情報科学 2007　久野クラス　＃14

久野　靖*

2008.1.29

はじめに

半年間に渡ってお楽しみ頂いた「情報科学」も今回で最終回ということで、最後は次の話題を取り上げます。

- スレッドとアニメーション
- 入力イベントの扱い
- オブジェクト指向グラフィクスとオブジェクト指向再考

今日はゲームを作っている余裕はないでしょうけれど、今回までにやったことを使えばゲームでも何でも作れるようになっているはずです。それは試験とか終わってからゆっくりお楽しみください。

1　スレッドとアニメーション

1.1　スレッドと並行性

ここまでに扱って来たプログラムは (Ruby でも Java でも)、「現在実行中の箇所」は 1 点だけである。たとえば、あるメソッドを呼んだとすると、実行はそのメソッドの中に移るが、それが終わったら呼んだ次の箇所に「戻って来て」続きを実行していく。つまり実行の流れは「一筆描き」になっているわけだ。このような実行の流れのことを「糸」にたとえて**スレッド** (thread) と呼ぶ。そして、これまでのプログラムでは実行の流れは「プログラム実行開始時に作り出されたもの」1 つだけだった。つまり単一スレッドで実行していたわけだ (図 1 左)。

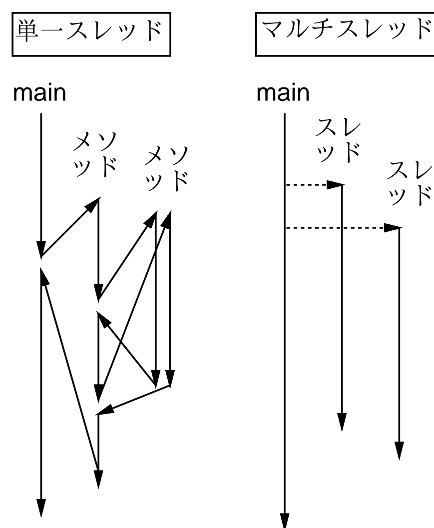

図 1: スレッドの概念

*筑波大学大学院経営システム科学専攻

これに対して、実行の流れが複数あるようなプログラムを**マルチスレッド**と呼ぶ (図 1 右)。マルチスレッドのプログラムは、ある作業をやりながら別の作業を**並行して**行えるという特徴がある。CPU が複数あるマシンであればこれによって計算を**物理的に並列**にして、短時間で終わらせることもできる。しかし CPU が 1 つだけでも、スレッドが役に立つ場合がある。

その具体例としてアニメーションを考えよう。アニメーションとは一定のペースで絵が変化していくものと考えると、次のようなコードを作りたくなる。

```
while(true) {
   0.1 秒待つ
   0.1 秒ぶん変化した絵を作り、描く
}
```

しかしこれを単一スレッドのプログラムでやるのは問題がある。というのは、この無限ループを実行し始めてしまうと、プログラムは GUI の操作にも何も応答せず、このコードだけに専念してしまうからである。これを避けるには、上記のループを「別のスレッドで」実行させればよい。実際には CPU は 1 個だが、それを複数のスレッドの間で細かく切り替えてあたかも各スレッドが独自に動いているかのように動作させる仕事は、言語処理系や OS が担当してくれる。このように、スレッドは**論理的に並行**な実行を実現するのに役に立つ。

1.2 Java におけるスレッドとアニメーション

Java ではスレッドは `Thread` というクラスで表される。すなわち、このクラスを `new` で作り出し、メソッド `start()` を呼ぶと並行な実行が開始される。しかし「何を」開始するのだろう? 自分がやらせたいことを並列にやってもらわないと役に立たないわけだが…`Thread` は実行開始させると、それが持っている引数が 0 個で値を返さないメソッド `run()` を並行実行開始させ、それが終わると並行実行を終了させる。そしてそのメソッドの中身は…何もしないようになっている。

