ユーザインタフェース＃2 — 人間の行動モデル/Fitzzの法則

久野　靖*

2012.10.16

1　ユーザインタフェースの様々な問題 — Readingから

1.1　「HCIは塩化水素か」

この文章は、自分の恩師である木村 泉先生が「bit」という雑誌 (コンピュータサイエンスを扱う日本で唯一の一般むけ雑誌だったのですが、コンピュータの普及とともに売り上げが低下して廃刊してしまいました) に連載されていた記事から取ったものです。

内容の冒頭は、HCIという用語の説明をしていますが、これについては前回もお話しました。木村先生の節は次のようになっています。

- 「ユーザインタフェース」 — コンピュータの側の工夫が中心であり (そこは同意)、馬鹿な利用者に対処してやるためものというニュアンスがある (そこはちょっと…) という分野
- 「ヒューマンファクター」 — 人間がからんだ時のさまざまな問題に焦点を充てている (ないし、人間工学 — 椅子の形とか机の高さとか — の分野)
- 「HCI」 — 人とコンピュータがどうやりとりするかを扱う広い分野 (ニュートラルな感じ)

まあ、この授業がユーザインタフェースと名乗っているのは、自分がHCIと言えるほど人間のことに詳しくない (反面、ソフトは作りたい)、ということを表しているわけです。^_^;

次に、いらいらする振込み画面という話題がありますが、こういう問題は今でも結構あるのではないでしょうか。「失敗したら1つ前まで戻れる」くらいはもっと実装して欲しいと思いますが… あと、この画面構成でなぜユーザは間違えるのか、という分析が面白いです。

1.2　「情報時代のなぞなぞ」

こちらは「アラン・クーパー, 山形訳, コンピュータは、むずかしすぎて使えない!, 翔永社, 2000」の抜粋ですが、今の機器は何でもコンピュータを組み込んで便利にしようとしてくれているけど、それが余計なお世話で非常に分かりにくくなっている、という話がうまく書かれています。本全体でも非常に面白いので興味があれば入手してみてください。

1.3　「毎日使う道具の精神病理学」

これは「 D. A. ノーマン, 野島訳, 誰のためのデザイン?, 新曜社, 1990」という非常に有名な本の1章です。この章だけでも色々な話題が盛り込まれていますが、簡単に内容をリストアップしておきます。

- 人の頭は世の中をうまく理解するように巧妙に作られているのに、なんでそんなに使えないものがあふれているのか?

*経営システム科学専攻

- 色々な使いにくいものの話 — ドア、スライド投影装置、映写幕の上下装置、新しい電話機
- アフォーダンス — 「こういう風に使うんだよ」と導いてくれるようなカタチ
- 概念モデルの重要性、可視性、対応性、フィードバック
- 技術の逆説 — 技術が発達して色々できるようになったおかげで、ろくに何もできなくなる → デザインの重要性

もともと、インタフェース (interface) とは、「何かと何かが接するところ」という意味なわけです。ですから、ユーザインタフェースは「コンピュータ」とそれを使っている「ユーザ (利用者)」とが接するところ、なわけです。これをコンピュータの側から見ると、「入出力装置」のうちで、人間とやりとりする (人間からの入力を受け取る/人間に出力を提示する) ものがユーザインタフェース (User Interface, UI) を構成しているわけです。

2 前回のおまけの再掲: キー入力とメニュー

前回の最後の例題を少し改良したものを示します。これは、英語の月名を問題として自動提示し、それと同じものをキー入力するか、または黄色いボタンから選択してクリックすると「正解」になるというものです。キー入力には「補完機能」がついています。さて、キーとマウスとどっちが速いでしょうか?

