第 27 章　プログラムの作成

個人的な話

この原稿には極力個人的な話は書かないようにしてきたのだけれど、ここだけは個人的な話から始めることを許していただきたい。

私は大学を卒業してある金融機関の情報システム部門に勤務するようになった 22 歳から 40 歳まで、その会社でプログラムを書いていた。最後に書いたプログラムは、私を含む数人で要件定義を行った情報システムの移行のためのファイル切り替えプログラムだった。

当時私は親会社の情報システム部門で仕事をしていて、要件定義とシステムテストの実施が表向きの仕事だった。プログラムの設計／開発や開発プロジェクトの管理は、情報子会社の技術者の仕事だった。

会社と会社の関係から、私がプログラミングすることは禁止されていた。私が書いたプログラムにバグがあった場合、子会社として責任の取りようがないというのが表向きの理由だったように思う。しかし私はプログラミングが大好きで、子会社のプロジェクト管理者もそのことをよくしっていて、こっそりと私に切り替えプログラムを書く機会を用意してくれた。私はその仕事を、大いに楽しんだ。

会社を退職して大学の教員になり、大学ではプログラム実習の科目で COBOL と C 言語を学生に教えた。それまで私は両方ともその言語でプログラムを作ったことがなかったので、学生に教える前に私自身がその言語の勉強をすることになった。

さらに日本情報システム・ユーザー協会（JUAS）で「企業 IT 動向調査」の仕事に携わるようになって、またプログラミングの機会を持つことができるようになった。アンケートの回答結果のデータを Excel のマクロ（VBA）で分析して、結果をグラフや表で表示するプログラムの作成である。この仕事は、70 歳ぐらいまで続いた。結果として私は、断続的にではあるが、50 年近くプログラムを書き続けていたことになる。

この 50 年間に使ったプログラム言語には、以下のものがある。

- USSC（ユニバック（現ユニシス）社製の中型コンピュータ）のマシンコード
- USSC のアセンブラ（S4）
- FORTRAN Ⅱ（USSC）
- UNIVAC Ⅲ（ユニバック社製の大型コンピュータ）のアセンブラ
- IBM OS／360 のアセンブラ
- PL／ I
- APL
- SmallTalk
- COBOL
- C 言語
- Visual Basic
- Java
- VBA（Excel のマクロ）

私が入社したとき、その会社にはまだ UNIVAC 120 というエクスターナル・プログラミン

グ方式[1]のコンピュータが動いていた。そのコンピュータのプログラミングを作成する機会を持つこともできたが、私は挑戦しなかった。今思えば、その経験もしておくべきだった。

私が経験したプログラム言語の中に、SmallTalk と Java の 2 つのオブジェクト指向言語が含まれている。しかし私はそれらを、言語仕様を勉強する目的で使っただけであり、仕事としてそれらの言語でプログラミングしたわけではない。

良いプログラムとは

プログラムとは、「プログラム言語」という書き言葉だけの特殊な言語で書かれた、「コンピュータに対する命令の集まり」である。プログラムには大小あるが、数行程度というようなものは少なく、「文章」として延々と続いていることが多い。

仮にプログラムを文章と見なすなら、それでは「良い文章」とはどういうものだろうか。読みやすく、理解しやすく、理路整然としている、などというのが 1 つの「要件」であろう。プログラムも全く同じで、読みやすく、理解しやすく、理路整然としていることが「良いプログラム」であることの必要要件である。

この原稿の多くのところで、「きっとこういうようになっているだろう」と多くの人が考えるようにするのが良い、と書いてきた。例えば設計でのモジュール分割なら、多くの人が「こういうようにモジュール分割がなされているに違いない」と考えるようにモジュールを分割するのが良い、と私は書いた[2]。プログラムも同じで、プログラム設計書を見て、多くの人が「こういうようにプログラミングされているだろう」と考えるようにプログラムを作るのが良い。

それには、他の人が書いたプログラムをしっかりと読みこむ必要がある。

他の人のプログラムを読む

あなたが所属する開発組織がプログラム・インスペクションを常時実施しているなら、他の人が書いたプログラムをしっかりと読む機会を仕事として持つことができる[3]。仮にプログラム・インスペクションが行われていないなら、積極的に他の人が書いたプログラムを読みこむ機会を持ってほしい。

そして「良い」と考えた所はしっかりと自分のプログラムに取り込み、「良くない」と考えた所はなぜ「良くない」のかを考えて、自分のプログラムがそうならないように注意してほしい。

これまでいくつかのプログラムを書いてきた人はすでに気がついているだろうが、プログラムにはパターンがある。ある種類の処理を行う場合にはこのパターン、別の種類の処理をする場合にはあのパターンというように、多くのパターンを自分のものにして使い分けているプログラマは強い。

全く新しい種類のプログラムを初めて書くときにはそのパターンを持っていないので、始めからコンピュータにどう処理させるかを考えなければならない。しかしすでに持っているパターンを適用し、処理の内容によってそれを一部変更するだけなら早いし確実だし、ある意味で

---

1 エクスターナル・プログラミング方式とは、多くの穴の開いたパネルにワイヤーを使ってその穴同士を結んでプログラミングする方式を言う。コンピュータを稼働させる時は、そのパネルをコンピュータの所定の場所にセットする。

