［フレッシュマンに向けたプログラミングのススメ］ 基般

8 サバイバルツールとしてのプログラミング
―プログラミング技術は一生使える基礎技能―

及川卓也 | Tably（株）

プログラマ以外にとってのプログラミング

フレッシュマンにプログラミングを勧める話を書いてほしいという依頼を引き受けたが，実は筆者は現在は製品に使われるプログラムは書いていない．以前はプログラマとして収入を得ていたが，その後，プロダクトマネージャやエンジニアリングマネージャという職種に異動し，プログラムを書く以外でソフトウェア開発にかかわるキャリアを歩んでいる．

最初に就職した会社でプログラマとして働いた後の2社目でプロダクトマネージャという職種に異動した．プロダクトマネージャは企画や設計などより広い範囲で開発に携わる職種だ．企業の最高意思決定者であるCEO（Chief Executive Officer）をなぞらえ「Mini CEO」とも呼ばれ，プログラマやデザイナなどの狭義の開発チームだけでなく，マーケティングや広報，営業や法務など，製品を世の中に出し，そして使ってもらえるようにするために必要なことを行うすべての部署の人たちを束ねる役割だ．

プロダクトマネージャとなってからは，製品に含めるためのプログラミングはしなくなったが，それでもPoC（Proof of Concept）としてのプログラミングは行った．書いたプログラムをプログラマに見せ，動作を説明した．最終的には，彼らがそれを製品に組み込んだのだが，筆者が書いたものがほぼそのまま採用されたものもあった．また，担当していた製品がOSだったため，開発している機能は直接エンドユーザを対象としていないことも多かった．API（Application Programming Interface）を開発し，社内外のプログラマにそのAPIを使ってエンドユーザ向けのソフトウェアを作ってもらう．そのような場合，APIの設計や検証には当然プログラミングが必要となる．自らがユーザとなることで企画や設計が初めて可能となった．

3社目となる会社ではエンジニアリングマネージャを経験した．エンジニアリングマネージャはプログラマが所属する組織をマネジメントする役割だ．エンジニアリングマネージャ自身もプログラミングをすることもあれば，ほかのプログラマが書いたコードをレビューすることもある．このコードレビューという行為はコードが意図した通りに動作するか，適切にエラー処理がされているか，より適切な処理方法はないかなどをコードを書いた本人以外の第三者によるレビューを通じて行うものだ．ただ，組織が大きくなると，直接自分がプログラミングする割合は減る．それでも自分の部下であるプログラマに指示を出し，その仕事を評価するためには，プログラミングの知識は必須だ．

このように，プロダクトマネージャやエンジニアリングマネージャとして働くようになってから，製品向けのプログラムは書かなくなったが，それでも自分や組織のためのプログラミングは続けている．

それは，仕事上必要なこともあるが，それに加えて，「人間がやる仕事ではない」と思えるものを自動化するためだ．たとえば，部下と1対1での定例の打合せをする際に，彼や彼女が書いたこの1週間のソースコードを見たいとする．現代では，書かれたコードはコー

ドレポジトリ（ソースコードがその変更履歴とともに記録されている保管場所．あるコードから派生するコードを作成することや特定の版のコードに戻ることも可能だ）に保存されていることがほとんどであり，その変更履歴も見ることができる．当然，打合せではそれを見たい．また，週報やコードレビューのやりとりなども参考にしたい．さらに言うと，以前の情報も時系列で見たい．これらをそれぞれのツールを開くことで参照してもよいが，それは思いのほか煩雑な作業であり，とても人がわざわざ貴重な時間を使って行うこととは思えない．そこで筆者はこれらを自動的にダッシュボードで見えるようなツールを自作していた．それにより手間はかなり少なくなった．マネージャとしての仕事は部下との対話であり，そこで適切なフィードバックを与えることであって，手間ひまかけて各種ツールを開き，そこからデータを取り出すことではない．それは一瞬にして目の前に表示されることが理想なのだ．理想だったら，その理想を実現すればよい．それを可能にするのがプログラミングだ．このように，製品に採用されるものではなくとも，自分以外が使うことがなくても，プログラミングはとても有効だ．「人間がやる仕事ではない」ことをやらずに済ませられるからだ．