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

public class Sample16 extends JPanel {
  Font fn = new Font("Courier", Font.PLAIN, 16);
  FontMetrics fm = null;
  String[] months = { "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" };
  String goal = "";
  String line = "";
  int cpos = 0;
  long time = System.currentTimeMillis();
  long goaltime = time;
  public Sample16() {
    setOpaque(false);
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        long t = System.currentTimeMillis();
        char ch = evt.getKeyChar();
        if(ch >= ' ' && ch < 127) {
          if(cpos > 0 && cpos < line.length() && line.charAt(cpos) == ch) {
            ++cpos;
          } else {
            line = line + ch;
            int k = 0, c = 0;
            for(int i = 0; i < months.length; ++i) {
              if(months[i].toLowerCase().startsWith(line.toLowerCase())) {
```

```
            ++c; k = i;
          }
        }
        if(c == 1) { cpos = line.length(); line = months[k]; }
      }
      repaint();
    } else if(ch == 10 || ch == 13) { // CR, LF
      System.out.println(t - goaltime);
      line = goal = ""; cpos = 0; repaint();
      Timer t1 = new Timer(
        1000+(int)(1000*Math.random()), new MyAdapter());
      t1.setRepeats(false); t1.start();
    } else if(ch == 8 || ch == 127) { // BS, DEL
      if(line.length() > 0) { line = line.substring(0, line.length()-1); }
      repaint();
    }
    if(ch > 32 && ch < 127) {
      System.out.println((t-time) + " : " + ch);
    } else {
      System.out.println((t-time) + " : " + (int)(ch));
    }
    time = t;
  }
});
addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent evt) {
    requestFocus();
    long t = System.currentTimeMillis();
    int x = evt.getX(), y = evt.getY(), k = -1;
    for(int i = 0; i < months.length; ++i) {
      if(10+(i/6)*180 < x && x < 10+(i/6)*180 + 100 &&
         120+(i%6)*50 < y && y < 120+(i%6)*50 + 25) { k = i; }
    }
    if(k >= 0) {
      System.out.println((t-goaltime)+" ("+x+","+y+") "+ months[k]);
      line = goal = ""; repaint();
      Timer t1 = new Timer(
        1000+(int)(1000*Math.random()), new MyAdapter());
      t1.setRepeats(false); t1.start();
    }
  }
});
}
class MyAdapter implements ActionListener {
  public void actionPerformed(ActionEvent evt) {
    goal = months[(int)(Math.random()*12)];
    repaint();
    long t1 = System.currentTimeMillis();
```

```
      System.out.println((t1-time)+" ! "+goal);
      goaltime = time = t1;
    }
  }
  public void paintComponent(Graphics g) {
    g.setFont(fn); g.drawString(line, 20, 100);
    g.drawString(goal, 20, 80);
    if(fm == null) { fm = g.getFontMetrics(); }
    int w = fm.stringWidth(line);
    g.fillPolygon(new int[]{20+w, 24+w, 28+w}, new int[]{105, 95, 105}, 3);
    for(int i = 0; i < months.length; ++i) {
      g.setColor(Color.yellow);
      g.fillRect(10+(i/6)*180, 120+(i%6)*50, 100, 25);
      g.setColor(Color.black);
      g.drawString(months[i], 20+(i/6)*180, 135+(i%6)*50);
    }
  }
  public static void main(String args[]) {
    JFrame frame = new JFrame();
    frame.add(new Sample16());
    frame.setSize(400, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

このプログラムの概要は次の通りです。

- RET が押されると、1～2 秒の範囲のランダム時間でタイマーを設定し、タイマーの時間が経過すると「問題」(英語の月名) を表示する。
- 普通のキーが押されると、表示文字列に連結する。表示文字列が月名の先頭部分に一意に一致するときは、表示文字列をその月名に置き換え、「補完位置」を設定する。
- 補完位置と同じ文字を入力したときは、補完位置を先に進めるだけで何もしない。

この最後の機能は、補完されたのにさらに続きを打つと余計な文字がくっついてしまうので嫌、という部分を直したものです。さて、このプログラムで「問題」に挑戦するとして、あなたはキーボードとマウスのどちらが高速だと思いますか?

