「きれいなコード」とはいったい何でしょうか?

プログラムのコードは、第一義的にはコンパイラ（なり、インタープリタなり）が読み込んで解釈するためにあります。そこには「きれいなコード」「きたないコード」といった区別などありません。言語仕様に則ってさえいれば、どんな見た目のコードだろうが関係なしです。それでは、「きれいなコード」はなぜ必要なのでしょうか。

コードはコンパイラが読み込むためのものであるのと同時に、人が読むものでもあります。ここでいう「人」は、他人とは限りません。でき上がったプログラムを修正している1年後の自分かもしれません。さらに言えば、ついさっき書いたコードを見直している1時間後の、1分後の自分だったりもします。コードは人と人とのコミュニケーションの手段でもあるのです。すると、「他人が見てもわかりやすい」「後で見直しても理解しやすい」コードでなければなりません。「きれいなコード」とは、そのようなコードのことです。

ではいったい、どうやったらきれいなコードを書くことができるのでしょうか。一言で簡単に言えるような極意は、残念ながら存在しません。そこには経験が必要です。しかし、いまではさまざまな手法が考案され、それらを組み合わせ、使い分けることによって、昔に比べればきれいなコードを書くことは容易になりつつあります。

たとえば初期のプログラミング言語には、きれいなコードを書くための機能はあまり備わっていませんでした。FORTRANやBASICといった言語の初期の仕様には、ループや条件分岐をわかりやすく記述するための構文すらなく、俗に「スパゲティプログラム」と呼ばれる、スパゲティが絡まったような解読不能なコードが量産されました。

それに対して「構造化プログラミング」という概念が提唱され、いまどきの言語では当たり前に使われているわけです。さらに「オブジェクト指向プログラミング」という概念も生まれ、これもC++やJavaをはじめ、ごく普通に使われるようになり、きれいなコードが書きやすい環境になってきました。

とは言え、たとえば「Javaで書きさえすれば、自動的にきれいなコードになる」というわけではないのは、本書を手に取られたみなさんなら、ご承知のとおりだと思います。本書では、主にJavaを例に、Cをはじめ他の言語も取り上げつつ、きれいなコードを書くためにはどうしたらいいのか、さまざまな観点から見ていくことにします。

2012年12月　Lepton

CONTENTS

第1部 コーディングルールの光と影 010

鉄則その1 メソッド（関数）は○○行以下に!? ……012
行数を制限するコーディングルール ……012
メソッド分割 ……014
行数制限は万能？ ……019
まとめ ……022

鉄則その2 命名規則の考え方 ……023
かつての命名規則 ……023
プログラミング言語による制限 ……027
何の名前か ……028
名前のスコープ ……030
言語の問題 ……032
名前の省略 ……033
検索性 ……034
まとめ ……035

鉄則その3 わかりやすいコードを書く ……036
条件式（等号） ……036
条件式（不等号） ……039
コメント ……041
（カッコを付ける） ……042

{カッコを付ける} ……046
breakとcontinue ……049
まとめ ……051

鉄則その4 見た目を美しく ……052
インデントと括弧の位置 ……052
1行の文字数 ……055
空白の入れ方 ……057
まとめ ……059

第2部 コーディングルールに書けないあんなこと・こんなこと 060

鉄則その5 再帰呼び出しの使いどころ ……062
ファイルの一覧を表示する ……062
階乗を計算する ……068
ハノイの塔と木構造 ……069
まとめ ……071

鉄則その6 探索アルゴリズムの選び方 ……072
逐次探索 ……072
二分探索 ……075
ハッシュテーブル ……081

データベース ……084
まとめ ……085

鉄則その7 整数型の使い方 ……086
どの型を使うべきか ……086
整数型いろいろ ……088
まとめ ……090

鉄則その8 フラグとビットフィールドは必要最小限に ……091
フラグ ……091
論理型 ……094
ビットフィールド ……095
まとめ ……097

鉄則その9 文字と文字列の落とし穴 ……098
文字集合と符号化方式 ……098
UTF-16とサロゲートペア ……099
内部コードと外部コード ……104
まとめ ……109

