 集約、継承等が Java にどうマッピングされるか。

磯部さん 本の担当 3.1-3.3。求めるもの→ (1) オブジェクト指向の話、(2)GoF のパターン、(3)WebSphere、(4) 設計例題を皆で書く。

中山さん 本の担当 3.4-3.5。求めるもの→オブジェクト指向言語でのよりよい設計方法、デザインパターンについての指針。

吉武さん 本の担当 3.6-3.9。求めるもの→オブジェクト設計やその Java による実装についての議論 (?)

全員の要望に応えるのはちょっと難しいでしょうし主題違いな要望もありますが…とりあえず、本を紹介しつつ手持ち材料からも資料を用意して、時間があれば演習もできるように準備しておくことにします。あと、2/12(水) は都合により授業は 20:00 までになりますので、その後「自習」用に演習を用意するかも知れません。

2　本の担当箇所紹介 (2 章)

3　Java で窓を開くプログラムを作る

以下、動くプログラムがあった方が納得してもらいやすいと思うので例題を見てもらいながら説明します。なお、この部分は某東京大学で (何が某だ?)1 年生の科目の非常勤に行っている時の資料から抜粋して調整しています。

最初に、知ってる人にはどうでもいいですが、窓を開くプログラムの作り方から。まず、「自前のウィンドウ」を作る機能はクラス java.awt.Frame が提供してくれている。ただし、Frame は「何も中身がない」窓を作るようになっているので、そのサブクラスとして「自分なりの中身を持った窓」を用意する。つまり次のような感じになる。

*筑波大学大学院経営システム科学専攻

```
import java.awt.*;

public class XXX extends Frame {
  public XXX() {
    // 初期設定...
  }
  // その他のメソッド (必要なら)...
}
```

次に、このクラスのインスタンスを生成し、窓を開くための main() がどこかに必要である。main() はどこかのクラスに public static メソッドとして入れておけばよい。ここでは簡単のため、「どこかのクラス」としてこのウィンドウのクラスをそのまま利用してしまおう。

```
import java.awt.*;

public class XXX extends Frame {
  public XXX() {
    setSize(幅, 高さ); // 窓の幅と高さはここで設定できる
    // 初期設定...
  }
  // その他のメソッド (必要なら)...
  public static void main(String[] args) { (new XXX()).setVisible(true); }
}
```

これで、「java XXX」のように実行開始するとまず main() が動き、XXX オブジェクト (というのは自分独自の窓) を生成して、それをメソッド setVisible() を呼び出して「見える状態にする」。

それはよいけど、このプログラムはいつ終るのだろう? これまでの普通のプログラムは main() が終ると終っていたが、今度は新しい窓を作ったため、main() が終っても窓が開いている限りプログラムは動き続ける。実は、このようなプログラムが終るためにはクラス System に備わっているクラスメソッド exit() を呼ぶ必要がある。

いつ呼ぶか…? それはたとえば「Quit ボタンが押された時」でもよいけど、実はウィンドウに対してウィンドウマネージャが「閉じる」操作 (「×」アイコンのクリックとか、「窓の削除」機能に対応) を実行すると「閉じようとしている」というイベントが送られるので、そのイベントに対する処理としてプログラムを終らせるようにした方がかっこうがよい。そのために次のようにする。

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

public class XXX extends Frame {
  public XXX() {
    setSize(幅, 高さ); // 窓の幅と高さはここで設定できる
    // 初期設定...
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
  }
  // その他のメソッド (必要なら)...
  public static void main(String[] args) { (new XXX()).setVisible(true); }
}
```

WindowListner はウィンドウ関係のイベントを受け取るためのインタフェース。WindowAdapter は興味のないイベントのメソッドを書かないで済ませるための「お役立ち」クラスで、すべてのイベントに対して「何もしない」メソッドが定義されているので、これを継承した無名内部クラスで必要なメソッドだけオーバライドする。他のイベントについては API ドキュメント参照。

4 継承とインタフェースによるクラス階層設計

4.1 継承による差分プログラミング

さて、クラスの集まりを構造化する方法として、「継承」を駆使して、既にあるクラスを土台に新しいクラスを次々作って行くやり方がある。これを「差分プログラミング」とよぶ。例を見てみよう。まず冒頭部分はこれまで使って来たようなインスタンス変数類の準備。

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

public class R10Sample2 extends Frame {
  Image offimage;
  Graphics offgraphics;
  boolean go = true;
  double time = System.currentTimeMillis()*0.001;
  Animation[] a = new Animation[20];
  int count = 0;

  public static void main(String[] args) {
    R10Sample2 app = new R10Sample2();
    app.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    app.setSize(400, 300); app.setVisible(true); app.start();
  }
```

