

[フレッシュマンに向けたプログラミングのススメ]



1 読みやすいコードが良いコード

平鍋健児 | (株) 永和システムマネジメント



“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

— Martin Fowler ¹⁾

学業プログラミングと商業利用される製品やサービスの商用プログラミングが圧倒的に違うのは、その寿命、分量、そして経済的な価値にある。学生時代のプログラミング対象の多くは、課題や研究対象としたコードで、数名の限定された人に使われる、数百行のコードだ。しかし、仕事でのプログラミング対象は、数年以上の時間生存し、多いときには100万行を超えるコードであり、数名以上でチーム開発する。そして、作られたプログラムはシステムとして稼働し、社会活動で利用され、対価を生む。

こうなると、書くコードの質の中心は、読みやすいこと、変更しやすいこと、テストしやすいこと、になる。コードは誕生の時点（はじめてあなたが書いた時点）から多くの人の目にふれ、手にかかり、成長していく。書くよりも読まれる回数と時間が圧倒的に多くなる。だから、最初のアドバイスはとにかく読みやすいコードを書くことである。書くのに費やす時間のコストを、（自分も含む）後で読む人のコストの削減のために払っておくのだ。プログラミングという活動は、製造ではなく設計であり、工業製品の生産活動より文芸に近い活動だと思う。それも複数人で書くソーシャルな文芸活動だ。私が知っている良いプログラマに文系出身の人が多くいるのも、そういうことではないか、と思う。読みやすいコード、さっと理解でき、修正しやすいコード。

それはどんなコードなのだろう。

このテーマを、3つの視点から書いていきたい。

1. コード片の微視的視点、2. 設計としての大域的視点、3. ソーシャル活動としての視点、である。

コード片の微視的視点

プログラミングという活動は、文字を使って意図を運ぶ創作活動だ。分かる人には分かる、という類の芸術ではない。できるだけ多くの人に読んでもらって素早く理解してもらうような「言葉選び」と「章立て」をしていく。言葉選びは「識別子の名前付け」、章立ては「モジュール分割」にあたるだろう。

良い名前を選ぶ

プログラミングでは、変数、関数、クラス、などなどの識別子にプログラマが自由に名前を付けることができる。たとえば変数名である。この名前は（処理系からすると）なんでもよく、a1 という名前でも OK だ。しかし、そのような意味のない名前は読み手をととても苦労させる。コードを読んだら実はこの変数はユーザの名前を保持していることが分かり、この変数名を user_name に変えた瞬間、圧倒的に読みやすさが増すだろう。

そのものが運んでいる意図をぴったりと伝える名前を探そう。時間を取って「探す」のだ。この名前選びに、プログラミングの時間の半分を使ってもよいくらいだ。オンラインのシソーラス辞書（類義語辞書）をあためよう。tmp, ret, data という名前をやめよう。既存の変数名の最後に“2”が現れ

たら注意しよう (findPlayer という関数名があるときに findPlayer2 という名前を付けるなど)。これはおそらく Copy&Paste して一部を修正した関数であり、名前付けに時間を使っていない証拠だ。さらに、頻出する名前の一部の語彙を注意深く選ぼう。remove か delete か、start か begin か、というレベルだ。一貫性を持って名前付けしよう。

問題領域から名前を取ろう。コンピュータの言葉でなく、ユーザが使う言葉。開発の会話で流通している言葉を使うことによって、その言葉が持つ認知度を、読みやすさに利用しよう。

変数のスコープを短くする

コードを読むときに、人間はいったん変数名などを短期記憶に置いてコードを読み進める。その名前がコードテキストの中でアクセスできる範囲を制限しよう。最も広いスコープを持つのはグローバル変数だ。まずこれを避ける。これで、常にこの変数の存在を気にかける必要がなくなる。次に、変数の宣言をできる限り遅らせる。宣言だけで初期化をしていない変数は人間の記憶コストが高い。コード上は可能な限り最も下に持っていく。さらに、使う直前まで宣言を遅らせよう。あるブロックの中だけ使われる変数であれば、ブロックの中だけに、たとえばループ内でしか使われない変数はループ外でなくループ内で宣言しよう。

特に、スコープが短い変数に限っては、本当に短い名前を使ってよい。たとえばループカウンタは、element_index でなく i の一文字でよい。これは意図を運ぶ名前の例外だが、短いスコープ (ループ内) に限ってはむしろこちらを良しとする。

同じ変数を別目的に使いまわさないことも重要だ。途中から意図が変わる名前は読解がひどく厄介になってしまう。別の名前の変数として別のタイミングで宣言しよう。

設計としての大域的視点

コードが大きくなると、全体をモジュールに分割する必要がある。モジュールを抽象的な言葉として使ったが、それは関数であったりクラスであったり、それらをまとめたファイルやパッケージ、といった構造物だ。さらにこれらの上位には「アーキテクチャ」という全体設計方針を司る大域構造も存在する。

良い名前を選ぶ (再度！)

ここでも名前の選択は重要だ。そのモジュールがモジュールとしてうまく役割を持ってまとまった単位となるかどうかは、「良い名前がつくか」という観点が良い指標となる。etc, util という名前が付いたらそのモジュールの必要性を疑うのがよい。まとめてはいるが、何かの入ったバケツという程度の意味にしかない。力強い名前が付く単位でまとめるべきだ。

良い名前を選んでいくと、識別子の意味の周りに他の識別子が自然に集まっている感覚を得ることがある。たとえばパッケージの中にクラスがあり、その中に変数とメソッドがあり、という名前の階層構造が、問題領域のオントロジから、素直に類推が効く名前付けになる。実行時にどう処理が流れるか、ということではなく、「世界はこう組み立てられているはず」、という感覚だ。