3　マウス操作とメニュー

3.1　マウス操作の用途

前回、キーボードの入力時間を計測しましたが、あなたの 1 打鍵あたりの所要時間はどれくらいでしたか。キーボードの場合、習熟した人は「キーボードを見ないで」「手で触った反応で正しいキーを打って行く」(タッチタイピング) という方法で入力をしているため、高速かつ安定した速さで入力ができるようになっています。一般的に、打鍵に掛かる時間は (考えずにひたすら打っている状態だと)、1 打鍵あたり 100～500msec とされています。では、マウスとキーはどちらが速いか…というのは、非常にずるい質問で、もちろんどちらの速さも状況によって大きく変わって来ます。

では、マウスの操作に要する時間は何によって変わってくると思いますか? どのような可能性があるか考えてみてください。その答えはまた後で取り上げますが。

それは少し保留して、先に「マウスは何をするのに使うか」を考えてみます。何をするのに使いますか?

たとえば例を挙げてみます。

- アイコン、窓、テキスト、絵など対象物を「選択」する。
- 選択したものをドラグする。
- ボタンを押す、リンクをクリックする。
- 絵を描く。ジェスチャを入力する。
- メニューの選択。

この中で、少なくとも WIMP インタフェースでは、さまざまな「操作」(何をするか) をコンピュータに指示するときに、圧倒的に多く使われているのが「メニュー」です。そこでここではまず、メニューを取り上げて検討して行きます。(ところでメニューって何ですか?)

3.2 メニューとその得失

メニューとは、レストランとかで料理を選ぶときに…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。

3.3 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?)。

例によって以下に、Java で書いた計測プログラムを示します。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。

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

public class Sample21 extends JPanel {
  Font fn = new Font("Courier", Font.PLAIN, 16);
  GeneralPath[] panes = new GeneralPath[4];
  String[] label = new String[]{"A", "B", "C", "D"};
  int[] pos;
  int menux, menuy, mousex, mousey, probno = 0;
  boolean showmenu = false;
  String current = "-";
  long time = System.currentTimeMillis();
  public Sample21() {
    setOpaque(false);
    probno = (int)(Math.random()*label.length);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        time = System.currentTimeMillis();
        menux = mousex = evt.getX(); menuy = mousey = evt.getY();
        showmenu = true; repaint();
      }
      public void mouseReleased(MouseEvent evt) {
        long t = System.currentTimeMillis();
        System.out.println(label[probno]+" "+current+" "+(t-time));
        probno = (int)(Math.random()*label.length);
        showmenu = false; repaint();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
```

```
        mousex = evt.getX(); mousey = evt.getY(); repaint();
      }
    });
    // for experiment, modify following coordinates
    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[]{40,20, 40,60, 40,100, 40,140};
  }
  public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(Color.black); g2.setFont(fn);
    g2.drawString(label[probno], 50, 50);
    if(!showmenu) return;
    g2.translate(menux, menuy);
    current = "-";
    for(int i = panes.length-1; i >= 0; --i) {
      Color col = Color.yellow;
      if(panes[i].contains(mousex-menux, mousey-menuy)) {
        col = Color.pink; current = label[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 frame = new JFrame();
    frame.add(new Sample21());
    frame.setSize(400, 400);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

プログラムの基本的な構造は前回と変わっていませんが、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)。
- 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。
- showmenu は true だとメニュー表示中という意味になります。
- current は選択された項目のラベル文字列です。

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。
- 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。

前回同様、メソッド paintComponent() で窓の中身を描きますが、その概要は次の通りです。

- まず黒色で問題ラベルを表示。メニューが出ていないならそれで終わり。
- メニューの起点を原点 $(0, 0)$ になるように座標を移動。
- 現在の選択を「-」にしておく。
- 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。

演習 1 このプログラムをコピーしてきて動かし、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。

3.4 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 2 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。