鉄則その10 日付と時刻の持たせ方 ……110
蔵書リスト作成プログラムと蔵書検索プログラム ……110
Java APIにおける日付と時刻 ……116

タイムゾーンに気をつけよう …… 117
場合に応じた日付の持たせ方 …… 119
まとめ …… 119

第3部 上手に楽をするために 120

鉄則その11 言語を選ぶ …… 122
どんな言語を使ったらいいか …… 122
とりあえずCで書いてみる …… 124
やはり文字列はJava のほうが楽 …… 129
Perlにすればもっと簡単? …… 135
awk（おーく）は言わねえ…… …… 138
コードを書かずに済ます …… 140
まとめ …… 141

鉄則その12 デザインパターンをどう使うか …… 142
デザインパターンとは …… 142
デザインパターンの例：Iteratorパターン …… 143
デザインパターンの例：Adapterパターン …… 151
デザインパターンをどこまで意識すべきか? …… 157
まとめ …… 159

第4部 プログラムはでき上がっても、それで終わりではない 160

鉄則その13 メソッドやクラスの変更と廃止 …… 162

名前が公開される範囲とその影響 …… 162
メソッドを廃止すると…… …… 163
deprecated（非推奨） …… 166
クラス名やメソッド名の変更 …… 171
まとめ …… 172

鉄則その14 小規模なシステムと大規模なシステムの違いを考える …… 173

ウォーターフォールモデル …… 173
小規模から大規模へ …… 177
仕様が変更になったら…… …… 180
まとめ …… 183

鉄則その15 リファクタリングを実行する …… 185

仕様と実装の狭間で…… …… 185
リファクタリングとは？ …… 188
オブジェクトによるデータ値の置き換え …… 191
「委譲による継承の置き換え」と「継承による委譲の置き換え」 …… 197
リファクタリングとテスト …… 201
設計が先か、とりあえず実装か …… 202
まとめ …… 203

第5部 きれいなコードを書くための最後の鉄則 204

鉄則その16 きれいなコードを書くための最後の鉄則……206

コミュニケーションも大切 ……206
コメント−プログラマ同士のコミュニケーション ……215
コードを読む ……217
最後に……220
まとめ……220

索引 ……221

コラム索引

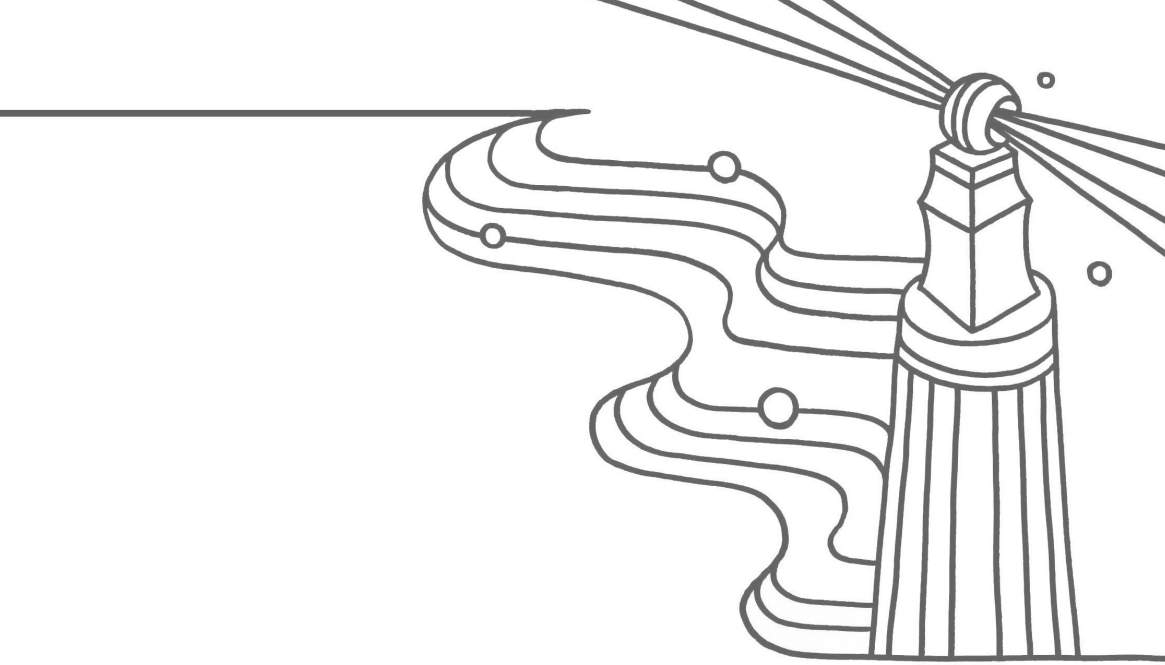
Javaからデータベースへの接続……085
ウォーターフォールモデル以外の開発モデル ……175
Java のアクセサ ……183
ポリモーフィズムによる条件記述の置き換え ……211