ここでは main() で窓を作ったあと、アダプタクラスの設定、大きさの設定、スレッドの開始などまで行っている。そしてコンストラクタは配列 a にさまざまなオブジェクトを要れるだけ。ここでは 2 番目以降はコメントアウトしてある (あとで順次復活させる)。

```
  public R10Sample2() {
    a[count++] = new MovingCircle(Color.red, 100, 100, -30, 40, 15);
  //a[count++] = new MovingJack(new Color(80, 120, 180), 10, 10, 80, 30, 30);
  //a[count++] = new LightOnOffJack(new Color(40, 120, 80),
  //                10, 30, 20, 20, 30, Color.red);
  //a[count++] = new MovingSnowman(new Color(60, 100, 80),
  //                110, 30, 30, 20, 30, Color.red, 1.2);
  //a[count++] = new WavingSnowman(new Color(120, 240, 80),
  //                150, 70, 40, 20, 30, Color.red, 1.2);
  //a[count++] = new LongNeckSnowman(new Color(150, 80, 80),
  //                110, 70, 40, -15, 30, Color.red, 1.2);
  //a[count++] = new RotateSnowman(new Color(110, 180, 120),
```

```
//                  110, 110, -10, -15, 30, Color.red, 1.2);
}
```

さて、アニメーションの開始/停止や描画関係はこれまでと同じなので特に説明不要。

```
public void start() {
  go = true;
  new Thread(new Runnable() {
    public void run() {
      while(go) {
        try { Thread.sleep(50); } catch(Exception ex) { }
        double dt = System.currentTimeMillis()*0.001 - time;
        for(int i = 0; i < count; ++i) a[i].addTime(dt);
        time += dt; repaint();
      }
    }
  }).start();
}
public void stop() { go = false; }
public void update(Graphics g) {
  int w = getSize().width, h = getSize().height;
  if(offimage == null) {
    offimage = createImage(w, h); offgraphics = offimage.getGraphics();
  }
  offgraphics.setColor(getBackground()); offgraphics.fillRect(0, 0, w, h);
  paint(offgraphics); g.drawImage(offimage, 0, 0, this);
}
public void paint(Graphics g) {
  for(int i = 0; i < count; ++i) a[i].draw(g);
}
```

インタフェース Animation は描画と時間を進めるメソッドを定義。

```
interface Animation {
  public void draw(Graphics g);
  public void addTime(double dt);
}
```

さて、ここから各図形クラスがはじまる。最初は「飛ぶ円」。

```
static class MovingCircle implements Animation {
  Color cl;
  double gx, gy, vx, vy, rad;
  public MovingCircle(Color cl1, double x, double y,
                      double vx1, double vy1, double rad1) {
    cl = cl1; gx = x; gy = y; vx = vx1; vy = vy1; rad = rad1;
  }
  public void draw(Graphics g) {
    g.setColor(cl);
    g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
  }
```

```
    public void addTime(double dt) {
      gx += vx * dt; gy += vy * dt;
      if (gx<0.0 && vx < 0.0) { vx = -vx;}
      if (gx>400.0 && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>300.0 && vy > 0.0) { vy = -vy;}
    }
  }
```