つまりここでも、フレームワークが使われる。すなわち、実際には `Thread` クラスそのものではなく、そのサブクラスを作り、そこで `run()` をオーバライドする。その独自の `run()` の中に自分がやって欲しいことを書いておけば、`start()` によってこれが並行実行されるわけである。

では実際に、これを使って Java で簡単なアニメーションを行わせてみよう。今回は GUI 部品は使わないが、図形を描画する領域は前回同様、`JPanel` のサブクラス (無名内部クラス) を用意してその `paint()` をオーバライドする形で用意する (アニメーションの場合こうしないと描画がうまく行かない)。描くのは赤い円が 1 個だが、その XY 座標をインスタンス変数 `xpos`、`ypos` から取るようにしている。これは、これらの値を変化させることで円の位置が変更できるようにしたため (これらの型は `double` なので、`fillOval()` を呼ぶ時に `int` に型変換させているのに注意)。このほか、インスタンス変数として `time` と `height` があるが、これらは後で使う。この `JPanel` を中身に追加して `pack()` を呼ぶところまでは前回と同じだが、`JPanel` を変数に入れずに直接 `add()` のところに書いている (この方が短い)。

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

public class Sample41 extends JFrame {
  double time = 0.0, xpos = 100.0, ypos = 100.0, height = 100.0;
  public Sample41() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 400));
    getContentPane().add(new JPanel() {
      public void paint(Graphics g) {
        g.setColor(getBackground()); g.fillRect(0,0,getWidth(),getHeight());
        g.setColor(Color.red); g.fillOval((int)xpos, (int)ypos, 50, 50);
      }
    }); pack();
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(100); } catch(Exception ex) { }
          time += 0.1; ypos = 100 + height*Math.sin(time); repaint();
        }
      }
    }.start();
  }
  public static void main(String[] args) { new Sample41().setVisible(true); }
}
```

さて、その後にスレッドの定義と開始がある。Thread から継承する無名内部クラスを定義し、そのメソッド run() の中にスレッドの動作をを書く。内容は先に説明した通りの無限ループで、100 ミリ秒時間待ちして (ここの記述は説明すると長くなるので省略させてください)、続いて time を 0.1 増やし、ypos の値を time の sin によって計算して変更し、変更し終わったら repaint() で画面を描き直すように指示している。これにより、2 π 秒周期で円の位置が上下に振動するわけだ (図 2)。

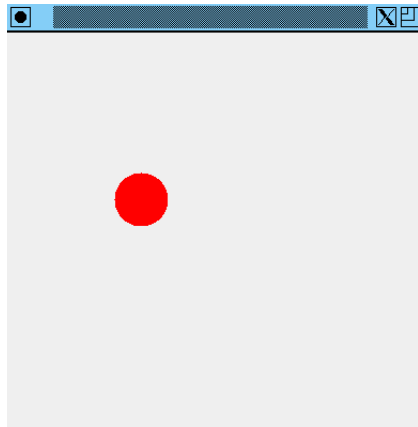

図 2: 上下に振動する円

演習 1 上の例題をそのまま打ち込んで動かせ。動いたら次のように手直ししてみよ。

a. 円を縦方向だけでなく横方向にも動くようにせよ。動き方はどのようなものでもよい。

b. 振動の振幅も時間とともに変化するようにせよ。

c. 時々(2 秒に 1 回くらい?)、円がワープするようにせよ。

d. 円以外のものが動くようにしてみよ。

e. 動く円の数を増やしてみよ。できれば、円だけでない複数種類のものが動くようにしてみよ。

1.3 おまけ: 入力イベントの受け取り

スレッドとは関係ないが、絵がただ動いているだけでは面白くないので、入力を与えて操作してみよう。ここで、GUI 部品は前回やったので、今回は画面上でマウスクリックしたりドラグしたりキーボードを打ったりする、「裸の」イベントを受け取る方法を説明しよう。考え方は前回のボタンベントと同じで、addXXXListener() というメソッド呼び出しを使って「窓に」入力受け取り用のオブジェクトを設定する。これらのオブジェクトは XXXListener というインタフェースを実装していて、その個々のメソッドでイベントを受け取る。

なお、ボタンの時は「押す」だけだからインタフェースにメソッドが 1 つしか無かったが、今度はマウスなら「押す」「離す」「クリック」「窓に入って来る」「窓から出て行く」、マウスドラグ (別のインタフェースに分けてある) では「(マウスボタンを押さずに) 移動する」「(マウスボタンを押して) ドラグする」、キーなら「押す」「離す」「タイプ」のように複数のメソッドがある。これらを以下にまとめる (「窓」と書いてあるが実際には JPanel など窓の中の部分領域にも適用できる)。