第1部

コーディングルールの光と影

プログラムは、文法に従って書きさえすれば、どんな書き方をしてもいいというわけではありません。趣味で個人的に書いて、ソースをどこにも公開する予定のないプログラムでしたら、どんな書き方をしようとそれは自由です。もちろん「自由」とは言っても、あとで自分で見直して、修正したいときに「自分でも何を書いているのか、何をしているのかさっぱりわからない」なんていうプログラムは自分で自分の首を絞めることになるでしょう。いっぽう、仕事でプログラムを書くとなると「どんな書き方をしようとそれは自由」とはいきません。プログラミングを行ううえでの何らかの指針が必要となります。

プログラムを書く（コーディング）うえでの「こんな書き方をしましょう」という取り決めを「コーディングルール」(「コーディング規約」とも)と呼びます。ある程度以上の大きさのチームでプログラミングするときには、たいていこのコーディングルールがあることと思われます。コーディングルールは、そのチーム（システム、職場、会社などなど）が定めた独自のものもあるでしょうし、一般的によく知られているルールを流用、準用することもあるでしょう。



いずれにしても、なんらかの取り決めがあるはずです。

いままで趣味で個人的にプログラミングをしていた人から見れば、「なんか窮屈だなあ」「面倒くさいなあ」と思われるかもしれません。でも、そういう人たちであっても、きれいなプログラムを書いてきた人はとくに、（明文化はしていないかもしれませんが）自分なりのルールを自分で決めて、それに従ってプログラミングしてきたと思われます。しかしチームとして、仕事としてプログラムを作ることになれば、明文化する必要がでてくるのです。変数名やメソッド名の命名規則、字下げや空白の入れ方といった書式、コメントの書き方、などなど。

コーディングルールを定め、それを守ることは「きれいなプログラム」を書くための必要条件と言えるかもしれません。ただし、それは「必要条件」であって、決して「十分条件」ではありません。つまり「コーディングルールを守ってさえいれば、自然ときれいなプログラムができあがる」ということではない、ということです。ここでは、コーディングルールの重要性とその限界について、いくつかのテーマで見ていくことにします。

鉄則 その1

メソッド（関数）は○○行以下に!?

行数を制限するコーディングルール

世の中に存在するコーディングルールのほぼすべてには、次のようなことが書かれています。

- ☛ 1メソッドの行数は○○行以下を目安として、多くても△△行程度にとどめること。
- ☛ 1メソッドの行数は原則として○○行以下とすること。
- ☛ 1メソッドの行数は○○行以下でなければならない。
- ☛ 1メソッドの行数を○○行以上とすることは禁止。

下になるほど強い表現になっていますが、いずれも、1つのメソッド（言語によっては「関数」「プロシージャ」等さまざまな呼び方をされますが、ここでは「メソッド」と呼んでおきます）の行数を制限するルールであることには変わりありません。このルールの言いたいことは、要するに、

