オブジェクト指向プログラミング'10 ＃4

久野　靖*

2010.9.28

今回は前回予告した通り、アニメーションを取り上げます。また、それに関連して、クラスの拡張、コンポジションの考え方、クラス構造の問題について取り上げます。

1　アニメーションの原理

アニメーション (動画) の原理は、「人間に少しずつ違った絵を短い時間間隔で次々に見せると動いて見える」ことです。昔は人間が少しずつ違った絵を長い時間掛けて手で描いていましたが (図 1 上)、コンピュータは高速なので、「短い時間間隔」ごとにその場で「少しずつ違った絵」を生成して表示することができます (図 1 下)。これを**実時間アニメーション**と呼びます。

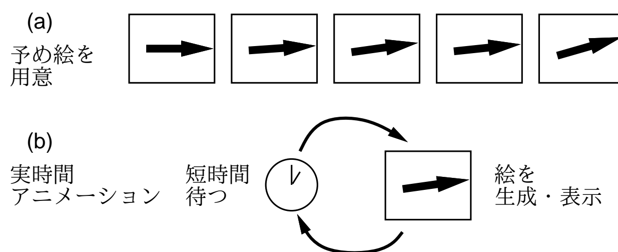


図 1: アニメーションの原理

ただし、実際に図 1 下のようにプログラムの実行を「待つ」のではなく、一定時間間隔で行う処理を登録すれば、あとはシステム側がその処理を繰り返し実行してくれます。具体的には、`javax.swing.Timer` オブジェクトを、時間間隔 (ミリ秒単位) とその間隔で実行させる「動作」を渡して生成し、そのオブジェクトのメソッド `start()` を呼び出すと、実行が開始されます。[1)]

```
new javax.swing.Timer(30, 動作).start(); // 30 ミリ秒間隔
```

それはいいのですが、「動作」はどうやって指定するのでしょうか? 実は動作として渡すオブジェクトは **AcitonListener** インタフェースを実装するオブジェクトということになっていて、このインタフェースが定めているメソッド `actionPerformed()` [2)]が動作として呼び出される、というふうになっています。つまりここでも、インタフェースの多相性を活用してさまざまな「動作」の実現を渡すことができるようになっているわけです。

1)なぜ `javax.swing.Timer` というふうに長く書いているかというと、`java.util.Timer` というクラスもあるので、単に `Timer` ではどちらのクラスか曖昧でコンパイルエラーになるためです。

2)引数として `ActionEvent` オブジェクトを 1 個受け取ります。

*経営システム科学専攻

そこで、以下では次のように、ActionListener を実装する無名内部クラスのインスタンスを作り、その中で actionPerformed() を定義することで動作の中身を記述します。

```
new javax.swing.Timer(30, new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    // 動作内容…
  }
}).start();
```

例題 6-1: 動く円

では例題として、円が時間とともに画面を右に動いていき、1 秒たつと最初の位置に戻る、なおかつ経過秒数が左上に表示される、というプログラムを作ってみましょう (図 2)。

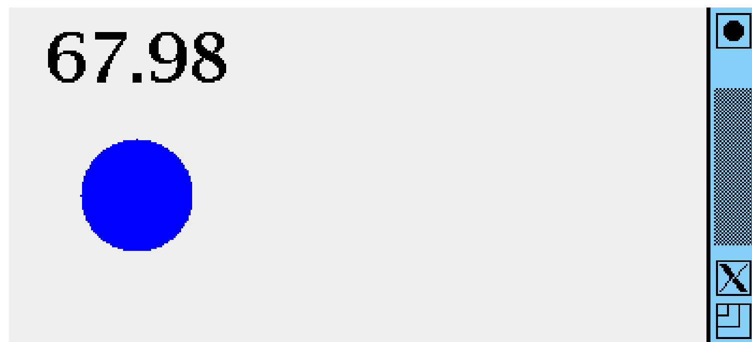


図 2: 動く円

このプログラムでは秒数を小数点以下 2 桁まで表示するために、String のクラスメソッド **format**() を使っているので、予め説明しましょう。

このメソッドは、第 1 引数の文字列の中に第 2 引数以下の内容を、それぞれ変換指定に応じて文字列に変換して埋め込むという機能を提供します。変換指定は「%」で始まり、次のような指定が行えます。3)

3)実際にはもっと沢山ありますが、全部説明すると大変なので主要なもの少しだけを説明しています。

- %wd — 整数を幅 w 文字に揃えて埋め込む。
- %wx — 整数を幅 w 文字に揃えて 16 進表現で埋め込む。
- %w.lf — 実数値を幅 w 文字に揃え、その中で小数点以下を l 桁取るようにして埋め込む。

表示に必要な最小の幅よりも幅 w の方が大きければ、幅が w になるように空白文字が追加されます。幅 w で足りない場合は必要なだけの幅が取られます。w を指定しなければ常に必要最小の幅が取られます。

では、プログラムの本体部分を見てみましょう。4)5)

4)表示のしかたやインタフェース Figure、Circle、Text などは前章までと同様です。

5)ActionListener インタフェースは java.awt.event パッケージに含まれているので、import が追加されています。

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

```
public class Sample61 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  Circle c1 = new Circle(Color.BLUE, 100, 100, 30);
  Text t1 = new Text(20, 40, "?",
                     new Font("Serif", Font.BOLD, 36));

  public Sample61() {
    setOpaque(false); figs.add(t1); figs.add(c1);
    final long tm0 = System.currentTimeMillis();
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double tm = 0.001*(System.currentTimeMillis()-tm0);
        t1.setText(String.format("%4.2f", tm));
        c1.moveTo(20 + (int)(tm*100) % 250, 100); repaint();
      }
    }).start();
  }
  public void paintComponent(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample61());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
```

ほとんどの仕事はコンストラクタで行っています。まず、setOpaque() 呼び出しと図形の登録があり、次に経過時間を計るための変数 tm0 に System.currentTimeMillis() の返り値を入れます。[6] 変数定義の先頭に **final** と指定がありますが、これにより、この変数は初期化以後は書き換えられなくなります。これは「無名内部クラス内のコードから外側のローカル変数を参照するときは、その変数が final 指定でなければならない」という制約があるためにそうしています。その後は上記の書き方で反復動作を指定していますが、その中身は次の通りです。

6) このメソッドは 1970 年 1 月 1 日午前 0 時からの経過ミリ秒数を long の値として返します。

1. 再び System.currentTimeMillis() を呼んで現在時刻を取得し、tm0 の値を引くことで実行開始時からのミリ秒数を計算し、それに 0.001 を掛けて秒数に換算します。その値を String.format() を使って幅 4 文字、小数点以下 2 桁の文字列に変換し、Text オブジェクトの文字列として設定します。
2. Circle オブジェクトの位置を X 座標は「経過秒数を 100 倍して整数にした後 250 で割った余り+20」、[7] Y 座標は 100 に設定します。
3. 最後に repaint() というメソッドを呼ぶと、適当なタイミングで画面が更新されます。この時、Text の内容や Circle の位置が変更されて

7)「%は「割った余り」を求める演算子です。これにより、時間とともに X 座標は増えて行きますが、2.5 秒たつと 100 倍した値が 250 になるので割った余りとしては 0 に、つまり最初の位置に戻ってまた増えていく、という繰り返しになります。

いるので、その変更された内容や位置に対応した画面が描かれます。

残りのインタフェースとクラスはこれまでと同様ですが、円は色を変化させたいことがあるので、色も設定できるようにしました。

```
  interface Figure {
    public void draw(Graphics g);
  }
  static class Circle implements Figure {
    Color col;
    int xpos, ypos, rad;
    public Circle(Color c, int x, int y, int r) {
      col = c; xpos = x; ypos = y; rad = r;
    }
    public boolean hit(int x, int y) {
      return (xpos-x)*(xpos-x)+(ypos-y)*(ypos-y)<=rad*rad;
    }
    public void setColor(Color c) { col = c; }
    public void moveTo(int x, int y) { xpos = x; ypos = y; }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
  static class Text implements Figure {
    int xpos, ypos;
    String txt;
    Font fn;
    public Text(int x, int y, String t, Font f) {
      xpos = x; ypos = y; txt = t; fn = f;
    }
    public void setText(String t) { txt = t; }
    public void draw(Graphics g) {
      g.setColor(Color.BLACK); g.setFont(fn);
      g.drawString(txt, xpos, ypos);
    }
  }
} // 外側クラスの終わりの「}」
```

