# ユーザインタフェース＃3 — GUIとGUI部品

久野　靖*

2012.10.23

## 1　人間の認知モデル/メンタルモデル — Readingから

### 1.1　「インタフェースの心理学」

これは「インタフェースの科学」という本の中の1章ですが、前回取り上げたHuman Virtual Machine (人間の認知情報処理モデル) のきちんとした解説です。授業では正確にお話できなかったことも詳しく書かれているので参考になったと思いますがどうでしょう。なお、Fitzzの法則のところは前回積み残したので今回実験をやります。

### 1.2　「ユーザのメンタルモデル」

これは「インタラクティブシステムデザイン」という結構古い訳本の1章ですけれど、今でも十分通用するよい解説になっています。これまでにも、ユーザのメンタルモデルが大切だという話はしましたが、具体的にどういうことを考えるというのはやっていませんでした。この文では、まずメンタルモデルがどのようなものか、なぜ大切かを電話などを例に挙げています。すごく思い当たりますよね。電流の「群衆流モデルと水流モデル」とか面白いです。

次に、典型的なメンタルモデルの具体例が挙げられています。いずれも、なるほどこういうモデルあるよね、と思えるしそれぞれ興味深いと思います。

- 状態遷移モデル
- オブジェクト・アクションモデル
- マッピングモデル
- 類推モデル

その後の、因果関係のモデルがちゃんと作れているかどうかが熟練者と初心者の違い、みたいな話も大変納得できるところです。全体としてとてもためになりますが、「じゃあどうやったらうまくメンタルモデルが作らせられるの?」については解答は (やっぱり) 無いんですね。^_^;

## 2　Human Virtual Machine (つづき)

### 2.1　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。

*経営システム科学専攻

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 1** 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 1 に示します。

表 1: $I_m = 100$ 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 |

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;
  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 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。

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、前回に戻ってもう 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。

# 3 GUI (Graphical User Interface)

## 3.1 GUI の定義、GUI の歴史

例によって: GUI (Graphical User Interface) の定義は何でしょうか?

GUI とは CUI、つまりキーボードから入力するコマンドと文字表示による応答と対比して作られた概念であり、「画面上にグラフィクスが表示され」「ポインティングデバイス等を通じて画面上を指すことでさまざまな操作が指示できる」インタフェースだと言えます。

GUI のはじまりは初回に説明した Xerox の Alto ですが、実際にはそれより前にもごく少しだけ、グラフィクスを直接指すようなインタフェースがありました (いずれも研究用の非常に高価な単品システム)。

- Ivan Sutherland の Sketchpad — 最初の対話的グラフィクスシステムであり、画面上の図形を指して操作できた (さまざまな指示を出したというわけではない)。

- Douglas Engelbart の NLS — 画面上にハイパーテキストが表示され、リンクを選択するとリンク先に飛べる (ちなみにこれのために Engelbart はマウスも発明した)。

さて Alto ですが、GUI を用いたさまざまなプログラム (エディタとか作図ソフトとかワープロソフトとか) がありましたが、これらはおおむね Mesa という言語で書かれていました。このほか、Smalltalk-80 言語も動いていて、これはこれ自体の中で閉じたグラフィクス環境でした。

このころ、GUI と関連して言われていたことに次のものがあります。

- モードは少ない方が (無い方が) よい。
- 操作を指示するときの順番は、これまでの「コマンド→対象 (オペランド)」よりも「オペランド→コマンド」の方がよい。

モードとは簡単にいえば、コマンドの働きが異なるような複数の状態を言います。たとえば、Emacs 以前のコマンド (ed とか vi) では「入力モード」と「コマンドモード」が分かれていましたが、Emacs ではすべてのコマンドを Control-○、Meta-○などに割り当てることで、「モードなし」を目指したわけです (でも細かいモードはあるけれど)。モードがあると当然、どのモードかを意識しなければならないので、インタフェースに対する敷居は高くなります。