長いメソッドは悪

ということです。世間のたいていのコーディングルールにこれが存在しているということは、つまりこの「長いメソッドは悪」というのは世の中の共通認識でもある、ということを暗示しています。

メソッドの行数が多くなることがなぜいけないのか、ということを考える前に、まずは現状調査をしてみることにします。ここでは、Javaのjava.lang、java.io、java.utilという3つのパッケージ[†1]のクラスで宣言されているメソッドについて、それぞれのメソッドの

†1　クラスの集まりのこと。java.lang、java.io、java.utilはいずれも、あらかじめ提供されているクラス（Java API）のパッケージ。

行数を調べてみました。

- Javaのバージョン：1.6.0_31
- メソッド数：約4000
- 行数の条件：コメントのみの行、空行は除く

行数	割合
10行以下	74.8%
20行以下	90.0%
30行以下	94.5%
40行以下	96.3%
50行以下	98.0%
100行以下	99.6%
200行以下	99.9%

表1-1 Javaのパッケージ（Javaのjava.lang、java.io、java.util）のメソッド行数

この結果を見ると、

全体の9割が20行以下

であることがわかります。「たった20行以下でいったいどんな処理を書けるんだろうか?」と疑問に思われた方もいらっしゃるかもしれません。

Javaのようなオブジェクト指向言語の場合、クラスのフィールド（またはインスタンス変数などとも呼ばれる）は外部から隠蔽され、アクセサメソッドを経由して参照されることが多いのはご存知のとおりです。そしてフィールドの内容を返すメソッド(ゲッター)や、フィールドに値を代入するメソッド（セッター）といった単純なアクセサメソッド（単純なので行数が少ない）が大量に宣言されることになります。[†2]したがって、行数の少ないメソッドが多くなる傾向にあります。

それなりに複雑な処理を行おうとした場合、20行程度では難しいことが多いのではないかと思いますが、たとえばJavaのコーディングルールでは20行というのを1つの目安にしているものが多いようです。

†2　オブジェクト指向言語であっても、「プロパティ」といった機能によりアクセサメソッドを記述する必要がない言語もある。

なお余談ですが、今回調べた3つのパッケージで宣言されているメソッドのうち、もっとも長いメソッドはjava.util.JapaneseImperialCalendarというクラスの

```
public void roll(int field, int amount)
```

というメソッドで、337行（コメント行と空行は除く）でした。“JapaneseImperialCalendar”というのは和暦のことのようで、英訳するとこういう表現になるんですね。“Imperial”というあたりが時代がかっているような気もしないでもありません。

このrollメソッドは、年月日時分秒などのうちどれか1つ（引数fieldで指定）に、amountだけ加算するメソッドです。日付の加算は、大の月、小の月だの、うるう年だの、それに和暦ともなれば年号だので、例外のかたまりのような処理になることは容易に予測できます。したがって、メソッド自体が肥大化してしまっているのでしょう。

さて、「長いメソッドは悪」ということですが、いったいなぜそうなのでしょうか？　一般的に言われている理由としては、次のようなものがあります。

- ☛人がぱっと見て、そのメソッドが何をやっているのかを即座に理解できるのは○○行程度まで（であろう）。
- ☛メソッド内で同じようなことを何度も繰り返しやっている（可能性が高い）。
- ☛1つのメソッドで複数の仕事を行っている（ことが多い）。
- ☛（ローカルな）変数を使いまわしている（かもしれない）。
- ☛ループや条件分岐のネストが深くなりすぎてしまっている（はず）。

メソッド分割

実例を見ていくことにします。リスト1-1を見てください。[†3]これは、オリンピックの開催年を表示するプログラムです。実行結果はリスト1-2のようになります。

mainメソッドだけですべての処理を行っているプログラムで、そのmainメソッドの行

†3　これは、拙著「世界一わかりやすいJavaプログラミングの授業」（ソシム刊）に載っているJavaプログラム。その本では、if文を使った多重分岐を説明するためにこのプログラムを持ち出した。そのため、あえてこのような（多少わかりづらい）書き方をしてある（と、一応言い訳しておく）。

