# 第1章　ソフトウェアとソフトウェアエンジニアリング

世界で最も有名なFPS（First-person shooter）ゲームの最新ビルドの紹介を終えたとき、その若き開発者は笑った。

「あなたはゲームをする人ではないですね」彼は訊ねた。

私は微笑みながら聞き返した。「どうしてそう思ったのですか」

半ズボンとTシャツ姿の彼は、仲間内では当たり前というように神経質な雰囲気を示し、足をピストンのように上下に動かした。

「もし、あなたがゲームをする人であれば」彼は言った。

「もっと興奮しているはずだ。次世代のプロダクトに触れて最高潮の気分に達しているはず。他の顧客であれば一撃でやられてしまうほどの魅力が…おっと、FPSにかけたシャレではありませんけど」

我々はこの惑星上で最も成功したといえるゲームの開発者がいる開発室で座っていた。何年かが経過し、彼がデモを行った新世代ゲームの販売数は5000万本を超え、10億ドル以上の収益を記録している。

「それでは、このバージョンはいつ市場に出るのですか」私は質問した。

彼は肩をすくめた。「5カ月以内でしょう。まだ多くの仕事が残っております。」

彼はゲームのプレイのみでは無く、300万行を超えるコードを含むAI（人工知能）のアプリケーションに対する責任があった。

「あなたたちはソフトウェアエンジニアリングの技術を何か使っているのですか」彼が笑って首を振るのを半分想像しながら、私は質問をした。

少しの間考え込み、そして彼はゆっくりとうなずいた。「我々の必要に応じてそれを取り入れていますね。技術は確実に使っています」

「どの部分でしょうか」私は探究心から訊ねた。

「我々の問題は、クリエイティブを要件という形式へ翻訳する段階で発生していることが多いのです」

「クリエイティブ、ですか」私は話をさえぎって確認した。

「そうですね。ストーリ、キャラクターを考える人、すべての人がゲームをヒットさせようとしています。我々は、彼らが出すアイデア、生み出すものをゲーム開発の技術的な要件にする必要があるのです。」

「その要件が確立された後はどうなるのですか」

彼は肩をすくめた。「ゲームの前バージョンのアーキテクチャをベースに拡張し、新しいプロダクトを作り上げなくてはなりません。コードは要件から作り上げ、テストを行い、毎日ビルドを通さなければなりません。あなたの本では他にも多くのことが書かれておりますよね」

「私の本を知っているのですか」正直驚いて言った。

「もちろんです。学校でも使われていました。たくさんのことが書かれていましたよね」

「あなたの同僚の方ともここで話しましたが、私の本に対して懐疑的のようでした」

彼は眉をひそめて言った。「見てください。我々はIT部門や航空宇宙会社ではありません。そのため、あなたが提唱している内容はカスタマイズして使わなければならないのです。しかし、現場と呼ばれる場所ではみな同じでしょう。我々は高い品質のプロダクトを生み出す必要があります。そして、高い品質を繰り返し生み出す方法は、自分達に適したソフトウェアエンジニアリング技術のサブセットを扱うことです」

「それでは、そのサブセットを今後の時代の経過にあわせてどのように変化させてくつもりですか」

未来への考えをめぐらしているらしく、彼は少し考えた後、話した。「ゲームの開発はより大きく、複雑へとなっていくことは確かです。そして、競合があらわれるたびに開発期間も短くなっていくでしょう。ゆっくりとですが、ゲーム業界も開発の原理、規律に従うこととなるでしょうね。そうしなければ、我々の開発も終わりを迎えてしまうでしょう」

コンピュータソフトウェアは世界で最も重要な技術のひとつであり続けている。そして、社会学でいうところの「予期せぬ結果の法則（the law of unintended consqusences）」としてきわめて重要な実例でもある。60年前にソフトウェアがビジネスや科学、工学にとって不可欠なものとなると誰が予測しただろうか。遺伝子工学やナノテクノロジーのような新しい技術の創造、通信のような既存の技術の拡張、医療のような古い技術に対しても抜本的な変化が起きている。パーソナルコンピュータの革新においてソフトウェアは陰の推進力であり、顧客はモバイル機器を用いてソフトウェアアプリケーションを入手可能である。ソフトウェアは少しずつプロダクトから「オンデマンドに」ブラウザを通じていつでも機能を提供できるサービスへ進化し、ソフトウェア企業は工業時代の企業を超える影響を及ぼしている。また、ソフトウェアにより実現される広範囲にわたるネットワークは、図書館での文献調査、普段の買い物、政治的な対談、老若男女の日々の習慣まで発展させ変化させている。

ソフトウェアが重要になるにつれ、より簡単に、素早く、低価格で高い品質のプログラムをビルドおよびサポートするための技術が次々と生み出されている。これらの技術は、Webサイトの設計や実装のような特定のドメインに絞られているものもあれば、オブジェクト指向システムやアスペクト指向プログラミング（aspect-oriented programming）のような技術的なドメインに焦点を当てているものもある。また、Linux OS等の広範囲で活用されるベース技術までさまざまである。我々はあらゆるドメインに対応するソフトウェア技術を開発しなければならないが、その技術向上の見込みは将来的には小さい。その割に人々は今のところ、仕事、快適さ、安全性、娯楽、意思決定、まさに生活のそのものをソフトウェアに委ねるという賭けを行ってる。分の悪い賭けにならないことを祈ろう。

本書では、生活に関わるソフトウェアを正しく作るソフトウェア関係者が使うべきフレームワークを紹介する。フレームワークにはプロセスや手法のセット、ツール類を含み、それらを「ソフトウェアエンジニアリング」と呼ぶ。