次に 2 番目のはどうでしょうか。実は Alto のインタフェースでは「コマンド (たとえば削除など) を選択」→「対象 (範囲) を指定」という順番で操作していましたが、それだとコマンドを選んだ後は「削除モード」ということになってしまいます。では今の GUI では? まず「範囲を選択」しておくことができ、次にその範囲に対して「削除」「コピー」「色つけ」などさまざまなコマンドが後から選べます。これならモードが無くなるわけです。しかしこの方式も産まれてから 20 年以上たっていますので、このままでいいかどうかは分からないところもありますが…

実際に GUI が広く普及したのは、これも既に説明しましたが、Apple の Lisa/Mac が GUI を採用し、続いて Windows がこれを追いかけて広まったことによります。そして、これらのシステムの GUI は「WIMP インタフェース」と呼ばれるのでしたね。

- Windows — 個別の内容/用途に応じた「四角い領域」
- Icons — さまざまな対象を表す「小さな絵」
- Menu — 操作を選択する手段 (定義は前回)
- Pointing Device — マウス、トラックボール、手、…

WIMP インタフェースは、標準化された部品を組み合わせて作るために作りやすく (後述)、また始めての人でも比較的容易い使えるようになることから、広く普及しました。結局、1980 年代〜2010 年くらいまでは WIMP の時代だということになるでしょう。

しかしその後、小さな画面しか持たないモバイル機器 (GUI 部品を出しておくのが困難) の普及や、タッチパネルをはじめとする入力機器の発達から、現在では「ポスト WIMP」なインタフェースが増えて来ています。これについては初回にやりましたし、次回にも少し取り上げます。

### 3.2 GUI 部品

ここまでにも出て来たように、WIMP 型の GUI では「ボタン」「メニュー」などの標準的な「ユーザとやりとりするための部品 (GUI 部品)」が用意されていて、それらを組み合わせることであまり苦労なく様々な用途のインタフェースを作ることができます (といっても、このような技術が確立するまでは結構大変でしたが)。

初期の GUI では C など手続き型言語のライブラリとして GUI 部品を実装していたので、色々と大変でした。しかし現在では、GUI 部品はオブジェクト指向言語のオブジェクトとして実装されているのが普通なので、「必要に応じてさまざまなオブジェクトを持って来て組み合わせる」ことで GUI が構成できます。

また、プログラムの構造も、初期のプログラムではイベントループ (イベントドリブンなプログラムで、イベントを取得しては内容によって枝分かれしてそのイベントを処理することを繰り返す構造) を自分で書いていましたが、現在では「部品ごとにその部品に対するイベントのハンドラを設定する」というより自然な形に変化してきています。その細かいことは次節以降で。

**演習 3** どのような「GUI 部品」が思い浮かぶか、、できるだけ沢山挙げてみなさい。ついでにその部品の「定義」「用途」も述べなさい。

### 3.3 Swing の部品群

Swing は Java の GUI 部品群です (歴史的には古いものが AWT、その後出て来て現在メインで使われているのが Swing)。そこに含まれている部品として主要なものを以下に挙げておきます。これらの利用方法の詳細については、JDK 1.6 API ドキュメントを参照してください。

#### 基本となる部品

- JButton — ボタン
- JCheckBox — チェックボックス
- JComboBox — コンボボックス (ドロップダウンリスト)
- JEditorPane — エディタ
- JLabel — ラベル (文字列を表示する)
- JList — 行のならびを表示
- JProgressBar — 進捗表示バー
- JRadioButton, ButtonGroup — ラジオボタン
- JScrollBar — スクロールバー
- JSlider — スライドレバー
- JTextArea — テキスト入力領域
- JTextField — テキスト入力欄
- JTable — 表 (テーブル)
- JTree — ツリー表示

#### 窓になる部品

- JFrame — 単なる窓
- JDialog — ダイアログボックス
- JFileChoser — ファイル選択ダイアログ

#### メニュー関係の部品

- JMenuBar — メニューバー
- JMenu — メニュー
- JPopupMenu — ポップアップメニュー
- JMenuItem — メニュー項目
- JSeparator — メニューの「区切り線」

コンテナ (中に別のものが入る部品)

