ビューティフルコードのための N 個の指針[1]

久野　靖†
筑波大学ビジネスサイエンス系
〒112-0012 東京都文京区大塚 3-29-1
kuno@gssm.otsuka.tsukuba.ac.jp

概要

プログラムのコードは美しくなければいけない、とは昔から言われていることである。かっては「よいプログラムを書くための指針」のような本がいくつもあり、そのようなテーマの議論も普通になされていたが、今日ではそれがあまり見られなくなっているように感じられる。なぜそうなのか、「指針」論は不要なのかということと併せて、本稿では今日における「指針」について議論の題材を提供することを試みる。

1　はじめに

本稿は 2012 年夏のプログラミングシンポジウム「ビューティフルコード　programming should be fun; programs should be beautiful」の発表原稿として作成している。この発表をすることになった原因は、2008 年に筆者がその名も「ビューティフルコード」という題名の書籍 [6] の翻訳に携わったことにあるらしい。それがもとで、筆者は「美しいコード」について一家言ある人だと思われたらしく、何回かそのようなテーマで話をするようにと依頼された。自分は翻訳しただけでしてそんな偉そうなことはどうも、と大体断って来たのだが、今回は上記のテーマにふさわしいプログラム構成を幹事団として設定する中でそういう話を担当せよ、ということになったため、ご期待に添えるか分かりませんと言いつつやらさせて頂くことになった。

ちなみに、和田委員長からのリクエストは「久野さんがプログラム, 特に公開するプログラムを書くとき, どこに注意するか; その理由は何かを 10 件くらいまとめて下さい.」ということだったのだが、それだけだと本当に放言だけになってしまうので、以前にプ会 [10] で話した「過去に遭遇したプログラミングの美しさの本」の話なども含めて取り上げさせて頂く。

[1] N Guidelines Toward "Beautiful Code" by †Yasushi KUNO, Faculty of Business Sciences, University of Tsukuba, Tokyo.

2　過去の「美しさ」本

2.1　プログラム書法

「美しさ本」の筆頭に挙げるべきなのはやはり、Kernighan らの「プログラム書法」[7] かと思う (以下「書法」と記す)。この本は原題が The Elements of Programming Style であり、英文のスタイルに関する超有名な本 The Elements of Style[16] をもじっているという点も大胆である。[2]

内容であるが、テーマがそのものズバリなだけに、まさに「よいプログラムを書くための知恵」がぎっしり詰まっている。言語が旧 FORTRAN と PL/I なので古いが、今でも十分読める。だから現在でも入手可能である。スタイルは、さまざまなテーマ (表現、制御構造、プログラム構造、…) ごとに「規則」を挙げ、コード例とともにその規則の背景や理由を解説する、というものである。巻末には参照しやすいように規則一覧が載っている。ここから筆者が独断で選んだ代表的なものを挙げておこう。

- わかりやすく書こう — うますぎるプログラムはいけない。
- プログラムを上から下へ読めるようにしよう。

[2] 個人事ながら、翻訳者が筆者の恩師である木村　泉先生であり、まだ学部 2〜3 年の頃、出版された訳本 (当時のは第 1 版) を繰り返し読んだ。研究室ではこの本やその後同じ著・訳者により刊行された「ソフトウェア作法」[8] についてゼミで議論されていたものと思うが、自分は研究室所属前であり残念ながら参加の機会を得なかった。

- 同じ表現の繰り返しは共通関数への呼び出しに変えよう。
- だめなプログラムを修正するのはやめて、全部書き直そう。
- まずわかりやすい擬似言語で書いて、それから目的の言語に翻訳しよう。
- 速くする前に、まず正しくしよう。

とくに「わかりやすく書こう」は冒頭にある規則で、これを読んだ時以来、自分の座右の銘になっている (これについてはまた後でも触れる)。

2.2 芸術としてのプログラミング

Knuth のチューリング賞記念講演「芸術としてのプログラミング」[11] の中に、プログラミングの趣味とスタイルに関する内容がある。「書法」にも言及していて、それを踏まえた Knuth の考え方が書かれている。

- ひとつの「最良の」スタイルはない。だれもがその人の好みを持っている。それが自身を表現しようとするのに最もよい方法である。
- 他人の好みを変えさせようとすべきでない。
- 美しいものが他人にも有用ならなおよい。
- 正しく動作し、変更しやすいとよい。そのためには読みやすく理解しやすく。
- 効率をよくない趣味だと非難すべきでない。効率について気にすべき時と場所がある。

とくに最初の点は Dijkstra の言「生徒たちが自分自身のスタイルを発見できるような教育をする」も引いて強調している。確かに好みについての指摘は重要だと筆者も思う。なお、この箇所の次で Knuth は「道具だてが少ないほど楽しみは多い」と題して厳しい制約下のコーディングの楽しさを述べ、「芸術のための芸術」から逃げるべきでないと述べている。

2.3 プログラミングの心理学

「プログラミングの心理学」[15] は Weinberg の古典的名著である[3]。この本はプログラミングに限らずコンピュータやソフトウェア開発全般にわたって人間の心理的側面との関わりを詳細に検討し問題提起している。その中で第 2 章では「よいプログラムとは」と題して、よいプログラムの要件を 4 つ挙げている。

- 仕様を満たしていること (または、どの程度満たしているか)。
- スケジュールどおり作られたか。
- 仕様が変わった時に変更できるか (どのくらいのコストか)。
- どの程度効率がよいか (その定義は。また他のものを犠牲にしていないか)。

これを見ると全般的にソフトウェア工学よりであり、実務の人だった Weinberg らしいと言える。一方で、「プログラミング言語」の章には次のような記述もある。

> 自分のプログラムを審美的対象として眺めてみる、ということをたまにはしてみないプログラマは、真のプログラマとはいえないだろう。「ここはどうも対称性がないなあ」とか、「あそこは汚らしくて、さらっと流れてないなあ」とか、「全体がページの中に、すんなり納まっていないなあ」とかいった具合に、である。

