［フレッシュマンに向けたプログラミングのススメ］ 基般

2 コードリーディングと写経のススメ

油井　誠 | トレジャーデータ（株）

業務としてのプログラミング

皆さんは業務としてプログラミングに取り組んだことがあるだろうか．大学や大学院でプログラミングをしてきたが，業務としてこれからプログラミングに取り組もうという方の中には，多人数での開発，コード品質の確保，また設計・テストといった開発プロセス面で大学時代に行ったプログラミングとの違いに最初は戸惑いを感じる人がいるかもしれない．

多人数での開発では，意思疎通のために設計ドキュメントの整備やコーディングルール等の共通化，Jira 等のタスク管理システムを利用した進捗管理，バージョン管理システムを利用したブランチ駆動開発，継続的インテグレーションツールを利用したソフトフェアビルド作業とテストの自動化による品質管理等が行われる．1）クライアントやチームメンバとの意思疎通，2）ソフトウェアの品質要求の 2 点は個人で利用する目的のプログラミングとの違いであり，こうした多人数での開発プロセスに慣れることは，プログラミングを業務として行う上で重要である．

本稿では，これらの項目からは離れて**「美しいコード」**を書くことの意義と，美しいコードを書く能力を培う上で個人的に重要だと思う**「コードリーディングと写経」**について触れる．なぜならば，チーム開発以前の根源的な問題として，コード品質の確保に美しいコードを書くことが重要であり，そうした人に見られる（見せられる）コードを書くという意識を持つことが業務でプログラミングをする上でも重要だと思うからである．

コードリーディング

コードリーディングの技術はソフトウェアエンジニアにとってきわめて重要である．チームで開発していると他のエンジニアの書いたコードや関連モジュールを理解するためにコードを読む必要がある．また，利用するライブラリの挙動の詳細を把握するために実装を追ったり，設計テクニックを参考にするために類似のソフトウェアのコードを読むこともあるだろう．ときには，障害の原因となるバグを発見するためやセキュリティ監査目的にコードを精査することが必要となるかもしれない．

一般的に，大きなプロジェクトであればあるほど全体を把握することは難しくなる．こうしたときは統合開発環境（IDE）や静的解析用ツール（Ctags 等）を有効に活用して目的のコード断片を見つけたり，関数やモジュール間の呼び出し関係を素早く把握することが求められる．コードリーディング自体が熟練と技巧が必要な技術といえる☆1．

コードリーディングのメリット

コードリーディングのメリットにはさまざまなものがあるが，1 つ挙げるとすれば，本格的な（優れた設計の）ソフトウェアのソースコードを読んで「ソースコードを読む力」を蓄えることで，「美しいソースコードを書く力」が身につくということである．言い換えれば，ソースコードを読む力なしに，優れたソフトウェアを開発することはできない

☆1 コードリーディングの技巧そのものについては『Code Reading―オープンソースから学ぶソフトウェア開発技法』などの書籍がある．

と言っても過言ではない．私の周りにいる優れたプログラマは，例外なく，他人のコードやライブラリの実装をよく読む習慣を持っているし，他人にコードを見られることを意識したコードを書いている．

幸いなことに，オープンソースソフトウェアやGitHub上で公開されているソフトウェアから，コード職人が書いた良質なコードをいくらでも読むことができる時代である．筆者自身もこれまでに多くの優れたオープンソースソフトウェアからモジュール設計方法，抽象化方法，高速化のための技巧，関数名や変数名の命名規則のあるべき姿，シンプルなデザインの重要性を学んできた．コードリーディングの習慣のない読者には，何か関心のある目標を1つ定めて，優れた設計のオープンソースソフトウェアのコードを隅々まで追い，その裏にある設計思想やデザイン技法について学ぶことをお勧めする．他人のコードを理解する研鑽を積むことは，業務でプログラミングに従事する上でも必ず助けとなろう．

