Java による UI プログラミング: MICS 実験 2-19-1c
＃06 新しいインタフェースの試み

久野　靖 *

2019.10.23

1　新しいインタフェースの必要性と試み

ここまでに Java 言語を用いたユーザインタフェースの実装をさまざまに行って来ましたが、この実験の最終的なテーマは「新しいインタフェースを作ってみる」ことでした。その必要性について考えて見ましょう。

今日の GUI の基本は 1970 年代に Xerox が Alto を作り、それを見た Apple 社の Steve Jobs がそれを参考に Apple Lisa、Macintosh を開発して売り出したことに始まります。つまりもう 40 年以上前のことですね。

それ以来、コンピュータのハードウェア性能は飛躍的に向上し、またタッチパネルのような以前なかった入出力装置も当り前になっています。それでいながら、「ウィンドウ、メニュー、アイコン、ポインティングデバイス」(WIMP) インタフェースという基本は変わらないままです。これは本当はおかしなことです。WIMP インタフェースはそれこそ初代 Mac(メモリは 128KB しかありませんでした) のように「しょぼい CPU、しょぼい画面」でも動作することを目標につくり出されたわけですから。

もちろん、ユーザインタフェースの研究者は日々さまざまな工夫やアイデアを思い付き研究していますが、それでも「これまでと画期的に違う」ことが普及する感じはあまりしません。

これを打破するには、本学のようにソフトウェアを一通り学ぶ学生さんたちが「普段は一般的な GUI を使っているにせよ、やりたいことによってはもっと別のインタフェースの方がずっと望ましい場合もあることを認識し、ある程度までは自分で作れる」ようになることが大切なのかなと思います。

そういうわけでこの授業では、GUI 部品はそこそこにして「基本的なメカニズムや人間の特性」に焦点をあて、必要なら「GUI 部品でないインタフェース」を作れるように練習してきたわけです。

今回は久野が考案した「これまでのインタフェースにあまりない」機能を 2 つほどお見せします。最終発表で皆様には「独自のインタフェース」というテーマをお願いしていますが、今回お見せするものを利用していただいても、もっと前の回の何かを利用していただいても、まったく別のものを作られても、なんでも構いません。とにかく「色々やってみようよ」ということがテーマだと思っていただければ結構です。

2　「くっつき」ドラグ

これまで画面上に色々なものを配置してきましたが、そのとき個々の「もの」はバラバラであり、多数のものを配置したいと持ったときにはそれらを「1 つずつ順番に」動かすことが基本になっていました。しかしそれでは、多数のものを扱うときに、明らかに不便ですね。

これまでの GUI のパラダイムでは、そのような「まとめて」をやりたいときは、その複数のものを選択した上で「グループ化」して操作するのが普通です。もうまとめて扱いたくない状態になったら、こんどは「グループ解除」をおこないます。

*電気通信大学

確かにこの方法で「まとめて」扱うことはできますが、グループ化したり解除する操作はけっこう心理的負担が大きく繁雑です。また、グループ化した状態ではその中の個々のものにはアクセスできないので、解除とグループ化をいったり来たりすることになり、それも面倒です。

私達の日常ではどうでしょう。グループ化したり解除するのは「ひもで縛ってまとめる」「ひもを解く」ことに対応しますが、繁雑ですね。むしろ、「磁石でくっつく」とかポストイットみたいに「粘着でくっつく」とかが便利で多く使われていると思いますがどうでしょう。

そこで、次にお見せするデモプログラムでは、短冊状のものが画面に現れていますが、それらは「隣どうしにあるときには」ある意味「くっついて」いて一緒に動かせます。このためには「ぴったり」隣にある必要があるので、短冊の配置時にはグリッドに沿うようにすることで「ぴったり」を実現しています (図 1)。