さて、ここから差分プログラミングになる。飛ぶ円の中に目を描いて飛ぶ顔に直す。そのために、MovingCircle のサブクラスを作って paint をオーバライドする。また、コンストラクタは継承できないのでこれも新しく作る。コンストラクタの中で「super(...)」により、親クラスのコンストラクタを呼び出していること、paint() の中で「super.paint(...)」により親クラスの paint() を呼び出していることに注意。このような、サブクラスの中から適宜親クラスの機能を参照できるような仕組みをマスターしておかないと、継承を使いこなすのはむずかしい。

```
  static class MovingJack extends MovingCircle {
    Color eyeColor, mouthColor;
    public MovingJack(Color c, double x, double y,
                      double vx1, double vy1, double rad1) {
      super(c, x, y, vx1, vy1, rad1); eyeColor = mouthColor = c.brighter();
    }
    public void draw(Graphics g) {
      int u = (int)rad/4;
      super.draw(g);
      g.setColor(eyeColor);
      g.fillPolygon(new int[]{(int)gx-3*u, (int)gx-2*u, (int)gx-u},
                    new int[]{(int)gy-u, (int)gy-2*u, (int)gy-u}, 3);
      g.fillPolygon(new int[]{(int)gx+3*u, (int)gx+2*u, (int)gx+u},
                    new int[]{(int)gy-u, (int)gy-2*u, (int)gy-u}, 3);
      g.setColor(mouthColor);
      g.fillPolygon(new int[]{(int)gx-u, (int)gx, (int)gx+u},
                    new int[]{(int)gy+u, (int)gy+2*u, (int)gy+u}, 3);
    }
  }
```

さて、次は目の色が時間とともに変わる顔を作る。このクラスは先のクラス MovingJack のサブクラスとし、addTime() を差し替えて時間の累計を取るようにした (その後で親クラスの addTime() も呼ぶ)。また draw() も差し替えて、現在の累計した秒が奇数か偶数かで eyeColor を切替えてから親クラスの draw() を呼ぶ。実は MovingJack で eyeColor を別の変数として保持していたのがポイントで、そうしてなければこう簡単に目だけ色を変えることはできない。このように、継承では親クラスが子クラスで書き換えるように変数の構成をうまく設計すること、子クラスはその親クラスの内部構造をわかった上で変数などを書き換えることが必要になる。

```
  static class LightOnOffJack extends MovingJack {
    double atime = 0.0;
    Color lightOnColor;
    public LightOnOffJack(Color c, double x, double y, double vx, double vy,
                          double rad1, Color c1) {
      super(c, x, y, vx, vy, rad1); lightOnColor = c1;
```

```
    }
    public void addTime(double dt) {
      atime += dt; super.addTime(dt);
    }
    public void draw(Graphics g) {
      if((int)atime % 2 == 0) eyeColor = cl.brighter();
      else eyeColor = lightOnColor;
      super.draw(g);
    }
  }
```

次に、顔だけではつまらないので「本体」もつけて雪ダルマにしよう。これも先の LightOnOffJack のサブクラスで、「本体」をどれくらい顔から離すかを変数 dx、dy に保持することにした (もちろん、さらにサブクラスを作る時にこれらを活用する)。

```
  static class MovingSnowman extends LightOnOffJack {
    double ratio, dx = 0, dy = 0;
    public MovingSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1); ratio = r; dy = rad*2;
    }
    public void draw(Graphics g) {
      g.setColor(cl);
      g.fillOval((int)(gx+dx-rad*ratio), (int)(gy+dy-rad*ratio),
                 (int)(rad*ratio*2), (int)(rad*ratio*2));
      super.draw(g);
    }
  }
```

さて、雪だるまの本体を上下に振動させてみよう。それには、MovingSnowman のサブクラスを作って、時間を累計するところで dy の値を時刻の sin 関数に従って振動させればよい。

```
  static class WavingSnowman extends MovingSnowman {
    public WavingSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
    public void addTime(double dt) {
      super.addTime(dt); dy = rad*2 - (ratio-1)*rad*(1+Math.sin(20*atime));
    }
  }
```

もっと別のいじくり方として、ろくろ首を作ってみよう。今度のクラスもまた MovingSnowman のサブクラスとして、今度は dy をのこぎり状に変化させ、なおかつ「首」の描画を追加している。ちょっとキタナイが、もともとの dy の値を壊さないために、super.draw() の直前で dy を増やし、その後 dy を戻している。

```
  static class LongNeckSnowman extends MovingSnowman {
    public LongNeckSnowman(Color c, double x, double y, double vx1,
             double vy1, double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
```

```
    public void draw(Graphics g) {
      int len = (int)(rad*(atime%1));
      g.setColor(cl);
      g.fillRect((int)(gx-0.3*rad),(int)gy,(int)(0.6*rad),(int)rad*2+len);
      dy += len; super.draw(g); dy -= len;
    }
  }
```

