オブジェクトシステム 2002 ＃ 4

久野　靖*

2003.2.12

1　はじめに

今回も出席メールをまとめます。なかなか面白いです。

打矢さん 教科書で、継承とコンポジションの使い分け、話としてはだいたい解りました。先生の補足説明でオブジェクトにおける再利用の効用とその留意が解り始めてきた様にも思います。実演して戴いたプログラムも動きと、おしゃりたい意味は面白くおもいました。しかしコーディンの意味や違いは追いきれていません。お話のなかからだけで、だいたいこんなことをおしゃりたいのだなくらいの理解でしかありません。皆さんのお話もそこにこだわるだけの意味の深さが理解できていない状態です。ただ皆さんが細かなことでも闊達に話すのはとてもよいことだと思いました。

久代さん 今回は、大きくインタフェースについて 3 つの点を学んだ。

その 1:
- インタフェースは、オブジェクト指向の多義性を実現するためのとっても大事な言語上の機構である。
- インタフェースにより、クラス間の結合を疎としつつ、メソッド多義性を活用することでクラス構造全体がわかりやすくなり、結果的にプログラミングの効率と品質が上がる (とうれしい)。

その 2:
- オブジェクト指向分析・設計のどのプロセスにインタフェースという概念が出現してくるのか?
- 出現は大きく 2 種類フェーズあると考えた。
- アーキテキチャ設計におけるドメイン分割時、このときに抽出される。
- プログラムをがぎがぎに書きながら、反省して自分で抽象化する。
- 本章に記載された戦術がわかりにくいのは、この 2 つの異なるフェーズの戦略が、ちゃんぽんに記載されていることにもよるのか。

その 3:
- オブジェクト指向初心者は、継承を覚えれば継承をインタフェースを覚えればインタフェースをと多様する傾向にある。
- このおさるさん現象の歯止めが必要となる (どうすればよいのかはまだ思いつかないが)。若気のいたりで、とことん継承、とことんインタフェースにしてしまう。
- うちの若い集も UML のイベントトレース図を書かせたら、まるで LSI パターン図のようなイベントトレース図を本日レビューに持ってきた。若いって恐ろしい。

吉武さん (メソッドのシグニチャを指定するということに限定しない) インタフェイス– 界面? – を指定することはシステム設計の上で重要であり、特にオブジェクト指向でものを考える場合には独特の色彩を帯びていることは論を待たないと思っています。したがって今回扱われた部分で強調されていることは至極もっともなことだとは思いますし、本文に挙げられている戦略に基づいて設計を行ってみたことがないのでよく分からない部分もありますが、これらの戦略も妥当なモノなのでしょう。敢えて言うならば、「Java による」と銘打っているのだから「このように実装できるのだから、このように設計しておくと

*筑波大学大学院経営システム科学専攻

自然な流れで云々」ともう少し Java の言語仕様と関連づけて説明されている方が飲みこみやすいように思います。授業中にも言いましたが、この点において久野先生が行っている動的分配などを強調したやり方の方が優れているように思いました。

話は少しそれますが、界面を設計することはソフトウェアシステムには必須なのか、という疑問はあります。「オブジェクト」を識別するためには必要でしょうから、オブジェクト指向風に設計する場合には界面を意識することは必要でしょうが、他のパラダイムに移れば不要だということはできないのか、といったこと。(もちろん「一枚岩強連結プログラム」に戻ろうと言っている訳ではないです) 「たゆたう水の如き」プログラミングをしたり、「和声」のようにシステムを把握したりできるとうれしかったりするような気もしないでもないです。「そういうのだと何ができるの？」と訊かれても何もアイデアはないのですし、「プログラム」ではなくて「アート」にしかならないかも知れませんが。

磯部さん
- インターフェイスの使い方の勘所がなんとなくしっくり来ました。何で継承があるのにインターフェイスなんかあるんだ?と悶々していましたが、「プラグ性」と言う語感でピンと来ました。でも、趣味で言えば「オブジェクト指向」の世界に実装的な妥協が紛れ込んでいるみたいで余り好きにはなれそうにありません。
- 吉武さんに「迷って良いんだよ」と言われたので安心しました。誰しも通る道なんだと分かれば気が楽になりました。(構造化の時のプログラム分割やサブルーチンの切り出しはスルスルと通れたのに。。。)

講義とは関係ないのですが、昨年 MS の TechED に行ったらオビジェクト指向でデリゲーション (?) と言う概念を紹介していました。どうも全然理解出来ないのですが、これは何でしょうか。

第 1 回目の参加者コメントで駒走さんが、クラスから Java の実装を知りたい旨のコメントをされていますが、これはそれほど難しくないことだと思います。私は自腹でオージス総研の「UML から Java への実装」コースを聞きに行きましたが、言ってた事は「クラス図が書けてたらあとはリファクタリングして行きましょう」だけでした。(これで 5 万円!)

