JavaによるUIプログラミング: MICS実験2-19-1c
＃05 人間の特性とユーザインタフェース

久野　靖*

2019.10.21

1　Human Virtual Machine

1.1　人間のモデル化の必要性

前回さまざまなGUIのデザインを考え「スムーズに使えるインタフェース」を工夫しようとしました。スムーズを端的にいえば、「操作に掛かる時間が短い」ということになるでしょうか。

操作に掛かる時間そのものは、実際に操作してみて測ればわかります。ただ、時間が測れるというだけでは「どうしたら操作時間を速くできるか」は分かりません。

操作時間を目標に設計するなら、「操作時間はこのようにして決まる」というモデルが必要です。そのモデルが正しいなら、そのモデルに基づいて操作時間が短くなるように設計すればよいわけです。そういうわけで、今回はソフト作りの前に「ユーザインタフェースに対した時の人間の行動」のモデルについて考えてみます。

たとえば、Normanは次のような問題解決のモデルを提唱しました。複雑なタスクの場合は、この系列の個々のアクションが再び小さな「目標」となり、上と同様に再帰的に分解されていきます。

(1) 目標を設定

(2) 目標を達成するアクションの持つ目的を明確化

(3) アクション系列の具体化

(4) アクション系列の実行

(5) 状態 (結果) の知覚

(6) 知覚したものの解釈

(7) 結果の評価→目標と比較するため (1) へ戻る

このモデルは確かに正しそうで、Webで何かをしようとするユーザがどのような作業系列を行うか、のような分析には使えそうです。ただ、現在やっているような「動作の速さ」のようなものよりはだいぶ上のレベルの話だという気もします。

これとは別のアプローチに、「認知情報処理アプローチ」と呼ばれるものがあります。これは、人間を一種の情報処理系としてモデル化し、その性能を理解しようとするものです。このモデルは「認知」とついてはいますが、単に認知心理学の研究のためのモデルというわけではなく、実際にシステム設計に関わるデザイナが予測に活用できることをめざしているという点に特徴があります。このモデルでは、「画面に文字が表示されたら、それと同じ文字のキーを打つ」という場面での人間の情報処理の構造は「(入力) →「知覚系」→「認知系」→「運動系」→ (出力)」のように、人間内部の複数のシステムが連携することで行われる、というふうに理解します。

*電気通信大学

演習 0a 紙とペン (ないし好みの筆記用具) で、次の 2 つの課題を行うものとする。それぞれにおいて「線の本数」がどれくらいになるか、予想しなさい。また、なぜそうなると思うのかも述べること。

a. 5 秒間で、2cm 間隔の平行線をまたぐように、できるだけ多くのジグザグ線を描く。
b. 5 秒間で、4cm 間隔の平行線をまたぐように、できるだけ多くのジグザグ線を描く。
c. 5 秒間で、2cm 間隔の平行線の外側にそれぞれ 1cm の間隔をおいた平行線の、内側の 2 本だけをまたぐように、できるだけ多くのジグザグ線を描く。

1.2 Model Human Processor

上でのべた「認知情報処理モデル」は、具体的には Model Human Processor (MHP) と呼ばれていて、S. K. Card, T. P. Moran, A. Newell によって最初に提唱されたものです。ここではもう少しその詳細を見て行きます。MHP の構造を図 1 に示します。

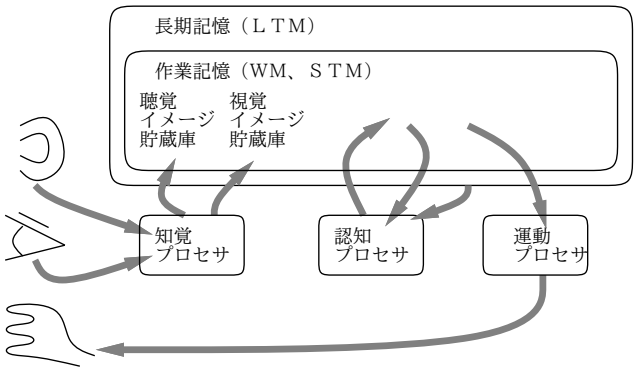

図 1: Model Human Processor の構造

まず、MHP より以前の前提として、人間の脳の中には作業記憶 (WM — Wroking Memory。短期記憶 — STM, Shot Term Memory —と呼ぶこともある) と長期記憶 (LTM — Long Term Memory) という 2 つの領域 (ないし機能) があると考えられています。