このように，製品レベルのプログラミングは行わなくなった今であっても，プログラミング技術を持っていることがプロフェッショナルとして仕事をする上での武器となっている．

プログラミングの自動化と高度化の狭間で

昨今の AI ブームを受け，プログラミング作業はやがて自動化され，人が行うことはなくなると言う人がいる．

筆者が社会人になった 30 年前から同じことは言われていたが，そんな未来はいまだに来ていない．

当時と比べると，ソフトウェア開発ツールは劇的に進化した．IDE（Integrated Development Environment）と呼ばれるツールはエディタから進化し，今では作るものに合わせてコードスニペット（プログラミングコードの断片）は自動挿入してくれるし，プログラミング言語やライブラリなどに応じて，コードの提案もしてくれる（自動補完機能）．その意味では「自動化」は確かにされた．しかし，その分，ソフトウェアへの要求は高度化し，複雑化している．

たとえば，ウィンドウ操作を行うウィンドウプログラミング．基本はウィンドウオブジェクトに対して発せられたウィンドウメッセージを処理するメッセージハンドリングだ．かつては，そのためのメッセージループを自分で用意し，飛んできたメッセージに応じて処理を書いた．しかし，今ではそのような処理は不要だ．イベントが発生した処理だけを記述すればよい．その分，ウィンドウシステムでできることも増え，自動化され楽になった分，ソフトウェアに要求されることも増えた．言うならば，いたちごっこの状態だ．

自動化は「人間がやる仕事ではない」部分から行われ，まだ人間による創意工夫が必要な部分にまで対応できていない．結果，人がやる仕事は決してなくならなかった．

AI は確かに自動化をさらに進めるだろう．マイクロソフトの VisualStudio に搭載された IntelliCode は GitHub のパブリックレポジトリのソースコードを学習し，よりアグレッシブにプログラミングコードを挿入する．しかし，まだ完全自動化には至っていないし，それがいつ実現されるかも分からない．いつ来るか分からない未来に期待するよりも，今必要とされていることを実現できる技術を身につける．これがプロフェッショナルとして活躍し続けるための考えだろう．

プログラミング的思考の習得

2020 年から小学校でのプログラミング教育が始まる．これはプログラミングそのものを教えるのではなく，プログラミング的思考を教えるものだ．

この「プログラミング的思考」は，新社会人がプログラミングをする上でも大事だ．説明したように，ソフトウェア開発ツールが進化し，多くが自動生成されるようになっ

たとしても，プログラミングする上で学んだプログラミング的思考はシステム全体の設計などで活用できる．

今日のモダンなシステムは機能を分割し，独立性を高めた別コンポーネントとして開発を進めるマイクロサービスを用いるようになってきているが，機能をどのようにマイクロサービスとして分離していくか，各マイクロサービス間の通信負荷などを考えた上で，それぞれの粒度はどうあるべきかなど，プログラミングでの知識や経験が活きる．ほかにも，最近ではエンジニアの組織設計にもプログラミングのテクニックを流用しようという考えもあるほどだ．

このプログラミング的思考は情報社会を生きる上での必須の思考術になってきていると言っても過言ではない．

n 次元空間における最大化を目指す

では，これから社会人になる皆さんは何を学んでいけばよいだろう．

人の成長の考えとして，T 字型人間を目指すのがよいと言われる．これは，広く浅く基礎素養として知っておくべき知識と深掘りし専門性を高める知識を 1 つ持つという考えだ．前者を T という文字の横棒に，後者を縦棒になぞらえている．この T 字型の能力を持つ情報処理技術者の例としては，ネットワーク技術やセキュリティ技術など情報処理に関係する技術を一通りは理解した上で，データ解析のための Python などの言語を駆使する能力を持つような技術者だ．フレッシュマンはまずはこの T 字型人間を目指すとよいだろう．

プログラミング言語の習得は，T 字型の考えに沿って，1 つの言語を習得した後に，さらに 2 つ目や 3 つ目の言語を習得するのが一般的だ．では，どの言語を習得すればよいだろうか．そのヒントは自然言語にある．