- JPanel — ただの四角い領域
- JScrollPane — 内容がスクロールできるような領域
- JSplitPane — 内容が 2 分割できるような領域
- JTabbedPane — タブを選ぶと内容が切り替えられる領域
- Box — 内容を縦または横に詰められる箱

### 3.4 Swing の部品を窓に入れる

ではさまざまな部品があることは分かったので、とりあえず基本的な部品を窓に入れてみることから始めましょう。とりあえず、これらの部品には共通で次のメソッドが使えることを注意しておきます。

- setBounds($X$, $Y$, 幅, 高さ) — 部品の位置と大きさを設定
- setForeground(色), setBackground(色) — 文字色/背景色を設定
- setFont(フォント) — 文字表示に使うフォントを設定。

では、GUI 部品を入れるだけのプログラムです。GUI 部品は自動的に自分の形を表示するので、このプログラムはではコンストラクタ中で初期設定を行うだけでよく、これまでのように `paintComponent()` を定義して内容を描く必要はありません。このプログラムでは、冒頭部分で各種の GUI 部品オブジェクトを生成すること、コンストラクタの冒頭で自動配置を off にして、あとは各部品を `add()` で領域に追加し、`setBounds()` で位置を設定しています (スライダなどは細かい設定も追加していますが)。

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

public class Sample31 extends JPanel {
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JTextField f1 = new JTextField();
  JLabel l1 = new JLabel("Label 1");
  JCheckBox c1 = new JCheckBox("check");
  JRadioButton r1 = new JRadioButton("red", true);
  JRadioButton r2 = new JRadioButton("green");
  ButtonGroup g1 = new ButtonGroup();
  JComboBox m1 = new JComboBox(new String[]{"Red","Blue","Green"});
  JList i1 = new JList(new String[]{"A","I","U","E","O"});
  JScrollPane s1 = new JScrollPane(i1);
  JSlider d1 = new JSlider(0, 100, 25);
  JTextArea a1 = new JTextArea();
  JScrollPane s2 = new JScrollPane(a1);

  public Sample31() {
    setLayout(null);
    add(b1); b1.setBounds(30, 20, 90, 25);
    add(b2); b2.setBounds(130, 20, 90, 25);
    add(f1); f1.setBounds(230, 20, 90, 25);
```

```
    add(l1); l1.setBounds(30, 50, 90, 25);
    add(c1); c1.setBounds(30, 75, 90, 25);
    g1.add(r1); g1.add(r2);
    add(r1); r1.setBounds(30, 100, 90, 20);
    add(r2); r2.setBounds(30, 120, 90, 20);
    add(m1); m1.setBounds(30, 145, 90, 25);
    add(s1); s1.setBounds(30, 175, 90, 50);
    d1.createStandardLabels(20);
    d1.setMajorTickSpacing(20);
    d1.setMinorTickSpacing(10);
    d1.setPaintLabels(true); d1.setPaintTicks(true);
    add(d1); d1.setBounds(150, 55, 180, 50);
    add(s2); s2.setBounds(150, 115, 180, 90);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample31());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

**演習 4** このプログラムをコピーしてきて動かせ。動いたら、部品の大きさや位置を変更してみなさい。また、別の種類の部品を追加してみてもよい。

**演習 5** 次のようなプログラムの画面をここまでに使った GUI 部品を使ってデザインしてみなさい。

a. 摂氏の温度を華氏に (またはその逆に) 変換する。どちらの変換かは実行中に選択できること。

b. 2 つの整数値を入力してもらい、最大公約数を表示する。

c. RGB の 3 原色の値 (0〜255 の整数) を受け取り、その色をどこかに表示する。

1. デザインを方眼紙に作成しなさい。
2. そのプログラムで 1 組のデータを処理するのに掛かる時間を予測しなさい。
3. その画面配置を実現するプログラムを作成しなさい。
4. その画面上で実際に操作してみて時間が合っているかチェックしなさい。

