オブジェクト指向技術'11 ＃1 — オブジェクト指向言語

久野　靖*

2011.4.14

1　はじめに

この科目「オブジェクト指向技術」は、今日のプログラミング言語界で主流であるオブジェクト指向プログラミング言語を前提に、言語技術がソフトウェアシステム作成においてどのような面から支援を行えるようになっているのかを概観することを目的としています。大まかな予定は次のように考えています。

- ＃1 基本概念 — オブジェクト指向言語の基本概念のおさらい、利用技術 (デザインパターン等)、フレームワークなどの考え方
- ＃2 メタ情報 — メタ情報、メタプログラミング、リフレクション、アノテーション、これらの機構の活用
- ＃3 言語処理系 — 言語処理系の構成、コンパイラコンパイラ、ソフトウェア言語工学 (SLE)、ドメイン固有言語 (DSL)
- ＃4 分散 — ネットワーク通信、メッセージモデル、RPC/RMI、オブジェクト移送、タプルスペースモデル
- ＃5 並列/並行 — 共有メモリモデル、ネットワークモデル、モニタ、共有メモリプリミティブ、トランザクション (STM)、MapReduce

実際にはやってみて内容配分を変えたりすることもあるかと思います。できるだけ具体的にモノを動かした方が面白いし納得できるので、演習を毎回行うようにします。言語は原則として Java を使用しますが、テーマによっては他の言語に登場頂くこともあるかも知れません。

成績ですが、各回の出席と、最終レポートによってつけます。レポートは各回資料に載っている演習から好きなものを 1 つ取り上げて自分でプログラムを作ってみて、結果を報告するという形にするつもりですので、毎回の演習はそのつもりでやっておいてください。では、よろしくお願いします。

2　オブジェクト指向言語とその機能

オブジェクト指向言語とは何か、という話をし始めると長くなるので (このような話題は本科目の表裏でやっている「プログラミング言語論」に適しています)、ここでは次のように天下りで決めます。

- オブジェクト指向とは、さまざまな「もの」とそれらが持つ固有の「機能」に基づく考え方であり、オブジェクト指向言語はその考え方をサポートするような言語である。
- このため、オブジェクト指向言語は「もの」(オブジェクト) を言語上で扱う機能と、「もの」ごとに固有の「機能」(メソッド) を持たせる機能がある。

*経営システム科学専攻

そしてさらに次のような限定を置きます。

- ここでは、オブジェクトを定義するために、そのひな型 (型枠) となるような言語上の単位である「クラス」を記述するような言語 (クラス方式のオブジェクト指向言語) を扱う。クラスでは、対応するオブジェクトが持つ変数 (インスタンス変数) と、オブジェクトが持つメソッド (インスタンスメソッド) を定義する。そのクラスに基づいて、個々のオブジェクト (インスタンス) が生成される。
- そしてさらに、複数のクラス間の「関係」を記述する機能として、「継承」を持つものとする。継承とは、子クラスの中に親クラスの変数定義やメソッド定義が (概念的には)「取り込まれて」きて利用可能になるという機構である。なお、メソッドについては子クラス側で親と同一のものを定義した場合、親のメソッドを「差し替える」ことができる。

これらの部分はクラス方式に固有ですが、まあ多くの言語がこれらの性質を持っているので今回はこれを前提にします。そして以下の点はクラス方式であってもなくても成り立つ、重要な点です。

- ある変数にオブジェクトが格納されているとして、それに対するメソッド呼び出しを書くと、実際に呼ばれるメソッドは**実行時点でその変数に入っているオブジェクトが持つメソッド** になる (動的分配)。

動的分配のおかげで、オブジェクト指向言語はそれ以前の型のある言語には無かったような柔軟性が多く得られ、複雑なプログラムをコンパクトに記述できます。

なお、強い型の言語の場合、ある変数に入れられるのはその変数の型に合ったオブジェクトだけです。そのために (クラス方式の場合) 可能なクラスの選択肢が 1 つだけになってしまうと動的分配もなにも無いので、ある程度広い範囲の値が入れられる規則になっています。