- 作業記憶 — 入力された情報がとりあえず入る場所だが、入る情報には限りがあり、およそ「かたまり」7 個ぶんくらいの容量である。その情報をずっと覚えているためには、意識して「反芻」する必要がある。
- 長期記憶 — 作業記憶に十分長くとどまった情報は最終的に長期記憶に格納される。その容量は極めて大きい。格納には時間が掛かるが、取り出しは瞬時である。ただし取り出し (検索) に失敗することもある。そのときも、その情報が無くなったのではなく、単に検索できていないだけである。

ほかにも色々な属性が言われていますが、これらは単なる推測ではなく、さまざまな実験によって裏付けられています。たとえば、数字をランダムに読み上げられて「できるだけ思い出して言ってください」というと 7 個くらが限界だとか…

そして、MHP でモデル化しているのは何かというと、次のようなことです。

- 人間の知覚は「知覚プロセサ」によって処理されて、それが WM 内の視覚メモリや聴覚メモリに入って「そのままの形で」短期記憶される。(視覚メモリや聴覚メモリが WM 内にある、ないし WM と資源を共有していることは、どういう実験をすれば分かると思いますか?)
- 見たものや聴いたものに対する認識 (どんな形とかどんな音とか) を判断するのは、認知プロセサによって行われる。認知プロセサは WM および LTM から情報を取り出して処理し、結果を

WM に書き込む。(認知プロセサが LTM からも直接情報を取り出せることはどういう事実から分かると思いますか?)

- 筋肉を動かす (→眼球、手、指などを動かす) 処理は、運動プロセサによって行われる。運動プロセサは WM の情報に基づいて筋肉をいつどのように動かすかを制御する。
- 人間の一連の動作は、これらのプロセサが連携して行っている。たとえば「ランプがついたらボタンを押す」という課題をやっているとして、「ランプがついた」様子は知覚プロセサによって WM に入り、認知プロセサが「ランプがついたからボタン」という判断をおこない、「ボタンを押せ」と指に命じるのは運動プロセサがおこなう。
- それぞれのプロセサにはサイクル時間があり、一番基本的な (簡単な) 処理を行うのに 1 サイクルの時間が掛かる。

ここで、各プロセサのサイクル時間はだいたい表 1 のように見積もられています。ということは、「ランプがついたらボタンを押す」タスクに掛かる時間は、平均的な人で $\tau_p + \tau_c + \tau_m = 240msec$ ということになるわけです。さて、あなたは「平均的な人」「速い人」「遅い人」のどれだと思いますか?

表 1: MHP の各プロセサのサイクル時間

プロセサ	記号	典型値	最小〜最大
知覚	τ_p	100	50〜200
認知	τ_c	70	25〜170
運動	τ_m	70	30〜100

1.3 単純反応時間の計測

ではさっそく、「ランプがついたらボタン」の時間を計測してみましょう。昔だったら、部品を持って来て工作することになるのですが、今なら PC さえあればプログラムで全部済みます。プログラムが動いているところを図 2 に示します。起動したあと、マウスで窓をクリックすると、しばらくして緑の四角が現れます。現れたらできるだけ速くどれかのキー (スペースバーが押しやすいかも) を打ちます。10 回やると終わりで、窓の上には平均反応時間 (ミリ秒) が表示されます。

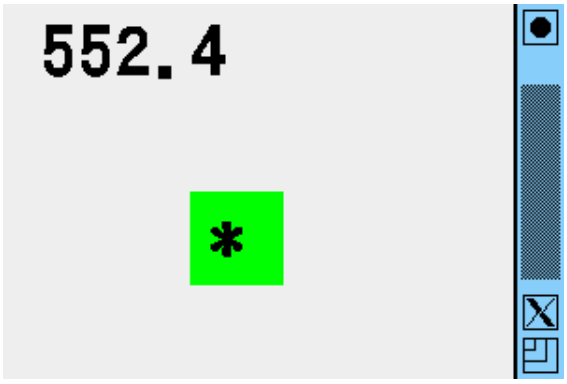

図 2: 反応時間プログラム

コードを示します。マウスクリック時にカウントと時間合計を 0 にし、`setTimer()` で一定時間後に四角を表示するようにさせます。また、キーを押された際は (計測中であれば) 掛かった時間を時間累計に加え、回数を 1 増やし、10 回未満なら再度 `setTimer()` を呼びます。`paintComponent()` の中では、平均時間は常に表示し、緑の箱は `showbox` が `true` のときだけ表示します (後で使うために文字も表示するようにしています)。