また別の MovingSnowman のサブクラスとして、本体が顔のまわりを円周運動するというのも作ってみた。もう十分わかったでしょう?

```
  static  class RotateSnowman extends MovingSnowman {
    public RotateSnowman(Color c, double x, double y, double vx1, double vy1,
                         double rad1, Color c1, double r) {
      super(c, x, y, vx1, vy1, rad1, c1, r);
    }
    public void addTime(double dt) {
      super.addTime(dt);
      dx = rad*2*Math.cos(atime); dy = rad*2*Math.sin(atime);
    }
  }
}
```

演習 上のプログラムは/u1a/kuno/work/R10Sample2.java に入っているのでコピーしてきてそのまま動かせ。動いたらどれかのクラスのサブクラスを作って新しい絵 (?) を追加してみよ。

4.2 複合 (コンポジション) による構造化

さて、差分プログラミングを見てどう思いましたか? 「なるほど、うまくやっているなあ」と思った人はだまされている。だって、この調子でつぎ足しつぎ足しして本当にきれいなプログラムができるとは思えないでしょう? たとえば「顔が四角いろくろ首」を作ろうと思ったら、ちょっとサブクラスを作って、というわけには行かない。一般に、N 個の選択肢と M 個の選択肢があった場合、その任意の組合せは $M \times N$ 通りになる。これをサブクラスでやろうとして $M \times N$ 個のサブクラスを作っていては、とても手に負えない。

これに対する回答は次のようなものである。つまり、サブクラスで親クラスの機能を書き換えて増やすのではなく、「機能だけを別のクラスにして」それと既存のクラスを「複合させて」必要なものを組み立てる。こうすれば、M 個のクラスと N 個のクラスをそれぞれ用意すれば済むわけだ。これをコンポジションなどと呼ぶ。

では、さっきの例題を (全部作るのは大変だから途中まで) コンポジション型に手直ししてみよう。冒頭部分は変わらない。

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

public class R10Sample3 extends Frame {
  Image offimage;
  Graphics offgraphics;
  boolean go = true;
  double time = System.currentTimeMillis()*0.001;
  Animation[] a = new Animation[20];
```

```
  int count = 0;

  public static void main(String[] args) {
    R10Sample3 app = new R10Sample3();
    app.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
    app.setSize(400, 300); app.setVisible(true); app.start();
  }
```

次に、図形を増やす部分はちょっと違って来ているが、どう違うかは少し後で説明する。

```
  public R10Sample3() {
    a[count++] = new FlyingMove(new Circle(Color.red, 100, 100, 15), -30, 40);
  //a[count++] = new CircleMove(new Circle(Color.blue, 120, 100, 10), 30, 5);
  //a[count++] = new CircleMove(new Triangle(Color.green,
  //                              100, 100, 140, 100, 80, 120), 40, 2);
  //a[count++] = new ChgColor(new Circle(Color.red, 100, 140, 20),
  //               new Color[]{Color.pink, Color.cyan, Color.black}, 0.5);
  //AnimGroup g1 = new AnimGroup(160, 100);
  //g1.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
  //g1.add(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10));
  //g1.add(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10));
  //g1.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
  //a[count++] = new FlyingMove(g1, 45, 35);
  //AnimGroup g2 = new AnimGroup(160, 100);
  //g2.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
  //g2.add(new ChgColor(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10),
  //         new Color[]{Color.red, Color.white}, 0.7));
  //g2.add(new CircleMove(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10),
  //         3.0, 5.0));
  //g2.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
  //a[count++] = new FlyingMove(g2, 25, 45);
  }
```

スレッドとか描画は前と同じ。

```
  public void start() {
    go = true;
    new Thread(new Runnable() {
      public void run() {
        while(go) {
          try { Thread.sleep(50); } catch(Exception ex) { }
          double dt = System.currentTimeMillis()*0.001 - time;
          for(int i = 0; i < count; ++i) a[i].addTime(dt);
          time += dt; repaint();
        }
      }
    }).start();
  }
```