中山さん インタフェースによる設計は，非常に有効な手法に感じられました．インタフェースを利用すべき場合についてのパターンも取り上げられているため，まずはこの指針に沿って設計を行えば，初心者でもなんとかなりそうな雰囲気があります．ただ，問題はどこをインタフェースの境界とし，どの程度の複雑さ・柔軟さを考慮するのかというところではないかと考えますが，これはやはり経験が必要というところでしょうか．

西森さん インターフェースを始めて知ったときは「継承とおんなじじゃん」と思ってました。そのころは継承と動的分配が頭の中では同一のものだったこともあります。

その後 C++ で「virtual hogehoge() = 0;」なる構文を良く使うようになり、コンストラクタも「protected:」に空実装で書くことが多くなって、プログラムを書いているうちにインターフェースって重要だなと思うようになってます。

本文中ではやたらたくさんインターフェースを使う場合が並んでましたが、それよりは授業中で「文脈をプログラムで表現するために…」という話の方が、納得がいきました (そんなこと言ったらプログラミングは全部そう、となってしまうかもしれませんが)。

でもそれなら Java のインターフェースには表明が欲しかったかなぁと思います。今の C みたいな「assert」はいらないので。あと例えばコマンドパターンで

```
interface Command {
  void exec();
  void undo();
}
```

と書いたりしたときに「undo() は exec() をやる前に呼んではいけない」というような制約が書ければ楽しいかな、と思います。シーケンス図で書くところをある程度言語で書ければいいのに。

```
void exec() : firstcall || after undo()
void undo() : after exec()
```

ダメかな。かっこ悪いですね。というか全然わかりやすくないような。

2 本の担当箇所紹介 (3 章残り、4 章)

3 並行プログラミングとスレッド

3.1 並行とその必要性

並行 (コンカレント) とは複数のことがらが「同時に」起きる (ないし起き得る) ことをいう。現実世界では多くのことが並行に起こっている。たとえば 2 人の人がすれちがうのも「並行して歩いている」から。

しかし、プログラミングの世界では「並行」は非常にマイナーな分野。多くのプログラミング言語は動作が「直列」(非並行) に起こるというモデル。

```
int z = a[i]; a[i] = a[i+1]; a[i+1] = z; // 交換
```

なんていうのも 3 つの代入が「順番に」起きるからこそ意味がある。並行に起こったら何がなんだかわからない。ただもちろん、並行に起こって欲しいこと (〜をしている間に〜も並行して行う) もある。ただ、そのようなことを直接扱う言語は今のところ主流ではない。

どれが正しいと思う?

a. プログラミングは「直列」モデルに従うのが分かりやすくてバグも出にくい。だからこれからも「直列」モデルを使うのがよい。

b. 我々は「直列」モデルのプログラミングに馴れているというだけ。「並行」モデルにまず馴れるようにすれば「並行」モデルでもいい。

c. プログラミング言語の枠組みとして「並行」をうまく手なずける何らかのブレークスルーが必要。それがうまく行けば「並行」が大変じゃなくなる。

伝統的に「並行」が必要なのは OS やリアルタイムなど「現実」と直接インタフェースするシステム部分だけ。これらの部分では並行を扱うため非常に大変な手間と注意を要していた。

一方、OS の上で動くアプリケーションは、OS が複数のタスク (プロセス) を (見かけ上) 並行に動かしてくれるので、その上で直列的にプログラミングすれば済んでいた。

なぜ並行を扱わないで済んで来たか? それは、CPU が非常に高速で、1 つずつ順番にやっても人間が「同時に沢山のことをさせている」と思ってくれるから並行モデルでプログラミングしなくても済んでいた。

どっちが正しいと思う?