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

public class Sam51 extends JPanel {
  Font fn = new Font("Courier", Font.BOLD, 36);
  boolean showbox = false;
  long time; int count; double atime = 0.0;
  public Sam51() {
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        requestFocus(); atime = count = 0; setTimer();
      }
    });
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        if(!showbox) { return; }
        long dt = System.currentTimeMillis() - time; atime += dt; ++count;
//      System.out.printf("time = %d, ch = '%c'\n", dt, evt.getKeyChar());
        if(count < 10) { setTimer(); } else { repaint(); }
      }
    });
  }
  public void paintComponent(Graphics g) {
    g.setFont(fn); g.setColor(Color.black);
    g.drawString("" + (count < 1 ? 0.0 : atime/count), 20, 40);
    if(!showbox) { return; }
    g.setColor(Color.green); g.fillRect(100, 100, 50, 50);
    g.setColor(Color.black); g.drawString("*", 110, 140);
  }
  public void setTimer() {
    showbox = false; repaint();
    Timer t1 = new javax.swing.Timer(2000 + (int)(2000*Math.random()),
      new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
          showbox = true; time = System.currentTimeMillis(); repaint();
        }
      });
    t1.setRepeats(false); t1.start();
  }
  public static void main(String args[]) {
    JFrame app = new JFrame();
    app.add(new Sam51());
    app.setSize(300, 200);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
      app.setVisible(true);
    }
  }
```

`setTimer()` ですが、箱を非表示にし、2 秒+ランダム時間だけ待ってから箱を表示するとともにその時刻を覚えるようにしています。なお、今回はタイマーは「1 回だけ」動作が実行されるモードで使います (毎回待ち時間を変えるため)。

演習 1 このプログラムを動かし、自分の反応速度を計測してみなさい。また、次のような場合はどうなるかやってみなさい。

a. 今は文字が常に「*」だが、「a」なら「a」のキー、「b」なら「b」のキーを打つことにした場合に反応時間はどれくらい変わるか。もっと文字の種類を増やした場合はどうか。
b. 数字を表示して、その数字に 2 足した値のキーを押すとした場合は上記とどのように違うか。
c. (成人かつ飲める人のみ) アルコール飲料を用意し、アルコールを飲んだ時の量と反応速度の関連について調べなさい (自宅等、飲んでも構わない場所でやること)。
d. 反応時間をミリ秒で表示では実験ぽいが、反応時間が速いほど高い得点が入り、間違ったキーだと得点が減るとかにしたらゲームになる。そういうゲームにしてみなさい。
e. 反応時間を競う上記とは別の内容のゲームを設計し作りなさい。

あと、この実験をやってみると「四角が現れるまで 1 箇所を見つめ続ける」というのが苦痛だと分かると思います。人間の眼球は「ふらふらさまよう」「止まって 1 点を見つめる」を交互に行うようになっているので、ずっと止まっているというのはかなり不自然なわけです。

1.4 運動システムと Fitzz の法則

単純反応時間 (最初の反応までの時間) は先に測った通りですが、実際のさまざまな動作では「何を指す」などのように行き先までの距離がさまざまであり、それに応じて掛かる時間も変化していくものと思われます。

具体的には、何かを指す位置ぎめ (ポジショニング) は「まずある程度指を動かし、近づき方を目で見て確かめ、もうちょっと動かし (または行きすぎたら戻し)、また確認し…」のようなサイクルを回ることになります。では、距離が遠い程、つまり距離に比例して、サイクルの回数が増えて、時間もそれに比例して増えると思いますか? 実験してみましょう。

演習 0b グラフ用紙に、1 辺が 1cm の正方形を中心間隔が 8cm になるように 2 つ描き、ペン先でそれらの中に交互に点を打って行くものとする。合計 20 個 (片側 10 個ずつ) 点を打つのに要する時間を計測し、それから 1 回あたりの所要時間を求めなさい。さらに、次のことも行いなさい。

a. 正方形の大きさを 1 辺が半分の 5mm にしたら、また倍の 2cm にしたら、時間はどのように変化するか (またはしないか)。
b. 正方形の間隔を半分の 4cm にしたら、また倍の 16cm にしたら、時間はどのように変化するか (またはしないか)。