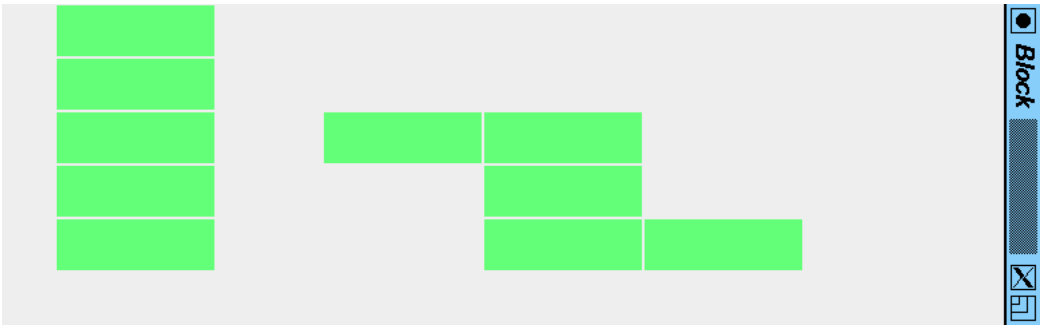

図 1: グループでドラグするデモ

また、くっつけたり離したり自由にしたいので、ある短冊を「持った」ときには、そこから「下/右につながっているもの」がつながっているものとして一緒に動きます。こうすれば、並んでいるもののどこを「持つ」かによって好きな場所で切り離したりくっつけたりできるからです。

ではコードを見ていただきます。冒頭部分は `Figure` オブジェクトの集まりを画面に表示するという基本に沿っていますが、さらにくっついたままドラグする対象を `DragSet` というオブジェクトに入れて扱うことにしたので、それもインスタンス変数 `dset` としています。あと、画面上のグリッドを 40 ピクセル単位としています (この後、半端な場所には置けないようにコードを書いています)。

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

public class Sam61 extends JPanel {
  static final int grid = 40;
  ArrayList<Figure> fset = new ArrayList<Figure>();
  DragSet dset = new DragSet();
```

コンストラクタですが、ここでは「短冊」を 10 個あらかじめ画面に置いておきます。これを後でドラグするわけです。マウスボタンが押されたときはこれまで通りに当たっているものを探し、あったときはそれを `dset` に入れてから「くっついているものを集め」ておきます。ボタンが離されたらすべて開放します。そして、ドラグするときは `dset` を通じてドラグすることで、「くっついているものをまとめて」ドラグするわけです。

```
  public Sam61() {
    for(int i = 0; i < 10; ++i) {
```

```
      Box b = new Box(40, 40*i, 120, 40); fset.add(b);
    }
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        int x = evt.getX(), y = evt.getY();
        Figure f = null;
        for(Figure g: fset) { if(g.hit(x, y)) { f = g; } }
        if(f == null) { return; }
        dset.add(f, x, y); hitCollect(x, y);
      }
      public void mouseReleased(MouseEvent evt) {
        dset.release(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        dset.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
  }
```

paintComponent() はこれまでと変わりません。次の hitCollect が今回の「きも」で、dset に 1 つ以上のものが入っている時に呼ばれると「それの右/下に隣接しているもの」を次々に追加します。外側ループは「全部しらべて 1 回も追加が起きなければそれ以上追加されることはないので終る」という形でフラグを使っています。

```
  public void paintComponent(Graphics g) {
    for(Figure f: fset) { f.draw(g); }
  }
  void hitCollect(int x1, int y1) {
    boolean done = false;
    while(!done) {
      done = true;
      for(Figure f: fset) {
        if(dset.isIn(f)) { continue; }
        for(Figure d: dset) {
          if(f.getX() == d.getX() && f.getY() == d.getY()+d.getH()) {
            dset.add(f, x1, y1); done = false;
          } else if(f.getY() == d.getY() && f.getX() == d.getX()+d.getW()) {
            dset.add(f, x1, y1); done = false;
          }
        }
      }
    }
  }
```

図形ですが、Figure にはドラグ中かどうかの情報を持たせるようにしています。さらに左上隅の

座標と幅/高さを持たせ、これらを取得できるようにしています。またグリッドに沿った位置にくっつくメソッドも用意しています。長方形オブジェクトはそのサブクラスで、当たっているかどうかの判定と描画 (中を塗るだけ、境界が分からなくならないように 1 ピクセル周辺を残しています) します。