```
窓.addMouseListener(MouseListener l)
MouseLister インタフェース   ← MouseAdapter クラス
  mouseClicked(MouseEvent e)   クリック
  mousePressed(MouseEvent e)   ボタン押し
  mouseReleased(MouseEvent e)  ボタン離し
  mouseEntered(MouseEvent e)   マウスカーソルが入る
  mouseExited(MouseEvent e)    マウスカーソルが出る

窓.addMouseMotionListener(MouseMotionListener l)
MouseMotionLister インタフェース   ← MouseMotionAdapter クラス
  mouseMoved(MouseMotionEvent e)    マウス移動
  mouseDragged(MouseMotionEvent e)  ドラグ
```

```
窓.addKeyListener(KeyListener l)
KeyLister インタフェース  ← KeyAdapter クラス
  keyTyped(KeyEvent e)     打鍵
  keyPressed(KeyEvent e)   キー押し
  keyReleased(KeyEvent e)  キー離し
```

そして、これらのインタフェースを実装するクラスを自分で定義する…が、今度はメソッドが一杯あるので、「使わない」ものまで含めて全部実装するのが面倒である (使わなくても定義はしないといけないので)。そこで、たとえば`MouseListener`インタフェースなら`MouseAdapter`というクラスが予め提供されていて、これらのメソッドを全部「何もしない動作」として用意しているので、これを継承したクラスを作って動作をつけたいメソッドだけをオーバライドすればよい。

なんだかごちゃごちゃで面倒そうだが、サンプルを見ればそうでもないと思うはず。これは前の例題を拡張したもので、マウスクリックしたとき円の位置がその位置に移り、「l/s」のキーで振動幅が変えられ、「+/-」のキーで振動周期が増減できるようになっている。

```
import java.awt.*;
import java.awt.event.*;  ←この import を増やすこと!!!
import javax.swing.*;

public class Sample42 extends JFrame {
  double time = 0.0, xpos = 100.0, ypos = 100.0, amp = 10.0, freq = 1.0;
  public Sample42() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 400));
    getContentPane().add(new JPanel() {
      public void paint(Graphics g) {
        g.setColor(getBackground()); g.fillRect(0,0,getWidth(),getHeight());
        g.setColor(Color.red); g.fillOval((int)xpos, (int)ypos, 50, 50);
      }
    }); pack();
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        xpos = evt.getX(); ypos = evt.getY(); repaint();
      }
    });
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        if(evt.getKeyChar() == 'l') { amp *= 1.1; }
        if(evt.getKeyChar() == 's') { amp *= 0.9; }
        if(evt.getKeyChar() == '+') { freq *= 1.1; }
        if(evt.getKeyChar() == '-') { freq *= 0.9; }
      }
    });
    new Thread() {
      public void run() {
        while(true) {
          try { sleep(100); } catch(Exception ex) { }
          time += 0.1*freq; ypos += amp*Math.cos(time); repaint();
        }
      }
    }.start();
  }
  public static void main(String[] args) { new Sample42().setVisible(true); }
}
```

ちょっと長くなったが、増えたのは (0) 使うクラスに応じて`import`を増やした、(1) 調節する事柄が増えたので対応してインスタンス変数を増やした、(2) マウスイベントとキーイベントの処理をつけた、(3) 一定時間毎の円の位置の計算方法を変えた (現在いる位置を起点として計算し、幅や周期を変数によって変えられるようにした) だけである。

演習 2 先の自分のプログラムを変更してマウスまたはキーボード入力によって動作を変えるようにしてみよ。

2　継承とインタフェースによるクラス階層設計

2.1　オブジェクト指向グラフィクスと情報隠蔽・多態

ここで少し長い話になるが、オブジェクト指向に関する重要な「おさらい」をしつつ、絵のプログラムを改良していくことにする。

ここまでで学んだアニメーションの作り方には大きな問題があるが、分かりましたか? それは、時間に応じて変化する変数が外側クラスのインスタンス変数であって、誰でも書き換えられる値になっているということ。このような状態では、コードが大きく複雑になってきたときに、変数が増えてごちゃごちゃになるのは間違いない。さらにまずいのは、どこかでついこれらの変数を間違って書き換えてしまったとき、どこに問題があるのか発見するのが極めて困難になるという点である。

そもそも、画面に描いているのが「動く円」なのに、その円というものを表すオブジェクト (クラス) がプログラム上にない、ということに疑問を持たなくてはいけない。オブジェクト指向は「もの」を「もの」として扱えることで分かりやすくなるのだから、これではオブジェクト指向を活かしているとは言えないわけである。

そこで、「動くぶ円」を次のように 1 つのクラスとして作って見た (今度はサイン曲線で動くのではなく、窓の隅まで来たらはね返る円になっているが)。