21世紀において取り組むに値するソフトウェアを構築するためには、次の現実を認識しなければならない。

- ソフトウェアは、人々の生活のほぼすべてに深く入り込んでいる。とあるアプリケーションが提供する機能やフィーチャに対して興味をもつ人[*1]の数も劇的に増大している。ソフトウェアによる解決策が構築される前に問題を理解する労力を費やすべきである。

[*1] 本書ではこれらの人々を「ステークホルダ」と呼ぶ。

- 個人や企業、政府により求められる情報技術の要件は、年々複雑さを増し続けている。いまや大規模なチームでプログラムが作られている。予測可能であり、必要なものが揃えられ、コンピュータ環境

にて実装された高度なソフトウェアは、電子機器や医療機器、自動車まであらゆるものに組み込まれている。設計が中枢を担う活動となる。

- 個人や企業、政府において、日々の活動や統制に加えて、戦略的、戦術的な決定判断をソフトウェアに頼る状況が増えてきている。ソフトウェアが問題を起こしたならば、人々や企業が少し不便になることも有れば、致命的な結果にもなることもある。ソフトウェアは高品質であるべきだ。

- 特定のアプリケーションにて得られる価値が高まることにより、アプリケーションのユーザ数増加に加え寿命も伸びる。アプリケーションのユーザ数と使用時間の増加につれて、さらなる適用先と拡張の要求が発生する。ソフトウェアは保守可能であるべきだ。

これらの現実により「形式やドメインにかかわらず、ソフトウェアはエンジニアリングされるべきである」という結論が得られる。そして、本書「実践ソフトウェアエンジニアリング」へと繋がる。

## 1.1 ソフトウェアの本質

今日、ソフトウェアは二重の役割を担っている。ソフトウェアはプロダクトそのものであり、同時にプロダクトを提供する手段となる。プロダクトとしてのソフトウェアは、コンピュータのハードウェアに組み込まれる場合もあれば、手元のデバイスでアクセスできるネットワーク上にて実行される場合もある。いずれにせよソフトウェアは情報処理能力を提供する。これらのソフトウェアは、モバイル機器、デスクトップPC、クラウド、メインフレームコンピュータ、自律型マシンに搭載され、生産、マネジメント、取得、変更、変換、表示、送信のような情報の変換機となる。情報は単一のビットようにシンプルな場合もあれば、多くの独立した情報と現実世界を重ね合わせるAR（拡張現実）のような複雑な場合も存在する。情報処理能力を有するプロダクトを提供する手段としてのソフトウェアは、コンピュータ制御の基礎的な仕組みの提供（OS）、情報の伝達（ネットワーク）、そしてプログラムの作成および管理（ソフトウェアツールや開発環境）を行う。

ソフトウェアは、現在において最も重要な「情報」を取り扱うプロダクトを提供する。身の回りで使いやすいように個人データ（例：個人の銀行取引）を変換し、企業の競争力を高めるためにビジネス情報を取り扱い、世界中の情報ネットワーク（例：インターネット）へのアクセスを可能とする等、さまざまな形式で情報を取得する手段を提供している。また、ソフトウェアは個々のプライバシーを脅かす手段や、悪意ある犯罪的な行為の入口となってしまう場合もある。

ソフトウェアの役割は、ここ60年以上で大きく変化している。ハードウェアの性能は劇的な向上を遂げ、アーキテクチャは大きく変化し、メモリやストレージは圧倒的な大容量化となり、入出力は多様化したことにより、コンピュータシステムはより洗練され複雑なものになった。洗練と複雑化はシステムが成功している場合にはまばゆいばかりの結果を生み出すが、システムを構築、維持する開発者に対して大きな問題をつきつけている。

今日、先進工業国の経済活動において巨大なソフトウェア産業は重要な要素を担っている。初期に行われていたようなプログラマひとりで仕事ができる時代ではなく、特有の技術をもつソフトウェアスペシャリストが複数名集まったチームが複雑なアプリケーションの提供に必要とされる。それでも、最新のコンピュータシステムが構築される際に、プログラマひとりの時代と同じような疑問が相変わらず問われている[*2]。

[*2] ソフトウェアビジネスに対する良書[DeM95]にて、Tom DeMarcoが次のように反論している。「なぜソフトウェアのコストがそんなに高いのか」と疑問を投げるのではなく、「コストを低く抑えるためには何をすべきだったか」という疑問を持ち始めるべきだ。この答えが分かれば、ソフトウェア産業は今後も並外れた成功を続けることができるだろう。

- なぜ、ソフトウェアができ上がるまでにそんなに時間がかかるのか

- なぜ、開発のコストはそんなに高いのか
- なぜ、顧客にソフトウェアを引き渡す前に、すべてのバグを見つけられないのか
- なぜ、既存のプログラムの保守にそんなに時間と労力を費やすのか
- なぜ、ソフトウェアの開発中に進捗を測ることがそんなに難しいのか

こうした疑問を抱くということは、ソフトウェアの開発に関心があることを示している。だからこそ、ソフトウェアエンジニアリングプラクティスを採用する気になるのである。

### 1.1.1 ソフトウェアの定義

最近ではプロフェッショナルを含め多くの人が、自分はソフトウェアを理解していると考えている。しかし実際はどうであろうか。教科書的なソフトウェアの定義は次のとおりである。

1. 実行されることによって必要な特性、機能、性能を提供する命令語群（コンピュータプログラム）
2. プログラムが適切に情報を扱うことを可能とするデータ構造
3. プログラムの操作や使用法を記述した情報