```
static abstract class Figure {
  boolean drag;
  int x, y, w, h;
  public boolean getDrag() { return drag; }
  public void setDrag(boolean b) { drag = b; }
  public int getX() { return x; }
  public int getY() { return y; }
  public int getW() { return w; }
  public int getH() { return h; }
  public void moveTo(int x1, int y1) { x = x1; y = y1; }
  public boolean hit(int x, int y) { return false; }
  public void snap() {
    int xd = x % grid, yd = y % grid; x -= xd; y -= yd;
    if(xd > grid/2) { x += grid; }
    if(yd > grid/2) { y += grid; }
  }
  public abstract void draw(Graphics g);
}
class Box extends Figure {
  Color col = new Color(100, 255, 120);
  public Box(int x1, int y1, int w1, int h1) { x=x1; y=y1; w=w1; h=h1; }
  public boolean hit(int x1, int y1) {
    return x <= x1 && x1 <= x+w && y <= y1 && y1 <= y+h;
  }
  public void draw(Graphics g) {
    g.setColor(col); g.fillRect(x+1, y+1, w-2, h-2);
  }
}
```

最後に DragSet が大物なのですが、これは ArrayList<Figure>の機能も一緒に持たせたためです。なぜそうしているかというと、ArrayList では foreach ループの中で要素を追加できないようになっているので、それを許すものが欲しかったためです (一般にはループ中で増やされるとバグの原因になりますが、ここではそれが問題になるような使い方をしていないので大丈夫です)。

```
static class DragSet implements Iterable<Figure> {
  static final int max = 200;
  Figure[] figs = new Figure[max];
  int[] dx = new int[max], dy = new int[max];
  int size = 0;
  public void add(Figure f, int x1, int y1) {
    if(size+1 >= max) { return; }
    figs[size] = f; dx[size] = f.getX()-x1; dy[size] = f.getY()-y1;
    f.setDrag(true); ++size;
  }
```

```
    public void release() {
      for(int i = 0; i < size; ++i) {
        Figure f = figs[i]; f.setDrag(false); f.snap();
      }
      size = 0;
    }
    public void moveTo(int x, int y) {
      for(int i = 0; i < size; ++i) { figs[i].moveTo(x+dx[i], y+dy[i]); }
    }
    public boolean isIn(Figure f) {
      for(int i = 0; i < size; ++i) { if(figs[i] == f) { return true; } }
      return false;
    }
    public Iterator<Figure> iterator() {
      return new Iterator<Figure>() {
        int i = 0;
        public boolean hasNext() { return i < size; }
        public Figure next() { return figs[i++]; }
        public void remove() { }
      };
    }
  }
```

一番最後の内部クラス `iterator()` が foreach ループのためのイテレータオブジェクトを返すメソッドで、ここでは無名内部クラスのインスタンスを返しています。

最後に `main` はこれまでと変更していません。