自然言語，すなわち我々が使う日本語や英語という言語には言語間の距離があると言われる．言語間の距離は言語の類似性を表すもので，母語からの距離が近い言語ほど習得が楽だ．日本語と韓国語，英語とドイツ語は近い距離の言語だと言われる．

プログラミング言語でも言語には類似性がある．それはプログラミングパラダイムと呼ばれる．C 言語を代表とする手続き型言語や Lisp や Scala などの関数型言語，そして現在主流となっているオブジェクト指向言語（クラスベースとプロトタイプベースに分類される）などがそれだ．現代のプログラミング言語はマルチパラダイムプログラミング言語と呼ばれ，2 つ以上のパラダイムに対応しているものもあるが，それでも普通は少数のパラダイムへの対応となる．

自然言語と同様に，同じパラダイムのプログラミング言語は理解しやすい．そのため，実務であるパラダイムの言語を習得したならば，意図的にそれとは異なるパラダイムの言語を習得し，2 つのパラダイムに将来のさらなる言語の習得の基軸となる言語を持つとよい．たとえば，オブジェクト指向言語として Java を習得し，関数型言語として Scala を習得すれば，将来もし Ruby を覚えなければならなくなったとしても，Java で理解したオブジェクト指向の考えが役に立つ．同じことは関数型言語にも言える．

これは自分の持つ能力を n 次元の空間において最大化することを意味する．イメージしやすいように 3 つの技能を身につけることを目的とした 3 次元空間を考えよう．原点を (0, 0, 0) とし，3 つの座標は x 座標が手続き型言語，y 座標がオブジェクト指向型言語，z 座標が関数型言語とする．単位は習熟度としよう．

ここで 1 つの言語を身につけたとする．その言語は純粋な手続き型言語であり，ほかのパラダイムは持たない．この状態では x 座標上に直線が引かれるだけとなる．もしここでその次に習得する言語が同じように純粋な手続き型言語だとしたら，能力として面になることはないが，関数型言語をパラダイムとして持つ言語を習得することで，能力が面を構成することとなる．同じように，第 3 の言語がオブジェクト指向型言語ならば，能力が立体を構成することとなる．いずれもその習熟度が高ければ能力の面や体積は増える．このように，能力開発はできるだけ異なる領域に足場を築くことも重要である．ここではプログラミング言語だけに

絞って説明したが，本来はこれを情報処理技術一般に広げて考える方がよい．機械学習に詳しい技術者がプロセッサーのアーキテクチャにも詳しくなったならば，きわめて希少性の高い技術者となる．

Apple の故 Steve Jobs は 2005 年のスタンフォード大学の卒業式典で有名なコネクティング・ドッツという話をした．思いもかけなかった形で点と点は将来つながるという考えを，大学で学んだカリグラフィの知識が 10 年後に Macintosh を開発する際に役立ったという自らのエピソードを通じて紹介した．プログラミング言語や情報処理技術にもこの考えは通ずる．

先程は意図的に異なるパラダイムの言語を習得することを勧めたが，結果として以前習得したものが活きてくることも多い．Jobs のコネクティング・ドッツのエピソードはその代表的な例だろう．これは，スタンフォード大学の John D. Krumboltz 教授により計画的偶発性理論として提唱されている．この理論は「慎重に立てた計画以上に，予想外の出来事や偶然の出来事がキャリアに影響を与える」という考えに基づくものだ．教授はこのような偶発性を起こすには次のような人の特性が必要となると論じている．

- 好奇心
- 持続性
- 柔軟性
- 楽観性
- 冒険心

計画性を持って技術の習得を進める場合も，偶発性を重んじる場合でも，この 6 つの特性は重要だ．興味という人間の持つ最大のモチベーションをいかに持つか，維持し続けるか，これが技術者として成長を続ける鍵となるだろう．

(2019 年 2 月 10 日受付)

■及川卓也（正会員） oikawa@tably.rocks

大学卒業後，外資系企業でソフトウェアエンジニアとして勤務後，プロダクトマネージャ，エンジニアリングマネージャとしてソフトウェア開発に携わる．現在，Tably（株）にて，テクノロジーによる課題解決と価値創造を行う．