### 3.5 部品に動作をつける

ここまででやったところでは、画面に部品を入れることはできましたが、ボタンを押しても何も起きません。動作を記述していないのだから当然ですが。これまでは「窓に対して」マウスイベントやキーイベントのハンドラを設定してきましたが、ここでは「部品に対して」動作をつけるやり方を説明します。

マウスやキーボードのイベントが、入力機器からの操作に直接対応していたのに対し、部品に対する動作は「ボタンが*押された*」「スライダが*変化させられた*」などのように、それぞれの部品の意味に応じたものになっています。細かい理屈はこの授業が Java の授業ではないので省略して、やり方だけ説明します。

ボタンに対するイベントは `ActionEvent` なので、次のようにして動作を設定してください。

```
ボタン.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    ここに動作の内容を書く
  }
});
```

スライダなど値が変化するものは ChangeEvent なので、次のようにして動作を設定してください。

```
ボタン.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent evt) {
    ここに動作の内容を書く
  }
});
```

どんな部品がどんなイベントを発生させ、ハンドラはどんな名前でつけるかなども、すべて API ドキュメントを見れば分かります。

では簡単な例として、単純な摂氏華氏変換の画面において、計算ボタンを押したときに計算するようにしてみます。

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

public class Sample32 extends JPanel {
  JButton b1 = new JButton("Convert");
  JTextField f1 = new JTextField("0.0");
  JLabel l1 = new JLabel("From:");
  JLabel l2 = new JLabel("To:");
  JLabel l3 = new JLabel("");
  JComboBox m1 = new JComboBox(new String[]{"F->C","C->F"});

  public Sample32() {
    setLayout(null);
    add(l1); l1.setBounds(30, 20, 90, 25);
    add(f1); f1.setBounds(130, 20, 90, 25);
    add(l2); l2.setBounds(30, 50, 90, 25);
    add(l3); l3.setBounds(130, 50, 90, 25);
    add(m1); m1.setBounds(30, 80, 90, 25);
    add(b1); b1.setBounds(130, 80, 90, 25);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        double r, d = Double.parseDouble(f1.getText());
        if(m1.getSelectedIndex() == 0) {
          r = 5.0 / 9.0 * (d - 32.0);
        } else {
          r = 9.0 / 5.0 * d + 32.0;
        }
        l3.setText(String.format("%.2f", r));
      }
    });
```

```
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample32());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

要は、ボタンが押された時に入力欄の内容を取り出して数値に変換し、次に選択メニューの現在の選択番号に応じて摂氏華氏か華氏摂氏の変換を行い、結果を文字列にして出力欄に設定する、ということです。

もう 1 つの例として、スライドレバーを動かすと摂氏と華氏の値が対応して変化する、というインタフェースに動作をつけたものも示します。

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

public class Sample33 extends JPanel {
  JLabel l1 = new JLabel("degree C.");
  JLabel l2 = new JLabel("degree F.");
  JLabel l3 = new JLabel("0.00");
  JLabel l4 = new JLabel("32.00");
  JSlider d1 = new JSlider(-50, 100, 0);

  public Sample33() {
    setLayout(null);
    add(l1); l1.setBounds(30, 20, 90, 25);
    add(l3); l3.setBounds(130, 20, 90, 25);
    add(l2); l2.setBounds(30, 50, 90, 25);
    add(l4); l4.setBounds(130, 50, 90, 25);
    add(d1); d1.setBounds(30, 80, 300, 50);
    d1.createStandardLabels(20);
    d1.setMajorTickSpacing(20);
    d1.setMinorTickSpacing(10);
    d1.setPaintLabels(true); d1.setPaintTicks(true);
    d1.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent evt) {
        double d = d1.getValue();
        double r = 9.0 / 5.0 * d + 32.0;
        l3.setText(String.format("%.2f", d));
        l4.setText(String.format("%.2f", r));
      }
    });
  }
  public static void main(String[] args) {
```

```
    JFrame app = new JFrame();
    app.add(new Sample33());
    app.setSize(400, 300);
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

こちらはスライダの値範囲-50 度〜100 度を予め設定してあるので (摂氏の温度のつもり)、スライダの値変化したときにこの値とこれを華氏に変換した値とを出力ラベルに設定するだけです。さて、この 2 つのインタフェース (と、あなたが考えたもの) のうちどちらが操作が速いでしょうか?