つまりこの本でも、整っていて読みやすい、というこれまでに出て来たよいプログラムの基準と同様のものに価値を置いているといえる。

2.4 プログラミング 250 の心得

筆者の書棚に学生の頃から入っている「プログラミング 250 の心得」[12] という本がある。これも規則とその説明の山なのだが、「書法」より実務指向なのが面白い。スタイルに関する規則を抜粋すると、次のようなことが書かれている。

- ユーザのニーズにプログラムを適合させよ。
- ユーザの作成した入力データを表示せよ。
- ループを重視して、これを単位として取り扱え。
- 同じ重みを持つ命令の頭を揃えよ (字下げをせよ)。
- ジョブ向きの正しい言語を使え。
- 実行順に読んで行けるように書け。
- ループの入れ子は控え目にせよ。
- ライブラリ関数を使え。

これを見るとスタイル本は「書法」だけでは無かったのだなと改めて確認させられる。

[3] 木村先生の愛読書でもあった。木村先生は Weinberg の本を多く訳されているが、この本の翻訳時はお忙しかったことから、研究室先輩の角田先生、白濱さんとともに筆者も訳出をお手伝いした。なお、最近、別の訳者による新訳が出版されている。

2.5 ソフトウェア作法

「ソフトウェア作法」[8] も「書法」と同じコンビによるものである。この本は Ratfor という特殊な言語で例題が書かれているのに、とても人気があり、32 年前の本なのに (!!) 今でも大きな書店に行けば並んでいる。[4]

内容は「書法」より上のレベルであり、さまざまなツール (圧縮、文字列置換、整列、パターンマッチ、マクロ展開、文書整形、プリプロセサ) について実際にコードを書いててみせている。論点はツールの仕様や設計が中心だが、スタイルに関する議論もそれなりに載っている。いくつか例を挙げる。

- まとまった概念は下請けの関数にするべき。
- if-else の連鎖は読みやすいので積極的に使いたい。
- 複雑な条件を書くより連鎖の枝分かれが多い方が明瞭。
- 予防的プログラミングをしよう。if-else の最後の else の活用。
- 関数の値を使わないときには「`junk = 関数(...)`」のように明示する。
- よい界面の重要性 (例: 入力の読み戻し)。

あと、作り方の話であるが、プログラムは段階的に作る (最小限動く状態にして、そこから機能を増やして行く)、などにも言及されている。

2.6 プログラミング作法

「ソフトウェア作法 (原著名: Software Tools)」が Ratfor で書かれていたので、その後で米国では言語を Pascal に直した Software Tools in Pascal が刊行されたが、邦訳はない。その後 Pascal も古くなったので、2000 年になって Kernighan らは「プログラミング作法」[9] を刊行した。これは言語が C と C++と Java だから現役の言語である。内容は「書法」以来の集大成で、第 1 章「スタイル」からはじまり「アルゴリズムとデータ構造」「設計と実装」「インタフェース」「デバッグ」「テスト」「性能」「移植性」「記法」とうまくテーマが選ばれている。規則もテーマ別になっていて、「書法」と同様、最後に規則集がある。代表的なものを示しておく。

- 「グローバルにはわかりやすい名前を、ローカルには短い名前を」(書法)
- 「多分岐の判定には else-if を使え」(書法)
- 「実装の詳細を隠蔽しよう」「ユーザに内緒で何かをするな」(インタフェース)
- 「エラーの検出は低いレベルで、その処理は高いレベルで」(インタフェース)
- 「打つ前に読め」「自分のコードを他人に説明してみよう」(デバッグ)
- 「回帰テストを自動化しよう」「テストの網羅範囲を測定しよう」(テスト)
- 「より優れたアルゴリズムやデータ構造を利用しよう」「関係ない部分を最適化するな」(性能)
- 「システム依存のコードは別個のファイルに」「システム依存部分はインタフェースの裏に隠蔽しよう」(移植性)

3 「美しさ」本の現状

3.1 今日、「美しさ」本は…

ここまで過去の「美しさ」本 (と筆者が思うもの) から抜粋して紹介してきたが、今日ではメジャーな美しさ本は「出ていない」と述べてよいと思う (「プログラム書法」「ソフトウェア作法」が今でも売れているのはそれが理由ではないかと想像される)。その理由を考えてみたが、次のようなことがあるのではないかと思う。

(a) 使用される言語が多様化した。
(b) ソフトウェア開発においてスタイル以外の部分の重みが増した。
(c) コードの美しさに対する関心の低下 (?)。
(d) 言うべきことの枯渇 (?)。

まず (a) について言えば、「書法」の頃よりずっと多くの言語が広く使われていることは確かである。そして「美しさ」に関わる具体的なところは、言語によってかなり違って来る。このため、「美しさ」本を書くとしても「C、C++、Java」のように代表的なものをまず定め、それでお話しま

[4] また私事で申し訳ないが、この本は「日本で初めてコンピュータ原稿を使って制作された本」であり、それが「木村研の日本語処理システムの成果」だった。そのため、制作のお手伝いを久野ほかがずっとやっていた (磁気テープ抱えて漢字プリンタ出力に行ったり先生の赤入れに対応するカタカナファイル原稿のエディタ編集をしたり…)。

すよ、などと限定することになり、その内容はスクリプト言語にはちょっと適用しにくかったりする。もっと特定の言語向けの本もあるが、Lisp であれば On Lisp[4] や Let Over Lambda[2] のように「その言語固有の世界」が大きくなってしまい、一般性はさらに乏しくなる (とくに Lisp の S 式とマクロの山は他の言語には無い世界なので)。

次に (b) については、ソフトウェアの複雑さが増すにつれて、コードそのものにおいても、動作するコードの部分 (手続きや関数の中身) 以外の比重が高くなっている。そしてその部分は、オブジェクト指向であればクラス階層の設計、関数型であれば関数群やそれが作用するデータ型の設計など、言語ごとにかなり異なる部分に焦点を当てることになり、やはり共通した「美しさ」の話にはなりにくい。さらに、コードそのものを書くときにも、フレームワーク、アーキテクチャ、ライブラリを理解し活用するという制約が大きくなり、自分の好きなようにコードを組み立てられる場面ばかりでは無くなって来ている。