2 プログラムの設計については、第 26 章で述べた。

3 レビューについては、すでに第 18 章で論じた。

楽である。私の 18 年に及ぶ会社でのプログラマとしての期間の中最初の 1 年少々と、オンライン・システムを導入することになってそれに従事することになった直後の 1 年間ぐらいはそれぞれのパターンを身につける期間だった。しかしそれ以外の期間は、習得したパターンを活用する期間だったといえる。

構造化プログラミングがオブジェクト指向プログラミングに変わったときに、構造化時代に身につけたパターンがオブジェクト指向でも活用できるかどうかを、私はしらない。しかい構造化プログラミングに限れば、プログラム言語が変わっても、ある言語で使ったパターンを別のプログラム言語で使用することができる。プログラム言語はそのパターンを、コンピュータで実行できる形に記述する手段に過ぎないからである。

人のプログラムを読むことは、このパターンの種類を広げることでも役立つ。積極的に、人のプログラムを読む機会を持ってほしい。

しっかりとしたコーディング・ルールを持ち、それに従うこと

開発組織は、まずしっかりとした「コーディング・ルール」を持たなければならない。そしてその組織のためにプログラムを書く人は、その開発組織に所属する技術者も外部から開発に参画している技術者も、しっかりとそのルールに従ってプログラミングすることが必須である。

プログラム・インスペクションを実施しているなら、当然処理するべき内容がしっかりとプログラムに書き込まれているかをチェックすることが第一である。しかしこの時に、インスペクタの誰か一人はこのコーディング・ルールに違反していないかどうかをチェックすることに責任を持つ人がいても良い。

アジャイル開発の 1 つであるエクストリーム・プログラミングで行う作業の中に、「リファクタリング」と呼ばれるものがある[4]。エクストリーム・プログラミングでは、設計書やプログラムなどの成果物はグループ全員の共同資産という考えをとっている。そしてもしある人が作成した設計書やプログラムを別の人が見て良くないと思えば、当初作成した人に断らずに自由に改訂、あるいは再作成しても良いとされている。

リファクタリングを適切に行うためにエクストリーム・プログラミングではたいへん厳格なコーディング・ルールを持つことになっているが、このルールに違反しているプログラムは即リファクタリングの対象になってしまう。

プログラムは、本来的にはコンピュータのために書くものである。しかしそれを読むのは、コンピュータだけではない。開発段階だけでなく保守段階も含めて、多くの人が読むものであることを想定して、プログラムを作成しなければならない。良いコーディング・ルールを持ち、それに従ってプログラムを記述することは、プログラムを読みやすく、理解しやすくするための重要な手段である。

コメントはどう入れるか

本来的にプログラムは、プログラム言語で書かれたソース・プログラムを読むだけで内容が理解できるものでなければならない。コメントとは、プログラム言語だけでは充分には理解できないと思われる場合に、自然言語（日本語、英語、など）などで説明を付け加えるものであ

[4] アジャイル開発に浮いては、第 14 章で記した。

る。従って本来なら、コメントはそれほど必要とされないはずである。

スティーブ・マコネル（Steve McConnel）はその著書で、良いコメントの入れ方について 1 つ提言している[MCC05]。具体的には、以下のように作業をしてプログラミングの作業を終了させる。

- 設計とそのチェックが終了したら、まず擬似コードを使って設計内容をプログラムに落とす。この場合に使用する擬似コードはプログラム言語に近いものではなく、日本語など自然言語に近いものが望ましい。
- この擬似コードによるプログラムをしっかりとチェックし、そのレベルで完成させる。
- この擬似コードを全てコメントとして位置づけ、本物のプログラム言語を使用してプログラムを完成させる。
- そのプログラムをテストする。

テストの結果や保守でソース・プログラムに手を入れなければならなくなったとき、コメントにも気を遣って手を入れなければならない。プログラムの内容とコメントで記述されていることが乖離することがないよう、気をつけたい。

上で述べた法だと、コメントとプログラムの内容が乖離することは避けられる。良い方法と、評価できる。

超高速開発との関係

「超高速開発」と呼ばれる、要件定義といくつかの指示を入力すれば情報システムが作成できる開発方法が広がろうとしている[5]。プログラムを作成しなくても情報システムが開発できる訳で、ある意味で画期的な方法といえる。

超高速開発では、「普通の」プログラムは作成する必要がなくなる。しかしこれで、世界中が全くプログラムレスになるわけではない。なぜなら、全ての状況で超高速開発を行う仕組みが構築される訳ではないからである。例えば、たいへん先進的なコンピュータが新たに開発されたとして、そのコンピュータのためのプログラムは、最初は超高速開発の方式では開発されず、これまでと同様人が作成しなければならないだろう。

また個人的な話になるが、私は超高速開発の時代になっても全てそれに依存することはせず、機会を見つけて Excel のマクロ（VBA）あたりを使って、自分でプログラムを作り続けたいと念願している。

キィワード

プログラム言語、プログラム・インスペクション、パターン、コーディング・ルール、リファクタリング、コメント、擬似コード、超高速開発

人名

スティーブ・マコネル（Steve McConnel）

参考文献とリンク先

[5]「超高速開発」については、次の第 28 章で議論する。

[MCC05] Steve McConnel 著、（株）クイーブ訳、「Code Complete 第 2 版」、日経 BP 社、2014 年.
この本の原書は、以下のものである。
Steve McConnel,” Code Complete, Second Edition, “Microsoft Corporation, 2005.

（2016 年（平成 28 年）5 月 12 日　新規作成）