ここで重要なのは，真に優れた設計のソフトウェアを選んでコードリーディングを行うことである．有名なソフトウェアであっても実際に読んでみると，実は古い設計であったり，歴史的な経緯による回避策的なコード（これはこれで勉強にはなる）が散見されるなど，設計的には優れないこともあるので実際にはある程度のスクリーニング的な探索も必要となるかもしれない．

具体例

筆者の専門とするデータベース領域でいえば，PostgreSQLはC言語で書かれた大規模ソフトウェアであるのによくモジュール化がなされた綺麗なコードベースである．データベース管理システムの実現方法を学ぶ上で参考となる．GnomeプロジェクトのGlib（glibcではない）はC言語でのオブジェクト指向的なことを実現する上で参考になる．Pythonでいえば，Pythonの疎行列ライブラリのscipy.sparseなどの内部構造を理解することはPythonで行列演算を行う場合や空間計算量の小さいプログラミングを実現する上で助けとなるだろう．また，機械学習ライブラリのScikit-learnはAPIの抽象化（共通化）の優れた実現例である．Javaライブラリでいえば，Google GuavaはJavaベースプロジェクトで利用されている優れたライブラリであり，内部実装やAPI設計からJavaコーディングのいろはやコーディングイディオム（慣習）を学ぶことができるだろう．

普段利用するプログラミング言語の標準ライブラリは，その言語の特有のベストプラクティスが詰まっているためAPIドキュメントを参照するだけでなく内部実装を積極的に読むことをお勧めする．

守破離と写経

優れたソフトウェアの設計を学び，それを実践する訓練を積むことではじめて，優れたソフトウェアを自ら開発することができる．ここで，プログラミング能力の成長過程を守破離の概念に当てはめてみると，次のような成長のプロセスがある．

- **守**：大学の授業や技術書からプログラミングの基礎的な知識を習得し，指示や習ったことに基づいて小規模なソフトウェアの実装を完遂することができる．
- **破**：中～大規模なソフトウェアの内部を把握して，問題点を修正したり，改善したりすることができる．
- **離**：事業品質が求められる開発難易度の高いソフトウェアを主体的に一から設計・開発できる．複数人で開発されるソフトウェアモジュールを設計・開発できる．

コンピュータサイエンスを履修する学生の多くは，卒業研究や修士研究を終えるころには何らかのソフトウェアを実装することができる「守」の段階にあると思う．ここから，さらにプログラミング能力を「破」「離」と進めるにあたって助けになると思うのが**写経**である．

写経のススメ

写経というと，習字の習い事のように，ただただ手本となるプログラムをそのままタイプする退屈な作業をイメージするかもしれないが，ここでいう写経は指と同時に頭を使った写経である．参考書のプログラムを読んで理解することや，既存のコードを眺めて内容を把握することも重要であるが，元となるプログラムを書いたプログラマの思考をトレースして理解と批評を重ねつつ，実際に自分でコーディングしたプログラムを動かしてみることで得られる理解はより深いものとなるだろう．場合によっては，元のコードをさらに見通しのよいものにリファクタリング（洗練）することをお勧めする．

写経から何を得るか

プログラミング入門書を読むだけでは習得できないベストプラクティスを，優れたコードの写経によって得ることができる．基本的な文法に加えて，写経から学べるものには次のようなものがある．

• 言語固有の文化

プログラミング言語によって，変数や関数の命名規則やコーディング方法のベストプラクティスが異なる．Ruby らしいコード，Python らしいコード，Java らしいコードなど，それぞれの言語ごとに綺麗な書き方が存在する．

たとえば，private/protected/public 等のアクセス修飾子がない言語では変数名によって変数のスコープを表現することがある．また，Python のようにリスト内包表記をサポートする言語では，同じ処理を実装する場合でも if 文とリスト内包表記の使い分けなどの設計判断が存在する．また，同じ言語でもラムダ式をサポートする言語環境（たとえば Java 8）としない言語環境（たとえば Java 7）でも同じ処理を実装する場合でベストプラクティスが異なる．