そして (c) だが、かってのコンピュータに何かをさせるには自分でコードを書くしか無かった時代と比べれば、今日では大抵のことはできあいのソフトでこなせるので、コンピュータに接している人 (ほどんどすべての人?) の中でコードを書く人に比率が極めて小さい、というのが根本にあるのかも知れない。しかしそれでも、プログラムを学んだり書く人はそれなりにいるし、さまざまな言語の解説本もこれまで無かったほど多数出版されている。にもかかわらず、「美しく書く」ことにあまり関心が集まらないのは、(a) や (b) の影響によるのだろうか。

しかしそもそも、「コードは美しく書くことが結局は得ですよ」という我々にとっての「常識」が世の中に知られていないとすれば、それは我々の宣伝不足であり、このことをもっと世の中に訴えなければならないのではないだろうか。たとえば、自分は非常勤先で大学 1 年生にコードを書かせる授業をやっているが、既に高校までにコードを書くようになったと思われる学生の提出コードが字下げがめちゃくちゃだったりするのを見ると、「この人たちは自己流である程度コードを書くようになったのだろうけど、周囲にコードの美しさを説いてくれる人がいなかったのだろうなあ」と思うわけである。

最後に (d) だが、すぐ後で述べるように、最近出た本に書かれている「美しさの指針」を眺めてみると、大体はどこかで見たことのあるようなものである。ということは、もうプログラムの美しさについて言うべきことは言い尽くされてしまったから、新しい本が出ない? もしそうだとすると、こと美しさに関してはプログラミングの世界は「書法」の頃までで進歩が終っていることになるが、それは悲しすぎる…

さて、ここまで「美しさ」本は無いという線で進めて来たが、実際には「美しさ」に関わる本はいくつかあるので、以下で簡単に整理したい。

3.2 網羅的なコード本

Code Complete[14](和訳は上下 2 分冊) はプログラミングとシステム開発全部に関わる本であり、その中にはスタイルに関する話題もかなり含まれている。変数名とか制御構造とか「書法」的な内容も当然あるが、ここまでに挙げた本にない (と思う) 目新しい指摘として次のものがあった。

- ループの途中から脱出するのは読みやすいという研究結果があり、むしろ推奨されるべきである。
- インデント方式は「制御構文によって制御される部分がちょうど 1 段深くなるように」「形を重視して」決めるべき。

しかしいずれも結構細かいところで、あとは大体見たことがあるような指針ばかりである。また、Code Complete ほどぶ厚くはないが、Code Craft[3] ほかいくつかの網羅的なシステム開発本も、その一部にスタイルの話題が含まれている。これらもやはり、そう目新しいことは載っていないようだ (やはり (d) の枯渇説?)

Clean Code [13] はオブジェクト指向を前提とした本なので、Law of Demeter(各オブジェクトが知識を持ったり呼び出したりするのは直接の知合いに限るべきという法則) なども言及している。オブジェクト指向は確かに古典の時代には一般的でなかったので (Simula 言語はとっくに生まれていたけれど)、それに関する言及は確かに古典には載っていない。あと、コメントについて次の主張が書かれている。

> コメントは時間を経て嘘になる宿命にあるので、コメントに時間を費すより解説的なコードを書くことに時間を費そう。

これにはコメントを書くのが嫌いな筆者は思わず喝采してしまう。

3.3 エッセイ

次のクラスの本として、ソフトウェア技術者による「エッセイ」な本にもスタイルの話が書かれているものがある。エッセイを収集しはじめると大変すぎるので、ここはエッセイ集であるビューテイフルコード [6] から関係しそうな章を抜粋させて頂く。この本は題名が題名だけに、「美しいコード」をテーマに「必ず何らかのコードを掲げて」語る、という規則でできているので、好都合である。しかしスタイル的な「美しさ」本に相当する内容の章は意外なほど少ない。たとえば 1 章は Kernighan が書いているが、「プログラミング作法」からの抜粋的な話題で、パターンマッチャの仕様設計や言語機構の活用に主眼がある。この他の章も、アーキテクチャ的/仕様的/アルゴリズム的な「美しさ」の話が多く、スタイルの美しさはあまり出てこない。やはり出尽くしていて書きづらいのだろうか。以下では、かろうじてスタイルの話と思うものを 2 章だけ紹介する。

32 章: 働くコード

この章は著者の一人 Christopher Seiwald の「プリティコードの 7 か条」が元になっているので、まさにスタイル論議である。7 か条はほとんどごく普通のものだが、「『本のよう』であること」というものがあり、これは「行を長くしないこと」を意味している。個人的には長くてもよいと思うが…あと、「カラフルな表示でなくても読めるコードであるべき」というのがあり、これはなるほどと思う。

29 章: エッセイのごときプログラム

これはまつもとゆきひろさんの章で、要するに Ruby をの提供する構文や標準オブジェクトを組み合わせると Java の長々しいコードがこんなに短くなりますよ、「簡潔さ」は大切、という話が書かれている。簡潔さはもちろん「書法」以来ずっと言われていることだが、言語がサポートすることでかなり違うという話なのはやはり Ruby のまつもとさんという感じである。

3.4 リーダブルコード

そういうわけで「美しさ」本はないですね、という話で済ませようとしていたのだが、調べてみたら近刊 (2012 年 6 月!発刊) でリーダブルコード [1] というのがあり、これは本 1 冊スタイルの話題でまさに「美しさ」本になっている。この本については、都合により、後で触れることにする。

4 N 個の指針

ここまでほとんど他人の書いていることの紹介ばかりで来たが、いよいよ逃げ場がなくなったので、以下では筆者なりの「美しさのための指針」を順不同に挙げて議論させて頂く。題名の「N 個の…」というのは、書いてみないといくつになるか分からないし、「7 つの約束」みたいな題名になるのも嫌だったからである。[5]

