オブジェクトシステム 2002 ＃5

久野　靖*

2003.2.19

1　はじめに

今回も出席メールをまとめます。

吉武さん インターフェイスを用いてシステムに柔軟性を持たせることに関して言うと、Java などの言語のインターフェイス表現は呼ばれる側のものしか分離して表現できないので、ソケットとプラグの対応アナロジーからするとまだ半分に思えます。呼ぶ側のホットスポットも分かるようになっていると言う意味では、ロリポップに線を引く UML の設計図にもコードより長じた点があるといえます。最近のコンポーネント記述言語などではこの辺は考慮されているのでしょうか。(それともあまりそうした必要性はない?)

Thread に象徴される並行性の記述は、人間が本来的に並行動作を認識するのが難しいのか、1 次元のプログラミング言語記述に頼るから難しいのか、(私にとっては) 感覚的には難しいモノですね。慣れによってそこそこかけるようにはなるものの、どうも制御している感覚にはなれません。それでも Java のスレッドによってほどほどの (自由すぎて破壊的になることが少ない) 並行性が書けるようになって楽にはなりました。

中山さん インタフェースによる設計は, 設計指針は示されているので初心者でもすぐに利用可能な程度だとは思いますが, 本書の例が良くないのか, 少し分かりづらい部分があるのと, もう少し集約できそうな雰囲気はあります.

assertion は, きちんと使おうとすると負担が大きいという印象がありますが, インタフェースと一緒に設計レベルから各部で必要な要件を明確にしておくという意義はあるような気があります. もし使うとすれば, 設計とコーディングの役割分担して行う場合になるのでしょうか. 吉武さんの資料の OCL からも, そういう方向性に向かっているような気がします. 一番現実的な assertion の使い方は, 自分の担当するクラスの中でのデバッグというイレギュラーなものでしょうか.

thread はさわりだけでしたが, グリッドコンピューティングも商用化が進んでいますので, 分散コンピューティングの利用とともに thread を使った設計は当然となるのでしょう. thread を使う場合には, 変数の更新タイミングや順序関係をきちんと把握しておかないと, まったく意図しない結果になってしまうので, なんとなく難しいイメージがあります.

西森 先週の講義での話からあとで思ったことなのです。

○ 実装を意識しないインターフェースはダメインターフェースでは?

低レベル (低脳ではないです…) な現場にいるせいか、インターフェースの設計の段階で効率をよく考えていなくてインターフェースが効率の足を引っ張ったりすることが時々あります。

少なくとも仕事プログラマーの人たちで「メモリも CPU も無限大」と思ってプログラムを書いている人はいないと思うので、インターフェースは必ず実装も意識しなければならないとは思うのですが。

*筑波大学大学院経営システム科学専攻

例えば、何か巨大でへんてこな SQL サーバーを使うプログラミングなら、その SQL サーバー意識しない設計はやっぱりダメな設計だと思います (そういったプログラミングをしたことないので想像です)。

○ 設計といってもいろいろ段階があったりすると思います。

大きなプロジェクトならとてもメタな設計をする人から実装直前の設計とか、実装の設計 (!) とかいろいろあると思います。教科書は結構実装に近い雰囲気がするのですが (デザインパターンとか後半多く出てきますし)。

でも、航空機予約システムとかサンプルが巨大なのでちょっとわかりづらい、というのもあるのですが…。

スレッド

私の担当部分を先生の話で完了にされてしまいました…。どうでもいい話なんですが、最近の家庭用機にもスレッドはあります。でも、

○ バックグラウンドで CD 回してデータを読みながらプログラム実行
○ VSYNC 処理 (テレビの垂直帰線期間の処理) を割り込みでやると、何かと面倒なので、スレッドにしてしまう。
○ 計算処理と描画処理でスレッドを分ける

ぐらいでしか使ったことないです。最後のは、並行同時処理ではなくて、コンテキストを別にするためです。描画処理はやたら複雑でごちゃごちゃするので、計算と描画のループが分かれるだけでも結構楽になったりします。(2 番目のも同時に処理してないですが)。

2 本読み (4 章)(可能なら 5 章も?)