数は24行あります。20行を1つの目安とするならば、微妙な線と言えるでしょうか。処理の内容ですが、「ぱっと見て、そのメソッドが何をやっているのかを即座に理解できる」でしょうか? ちょっとわかりづらいですね。for文でyearを1890から2020まで繰り返しているのはすぐにわかるかもしれませんが、そのfor文の中にif文がたくさんあって、いったいどこで何をやっているのか、すぐには理解できないかもしれません。

リスト1-1　オリンピックの開催年を表示するプログラム その1

```
public class P1_01 {

    //
    // 1890年から2020年まで、夏季・冬季オリンピックが開催される年かどうかを表示
    //
    public static void main(String[] args) {
        for (int year = 1890; year <= 2020; year++) {       // 1890年から2020
                                                            // 年まで繰り返す
            int mod = year % 4;
            System.out.print(year + " : ");
            if (year <= 1895) {                             // 1895年まで
                ;
            } else if (year <= 1923) {                      // 1923年まで
                if (mod == 0) {
                    System.out.print("summer");
                }
            } else if (year <= 1993) {                      // 1993年まで
                if (mod == 0) {
                    System.out.print("summer winter");
                }
            } else {                                        // 1994年以降
                if (mod == 0) {
                    System.out.print("summer");
                } else if (mod == 2) {
                    System.out.print("winter");
                }
            }
            System.out.println();
        }
    }
}
```

リスト1-2　リスト1-1の実行結果

```
1890 :
1891 :
1892 :
1893 :
1894 :
1895 :
1896 : summer
1897 :
1898 :
1899 :
1900 : summer

  ⋮

1923 :
1924 : summer winter
1925 :
1926 :
1927 :
1928 : summer winter
1929 :
1930 :

  ⋮

1988 : summer winter
1989 :
1990 :
1991 :
1992 : summer winter
1993 :
1994 : winter
1995 :
1996 : summer

  ⋮

2020 : summer
```

ところでオリンピックの開催年の条件はご存知でしょうか? 表1-2を見てください。今は、夏季オリンピックは西暦が4で割り切れる年に、冬季オリンピックは西暦を4で割った余りが2の年に開催されていますが、こうなるまでには紆余曲折がありました。

西暦	～1895年	1896年～1923年	1924年～1993年	1994年～
夏季	なし	西暦が4で割り切れる	西暦が4で割り切れる	西暦が4で割り切れる
冬季	なし	なし	西暦が4で割り切れる	西暦が4で割った余りが2

表1-2 オリンピックの開催年

オリンピックの開催年であるかどうかを調べるには、まず表1-2にあるとおり、4つの期間に分けて考える必要があります。そして、それぞれの期間について、4で割ったときの余りによって、夏季、冬季それぞれのオリンピックが開催されるかどうかが決まります。「4つの期間に分ける」「4で割った余りで判断する」という2つのことを実行しているのがリスト1-1のfor文の中身だったのです。

この「わかりづらさ」を解消するためにはどうしたらいいでしょうか? リスト1-3を見てください。ここでは、isSummerOlympicGamesHeldとisWinterOlympicGamesHeldという2つのメソッド（いずれもprivateなクラスメソッド）を作っています。メソッド名からもわかるように、引数として「年」を渡すことにより、それぞれ夏季オリンピック、冬季オリンピックが開催される年かどうかを返すメソッドです。

これらのメソッドを使ってmainメソッドのほうも修正しました。今度のmainメソッドは、かなりわかりやすくなったのではないでしょうか。

- 1890年から2020年まで繰り返す。
- 夏季オリンピックの開催年だったら、“summer”を出力する。
- 冬季オリンピックの開催年だったら、“winter”を出力する。

と同時に、各メソッドの行数も次のようになりました。

- mainメソッド：12行
- isSummerOlympicGamesHeldメソッド：6行
- isWinterOlympicGamesHeldメソッド：12行