指針 1 プログラムは分かりやすく書く (書)

これは由来は「書法」で、筆者が学生時代から散々言われて身体に染みついている、原理以前の原則である。しかしたまにそれを口にしたとき「分かりやすく書くってどういうことですか」と聞かれてぐっと詰まったりする。確かに抽象的な目標であり、この後出て来る事項の多くはこの具体化だとも言える。このため、具体的には説明しにくい。

この事項が意識されるのはむしろ、他の事柄を優先してしまったプログラムコードやそれを擁護する人に遭遇した時である。ネットの掲示板などではよく「同じことをするのにもこの命令よりこの命令の方が効率がよい」という発言を目にする。たとえば JavaScript では a が配列だとして、その最後に要素を追加する時に次の 2 つのどちらがよいか、という話がある。

```
a.push(v);
a[a.length] = v;
```

JavaScript では配列の添字のこれまで使っていない位置に格納すると配列が拡張されるので、このどちらも同じ動作になり、速度的にはメソッドを呼ばない後者の方が少し速かったりする。しかし、「素直にやりたいことを書いていて分かりやすい」ので自分では前者が当然と思う。ちなみに、少し

[5] なお、以下で指針末尾の (書) は「書法」、(K) は Knuth に由来していることを示す。これら以外にはとくにどれに由来している、という意識があるものはない (たまたま同じものはあるかも知れない)。

でも速い方がよいという人を説得するときには次のようなことを言うことが多い。

- 「1 週間後の自分は別の自分」なのだから、意図が分かるように読みやすく書いておかないと読めなくて困る。
- 読みやすくないことで被る時間的損失 (読み書き時のちょっとした遅延の累積や、バグを入れて仕舞ったことによる損失) を考えると、効率優先はむしろ損。
- コードの実行時間の 90%は 20%の範囲で消費されるのだから、問題のコードが本当にネックのところにあるときだけ、涙を飲んでそこだけ「速度優先」に切り替えるのがよい (当然そのことはコメントで注記する)。

補題 1-1 うますぎるコードはよくない (書)。

補題 1-2 速さのために分かりやすさを犠牲にすべきでない (書)。

指針 2 自分にとっての「美しさ」を確立する (K)

これは由来は [11] だが、やはり自分が日頃から思っていることでもある。学生さんにプログラミング入門的な科目を教える際、よく取り上げるテーマに「次の 3 つのメソッドのどれが好みか」というものがある。

```
def larger1(x, y)
  if x > y then
    larger = x
  else
    larger = y
  end
  return larger
end

def larger2(x, y)
  if x > y then
    return x
  else
    return y
  end
end

def larger3(x, y)
  larger = x
  if larger < y then result = y end
  return larger
end
```

好みのものに手を上げてもらうと、だいたい 1 番目か 2 番目が多いのだが、そこで「では 3 つのうちから大きいもの」という課題にすると、今度は 3 番目を元にするのが一番楽というオチになっている。その後で学生さんに言うことは「プログラミングでは 1 つの事柄を多くの書き方で書け、どれか 1 つだけが正解ということは無い」「どの書き方が有利かは、状況によって変わって来る」「だから、自分の中に『美しい』と思う書き方の規準を見つけて持つように」ということである。

本稿のために改めて [11] を読み返して思ったのは、やはり自分が好きな書き方で楽しく書かなければ、能率も上がらないし、品質のよいコードもできない、好きこそものの上手なれ、ということである。まさに「programming should be fun」。

補題 2-1 自分の好みによることで、一番自分を表現できる (K)。

補題 2-2 他人の好みを変えさせようとすべきでない (K)。

この指針はとても便利である: この後次第に筆者の独断的なものが出て来るのだが、あくまでも筆者の「好み」ということでご理解頂きたい。

指針 3 まず動くようにして、それから改良する

筆者の持論の 1 つに、「人間は自分の頭に載る範囲でしかプログラムを書けない」というのがある[6]。もちろん、載る量には個人差があるけれど、限界があることに変わりはない。そこでその限界を押し広げる 1 つの方法は、実際にコードを動かして見ることだと考える。というのは、動く前のコードを読んでいる時には、「ここはもしかしたらこうだけれど、ひょうっとするとこっちかも」という小さな保留を沢山つけながら読んでいるから。その保留で頭が一杯になってしまうと、それ以上載らなくなる。そこでコードが動けば、保留していたことは決着がついて解消し、余裕ができる。また、動くことで具体的イメージができて理解が進むし、嬉しい (!) ので頭脳の容量が増大する。だから、動くことは多くの効果を生み出す。

そこで、プログラムを全部作って一気に動かすのでなく、簡単なケースで動く状態にして、そこ

[6] さらに言語屋としては「よりよい言語は、頭に載る量を増大させてくれる効果がある」というのもあるのだが、それは別の話ということで。

から動くことを確認しつつ機能を追加し、最終的な目標に至るのがよい (「ソフトウェア作法」にもそうある)。もちろん、「改良」というのは機能の追加だけでなく、書き方の改良も含めている。動かしたばかりのときは、頭にそのコードが載っているので、美しくするのもやりやすい。そして書き方が整理されて分かりやすくなれば、自分の頭に載る量はさらに増える。なお、「すべて書き直す」というのも当然ながら改良のうちである。

補題 3-1 動いたらすぐに美しく整理しよう。

補題 3-2 だめなプログラムを修正するのはやめて、書き直そう (書)。

指針 4 できるだけ単純明快に書く

英語では「KISS (Keep It Simple, Stupid — バカみたいに単純にしておけ) の原理」として知られていることで、プログラミングにあてはめるなら、複数の書き方があるときに、一番簡潔で単純な書き方を選ぼう、ということになるだろうか。結局、書かなければならない (書きたい) コードはどんどん複雑になる宿命なのだから、選択肢があるなら単純なものを選んでこれから遭遇する複雑さに備えたい、ということだと思う。