```
  public static void main(String[] args) {
    JFrame app = new JFrame("Block");
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.add(new Sam61()); app.setSize(800, 600); app.setVisible(true);
  }
}
```

演習 1 例題をそのまま動かして動作を観察しなさい。納得したら次のことをやってみなさい。

a. 短冊の幅や高さをさまざまなものにして動作させてみる (色も変えてみてよい)。幅や高さが `grid` の倍数でないとくっつかないので注意。

b. 短冊の中に文字や画像をいれて意味のある動作をさせる。たとえば「俳句を組み立てる」とか「間取り図を作る」とか。

c. 短冊のくっつき方を「縦だけくっつく」「横だけくっつく」「ずれていてもくっつく」のように変更してみなさい。

d. 水道管ゲームのように「特定の場合にだけくっつく」ように変更してみなさい。

e. 短冊がずっと置いてあるのでなく、「取り出すところ」からドラグし始めると現れるようにしてみなさい。

f. その他、くっつきを使って面白いプログラムを作りなさい。

3　補足: 文字や画像の表示

演習 1 に出て来る「文字の表示」や「画像の表示」そして「取り出すと新たなのが現れる」をどのように実現するか、例を示しましょう (図 2)。これを先にやれと言われそうですが、こういう機能は別に目新しいものではないので、あくまでも補足でありおまけということです。

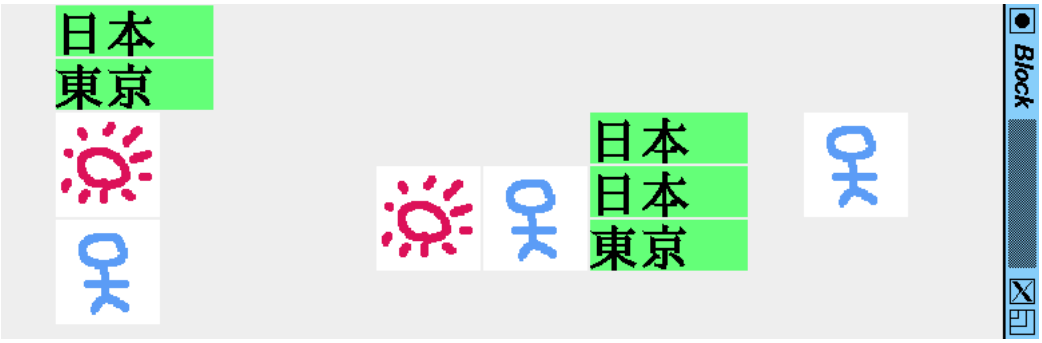

図 2: 文字や画像を表示するブロック

まず、import がさらに増えています。画像の読み込みのためのものが大半です。インスタンス変数は今度はグループでドラグしないので、`fset` のほかは以前やった `target` のみでやります。あと、クラス変数としてこのプログラムで使用する画像を保持する場所を用意しました。その値は `main` から直接書き込みます。

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

public class Sam62 extends JPanel {
  static final int grid = 40;
  static BufferedImage sun, human;
  ArrayList<Figure> fset = new ArrayList<Figure>();
  Figure target = null;
```

文字の入った矩形と画像の入った矩形が現れて来る場所を用意します。その中身は後で読みます。ともかく、以前やったと同じようにしてマウスボタン押し下げ時に `target` を取得しますが、ただしこんどはその取得できたものが `Genetator` インタフェースを実装している図形であれば、`generate()` というメソッドを読んで新たな図形を作り、それをドラグ開始します (新たに作ったので `fset` への追加も必要)。ドラグやボタンを離したときの動作は以前のように簡単な版になっていますが、離した時は `snap()` でグリッドに合わせます。

```
  public Sam62() {
    fset.add(new StrGenerator(40, 0, 120, 40, "日本"));
    fset.add(new StrGenerator(40, 40, 120, 40, "東京"));
    fset.add(new ImgGenerator(40, 80, 80, 80, sun));
    fset.add(new ImgGenerator(40, 160, 80, 80, human));
```

```
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        int x = evt.getX(), y = evt.getY();
        target = null;
        for(Figure f: fset) { if(f.hit(x, y)) { target = f; } }
        if(target == null || !(target instanceof Generator)) { return; }
        target = ((Generator)target).generate(x, y);
        fset.add(target);
      }
      public void mouseReleased(MouseEvent evt) {
        if(target == null) { return; }
        target.snap(); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        if(target == null) { return; }
        target.moveTo(evt.getX(), evt.getY()); repaint();
      }
    });
  }
```

paintComponent、Figure、Boxは変更していません。

```
  public void paintComponent(Graphics g) {
    for(Figure f: fset) { f.draw(g); }
  }
  static abstract class Figure {
    boolean drag;
    int x, y, w, h;
    public boolean getDrag() { return drag; }
    public void setDrag(boolean b) { drag = b; }
    public int getX() { return x; }
    public int getY() { return y; }
    public int getW() { return w; }
    public int getH() { return h; }
    public void moveTo(int x1, int y1) { x = x1; y = y1; }
    public boolean hit(int x, int y) { return false; }
    public void snap() {
      int xd = x % grid, yd = y % grid; x -= xd; y -= yd;
      if(xd > grid/2) { x += grid; }
      if(yd > grid/2) { y += grid; }
    }
    public abstract void draw(Graphics g);
  }
  class Box extends Figure {
```

```
  Color col = new Color(100, 255, 120);
  public Box(int x1, int y1, int w1, int h1) { x=x1; y=y1; w=w1; h=h1; }
  public boolean hit(int x1, int y1) {
    return x <= x1 && x1 <= x+w && y <= y1 && y1 <= y+h;
  }
  public void draw(Graphics g) {
    g.setColor(col); g.fillRect(x+1, y+1, w-2, h-2);
  }
}
```