- クラス型 A の変数には、クラス A およびその任意のサブクラスのインスタンスが入れられる。
- インタフェース型 I の変数には、I を実装する任意のクラスのインスタンスが入れられる。

これは Java の場合ですが、Java 以外の強い型のオブジェクト指向言語でも、多くがこのような規則になっています (インタフェースについてはそのような概念が無い言語も多くあります)。

では実際に Java でこのあたりを見てみましょう。[1]

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

public class Sample11 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  public Sample11() {
    figs.add(new Circle(Color.blue, 100, 150, 30));
    figs.add(new Circle(Color.green, 120, 90, 20));
    figs.add(new Rect(Color.pink, 200, 80, 60, 40));
  }
  public void paintComponent(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
  }

  static abstract class Figure {
```

[1] 細かいコードの意味はいちいち資料で説明していられないので、知りたい方は「Java によるプログラミング入門 第 2 版」を見てください。

```
    Color col;
    int xpos, ypos;
    public Figure(Color c, int x, int y) {
      col = c; xpos = x; ypos = y;
    }
    public abstract void draw(Graphics g);
  }
  static class Circle extends Figure {
    int rad;
    public Circle(Color c, int x, int y, int r) {
      super(c, x, y); rad = r;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
  static class Rect extends Figure {
    int width, height;
    public Rect(Color c, int x, int y, int w, int h) {
      super(c, x, y); width = w; height = h;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect(xpos-width/2, ypos-height/2, width, height);
    }
  }

  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample11());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

このプログラムは図??のように、円と長方形が画面に表示されるというだけですが、その主要部分では Figure 型のオブジェクトを ArrayList に入れて保持し、画面表示時にはその中にある個々のオブジェクトの draw() を呼ぶだけです。実際にどの図形が描かれるかは、個々の Figure がどのオブジェクトであるかによって決まります (当然)。

次に、クラス Figure を見てみると、これは抽象クラスであってインスタンスを生成しません。そこで定義しているのは色と XY 座標というすべての図形に共通する変数と、これらを初期設定するコンストラクタ、および draw() のシグニチャ(名前と引数/辺値の型の情報) だけです (このようにシグニチャだけ定義して本体のないメソッドを抽象メソッドと呼びます)。

具体的な図形のクラスとしては Circle と Rect があり、これらは必要なインスタンス変数を追加し、コンストラクタで初期設定を行い (その中で親クラス Figure のコンストラクタを呼んで親クラスで定義している変数の初期設定をしてもらいます)、あとは自分固有のメソッド draw() をオーバラ

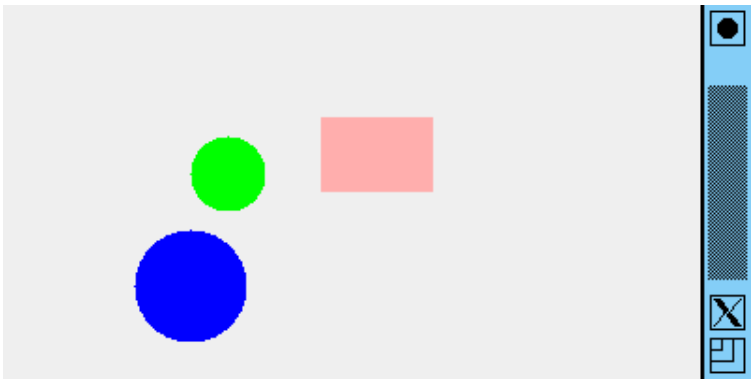

図 1: Sample11.java の実行結果

イドしています。

このように、継承とクラス (とくに抽象クラス) を使って「共通部分をくくり出す」ことで、個々のクラスの実装における重複が削減でき (なぜ重複の削減が重要なのか分かりますね?)、また親クラスの型 (この場合は `Figure` を用いて多種類のものを総称的に扱えるという特性が得られます。