3 並行プログラミングと同期

3.1 排他領域と Java の synchronized

並行性→ Java ではスレッドモデルによる (従来型の) 並行モデル。

並行性がある場合、並行実行単位間での「干渉」が問題となる。たとえば「カウンタを増やす」という簡単な例題でも (ただの「`count += n`」なのに!) 複数スレッドが並列実行するとメチャクチャな値になる。

まず最初に考えるべきことは、できるだけ並行実行単位どうしが「やりとり」する機会を減らす。減らせば減らすほど、性能上もプログラム設計上も有利。いい例→行列などの大量データの並列計算。

しかし、どうしても複数の並行実行単位どうしでの干渉が避けられない箇所 (共通のデータをいじくる箇所) は存在する→そこはどうするか? →排他領域==「鍵を掛けてから使う」

```
lock
  ... // 干渉を避けたい (一人だけで実行したい) 動作
unlock
```

しかしこれでは弱点が…

通常は、干渉を避けたい「島」「かたまり」が複数ある→それぞれに別の鍵を用意するのがよい

```
lock(object)
  ...use object... // 干渉を避けたい動作
unlock(object)
```

これにも弱点が…

鍵を掛けたらかならず外すことが必要だが、そのことが言語上は強制できない。これを解決するためには、言語の構文に組み込む。

```
synchronized(object) {
  ...use object... // 干渉を避けたい動作
}
```

または object のインスタンスメソッドで

```
synchronized aMethod(...) {
  ...use object... // 干渉を避けたい動作
}
```

ただし後者の場合、鍵を掛けたい最小範囲だけを含むメソッドになるように設計に注意する。

4 条件同期

上のように排他領域があればそれで済むというわけではない (なぜか?)。
鍵を掛けて仕事に掛かろうと思ったら、今ある材料では用が足りないと判った場合どうするか…

(1) いったん鍵を明けて、寝て待つ。

(2) 別の人が鍵を掛けて入ってくる。

(3) その人が、材料を用意したりしてくれる。

(4) その人が、「いいですよ」と起こしてくれる。

(5) その人が鍵を明けて出て行くと起きることができる。

(6) いざ材料がそろったので仕事に掛かる。

しかし…次のような問題が。

(1) 別の人が実は自分と同じ材料を待っている人だったら?

(2) 自分も誰かを起こさないといけないの?

(3) 誰でも一人起こせばいいの? 適切な人を起こさないといけないの?

このように、「ある条件が満たされるまで待つ」のを「条件同期」と呼び、並行性のある言語では重要な機能。条件同期を表す方法も色々あるが…

- パス式
- ガード
- 受理集合
- 抽象状態同期 ←久野
- wait/notify ← Hoare のモニタ…Java でもこれを採用

たとえば、典型例として「バッファに読む人とバッファから書く人の競合」を例題にしてみた。

```
public class Sample51 {
  public static void main(String args[]) {
    Buffer buf = new Buffer();
    Putter p1 = new Putter('A', 500, buf);
    Putter p2 = new Putter('B', 1500, buf);
    Getter g1 = new Getter('a', 1000, buf);
    Getter g2 = new Getter('b', 1000, buf);
    p1.start(); p2.start(); g1.start(); g2.start();
    try {
      p1.join(); p2.join(); g1.join(); g2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
  static class Buffer {
    char data;
    boolean full = false;
    synchronized void put(char c) {
      while(full) try { this.wait(); } catch(InterruptedException e) { }
```

```
      data = c; full = true; this.notifyAll();
    }
    synchronized char get() {
      while(!full)  try { this.wait(); } catch(InterruptedException e) { }
      char c = data; full = false; this.notifyAll();
      return c;
    }
  }
  static class Putter extends Thread {
    int count;
    char ch;
    Buffer buf;
    public Putter(char c, int n, Buffer b) { ch = c; count = n; buf = b; }
    public void run() {
      for(int i = 0; i < count; ++i) buf.put(ch);
    }
  }
  static class Getter extends Thread {
    int count;
    char ch;
    Buffer buf;
    public Getter(char c, int n, Buffer b) { ch = c; count = n; buf = b; }
    public void run() {
      for(int i = 0; i < count; ++i) {
        char c = buf.get();
        System.out.println(ch + "[" + i + "] got char " + c);
      }
    }
  }
}
```