演習 6-1 例題Sample51.javaをそのまま打ち込んで動かしなさい。動いたら次のように変更してみなさい。

a. 円の横位置だけでなく、縦位置も動くようにしてみなさい。動き方は好きに選んで構いません。

b. 円を複数にして、それぞれ違う動き方をさせてみなさい。

c. 円の色が時間とともに連続的に変化するようにしてみなさい。

d. 円の色が 1 秒単位で変化するようにしてみなさい。色は 2 色交互でもよいですが、3 色以上だとなおよいです。

e. その他、好きな図形を好きなように動かしてみなさい。

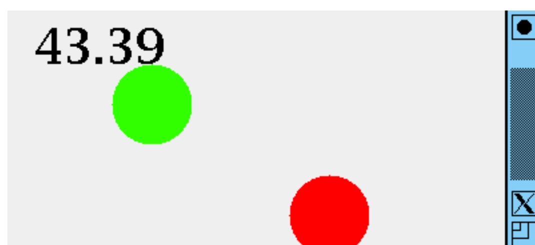
例解 6-1-abcd

文字のほかに円を 2 つ描き、円 1 は水平方向には例題 6-1 と同じ往復運動、縦方向には乱数をつかって「ふらふらと」動かします。円 2 は円 2 は楕円運動させます。そして円 1 は時間とともに連続的に色が変化し、円 2 は 3 つの色を 3 秒ずつ交替で切り替えます。[8][9][10]

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

public class ex61abcd extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  Circle c1 = new Circle(Color.BLUE, 100, 100, 30);
  Circle c2 = new Circle(Color.RED, 160, 100, 30);
  int ypos = 100;
  Text t1 = new Text(20, 40, "?", new Font("Serif", Font.BOLD, 36));

  public ex61abcd() {
    setOpaque(false); figs.add(t1); figs.add(c1); figs.add(c2);
    final long tm0 = System.currentTimeMillis();
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double tm = 0.001*(System.currentTimeMillis()-tm0);
        t1.setText(String.format("%4.2f", tm));
        ypos += (int)(40*Math.random() - 20);
        c1.moveTo(20 + (int)(tm*100) % 250, ypos);
        c2.moveTo(200 + (int)(50*Math.sin(tm)),
                  80 + (int)(80*Math.sin(tm + 1.0)));
        c1.setColor(Color.getHSBColor((float)((0.03*tm)%1), 1f, 1f));
        c2.setColor(new Color[]{Color.PINK, Color.RED, Color.BLUE}
                    [(int)(0.3*tm) % 3]);
        repaint();
      }
    }).start();
  }
  // paintComponent 以下は Sample61 と同じ
```



8) 円 c1 の Y 座標は変数 ypos を追加してここに保持します。初期値は 100 で、1 フレームごとに-20〜20 の一様乱数を生成してその値を足し込むことでランダムに上下させます。

9) 円 c2 の XY 座標は、秒数を元に *sin* 関数で計算し、2 π (約 6.28) 秒ごとに元の値に戻るようにしています (X と Y で少し位相をずらして楕円になるようにしています。

10) 色ですが、c1 については Color.GetHSBColor() を使って色相を連続的に変化させ、c2 については秒数を 0.3 倍して整数にするので、約 3.3 秒ごとに色が切り替わります。色は 3 つの色から成る配列を作り、その添字として 0、1、2 を指定して取り出すので、0.3 倍して整数にした値をさらに 3 で割った余りを取って添字に使います。

2 アクションゲーム

では、図形が動かせるようになったところで、これと入力を組み合わせることで、簡単なゲームを作って見ましょう。ゲームといっても色々なものが

ありますが、ここではいわゆる「反射神経」のよしあしを競うようなゲーム (アクションゲーム) を取り上げます。

例題 6-2: 反応時間ゲーム

反応時間をそのまま競うゲームとして、上で挙げた「青い円が赤くなったらすぐクリック」の時間を計測するプログラムを作ってみましょう (図 3)。開始時刻、「色が変わる」時刻、実際に変わった時刻を 3 つのインスタンス変数に保持するようにしました。

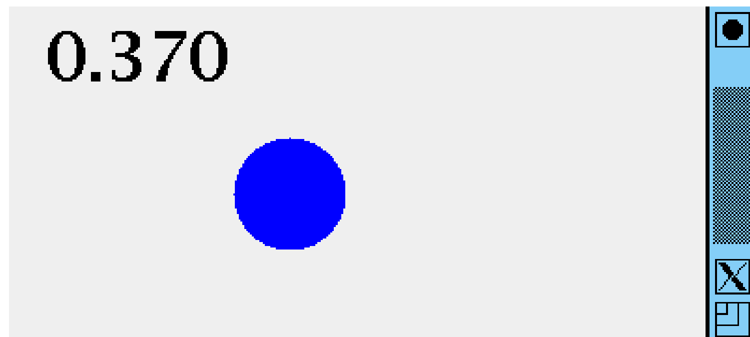


図 3: 反応時間ゲーム

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

public class Sample62 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  Circle c1 = new Circle(Color.BLUE, 150, 100, 30);
  Text t1 = new Text(20, 40, "Click whien red",
                     new Font("Serif", Font.BOLD, 36));
  long tm0 = System.currentTimeMillis();
  double tmc = 3.0 + 3.0*Math.random();
  double tm1 = 0.0;

  public Sample62() {
    setOpaque(false); figs.add(c1); figs.add(t1);
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double tm = 0.001*(System.currentTimeMillis()-tm0);
        if(tmc != 0.0 && tm >= tmc) {
          tm1 = tm; tmc = 0.0; c1.setColor(Color.RED);
          repaint();
        }
      }
```

```
  }).start();
  addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent evt) {
      if(tmc == 0.0 && c1.hit(evt.getX(), evt.getY())) {
        double tm = 0.001*(System.currentTimeMillis()-tm0);
        t1.setText(String.format("%.3f", tm-tm1));
        c1.setColor(Color.BLUE);
        tmc = tm + 3.0 + 3.0*Math.random(); repaint();
      }
    }
  });
}
// paintCompnent 以下は Sampel61 と同じ
```

大部分の処理は反復動作とマウスアダプタの中で行っています。[11][12]

演習 6-2 例題 Sample62.java をそのまま打ち込んで動かしなさい。動いたら次のように変更してみなさい。

a. 円を赤だけでなくランダムにさまざまな色になるようにしてみなさい。そして、「どの色 (とどの色) の時だけクリック」するようにした場合、反応時間はどうなるか計測してみなさい。[13]

b. 円が赤くなる瞬間に、ランダムに画面上の別の位置に「ワープ」するようにしてみなさい。

c. 円が赤くなった瞬間にワープした後、さまざまな動きをすることでクリックしにくくしてみなさい。

d. 円が 5 つ並んでいて、赤くなるのはそのうちの 1 つであるようにしてみなさい。

e. さらに、赤くなった円が 1 秒程度手前に動き、元の位置に戻ると青くなるようにしてみなさい。

f. 反応時間を計測する代わりに、30 秒間でいくつ赤い状態の円をクリックできるかを競うゲーム (もぐらたたき?) にしてみなさい。

g. ここまでに学んだことで作れる、好きなゲームを作ってみなさい。

11)反復動作側では、色が変化すべき時刻 tmc が過ぎたら円の色を赤に変え、その時刻を変数 tm1 に覚えます。tmc は「色の変化は終わった」印として 0 にしています。

12)マウスイベントの処理では、色の変化が済んでいて、かつクリック位置が円の中であれば、記録してあった時刻 tm1 と現在時刻の差を反応時間として (小数点以下 3 桁まで) 表示し、円の色を戻し、次に色が変わるべき時刻を乱数で決定します。

13)複数の色から選ぶなどの条件が増えると、認知プロセサの処理が 1 回では済まなくなるため、τ_c の部分が増え、対応して全体の反応時間も増えるはずです。

例解 6-2-b と Fitzz の法則

6-2-a の変更はとても簡単で、円を赤く変更した直後に例えば次のようなコードによって円の位置をランダムに変更すればよいでしょう。

```
c1.moveTo(20+(int)(360*Math.random()),
          20+(int)(260*Math.random()));
```

例解 6-2-c

この問題はどのような動きにしてもよいのですが、例解として「縦/横方向に一定速度で動き、窓の端まで来たら跳ね返る」ようにしてみました。このため、まず円の現在位置を保持するインスタンス変数 xpos、ypos と 50 ミリ秒あたりの XY 方向それぞれに動く量を保持するインスタンス変数 dx、dy を追加します。

```
int xpos = 150, ypos = 100, dx, dy;
```

あとは動作本体だけを示します。具体的には、動く時間が来たら dx と dy を乱数で選んだ値に設定します。それ以外の場合は「動き続ける」ために、現在位置に dx、dy を加えますが、「左に動いていて窓の左から出た場合」「右に動いていて窓の右から出た場合」は dx、「上に動いていて窓の上から出た場合」「下に動いていて窓の下から出た場合」は dy をそれぞれ反転することで、窓のふちから出て行かないようにします。

```
public void actionPerformed(ActionEvent evt) {
  double tm = 0.001*(System.currentTimeMillis()-tm0);
  if(tmc != 0.0 && tm >= tmc) {
    tm1 = tm; tmc = 0.0; c1.setColor(Color.RED);
    dx = 10 + (int)(30 * Math.random());
    dy = 10 + (int)(20 * Math.random()); repaint();
  } else if(tmc == 0.0) {
    xpos += dx; ypos += dy;
    if(xpos < 0 && dx < 0 ||
       xpos > getWidth() && dx > 0) dx = -dx;
    if(ypos < 0 && dy < 0 ||
       ypos > getHeight() && dy > 0) dy = -dy;
    c1.moveTo(xpos, ypos); repaint();
  }
}
```