ただし、継承は「総称的に扱える」「実装が共有できる」という 2 つの側面を一緒くたにしているという弱点もあります。これから「総称的に使える」側面だけを分離したのがインタフェースですが、では「実装が共有できる」という側面を分離するのはどうしたらよいでしょうか? この問題は少し後で取り上げます。

演習 1-1 上のプログラムを動かしなさい。動いたら、次のように改造してみなさい。

a. 新たな単純図形、たとえば「三角形」や「正 N 角形」を追加してみなさい。

b. 複合図形、たとえば「長方形の中に円が入った図形 (どこかの国の旗みたいなもの」、その他円と長方形を複数組み合わせた図形を追加してみなさい。

これらのことから分かることは何か考えてみなさい。

3 例解および継承の問題点

必要な部分だけ示します。三角形は 1 つ目の頂点が `xpos`、`ypos` だとして、そこから 2 つ目、3 つ目の頂点へのずれ (差分) を追加して覚えることで表現します。一方、複合図形はその構成要素である個々の図形のオブジェクトをインスタンス変数として持てば済みます。

```
public Sample11a() {
  figs.add(new Circle(Color.blue, 100, 150, 30));
  figs.add(new Circle(Color.green, 120, 90, 20));
  figs.add(new Rect(Color.pink, 200, 80, 60, 40));
  figs.add(new Triangle(Color.red, 100, 100, 200, 100, 180, 30));
  figs.add(new Flag(Color.red, Color.white, 300, 80, 20));
}
...
static class Triangle extends Figure {
  int dx1, dx2, dy1, dy2;
  public Triangle(Color c, int x0, int y0, int x1, int y1, int x2, int y2) {
    super(c, x0, y0); dx1 = x1-x0; dy1 = y1-y0; dx2 = x2-x0; dy2 = y2-y0;
  }
```

```
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillPolygon(new int[]{xpos, xpos+dx1, xpos+dx2},
                    new int[]{ypos, ypos+dy1, ypos+dy2}, 3);
    }
  }
  static class Flag extends Figure {
    Circle c1;
    Rect r1;
    public Flag(Color c, Color d, int x, int y, int r) {
      super(c, x, y);
      c1 = new Circle(c, x, y, r);
      r1 = new Rect(d, x, y, r*4, r*3);
    }
    public void draw(Graphics g) {
      r1.draw(g); c1.draw(g);
    }
  }
```

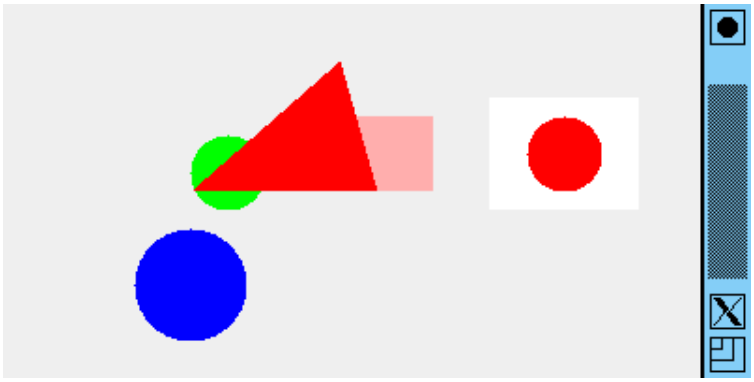

図 2: 例解の実行結果

この例解からも、継承の問題点が見て取れます。三角形について言えば、3 つの頂点は平等だと思われるのに、親クラス Figure で 1 頂点ぶんの変数が予め用意されているため、残り 2 つを別扱いしています。旗についてはもっと問題で、2 つの要素図形 (円と長方形) を持っているので親クラスで定義した変数は全く使われなくなっていますが、定義されている以上は無くなるわけにはいかず、さらにコンストラクタを呼んで初期設定しておくことも強制されます。「実装の共有がくっついている」ことの問題点がお分かりいただけますね?

4 いかにして機能拡張するか/それを組み合わせるか?