これを動かすとちゃんと並列に動く。しかし `notifyAll()` を `notify()` に変えるとパタッと止まってしまう。

Java では「複数の条件に対応した複数の寝かた」を直接サポートしていない→かなり不便、非効率。全員を起こして、起きた全員が「自分がやってもいい番かどうか調べてまだだめなら寝直す」ことに。

結局私 (久野) としては、Java の条件同期はせっかく効率がよかった (しかし goto みたいで危険だけど)Hoare のモニタをいじくっていいところを消してしまった (悪いところは同じまま) 代物みたいに思える。

5 アダプタと内部クラス

アダプタとは教科書にもあるように、あるインタフェースから別のインタフェースへの仲介/変換を行うことを目的としたクラス。

たとえばボタンを押すとボタンは「`actionPerformed()`」というメソッドを呼び出してくれる (予め設定されているオブジェクトに対して)。しかし実際にボタンをきっかけに動作させたいオブジェクトがそのような都合のよい名前のメソッドを持っているとは限らない。そこで、これらの間を仲介するのがアダプタ。

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

public class Sample52 extends Frame {
  private int count = 0;
  private Label l0 = new Label("0");
  private Button b0 = new Button("+");
  private Button b1 = new Button("-");

  public Sample52() {
    setLayout(null);
    add(l0); l0.setBounds(40, 40, 80, 30);
    add(b0); b0.setBounds(40, 80, 60, 40);
    add(b1); b1.setBounds(110, 80, 60, 40);
    b0.addActionListener(new Sample52BtnAdapter(this, 1));
```

```
    b1.addActionListener(new Sample52BtnAdapter(this, -1));
  }
  public void addCount(int d) { count += d; l0.setText(""+count); }
  public static void main(String[] args) {
    Sample52 app = new Sample52();
    app.addWindowListener(new Sample52WinAdapter());
    app.setSize(400, 400); app.setVisible(true);
  }
}

class Sample52WinAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent evt) { System.exit(0); }
}

class Sample52BtnAdapter implements ActionListener {
  Sample52 app;
  int delta;

  public Sample52BtnAdapter(Sample52 a, int d) { app = a; delta = d; }
  public void actionPerformed(ActionEvent evt) { app.addCount(delta); }
```

しかしこの方法だと小さいクラスがごちゃごちゃになる。入れ子クラスと呼ばれる機能を使えばアダプタクラスが散らかるのはある程度避けられるが基本的には同じこと。

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

public class Sample53 extends Frame {
  private int count = 0;
  private Label l0 = new Label("0");
  private Button b0 = new Button("+");
  private Button b1 = new Button("-");

  public Sample53() {
    setLayout(null);
    add(l0); l0.setBounds(40, 40, 80, 30);
    add(b0); b0.setBounds(40, 80, 60, 40);
    add(b1); b1.setBounds(110, 80, 60, 40);
    b0.addActionListener(new Sample53BtnAdapter(this, 1));
    b1.addActionListener(new Sample53BtnAdapter(this, -1));
  }
  public void addCount(int d) { count += d; l0.setText(""+count); }
  public static void main(String[] args) {
    Sample53 app = new Sample53();
    app.addWindowListener(new Sample53WinAdapter());
    app.setSize(400, 400); app.setVisible(true);
  }
  static class Sample53WinAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent evt) { System.exit(0); }
  }
  static class Sample53BtnAdapter implements ActionListener {
    Sample53 app;
    int delta;

    public Sample53BtnAdapter(Sample53 a, int d) { app = a; delta = d; }
    public void actionPerformed(ActionEvent evt) { app.addCount(delta); }
  }
}
```