```
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>getWidth() && vx > 0.0) { vx = -vx;}
    if (gy<0.0 && vy < 0.0) { vy = -vy;}
    if (gy>getHeight() && vy > 0.0) { vy = -vy;}
  }
}
```

つまり、色、位置、速度、半径をクラスのインスタンス変数として保持することで、これらの値は外部からは勝手にアクセスできなくなり、このクラスのメソッドだけがアクセスするので (厳密に言えば Java ではちょっと違うが当面そう思っていただく)、これらのメソッドさえ間違っていなければ、これらの変数がおかしくなることはないはずである。

このように、クラス内部のデータを外部から直接アクセスさせないように保護することを**情報隠蔽**ないし**カプセル化**と呼び、プログラムのトラブル (バグ) を出にくくする上で大きな効果がある。

ところで `Animation` というのは? これは、次のインタフェースである。

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

つまり、「画面に描くことができ」「時間を進めることができる」ような「もの」がアニメーション (つまり時とともに動く絵) だ、というふうにここでは設計しているわけである。

こうしておけば、動く円以外にもさまざまな `Animation` オブジェクトを作成し、それを配列 (たとえば a に入れておいて、次のように呼び出すことでまとめて描くことができる。

```
for(int i = 0; i < count; ++i) a[i].draw(g);
```

ここで大事なのは `a[i].draw(g)` で実際に呼び出される `draw()` は、`a[i]` が円であれば円を描く `draw()`、三角であれば三角を描く `draw()` というふうに、実行時に切り替わることである。これを**動的分配**と呼ぶのでしたね。別の言い方をすると、`a[i].draw(g)` という呼び出しは実行時に `a[i]` が何であるかによってさまざまに変化する。これを**多態** (polymorphism、**ポリモルフィズム**) と呼ぶ。

なお、Java では多態は上記のようにインタフェースを用いて利用することもできるし、「あるクラス X の変数には、X の (直接・間接の) サブクラス Y のインスタンスも入れられる」という互換性規則があるので、これを用いて利用することもできる。たとえば、Animal を継承したクラスとして Dog と Cat があるとすると、次のようになるわけである。

```
Animal a = new Dog(...)
...
if ...  a = new Cat(...)
...
a.bark();  ←ワンまたはニャー
```

それで多態の何がいいかというと、ある時点で a[i] に入っているものが円なのか三角なのかは気にしなくても、「何らかのものを描く」というふうに抽象的に考えることができ、従って人間が気にしなければならないことがらが減る (その分の注意をプログラムの他の部分に振り向けられる) ようになることである。多態が使える言語の普及以前は、次のようなコードを長々と書くのが当り前だった。