演習 3 先のプログラムを手直しして、グラフ用紙に描いたメニューを実際に作って動作させ、時間計測してみなさい。他の人にもやってもらって、相互に比較すること。

4 Human Virtual Machine

4.1 人間のモデル化の必要性

ここまでに色々なものの時間を測って来ましたし、測れば測れることも分かりました。しかし、もし「測らなくてもモデルから時間が推測できる」なら、もっと様々な検討をしてから、よい見込みのあるものだけ作って試せることになり、とても嬉しいですね。つまり、「ユーザインタフェースに対した時の人間の行動」のモデルを作ることができれば、そのモデルで扱えるような側面については、モデルを用いてデザインを評価できるわけです。

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。

演習 4 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。

4.2 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。

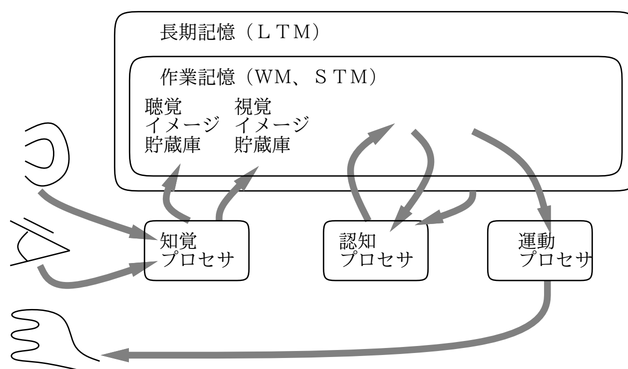

図 1: Model Human Processor の構造

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

4.3 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ね。

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

public class Sample22 extends JPanel {
  long time = System.currentTimeMillis();
  Color col = Color.blue;
  boolean showbox = true;
  public Sample22() {
    setOpaque(false);
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        requestFocus(); setTimer();
      }
```

```
    });
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        if(!showbox) { return; }
        long t = System.currentTimeMillis();
        char ch = evt.getKeyChar();
        if(ch >= ' ' && ch < 127) {
          System.out.println((t - time) + " : " + ch);
        } else {
          System.out.println((t - time) + " : " + (int)ch);
        }
        setTimer();
      }
    });
  }
  public void paintComponent(Graphics g) {
    if(!showbox) { return; }
    g.setColor(col); g.fillRect(100, 100, 50, 50);
  }
  public void setTimer() {
    showbox = false; repaint();
    Timer t1 = new 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 frame = new JFrame();
    frame.add(new Sample22());
    frame.setSize(200, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

このプログラムでは、色つき正方形の表示の有無を`showbox`という変数で制御していて、`setTimer()`という下請けメソッドの中で 2〜4 秒のランダム時間を選んでタイマーをセットするとともに四角を消し、タイマー完了時に四角を表示して時間を記録します。キーボードハンドラの方では記録した時間との差を表示し、`setTimer()` を呼んで次の時間待ちをします。最初だけはマウスクリックが必要で、このときは `requestFocus()` を呼んでから `setTimer()` するだけです。ではさっそく、実験してみていただきます。

演習 5 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. 四角が赤色なら左手のキー (たとえば F)、青色なら右手のキー (たとえば j) を打つとした場合、反応時間はどれくらい変わるか。

b. 同様にして、色の数を増やしていった場合、反応時間はどのように変わるか。

c. 色ではなく、1、2 の数字が表示されたら (または f、j の英字が表示されたら) そのキーを打つとした場合はどうか。

d. 数字や英字の数が増えていった場合、反応時間はどのように変わるか。

e. (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)。

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。

4.4 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 6 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。

では、先の紙でやった実験のプログラム版を示します。

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

public class Sample23 extends JPanel {
  long time = System.currentTimeMillis();
  int size = 20;
```

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