```
  public void stop() { go = false; }
  public void update(Graphics g) {
    int w = getSize().width, h = getSize().height;
    if(offimage == null) {
      offimage = createImage(w, h); offgraphics = offimage.getGraphics();
    }
    offgraphics.setColor(getBackground()); offgraphics.fillRect(0, 0, w, h);
    paint(offgraphics); g.drawImage(offimage, 0, 0, this);
  }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }
```

インタフェース Animation はメソッドがかなり増えている。というのは、継承を使わずに「はめ込み」で組み立てるためには、はめ込むクラスがどういうメソッドを持っているかをきちんと定義しておく必要があるから。

```
  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
    public void moveTo(double x, double y);
    public double getX();
    public double getY();
    public void setColor(Color c);
    public Color getColor();
  }
```

ではまず、円クラスを用意する。このクラスは勝手に飛んだりしない、単なる円を表している。

```
  static class Circle implements Animation {
    Color cl;
    double gx, gy, rad;
    public Circle(Color c, double x, double y, double r) {
      cl = c; gx = x; gy = y; rad = r;
    }
    public void draw(Graphics g) {
      g.setColor(cl);
      g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
    }
    public void addTime(double dt) { }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
```

次は「飛ぶ」機能だけをクラスとして用意する。このクラスは Animation を実装したオブジェクト (つまり図形) を 1 つ受け取り、その図形を「飛ばす」部分だけを受け持つ。それには、時間とともに gx、gy を変化させ、その位置に保持している図形を動かせばよい。これを利用する側では、FlyingMove オブジェクトを作る時に、飛ばしたい図形をパラメタとして渡してやればよい。

```
  static class FlyingMove implements Animation {
    Animation anim;
    double gx, gy, vx, vy;
    public FlyingMove(Animation a, double vx1, double vy1) {
      anim = a; gx = a.getX(); gy = a.getY(); vx = vx1; vy = vy1;
    }
    public void draw(Graphics g) { anim.draw(g); }
    public void addTime(double dt) {
      anim.addTime(dt); gx += vx * dt; gy += vy * dt;
      if (gx<0.0 && vx < 0.0) { vx = -vx;}
      if (gx>400.0 && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>300.0 && vy > 0.0) { vy = -vy;}
      anim.moveTo(gx, gy);
    }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { anim.setColor(c); }
    public Color getColor() { return anim.getColor(); }
  }
```

こうしておけば、飛ぶ代りに円周上を動くようにするという別の機能を用意してそれと Circle を組み合わせるのも問題ない。

```
  static class CircleMove implements Animation {
    Animation anim;
    double gx, gy, rad, vtheta, theta = 0;
    public CircleMove(Animation a, double r, double vt) {
      anim = a; gx = a.getX(); gy = a.getY(); rad = r; vtheta = vt;
    }
    public void draw(Graphics g) {
      anim.moveTo(gx+rad*Math.cos(theta), gy+rad*Math.sin(theta));
      anim.draw(g);
    }
    public void addTime(double dt) { anim.addTime(dt); theta += vtheta * dt; }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { anim.setColor(c); }
    public Color getColor() { return anim.getColor(); }
  }
```

図形の方も円だけでなく三角形を増やそう。三角形も円と同様に飛んだり円周軌道で動いたりさせられる (つまり任意に組み合わせられる)。

```
  static class Triangle implements Animation {
    Color cl;
    double gx, gy, dx0, dy0, dx1, dy1, dx2, dy2;
    public Triangle(Color c, double x0, double y0, double x1, double y1,
```

```
                    double x2, double y2) {
      cl = c; gx = (x0+x1+x2)/3; gy = (y0+y1+y2)/3;
      dx0 = x0-gx; dx1 = x1-gx; dx2 = x2-gx;
      dy0 = y0-gy; dy1 = y1-gy; dy2 = y2-gy;
    }
    public void draw(Graphics g) {
      int[] x = new int[]{(int)(gx+dx0),(int)(gx+dx1),(int)(gx+dx2)};
      int[] y = new int[]{(int)(gy+dy0),(int)(gy+dy1),(int)(gy+dy2)};
      g.setColor(cl); g.fillPolygon(x, y, 3);
    }
    public void addTime(double dt) { }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
```

もうちょっと別の機能として、「色を変化させる」ことだけをクラスとして分離したものも作ってみた。これはコンストラクタで色の配列を受け取り、その色の順番に色が変化していく。