次の題材として、先の絵を動かしてみましょう。初期設定のコンストラクタ内に「各オブジェクトに対して一定時間間隔でメソッド setTime() を呼び出す」処理を追加し、また画面書き換えのための対処 (setOpaque() と repaint()) も追加しています。

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

```
import java.util.*;

public class Sample12 extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  long baset = System.currentTimeMillis();
  public Sample12() {
    figs.add(new Circle(Color.blue, 100, 150, 30));
    figs.add(new Circle(Color.green, 120, 90, 20));
    figs.add(new Rect(Color.pink, 200, 80, 60, 40));
    setOpaque(false);
    new javax.swing.Timer(50, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double t = (System.currentTimeMillis() - baset)*0.001;
        for(Figure f: figs) { f.setTime(t); }
        repaint();
      }
    }).start();
  }
  public void paintComponent(Graphics g) {
    for(Figure f: figs) { f.draw(g); }
  }

  static abstract class Figure {
    Color col;
    int xpos, ypos, xbase, ybase;
    public Figure(Color c, int x, int y) {
      col = c; xpos = xbase = x; ypos = ybase = y;
    }
    public abstract void draw(Graphics g);
    public void setTime(double t) {
      xpos = xbase + (int)(50*Math.sin(t));
      ypos = ybase + (int)(50*Math.cos(t));
    }
  }
  static class Circle extends Figure {
    int rad;
    public Circle(Color c, int x, int y, int r) {
      super(c, x, y); rad = r;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
  }
  static class Rect extends Figure {
    int width, height;
    public Rect(Color c, int x, int y, int w, int h) {
```

```
      super(c, x, y); width = w; height = h;
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect(xpos-width/2, ypos-height/2, width, height);
    }
  }

  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample12());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

あとは各オブジェクトでどのように setTime() を扱うかです。このコードでは、一番上のクラス Figure に setTime() を追加し、円運動ができるようにしています。

それはいいのですが、これだとすべての図形が同じ運動になりますよね。もちろん、図形ごとにさまざまな運動をさせたいわけですが、そのためにはここからどのように直したらいいと思いますか?

- 問: 「図形の種類ごとに」異る動き方をするには、どうしたらいいか?
- 問: 「個々の図形ごとに」異る動き方をするには、どうしたらいいか?
- 問: 「1 つの図形でも時点によって」異る動き方をするには、どうしたらいいか?

ここで、「サブクラスを作ってそこでオーバライドする」とうい答えはあり得ますがあまりよろしくないです。それをやると、「周回する円」「ジグザグに動く円」など山のようにクラスができてしまいます (組み合わせ爆発) し、実行途中に動き方を変えることなど望めません。

そこで、次のように「動き方」という部分を別のオブジェクトに分離してみます。

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

public class Sample12a extends JPanel {
  ArrayList<Figure> figs = new ArrayList<Figure>();
  long baset = System.currentTimeMillis();
  int count = 0;
  public Sample12a() {
    figs.add(new Circle(Color.blue, 100, 150, 30));
    figs.add(new Circle(Color.green, 120, 90, 20));
    figs.add(new Rect(Color.pink, 200, 80, 60, 40));
    figs.get(0).setMover(new CircleMover(100, 100, 5, 0.5, 80));
    figs.get(1).setMover(new SawMover(40, 200, 280, 30, 2.5));
    setOpaque(false);
    new javax.swing.Timer(50, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
```

```
      double t = (System.currentTimeMillis() - baset)*0.001;
      for(Figure f: figs) { f.setTime(t); }
      repaint();
    }
  }).start();
  addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent evt) {