例解 6-2-def

これは要するに「5 個の円のモグラ叩き」を作ってみればいいですね。ゲームらしくするため、40 秒間の時間を区切り、その間だけ動くようにします。[15] そして、Text オブジェクトを 2 つ用意し、1 つは残り時間、もう 1 つは「赤い円を叩けた数」を入れるようにします。円は「自分で動く」ように機能を追加するため、これまでの Circle のサブクラス GameCircle として作成します。また、円を 5 個にするため、大きさ 5 の配列を用意し、ここに 5 個の GameCirlce オブジェクトを入れるようにします。変数 count は叩けた数を保持します。コンストラクタの冒頭で配列に円を入れ、これまでと同様のアニメーション処理に入ります。[16]

[15] 実際にはいきなり始まると困るので、最初の 10 秒は円は動きません。

[16] 今回は時間が 40 秒になったら以後は何もせずに戻るようになっています。

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

public class ex62def extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  Font fn = new Font("Serif", Font.BOLD, 36);
  Text t1 = new Text(20, 30, "Click when red", fn);
  Text t2 = new Text(20, 60, "0", fn);
  GameCircle[] a = new GameCircle[5];
  int count;
  public ex62def() {
    setOpaque(false); figs.add(t1); figs.add(t2);
    for(int i = 0; i < 5; ++i) {
      a[i] = new GameCircle(Color.BLUE, 50+50*i, 100, 20);
      figs.add(a[i]);
    }
    final long tm0 = System.currentTimeMillis();
    new javax.swing.Timer(30, new ActionListener() {
      double tmc = 10.0 + 3.0*Math.random();
      double tm = 0.0;
```

```
      public void actionPerformed(ActionEvent evt) {
        if(tm >= 40.0) { return; }
        tm = 0.001*(System.currentTimeMillis()-tm0);
        if(tm >= tmc) {
          tmc = tm + (40.0-tm)*0.1*Math.random();
          int i = (int)(a.length * Math.random());
          if(a[i].getColor() != Color.RED) {
            a[i].setColor(Color.RED); a[i].start();
          }
        }
        for(GameCircle c1: a) { c1.proceed(); }
        t1.setText(String.format("%2.2f", 40.0-tm));
        repaint();
      }
    }).start();
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        for(GameCircle c1: a) {
          if(c1.getColor() == Color.RED &&
             c1.hit(evt.getX(), evt.getY())) {
            c1.setColor(Color.BLUE); ++count;
            t2.setText(String.format("%d", count));
          }
        }
      }
    });
  }
  // paintComponent 以下は GameCircle の追加を除き同じ
```

GameCiecle オブジェクトはそれぞれが、「赤い色になって下に動いて行き、戻って来て青になる」機能を持ちます。このため、待ち時間になったら円をランダムに選び、選んだ円が既に赤くない場合に限り「動きを開始」するメソッド start() を呼びます。その後、各円について時間を進めるメソッド proceed() を呼び、残り時間表示を更新し、repaint() で画面を更新します。

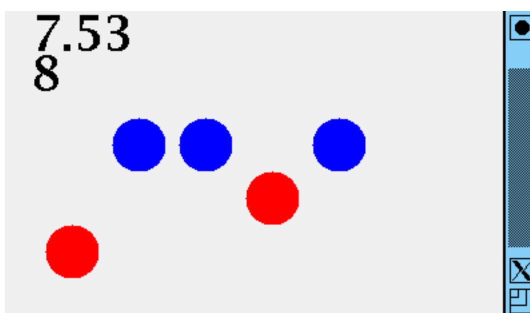
マウスクリックがあった場合は、各円について順に調べて行き、クリック位置にある円が見つかり、なおかつその円が赤いなら、青に変更してカウントを増やします。

最後にクラス GameCircle を示します。下に動くようにするため、基準となる Y 位置を変数 ybase に覚えておくようにし、また「動いて戻る」のは 20 ステップで行うのでそのステップを覚える変数 mcount も追加します。[17] そして、start() が呼ばれると色を赤くするとともに mcount を 20 に設定します。proceed() は mcount が 0 なら (動いていないので) 何もしません。動いている場合はまず mcount を 1 減らし、mcount の値に応じて現在位置を基準の Y 座標からずれた値に設定します。そして、mcount が 0 になったら元の位置に戻ったことになるので、色を青に戻します。

[17] 新たに追加した変数の初期化が必要ならコンストラクタで直接代入しますが、親クラスから継承している変数は super() で親クラスのコンストラクタを呼ぶことで初期化するのでしたね。忘れた場合は 5 章の解説を見てください。

```
  static class GameCircle extends Circle {
    int ybase, mcount;
    public GameCircle(Color c, int x, int y, int r) {
      super(c, x, y, r); ybase = y;
    }
    public Color getColor() { return col; }
    public void start() { mcount = 20; }
    public void proceed() {
      if(mcount <= 0) { return; }
      mcount -= 1;
```

```
        ypos = ybase + 100 - Math.abs(mcount-10) * 10;
        if(mcount == 0) { col = Color.BLUE; }
      }
    }
```



このように、動かし方が複雑になってきた場合は、動作本体側で全部個別に動かすよりも、個々のオブジェクトそれぞれに「自分の動きや変化」を管理させる方がプログラムの見通しがよくなります。このような見通しの良さが、オブジェクト指向の大きな利点なのです。

差分プログラミングとその限界

最後の例解の中で、既存のクラス Cirlce はいじらずに、サブクラスを作って拡張部分を追加したことに注目してください。こうすることで、追加する機能と元の機能がごちゃまぜにならなず、土台のクラスはいじらずに (誤って壊したりせずに)、複数のさまざまな拡張 (サブクラス) からそのまま共有できる、という利点があります。このように、既存クラスに対する「$+\alpha$」つまり差分をサブクラスとして記述することで新しい機能を持つクラスを増やして行く方法を**差分プログラミング**と呼びます (図 4)。

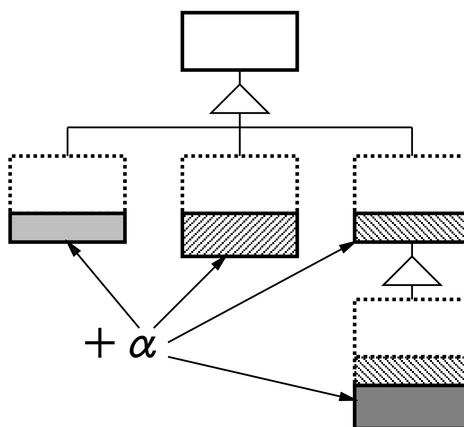


図 4: 差分プログラミング

すばらしいアイデアだ、と思いましたか? 先人たちもそう思ったのですが、これを多用し始めると、互いに強く依存し合った多数のクラスができてしまって理解や修正が難しくなるため、[18] 差分プログラミングは必ずしもよい方法ではない、というのが今日の一般的な認識です。[19]

差分プログラミングのもう 1 つの弱点として、複数の拡張を組み合わせるのが難しいということも挙げられます。たとえば、土台となるクラス A を

18)サブクラスで親クラスの変数を書き換えるコードがあったとすると、親クラスのコードも読まないとその動作が理解できません。たとえば GameCircle の proceed() の中で ypos を変更していますが、これが円を Y 方向に動かしているということは、Circle の中身を知っていないと分かりませんね。そして親クラスの方ではせっかく内部のデータ構造をカプセル化して外部から干渉されなくしたのに、サブクラスからいじられるとその前提が崩れてしまいます。たとえば Circle の側で円の位置を変更する時は必ず moveTo() が呼ばれる、という前提でコードを構成していたとすると、サブクラスで ypos をいきなり変更されたら何かがうまく行かなくなるでしょう。

19)ここの例のように限定された場面で小規模に注意して使うのなら問題はないのですが。

もとに、X という拡張を施したサブクラスと、Y という拡張を施したサブクラスを作ったとします。しかしもしも、X という拡張と Y という拡張を併せ持ったクラスを作りたいとしたら、どうしましょうか? これを素直に行う方法がない、というのが弱点なわけです。[20][21]

では、どうしたらいいのでしょうか? この点については、次章で取り扱います。

20) X と Y を併せ持ったクラスを新たに作るとすると、せっかく分離してあった拡張が一緒くたになりますし、X や Y を修正したらまぜたクラスも手で修正することになりますし、何より N 種類の拡張をさまざまに組み合わせて使うとすると、2^N 個のクラスを作ることになって非現実的です (このように組み合わせの個数が非常に多くなることを**組み合わせ爆発**と呼びます)。

21) 言語によっては、1 つのクラスが複数の親クラスを持てる**多重継承**と呼ばれる機能を持っていて、これを利用して X や Y などの拡張部分を独立したクラスとして作成し、土台のクラスと X や Y など必要な機能のクラスすべてを親として持つようなサブクラスを作ることでさまざまな拡張機能を組み合わせたクラスを得る方法が使えます。この場合、X や Y などに対応するクラスは、「他のクラスと混ぜて機能を追加する」という位置付けの特殊なクラスであり、**mixin クラス**と呼ばれます。mixin クラスは確かに 1 つの解決策ですが、複数の mixin を混ぜて使った時に予期しない干渉が起きないことを保証するのは難しいという弱点があります。そして Java では親クラスは 1 つしか持てないので、この方法は使えません。

3 付録: パラメタつきクラスを定義する

ここまで、パラメタつきのクラス ArrayList を使ってきましたが、このようなクラスを自分で定義することもできます。細かいところを説明しはじめると大変ですが、大まかな考え方だけ説明しておきましょう。たとえば、ArrayList の代わりに自前で同様の機能を提供するクラス MyList<E>を定義してみます。