**演習 6** 上記の例題 (およびあなたの版) それぞれの操作時間を予測しなさい。その上で、それらを実際に動かして所要時間を実際に計測してみなさい。2〜3 人で組になって、問題 (「摂氏○度は華氏何度?」「華氏×度は摂氏何度?」) を紙に書いてぱっと見せ、答えが得られるまでの時間を計測します。

### 3.6 部品の自動配置

ここまでは部品をすべて位置と大きさを数値で指定して設定してきましたが、この方法だと窓のサイズが変更されたときに悲しいことになります。そこで実際の GUI プログラムでは、使用できる領域のサイズに応じて自動的に部品を配置する機能が備わっているのが普通です。

Java ではそのようなことを行うための機能を「レイアウトマネージャ」と呼びます。そして、さまざまな機能 (と複雑さ) のレイアウトマネージャが予め用意されています。とくに簡単なものとして、次の 2 つがあります。

- `FlowLayout` — 単に部品を左から順につめて行く。`JPanel` ではこれが標準で設定されている。
- `BorderLayout` — 部品の位置を「上」「下」「左」「右」「中央」のいずれかで指定する。`JFrame` ではこれが標準で設定されている。

ここでは、もっと柔軟なレイアウトができる `BoxLayout` について簡単に説明しておきます。`BoxLayout` では、領域を沢山の「箱」から成っているものと見なし、箱どうしを縦や横にならべて、間に「詰めもの」を詰めることで間隔を調整するようになっています。次の例では、領域の中を上と下の箱に分け (間は 5 ピクセルあき)、上の箱には横にボタンが 3 つ並び、下の箱には横に入力欄が 3 つ並ぶ (間は 3 ピクセルあき)、というふうに設定してあります。

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

public class Sample34 extends JPanel {
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  JTextField t1 = new JTextField("1");
  JTextField t2 = new JTextField("2");
  JTextField t3 = new JTextField("3");
  Box x1 = new Box(BoxLayout.X_AXIS);
  Box x2 = new Box(BoxLayout.X_AXIS);
```

```
  public Sample34() {
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    add(x1); add(Box.createVerticalStrut(5)); add(x2);
    x1.add(b1); x1.add(b2); x1.add(b3);
    x2.add(t1); x2.add(Box.createHorizontalStrut(3));
    x2.add(t2); x2.add(Box.createHorizontalStrut(3));
    x2.add(t3);
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample34());
    app.setPreferredSize(new Dimension(280, 80));
    app.pack();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }
}
```

# 4 Keystroke Level Model (KLM)

認知情報処理モデルを使うと、「画面上の対象物の位置や、人間の判断に要するサイクル数などまですべて分かっていれば」特定タスクに要する時間が見積もれることは既にお分かりだと思います。しかし、それを正確に計算するのは非常に大変です。

そこで Card, Moran, Newell はもっと簡単なモデルとして、「キーやマウス等の操作の列だけ」が分かればそれを元にして大体の見積もりができる、というモデルも作成しました。これを「キーストロークの水準での」見積もりを行うことから、Keystroke Level Model (KLM) と呼びます。KLM では、ユーザインタフェースの操作を次のように抽象化します。

- K → 打鍵動作 (キー 1 つ) → 0.35 秒
- H → キーボードとマウス間での手の移動 → 0.4 秒
- P → マウスポインタの目標への移動 → 1.1 秒
- R → マシンのレスポンスを待つ → 実際に掛かる時間
- M → 「メンタル操作」 → 1.35 秒

たとえば、「File メニューから Save As を選び、ファイル名 `file-2.4` を入力し、リターンする」だと、次のようになるはずです。

```
H P K R P K R H K K K K K K K K K
   menu    sel  f i l e - 2 . 4 [RET]
```

一方、Emacs だと同じ動作が「CTRL-x CTRL-w file-2.4 RET」ですから、次のようになるはずです。