それじゃあ単純な線形探索がいいの、という突っ込みがあるかも知れないが、実行して遅ければ、遅い原因箇所は改良する。そもそも探索の本体は手続きなどで抽象化した中に書くのだから、それを書いている部分では「普通の」アルゴリズムから選べばよい。それに、探索などは言語やライブラリに始めから含まれているかも知れないので、それならそれを利用するだけで済ますのが一番単純である。

また別の関連する話題として、筆者はコードはワンパターンで平凡なものが最善だと思っている。たとえば fizzbuzz 問題 (i をカウントアップしながら、3 の倍数なら fizz、5 の倍数な buzz、両方の倍数なら fizzbuzz、どれでもなければその数値を出力する) の場合、平凡に if-else の連鎖で分岐してそれぞれ処理することもできるし、3 の倍数の処理、5 の倍数の処理をそれぞれやって合流することもできる。

```
def fizzbuzz1(n)
  1.step(n) do |i|
    if i % 15 == 0 then
      puts "fizzbuzz"
    elsif i % 5 == 0 then
      puts "buzz"
    elsif i % 3 == 0 then
      puts "fizz"
    else
      puts i
    end
  end
end

def fizzbuzz2(n)
  1.step(n) do |i|
    num = i
    if i % 3 == 0 then
      print 'fizz'; num = ''
    end
    if i % 5 == 0 then
      print 'buzz'; num = ''
    end
    puts num
  end
end
```

それで結局、一番「平凡でよく使う」のは if-else の連鎖なのだから、それを使うのがその場でロジックを考えるよりもよい。平凡なものは、書くのにも読むのにも慣れていて速く、間違えにくいからである。

補題 4-1 ライブラリ関数を使おう (書)。

補題 4-2 ワンパターンで平凡なコードが美しい。

指針 5 深い入れ子は避ける

Algolo-60 以来「入れ子」はプログラミング言語の主要概念の 1 つだが、人間にとっては入れ子に対処するのは簡単ではない。つまり、入れ子の入口で「現状を覚え」、出口で「その状態に戻る」必要があるから。コンピュータにとってはスタックを使えば簡単だが、人間にはそれを 2 レベルも 3 レベルもやるのは大変。しかし今日のオブジェクト指向言語では、まずクラスがあって、その中のメソッドがあって、平らなコードが既に 2 レベル目である。その中に 2 重ループがあって、その内側で枝分かれしたら、5 レベルということになる。これくらいまでが人間的には限界と考えるべきではないか。

それを超えそうなときは、ループの内側部分を手続きにして分けて書くようにする。そうすれば、それぞれの部分ではレベルが深くならないで済む。たとえば次のバブルソートは 3 レベルだから上記の規準の許容内だけれど、入れ子を浅くしたければ下のように書き換える。

```
def bubsort1(a)
  0.step(a.length-1) do |i|
    0.step(i) do |j|
      if a[i] > a[j] then swap(a,i,j) end
    end
  end
end

def bubsort2(a)
  0.step(a.length) do |i|
    0.step(i) do |j|
      order(a, i, j)
    end
  end
end
def order(a, i, j)
  if a[i] > a[j] then swap(a, i, j) end
end
```

ただし、浅くするために手続きを多くすることはその面でのマイナスもあるので (手続きの把握が負担)、自分にとって適度な深さまで許容し、つらくなるところで手続きに分ける、ということかと思う。また、次の指針と関連するが、ちょっと普通でなくても書き方の工夫で入れ子を浅くできるならその方が良いとも考えている。

補題 5-1 自分に気持ち良いレベルの手続き分割と入れ子深さのバランスを取ろう。

補題 5-2 書き方の工夫で入れ子が浅く認識できればそれでもよい。

指針 6 複数出口は積極的に使う

次第に物議を醸しそうな話題になってきました。ものの本には「複数の出口があるコードはよくない」という話が書かれていることがあり、たとえば Pascal などは手続きの出口が 1 箇所 (末尾) しかない。それはそれで哲学というものだけれど、自分は C などで `return` を多用している。

なぜそうするかというと、ある手続き X を書いている時は、「X の動作を実現する」ことが目的なので、あれこれの場合に対処していて「この場合の X の動作は完了」となったとき、そこに `return` を書いてその場合のことは頭から消去したいからである。たとえば探索なら、見つかったときに「仕事は終わり」ですよね?

```
def is_in(a, x)
  a.each do |y|
    if x == y then return true end
  end
  return false
end
```

それをわざわざ、他の流れと合流するまで待って、合流してから行儀よく出口に、というのはその分だけ覚えておかなければならないので、人間にとって負担だと考える。その意味では、「この場合はループ終わりね」の `break`、「ここはもっかいね」の `continue`/`next` も愛用している。たとえば、ループの中で枝分かれがある場合を考える。

```
loop ...
  if ... then
    // case A
  else
    // case B
  end
end
```

ここで場合 A と B が同じくらいの複雑さならこれでいいのだけれど、A だけが簡潔だったり、場合によっては「空っぽ」だったりすると、次のように直してしまう。

```
loop ...
  if ... then
    // case A
    next
  end
  // case B
  ...
end
```

要するに、A の処理は終わったんだからもう「もっかいね」と言っておけばあとは考えなくていい、だからそう書きたい、ということである。そうすればそのことは忘れて、あとは面倒な B に取り組めばよい。それに、こういうときはだいたい B の中に入れ子があるので、こうすると入れ子の深さが 1 段もうかる。

補題 6-1 「以上です」と書いてあとは忘れてしまえるならそうしよう。

補題 6-2 複雑でない場合はなるべく早く片付けて複雑なものに集中しよう。

指針 7 コードは短く

もちろん、1 行プログラムのような難解なものをめざしているわけではないが、コードは短いほどよいと思う。島内剛一先生の教えにも「プログラムは短くして速くしよう」というのがあったが、個人的には「一覧性」つまりスクロールしたり視線を動かさなくても見渡せて読める、というのが重要だと考えている。あるコードを理解したり直したりするには、それが頭に載るまで反芻する必