• 最新の設計ベストプラクティス

言語固有の文化も時間とともに変化することがある．こうしたベストプラクティスは開発者コミュニティで議論されることで変化していくこともある．たとえば，ひと昔前はオブジェクト指向が持て囃されたが，関数型言語の人気の高まりとも関連して過剰なオブジェクト指向設計は現在では忌避される傾向にある．また，依存性の注入（Dependency Injection）あるいは制御の反転（Inverse of Control）と呼ばれるプログラムの制御方法などはある時期から開発者コミュニティの中で積極的に取り入れられるようになった[☆2]．もちろん，こうした現在のベストプラクティスも時間とともに変化していくことがあることに注意されたい．5 年前に自分で書いたプログラムを見返すと，今ではどこか陳腐化していて再設計を施したいと思った経験のあるプログラマ諸氏は多いだろう．

何を写経するか

基本的には，業務に関連する興味のある分野のソフトウェアのコードを写経することをお勧めするが，いきなり言われても何を写経したしたらよいか分からないという読者もいるかもしれない．そういった方には，プログラミング言語固有のコレクションライブラリの一部を写経する，そして拡張してみることをお勧めする．コレクションライブラリとは，配列，リスト，集合，連想配列，キューなどのデータ構造とアルゴリズムに関するライブラリである．言語によってはリングバッファ（Ring Buffer）やスタック（Stack），両端キュー（Double-ended Queue）の実装が標準ライブラリには存在しないこともあるだろう．また，既存のリストの実装に性能や機能面で不足があるかもしれない．こうしたデータ構造を既存のオープンソースライブラリを参考に実装してみることでコーディング能力を培うことができ，また言語特有の慣習についても学ぶことができるだろう．

☆2 Martin Fowler 氏の記事がきっかけの 1 つであった．
https://martinfowler.com/articles/injection.html

あるいは，デバッグ用のロギングライブラリはファイル I/O や排他制御も絡むため，応用編として取り組んでもよいかもしれない．各言語で設計思想の異なる複数のロギングライブラリが存在するだろう．それらの設計の良し悪しをコードリーディングおよび利用例の写経により，批評できるようにするだけでも良い訓練となる．

継続的学習のススメ

最後となるがプログラマとして最も重要だと思うのが，学習を継続的に行い，知識を常に最新のものにアップデートしていくことである．

筆者が最初に企業に就職した 2004 年当時，職場でそれまで使われていたバージョン管理システムは CVS であったが変遷期ですぐに Subversion に置き換わった．Subversion は当時としては優れたシステムであったが，今は Subversion も陳腐化し，Git がその地位を奪った．当時はグリッドコンピューティングが全盛でクラウドコンピューティングはまだ登場していなかった．仮想化技術も Xen，KVM から Docker によるコンテナ仮想化へとトレンドが急速に変わっている．GoF のデザインパターン☆3 は当時のプログラマならばほとんどの方が読んでいたが，23 個のデザインパターンは今でも頻繁に利用されるものとほとんど利用されずに淘汰されるものに分かれている．

このように，IT 業界の変化の早さは犬の成長が人と比べて速いことに例えて**ドッグイヤー**とも評される．もちろん変わらないものも存在するが，変化への適応なしに生き残れない分野であるという認識を持ち，継続的に新しい知識を取り入れる姿勢を持つことが重要である．世の中には「プログラマ 35 歳定年説」という俗説も存在するが，私の身の回りには 40 歳を優に超える第一線のプログラマも多数存在するので，これは間違いである．「自ら変化できなければプログラマの仕事は続けられない」，と言えばより正確だろうか．

末筆ながら本稿のメッセージがこれから業務でプログラミングに携わる会員諸氏の何かの参考になれば幸いである．

(2019 年 2 月 12 日受付)

☆3 Gang of Four と呼ばれる 4 名の共著による書籍『オブジェクト指向における再利用のためのデザインパターン』でカタログ化された設計パターン．

■**油井　誠（正会員）** myui@apache.org

トレジャーデータ（株）プリンシパルエンジニア．2009 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．機械学習の研究開発およびそのサービス化に従事．Apache Hivemall の開発を主導．2003 年未踏ソフトウェア創造事業未踏ユース スーパークリエータ．