```
K K K K R K K K K K K K K K
C-x C-x   f i l e - 2 . 4 [RET]
```

しかし、これで K の時間に 0.35、マシンの応答時間としてたとえば 0.2 を入れると、明らかに短かすぎます。これは人間が「考える時間」が入っていないためです。そこで、次のような規則で M を挿入します。

- 引数文字列 (まとまったテキストや数値) の一部でないすべての K の直前に M を挿入する。また、コマンドを選択する P の前にも M を挿入する。
- M の直後の操作が M の直前の操作から完全に予測される場合はその M を削除する。
- MK の列が認知の単位 (たとえばコマンド名) に属しているなら、最初の M だけを残して残りの M を削除する。
- K が冗長な区切りであれば、その直前の M を省く。
- K が不変な文字列 (たとえばコマンド名) を終わらせる区切りであるなら、その直前の M を削除する。

これらの規則を適用すると、上の 2 例はそれぞれ次のようになるでしょう。

```
H M P K R M P K R H 8K M K → 11.65 秒

M K K K R 8K M K → 7.10 秒
```

KLM は明らかに非常に「おおざっぱな見積もり」のようですが、実際にやってみると結構よく合うとされています。

**演習 7** 演習 6 で行った時間計測について、KLM による操作時間の見積もりを行ってみて、どのくらい一致するか調べなさい。

# 5 ユーザインタフェースとフレームワーク

## 5.1 フレームワークと MVC

フレームワーク (framework) とは、そのままでは「枠組み」という意味の英語ですが、ソフトウェアの文脈でいうと「プログラムの構造を一般化したもの」という意味合いがあります。たとえば、自動車の設計は車種ごとに様々ですが、「車体に車輪がついていて、エンジン→クラッチ→変則器→車軸の順に動力を伝達して駆動」というフレームワークは一緒です。これと同様に、ユーザインタフェースを持つソフトウェアや、単体の GUI 部品についても、「何でも勝手に設計してね」というよりも「このようなフレームワークに沿って作ります」という形にした方が設計も理解しやすく、動作にも統一性が持たせられるわけです。

フレームワークの代表的なものに、Alto の Smalltalk-80 システムで最初に作り出された「MVC(Model-View-Controller) フレームワーク」があります。これは、ユーザインタフェースを持つソフトウェアを次の 3 つの要素に分解して扱うものです。

- Model — インタフェースによって操作されているものの「本質部分」。たとえば 3 次元グラフィクスなら「空間内の物体の形状や色や光源の配置や明るさ」など。
- View — モデルを画面に表示している部分。1 つのモデルに対してビューが複数あってもよいし (例: 立面図、平面図、…)、ビューごとに表示のさせ方が違ってもよい。
- Controller — スライダなど、モデルやビューの見え方に変更を及ぼすための機能を持つ部分。

MVC の 3 要素はそれぞれ互いに依存関係を持ちます。たとえば、コントローラの操作によってモデルが変更を受け、モデルが変化すると、それを表示しているビューの内容が変化する必要があります。このような MVC のフレームワークに基づいて構成することで、さまざまなユーザインタフェースがうまく作れることが分かった、というのが Smalltalk-80 の貢献です。

さて、一般的なインタフェースは MVC もいいのですが、「GUI 部品」のレベルで考えると規模が小さすぎて 3 つには分けにくいです。そこで、Swing では V と C を合わせたものを delegate と呼び、これとモデルとが合わさったものが GUI 部品、という考え方を取っています。モデルと delegate の

間のインタフェースは部品ごとに決まっています。そして、とくに複雑な機能を持つ部品では、プログラマが複雑なモデルを提供し、それをもとに GUI 部品が表示と操作を受け持つ、ということができます。具体的には JList、JTable、JTree、JTextPane、JEditorPane などがそうです。

ただし、これらの部品それぞれについて、いちいちモデルから書かないと使えないというのでは不便なので、「デフォルトの」モデルも用意されています。たとえば、JTable のためのデフォルトのモデル DefaultTableModel は、縦横のセルに値を保持するようなものです (確かにそれが一般的ではあります)。しかし、自前でモデルを用意することで、これとは違ったものが色々実現できます。

### 5.2 例題: JTable

上で説明したように、JTable は (ちょうど Excel の画面のように) 縦横にます目が並んだ形の GUI 部品です。そして、その中に表示されるものは TableModel というインタフェースに従うモデルオブジェクトによって定まります。

ここでは、JTable 向けにさまざまなモデルを作るときの土台として使える AbstractTableModel オブジェクトを継承して、簡単なモデルを作ってみます。作成するクラスで必ず定義しなければならないメソッドは次のものです。

- `int getRowCount()`、`int getColumnCount()` — 表の行数、列数を返す。
- `boolean isCellEditable(int r, int c)` — $r$ 行 $c$ 列が編集可能かどうかの真偽値を返す。
- `Object getValueAt(int r, int c)` — $r$ 行 $c$ 列に表示すべき内容を返す。
- `setValueAt(Object o, int r, int c)` — $r$ 行 $c$ 列に設定すべき内容を渡される。

ここで「内容」としては Object をやりとりしますが、実質としては文字列がやりとりされると思っておけばいいでしょう。

では、「等差数列を表示する」という機能を持ったモデルを使った例を見てみましょう。このプログラムでは、BorderLayout を使って中央に JTable をスクロール機能つきではめるだけで、あとの処理はすべてモデルの中で行っています。

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

public class Sample35 extends JPanel {
  public Sample35() {
    setLayout(new BorderLayout());
    add(new JScrollPane(new JTable(new MyTableModel())));
  }
  public static void main(String[] args) {
    JFrame app = new JFrame();
    app.add(new Sample35());
    app.setPreferredSize(new Dimension(300, 400));
    app.pack();
    app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    app.setVisible(true);
  }

  class MyTableModel extends AbstractTableModel {
    double init = 1.0, step = 0.1;
```

```
    public int getRowCount() { return 25; }
    public int getColumnCount() { return 3; }
    public boolean isCellEditable(int r, int c) {
      if(r == 1 && c == 1) return true;
      return false;
    }
    public Object getValueAt(int r, int c) {
      if(c == 0) {
        return new Integer(r);
      } else if(c == 1) {
        return String.format("%.5f", 1.0 + r*step);
      } else {
        return "?";
      }
    }
    public void setValueAt(Object o, int r, int c) {
      if(r == 1 && c == 1) {
        step = Double.parseDouble(o.toString()) - 1.0;
      }
      fireTableDataChanged();
    }
  }
}
```