```
    int x0 = 60, y0 = 150;
    int x1 = x0 + size*8 - size/2, y1 = 150;
    public Sample23() {
      setOpaque(false);
      addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent evt) {
          long t = System.currentTimeMillis();
          int x = evt.getX(), y = evt.getY();
          System.out.println((t-time)+" ("+x+","+y+")");
          time = t;
        }
      });
    }
    public void paintComponent(Graphics g) {
      g.fillRect(x0, y0, size, size);
      g.fillRect(x1, y1, size, size);
    }
    public static void main(String args[]) {
      JFrame frame = new JFrame();
      frame.add(new Sample23());
      frame.setSize(800, 400);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
    }
  }
```

このプログラムは非常に簡単で、単に四角を 2 つ表示し、マウスクリックの時間間隔を次々に表示するだけです。実験に使うときは、正方形の 1 辺の大きさ `size` と、2 つの四角を `size` の何倍離すか (上のリスティングでは 8 倍) をさまざまに変えて試します。

演習 7 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。

ここまでに学んだ知見を元にすると、「メニュー選択を速くするには」どうするのがいいか、新たに分かったことがあると思います。そうしたら、演習 1 に戻ってもう 1 回「速く選択できるメニュー」

にチャレンジしてみるのはどうでしょうか。

5　おまけ: インタフェースと可視性

インタフェースのデザインでは、「ユーザが操作する何がインタフェースの何に結び付いているか」が見えること (可視性) が重要になります。ノーマンの冷蔵庫の例にヒントを得て、次のようなプログラムを作ってみました。これは、マウスをドラグすると左右の円の大きさが「ある法則により」変化します。目標は「すべての円の大きさを揃える」ことですが、あなたは何秒でできるでしょうか。

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

public class Sample24 extends JPanel {
  int[] rad = { 50, 50, 50 };
  int[] xpos = { 300, 120, 480 };
  int[] ypos = { 300, 300, 300 };
  int px, py;
  public Sample24() {
    setOpaque(false); randomize();
    addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent evt) {
        randomize(); repaint();
      }
    });
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent evt) {
        requestFocus(); px = evt.getX(); py = evt.getY();
      }
    });
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent evt) {
        int x = evt.getX(), y = evt.getY();
        int dx = x - px, dy = y - py;
        rad[1] += dx + dy; rad[2] += dx - dy;
        repaint(); px = x; py = y;
      }
    });
  }
  private void randomize() {
    rad[1] = (int)(10 + 90*Math.random());
    rad[2] = (int)(10 + 90*Math.random());
  }
  public void paintComponent(Graphics g) {
    for(int i = 0; i < rad.length; ++i) {
      g.setColor((rad[i]>=47&&rad[i]<=53) ? Color.yellow : Color.pink);
      g.fillOval(xpos[i]-rad[i], ypos[i]-rad[i],  rad[i]*2, rad[i]*2);
    }
```

```
    }
    public static void main(String args[]) {
      JFrame frame = new JFrame();
      frame.add(new Sample24());
      frame.setSize(600, 600);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
    }
  }
```

演習 8 「左右のマウスの大きさをマウスポインタで調節する」という目標を実現する、別のインタフェースを考えて実装してみなさい。それが上の例題より優れているとしたら、それは可視性の違いによるものかどうか、検討しなさい。

6 宿題

今回は Reading として、次の 2 つを用意しました。

- 高橋由美子, インタフェースの心理学 — 認知情報処理からのアプローチ —, in 淵監修, 古川・溝口編, インタフェースの科学, 共立, pp. 49-75, 1987.
- ニューマン, ラミング著, 北島監訳, インタラクティブシステムデザイン 13 章, ピアソン, 1999.

前者は今回少し紹介した MHP をきちんと解説したものです。もう 1 つはメンタルモデルの作り方に関するもので、今回の授業ではこういう話はあんまりできないので、読んでみて頂きたいと思いました。

あとは、演習課題については本資料にあるもので「やってないもの」「まだやり足りないもの」を 1 つ選んでやってみて頂ければと思います。