```
if a[i] が円
  円を描く (a[i])
elsif a[i] が三角
  三角を描く (a[i])
elsif a[i] が直線
  直線を描く (a[i])
... (延々とつづく) ...
```

このような長いコードが 1 行で済むため、プログラマが気を使う量も相応に減らせるわけである。

2.2 継承による差分プログラミング

さて、皆様もだいぶ大きなプログラムを書くように (おおむね :-) なったと思うので、プログラムが大きくなった時そのクラスの集まり方をどういう構造にするか、という問題を取り上げておこう。1 つの考え方として、**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と、コンストラクタの冒頭で JPanel を継承した領域を用意。この中では上記の方法でアニメーションの絵をまとめて描いている。

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

public class Sample43 extends JFrame {
  double time;
  Animation[] a = new Animation[20];
  int count = 0;

  public Sample43() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300));
    getContentPane().add(new JPanel() {
      public void paint(Graphics g) {
        for(int i = 0; i < count; ++i) a[i].draw(g);
      }
    }); pack();
```

コンストラクタでの主な初期化は配列 a にさまざまなオブジェクトを入れること。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 void init() {
    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);
  }
```

時間を進めるスレッドは、50 ミリ秒待っては「さっきからこれだけ時間が進んだよ」ということを各アニメーションオブジェクトに addTime() を呼び出して通知し、それが終わったら画面を再描画する。

```
    time = 0.001 * System.currentTimeMillis();
    new Thread(new Runnable() {
      public void run() {
        while(true) {
          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 static void main(String[] args) { new Sample43().setVisible(true); }
```

main() はいつもと同じ。インタフェース Animation は上で述べた通り。

```
  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。上で見た「飛ぶ円」。なお、このクラスで implements Animation を指定していと、このクラスの子クラス群もすべて Animation インタフェースに従うことになる (つまりインタフェースの implements 関係も継承される) ことに注意。

```
  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>getWidth() && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>getHeight() && vy > 0.0) { vy = -vy;}
    }
  }
```

ここから差分プログラミングになる。クラスの継承関係をぱっと見て読み取るのが難しいので、図を描いてみた (図 3)。

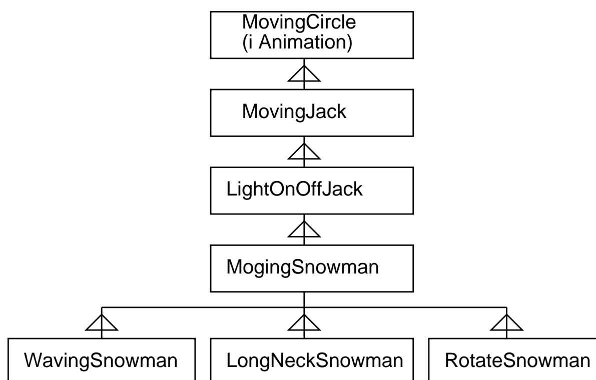

図 3: 差分プログラミングの例のクラス図

ではまず、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 draw() 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、draw() の中で「super.draw(...)」により親クラスの draw() 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。

```
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。

```
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)。

```
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。

```
  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。

```
  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?

```
  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 3 上のプログラムもコピーできるように用意したので、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。

2.3 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**と呼ぶのでしたね。

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 javax.swing.*;

public class Sample44 extends JFrame {
  double time;
  Animation[] a = new Animation[20];
```

```
  int count = 0;

  public Sample44() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300));
    getContentPane().add(new JPanel() {
      public void paint(Graphics g) {
        for(int i = 0; i < count; ++i) a[i].draw(g);
      }
    }); pack();
```

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。

```
    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。

```
      time = 0.001 * System.currentTimeMillis();
      new Thread(new Runnable() {
        public void run() {
          while(true) {
            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();
    }
```

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。

```
  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。