さていよいよ、文字列や画像を表示する Box ですが、これらは Box のサブクラスとして作り、最小限の拡張をおこないます。文字列の方は文字列とフォントをインスタンス変数として持ち、draw() で文字列を描画します。画像の方は BufferedImage オブジェクトを保持し、それを描画します。

```
class StrBox extends Box {
  String str;
  Font fn = new Font("Serif", Font.BOLD, 36);
  public StrBox(int x, int y, int w, int h, String s) {
    super(x, y, w, h); str = s;
  }
  public void draw(Graphics g) {
    super.draw(g); g.setColor(Color.BLACK); g.setFont(fn);
    g.drawString(str, x, y+36);
  }
}
class ImgBox extends Box {
  BufferedImage img;
  public ImgBox(int x, int y, int w, int h, BufferedImage i) {
    super(x, y, w, h); img = i;
  }
  public void draw(Graphics g) {
    g.drawImage(img, x+1, y+1, w-2, h-2, null);
  }
}
```

あとは Generator ですが、その中身は genearte() を持つということだけです。それらは StrBox や ImgBox を継承し、見ためはすべてこれらにお願いしたうえで、generate は自分が持っている文字列や画像を持つ StrBox や ImgBox を作って返します。

```
interface Generator {
  public Figure generate(int x, int y);
}
class StrGenerator extends StrBox implements Generator {
  public StrGenerator(int x, int y, int w, int h, String s) {
    super(x, y, w, h, s);
  }
  public Figure generate(int x1, int y1) {
    return new StrBox(x1, y1, w, h, str);
```

```
    }
  }
  class ImgGenerator extends ImgBox implements Generator {
    public ImgGenerator(int x, int y, int w, int h, BufferedImage i) {
      super(x, y, w, h, i);
    }
    public Figure generate(int x1, int y1) {
      return new ImgBox(x1, y1, w, h, img);
    }
  }
```

最後に main ですが、クラス変数に画像を読み込むのをここで行います。画像ファイルが無ければ例外を出して終わるようにしています。それ以外は変わっていません。

```
  public static void main(String[] args) throws Exception {
    sun = ImageIO.read(new File("sun.png"));
    human = ImageIO.read(new File("human.png"));
    JFrame app = new JFrame("Block");
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.add(new Sam62()); app.setSize(800, 600); app.setVisible(true);
  }
}
```

4 「投げる」インタフェース

ユーザがさまざまなインタフェースを操作するときに掛かる時間は人間の認知特性によって制約されている (その結果 Fitzz の法則のようなものが現れて来る) ことは既に学びました。ただ、その限界はインタフェースの工夫で変えられる可能性があります。

たとえばドラグ&ドロップを考えてみましょう。アイコンを選択し、ドラグして行き先のアイコンに重ねてドロップします。しかしその操作を全部やらなくても、ドラグしている途中では「行き先に向かっている」わけですから、そこでマウスボタンを離してもそのまま「同じ方向に動き続けて」行けば行き先にたどりつくように思えます。これを「アイコン投げ」と呼んでいます。

アイコン投げでは操作は全部やらずに途中でやめてしまうわけですから、通常のドラグ&ドロップより短時間でできます。そのかわり、投げる方向が正確でなければ正しく行き先につきません。つまりこれは「熟練が必要だが、熟練したときは高速にできる」ようなインタフェースなわけです。

今回はこれを先のグループドラグのインタフェースに追加してみます。基本的な考え方は次の通りです。

- ドラグしているとき、ドラグ位置の XY 座標に加えて時刻を供給することで、ドラグされている物体が自分の「速度」を計算できるようにする。
- マウスボタンを離した後はその「速度」に従ってそれぞれのものが動くようにする。動かすこと自体は前にやったアニメーション機能と同様にして行う。