要があるけれど、その時にすぐ見渡せるというのは有効である。

ここでいう「短い」というのには、簡潔に書いてあるということに加えて「1 行に沢山詰め込んで短くする」というのも含まれているので注意。たとえば筆者は、else 部の無い if で then 部が短くて 1 行に書けるならそのようにしたくなる。

```
if ... then
  A
  B
  C
end
// following code...

if ... then A; B; C end
// following code...
```

なぜなら、1 行に書ける程度の if 文であれば、「もしこの時はこれをするのね、まあどちらにせよ…」のように一瞬で読み終わってその先に注意を払いたいからである。このあたりは、標準的な書き方から外れるのでかなり異論があると思うが。

if 文に限らず、通常の文の並びも横に並べると行数が減らせる。通常の言語は (アセンブリ言語以来の伝統か) 個々の文は縦に並べて書くのに適した設計なので、横に長くするのは違和感がある。Lisp の場合は全部式だから詰め込んでもあまり問題がない。たとえば筆者らが設計したドリトル [5] はメッセージ送信のカスケード (メソッドの返値に対して続けてメソッドを呼ぶこと) を用いて横に長く書くことを前提としていて、その方が自然である。

```
「カメ!50 歩く 90 右回り
      50 歩く 90 左回り」!4 繰り返す。
```

ドリトルのプログラムを見ると、縦に並べている流派 (他の言語をやったことのある人?) と横に長い流派 (ドリトルでプログラミングを始めた人?) があるが、後者の方が短くて済むので読みやすい。

補題 7-1 1 画面で済む手続きは読みやすい。

補題 7-2 1 行で終わる枝分かれは読みやすい。

指針 8 名前の長さは公開度につれて長く

一般的には「変数名には分かりやすい名前をつけ、へんに省略しないように」と言われることが多いが、そうだろうか? 変数はコード中に沢山現れるので、変数名が長いとコードの文字数が多くなり、1 行に入る量も少なくなる。なので筆者は「1 文字の変数大好き」である。ただしもちろん、ローカル変数に限るけれど。世の中では 1 文字変数はよくない、という論が多いようだけれど、数式なんて変数名はみんな 1 文字ですよ?

さらに変数名に数字を使うなという指針が書かれている本もあるけれど、たとえば `s` に格納されている値をもとにちょっと加工したものは `s'`、`s''`…としたいけれど、普段使っている言語ではそう書けないので、代わりに `s1`、`s2`…としている。

もうちょっとスコープの広い名前 (インスタンス変数とかクラス内だけのメソッド) も、遠慮なく省略して短くする。それこそ Unix の rm、cp、mv とかである。この範囲の名前はそのプログラム単位を本腰を入れて読む人が扱うので、省略の規則はこうですよ、というのを頭に入れてから読むのでよいと考えている。なお、このときその変数のクラス名がフルスペルになっていれば、説明としてちょうどよい。

```
VarTable vtbl = new VarTable(...);
```

ただしこれは「用途のはっきりした」クラス向けで、リストみたいに汎用のクラスに対してこれをやると「実装を表す」名前になってしまうという弱点がある (少数のインスタンス変数対象ならそれでもよいと思う)。

最後に、一般に参照されるもの (グローバルなクラス名や公開されたメソッド名) は、それはやっぱりフルスペルだと思う。ただ、メソッド名はクラス名を前置して (ないしクラスの文脈で) 扱うので、その範囲でちゃんと分かればよいと思っている。

補題 8-1 局所変数は積極的に 1 文字でよい。

補題 8-2 モジュール内だけで参照する名前は分かる範囲で省略する。

指針 9 クラスを積極的に作る

C++や Java など主要な開発言語がクラス方式のオブジェクト指向言語であるにも関わらず、どのようなクラスを作るかについては、新しい本にもあまり載っていない。もちろん、オブジェクト指向設計の本にはその話題があるが、ここでテーマにしているスタイルレベルの本はクラスは守備範囲外ということだろうか。つまり、プログラム構造を検討する中でクラスを規定するので、コーディングレベルの話ではない…

しかし筆者は、コーディング時にもクラスは積極的に活用すべきだと考えている。もともと、昔

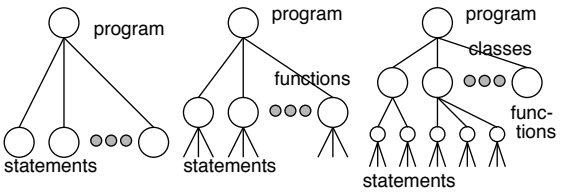

図 1: 階層構造の進化

は巨大な何千行もの文だけから成るメインプログラム 1 個、というプログラムもあったのが、構造化プログラミング運動によって「関数→文」という 2 階層になることで大きなプログラムが作れるようになった。それは完成した関数の中は見なくても呼べばよいという抽象化が行えるからである。そしてプログラムがさらに大きくなったことに対応してできたのが、関数とデータを集めたモジュール (抽象データ型、クラス型のオブジェクト指向言語ではクラス) という単位なわけである (図 1)。

従って、これまで「複数箇所の共通処理は関数としてくくり出す」ことでコードを整理してきたのが、関数や変数 (〜データ構造) が多くなってしんどくなった場合、次にやるべきことは「クラスとしてくくり出す」だと考えている。そのきっかけは、「複数のデータが組になって働く場合」だと思う。

例えば、名前に 0 から始まる番号を振って、番号から名前、名前から番号の両方向に行き来したいとする。それには、番号から名前はリスト型、名前から番号はマップ型でできるので、その 2 つのデータ構造を組で持つことになる。

```
List<String> id2name = new ...;
Map<String,Integer> name2id = new ...;
```