```
  static class ChgColor implements Animation {
    Animation anim;
    Color[] colors;
    double time = 0, period;
    public ChgColor(Animation a, Color[] c, double p) {
      anim = a; colors = c; period = p;
      if(c.length == 0) throw new RuntimeException("empty color array");
    }
    public void draw(Graphics g) { anim.draw(g); }
    public void addTime(double dt) {
      anim.addTime(dt); time += dt;
      anim.setColor(colors[(int)(time/period) % colors.length]);
    }
    public void moveTo(double x, double y) { anim.moveTo(x, y); }
    public double getX() { return anim.getX(); }
    public double getY() { return anim.getY(); }
    public void setColor(Color c) { }
    public Color getColor() { return anim.getColor(); }
  }
```

さて、顔とかはどうすればいいのか? それには、「複数の図形をくっつけた図形」が作れるようにすればいい。

```
  static class AnimGroup implements Animation {
    Animation[] a = new Animation[20];
    int count = 0;
    Color cl = Color.black;
    double gx, gy;
    public AnimGroup(double x, double y) { gx = x; gy = y; }
    public void add(Animation anim) {
```

```
      if(count+1 < a.length) a[count++] = anim;
    }
    public void draw(Graphics g) {
      for(int i = 0; i < count; ++i) {
        double x = a[i].getX(), y = a[i].getY();
        a[i].moveTo(x+gx, y+gy); a[i].draw(g); a[i].moveTo(x, y);
      }
    }
    public void addTime(double dt) {
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
    }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
}
```

複合図形クラスを使って円と 3 つの三角形を組み合わせれば顔ができる。しかし動かない三角形の代わりに、色が変化する三角形とか、円周軌道で動く三角形とかを使ってもよい。このように、コンポジションを基本にしておけば継承よりもずっと用意に機能の任意の組み合わせが活用できる。

演習 2 上のプログラムは/u1a/kuno/work/R10Sample3.java に入っているのでコピーしてきてそのまま動かせ。動いたら機能や図形の組み合わせを変えて別の動きをする絵を作ってみよ。もし余裕があれば、新しい機能を追加するとなおよい。

4.3 再び継承によるくくり出し+抽象クラス

コンポジションの利点は分かったが、どうも同じメソッド等を繰り返し書くのでプログラムが長くて冗長だ、と思った人もいるかと思う。そう、そこを何とかしたいですね? それには…その「共通部分」をくくり出すために継承を使えばよい。この場合は、差分プログラミングのように延々と長い継承の連鎖ができるというより、複数の類似したクラスの共通部分を 1 つの親クラスにくくり出す、という感じになる。

ところで、そのようなくくり出しを行う時、くくり出した親クラスは「共通部分の置き場所」であり、実際にインスタンスを生成することはない場合が多い。たとえば複数の図形の共通部分をくくり出した場合、その共通部分を集めた親クラスはメソッド `draw()` が定義できない (だって特定の形は持っていないから)。このようなメソッドは、インタフェースは定義されているが本体 (コード部分) をただの「;」だけにして、キーワード `abstract` をつけて定義する。または、インタフェースで定義されているメソッドに本体 (コード) をつけない場合もこれと同じ扱いになる。このようなメソッドを「抽象メソッド」(abstract method)、抽象メソッドを持つようなクラスを「抽象クラス」と呼ぶ (分かりにくいかも知れないが、要するにインタフェースでのメソッド定義は全て自動的に `abstract` がつけられて処理されると思ってください)。

先の例題を上のような考え方で整理してみる。各種クラスのはじまりまでずっと同じ。

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

public class R10Sample4 extends Frame {
  Image offimage;
```

```
Graphics offgraphics;
boolean go = true;
double time = System.currentTimeMillis()*0.001;
Animation[] a = new Animation[20];
int count = 0;

public static void main(String[] args) {
  R10Sample4 app = new R10Sample4();
  app.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) { System.exit(0); }
  });
  app.setSize(400, 300); app.setVisible(true); app.start();
}
public R10Sample4() {
  a[count++] = new FlyingMove(new Circle(Color.red, 100, 100, 15), -30, 40);
  a[count++] = new CircleMove(new Circle(Color.blue, 120, 100, 10), 30, 5);
  a[count++] = new CircleMove(new Triangle(Color.green,
                                100, 100, 140, 100, 80, 120), 40, 2);
  a[count++] = new ChgColor(new Circle(Color.red, 100, 140, 20),
                new Color[]{Color.pink, Color.cyan, Color.black}, 0.5);