c. これらの結果から、ポジショニングに掛かる時間にはどのような性質があると考えられるか仮説を作れ。

やってみると分かると思いますが、的が小さいと「微調整」に手間が掛かるため、ポジショニング時間は「距離 D が大きいと長くなり」「的の大きさ S が大きいと小さくなり」ます。さらに言えば、

ポジショニング時間は「$\frac{D}{S}$ の関数」となります。これを定式化した次の式は Fitzz の法則と呼ばれています。

$$T_{pos} \ = \ I_m \ \log_2(\frac{D}{S} + 0.5)$$

ここで、定数 I_m も個人差のある値で、典型的な人で 100、範囲は 50〜120 とされています (単位は「msec/bits」)。典型値 100 の場合の、$\frac{D}{S}$ と反応時間の表を 2 に示します。

表 2: $I_m = 100$ での $\frac{D}{S}$ と反応時間

D/S	Tpos
1.00	58.50
2.00	132.19
4.00	216.99
8.00	308.75
16.00	404.44
32.00	502.24
64.00	601.12
128.00	700.56
256.00	800.28
512.00	900.14
1024.00	1000.07

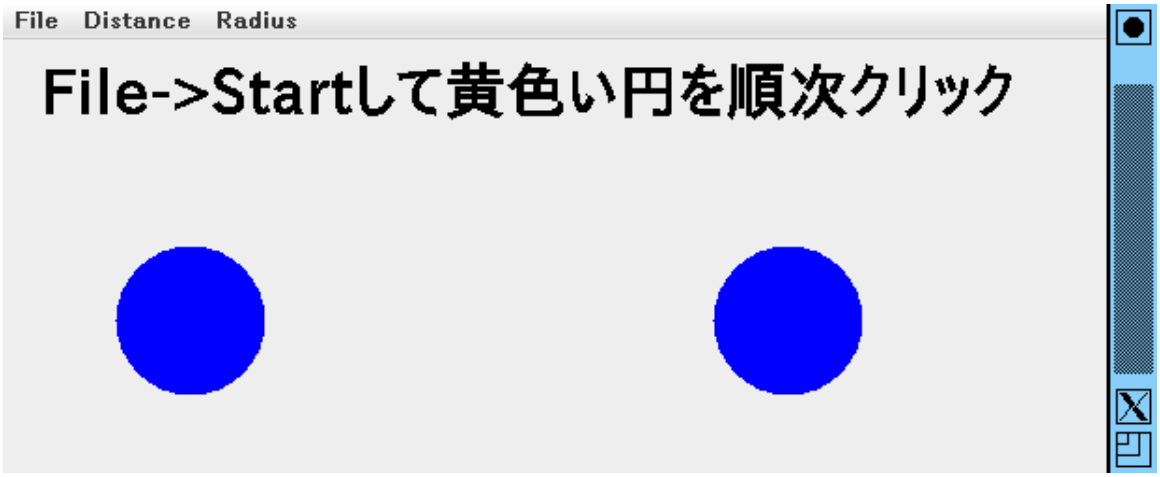

図 3: Fitzz の法則実験用プログラム

ということで、先の紙でやった実験もプログラムで計測できるようにしてみました (図 3)。このプログラムでは円の半径や円どうしの距離を適宜選択してから「start」を選ぶことで、計測が開始でき、黄色い円を交互にクリックしていくと平均クリック間隔が表示され続けます。適当な回数まで来たらやめて記録すればよいわけです。

プログラムの方を見ましょう。いろいろ設定があるのは、すべてプルダウンメニューで行います。File メニューは「Start」「Quit」の 2 つ、そして距離メニューと半径メニューで複数の距離と半径を選べます。距離や半径のメニュー項目はやることが同様なので、無名内部クラスではなく名前つき内部クラスにして、項目ごとに円の位置や半径を変更する動作をするようにしました。これらからアクセスするため、円のオブジェクトや時間等の変数は外側クラスのインスタンス変数にしています。

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

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

public class Sam52 extends JFrame {
  Font fn = new Font("SansSerif", Font.BOLD, 32);
  String msg = "File->Startして黄色い円を順次クリック";
  Circle c1 = new Circle(Color.BLUE, 100, 150, 20);
  Circle c2 = new Circle(Color.BLUE, 140, 150, 20);
  int count; double atime; long time;
  public Sam52() {
    JMenuBar b1 = new JMenuBar(); setJMenuBar(b1);
    JMenu m1 = new JMenu("File"); b1.add(m1);
    JMenuItem i0 = new JMenuItem("Start"); m1.add(i0);
    i0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        atime = count = 0;
        c1.setColor(Color.YELLOW); c2.setColor(Color.BLUE);
        repaint();
      }
    });
    JMenuItem i1 = new JMenuItem("Quit"); m1.add(i1);
    i1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) { System.exit(0); }
    });
    JMenu m2 = new JMenu("Distance"); b1.add(m2);
    for(int i = 40; i < 1200; i *= 2) {
      JMenuItem i2 = new JMenuItem(""+i); m2.add(i2);
      i2.addActionListener(new ItemPosition(100+i, 150));
    }
    JMenu m3 = new JMenu("Radius"); b1.add(m3);
    for(int i = 10; i < 100; i += 10) {
      JMenuItem i3 = new JMenuItem(""+i); m3.add(i3);
      i3.addActionListener(new ItemRadius(i));
    }
    add(new Panel1());
  }
  class ItemPosition implements ActionListener {
    int x, y;
    public ItemPosition(int x1, int y1) { x = x1; y = y1; }
    public void actionPerformed(ActionEvent evt) { c2.moveTo(x, y); repaint(); }
  }
  class ItemRadius implements ActionListener {
    int r;
    public ItemRadius(int r1) { r = r1; }
    public void actionPerformed(ActionEvent evt) {
      c1.setRadius(r); c2.setRadius(r); repaint();
```

```
    }
  }
```