      if(++count % 2 == 0) {
        figs.get(2).setMover(new CircleMover(200,50,50,0.7,40));
      } else {
        figs.get(2).setMover(new SawMover(200,50,100,100,4));
      }
    }
  });
}
public void paintComponent(Graphics g) {
  for(Figure f: figs) { f.draw(g); }
}

static abstract class Figure {
  Color col;
  int xpos, ypos;
  Mover mv = null;
  public Figure(Color c, int x, int y) {
    col = c; xpos = x; ypos = y;
  }
  public abstract void draw(Graphics g);
  public void setMover(Mover m) { mv = m; }
  public void setTime(double t) {
    if(mv == null) { return; }
    mv.setTime(t); xpos = (int)(mv.getX()); ypos = (int)(mv.getY());
  }
}
static class Circle extends Figure {
  int rad;
  public Circle(Color c, int x, int y, int r) {
    super(c, x, y); rad = r;
  }
  public void draw(Graphics g) {
    g.setColor(col);
    g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
  }
}
static class Rect extends Figure {
  int width, height;
  public Rect(Color c, int x, int y, int w, int h) {
    super(c, x, y); width = w; height = h;
```

```
    }
    public void draw(Graphics g) {
      g.setColor(col);
      g.fillRect(xpos-width/2, ypos-height/2, width, height);
    }
  }
  interface Mover {
    public void setTime(double t);
    public double getX();
    public double getY();
  }
  static class CircleMover implements Mover {
    double time, xbase, ybase, theta, freq, rad;
    public CircleMover(double x, double y, double t, double f, double r) {
      xbase = x; ybase = y; theta = t; freq = f*2*Math.PI; rad = r;
    }
    public void setTime(double t) { time = t; }
    public double getX() { return xbase+rad*Math.cos(freq*(time + theta)); }
    public double getY() { return ybase+rad*Math.sin(freq*(time + theta)); }
  }
  static class SawMover implements Mover {
    double time, xbase, ybase, dx, dy, freq;
    public SawMover(double x0, double y0, double x1, double y1, double f) {
      xbase = x0; ybase = y0; dx = x1-x0; dy = y1-y0; freq = f;
    }
    public void setTime(double t) { time = t; }
    public double getX() { return xbase + dx*((time % freq)/freq); }
    public double getY() { return ybase + dy*((time % freq)/freq); }
  }

  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample12a());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

こうすれば、「図形」と「動き方」は独立ですから、どのようにでも組み合わせられますし、実行途中で切り替えることもできます (実際、長方形の動きはクリックするごとに変化します)。そして、「どちらの動き方でも入れられる」という部分は総称性と動的分配によって実装されているわけです (今回はインタフェースを使っています)。

このように、オブジェクト指向言語の機能を活かすということは現在では、(継承による差し替えよりも) 複数のオブジェクトをうまく組み合わせることが大切という意識になっています。そうすると、具体的にどのような場合にはどのように組み合わせるのがよいかというノウハウが大切になります。これを「デザインパターン」として切り出す、というのがデザインパターン活動のもともとの始まりです。たとえば、上で示したような組み合わせ方は Strategy パターンとして知られています。つ

まり「動き方の戦略を独立したオブジェクトとしして保持し、図形ごとにどの戦略にするかを設定する」ということですね (図**??**)。

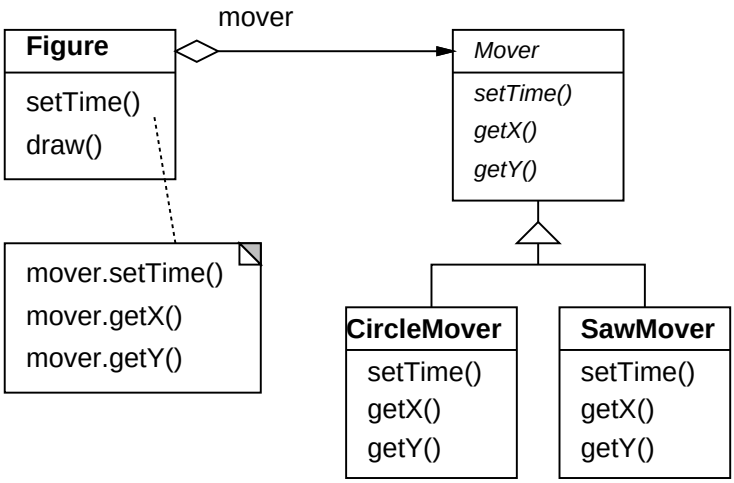

図 3: Strategy パターン

演習 1-2 上のプログラムを動かしなさい。動いたら、次のように改造してみなさい。

a. もっと図形を増やして、動き方を自分で設定してみなさい。
b. 新たな動き方を追加してみなさい。たとえば、等速直線運動で動いていって窓の端で跳ね返るなど。
c. 「のこぎり運動しながら円周運動する」みたいなものはどうしたら作れるか考えてみなさい。

5 例解および補足

実際に「等速直線運動」と「動きの合成」を作ってみました。要点だけを示します。