a. それは今後とも同様で、プログラミングそのものは並行を取り入れなくても済むだろう。

b. 今後はプログラミングの上でも並行モデルを扱うようになる必要があると思う。

実際にはプログラミングに並行モデルを入れたいさまざまな要因が今日においては存在している。

1. マイクロチップのように実世界に接したシステムのプログラミングが増えている → 並行性の扱いが必要
2. ユーザの操作に密に追従するようなインタフェースが普及している → 追従しながら本来の動作もするといった並行性が必要
3. 普通のシステムでも CPU が沢山つくようになっている → 複数の CPU を活かすには並行性が必要
4. ネットワークの普及 → ネットの向こう側 (複数) との通信をしながら、同時にこちらでも何かをするというのは本質的に並行性。

3.2 さまざまな並行プログラミングモデル

並行プログラミングを行うに当たっては、さまざまなモデルが存在している。その中には言語という形でサポートできるものも、そうでないもの (言語は従来の言語) もある。

- 聖徳太子モデル — 1 つのプログラム実行でありとあらゆることに気を配りながら進んで行く。非常に大変。
- 割り込みモデル — 外部からの割り込みがあると特定番地へ飛び出し、そこで必要な処理を行ったら元へ復帰する。割り込み中にすることと通常処理ですることをうまく切り分け設計する必要がある。割り込み中に下手なことをすると通常コードでは「突然データが化ける」ように見えてパニックに。
- 関数型モデル、並列論理型モデル — プログラムの実行を関数の評価や論理式の推論としてモデル化 → どの部分から評価する、という順序は問題にならないのでプログラミング的には安全。しかし、すべてのコードを関数型/論理型で書くのは抵抗があるという説。
- プロセスモデル — 複数の直列プログラムがプロセスとして実行することで並行性を扱う。安全だが、プロセス間の通信の手間とオーバヘッドが大変。
- スレッドモデル — 1 つのプロセスの中で複数の「実行の糸」が並行に動くモデル。プロセスモデルよりオーバヘッドが小さいが並行制御の点でかなり注意が必要。
- 並行オブジェクトモデル — オブジェクト指向のオブジェクト 1 つひとつが並行実行単位となる。さらに、1 つのオブジェクト内には実行の流れが 1 つしかないモデルと複数あるモデルとがある。

3.3 スレッドモデル

結局、現在のお仕事プログラミング界ではスレッドによる並行記述が花盛りだが、これは次の要因による。

- POSIX などによるスレッド API の標準化のおかげで、ポータブルな記述ができる/一度学習しておけばさまざまなところで役立つ
- とりあえず言語は C/C++で済むので新しい言語を学ばないで済む
- プロセスモデルに比べて軽い → 性能が出やすい
- サーバなどの共有メモリマルチプロセサとスレッドのモデルがよく合っている → 性能が出やすい

ただし、スレッドモデルでは共有メモリを並行実行単位がアクセスするため、競合をきちんと解決しないとデッドロック等の問題が起きる。C や C++ではコンベンションに従ってライブラリ API を呼ぶ必要があるが、Java ではある程度これを言語に組み込んでくれているという利点。

3.4 Java のスレッド

Java ではスレッドはクラス `Thread` によって表される。ただし、実際にスレッドを作る場合は何らかの動作をさせたいわけで、その動作を書くのに次の 2 つの方法がある。

- `Thread` のサブクラスを作ってメソッド `run()` をオーバライドする。
- インタフェース `Runnable` というのがある。

```
public class Runnable {
  public void run();
}
```

このインタフェースを実装するクラスのインスタンスをコンストラクタで渡して `Thread` オブジェクトを作る。

どっちにせよ、とにかくスレッドが実行開始すると run() の内容を実行することになる。そして run() が終わるとスレッドの実行も終わる。

主要なメソッドとして次のものがある。

- start() — 実行を開始させる
- join() — 実行終了を待ち合わせる

× stop() — 実行を終わらせる

× susupend() — 実行を一時停止させる

× resume() — 一時停止しているスレッドを再開

×がついたものは deprecated、つまり現在では「使うべきでない」メソッド。これらを使うと色々危ない (なぜかわかりますか?)