```
static class MyList<E> implements Iterable<E> {
  ArrayList<E> a = new ArrayList<E>();
  public void add(E x) { a.add(x); }
  public void set(int i, E x) { a.set(i, x); }
  public E get(int i) { return a.get(i); }
  public int size() { return a.size(); }
  public void remove(E x) { a.remove(x); }
  public Iterator<E> iterator() { return a.iterator(); }
}
```

このように、クラスの冒頭でパラメタ E を持つことを指定したら、[22] あとは E を普通の型のように使うだけです。その中では、実際に並びの要素を保持するのに ArrayList<E>を使っているので、各メソッドも ArrayList の対応するものを呼び出すだけです。

ArrayList を使ったら簡単すぎるので、代わりに内部では配列を使って要素を保持するようにしてみたいですね? しかし残念、Java にはパラメタつきクラスの中で型パラメタの配列を作ることができないという制約があるのです。[23]

22) foreach 文で使えるようにするためには、implements Iterable<E>が必要だったことにも注意。

23) なぜこのような制約があるのか説明するのは、本書の範囲を超えています。ただ、型パラメタ機構が Java 言語に後から入れられた関係で、色々と使いにくい制約が生まれてしまった、ということだけ覚えておいてください。

ところで、全部 ArrayList にあるメソッドだけでは芸がないので、たとえば draw() というメソッドも持たせるようにして、このメソッドを呼ぶと中身の図形をすべて描画してくれる、というふうにしてみましょう。

ただしそれには、型パラメタ E が Figure インタフェースに従っていることを宣言する必要があります (そうしないと、E 型の値それぞれが draw() を持っているということが保証されないですから)。これを宣言するためには、パラメタの定義のところを「<E extends Figure>」と書くことになっています。そのように直した版を示しましょう。

```
static class MyList<E extends Figure>
    implements Iterable<E>, Figure {
  ArrayList<E> a = new ArrayList<E>();
  public void add(E x) { a.add(x); }
  public void set(int i, E x) { a.set(i, x); }
```

```
    public E get(int i) { return a.get(i); }
    public int size() { return a.size(); }
    public void remove(E x) { a.remove(x); }
    public Iterator<E> iterator() { return a.iterator(); }
    public void draw(Graphics g) {
      for(Figure f1: a) { f1.draw(g); }
    }
  }
```

なお、このクラスそのものが Figure を実装するということは別に書かなくてもいいのですが、draw() を持っているわけですから書いてみました。ちなみに、このクラスがあったとすると、本体の方は次のように直します。まず冒頭で ArrayList の代わりにこちらを使います。

```
MyList<Figure> figs = new MyList<Figure>();
```

そして、paintComponent ではこれの draw() を呼びます。

```
public void paintComponent(Graphics g) {
  figs.draw(g);
}
```

4　コンポジションで絵を動かす

上で、継承を用いて親クラスに拡張部分を追加していく方法には複数の問題があると述べました。[24] では、これらの問題を起こさないような拡張の方法はあるのでしょうか? あります。それは**コンポジション** (複合) と呼ばれるもので、名前が示す通り、複数の (別個の) オブジェクトを「組み合わせて」使います (図 5 右)。

24)クラス間の依存関係の増大、カプセル化の破壊、拡張の組み合わせが困難、の 3 点がその主なものです (図 5 左)。

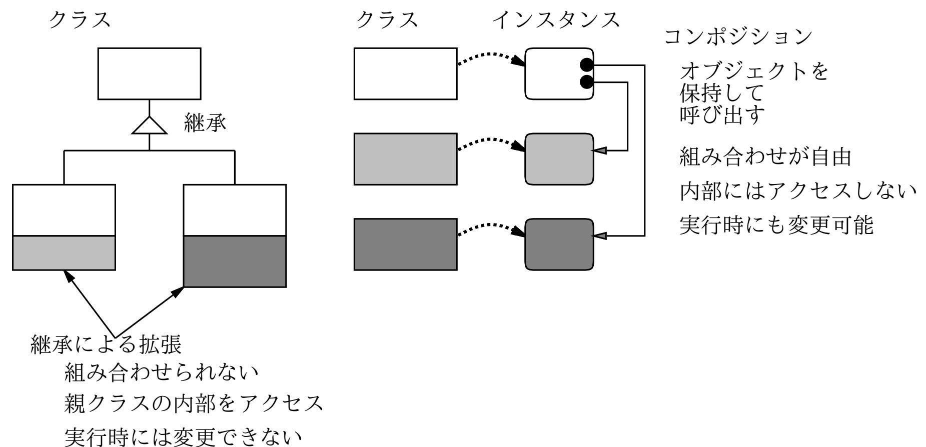


図 5: 継承とコンポジション

つまり、オブジェクト A 別のオブジェクト B をインスタンス変数に保持することはいつでもできますし、その変数を通じて B のメソッドを呼び出すことも自由です。[25] 当り前みたいですが、この当り前の機能を用いて複数のオブジェクトを連携させるのがコンポジションです。

25)メソッドを呼び出すということは、B の機能を使うということですから、A の制御に従って B を動作させることができます。

例題 7-1: 動く円

では、その最初の例として、前にもやった「動く円」を作ってみます。[26] 正確には、「直線的に動き、途中から色が徐々に変化する円」です (図 6)。

26)動き方はちょっと変えてありますが、練習問題にあるように、例題 6-1 と同じ動き方のものも簡単に作れます。

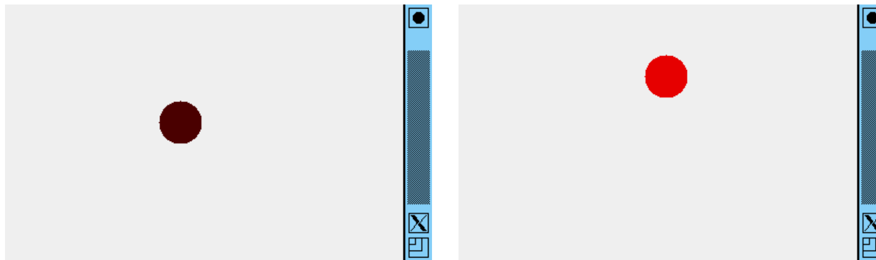


図 6: 直線的に移動し変色する円

その基本的な考え方は、「円」オブジェクトと「動かす」「色を変える」オブジェクトを分けて、両者を組み合わせて使う、というものです。具体的に見てみましょう。まず、「図形」と「動かす」や「色を変える」を分けたので、図形を入れるコンテナ figs と、時間とともに変化させるためのオブジェクトを入れるコンテナ anim の 2 つのインスタンス変数があります。そしてコンストラクタの冒頭で、円を作って figs に追加したあと、その円を「時刻 3 の位置が (100, 200) で、時刻 5 までの間に位置 (200, 60) まで等速度で動く」オブジェクトと「時刻 4 から 5 までの間に黒から赤に色を徐々に変化させる」オブジェクトを作り、anim に入れます。[27]

27)時刻の単位は「プログラム開始時点からの経過秒数」です。

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

public class Sample71 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  ArrayList<Animation> anim = new ArrayList<Animation>();

  public Sample71() {
    Color col1 = Color.BLACK, col2 = Color.RED;
    Circle c1 = new Circle(col1, 100, 200, 20);
    figs.add(c1);
    anim.add(new LinearMove(c1, 3, 100, 200, 5, 200, 60));
    anim.add(new ColorTrans(c1, 4, col1, 5, col2));
    setOpaque(false);
    final long tm0 = System.currentTimeMillis();
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        float tm = 0.001f*(System.currentTimeMillis()-tm0);
        for(Animation a: anim) { a.setTime(tm); }
        repaint();
      }
```

```
  }).start();
}
```