では実際に見てみましょう。Mover を使い、時間を計測するためにインスタンス変数 baseTime と time を増やすのは前にやりました。

```
import java.util.*;
import java.awt.*;
import java.awt.geom.*;
```

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

public class Sam63 extends JPanel {
  static final int grid = 40;
  ArrayList<Figure> fset = new ArrayList<Figure>();
  DragSet dset = new DragSet();
  ArrayList<Mover> mset = new ArrayList<Mover>();
  long baseTime = System.currentTimeMillis();
  double time = 0.0;
```

コンストラクタですが、これまとまず違うところは、各図形を fset だけでなく mset にも登録することと、ドラグ時とマウスボタン離し時に時刻を渡すところです。これによって時間にかかわる情報を各図形が受け取り動作できるようになります。タイマーで定期的に Mover を動かすところは同じです。

```
  public Sam63() {
    for(int i = 0; i < 10; ++i) {
      Box b = new Box(40, 40*i, 120, 40); fset.add(b); mset.add(b);
    }
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        int x = evt.getX(), y = evt.getY();
        Figure f = null;
        for(Figure g: fset) { if(g.hit(x, y)) { f = g; } }
        if(f == null) { return; }
        dset.add(f, x, y); hitCollect(x, y);
      }
      public void mouseReleased(MouseEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        dset.release(time); repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        dset.moveTo(evt.getX(), evt.getY(), time); repaint();
      }
    });
    new javax.swing.Timer(30, new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        time = 0.001 * (System.currentTimeMillis()-baseTime);
        for(Mover m: mset) { m.setTime(time); }
        repaint();
      }
    }).start();
```

```
  }
```

paintComponent と hitCollect については変更していません。Figure も変更していません。

```
  public void paintComponent(Graphics g) {
    for(Figure f: fset) { f.draw(g); }
  }
  void hitCollect(int x1, int y1) {
    boolean done = false;
    while(!done) {
      done = true;
      for(Figure f: fset) {
        if(dset.isIn(f)) { continue; }
        for(Figure d: dset) {
          if(f.getX() == d.getX() && f.getY() == d.getY()+d.getH()) {
            dset.add(f, x1, y1); done = false;
          } else if(f.getY() == d.getY() && f.getX() == d.getX()+d.getW()) {
            dset.add(f, x1, y1); done = false;
          }
        }
      }
    }
  }
  static abstract class Figure {
    boolean drag;
    int x, y, w, h;
    public boolean getDrag() { return drag; }
    public void setDrag(boolean b) { drag = b; }
    public int getX() { return x; }
    public int getY() { return y; }
    public int getW() { return w; }
    public int getH() { return h; }
    public void moveTo(int x1, int y1, double t) { x = x1; y = y1; }
    public boolean hit(int x, int y) { return false; }
    public void snap() {
      int xd = x % grid, yd = y % grid; x -= xd; y -= yd;
      if(xd > grid/2) { x += grid; }
      if(yd > grid/2) { y += grid; }
    }
    public abstract void draw(Graphics g);
  }
```

Mover のインタフェースは前にやったときと同じです。そして動く機能を実装する MovingFigure が今回の機能の中心となります。まず、動きを持つため、インスタンス変数 vx と vy を追加します。また、最後に位置を更新した時刻を lastt に記憶します。そのうえで、moveTo() で時刻を受け取り、先の時刻との差分に基づいて XY 方向の速度をそれぞれ求めますが、いきなり変化するのでなく少しなめらかにするため、これまでの値と新しい値の平均を取るようにしています。