しかしこのままでは、2 つのデータ構造が組になって働くことは「個々のアクセス箇所を読む」ことでしか分からない (まあコメントは書くのだろうけど)。そこで、これら 2 つを包むクラスを作ることを考える。

```
class NameId {
  List<String> lst = new List<String>();
  Map<String,Integer> map = new Map<...>();
  void add(String s) {
    map.put(s, lst.size()); lst.add(s);
  }
  int toId(String s) { return map.get(s); }
  String toName(int i) {return lst.get(i);}
}
```

これにより、次のような利点が得られる。

1) 裸のデータ構造だと何が組になりそれをどのようにアクセスするかの手がかりがないが、クラスにすることで用途や使い方が明示され、かつ用途外の使い方が排除できる。
2) 引数や変数の数が減らせ、それらが何であるかが明確になり、型検査により取り扱いの間違いが減らせる。
3) 「追加だけできる」「参照だけできる」みたいな制約がインタフェースにより表現可能で、そのような用途だけの場面 (手続きに渡すなど) で制約をコードに明示でき検査可能となる。

なお、このようなクラスの抽出はどちらかというとオブジェクト指向文化の中ではリファクタリングとして扱われていると思う。このような場合にとどまらず、「プログラムが扱う単一の概念はそれぞれクラスにする」という普通のオブジェクト指向的指針も当然だと考える。

補題 9-1 組になるデータ構造はクラスにすることを考える。

補題 9-2 プログラムで扱う単一概念はそれぞれクラスにすることを考える。

5 リーダブルコード、そしてまとめ

そういうわけで、このあたりまで本稿の内容を大体まとめたところで、リーダブルコード [1] が入手できたので読んでみたが、これが前述のように「最近は無いですね」と言っていた 1 冊まるごとの「美しさ」本であり、しかも前述の「指針」といくらか被っている。大変困ったけれど、今更どうにもならないので、概要を紹介しつつ「まとめ」を書かせて頂くことにする。

まず、この本では 1 章で「基本定理」として「コードは他人が最短時間で理解できるように書く」ことを挙げている (←指針 1 に相当)。続いて各章のテーマは次のとおり。

- 2 章: 名前にうまく情報を詰め込むべし。スコープを勘案して長さを選ぶ (←指針 8)。3 章: 誤解されない名前を選ぶ。
- 4 章: 美しさは分かりやすさのため大切 (一貫性、揃える、等)。

- 5 章: コメントすべきことを知る。6 章: コメントは正確で簡潔に。
- 7 章: 制御フローを読みやすく (←指針 5・6)。
- 8 章: 巨大な式は分割。9 章: 変数は少なく/スコープを小さく/書き換えを避ける。
- 10 章: 共通の下位問題を関数に抽出。11 章: 1 度に 1 つのことをする。
- 12 章: ロジックを簡潔にし、ライブラリを使う (←指針 4)。
- 13 章: 短いコードを書く。14 章: テストと読みやすさ。15 章: 具体例。

まあ、こうして整理してみると共通するテーマはあってもアプローチは筆者と違うことが多いのでそんなには被っていないかな…と少し安堵[7]。また、名前のつけ方とコメントの書き方の箇所は非常に丁寧で説得力があり、コメント嫌いの筆者もコメント書こうかなという気になる位であった。

一方でお仕事の本らしく「正しさよりもプロジェクトでの一貫性を優先」のように自分のやっていること (指針 2) と違うところもあるし、オブジェクト指向は前提のようだがクラスの積極活用的なところ (指針 9) は無い。

ともあれ、内容的には古典の本からそう大きく違っていないとしても、このように単独の (そしてぶ厚くない)「美しさ」本が新たに刊行されて人目に触れ、その内容が納得のいくものであるというのは、喜ばしいことだと思う。

さて、ここまで過去の本とか最近の/新しい本も検討しつつ自分なりの「美しさの指針」をまとめたわけだけれど、改めて振り返って見ると自分の一番重視している事柄は次のものだと思う。

> 人間は自分の頭に載る範囲でしかプログラムを書けない。だから、その限界を押し広げ、できるだけ多くの必要なことが頭に載り、必要でないことは載せなくて済むようにプログラムを書くのがよい。

これをもっと短くするなら「美しいコードは頭に載りやすい」ということか。そして、頭に載る載らないというのは人間の認知的特性と直接的に関連しているので、上記を言い替えると「認知指向プログラミングスタイル」ということになるだろう。[8]

そして、ここから先は全然実践できていないのだが、アートの世界でよく出て来るリズム、規則性、対比、変化 (バリエーション) などはすべて、人間の認知的特性に根ざして「気持ち良さ」を提供してくれている「美しさの要素」だと思う。だから、自分の書くコードでもいつかそれらの概念が具体的に取り入れられるようになりたいな、とおぼろげに考えているのだった。

[7] `return`、`continue`/`next` のところは同じことが書いてあるので結構ぎょっとした。

[8] またまた私事なのだが、木村先生が人間の認知特性の研究をされていた頃、助手をやっていた筆者はシステムや言語に夢中であり、人間みたいな複雑でよく分からないものを研究するなんて面倒、みたいに考えていた。しかし今になってみると、そのころ興味ないと思いながら「門前の小僧」で知ったことが沢山役に立っていて、昔の自分を叱責したい思いで一杯である。

謝辞

本発表の機会を与えて下さった、和田英一先生をはじめとする 2012 年夏のプログラミングシンポジウム幹事団の皆様に感謝します。また、本稿で取り上げた初期の本の多くに接する機会を与えてくださり、またそこに出て来る考え方を繰り返し具体的に教えてくださった、木村　泉先生および木村研究室 (当時) の皆様に感謝します。

参考文献

[1] Dustin Boswell, Trevor Foucher, 角訳, リーダブルコード, オライリー, 2012.

[2] Doug Hoyte, HOP プロジェクト訳, Let Over Lambda, エスアイビーアクセス, 2009.

[3] Pete Goodliff 著, 後藤ほか訳, Code Craft, 毎日コミュニケーションズ, 2007.