中にはめる Panel1 が計測を行う部分です。表示するものは文字列 (メッセージや計測結果) と円 2 つで、マウスクリックが黄色い側の円に当たっていると前のクリックからの時間を累計し回数を増やします。そしてそこまでのクリック時間の平均値を画面に表示します。

```
  class Panel1 extends JPanel {
    public Panel1() {
      setOpaque(false);
      addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent evt) {
          if(count % 2 == 0) {
            if(!c1.hit(evt.getX(), evt.getY())) { return; }
            c1.setColor(Color.BLUE); c2.setColor(Color.YELLOW);
          } else {
            if(!c2.hit(evt.getX(), evt.getY())) { return; }
            c2.setColor(Color.BLUE); c1.setColor(Color.YELLOW);
          }
          long t = System.currentTimeMillis();
          if(count > 0) {
            atime += t - time;
            msg = String.format("%d: %8.2f", count, atime/count);
          }
          ++count; time = t; repaint();
        }
      });
    }
    public void paintComponent(Graphics g) {
      g.setColor(Color.BLACK); g.setFont(fn);
      g.drawString(msg, 20, 40); c1.draw(g); c2.draw(g);
    }
  }
```

円オブジェクトとか main とかはもう説明するまでもないですね。

```
  static class Circle {
    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 void setColor(Color c) { col = c; }
    public void moveTo(int x, int y) { xpos = x; ypos = y; }
    public void setRadius(int r) { rad = r; }
    public void draw(Graphics g) {
      g.setColor(col); g.fillOval(xpos-rad, ypos-rad, rad*2, rad*2);
    }
```

```
    public boolean hit(int x, int y) {
      return (x-xpos)*(x-xpos) + (y-ypos)*(y-ypos) <= rad*rad;
    }
  }
  public static void main(String args[]) {
    JFrame app = new Sam52();
    app.setSize(800, 400);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

演習 2 上の例題プログラムを使って Fitzz の法則が成り立っているかどうか試してみなさい。

演習 3 上記の通り、円の位置やサイズをうまく調節すると「クリックまでに時間が掛かるのでそれを競うゲーム」とかができます。次のようなゲームを作ってください。

a. 回数を固定して (たとえば 20 回)、最速を競うゲーム。
b. 同様だが円が 1 つで、クリックするたびに乱数で新しい位置に飛ぶ。
c. 同様だが円が止まってないでふらふら動いてクリックしづらくする。
d. 同様だが円がダミーのものも含め沢山あり、間違ったのをクリックするとアウトになる。
e. その他好きなゲーム設計で作る。

2 マウス操作とメニュー

2.1 マウス操作の用途

ここまでで見て来たように、マウスポインタを目的地まで移動する速さは Fitzz の法則によって決まります。そのことを念頭において、GUI において頻繁に使われる部品であるメニューのデザインという問題を検討してみましょう。

ここではメニューとは、レストランとかで料理を選ぶときに見るもの…ではなく、この場合は次の両方の条件を満たすような GUI 部品ということにします。

- 何らかの操作で画面に現れる。
- 項目が一覧のように並んでいて、その中から選択する。
- 選択が完了すると消えて元の状態になる。