単純であるが、これ以上の質問することもない完璧なものである。しかし、形式的な定義であり、ソフトウェアへの理解が少ない人に対して、理解を促すことにはならないだろう。ソフトウェアを理解するためには、ソフトウェアが人が生み出した他のものと異なる特徴箇所を明確にすることが重要である。ソフトウェアは、物理的というより論理的なシステム要素である。そのため、ソフトウェアはハードウェアと大きく異なり「ソフトウェアは劣化（wear out）しない」という重要な特性をもつ。

```
図1.1 ハードウェアの故障曲線
Failure rate 故障率
Time 時間
"Infant motality" 初期の故障
"Wear out" 劣化
```

ハードウェアの故障発生率と時間の関係を図1.1に示す。この関係性は「バスタブ曲線」とも呼ばれ、製造直後の初期においてハードウェアが高い故障発生率を示すことがわかる（これらは、設計や制作段階での欠陥がほとんどである）。その後、欠陥が修正されるにつれて故障発生率は安定した（理想としては十分に低い）値まで低下する。しかし、時間の経過により、ほこりや振動、想定しなかった使い方、温度変化等の環境による要因が蓄積することで故障発生率は再び上昇する。簡単にいうと、ハードウェアの「劣化」の始まりである。

```
図1.2 ソフトウェアの故障曲線
Failure rate 故障率
Time 時間
Change 変更
Actual curve 実際の曲線
Idealized curve 理想的な曲線
Increased failure rate due to side effects 変更の副作用による故障率の増加
```

ソフトウェアは、ハードウェアを劣化させるような環境的な要因の影響は受けない。理論的にはソフトウェアの故障発生率曲線は図1.2で示される「理想的な曲線」が描かれるはずである。初期段階では、プログラムの見つかっていない欠陥により故障率は高くなる。これらが修正されることで曲線は図で示されているよう

に平らな定常状態になる。この理想的な曲線は、実際のソフトウェアの故障モデルを大まかに単純化したものである。ここで言いたいことは、ソフトウェアは劣化はしないが「悪化（deteriorate）」はするということだ。

一見矛盾しているようだが、図1.2上における「実際の曲線」を用いて説明ができる。ソフトウェアはどの段階においても変更が発生する[*3]。変更によって新しいエラーが混入することもある。このエラーが図1.2の「実際の曲線」のように故障率を瞬間的に高めてしまうのである。故障率曲線が理想的な定常状態となる前に別の変更が要求されることで、故障率は再上昇してしまう。そしてゆっくりと故障率の下限レベルが上昇し始める。これが変更によるソフトウェアの悪化である。

[*3] 実際、開発開始の瞬間から最初のバージョンが納品されるまでの間に、さまざまなステークホルダにより変更が要求される。

ソフトウェアとハードウェアの違いを示すもう1つの劣化の側面を示してみよう。ハードウェアの部品が劣化した場合は、スペア部品によって交換できる。しかし、ソフトウェアにスペア部品はない。すべてのソフトウェアの故障は、設計過程におけるエラー、もしくは設計をコーディングをする過程におけるエラーによるものである。したがって、ソフトウェアの変更を含む保守作業はハードウェアの保守と比べてはるかに複雑である。

### 1.1.2 ソフトウェアアプリケーションのドメイン

今日、7つに大きく分類されたコンピュータソフトウェアのドメイン[訳注1]に対して、ソフトウェアエンジニアによる取り組みが行われている。

[訳注1] この分類は必ずしも1つのシステムが1つのドメインと対応するとは限らない。例えば第1章冒頭におけるゲームソフトについて。据え置き型ゲーム機では組込みソフトウェアやSDK等のシステムソフトウェア、ゲームアプリとしてはアプリケーションソフトウェアや場合によっては人工知能ソフトウェアまで幅広いドメインが関連する。ソーシャルゲームではWeb/モバイルアプリケーションも対象ドメインとなる。また、クライアント/サーバ型と呼ばれるソフトウェアは、過去はアプリケーションソフトウェアに分類されていたが、現在は多くがクラウド上で動作するWeb/モバイルアプリケーションへと変遷している。

#### システムソフトウェア

システムソフトウェアは他のプログラムを支援するプログラムを指す。コンパイラ、エディタ、ファイル管理ユーティリティといったシステムソフトウェアは、複雑だが確定的なデータ構造を扱う[*4]。OSの要素、ドライバ、ネットワークソフトウェア、通信のようなシステムアプリケーションは、主に非確定的なデータを処理する。

[*4] 入力、処理、出力のタイミングの順序が決まっており予測可能な場合、ソフトウェアは「確定的」となる。入力、処理、出力のタイミングが予測不可能な場合には「非確定的」となる。

#### アプリケーションソフトウェア

アプリケーションソフトウェアは特定のビジネスニーズに応えるスタンドアローンのプログラムから構成される。この分野のアプリケーションの目的は、業務あるいは技術的なデータを処理することによって、業務の遂行や経営的、技術的な意思決定をスピードアップすることである。

#### 科学技術計算ソフトウェア

「数値演算（numbrer crunching）」やデータサイエンスプログラムは、天文学から火山学、自動車の応力解析、軌道力学、CADによる設計、消費者行動分析、遺伝子解析、気象学まで広い範囲で使われる。

**組込みソフトウェア**

組込みソフトウェアはプロダクトもしくはシステムに書き込まれ、ユーザとのインタフェースやシステムの機能制御に使われる。電子レンジのキー入力制御のような限定的で特殊な機能や、自動車の燃料制御やダッシュボード表示、ブレーキシステム、ロボット制御等、非常に重要な機能や制御に用いられる。

**プロダクトラインソフトウェア**

プロダクトラインソフトウェアは、多くの異なる顧客が使用することを目的に設計された再利用可能なコンポーネントである。在庫管理製品等の限定的で特殊な市場、もしくは、一般顧客向け市場へ適用される。