知識領域の混在を避ける

あるモジュールの中に、複数の知識領域の言葉が混在するのは良くない設計の例だ。たとえば、1つの関数の中に、コンピュータのメモリを扱うコードと、ビジネスロジック (業務のルールを表現するコード) が混在したら、それは間違った分割を示唆している。同様に、UI のコード、通信のコード、データベースの初期化コードなど、こういった出自も必要知識も別のものは別々の関数に、さらに上位のモジュールでも分割すべきだ。

コードを読んでいき、これらの別の領域のコードが混ざって現れると読んだり修正したりするのに大量の知識が必要となる。あっちを調べてこっちを調べて、という作業が必要になる。さらに、ある領域での変更が大域構造を伝わって横断的に広まり、全体の修正が必要になる。1つの領域の修正は、1カ所にとどまるべきだ。

Design Patterns, Principles, Refactoringの存在を知る

どうモジュール分割すべきか、という課題は歴史的にも古く、多くの知見が存在する。繰り返し現れる問題とその文脈、解決方法に名前をつけたものが Software Patterns として多く知られている。最も有名な GoF の 23 パターン²⁾には目を通しておくことをお勧めする。それと同時に、ソフトウェア設計の基本原則を集めた Robert C. Martin の SOLID 原則、さらに、設計から実装という通常の流れではなく、実装から再設計する Refactoring という手法が存在する。これらにも良い設計の考え方が含まれるが、紙面の都合で説明できないが、目を通しておくことを強くお勧めする。

これらは、ともにオブジェクト指向プログラミングを前提としてのものである。今後より一層、データ指向や関数型、さらに AI を活用したプログラミングが増えてくるものと思われる。これらの分野でも、新たに Patterns や Principles, Refactoring が登場するであろう。もし読者の中にこの分野（設計知識の再利用としてのパターン）に興味がある方がいたら、パターンのライティング活動にも参加することをお勧めする。

ソーシャル活動としての視点

多くのソフトウェア開発は、チームとしてのソーシャルな活動である。この視点から、いくつか紹介しよう。

チームでコーディング標準を合意しよう

Extreme Programming のプラクティスの1つである。チームで集まり、コーディングの作法についてのルールづくりを一緒にしようというプラクティスだ。人それぞれにインデントの数やカッコの付け方の位置、if の後にスペースを置くかどうか、return の後にカッコを付けるかどうか、などなど、多くの「好み」が存在する。これらは好みの問題である反面、チームでこの好みに一貫性がないと、非常に読みにくいコードになる。大切なのはどちらが良い悪いではなく、「一貫性がある」ことだ。チームで決めよう。

この活動は、チームの個々人の好みの表明にもなり、会話を活発にするきっかけにもなる。ネット上に各言語のポピュラーなコーディング標準がいくつも発見できるだろう。これらをもとに話し合ってみよう。

また、各言語、各 IDE（開発環境）には、このルールに則らないものに警告を出すツールが存在する。決めたルールを元にこれらのツールを設定し、コードをチーム共有のリポジトリ（共同のコードベース）にコミット（統合）する際に、再度自動で統一するのがよいだろう。

オープンソースに参加しよう

業務でのプログラミング活動をしていても、必ずオープンソースのライブラリやフレームワークを利用することになるだろう。もし、そのオープンソースが気に入ったら、実際にそのコードにコントリビュートしよう。ちょっとした機能追加や、バグ修正などを、GitHub を経由して作者に伝えるのだ。

最初は敷居が高く感じるかもしれないが、そこでのコミュニケーションは、自分のプログラマとしての腕前をあげると同時に、世界各地のプログラマとのソーシャルな交流の場となり、人脈づくりにも役立つだろう。ぜひトライしてみてほしい。

エピソード

最後に、こういったコードが読みやすいコードだろうか、ということを思い出させるエピソードを1つ紹介したい。Wikiを作った Ward Cunningham がインタビューで、彼が会った良いコードについて、こんな話をしている。

“そのコードは、素晴らしいコードだった。あるとき、私はそのソースリストをプリンタに出力したのだが、それを落としてしまい、ページがバラバラになってしまった。しかし、それでも、そのコードは読めたのだ。そのコードは、どこからでも読み始めることができ、意図が明確だった。”

「どこからでも読み始めることができ、意図が明確なコード」、これが素晴らしいコードの1つの例ではないだろうか。

参考文献

- 1) Fowler, M. : Refactoring : Improving the Design of Existing Code, Addison-Wesley Professional (1999).
- 2) Gamma, E. et al. : Design Patterns : Elements of Reusable Object-Oriented Software, Addison-Wesley Professional (1994).

(2019 年 1 月 15 日受付)

■平鍋健児 k-hiranabe@esm.co.jp

(株) 永和システムマネジメント代表取締役社長、(株) チェンジビジョン CTO。福井で受託開発を続けながら、オブジェクト指向設計、組込みシステム開発、アジャイル開発を推進し、UML エディタ astah* を開発。国内外で、アジャイル開発の普及に努める。ソフトウェアづくりの現場をより生産的に、協調的に、創造的に、そしてなにより、楽しく変えたいと考えている。著書『アジャイル開発とスクラム〜顧客・技術・経営をつなぐ協調的ソフトウェア開発マネジメント』、翻訳『リーン開発の本質』、『アジャイルプロジェクトマネジメント』など多数。