```
figs.get(0).setMover(new FlyMover(100, 100, 55, 30));
figs.get(2).setMover(
   new AdditiveMover(new CircleMover(200,50,50,0.7,40),
                     new SawMover(0,0,150,230,4)));
```

飛ぶ方は単に 1 つの新しい `Mover` を実装しているだけですが、組み合せの方は「2 つの `Mover` を受け取って合成する」というメタな `Mover` を作っています。ちょうど「数と数を足したものもまた数である」のと同じように、オブジェクトを代数 (?) にすることで柔軟な組合せを可能にするわけです。実装も見てみましょう。`AdditiveMover` はあっけないほど簡単です。

```
static class AdditiveMover implements Mover {
  Mover mv1, mv2;
  public AdditiveMover(Mover m1, Mover m2) { mv1 = m1; mv2 = m2; }
  public void setTime(double t) { mv1.setTime(t); mv2.setTime(t); }
  public double getX() { return mv1.getX() + mv2.getX(); }
  public double getY() { return mv1.getY() + mv2.getY(); }
}
class FlyMover implements Mover {
```

```
    double lastt, xpos, ypos, vx, vy;
    public FlyMover(double x0, double y0, double vx1, double vy1) {
      xpos = x0; ypos = y0; vx = vx1; vy = vy1;
    }
    public void setTime(double t) {
      double dt = t - lastt; lastt = t;
      xpos += vx*dt; ypos += vy*dt;
      if(xpos < 0 && vx < 0 || xpos > getWidth() && vx > 0) { vx *= -1; }
      if(ypos < 0 && vy < 0 || ypos > getHeight() && vy > 0) { vy *= -1; }
    }
    public double getX() { return xpos; }
    public double getY() { return ypos; }
  }
```

FlyMover はクラスに static 指定がついていません。このクラスは外側の JPanel オブジェクトのメソッド getWidth()、getHeight() を呼ぶ必要があるためそうなっているのですが、外側オブジェクトの「全てを」図形側からアクセスされたら良くないですよね。これに対し、Stragegy パターンを使うと「あまり広めたくない情報」にアクセスできるものを Strategy オブジェクトに限定できるわけです。

6 フレームワーク的考え方

ここまでは、「プログラムすることは部品オブジェクトを用意して組み合わせることである」という考え方でやって来ました。しかし現在のソフトウェアは非常に大規模で複雑化しているため、その大量の部品を正しく組み合わせることも容易ではありません。

そこで視点を逆転して既に大枠が出来ているプログラムの中に一部だけ「自分が行いたいこと」を作ってはめ込むことで自分の作りたいプログラムを完成させる、ということが行われるようになってきました。これをアプリケーションフレームワークと言います。

少し長いですが、サンプルを見てみましょう。

```
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 Sample13 extends JPanel {
  Scene cur = new Scene1();
  long tm0 = System.currentTimeMillis();
  public Sample13() {
    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();
  }
  public void paintComponent(Graphics g) { cur.draw(g); }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample13());
    app.setSize(400, 240);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
```

このプログラムでは、Scene というクラスが 1 つの「場面」を表していて、その場面における描画と時間経過とマウスクリックと次の場面への移行をすべて処理します (そして、最初の場面はそのサブクラスである Scene1 です)。

まず Scene を見てみましょう。