**Web/モバイルアプリケーション**

ネットワークを中心としたソフトウェア分野は、多くの種類のアプリケーションへ広がっている。この分野には、ブラウザベースのアプリケーション、クラウドコンピューティングやSaaS、モバイル機器上で動作するソフトウェアが含まれる。

**人工知能ソフトウェア**

人工知能（AI）ソフトウェアは、計算では解くことが容易では無く単純には解析できない、複雑な問題を解決するため、ヒューリスティック[*5]な手法を用いる。この分野のアプリケーションは、ロボット工学や意思決定システム、画像や音声のパターン認識、マシンラーニング、数学の定理証明、ゲームAI等に用いられる。

[*5] ヒューリスティックな手法とは決して完ぺきとは言えないが、十分とみなせる実用的な手法もしくは、経験則を採用した問題解決のアプローチである。

世界中で何百万人というソフトウェアエンジニアがこれらの分類におけるソフトウェアプロジェクトにおいて労力を費やしている。新規システムが構築される場合もあるが、ほとんどは既存のアプリケーションが修正・適応・拡張される派生開発である。若きソフトウェアエンジニアが、自分の年よりも古くからあるプログラムに取り組むことも珍しいことではない。すでに紹介した各分野において、前世代のソフトウェアエンジニアが遺産を残している。前世代によって残された遺産は、将来のソフトウェアエンジニアの重荷とならないものであってほしい。

### 1.1.3 レガシーソフトウェア

多くのコンピュータプログラムは、前節で紹介した7つのアプリケーションドメインのどれかに分類される。これらは最先端のソフトウェアもあれば、古い（時にはかなり古い）プログラムでもある。

レガシーソフトウェア（Legacy Software）とも呼ばれる古きプログラムは、1960年代以降、注目および憂慮され続けている。Dayani-Fardらはレガシーソフトウェアを次のように記述している[Day99]。

> 「レガシーソフトウェアシステムは何十年も前に開発されたものの、ビジネス要求やプラットフォームの変化に見合うよう継続的に修正が続けられている。大きな組織において、保守のコストや成長へのリスクが伴うシステムの急増は頭痛の種である。」

これら修正の副作用として、レガシーソフトウェアには「乏しい品質[*6]」が見られることがある。レガシーシステムは時として、拡張性のない設計、複雑に絡み合ったコード、ドキュメントが貧疎もしくは存在しない、成功することのないテストケース結果、乏しい変更管理履歴のように数えるときりがないようなものを持っている。しかし、これらのシステムは「ビジネスの中核が機能するための支えであり、ビジネスのために必要不可欠なもの」なのである。我々は何ができるのだろうか。

[*6] 近年のソフトウェアエンジニアリングの考えに基づいて、このケースの品質を「乏しい」と判断している。ただ、近年のソフトウェアエンジニアの概念や原理は、レガシーソフトウェアが開発された時代にはあまり認識されていなかったため、やや不公平な基準である。

無難な回答のひとつは「何もしない」であろうか。少なくとも大きな変更が必要とされるまでは。レガシーソフトウェアがユーザのニーズに適合しており、信頼性のある動作をしているならば、壊れたり修正の必要もない。しかし、時間の経過につれて、次のような理由によりレガシーシステムに進化が発生する。

- ソフトウェアは新しい環境や技術のニーズを満たすよう適応しなければならない。
- ソフトウェアは新しいビジネス要求を実現するために強化されなければならない。
- ソフトウェアはより近代的なシステムやデータベースと相互運用するため拡張されなければならない。
- ソフトウェアは進化し続ける環境で実行可能となるよう、再構築されなければならない。

これらの進化が発生した場合、レガシーシステムは将来まで実行可能であり続けるためにリエンジニアリングされなければならない。現代のソフトウェアエンジニアリングのゴールは、「進化論に基づく（ソフトウェアシステムが変化し続ける状況に対応する）方法論を考え出す」ことである。新しいソフトウェアが古いシステムから構築され、すべては互換性を持ち、他システムと相互運用しなければならない[Day99]。

## 1.2 ソフトウェアエンジニアリングと規律

IEEEはソフトウェアエンジニアリングの定義[IEE17]を以下のように行っている。

> ソフトウェアエンジニアリング：ソフトウェアの開発、運用、保守に対するシステマティックで規律（discipline）ある、定量化できるアプローチの適用、すなわちソフトウェアに対するエンジニアリングの適用。

あるソフトウェア開発チームに適用された「システマティックで規律ある、定量化できる」アプローチは、他のチームにとってわずらわしい場合もある。規律は必要であるが、同時に適応性や俊敏さ（agility）も必要である。

```
図1.3 ソフトウェアエンジニアリングの階層
Tools　ツール
Methods　手法
Process　プロセス
A quality Focus　品質に焦点を合わせる
```

ソフトウェアエンジニアリングは階層的な技術で構成されている（図1.3）。ソフトウェアエンジニアリングを含むすべてのエンジニアリングアプローチは、品質に対する組織的な責任を持って実施されなければならない。総合的品質管理（TQM）やシックスシグマ（Six Sigma）のような、プロセスを常に改善し続けるという文化を育むという哲学[*7]を耳にしたことがあるだろう。こうした文化こそが、究極的にソフトウェアエン

ジニアリングアプローチをさらに効果的にしていく。品質に焦点を合わせること（quality focus）が、ソフトウェアエンジニアリングを支える基盤となる。

[*7] 品質マネジメントと関連するアプローチは本書の3部全般で取り扱う。

ソフトウェアエンジニアリングの土台はプロセス層である。ソフトウェアエンジニアリングプロセスは、技術の層を1つにまとめる役割を持ち、ソフトウェアが合理的かつ納期どおり開発できるようにする。ソフトウェアプロセスは、ソフトウェアエンジニアリングの技術を効果的に適用するためフレームワークを定義する。ソフトウェアプロセスによってソフトウェアプロジェクトのマネジメントの基礎が形作られ、技術的手法の適用、成果物（モデル、ドキュメント、データ、報告、書式）の作成、マイルストン設定、品質の確認、適切な変更管理が行われる。