```
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>getWidth() && vx > 0.0) { vx = -vx;}
    if (gy<0.0 && vy < 0.0) { vy = -vy;}
    if (gy>getHeight() && 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。draw() のところで不思議なことをしているが、これは部品の位置は「グループの中心位置原点とする相対座標」を入れてあるのに対し、描く瞬間だけは画面座標にしておかないと正しく描けないので、一時的に画面座標を設定し、描き終わったら元に戻すというワザを駆使しているわけである。

```
  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。今度はクラス図を描いてみても継承関係がないのが分かる (図 4)。

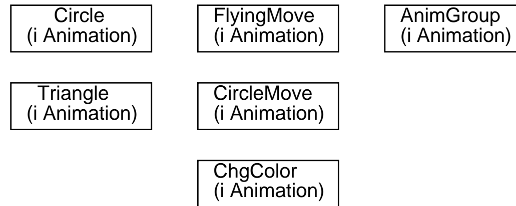

図 4: コンポジションの例のクラス図

演習 4 上のプログラムもコピーできるように用意したので、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。

2.4 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、インタフェースでのメソッド定義と同様、本体 (コード部分) を「`;`」だけにして、さらにキーワード `abstract` をそのメソッドおよびクラス定義の冒頭の両方に指定する。また、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。[1]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 javax.swing.*;

public class Sample45 extends JFrame {
  double time;
  Animation[] a = new Animation[20];
  int count = 0;

  public Sample45() {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setPreferredSize(new Dimension(400, 300));
    getContentPane().add(new JPanel() {
      public void paint(Graphics g) {
        for(int i = 0; i < count; ++i) a[i].draw(g);
      }
    }); pack();
    a[count++] = new FlyingMove(new Circle(Color.red, 100, 100, 15), -30, 40);
```

[1] 時々「(クラス名) は abstract として宣言する必要があります。(メソッド指定) を (クラス名) で定義していません。」というコンパイルエラーで悩んでいる人がいるが、これはまさに前記の条件に関係するもので、インタフェースを implements したのに (スペルミスなどで) そのインタフェースが定義しているメソッドを定義し損なっている場合に出る。

```
    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);
    time = 0.001 * System.currentTimeMillis();
    new Thread(new Runnable() {
      public void run() {
        while(true) {
          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 static void main(String[] args) { new Sample45().setVisible(true); }
  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; }
  }
```

ここでクラス図を示しておこう (図 5)。
　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);
```

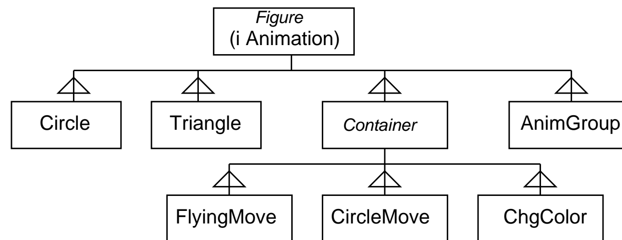

図 5: 抽象クラス+コンポジションの例のクラス図

```
      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>getWidth() && vx > 0.0) { vx = -vx;}
      if (gy<0.0 && vy < 0.0) { vy = -vy;}
      if (gy>getHeight() && 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 4' 上のプログラムもコピーできるように用意したので、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 4 と演習 4' とどちらがやりやすいか考えてみよう。

3 例題: GUI 部品とアニメーションの共存

では皆様の「総合制作」(課題ではなく、今後趣味でやってくださいというつもり) の参考になるような最後の例題として、前回やった GUI 部品と今回やったオブジェクト指向グラフィクスによるアニメーションを組み合わせた例題を示そう。画面構成は前回と同じく図形を表示する中央部分と、下側の GUI 部品の領域に分かれている。GUI 部品としては押しボタンを 3 つ、テキスト入力欄を 1 つ作成する。

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

  public class Sample46 extends JFrame {
    Animation hit = null;
    Animation[] a = new Animation[20];
```