```
  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; }
  }
```

figs はこれまでやってきたように図形の集まりで、これに対して anim は時間経過につれて絵を変化させるためのオブジェクト (Animation インタフェースに従います) の集まりです (先の例題ではこのようなオブジェクトは図形の中に入れていましたが、それだと複数持たせられないのでここでは外に出しています)。そして、描画は各図形を描画し、時間進行は各 Animation オブジェクトの時間を進めるだけです。具体的なことが何も書いてないのでびっくりしますが、要はこのクラスのサブクラスを作ってはめ込むことで個々のシーンができるわけです。

では、最初の場面を見てみます。

```
  static class Scene1 extends Scene {
```

```
  Rect r1 = new Rect(Color.YELLOW, 100, 100, 100, 100);
  Rect r2 = new Rect(Color.YELLOW, 250, 100, 100, 100);
  public Scene1() {
    Picture p1 = new Picture("kuno1.png", 0, 0);
    Picture p2 = new Picture("sun1.png", 0, 0);
    figs.add(r1); figs.add(r2); figs.add(p1); figs.add(p2);
    anim.add(new SawMove(p1, 1f, 95, 100, 105, 100));
    anim.add(new SawMove(p2, 1f, 250, 95, 250, 105));
    figs.add(new Text(20, 25, "Click your favorite.",
                      new Font("serif", Font.BOLD, 18)));
  }
  public void press(int x, int y) {
    if(r1.hit(x, y)) {
      next = new Scene2(); ended = true;
    } else if(r2.hit(x, y)) {
      next = new Scene4(); ended = true;
    }
  }
}
```

2 つの長方形と 2 つの絵を配置し、絵は振動させ、クリック位置に応じて別の場面に移動します。

場面 2 はもっと簡単です。

```
static class Scene2 extends Scene {
  public Scene2() {
    next = new Scene3();
    figs.add(new Text(20, 90, "Watch for our movie.",
                      new Font("serif", Font.BOLD, 18)));
  }
  public void press(int x, int y) { ended = true; }
  public void setTime(float t) { ended |= (t > 5); }
}
```

単に決まった時間だけ文字を表示するだけです。

場面 3 はアニメーション作品になります。地面、空、太陽/月、ピラミッドを図形で用意し、位置や色を時間とともに遷移させます。

```
static class Scene3 extends Scene {
  public Scene3() {
    next = new Scene1();
    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));
  }
  public void setTime(float t) {
    super.setTime(t); ended |= (t > 14);
  }
}
```

場面 4 はゲームです。これは円を動かし、クリックが当たったらその時間に応じてメッセージを出すだけです。

```
static class Scene4 extends Scene {
  Font fn = new Font("serif", Font.BOLD, 18);
  Text t1 = new Text(20, 25, "Click onto the circle.", 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 SawMove(c1, 0.7f, 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)); }
  }
}
```

ここから先は既に知っているようなものばかりです。継承の弱点をさらに限定するため、Figure はインタフェースにして、それを実装する抽象クラス SimpleFigure に図形の共通動作を集めました。

```
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);
}
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); }
}
```

円や矩形や三角形はこれまで見て来た通りです。

```
static class Circle extends SimpleFigure {
  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);
  }
}
static class Rect extends SimpleFigure {
  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 {
  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);
```

```
  }
}
```

文字や絵は新しく追加しています。

```
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);
  }
}
```

直線的動作、時間につれての色変化、ジグザグが基本的な変化です。

```
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) {
      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)));
    }
  }
  static class SawMove implements Animation {
    Figure fig;
    float time1, xpos1, ypos1, xpos2, ypos2;
    public SawMove(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);
    }
  }
```

そして、時間限定で動かす、時間限定で現れる、などの機能も用意しました。時間限定で現れる方は、中に現れたり消えたりする図形を保持するので、Figure インタフェースも実装しています。

```
  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); }
    }
  }
}
```

このように、基本となる部品と、土台となるフレームワークを用意することで、ほんの少しのプログラムコードだけで「自分がやりたいこと」を表現することが可能になります。ただし、そのためには「用意されているもの」をきちんと知らないといけない、という部分があり、これはどのようなプログラミングモデルでも変わらないものと思います。

演習 1-3 上の例題をそのまま動かしなさい。動いたら、どれかの場面を変更したり、新しい場面を追加してみなさい。