現れたり消えたりしないもの、たとえばボタンが並んでいるだけの部品は「パレット」などと呼ばれ、メニューとは区別して扱うことが普通です。メニューの現れ方については、「何もないところで」「マウスカーソル位置に」表示されるポップアップメニュー、「メニューボタンを押すと」「その位置に開く」プルダウンメニュー (下とは限らない) の 2 種類があります (他にありますか?) では、メニューは何がよくてこんなに使われているのでしょうか?

メニューの利点としては、次のものが挙げられます。

- 必要なときに操作により表示されるので、画面を占有しない。このため、パレットのようにボタンを小さくしなくて済み、十分な大きさを取ることができ、選択項目を多くできる。
- 選択する対象が向こうから表示されるので、どのような選択肢があるかを覚えなくて済む。

- 項目は表示時点で設定可能であり、毎回変化しても構わない (動的)。また、文脈によって内容が変わることもできる (Windows の右ボタンメニューなど) → 「コンテキストメニュー」とも呼ぶ。

項目数については、とくにメニュー項目の上にカーソルを置くとさらにサブメニューが開く、階層メニューを多段で使うと、非常に多くの選択肢を持たせることができます。

では、メニューは「いいことづくめ」でしょうか? もちろんそんなことは無くて、弱点も沢山あります。

- 表示させるまで「どこを選択するか」が決められない。このため、「表示動作」→「表示」→「見る」→「選択の判断」→「選択動作」という系列が必要となり、時間が掛かる (とくに階層メニューの場合)。
- 項目数が多かったり動的に変化する場合、「毎回使っているうちにどこを選ぶか学習する」ということが難しい。

前者の弱点を補うために、「ちぎって画面上に置いておける」メニューというのもありますが、置いておくとその分だけ場所ふさぎになるので善し悪しだと言えます。やはり、置いておくのならもともと小さくデザインされたパレットの方が良さそうです (そのかわり、パレットではどのアイコンがどの機能かを覚えておく必要がありますが)。

別の方法として、メニューに「キーボードショートカット」を割り当てておき、メニューと一緒にそれが表示されるので、よく使うものは自然に覚えてしまってショートカット経由で起動するようになる、という工夫もよくあります。これは「自然に学べる」という点でうまい方法だと思います。

2.2 メニューの時間計測

では今回も、時間を測ってみましょう (その前に、ポップアップメニューから項目を選択するのに掛かる時間はどのくらいだと予想しますか?)。そのために、できあいのメニューではなく「自前ですべて」メニューの機能を実装してみました。このプログラムは、A/B/C/D という 4 つの文字が表示され、それと同じ名前の項目をポップアップメニューから選ぶようになっています (図 4)。

プログラムの基本的な構造は同じですが、メニューを表示するための仕掛けが多少面倒で、変数も多くなっています。

- `GeneralPath` というのは任意の輪郭線を描くためのオブジェクトであり、今回は多角形のデータを指定してそれをもとに各メニュー領域を作成します。
- 「問題」は A〜D の 4 択ですが、もっと増やすこともできます。
- `pos` は各メニュー項目のラベルの表示位置の XY 座標を保持するのに使います (2 つずつペアで XY 座標を表します)。
- `count` は各項目の選択個数、`atime` は各項目の問題提示から選択完了までに掛かった時間の累計 (ミリ秒) です。
- `menux`、`menuy` はメニュー位置、`mousex`、`mousey` はマウス位置、`probno` は「問題番号」です。
- `prob` は現在表示されている「問題」(どの項目かのラベル)、`cur` はメニューで選択した項目の番号です。
- `showmenu` は `true` だとメニュー表示中という意味になります。
- `msg` は表示メッセージ、`time` は問題提示時刻です。

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

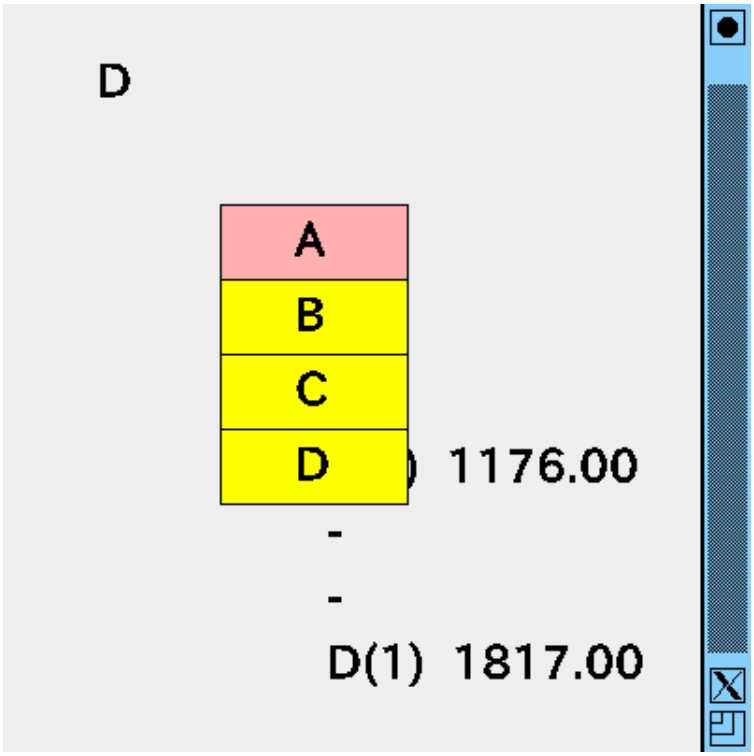

図 4: メニューの実験用プログラム

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

public class Sam53 extends JPanel {
  String[] label = new String[]{"A", "B", "C", "D"};
  final int N = label.length;
  GeneralPath[] panes = new GeneralPath[N];
  int[] pos, count = new int[N];
  double[] atime = new double[N];
  int menux, menuy, mousex, mousey, prob = -1, cur = -1;
  boolean showmenu = false;
  String msg = "窓内をクリックすると開始します";
  long time;
  Font fn = new Font("SansSerif", Font.BOLD, 24);
```