```
interface Mover {
  public void setTime(double t);
  public Figure getFigure();
}
abstract class MovingFigure extends Figure implements Mover {
  double lastt = 0.0, vx = 0.0, vy = 0.0;
  public void moveTo(int x1, int y1, double t) {
    int dx = x1-x, dy = y1-y; x = x1; y = y1;
    if(lastt == 0.0) { lastt = t; return; }
    double dt = t - lastt; lastt = t;
    vx = 0.5*vx + 0.5*dx/dt; vy = 0.5*vy + 0.5*dy/dt;
  }
  public void setTime(double t) {
    if(lastt == 0.0) { lastt = t; return; }
    if(drag || vx == 0.0 && vy == 0.0) { return; }
    double dt = t - lastt; lastt = t;
    x += vx*dt; y += vy*dt; lastt = t;
    if(vx < 0 && x < 0 || vx > 0 && x+w > getWidth() ||
       vy < 0 && y < 0 || vy > 0 && y+h > getHeight()) {
      if(x < 0) { x = 0; }
      if(x+w > getWidth()) { x = getWidth()-w; }
      if(y < 0) { y = 0; }
      if(y+h > getHeight()) { y = getHeight()-h; }
      vx = vy = 0.0; snap();
    }
  }
  public void release(double t) {
     double dt = t-lastt; lastt = t;
     if(dt > 0.2) { vx = vy = 0.0; }
  }
  public Figure getFigure() { return this; }
}
```

マウスボタンを離したあとは setTime() により位置を移動します。ここでは XY 方向の速度を用いて位置を変更しますが、窓の端から出てしまいそうなときはそこで止まるようにしています。止まるときは XY 方向の速度を 0 にしてから snap() によりグリッドに合わせます。さらに、release() はマウスボタンが離されたときに時刻とともに呼ばれますが、ボタン離し時に直前のイベントから 0.2 秒以上経過していたら「いちど止まってから離した」と見なして XY 方向の速度を 0 にしています (そのためだけにこのメソッドを追加しました)。

Box についてはとくに変更していません。

```
class Box extends MovingFigure {
  Color col = new Color(100, 255, 120);
  public Box(int x1, int y1, int w1, int h1) { x=x1; y=y1; w=w1; h=h1; }
  public boolean hit(int x1, int y1) {
    return x <= x1 && x1 <= x+w && y <= y1 && y1 <= y+h;
  }
```

```
    public void draw(Graphics g) {
      g.setColor(col); g.fillRect(x+1, y+1, w-2, h-2);
    }
  }
```

DragSet では、moveTo で時刻を渡すようにしたところと、release のときに MovingFigure である場合は時刻を渡すようにしたところです。あとは変更はありません。

```
  static class DragSet implements Iterable<Figure> {
    static final int max = 200;
    Figure[] figs = new Figure[max];
    int[] dx = new int[max], dy = new int[max];
    int size = 0;
    public void add(Figure f, int x1, int y1) {
      if(size+1 >= max) { return; }
      figs[size] = f; dx[size] = f.getX()-x1; dy[size] = f.getY()-y1;
      f.setDrag(true); ++size;
    }
    public void moveTo(int x, int y, double t) {
      for(int i = 0; i < size; ++i) { figs[i].moveTo(x+dx[i], y+dy[i], t); }
    }
    public void release(double t) {
      for(int i = 0; i < size; ++i) {
        Figure f = figs[i];
        if(f instanceof MovingFigure) { ((MovingFigure)f).release(t); }
        f.setDrag(false); f.snap();
      }
      size = 0;
    }
    public boolean isIn(Figure f) {
      for(int i = 0; i < size; ++i) { if(figs[i] == f) { return true; } }
      return false;
    }
    public Iterator<Figure> iterator() {
      return new Iterator<Figure>() {
        int i = 0;
        public boolean hasNext() { return i < size; }
        public Figure next() { return figs[i++]; }
        public void remove() { }
      };
    }
  }
  public static void main(String[] args) {
    JFrame app = new JFrame("Block");
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.add(new Sam63()); app.setSize(800, 600); app.setVisible(true);
  }
```