反復動作部分では前と同様、経過秒数を求め、[28] その時刻をすべての「動かす」オブジェクトに設定してから画面を再描画し、次の周回前に 50 ミリ秒待つことを繰り返します。[29]

28)今回は単精度実数型 float を使っていますが、それは float も使って見せたいということと、型名のスペルが double より短いのでソースコードの横幅が少し収めやすくなるという読者にはあまり関係のない都合によります。

29)そうすると、「動かす」オブジェクトと「色を変える」オブジェクトの機能により、円が動いて変色するわけです。

paintComponent() と main() は変わっていません。

```
public void paintComponent(Graphics g) {
  for(Figure f: figs) { f.draw(g); }
}
public static void main(String[] args) {
  JFrame app = new JFrame();
  app.add(new Sample71());
  app.setSize(400, 240);
  app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  app.setVisible(true);
}
```

Figure インタフェースは、draw() のほかに位置を変更する moveTo()、色を変更する setColor() を持つものとしました。それとは別に、時刻の変化を受け取るインタフェース Animation を用意しています。

```
interface Figure {
  public void draw(Graphics g);
  public void moveTo(float x, float y);
  public void setColor(Color c);
}
interface Animation {
  public void setTime(float dt);
}
```

図形は、別の図形と共通部分をくくり出す抽象クラス SmpleFigure を用意し、Circle はそのサブクラスにしています。今回は SimpleFigure にも draw() を持たせ、色はここで設定しています。[30][31]

30)サブクラスからは、「**super.** メソッド名 (…)」という書き方で親クラスのさまざまなメソッドを呼ぶことができるので、これを使って SimpleFigure の draw() を呼びます。

31)Circle のメソッド hit() はこの例題では不要ですが、次の例題 (例題 7-2) で使うので入れてあります。

```
static abstract class SimpleFigure implements Figure {
  Color col;
  float xpos, ypos;
  public SimpleFigure(Color c, float x, float y) {
    col = c; xpos = x; ypos = y;
  }
  public void moveTo(float x, float y) {xpos=x;ypos=y;}
  public void setColor(Color c) { col = c; }
  public void draw(Graphics g) { g.setColor(col); }
}
static class Circle extends SimpleFigure {
  Color col;
  float rad;
```

```
  public Circle(Color c, float x, float y, float r) {
    super(c, x, y); rad = r;
  }
  public boolean hit(float x, float y) {
    return (xpos-x)*(xpos-x)+(ypos-y)*(ypos-y)<=rad*rad;
  }
  public void draw(Graphics g) {
    int x = (int)(xpos-rad), y = (int)(ypos-rad);
    super.draw(g);
    g.fillOval(x, y, (int)rad*2, (int)rad*2);
  }
}
```

最後に、位置を変化させる `LineaMove` と色を変化させる `ColorTrans` を見てみましょう。これらは作成時に図形と「変化を開始する時刻とその時の XY 座標 (ないし色)」「変化が完了する時刻とその時の XY 座標 (ないし色)」を受け取り、インスタンス変数にこれらを覚えます。[32]そして、`setTime()` では、渡された現在時刻 `t` が開始時刻より前か終了時刻より後なら何もせずに終わります。そうでない場合は、まず変数 `p`、`q` を計算しますが、これは次の式によっています。[33]

$$p = \frac{t_2 - t}{t_2 - t_1}, \qquad q = 1.0 - p \left(= \frac{t - t_1}{t_2 - t_1} \right)$$

これで、開始時と終了時の XY 座標や RGBA 値について、$px_1 + qx_2$ などの値を計算することで、連続的に変化するとした場合の現時点における値が計算できるわけです。そして、これらの値に基づいて保持している `Figure` オブジェクトの XY 座標や色を設定します。

```
static class LinearMove implements Animation {
  Figure fig;
  float time1, xpos1, ypos1, time2, xpos2, ypos2;
  public LinearMove(Figure f, float t1,
      float x1, float y1, float t2, float x2, float y2) {
    time1 = t1; xpos1 = x1; ypos1 = y1;
    time2 = t2; xpos2 = x2; ypos2 = y2; fig = f;
  }
  public void setTime(float t) {
    if(t < time1 || time2 < t) { return; }
    float p = (time2-t)/(time2-time1), q = 1.0f - p;
    fig.moveTo(p*xpos1 + q*xpos2, p*ypos1 + q*ypos2);
  }
}
static class ColorTrans implements Animation {
  Figure fig;
  float time1, time2;
  int r1, g1, b1, a1, r2, g2, b2, a2;
  public ColorTrans(Figure f, float t1, Color c1,
                              float t2, Color c2) {
```

[32]色の方は、後で使いやすいように RGBA 値を取り出して覚えます。

[33]つまり、開始時刻から終了時刻までの時間を 1.0 として、`p` は「現時点でどれだけ残っているかの比率」、`q` は「現時点でどれだけ経過し終わったかの比率」です。

```
      fig = f; time1 = t1; time2 = t2;
      r1 = c1.getRed(); g1 = c1.getGreen();
      b1 = c1.getBlue(); a1 = c1.getAlpha();
      r2 = c2.getRed(); g2 = c2.getGreen();
      b2 = c2.getBlue(); a2 = c2.getAlpha();
    }
    public void setTime(float t) {
      if(t < time1 || time2 < t) { return; }
      float p = (time2-t)/(time2-time1), q = 1.0f - p;
      fig.setColor(new Color((int)(p*r1+q*r2),
       (int)(p*g1+q*g2),(int)(p*b1+q*b2),(int)(p*a1+q*a2)));
    }
  }
} // ←外側クラスの終わりの「}」
```

これにより、時刻 1 秒から 3 秒までの間に直線的に円が窓の中を移動し、最初は色は黒だけれども、2 秒目から変化し始めて最後は赤になるわけです。このように、図形と動きや色変化を別のオブジェクトにしておくことで、それらを独立に、しかも演習問題にあるように必要なら複数組み合わせて、使うことができるのです。

演習 7-1 例題 Sample71.java をそのまま打ち込んで動かしなさい。動いたら次のように変更してみなさい。

a. 動く円の数を増やしたり、円以外の図形を増やしてみなさい。それぞれ、動き方や色の変化のしかたを変え、複雑な変化もさせてみること。[34]

b. 動き方として例題 6-1 でやったような往復運動を作ってみなさい。さらに、「一定時間だけ」往復運動を行ったり、往復運動をしている前後の時間だけ見えているようにしてみなさい。[35]

c. 「夜空の月が沈み、 太陽が昇って、空が明るくなる」のような、ストーリ性のあるアニメーションを作成してみなさい。

d. 4 章で取り上げたような複合図形も表示できるようにして、複合図形を使ったストーリ性のあるアニメーションを作成してみなさい。とりあえず複合図形は動いたり色が変わったりしなくてもよいです。[36]

e. やはり、複合図形も動いたり色が変わったりするようにしてみなさい。[37]さらにできれば、複合図形の「形も」時間とともに変わるようにしてみなさい。[38]

f. 自分が作ってみたいと思うようなストーリ性のあるアニメーションを作成しなさい。

例解 7-1-b

往復運動のクラス、一定時間だけ動くクラス、一定時間だけ見えるクラスはたとえば次のようになるでしょう。

34) A → B → C のように複数段階にわたって位置や色を変化させたり、一度変化した後、逆向きに変化して元に戻ったりさせてみるとよいでしょう。

35) ヒント: コンポジションの考え方で、「Animation オブジェクトの動作を一定時間 ON にする Animation オブジェクト」「Figure オブジェクトを一定時間だけ見えるようにする Figure オブジェクト」などを作るとよいでしょう。

36) 複合図形クラスに「何もしない」(メソッド本体が空っぽの) moveTo() や setColor() を用意すればよいでしょう。

37) 2 色使う図形を 1 つの setColor() で色を変えられるようにするには、Color オブジェクトのメソッド darker()、brighter() で「ある色を暗くした/明るくした色」を作り出して 2 色目として使用するとよいでしょう。

38) 複合図形そのものも Animation インタフェースを実装するようにして、内部の図形の位置などを時刻につれて変化させるようにします。

```
static class ZigzagMove implements Animation {
  Figure fig;
  float time1, xpos1, ypos1, xpos2, ypos2;
  public ZigzagMove(Figure f, float t1,
                    float x1, float y1, float x2, float y2) {
    time1 = t1; xpos1 = x1; ypos1 = y1;
    xpos2 = x2; ypos2 = y2; fig = f;
  }
  public void setTime(float t) {
    float q = (t % time1) / time1, p = 1.0f - q;
    fig.moveTo(p*xpos1 + q*xpos2, p*ypos1 + q*ypos2);
  }
}
static class TimedAnimation implements Animation {
  Animation anim;
  float time1, time2;
  public TimedAnimation(Animation a, float t1, float t2) {
    anim = a; time1 = t1; time2 = t2;
  }
  public void setTime(float t) {
    if(t < time1 || time2 < t) { return; }
    anim.setTime(t - time1);
  }
}
static class TimedAppearance implements Figure, Animation {
  Figure fig;
  float time, time1, time2;
  public TimedAppearance(Figure f, float t1, float t2) {
    fig = f; time1 = t1; time2 = t2;
  }
  public void moveTo(float x, float y) { fig.moveTo(x, y); }
  public void setColor(Color c) { fig.setColor(c); }
  public void setTime(float t) { time = t; }
  public void draw(Graphics g) {
    if(time1 <= time && time <= time2) { fig.draw(g); }
  }
}
```