ソフトウェアエンジニアリング手法は、ソフトウェアを構築するための技術的な方法を提供する。手法には、コミュニケーション、要求分析、モデル設計、プログラム実装、テスト、保守といった幅広いタスクが含まれる。ソフトウェアエンジニアリング手法は、モデリング活動やその他の技術を含み、各技術分野の根底となる一連の原則に基づいている。

ソフトウェアエンジニアリングツールは、プロセスや手法に全自動化もしくは半自動化の恩恵をもたらす。あるツールが作成した情報を別ツールで使えるよう統合したソフトウェア開発支援システムは、CASE（computer-aided software engineering）と呼ばれる。

## 1.3 ソフトウェアプロセス[訳注2]

プロセスとは、アクティビティ、アクション、そして何らかの成果物を生み出すタスクを含む。アクティビティは、ステークホルダとのコミュニケーションのような上位の目的を成し遂げるために取り組むことを意味し、アプリケーションドメインやプロジェクトの規模、その活動の複雑さ、厳密さの度合いにかかわらず実施される。アクションは、主要な成果物（例：アーキテクチャ設計というアクションでは、アーキテクチャモデルが成果物）を生み出す複数のタスクをもつ。タスクは成果を生み出すために細かく分割され、明確に定義された目標（例：ユニットテストの実施）をもつ活動である。

[訳注2] 本章に出てくるプロセスフレームワーク、フレームワークアクティビティ、包括的なアクティビティ、アクション、タスクという用語については図2.1の構造を参照することを推奨する。

ソフトウェアエンジニアリングのコンテキストにおいて、プロセスはソフトウェアをビルドする方法を厳格に規定するものではない。むしろ、業務を行うソフトウェアチームが適切なアクションとタスクを選択する適応行動である。プロセスは、開発の支援および成果物を使用する人々が満足する十分な品質をもつソフトウェアを常に納期内に届けるためのものである。

### 1.3.1 プロセスフレームワークとフレームワークアクティビティ

プロセスフレームワークは、いくつかのフレームワークアクティビティによって定義され、ソフトウェアエンジニアリングプロセスを表現する。フレームワークアクティビティは、ソフトウェアプロジェクトの規模や複雑さにかかわらず適用される。さらに、プロセスフレームワークは、ソフトウェアプロセス全体に対して横断的に適用される包括的なアクティビティ（umbrella activities）をもつ。プロセスフレームワークは、次の5つの一般的なフレームワークアクティビティ[訳注3]から構成される。

[訳注3] 本書全般では、5つの一般的なフレームワークアクティビティにて各プロセスモデルを表現しており、繰り返し登場する。これらアクティビティは一般的で使いやすいが、必ずしも記載のアクティビティしか使ってはいけないわけではない。チームで名前やタスク等を定めた独自アクティビティを定義してもよい。

#### コミュニケーション（communication）アクティビティ

技術的な業務を開始する前に、顧客やステークホルダ[*8]とのコミュニケーションおよび協業は極めて重要である。ステークホルダのプロジェクトにおける目的を理解することや、要望を聞くことは、ソフトウェアの機能やフィーチャを決めるために役に立つ。

[*8] ステークホルダ（stakeholder）は、プロジェクト成功時に利害を得る人を指す。ビジネスマネージャ、エンドユーザ、ソフトウェアエンジニア、サポートスタッフ等である。「ステークホルダは大きく鋭い棒（stake）をもつ人である。ステークホルダの面倒をみなければ、その棒がどこに振り下ろされるかわかるだろう」とRob Thomsettは冗談めかして言っている。

#### 計画策定（planning）アクティビティ

地図が存在していれば、複雑な旅程も単純になる。ソフトウェアのプロジェクトは、複雑な旅程に相当する。計画策定のアクティビティではチームをガイドする「地図」を作り出す。ソフトウェアプロジェクト計画と呼ばれる地図には、ソフトウェアエンジニアリング業務に関する技術的なタスク、発生しそうなリスク、必要なリソース、作成すべき成果物、スケジュールを記述する。

#### モデリング（modeling）アクティビティ

庭師や橋梁架設者、航空機技術者、大工、建築家はいずれもモデルを用いて毎日の業務をこなしている。スケッチをすることで、物ごとの全体像を理解する。構造的にはどうか、要素部分をどのようにつなぎ合わせるか、等について考察する。より深く問題を理解し、どのように問題を解くかを考えるために、必要に応じて対象を深く詳細まで掘り下げ、描いたスケッチを書き直す。同様にソフトウェアエンジニアは、ソフトウェアの要求、実現のための設計をより深く理解するためにモデルを作成する。

#### 構築（construction）アクティビティ

設計されたものはビルドされなければならない。構築アクティビティには、手動あるいは自動でのコード生成、コード内の欠陥を発見するテストが含まれる。

#### デプロイ（deployment）アクティビティ

完全な製品もしくは部分的ものだとしても、ソフトウェアは顧客に納品され、成果物が評価され、評価に応じたフィードバックが行われる。

上記5つの一般的なフレームワークアクティビティは、小さなプログラム開発、Webアプリケーションの構築、大規模で複雑なコンピューターシステムの設計、いずれにも使用できる。ソフトウェアプロセスの詳細は、それぞれのケースでは大きく異なるが、個々のフレームワークアクティビティは同じである。