コンストラクタで初期設定時にイベントハンドラ (今回はマウスのものだけ) を指定しますが、その説明は次の通りです。

- マウスボタン押しのときは、マウスの XY 座標を覚え、メニュー表示状態にして再表示します。ただし初回 (prob が負) のときは問題を提示して時刻を覚えます。
- マウスボタンが離されたら、問題と選択内容と時刻を出力し、次の問題を設定してメニューは消して再表示します。
- マウスポインタが動いたら、その XY 座標を覚えて再表示します。

その後で、4 つのメニュー領域とラベルを書く位置を初期設定しています。演習時にメニューの配置を変更するときは、この部分だけ変更してください。

```
  public Sam53() {
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        menux = mousex = evt.getX(); menuy = mousey = evt.getY();
        if(prob < 0) {
          time = System.currentTimeMillis();
          prob = (int)(Math.random()*N);
          msg = label[prob]; repaint();
        } else {
          showmenu = true; repaint();
        }
      }
      public void mouseReleased(MouseEvent evt) {
        long t = System.currentTimeMillis();
        if(cur == prob) {
          ++count[prob]; atime[prob] += t - time; time = t;
//        System.out.printf("%d %d %8.2f\n", prob, count[prob], atime[prob]);
          prob = (int)(Math.random()*N); msg = label[prob];
        }
        showmenu = false; repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        mousex = evt.getX(); mousey = evt.getY(); repaint();
      }
    });
    // メニューの形状変更は以下のデータを変更すればよい
    panes[0] = mp(new int[]{0,0, 0,40, 100,40, 100,0});
    panes[1] = mp(new int[]{0,40, 0,80, 100,80, 100,40});
    panes[2] = mp(new int[]{0,80, 0,120, 100,120, 100,80});
    panes[3] = mp(new int[]{0,120, 0,160, 100,160, 100,120});
    pos = new int[]{40,28, 40,68, 40,108, 40,148};
  }
```

メソッド paintComponent() はいつも通り窓の中身を描きますが、その概要は次の通りです。

- まず黒色でメッセージを表示し、続いて各項目の時間表示を行います。メニューが出ていないときはこれで終わり。
- メニューの起点を原点 $(0, 0)$ になるように座標を移動。
- メニューの各項目について、中身を黄色で塗り、輪郭を黒で描く。ただしマウス座標が項目の領域に入っていれば、塗る色はピンクに変更し、またそのラベルを覚える。
- メニューを表示し終わったら、座標を元に戻す。