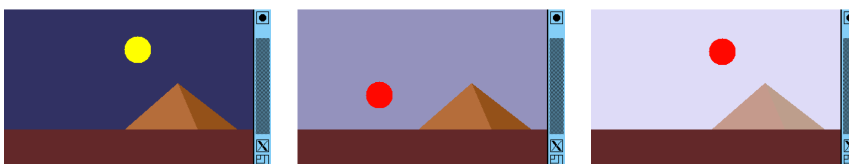
ZigzagMoveはLinearMoveと同様に内部に図形を持つAnimationで、コンストラクタで指定された周期で始点から終点まで直線移動し、始点にワープすることを繰り返します。[39]TimedAnimationは、内部にAnimationを持ち、指定された時間範囲内だけ内部のAnimationのsetTime()を呼んで動作させせます。TimedApperanceは、内部にFigureを持ち、自分のdraw()が呼ばれた際、指定された時間範囲内だった場合のみ内部のFigureのdraw()を呼びます。[40]これらを用いてアニメーションを組み立てる部分 (コンストラクタの冒頭部分) は次のようになります。

```
Circle c1 = new Circle(Color.RED, 100, 200, 20);
TimedAppearance t1 = new TimedAppearance(c1, 1, 8);
figs.add(t1); anim.add(t1);
Animation a1 = new ZigzagMove(c1, 1, 100, 200, 200, 100);
anim.add(new TimedAnimation(a1, 3, 7));
```

39) 最初の点に戻るのは例題6-1と同様、周期time1による剰余演算%を使い、その結果をtime1で割ることで0.0f〜1.0fの値を計算します。0から始まる方がqになることに注意。

40) このクラスは「変化する図形」なのでFigure、Animationの両方をimplementsします。そして、メソッドmoveTo()やsetColor()は、内部のFigureに呼び出しを中継します。

例解 7-1-c



月が沈み日が昇ると空が明るくなり、登りきるとピラミッドがうすくなって消えていく、というのを作ってみます。コンストラクタの冒頭部分を示します。[41]

41)消えて行くピラミッドは透明度が最大の色に変化させることで実現しています。なお、「月」の円をそのまま色だけ変えて「太陽」にしています。手抜きですがそれらしく見えればよいということで…

```
Color s1 = new Color(50,50,100), s2 = new Color(220,220,250);
Rect sky = new Rect(s1, 200, 100, 400, 210);
Circle c1 = new Circle(Color.YELLOW, 200, 60, 20);
Color pc1 = new Color(180,110,60), pc2 = new Color(150,80,30);
Color pc3 = new Color(160,100,40,0);
Triangle p1 = new Triangle(pc1, 260, 110, 180, 180, 290, 180);
Triangle p2 = new Triangle(pc2, 260, 110, 290, 180, 350, 180);
Rect grnd = new Rect(new Color(100,40,40), 200, 220, 400, 80);
figs.add(sky); figs.add(c1); figs.add(p1); figs.add(p2);
figs.add(grnd);
anim.add(new LinearMove(c1, 3, 200, 60, 5, 20, 220));
anim.add(new ColorTrans(c1, 5, Color.YELLOW, 6, Color.RED));
anim.add(new LinearMove(c1, 6, 20, 220, 8, 200, 60));
anim.add(new ColorTrans(sky, 6, s1, 8, s2));
anim.add(new ColorTrans(p1, 10, pc1, 12, pc3));
anim.add(new ColorTrans(p2, 10, pc2, 12, pc3));
```

長方形と三角形のクラスも示しておきます。[42][43][44]

42)長方形にも後で使うので `hit()` を用意しました。

43)三角形は移動に便利なように作成時に 2 番目と 3 番目の点については最初の点からの変移を覚えるようにしました。

44)配列を初期値つきで作って、すぐ変数に入れる場合は、このように「`new int[]`」を省略できます。

```
static class Rect extends SimpleFigure {
  Color col;
  float width, height;
  public Rect(Color c, float x, float y, float w, float h) {
    super(c, x, y); width = w; height = h;
  }
  public boolean hit(float x, float y) {
    return xpos-width/2 <= x && x <= xpos+width/2 &&
           ypos-height/2 <= y && y <= ypos+height/2;
  }
  public void draw(Graphics g) {
    int x = (int)(xpos-width/2), y = (int)(ypos-height/2);
    super.draw(g); g.fillRect(x, y, (int)width, (int)height);
  }
}
static class Triangle extends SimpleFigure {
  Color col;
  float dx1, dy1, dx2, dy2;
  public Triangle(Color c, float x, float y,
                  float x1, float y1, float x2, float y2) {
    super(c, x, y);
    dx1 = x1-x; dy1 = y1-y; dx2 = x2-x; dy2 = y2-y;
  }
  public void draw(Graphics g) {
    int[] xs = {(int)xpos, (int)(xpos+dx1), (int)(xpos+dx2)};
    int[] ys = {(int)ypos, (int)(ypos+dy1), (int)(ypos+dy2)};
    super.draw(g); g.fillPolygon(xs, ys, 3);
  }
}
```

例解 7-1-e

車がやって来て、人が家の影から現れて車の上に (!) 乗り、車が人を載せて去って行きますが、その時は車のタイヤが上下に動く、というアニメーションを作ってみました。コンストラクタの冒頭部分を示します。

```
    Human m1 = new Human(new Color(200,80,120), 10, 100, 120);
    House h1 = new House(new Color(50,150,200), 25, 100, 150);
    Car c1 = new Car(Color.GREEN, 10, 350, 50);
    figs.add(m1); figs.add(h1); figs.add(c1);
    anim.add(new LinearMove(c1, 2, 350, 50, 4, 200, 180));
    anim.add(new LinearMove(m1, 4, 100, 120, 6, 200, 120));
    anim.add(new TimedAnimation(c1, 7, 10));
    anim.add(new LinearMove(c1, 7, 200, 180, 10, 500, 180));
    anim.add(new LinearMove(m1, 7, 200, 120, 10, 500, 120));
```

人と家のクラスは省略します。[45] 車は、時間の進行につれてタイヤの位置が「通常→凸凹→通常→凹凸」の 4 段階に順番に変化するため、Figure と Animation の両方のインタフェースを実装します。[46]

```
  class Car implements Figure, Animation {
    Triangle t1, t2;
    Rect r1;
    Circle c1, c2;
    float unit, time;
    public Car(Color c, float u, float x, float y) {
      t1 = new Triangle(c, x-u*3, y, x-u*2, y-u*2, x+u*3, y);
      t2 = new Triangle(c, x-u*2, y-u*2, x+u*2, y-u*2, x+u*3, y);
      r1 = new Rect(c, x, y+u, u*8, u*2);
      c1 = new Circle(c.darker(), x-u*2, y+u*2, u);
      c2 = new Circle(c.darker(), x+u*2, y+u*2, u); unit = u;
    }
    public void moveTo(float x, float y) {
      float d = {0f, unit/4, 0f, -unit/4}[(int)(10*time) % 4];
      t1.moveTo(x-unit*3, y); t2.moveTo(x-unit*2, y-unit*2);
      r1.moveTo(x, y+unit); c1.moveTo(x-unit*2, y+unit*2+d);
      c2.moveTo(x+unit*2, y+unit*2-d);
    }
    public void setColor(Color c) {
      t1.setColor(c); t2.setColor(c); r1.setColor(c);
      c1.setColor(c.darker()); c2.setColor(c.darker());
    }
    public void setTime(float t) { time = t; }
    public void draw(Graphics g) {
      t1.draw(g);t2.draw(g);r1.draw(g);c1.draw(g);c2.draw(g);
    }
  }
```

[45] 座標を float にして色を 1 色指定に直し、moveTo() で各部品の moveTo() を呼び出すようにしました。車もこの点は同じです。

[46] 例解ではタイヤの変化は人が乗ってからだけなので、TimedAnimation を介して 7 秒目以後のみ変化するようにしています。

5 状態遷移を実現する

ここまででアニメーションを作ったりゲームを作ったり色々できるようになりましたが、現実のアプリケーションでは「オープニングアニメーションがあって、それからゲームが始まる」のような場面転換が普通に行われます。

コンピュータサイエンスの用語では、プログラムがいくつかの**状態** (state) を持っていて、その間で**遷移** (transition) が行われる、という考え方はよく使われます。上述の場面と場面転換も、状態と状態遷移に対応させることができます。状態とその間の遷移は、**状態遷移図** (state transition diagram) を描くと把握しやすくなります。