  AnimGroup g1 = new AnimGroup(160, 100);
  g1.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
  g1.add(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10));
  g1.add(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10));
  g1.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
  a[count++] = new FlyingMove(g1, 45, 35);
  AnimGroup g2 = new AnimGroup(160, 100);
  g2.add(new Circle(new Color(200, 155, 120), 0, 0, 40));
  g2.add(new ChgColor(new Triangle(Color.blue, -30, 0, -10, 0, -20, -10),
           new Color[]{Color.red, Color.white}, 0.7));
  g2.add(new CircleMove(new Triangle(Color.blue, 30, 0, 10, 0, 20, -10),
           3.0, 5.0));
  g2.add(new Triangle(Color.blue, -10, 10, 10, 10, 0, 20));
  a[count++] = new FlyingMove(g2, 25, 45);
}
public void start() {
  go = true;
  new Thread(new Runnable() {
    public void run() {
      while(go) {
        try { Thread.sleep(50); } catch(Exception ex) { }
        double dt = System.currentTimeMillis()*0.001 - time;
        for(int i = 0; i < count; ++i) a[i].addTime(dt);
        time += dt; repaint();
      }
    }
  }).start();
```

```
  }
  public void stop() { go = false; }
  public void update(Graphics g) {
    int w = getSize().width, h = getSize().height;
    if(offimage == null) {
      offimage = createImage(w, h); offgraphics = offimage.getGraphics();
    }
    offgraphics.setColor(getBackground()); offgraphics.fillRect(0, 0, w, h);
    paint(offgraphics); g.drawImage(offimage, 0, 0, this);
  }
  public void paint(Graphics g) {
    for(int i = 0; i < count; ++i) a[i].draw(g);
  }

  interface Animation {
    public void draw(Graphics g);
    public void addTime(double dt);
    public void moveTo(double x, double y);
    public double getX();
    public double getY();
    public void setColor(Color c);
    public Color getColor();
  }
```

さて、「図形」という抽象クラスを用意し、ここに図形の基本的な機能を集める。このクラスでは Animation インタフェースにあるメソッド draw() を定義していないので、このメソッドは抽象メソッドになる。そのため、クラスそのものも abstract とつける必要がある。

```
  static abstract class Figure implements Animation {
    Color cl;
    double gx, gy;
    public Figure(Color c, double x, double y) { cl = c; gx = x; gy = y; }
    public void addTime(double dt) { }
    public void moveTo(double x, double y) { gx = x; gy = y; }
    public double getX() { return gx; }
    public double getY() { return gy; }
    public void setColor(Color c) { cl = c; }
    public Color getColor() { return cl; }
  }
```

円や三角形はこの Figure クラスから共通部分を継承して来て、独自の部分だけを定義するので簡単になる。

```
  static class Circle extends Figure {
    double rad;
    public Circle(Color c, double x, double y, double r) {
      super(c, x, y); rad = r;
    }
    public void draw(Graphics g) {
      g.setColor(cl);
      g.fillOval((int)(gx-rad),(int)(gy-rad),(int)(rad*2),(int)(rad*2));
```

```
    }
  }

  static class Triangle extends Figure {
    double dx0, dy0, dx1, dy1, dx2, dy2;
    public Triangle(Color c, double x0, double y0, double x1, double y1,
                    double x2, double y2) {
      super(c, (x0+x1+x2)/3, (y0+y1+y2)/3);
      dx0 = x0-gx; dx1 = x1-gx; dx2 = x2-gx;
      dy0 = y0-gy; dy1 = y1-gy; dy2 = y2-gy;
    }
    public void draw(Graphics g) {
      int[] x = new int[]{(int)(gx+dx0),(int)(gx+dx1),(int)(gx+dx2)};
      int[] y = new int[]{(int)(gy+dy0),(int)(gy+dy1),(int)(gy+dy2)};
      g.setColor(cl); g.fillPolygon(x, y, 3);
    }
  }
```