ここで、入れ子クラスの `static` を取ると、インスタンスに付属するクラス (内部クラス) になる。内部クラスからはインスタンス変数がアクセスできるので、初期設定とか外からいじるためのメソッドとかがなくせる。一方ではらわたに手を突っ込んでいるみたいで気持ちは悪い。

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

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

public class Sample54 extends Frame {
  private int count = 0;
  private Label l0 = new Label("0");
  private Button b0 = new Button("+");
  private Button b1 = new Button("-");

  public Sample54() {
    setLayout(null);
    add(l0); l0.setBounds(40, 40, 80, 30);
    add(b0); b0.setBounds(40, 80, 60, 40);
    add(b1); b1.setBounds(110, 80, 60, 40);
    b0.addActionListener(new Sample54BtnAdapter(1));
    b1.addActionListener(new Sample54BtnAdapter(-1));
  }
  public static void main(String[] args) {
    Sample54 app = new Sample54();
    app.addWindowListener(new Sample54WinAdapter());
    app.setSize(400, 400); app.setVisible(true);
  }
  static class Sample54WinAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent evt) { System.exit(0); }
  }
  class Sample54BtnAdapter implements ActionListener {
    int delta;
    public Sample54BtnAdapter(int d) { delta = d; }
    public void actionPerformed(ActionEvent evt) {
      count += delta; l0.setText(""+count);
    }
  }
}
```

さらに、

(1) 内部クラスを 1 箇所でしか使わない

(2) そのクラスが 1 つのクラスを extends しているか、または 1 つのインタフェースを implements しているだけ

という条件が満たされていると、

```
AAA... new InnerClass() ...BBB

class InnerClass extends AnSuperClass {
  // 中身
}
```

を次のようにまとめて書くことができる。これを無名内部クラスと言う。

```
AAA... new AnSuperClass() {
  // 中身
} ...BBB
```

無名内部クラスはかなり構文的には気持ち悪いが、プログラムがコンパクトになる。上の例を無名内部クラスにすると次のように。

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

public class Sample55 extends Frame {
  private int count = 0;
  private Label l0 = new Label("0");
  private Button b0 = new Button("+");
  private Button b1 = new Button("-");
```

```
  public Sample55() {
    setLayout(null);
    add(l0); l0.setBounds(40, 40, 80, 30);
    add(b0); b0.setBounds(40, 80, 60, 40);
    add(b1); b1.setBounds(110, 80, 60, 40);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        count += 1; l0.setText(""+count);
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        count -= 1; l0.setText(""+count);
      }
    });
  }
  public static void main(String[] args) {
    Sample55 app = new Sample55();
    app.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    app.setSize(400, 400); app.setVisible(true);
  }
}
```

しかしこの方法だと「+」ボタンと「-」ボタンはほぼ同じなのに 2 重に書いている。これをやめるにはどうしたらいいと思うか?

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

public class Sample56 extends Frame {
  private int count = 0;
  private Label l0 = new Label("0");
  private Button[] b = new Button[]{new Button("+"), new Button("-")};

  public Sample56() {
    setLayout(null);
    add(l0); l0.setBounds(40, 40, 80, 30);
    add(b[0]); b[0].setBounds(40, 80, 60, 40);
    add(b[1]); b[1].setBounds(110, 80, 60, 40);
    for(int i = 0; i < 2; ++i) {
      final int delta = 1 - (2*i);
      b[i].addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
          count += delta; l0.setText(""+count);
        }
      });
    }
  }
  public static void main(String[] args) {
    Sample56 app = new Sample56();
    app.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    app.setSize(400, 400); app.setVisible(true);
  }
}
```

このように 1 箇所に書いてそこをループで繰り返し通ればよい! なお、内部クラスの内側から参照できるのは

(1) インスタンス変数、インスタンスメソッド

(2) final 指定の局所変数およびパラメタ

なので、変数 `delta` は `final` 指定になっている。たとえば変数 `i` を直接書くことはできない。これは「はらわたが出すぎる」から。というか、`final` に指定してあるなら変数の値が変化しないので単に「通った時の値を覚えておけば」済むし安全。つまり初期設定の一種だと思えばよい。