リスト1-3　オリンピックの開催年を表示するプログラム その2

```
public class P1_03 {
    //
    // 1890年から2020年まで、夏季・冬季オリンピックが開催される年かどうかを表示
    //
    public static void main(String[] args) {
        for (int year = 1890; year <= 2020; year++) {  // 1890年～2020年
            System.out.print(year + " : ");
            if (isSummerOlympicGamesHeld(year)) {      // 夏季オリンピック
                System.out.print("summer ");
            }
            if (isWinterOlympicGamesHeld(year)) {      // 冬季オリンピック
                System.out.print("winter");
            }
            System.out.println();
        }
    }

    // 夏季オリンピックが開催される年かどうかを返す
    private static boolean isSummerOlympicGamesHeld(int year) {
        if (year >= 1896 && year % 4 == 0) {
            return true;
        }
        return false;
    }

    // 冬季オリンピックが開催される年かどうかを返す
    private static boolean isWinterOlympicGamesHeld(int year) {
        if (year >= 1924 && year <= 1993) {
            if (year % 4 == 0) {
                return true;
            }
        } else if (year >= 1994) {
            if (year % 4 == 2) {
                return true;
            }
        }
        return false;
    }
}
```

このように、できあがったプログラムであるリスト1-1を、後でリスト1-3に書き換えるような作業を

リファクタリング

と呼びます（メソッドの抽出）。リファクタリングについては後でまた取り上げますが、後から「正常に動作しているプログラムを書き換える」作業を行うのは大変ですし、正常に動かなくなる危険も伴います。できれば、はじめからリスト1-3のようなコードを書くようにするべきです。

とは言うものの、それは

言うは易く行うは難し

です。私も、最初はリスト1-1のようなコードを書いてしまうでしょう。でも、そこからが重要です。自分の書いたコードを見直してみて、それをたとえば他人に説明しようとしたときに簡単に説明できるだろうか、という目で見ていくと、問題点が明らかになります。

しかし、それでも問題点が発見されない場合、コーディングルールの「メソッドの行数は○○行以下」というのが1つのきっかけになるかもしれません。長すぎるコードには、何らかの問題点が潜んでいる可能性が高いと言えます。

行数制限は万能？

もちろん、メソッドの行数制限のコーディングルールは万能ではありません。「1メソッドは20行以下」というルールがあったとき、リスト1-1のmainメソッドはそれに違反しています。そして、mainメソッドの内容を吟味した結果、リスト1-3のよう書き換え、すべてのメソッドの行数が20行以下になりました。これはこれでうまく行きました。

めでたし、めでたし。

しかし、世の中、そう簡単には行かない場合も少なくありません。一例を挙げます。入力された情報（ここでは、顧客の個人情報とでもしておきます）を、それぞれの項目、たとえば、姓、名、郵便番号、住所（都道府県）、住所（市区町村）、住所（その他）、電話番号（固定電話）、電話番号（携帯）、等々について、正しい値かどうかをチェックし、必要があれば整形するようなメソッドを考えます。各項目ごとにチェックするメソッドは別に

作るとしても、1項目あたり、最低でも次のような3行は必要でしょう。[†4]

```
if (!checkFirstName(input_data.first_name)) { // input_dataは入力値、
                                              // checkFirstNameは「名」
                                              // をチェックするメソッド
    throw IllegalArgumentException;
}
```

仮に入力項目が10個あれば、それだけで30行必要です。20項目なら60行です。かといって、「1メソッド20行以内」というルールにとらわれて、この処理を複数のメソッドに分割したとしても、それはたぶん無意味です。ひとかたまりのデータの処理は1つのメソッドで行うべきでしょう。

入力項目のチェックというこの例のような

泥臭い処理

は、例外処理のかたまりのようになりますから、どうしても長くなってしまいがちです。それを無理やり複数のメソッドに分割しても、同じデータの一連の処理が複数のメソッドに分散してしまっただけで、わかりづらくなったに過ぎません。それでは意味がないわけです。