```
}
```

注意! このデモはバグがあるようで、時々ブロックがへんなところ (場合によっては画面外) にワープすることがある。ぜひバグを取ってみてください。^_^;;

演習 2 「投げる」デモをそのまま動かし、試せ。納得したら次のことをやってみなさい。

a. 投げた時は「これまでドラグされてきた速度を維持」していたが、「その速度の α 倍 (1 より大、1 より小、負など色々あり得る)」にするとどんな感じか試してみなさい。
b. 投げたものが飛んでいる間は等速直線運動しているが、これを「加速度がある」「減速度がある」「カーブする」「重力がある」などに変更してみなさい。減速する場合は一定速度以下になると「停止してグリッドにスナップする」ようにしないと使いづらいと思います。重力は普通画面の下方向に働かせますが、お遊びとしては横方向や上方向にしてもよいです。
c. 現在は壁に当たるとそこで停止しているが、「跳ね返る」ようにしてみなさい。そのとき完全弾性反射だといつまでも止まらないので「エネルギーを奪われる」ようにした方がいいかも。お遊びとしては逆に「エネルギーが加わる」ようにしてもよいです。
d. 今は他のブロックがあっても一切干渉せず通過しているが、「他のブロックがあるとそこで止まる」「反射する」「弾性衝突 (弾性率 1 でももっと別の値でも)」「通過するがエネルギー吸収され、ある程度以上遅いとくっついて止まる」などをやってみなさい。
e. 現在はグループドラグしていちどに投げても動いている間はたまたま同じ動きなので形を保っているだけで、壁で止まる時はばらばらに止まるが、「グループで投げたらそのグループのまま止まったり跳ね返ったりする」ようにしてみよ。
f. グループドラグしていて、マウスボタンを離さなくても「大きな加速度を与える」と、端っこのブロックが「ちぎれて」飛んでいくようにしてみなさい。
g. その他面白いと思う機能を盛り込んでみなさい。

課題 X ユーザインタフェースがありきたりでない (自分なりの独自の工夫があるような) プログラムを作成しなさい。ジャンル (ゲーム、鑑賞、実用、デモ等) は自由である。

07 プレゼンテーションの指示

「課題 X」のプログラム (のできかけ) の方針、設計、(動いている場合) やってみた結果どうだったか、などの内容でプレゼンしていただく。1 人あたりの時間は 5〜7 分 (質疑の時間は別枠)。極端に短い場合は減点対象。時間オーバーは打ち切ることがある。

内容構成や手法は自由だが、スライドを作ってしゃべらないと時間が余って「極端に短い」になるおそれがある。5〜7 分なのでスライドも 5〜7 枚程度がよいとおもわれる。動く場合はもちろんデモをした方がよい。他人のプレゼンの質疑にも積極的に参加すること。

プレゼンの環境は CED のマシンなので、そこでできるように準備 (各自のアカウントでログインしてもらいます)。プロジェクタ画面はマシンの全画面のうち「1024x768」の部分しか出ないので注意すること。

06 の後半 (予定) の時間、および # 07 の冒頭 30 分程度を準備の時間として割り当てますが、たぶんそれだけだと足らないと思うので、その間に自分の時間でも作業した方がよいです。

相互評価のため、シートを配布し、自分以外の全員について評価とコメントを記入していただきます (筆記用具を持参すること)。

レポートの指示

課題 X について (できればプレゼンでの質疑内容も含めて) レポートを作成していただく。レポートは初回告知通り、11 月いっぱいのうちに久野までメールで「PDF を」送付すること。LaTeX を希望するがそれ以外でも PDF ならよい。次の内容が含まれること。

1. 表紙 (1 枚の紙でなくて 1 枚目の冒頭部分ということでもよい)。タイトル (作成した作品のタイトルが普通だと思う)、学籍番号 (レポートもネットに掲示するので氏名は省く)、提出日付。
2. 課題に対する方針や作品の構想。
3. プログラムの実装の方針や構造・設計。
4. ソースコード。適宜説明を加えること (コメントでもよい)。
5. 動いている画面の図と、動き方の説明。
6. 評価 (自己評価やプレゼン時のコメント・質疑など)
7. 考察 (課題 X をやってみて分かったこと)

ついでに以下のアンケートの回答も末尾にくっつけて記してください。

Q1. ユーザインタフェースのプログラミングに対する自分の認識・知識・腕前はこの実験を通じて変化しましたか。YES の場合、具体的にはどのように?

Q2. このテーマの内容・進め方両面について、よかったこと、改善して欲しいことをそれぞれ (最大 3 つ程度) 書いてください。

Q3. 全体としての感想・要望