```
int count = 0;
JPanel panel = new JPanel() {
  public void paint(Graphics g) {
    g.setColor(getBackground()); g.fillRect(0, 0, getWidth(), getHeight());
    for(int i = 0; i < count; ++i) { a[i].draw(g); }
  }
};
JButton b1 = new JButton("Fast");
JButton b2 = new JButton("Slow");
JButton b3 = new JButton("Text");
JTextField t1 = new JTextField();
JPanel lowpanel = new JPanel();
```

さて、コンストラクタによる初期化であるが、まず窓の中には「中心に」panel、「南 (下端) に」lowpanel を入れる。lowpanel のレイアウトマネージャは FlowLayout に変更する。その後、lowpanel の中に部品を 1 つずつ詰めていくが、ボタンについてはそのボタンが押された時の動作を指定したイベントハンドラを設定していく。

```
public Sample46() {
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  getContentPane().add(panel, BorderLayout.CENTER);
  getContentPane().add(lowpanel, BorderLayout.SOUTH);
  lowpanel.setLayout(new FlowLayout());
  lowpanel.add(b1);
  b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(hit != null) { hit.changeSpeed(1.1); }
    }
  });
  lowpanel.add(b2);
  b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(hit != null) { hit.changeSpeed(0.9); }
    }
  });
  lowpanel.add(b3);
  b3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String s = t1.getText();
      if(count+1>=a.length || s.equals("")) { return; }
      a[count++] = new WavingText(
        Color.getHSBColor((float)Math.random(), 1f, 1f), s,
        400*Math.random(), 200*Math.random(), 100*Math.random(), 1);
    }
  });
```

最初の 2 つのボタンは現在選ばれている図形の速度を変更する (このためにインタフェース Animation に changeSpeed() の定義を追加し、各図形クラスでもメソッドを増やしている)。3 番目のボタンは入力欄に打ち込まれた文字列を持った WavingText を生成して追加する。JTextField については、イベントハンドラは必要ないが、どれくらいの大きさにするかを setPreferredSize() で指定する必要がある (何文字くらい打ち込むのに使うかで適正な大きさが違ってくるから)。それ以後はこれまでの例題と同様である。

```
  lowpanel.add(t1); t1.setPreferredSize(new Dimension(60, 25));
  pack(); setSize(400, 400);
  panel.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) { press(e.getX(), e.getY()); }
  });
  a[count++] = new FlyingCircle(Color.red, 100, 100, 40, 25, 40, panel);
  a[count++] = new FlyingCircle(Color.blue, 100, 100, 30, 60, -90, panel);
  a[count++] = new WavingText(Color.green, "Hello", 80, 200, 100, 0.5);
  new Thread() {
    public void run() {
      while(true) {
        try { sleep(20); } catch(Exception ex) { }
        for(int i = 0; i < count; ++i) { a[i].addTime(0.02); }
        panel.repaint();
      }
    }
  }.start();
```

```
    }
    public static void main(String[] args) {new Sample46().setVisible(true);}
```

press() では一番後ろ (上) から順に図形を調べて、マウスの位置に当たる図形があればそれが一番上になるように配列の格納順序を入れ替える。このためにインタフェース Animation にメソッド hit() を増やしている。1 つもなければ、代わりに新しい FlyingCircle オブジェクトを乱数パラメタで作り、それを最後に入れる。hit というインスタンス変数は、当たったオブジェクトまたは新しく生成したオブジェクトを記憶する

```
    public void press(int x, int y) {
      hit = null;
      for(int i = count-1; i >= 0; --i) {
        if(a[i].hit(x, y)) {
          hit = a[i];
          for(int j = i; j < count-1; ++j) { a[j] = a[j+1]; }
          a[count-1] = hit; return;
        }
      }
      if(count+1 >= a.length) { return; }
      hit = a[count++] = new FlyingCircle(
        Color.getHSBColor((float)Math.random(), 1f, 1f), x, y,
        10+20*Math.random(), 10+40*Math.random(), 10+40*Math.random(), panel);
    }
```

インタフェース Animation と 2 つの図形はこれまでにやったようなものの簡略版だが、上述のとおりメソッド changeSpeed() と hit() が増えている。