同様に、1 つ図形を中に持ち、色々な動き等をつけるための抽象クラス Container を用意した。

```
  static abstract class Container extends Figure {
    Animation anim;
    public Container(Animation a) {
      super(a.getColor(), a.getX(), a.getY()); anim = a;
    }
    public void draw(Graphics g) { anim.draw(g); }
    public void addTime(double dt) { anim.addTime(dt); }
    public void setColor(Color c) { anim.setColor(c); }
    public Color getColor() { return anim.getColor(); }
  }
```

Container を土台にして、違う部分だけを書くことで飛ぶ動き、円周軌道の動き、色の変化とも短く書ける。

```
  static class FlyingMove extends Container {
    double vx, vy;
    public FlyingMove(Animation a, double vx1, double vy1) {
      super(a); vx = vx1; vy = vy1;
    }
    public void addTime(double dt) {
      super.addTime(dt); gx += vx * dt; gy += vy * dt;
      if (gx<0.0 && vx < 0.0) { vx = -vx;}
      if (gx>400.0 && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>300.0 && vy > 0.0) { vy = -vy;}
      anim.moveTo(gx, gy);
    }
  }

  static class CircleMove extends Container {
    double rad, vtheta, theta = 0;
```

```
    public CircleMove(Animation a, double r, double vt) {
      super(a); rad = r; vtheta = vt;
    }
    public void draw(Graphics g) {
      anim.moveTo(gx+rad*Math.cos(theta), gy+rad*Math.sin(theta));
      anim.draw(g);
    }
    public void addTime(double dt) { super.addTime(dt); theta += vtheta * dt; }
  }

  static class ChgColor extends Container {
    Color[] colors;
    double time = 0, period;
    public ChgColor(Animation a, Color[] c, double p) {
      super(a); colors = c; period = p;
      if(c.length == 0) throw new RuntimeException("empty color array");
    }
    public void addTime(double dt) {
      super.addTime(dt); time += dt;
      anim.setColor(colors[(int)(time/period) % colors.length]);
    }
    public void moveTo(double x, double y) { anim.moveTo(x, y); }
    public double getX() { return anim.getX(); }
    public double getY() { return anim.getY(); }
    public void setColor(Color c) { }
  }
```

複合図形については中に沢山の図形が入るので、Container ではなく Figure が親クラスとなっている。

```
  static class AnimGroup extends Figure {
    Animation[] a = new Animation[20];
    int count = 0;
    public AnimGroup(double x, double y) { super(Color.black, x, y); }
    public void add(Animation anim) {
      if(count+1 < a.length) a[count++] = anim;
    }
    public void draw(Graphics g) {
      for(int i = 0; i < count; ++i) {
        double x = a[i].getX(), y = a[i].getY();
        a[i].moveTo(x+gx, y+gy); a[i].draw(g); a[i].moveTo(x, y);
      }
    }
    public void addTime(double dt) {
      for(int i = 0; i < count; ++i) a[i].addTime(dt);
    }
  }
}
```

このように、継承とコンポジションをうまく組み合わせて、冗長性が小さく、組み合わせて使いやすいクラス群を作れると「美しい」と思うがいかがだろうか? ここから先は皆様も色々悩んでみていただきたい。

演習 3 上のプログラムは/home/kuno/work/R10Sample4.java に入っているのでコピーしてきてそのまま動かせ。動いたら機能や図形の組み合わせを変えて別の動きをする絵を作ってみよ。もし余裕があれば、新しい機能を追加するとなおよい。演習 2 と演習 3 とどちらがやりやすいか考えてみよう。