例題 7-2: 状態遷移を持つプログラム

それでは実際に状態遷移を活用してみましょう。ここで示す例題は、図 7 に示す状態遷移図[47] を持つプログラムで、最初に動画かゲームかを選択し、動画の場合は予告説明があってから動画が表示され、終わると選択画面に戻ります。ゲームを選択するとゲームができ、終わると選択画面に戻ります。その画面を図 8 に示します。

47)○が状態を表し、矢線が遷移を表します。○の横に番号が書いてありますが、これは例題プログラムでの「場面番号」を示しています。

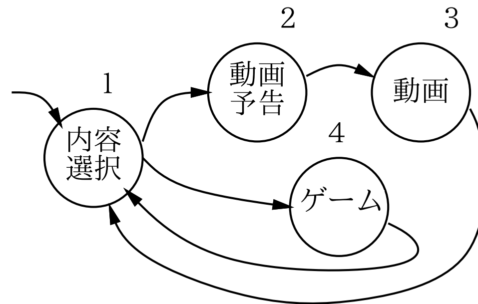


図 7: 4 つの状態から成る状態遷移図

図 8: 4 つの場面から成るプログラム

オブジェクト指向で状態と状態遷移を表す自明な方法は、個々の状態を 1 つのオブジェクト (インスタンス) に対応させ、各状態オブジェクトに「次の状態」を教えてもらうことです。

今回もそうすることにして、さらにそれぞれの状態 (場面) がしばらくの時間続くので、その場面が終わりかどうかも教えてもらうことにします。この両者とも、内部に対応するインスタンス変数を持ち、その値を返すメソッドを用意します。

また、使用する `Figure` オブジェクトや `Animation` オブジェクトを場面ごとに分けたいので、コンテナ `figs`、`anim` も場面オブジェクトのインスタンス変数にします。あと、マウスクリックも扱いたいので、それを受け取るメソッドを用意します。

これらをあわせた、1 つの場面を扱うクラス `Scene` は、次のようになります。

```
static class Scene {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  ArrayList<Animation> anim = new ArrayList<Animation>();
  boolean ended = false;
  Scene next = null;
  public void draw(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
  }
  public void setTime(float t) {
    for(Animation a: anim) { a.setTime(t); }
  }
  public void press(int x, int y) { }
  public boolean isEnded() { return ended; }
  public Scene getNext() { return next; }
}
```

しかしこれでは、枠組みはできていても、何の図形も表示されず、クリックされても何も渡されない、と思いましたか? その通りです。では、このクラスは何の役に立つのでしょうか? それは…これまでずっとやってきたのと同じでして、このクラスのサブクラスを作って具体的な場面のための初期設定や動作を記述すればいいのです。実際にやって見ましょう。まず、場面 1 ですが、これは 2 つの円と 2 つの動く画像、1 つのテキストを持ち、クリックされた円に応じて次の状態を選択します。

```
static class Scene1 extends Scene {
  Circle c1 = new Circle(Color.YELLOW, 100, 100, 60);
  Circle c2 = new Circle(Color.YELLOW, 250, 100, 60);
  public Scene1() {
    Picture p1 = new Picture("kuno1.png", 0, 0);
    Picture p2 = new Picture("sun1.png", 0, 0);
    figs.add(c1); figs.add(c2); figs.add(p1); figs.add(p2);
    anim.add(new ShakeMove(p1, 100, 100, 10, 10));
    anim.add(new ShakeMove(p2, 250, 100, 10, 10));
    figs.add(new Text(20, 25, "お好きな方をクリック",
                      new Font("serif", Font.BOLD, 18)));
  }
  public void press(int x, int y) {
    if(c1.hit(x, y)) {
      next = new Scene2(); ended = true;
    } else if(c2.hit(x, y)) {
      next = new Scene4(); ended = true;
    }
  }
}
```

次に場面 2 ですが、これはテキストを 5 秒間表示するだけです。ただし、途中で画面がクリックされた場合は 5 秒たっていなくてもすぐに終わりになります。[48]

48)「`ended |= (t > 5)`」というのは、`ended` と `t > 5` の論理和 (少なくともどちらか 1 方が真なら真) を計算してそれを `ended` に入れます。ですから、既に `true` になっていればそのままだけれど、`false` であれば 5 秒経っていたら `true` にする、ということです。

```
static class Scene2 extends Scene {
  public Scene2() {
    next = new Scene3();
    figs.add(new Text(20, 90, "動画作品、見て下さい。",
                      new Font("serif", Font.BOLD, 18)));
  }
  public void press(int x, int y) { ended = true; }
  public void setTime(float t) { ended |= (t > 5); }
}
```

場面3ですが、作品を用意するところはこれまでと同じなので省略します。[49] それ以外の仕事は、次の場面の設定と、`setTime()` で親の `setTime()` を読んでから、終わりの時刻なら `ended` を `true` にすることだけです。

49)そのために `figs` と `anim` をまったく同じように使えばよくしてあるのでした。

```
static class Scene3 extends Scene {
  public Scene3() {
    next = new Scene1();
    // 省略; 図形と動きを figs と anim に入れる
  }
  public void setTime(float t) {
    super.setTime(t); ended |= (t > 14);
  }
}
```

場面4は、テキスト2つ、動かす円1つ、成功したかを表す変数 `ok`、現在時刻、終了時刻をインスタンス変数として保持します。コンストラクタでは図形を配置した後、円を `ZigzagMove` で動かすように設定します。

```
static class Scene4 extends Scene {
  Font fn = new Font("serif", Font.BOLD, 18);
  Text t1 = new Text(20, 25, "円をクリックしてください", fn);
  Text t2 = new Text(20, 55, "0.00", fn);
  Circle c1 = new Circle(Color.BLUE, 60, 200, 20);
  boolean ok = false;
  float curtime = 0f, endtime = 60f;
  public Scene4() {
    next = new Scene1();
    figs.add(t1); figs.add(t2); figs.add(c1);
    anim.add(new ZigzagMove(c1, 1f, 60, 200, 340, 40));
  }
  public void press(int x, int y) {
    if(!c1.hit(x, y)) { return; }
    ok = true; c1.setColor(Color.RED); endtime = curtime + 5;
    t1.setText((curtime<3f)?"Good Job!":"So-so");
  }
  public void setTime(float t) {
    super.setTime(t); curtime = t; ended |= (t > endtime);
    if(!ok) { t2.setText(String.format("%4.2f", t)); }
```

```
    }
  }
```

press() では、円に当たっていなければ何もせず戻ります。当たっていれば、ok を true にし、円を赤くし、終了時刻を現在から 5 秒後にして、最後に所要時間に応じて「Good Job!」か「So-so」を表示します。setTime() では、まず親の setTime() を呼んで図形を動かし、現在時刻を記録し、終了時刻を過ぎていたら終わりにします。最後にまだ ok でなければ現在時刻の表示を進めます。

以上で場面群は終わったので、プログラム本体を先頭から見て行きます。今回はメインクラスのインスタンス変数は現在場面を表す変数 cur だけです。[50] 動作の方は、マウスクリックに対応して cur の press() を呼び出し、時刻についてはこれまで同様、反復動作から setTime() を呼ぶだけです。[51]

50)初期値として場面 1 を入れています。

51)主要な仕事を場面オブジェクトの中に移したので、本体はずっと簡単になっていますね。

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.util.*;
import javax.imageio.*;
import javax.swing.*;

public class Sample72 extends JPanel {
  Scene cur = new Scene1();
  long tm0 = System.currentTimeMillis();
  public Sample72() {
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        cur.press(evt.getX(), evt.getY());
      }
    });
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        float tm = 0.001f*(System.currentTimeMillis()-tm0);
        cur.setTime(tm); repaint();
        if(cur.isEnded()) {
          cur = cur.getNext(); tm0 = System.currentTimeMillis();
        }
      }
    }).start();
  }
```

あとのクラスは、ほとんどこれまでに示したものばかりですが、Text と Picture は SimpleFigure のサブクラスにするように多少手直ししました。