下請けメソッド mp() は、配列に XYXY…の順で格納した輪郭線のデータから対応する GeneralPath オブジェクトを生成しています。main はいつも通りです。

```
  public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(Color.black); g2.setFont(fn); g2.drawString(msg, 50, 50);
    int x = getWidth() - 200, y = getHeight() - N*36;
    for(int i = 0; i < N; ++i) {
      String s = "-";
      if(count[i] > 0) {
        s = String.format("%s(%d) %8.2f", label[i], count[i], atime[i]/count[i]);
      }
      g2.drawString(s, x, y); y += 35;
    }
    if(!showmenu) return;
    g2.translate(menux, menuy);
    cur = -1;
    for(int i = panes.length-1; i >= 0; --i) {
      Color col = Color.yellow;
      if(panes[i].contains(mousex-menux, mousey-menuy)) {
        col = Color.pink; cur = i;
      }
      g2.setColor(col); g2.fill(panes[i]);
      g2.setColor(Color.black); g2.draw(panes[i]);
      g2.drawString(label[i], pos[2*i], pos[2*i+1]);
    }
    g2.translate(-menux, -menuy);
  }
  private GeneralPath mp(int[] a) {
    GeneralPath p = new GeneralPath();
    p.moveTo(a[0], a[1]);
    for(int i = 2; i < a.length; i += 2) { p.lineTo(a[i], a[i+1]); }
    p.closePath(); return p;
  }
  public static void main(String args[]) {
    JFrame app = new JFrame();
    app.add(new Sam53());
    app.setSize(400, 400);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

演習 4 このプログラムを動かし、メニュー選択の時間がどれくらいか計測しなさい。また、選択時間をより短くするために、どのような工夫があり得るかを考えなさい。

2.3 メニュー選択の所要時間と改良

先に触れたように、メニューを選択するときは「メニューを出そうと思う」→「ボタン押し」→「メニューが出る」→「選択項目を決定する」→「ドラグして選択項目が選べるまで移動」→「ボタン離

し」という一連の (複雑な) 動作が起こっています。では、どのようなことを工夫すれば、この時間を「全体として」小さくできるでしょうか?

たとえば、次のような可能性が考えられます。

- マウスのドラグ距離を小さくした方がいいかも知れない。それには、メニューを「薄く」(1 項目の高さを小さく) するのがいいかも知れない。
- 移動方向は現在の「下向き」がいいのかどうか? もしかしたら、上向きとか左右どちらか向きがいいかも知れない。
- メニューの表示位置が「一番上の項目がポインタに合っている」必要はないかも知れない。たとえば、メニューの中央にポインタがある方がいいかも知れない。
- メニューの項目は四角である必要はないわけだが、別の形にしてみた方がいいかも知れない。
- 同心円状になっていて、「どちらへドラグしてもいい」ようにしたらよいかも知れない。
- メニュー項目が隣接している必要はないかも知れない。「4 つの島」になっていても別によいわけだが、そのようにして均等の距離に現れるようにしたらどうか。

もちろん、ユーザインタフェースの研究としてメニューの研究は沢山行われていて、代表的なものとして次のようなメニュー方式があります。

- パイメニュー — 円形のメニューで放射状に区切られていて、どの項目もちょっと動かすだけで選択できる。
- 隠れにくいパイメニュー — 上と同様だが、ラベルだけが表示される (クリック点が隠されないという利点がある)。
- マーキングメニュー — メニューが出るのを待っていると遅いので、メニューが出なくてもその方向にドラグしてすぐ離したら選択できる。

しかし、いずれも結構前からある研究で、その論文では「効果があった」ということになっているのですが、いずれも現実には普及していないわけです。なぜでしょうね?

演習 5 グラフ用紙に、「自分が速いだろうと思う」4 項目ぶんのポップアップメニューの形・配置をデザインしてみなさい。表示開始時のマウスポインタ位置も明示すること。なお、実装の制約上、輪郭は多角形にする必要があります。デザインできたら、実際にプログラムを改造して実装し、時間を計測してみなさい。

演習 6 「速さ」でなく「奇抜さ」を目標としていろいろなメニュー項目の配置のデザインを考案し試してみなさい。もちろん時間も計測してみなさい。

演習 7 例題プログラムでは項目数が「4」だったが、「8」「16」など項目を多くした場合はどのようになるか。また、メニュー構成を工夫することで多くした場合の問題点をカバーできるかどうかも検討してみなさい。