「無理やり複数のメソッドに分割」と言えば、その昔、（Cのコードだったので「メソッド」ではなく「関数」でしたが）こんなコードを見たことがありました（リスト1-4）。

リスト1-4　関数を無理やり分割？

```
void hoge(int param1, int param2, int param3)
{
    int m, n;

      ⋮
      ⋮ （数10行くらい）                                  (a)
      ⋮
```

† 4　このコードは、入力された値を保持するオブジェクトを引数（input_data）として受け取って、checkFirstNameなどのメソッド（別に作っておく）でチェックを行う処理を想定している。ここでは、受け取った値に問題がある場合には、IllegalArgumentExceptionを投げているが、エラーコードを返すようにしたり、その他のエラー処理も行うようにしたりすれば、3行では済まずさらに行数が増えることになる。

```
    hogeX(&param1, &param2, &param3, &m, &n);
}

/* 長くなりすぎたので関数hogeを分割 */
void hogeX(int *param1, int *param2, int *param3, int *m, int*n)
{

      ⋮
      ⋮ （数10行くらい）                                    (b)
      ⋮

    hogeXX(param1, param2, param3, m, n);
}

/* 長くなりすぎたので関数hogeXも分割 */
void hogeXX(int *param1, int *param2, int *param3, int *m, int*n)
{

      ⋮
      ⋮ （数10行くらい）                                    (c)
      ⋮

    hogeXXX(param1, param2, param3, m, n);
}

/* 長くなりすぎたので関数hogeXXもやっぱり分割 */
void hogeXXX(int *param1, int *param2, int *param3, int *m, int*n)
{

      ⋮
      ⋮ （数10行くらい）                                    (d)
      ⋮

}
```

これは何をやっているかというと、関数hogeの最後で関数hogeXを呼び出して、その関数hogeXの最後で関数hogeXXを呼び出して、さらにその関数hogeXXの最後で関数hogeXXXを呼び出しています。つまり（a）→（b）→（c）→（d）の順番に文が実行されることになります。しかも関数hogeの引数param1、param2、param3とローカルな変

数m、nを、その下のすべての関数に持ちまわっています。[†5]

これってつまり、(A) ～ (D) のすべての文を関数hogeに書いたのと同じことです。「どうしてこんなややこしいことをするんだろう?」とそのときには思ったのですが、つまりこれはコーディングルールでメソッド（この場合は関数ですが）の行数が制限されていて、無理やり複数の関数に分割した、ということですね。

あたりまえのことですが、こんなことをやっても何の意味もありません。逆にわかりづらくなっただけです。それでも、

> ☛ 1メソッドの行数を○○行以上とすることは禁止。

というような強い表現でのルールがあると、どうしてもそれに従わざるを得ない、というか、従っていないと通らないことによる苦肉の策とも言えます。

「長いメソッドは悪」というのは、大筋においては正しいとは思います。長すぎるメソッドはどこかに問題があることが多いでしょう。コーディングルールで決められた行数より長いメソッドができあがってしまったら、「何かおかしいところはないだろうか」という目でコードを見直すのはいいことです。

しかし、処理の内容によっては、どうしても長くならざるを得ないこともありますし、まして、ルールを守るということだけのためにリスト1-4のようなことをしてしまっては本末転倒といえます。

I. 長すぎるメソッドができあがってしまったら、どこかおかしなところはないか、わかりづらいところはないか、という目でそのメソッドを見直してみよう。

II. 処理の内容によっては、どうしても短くできないメソッドというものも存在するので、メソッドの行数制限を金科玉条にしていいとは限らない。

III. コーディングルールを守るためだけの理由で、無理やりメソッドを分割するのは無意味。

†5　関数hogeからparam1 ～ nの各変数のポインタを関数hogeXに渡している。Cのポインタは Javaの「参照」のようなもので、こうすることによって関数hogeXから関数hogeの変数（引数を含む）であるparam1 ～ nのそれぞれの変数の内容を（*param1のように先頭に*を付ける必要はあるが）参照することも、書き換えることもできる。