```
  // paintComponent, main, 先の Scene〜Scene4
```

```
  // Figure, Animation, SimpleFigure, Circle, Rect, Triangle
  static class Text extends SimpleFigure {
    String txt;
    Font fn;
    public Text(int x, int y, String t, Font f) {
      super(Color.BLACK, x, y); txt = t; fn = f;
    }
    public void setText(String t) { txt = t; }
    public void draw(Graphics g) {
      super.draw(g); g.setFont(fn);
      g.drawString(txt, (int)xpos, (int)ypos);
    }
  }
  static class Picture extends SimpleFigure {
    BufferedImage img;
    int width, height;
    public Picture(String fname, int x, int y) {
      super(Color.WHITE, x, y);
      try {
        img = ImageIO.read(new File(fname));
      } catch(Exception ex) { }
      xpos = x; ypos = y;
      width = img.getWidth(); height = img.getHeight();
    }
    public void draw(Graphics g) {
      int x = (int)xpos-width/2, y = (int)ypos-height/2;
      g.drawImage(img, x, y, null);
    }
  }
  // その他アニメーションに必要なクラス群
} // 本体クラスの終わりの「}」
```

このように、Sceneというクラスを土台としてサブクラスを作ってカスタマイズすることで、柔軟にさまざまな場面を作り出せるわけです。

演習 7-2 Sample72.javaをそのまま打ち込んで動かしなさい。動いたら、次のような変更を施してみなさい。

a. 例題の状態遷移を変更してみなさい。たとえば、動画を見せた後最初に戻るかわりにゲームに進むようにしてみなさい。いきなりプログラムを修正するのではなく、状態遷移図を描いてから着手すること。

b. ゲームが終わった時、「Good Job」なら動画が見られるが、そうでなければ先頭に戻るようにしてみなさい。

c. 自分がこれまでに作ったアニメーションやゲームなどをSheneのサブクラスとしてパッケージし、例題に組み込んでみなさい。

d. Scene2のような「メッセージを表示する」場面はあちこちで役立ちます。そこで、Scene2をもとにして「表示するTextオブジェ

クト、秒数、次の状態」をコンストラクタで渡すようにした場面クラス TextScene を作ってみなさい。それを用いて、4 つのメッセージ表示が順番に切り替わって行き、最後からはまた先頭に戻るような構成を作ってみなさい。[52]

e. 「すごろく」「おみくじ」のようなプログラムを作る時には、複数の後続場面からランダムに 1 つ選ぶような場面があると便利です。コンストラクタで N 個の Scene の配列を受け取り、クリックするとその中から 1 つが選ばれてそこに進む場面クラス DiceScene を作り、それを使って「おみくじ」を作ってみなさい。

f. 複数場面から成る好きなゲームプログラムを作ってみなさい。[53]

52)「先頭に戻る」ために、「次の場面を設定する」メソッドを用意するとよいでしょう。また、getNext() をオーバライドして、次の状態を返す (切り替わる) 時に、自分は「終わりでない」状態に戻す必要があるでしょう。

53)ロールプレイイングでも、複数ステージのアクションゲームでも、その他のものでもよいです。

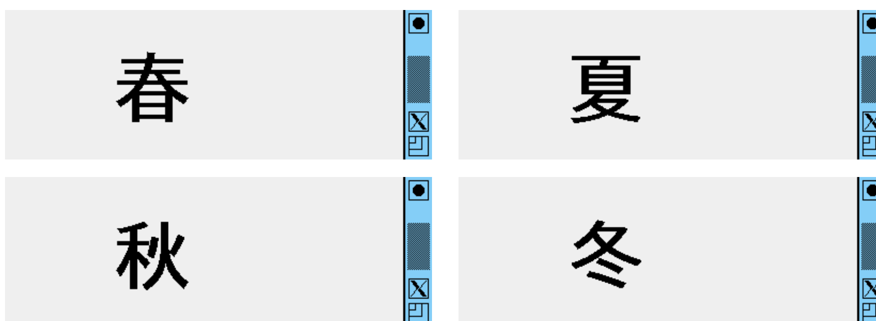
例解 7-2-d

この問題は、これまで場面ごとにクラスを分けて決め打ちで書いていたものを、1 つのクラスでインスタンスごとに別の場面にしようとした場合の問題に気づかせてくれます。まず、TextScene を見てみましょう。

```
static class TextScene extends Scene {
  Text t1;
  float endtime;
  public TextScene(Text t, float e, Scene n) {
    t1 = t; endtime = e; next = n; figs.add(t);
  }
  public void press(int x, int y) { ended = true; }
  public void setTime(float t) { ended |= (t > endtime); }
  public void setNext(Scene s) { next = s; }
  public Scene getNext() { ended = false; return next; }
}
```

とても短いですが、ヒントの通り getNext() をオーバライドして状態を戻すようにしています。setNext() は本体側を見れば必要性が分かります。[54] コンストラクタの冒頭だけ示します。[55]

```
Font fn = new Font("sansserif", Font.BOLD, 72);
TextScene s3 = new TextScene(new Text(90,90,"冬",fn),5f,null);
TextScene s2 = new TextScene(new Text(90,90,"秋",fn),5f,s3);
TextScene s1 = new TextScene(new Text(90,90,"夏",fn),5f,s2);
cur = new TextScene(new Text(100, 100, "春", fn), 5, s1);
s3.setNext(cur);
```



54)つまり、冬から逆順に作ることで各季節の「次」を設定できますが、循環しているので「冬」の次が「春」というところだけ後で設定する必要があるわけです。

55)フォント名のところで指定している「**"sansserif"**」というのは、「セリフ (ひげ) の無い書体」という意味で、日本語でいうとゴシック体になります。

例解 7-2-d

さっそく、DiceScene を見てみましょう。絵の配列と次の場面の配列を受け取り、時間進行につれて表示される絵が切り替わります。[56]クリックするとその時の絵で止まり、対応する状態が次の状態になります。[57]

56)このため、draw() を自前で実装しています。その方がこの場合は簡単ですから。

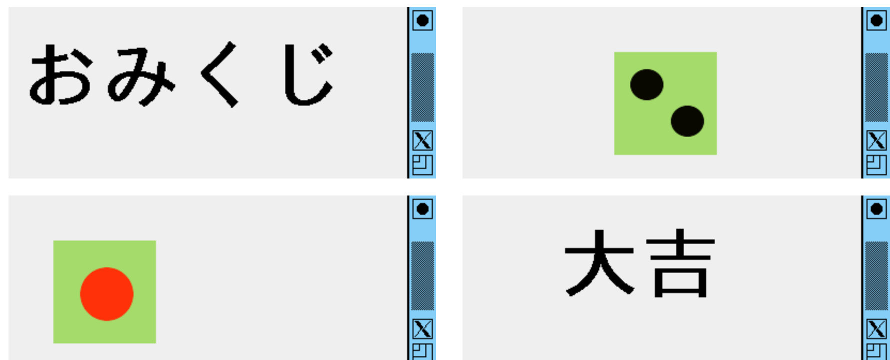
57)ここでも状態を再利用するため、getNext() をオーバライドして必要なインスタンス変数を元に戻しています。

```
static class DiceScene extends Scene {
  Picture[] pics;
  Scene[] scenes;
  int sel = 0, fin = -1;
  float curtime, endtime = 20f;
  public DiceScene(Picture[] ps, Scene[] ss) {
    pics = ps; scenes = ss;
  }
  public void draw(Graphics g) { pics[sel].draw(g); }
  public void press(int x, int y) {
    fin = sel; next = scenes[sel]; endtime = curtime + 3;
  }
  public void setTime(float t) {
    curtime = t; ended |= (t > endtime);
    if(fin < 0) { sel = (int)(t*8) % pics.length; }
  }
  public Scene getNext() {
    ended = false; fin = -1; endtime = 20f; return next;
 }
}
```

メインクラス側のコンストラクタの冒頭部分は次の通りです。[58]

58)画像はサイコロの 1 と 2 と 3 にしてあります。6 個作ってもコードが長くなるだけですから。

```
Font fn = new Font("sansserif", Font.BOLD, 72);
TextScene s0 = new TextScene(new Text(10,90,"おみくじ",fn),
                             5f, null);
TextScene s1 = new TextScene(new Text(90,90,"大吉",fn),5f,s0);
TextScene s2 = new TextScene(new Text(90,90,"中吉",fn),5f,s0);
TextScene s3 = new TextScene(new Text(90,90,"小吉",fn),5f,s0);
DiceScene d1 = new DiceScene(
  new Picture[]{new Picture("d1.png",90,90),
    new Picture("d2.png",190,90),new Picture("d3.png",290,90)},
  new Scene[]{s1, s2, s3});
s0.setNext(d1); cur = s0;
```



6 まとめ

この章の前半では、図形に変化をつけるためのクラスを図形から分離することで、コンポジションによりさまざまな動きを自由に組み合わせられることを示しました。そして後半では、状態遷移の考え方を取り上げ、個々の状

態を1つのオブジェクトに対応させて次の状態も教えてもらうことで、柔軟な状態遷移を作り出せることを学びました。

ところで、冒頭に差分プログラミングは要注意でコンポジションがおすすめだと書いてあったのに、Sceneではサブクラスによる拡張を使っていますね。これは、figsやanimというコンテナやこれらを使う標準的な動作はどの場面でも共通な「同じもの」なので、それを継承して来るのが自然だからです。ただし、サブクラスの中にはSceneの機能をほとんど使っていないものもあります。そのようなものが多くなってきたら、やっぱりSceneの機能を分けてコンポジションにした方がいいかも知れません。

このように、オブジェクト指向のクラス設計は「どうするのがいい」と一概に言うことはできず、それぞれの場面ごとに検討する必要があります。ここまでに学んで来たことをもとに、考えてみてください。