[4] Paul Graham 著, 野田訳, On Lisp, オーム社, 2007.

[5] 兼宗　進, 久野　靖, ドリトルで学ぶプログラミング — グラフィックス, 音楽, ネットワーク, ロボット制御 — 第 2 版, イーテキスト研究所, 2011.

[6] Brian W. Kernighan, John Bentley, まつもとゆきひろ他著, Andy Oram 他編, 久野禎子, 久野　靖訳, ビューティフルコード, オライリー, 2008.

[7] Brian W. Kernighan, P. J. Plauger 著, 木村訳, プログラム書法 第 2 版, 共立, 1982.

[8] Brian W. Kernighan, P. J. Plauger 著, 木村訳, ソフトウェア作法, 共立, 1980.

[9] Brian W. Kernighan, Rob Pike 著, 福崎訳, プログラミング作法, アスキー, 2000.

[10] 久野 靖, ビューティフルコード (プ会 2008.4), プログラミング・情報教育研究会資料, 2008. http://www.oreilly.co.jp/editors/p0804.pdf

[11] Donald E. Knuth, 有澤訳, 芸術としてのプログラミング (1974 年 ACM チューリング賞記念講演), in 文芸的プログラミング, アスキー, 1994.

[12] George Ledin Jr., Victor Ledin, 篠原訳, プログラミング 250 の心得, 多賀出版, 1981.

[13] Robert C. Martin 著, 花井訳, Clean Code, アスキー, 2009.

[14] Steve McConnell 著, クイープ訳, Code Complete 第 2 版 上・下, 日経 BP ソフトプレス, 2005.

[15] Gerald M. Weinberg 著, 木村ほか訳, プログラミングの心理学 25 周年記念版, 毎日コミュニケーションズ, 2005.

[16] William Strunk Jr., E. B. White, The Elements of Style, 3rd ed., Macmillan, 1979.

A 質疑

Q1. (笹田@heroku) 長い行というのは具体的にどれくらいでしょう。大きいいディスプレイもあるけれど…

A1. 長くしすぎて表示が折り畳みというのは駄目だと思います。自分は 80 × 24 が身についているので、今 24 はさすがに短いですけれど、幅は 80 くらいでどうですか。

Q1. 80 は駄目だと思いますけれど。高解像度なディスプレイでは非常に長い行も作れますから…

A1. 非常に長い行があると、残りの行はその部分は空白なわけでもったいないです。だから読みやすい長さを決めて、自分では 80 で頑張ることにして、その長さ内では 1 行に入れる、というふうに考えています。

Q2. (???) 今日では横に長いディスプレイがあって長い行が使われるけれど、自分は右目が悪いので長いと非常に困ります。だから大きいディスプレイでも窓は縦長で複数並べたりします。

A2. 横幅がどれくらいが心地よいというのは人によって違うのが当然だと思います。その意味では Lisp のようにプリティプリントすれば幅が変えられるというのがよいのであって、普通の言語のように手で調整するのは不便であると。あと、本なども多段組みするのは横に長いと見づらいからだと思います。つまり横に眼球運動が必要だと見づらいので、あまり狭すぎるのはおかしいけど、ある程度までの長さの上限を決めてそれでやるということではないでしょうか。

Q3. (後藤@???) 関数は最初に機能を設計して作成するわけだから、後から長いので分割しよう、ということは起こらないように思う。というか、そのような状況が想像できない。

A3. 書いて頭に載っている時はよいと思っても、後から読んで長くてつらいとかであれば、そこで分けるというのはあるかと思います。また、書いていると長くなってしまった、だから分ける、ということもあると思います。

Q3. 後から分けるというのはやはりないように思う。他人に読ませることを考えてそのために後から大きさを調節するとかならあるかも知れませんが。それでも最初からそれを前提に作るのなら後から分けるということはないと考えます。

A3. そのようなお考えなのは分かりましたが、自分ではやはりプログラムは書いてみないとどれくらいになるか分からない、ということがあると思います。ですから、書いてみた結果長くなったので構成を変更する、というのはありだと考えています。

Q4. (大島@東大) 与えられたコードが目を塞ぎたくなるようなひどいものだった場合どうするのでしょうか。

A4. 他の人のコードを読むことは多くないので…自分でそのコードにコミットして直したり改良するということなら、自分で正しいと思うスタイルに直すのだと思います。しかしそうでなくて一時的に見るだけということだと、直すのは大変ですから、書いた人のスタイルをそのまま観賞するということかなと思います。

Q5. (河内谷@IBM) 協同で作業する場合、価値観が衝突することがあるかと思いますが。また、動いた後でより美しいコードにしたいというモチベーションを作る方法などはないでしょうか。

A5. ワインバーグの「エゴレスチーム」のように、価値観を共有するようなチームができればいいわけで、そのために美しさとは何かとか、美しくしたいということを日常的に話し合うような環境を作ればいいのかもと思います。自分一人では難しくても、仲間で美しくしたいという合意ができていればそれに力を割けると。河内谷さんのところでそういう環境ができたらぜひコードを読ませてください。

Q6. (???) Python とかは言語仕様でインデントを制約しているが、同様に言語仕様で行数を短く制約したりとか、コピーペーストが弾かれるとかも可能と思うが、そのようなものはどうでしょう。

A6. 今日喋ったのも自分はこうだけれどという提案であり、どれも一概には言えないと思います。コピーペーストも行数が少なくてその方がぱっと見えてよいような場合にはそれでもいいと思うし、機械的に弾くというより、それぞれの人のなかで統一が取れていればいいのではないでしょうか。

Q6. 人間は悪いコードを書くという前提に立った言語もあっていいと思いますが。

A6. Haskell みたいに関数は副作用なしというのを前提とした言語もあるけれども、自分は「書かせない」というより、ぐちゃぐちゃにも書けるけれども、人間がぐちゃぐちゃに書いたあとで顧みて直すみたいなのもいいんじゃないかと思っています。