多くのソフトウェアプロジェクトでは、進捗に応じてフレームワークアクティビティ（コミュニケーション、計画策定、モデリング、構築、デプロイ）が反復的に実施される。それぞれの反復[訳注4]が行われることにより、ステークホルダにソフトウェアの機能やフィーチャを提供するソフトウェアがインクリメントされる。インクリメントによる成長につれて、ソフトウェアはより複雑になっていく。

[訳注4] この反復は、繰り返し実施されるソフトウェア開発サイクルにおいて、リリースを一区切りとする一連の活動（1回分のサイクル）を意味する「イテレーション」という言葉で表現される場合もある。

### 1.3.2 包括的なアクティビティ（Umbrella Activities）

ソフトウェアエンジニアリングプロセスにおけるフレームワークアクティビティは、いくつかの包括的なアクティビティにより補完される。一般的には、包括的なアクティビティはソフトウェアプロジェクト全体において適用され、ソフトウェアチームが進捗、品質、変更、リスクを管理するために役立つ。よく見られる包括的なアクティビティを次に示す。

**ソフトウェアプロジェクトの追跡と管理**

ソフトウェアチームはプロジェクト計画に対して進捗を評価し、スケジュールを守るために必要となる活動をする。

**リスクマネジメント**

プロジェクトの成果あるいはプロダクトの品質に影響を及ぼすリスクを見極める。

**ソフトウェア品質保証**

ソフトウェア品質を確保するために必要な活動を定義し遂行する。

**テクニカルレビュー**

次のアクティビティに影響を与えないよう、ソフトウェアエンジニアリングの成果物を評価し、エラーを発見して取り除く。

**測定**

チームが顧客のニーズを満たすソフトウェアを提供できるようにプロセス、プロジェクト、プロダクトの測定方法を定義し収集する。これらは他のフレームワークや包括的なアクティビティと併せて利用可能である。

**ソフトウェア構成管理**

ソフトウェアプロセス全体を通じて変更による影響を管理する。

**再利用管理**

ソフトウェアコンポーネントを含めた成果物に対する再利用の判定基準を定義し、再利用コンポーネントを実現する仕組みを構築する。

**成果物の準備と作成**

モデル、ドキュメント、ログ、フォームやリストといった成果物を作成するために必要なアクティビティが含まれる。

包括的なアクティビティの詳細については本書にて後ほど記述する。

### 1.3.3 プロセス適合

本節の前半において、ソフトウェアエンジニアリングプロセスはソフトウェアチームが盲目的に従わなければならない厳格な規定ではなく、俊敏（Agile）かつ問題、プロジェクト、チーム、組織文化に対して適合すべきと述べた。したがって、あるプロジェクトに採用されたプロセスは、他プロジェクトに採用されたプロセスとは大きく異なる場合もある。プロセスの違いは以下によるものである。

- アクティビティやアクション、タスクの全体的な流れとそれらの相互依存性。

- 各フレームワークアクティビティ内で定義されるアクションやタスクの粒度。
- 把握および必要とされる成果物の度合。
- 適用される品質保証アクティビティ。
- 適用されるプロジェクト追跡と管理アクティビティ。
- プロセスの記述に対する詳細さと厳密さ。
- 顧客とその他ステークホルダがプロジェクトに関与する度合。
- ソフトウェアプロジェクトチームの自律レベル。
- チーム組織と役割の規定された度合。

本書の1部では、ソフトウェアプロセスを詳細まで紹介する。

## 1.4 ソフトウェアエンジニアリングプラクティス

1.3節では、一連のアクティビティから構成される一般的なソフトウェアプロセスモデルを紹介した。これらは、ソフトウェアエンジニアリングプラクティスに対するフレームワークにもなる。コミュニケーション、計画策定、モデリング、構築、デプロイといった一般的なフレームワークアクティビティ、および包括的なアクティビティはソフトウェアエンジニアリング業務における骨組みを提供する。ここにソフトウェアエンジニアリングのプラクティスはどのようにあてはまるのだろうか。この後の節では、フレームワークアクティビティを適用する際の概念と原則について紹介する[*9]。

[*9] 本書後半のソフトウェアエンジニアリング手法や包括的なアクティビティを議論する際に、この章における関連した節を読み返そう。

### 1.4.1 プラクティスの本質

近代的なコンピュータが存在する前に書かれた古典的な著書である「How to solve it [Pol45]」のなかでGeoge Polyaは問題解決の本質について述べている。これはソフトウェアエンジニアリングプラクティスの本質ともいえる。

1. 問題を理解する（コミュニケーションと分析）
2. 解決策を計画する（モデリングとソフトウェア設計）
3. 計画を実行に移す（コード実装）
4. 結果が正しいことを確認する（テストと品質保証）

ソフトウェアエンジニアリングにおいても、これらの当たり前の手順により本質的な質問が導かれる[Pol45を参考とした]。

**問題を理解する**

真実を認めることは困難なことが多い。問題を指し示す際には、多くの人はこの傲慢さに苦しんでいる。数秒聞いて、そして考える。「理解したよ。それではこの問題を解決しようか」不幸なことに、大抵の問題は理解が簡単ではない。次の質問の回答に少し時間を費やす価値はあるだろう。

- 誰がその問題解決について利害があるか。すなわち、誰がステークホルダか。
- 何が分かっていないのか。問題を正しく解決するためにはどんなデータや機能、フィーチャが必要であるか。
- 問題は切り分けることができるか。より小さな、理解しやすい問題として表現ができるか。
- 問題は視覚的に表すことができるか。分析モデルをつくることができるか。

**解決策を計画する**

この時点では、問題について理解もしくは考えることができており、プログラミングを開始したくてしかたない状況である。しかし、プログラミングの前に、少しだけ落ち着いて小さく設計してみよう。