ではスレッドを使う簡単なデモ。

```
public class Sample41 {
  public static void main(String args[]) {
    Runnable r1 = new MyRun("A", 10, 2000);
    Runnable r2 = new MyRun("B", 15, 1500);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start(); t2.start();
    try {
      t1.join(); t2.join();
    } catch(Exception e) { e.printStackTrace(); }
  }
  static class MyRun implements Runnable {
    String mesg = "???";
    int count = 10, time = 1000;
    public MyRun(String m, int c, int t) {
      mesg = m; count = c; time = t;
    }
    public void run() {
      try {
        for(int i = 0; i < count; ++i) {
          Thread.sleep(time);
          System.out.println(mesg);
        }
      } catch(Exception e) { e.printStackTrace(); }
    }
  }
}
```

演習 1 このプログラムは/u1a/kuno/work/Sample43.java に置かれている。持って来てそのまま動かせ。うごかし方:

1. cp /u1a/kuno/work/Sample43.java Sample43.java
2. javac Sample43.java
3. java Sample43

動いたらスレッドの数を増やしたり待ちのタイミングを変えてテストしてみよ。

3.5 スレッドと並列実行

さて、上の例は単に時間待ちをしていたが、もっと目一杯仕事をさせてみよう。

```
public class Sample42 {
  public static void main(String args[]) {
    final int n = 1;
    final int times = 16000000;
    Counter[] ct = new Counter[n];
    for(int i = 0; i < n; ++i) ct[i] = new Counter();
    Thread[] th = new Thread[n];
    for(int i = 0; i < n; ++i) th[i] = new Thread(new MyRun(ct[i], times));
    long t1 = System.currentTimeMillis();
    for(int i = 0; i < n; ++i) th[i].start();
    try {
      for(int i = 0; i < n; ++i) th[i].join();
    } catch(Exception e) { e.printStackTrace(); }
    long t2 = System.currentTimeMillis();
    System.out.println("time = " + (t2 - t1) + "ms");
    for(int i = 0; i < n; ++i) System.out.println(ct[i].getCount());
  }
  static class Counter {
    int count = 0;
    public void addCount(int d) { count += d; }
    public int getCount() { return count; }
  }
  static class MyRun implements Runnable {
    Counter count;
    int times;
    public MyRun(Counter c, int t) { count = c; times = t; }
    public void run() {
      for(int i = 0; i < times; ++i) count.addCount(1);
    }
  }
}
```

このプログラムは変数 n を変えることでスレッドの数を変更でき、また変数 times を変えることで各スレッドがいくつカウントするかを変えることができる。

演習 2 上のプログラム Sample42.java をそのまま動かせ。動いたら次の質問に回答すべく実験してみよ。

a. 1 つのスレッドで 16,000,000 数えるのと 16 スレッドでそれぞれ 1,000,000 数えるのとどっちが速いと思うか?

b. マシンを取り換えてみたらどうか? smp や smm ではどうか?

できれば速さの差などを見積もる方法を考案できるとすばらしい。

3.6 同期の必要性

ところで、上の例を改造してカウンタは 1 つにし、全スレッドがこれを一斉にカウントアップするようにした。

```
public class Sample43 {
  public static void main(String args[]) {
    final int n = 16;
    final int times = 1000000;
    Counter ct = new Counter();
    Thread[] th = new Thread[n];
    for(int i = 0; i < n; ++i) th[i] = new Thread(new MyRun(ct, times));
    long t1 = System.currentTimeMillis();
    for(int i = 0; i < n; ++i) th[i].start();
    try {
      for(int i = 0; i < n; ++i) th[i].join();
    } catch(Exception e) { e.printStackTrace(); }
    long t2 = System.currentTimeMillis();
    System.out.println("time = " + (t2 - t1) + "ms");
    System.out.println(ct.getCount());
  }
  static class Counter {
    int count = 0;
    public void addCount(int d) { count += d; }
    public int getCount() { return count; }
  }
  static class MyRun implements Runnable {
    Counter count;
    int times;
    public MyRun(Counter c, int t) { count = c; times = t; }
    public void run() {
      for(int i = 0; i < times; ++i) count.addCount(1);
    }
  }
}
```

演習 3 上のプログラム Sample43.java をそのまま動かせ。動いたら次の質問に回答すべく実験してみよ。

a. 答えが正しくならないのはなぜか。正しくするにはどうしたらいいか。

b. 同期を入れることによるオーバヘッドはどれくらいであると見積もれるか。CPU 数やスレッド数によってこれは変わるか。