モデルクラスは上述のように `AbstractTableModel` を土台としていて、その概要は次の通りです。

- 初期値では初校は 1、階差は 0.1。
- 表の行数は 25、カラム数は 3。
- 1 行 1 列のみ編集可能。
- 0 カラムには行番号、1 カラムには等差数列、2 カラムには「?」を表示。
- 1 行 1 列に値がセットされたら、実数値に変換して初校を引くことで新しい階差を設定。

なお、`fireTableDataChanged()` は「内容が変わったから表示し直して」という意味のメソッドです。

**演習 8** 例題をそのまま動かしなさい。動いたら、次のような変更を行ってみなさい。

a. カラム 2 にカラム 1 の 2 倍の値が表示されるようにする。
b. さらに、カラム 4 まで用意して 3 倍、4 倍も表示する。
c. カラム 1 行 0 も編集可能にして、初項を入力できるようにする。
d. カラム 1 の他の行も編集可能にして、それに応じて階差を計算する。

## 6 宿題等

今回も興味を持った課題はさらにやって頂くとよいかと思います。Reading については次のものを用意しましたので次回までに読んできてください。

- Jef Raskin 著, 村上訳, ヒューメイン・インタフェース, ピアソン, 2001. の 3 章「意味・モード・モノトニーそして神話」と 4 章「定量化」

あとは、第 5 回の論文紹介の準備をお願いします。論文を私あて知らせていない人は出席メールのついでに知らせてください。よろしく。