- 以前に類似の問題を見たことがあるか。解決案にパターンを見出すことができるか。求められるデータや機能、フィーチャを実装したソフトウェアがすでにあるか。
- 類似の問題を解決したことはあるか。もしあれば、そのときの解決策は部分的に再利用できないか。
- 関連する問題（subproblem）を思いつくことができるか。思いつく場合、その問題も解消できるか。
- 効果的な実装となる解決策を示すことができるか。設計モデルを作ることができるか。

**計画を実行に移す**

設計は、構築するシステムのロードマップを示している。それは期待していない回り道で、よりよい方法を見つける可能性もある。しかし、この「計画」は道に迷わず進むことを可能にする。

- 解決策は計画にしたがっているか。ソースコードは設計モデルまで追跡可能か。
- 解決策の構成要素は正しそうか。設計とコードはレビューされているか。もしくは、アルゴリズムの正当性は証明されているか。

**結果が正しいことを確認する**

まだ解決策が完璧なものであることは確信できていない。しかし、可能な限りのエラーを発見するための十分な数のテストを設計することで確信に至ることができる。

- 解決策の各構成要素をテストできるか。適切なテスト戦略が設定されているか。
- 解決策は求められるデータ、機能やフィーチャと合致した結果になるか。すべてのステークホルダの要求に対してソフトウェアの妥当性が確認されているか。

アプローチの多くは当たり前の内容であり、驚くようなものではない。実際、ソフトウェアエンジニアリングにおけるアプローチは、決して道に迷わせるようなものではない。

### 1.4.2 一般原則

原則（principle）という用語は辞書によると「基盤となる重要な法則、体系的な思考に必要な前提」と定義される。本書全般において、原則という言葉をさまざまな抽象度で用いる。あるときはソフトウェアエンジニアリング全体としての原則、他方では、コミュニケーションのような特定の一般的なフレームワークアクティビティとしての原則。 他にも、アーキテクチャ設計のようなソフトウェアエンジニアリングのアクション、利用シナリオを書くといった技術的なタスクに対する原則の場合もある。抽象度に関わらず、原則は確固たるソフトウェアエンジニアリングプラクティスのマインドセットを確立するために役立つ。このような理由により原則は重要である。

David Hookerは、ソフトウェアエンジニアリングプラクティス全般に対する7つの原則を提唱した[Hoo96]。以下に、文書を引用する。[*10]

[*10] 著者の許可を得て引用[Hoo96]している。Hookerはこれらの原則についてのパターンについて次のサイトで定義している。http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment

**第1原則：システムが存在する唯一の理由**

システムは「ユーザーに価値を提供する」という唯一の理由のために作られ、存在している。ソフトウェアが搭載されるシステムに対するすべての決定はこのことが念頭に置かれている。システムの要件を特定する前、システム機能の一部を特定する前、ハードウェアプラットフォームや開発プロセスを決定する前に、「これは真の価値をシステムへ与えるのか」と自問すべきである。その答えが「ノー」であれば、するべきではない。他の原則はこの第1原則をサポートするものでしかない。

**第2原則：シンプルにしておけ！（KISS：Keep It Simple, Stupid!）**

あらゆるソフトウェアシステムの設計作業において、考慮すべき要素は多い。すべての設計は可能な限りシンプルであるべきだ。しかし、必要以上にシンプルとする必要はない。このことは、システムをより理解しやすく、そして保守しやすいものにする。これは、シンプルさの名のもとにフィーチャを放棄せよという主張ではない。実際のところ、より洗練された設計であるほどシンプルなものである。シンプルとは「急いで雑につくる」意味ではない。事実、設計のシンプルさのために何度となく熟慮を繰り返す。その見返りは、より保守が容易でエラーの少ないソフトウェアシステムを得られることにある。

**第3原則：ビジョンを持ち続けよ**

ソフトウェアプロジェクトの成功には明確なビジョンが不可欠である。考え方に一貫性がないと、システムは互換性のない設計で継ぎ接ぎだらけとなり、間違った種類のネジで繋がれる危険性がある。例えば、ソフトウェアシステムのアーキテクチャに対するビジョンの妥協をすると、たとえよい設計のシステムであってもそれが台無しになり、結局は崩壊する。ビジョンを持ち、その徹底ができる強力な権限をもったアーキテクトの存在は、ソフトウェアプロジェクトの成功に大いに寄与する。

**第4原則：あなたのつくったものを他の人が使用する**

あなたのやっていることを他人が理解しなければならないということを常に考えながら仕様を決め、設計し、ドキュメントを書き、実装するべきである。ソフトウェア開発のすべての成果物には潜在的に多くの読者がいる。仕様はユーザの視点で決める。設計は実装者を念頭において行う。コーディング時にはシステムの保守や拡張の担当者を考慮する。あなたの書いたコードを誰か他の人がデバッグする場合もある。その人々はあなたのコードのユーザといえる。彼ら/彼女らの仕事を容易にすることは、システムの価値を高めることになる。

**第5原則：未来へオープンであれ**

近年のコンピュータ環境では、仕様はめまぐるしく変わり、ハードウェアプラットフォームはたった数カ月で陳腐化し、ソフトウェアの寿命は年単位ではなく月単位で計測されつつある。しかしながら、真に強いソフトウェアシステムは長寿命でなければならない。長寿命を達成するためには、システムはさまざまな変更に対応できるように準備されていなければならない。これを実現するシステムは、最初からそのように設計されている。自らを窮地に立たせるような設計は決してしてはならない。常に「もし...ならどうなるか」という問いかけを行い、特定の問題だけでなく、一般的な問題を解決するシステムを構築して、可能な限り多くの選択肢を与えられるよう準備する[*11]。