```
    interface Animation {
      public void draw(Graphics g);
      public void addTime(double dt);
      public boolean hit(double x, double y);
      public void changeSpeed(double r);
    }
    static class WavingText implements Animation {
      static Font fn = new Font("Helvetica", Font.BOLD, 20);
      Color col; String text;
      double xpos, ypos, yrad, theta = 0.0, vtheta;
      public WavingText(Color c, String s,
                        double x, double y, double r, double v) {
        col = c; text = s; xpos = x; ypos = y; yrad = r; vtheta = v;
      }
      public void draw(Graphics g) {
        int x = (int)xpos, y = (int)(ypos + yrad*Math.sin(theta));
        g.setColor(col); g.setFont(fn); g.drawString(text, x, y);
        g.setColor(Color.black);
      }
      public void addTime(double dt) { theta += vtheta*dt; }
      public boolean hit(double x, double y) {
        int xp = (int)xpos, yp = (int)(ypos + yrad*Math.sin(theta));
        return xp < x && x < xp+15*text.length() && yp-20 < y && y < yp;
      }
      public void changeSpeed(double r) { vtheta *= r; }
    }
    static class FlyingCircle implements Animation {
      Color col; double xpos, ypos, rad, vx, vy; JPanel panel;
      public FlyingCircle(Color c, double x, double y, double r,
                          double vx1, double vy1, JPanel p) {
        col = c; xpos = x; ypos = y; rad = r; vx = vx1; vy = vy1; panel = p;
      }
      public void draw(Graphics g) {
        g.setColor(col);
        g.fillOval((int)(xpos-rad), (int)(ypos-rad), (int)rad*2, (int)rad*2);
      }
      public void addTime(double dt) {
        xpos += vx * dt; ypos += vy * dt;
        if(xpos<0 && vx<0 || xpos>panel.getWidth() && vx>0) { vx = -vx; }
        if(ypos<0 && vy<0 || ypos>panel.getHeight() && vy>0) { vy = -vy; }
      }
      public boolean hit(double x, double y) {
```

```
      return (xpos-x)*(xpos-x) + (ypos-y)*(ypos-y) <= rad*rad;
    }
    public void changeSpeed(double r) { vx *= r; vy *= r; }
  }
}
```

どうですか、かなり読めるでしょう?

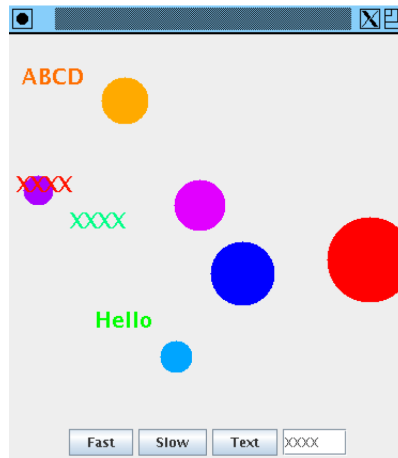

図 6: GUI 部品とアニメーションを持った例題

演習 5 この例題 (コピーできるようにしました) をそのまま動かせ。動いたら好きなように改造してみよ。

4 さいごに

これで「情報科学」久野クラスの講義はすべておしまいです。あとは試験がんばってください。最後にひとこと。皆様は理 2・3 類なのでソフトウェア系の専門に進まれるかたはほとんどいないと思いますが、たとえそうだとしても、情報科学の基本部分を見聞し、また実際にプログラミングができるようになったことには多くの価値があると思います。たとえば、できあいのソフトではやりにくいことでも、プログラムを書けばすぐにできたり楽に仕事ができることは多数あります。皆様が今後ともそういう形で、情報科学やプログラミングの知識を活用して行って頂けるようになれば、私としては大変幸せです。では半年間、ありがとうございました。

A 本日の課題 13A

プログラム作成を内容とする演習問題から 1 つ選び、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 13A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

Q1. アニメーションや入力イベントの扱い方が分かりましたか?

Q2. オブジェクト指向におけるクラスの構成方法についていろいろ工夫すべき点があることに納得しましたか?

Q3. 最後ですので全体を通じてのご意見・ご感想を。

次回までの課題はありません。