[*11] このアドバイスは極端な解釈をすると危険である。「一般的な問題」に備えて設計する際には、性能を妥協し、非効率な解決策が必要とされることもある。

**第6原則：再利用に先駆けて計画せよ**

再利用は時間と工数を節約する[*12]。ソフトウェアシステムの開発において、高いレベルでの再利用を達成することは、達成するのが非常に困難であるといえるであろう。コードと設計の再利用はオブジェクト指向技術を用いる主要な利点の1つだと言われてきた。しかしながら、この費用対効果は約束されていない。再利用に先駆けて計画することは、コストを低減し、再利用するコンポーネントとそれを取り込むシステムの両方の価値を高めることである。

[*12] これは将来のプロジェクトでソフトウェアを再利用する側にとってはそのとおりであるが、再利用可能なコンポーネントを設計し、構築する側には高価となることがある。研究結果によると再利用可能なコンポーネントを設計、構築することは、対象のソフトウェアを構築するよりも25〜200%のコストが余計にかかる。このコスト差異は正当化されない場合もある。

**第7原則：考えよ！**

最後の原則がおそらく最も見落とされがちである。行動を起こす前に明晰で完結した考えをもつことは、必ずと言ってよいほどよい結果を生む。何かについて考えた際は、正しい行動となる可能性は高くなる。そして、正しい行動を繰り返すための知見を得ることにもなる。考えたにもかかわらず正しい行動とならなかった場合でも、それは貴重な経験になる。考えることの副産物は、知らなかった何かに気づく学びであり、それにより答えを探すことができる。明晰な考えがシステムに入り込むことで価値が生まれるのである。前出の6つの原則を適用するには深い思慮を必要とするが、それによる見返りは極めて大きい。

もしすべてのソフトウェアエンジニアおよびチームがHookerの7原則に従うならば、複雑なコンピュータシステムを構築する際に直面する困難の多くは取り除かれるだろう。

## 1.5 どのようにはじまるか

ソフトウェアプロジェクトは、何らかのビジネスニーズによって立ち上げられる。そのニーズとは、既存のアプリケーションにおける欠陥を修正するもの、レガシーシステムをビジネス環境の変化に追従させるもの、既存のアプリケーションの機能を拡張させるもの、新しいプロダクトやサービスもしくはシステムを求める場合等が存在する。

ソフトウェアエンジニアリングプロジェクトの立ち上げ時には、ビジネスニーズは非公式な会話の中に含まれている。ソフトウェアを話題として取り上げることはほとんどないが、プロダクトを生かすも殺すもソフトウェア次第である。ソフトウェアが成功した場合のみ、開発が成功したといえる。言葉にされていないものを含め、顧客のニーズに適したソフトウェアがプロダクトに組込まれた場合のみ、市場にプロダクトが受け入れられる。以降の章を読む際に、ソフトウェアがどのように発展するか意識するとよい。

## 1.6 まとめ

ソフトウェアは、コンピュータシステムやプロダクトの進化における重要な要素であり、世界において最も重要な技術の1つである。過去60年以上もの間で、ソフトウェアは問題解決や情報分析の特別なツールからソフトウェア自身が産業となるまでに進化を遂げた。それでもまだ、高い品質のソフトウェアを予算および納期を守って開発することは困難である。

ソフトウェア（プログラム、データ、コンテンツを含む）は、幅広い技術やアプリケーション分野に広がっている。保守担当者によって、レガシーソフトウェアに対する格別な取り組みが続けられている。

ソフトウェアエンジニアリングは、複雑なコンピュータシステムを品質を確保しつつ納期内に構築するためのプロセス、手法、ツールを含んでいる。ソフトウェアプロセスは、すべてのソフトウェアプロジェクトに適用されるコミュニケーション、計画策定、モデリング、構築、デプロイという5つのフレームワークアクテ

ィビティをまとめたものである。ソフトウェアエンジニアリングプラクティスは、中核となる一連の原則に従った問題解決の活動である。

ソフトウェアエンジニアリングを学ぶことで、なぜソフトウェアプロジェクトを始める際に中核となる原則を考える必要があるか理解していくだろう。

## 問題と考察のポイント

1.1 予期せぬ結果の法則がコンピュータソフトウェアにどのように適用されてきたのか、少なくとも5つの例を挙げよ。

1.2 社会に対するソフトウェアの影響を示す良い例・悪い例をいくつか挙げよ。

1.3 1.1節の最初に問われた5つの質問に対する自分の答えを作成せよ。クラスメートとその答えについて議論せよ。

1.4 現代のアプリケーションの多くは、エンドユーザの手に渡る前にも最初のバージョンが利用可能になった後でも頻繁に変更される。変更が発生してもソフトウェアが悪化しない方法をいくつか提案せよ。

1.5 1.1.2項に記載のソフトウェアの7分野を考察せよ。それぞれに同一のソフトウェアエンジニアリングの方法を適用することが可能か。回答に対して理由を説明せよ。

1.6 ソフトウェアが普及するにつれ、欠陥のあるプログラムによる公共へのリスクがさらなる心配ごととなる。プログラム欠陥によって経済および人類に最悪の被害を受ける「最後の審判日」を現実的なシナリオで示せ。

1.7 プロセスフレームワークを自分なりの言葉で表現せよ。フレームワークアクティビティがすべてのプロジェクトに適用できるということは、規模や複雑性にかかわらず、すべてのプロジェクトに同じタスクを適用するということを意味するか。自分の意見について説明せよ。

1.8 包括的なアクティビティはソフトウェアプロセス全体で発生する。包括的なアクティビティはプロセス全体にバランスよく適用されるか、もしくは特定のフレームワークアクティビティに集中して適用されるか、考えを示せ。