1. ソフトウェアの進化と寿命

|  |
| --- |
| The Creator God uses evolution to bring about his plan.  創造主である神は、進化によってその計画を実行された。   * Asa Gray (アメリカの植物学者・有神論者) |

# ソフトウェアの進化と寿命

図1 - 1に、ソフトウェア製品がどのように起こり、どのように途絶えるのか、そのライフサイクルを示します。



図1 - 1 ソフトウェア製品が進化するようす

これは、とある架空のソフトウェア製品がどのようにバージョンアップを繰り返してきたかを示したものです。この系図の枝上に書かれた丸印が、この製品のバージョン (世代) を表しています。小さな丸印は開発プロジェクトチームの内部だけにリリースされ、大きな丸印はユーザーにリリースされたことを意味しています。世代を重ねるごとに、この製品には機能が追加され、バグが手当てされて品質が良くなっていきます(バグとは、ソフトウェアの病気[[1]](#footnote-1)のようなものと考えてください)。この、丸印で表されたソフトウェアの各世代のことを「ビルド」といいます[[2]](#footnote-2)。

一般に、ソフトウェアのバージョン番号は、大きな丸印の単位で、数ヶ月から数年の単位で更新されます。しかし、小さな丸印で示された「ビルド」は1週間おき(もしくは1日おき、数時間おきといった程度の間隔)で新しくなり、ビルド番号という番号が付けられて履歴が管理されます。

# ソフトウェアの進化の枝分かれ

ソフトウェア製品は、さまざまな事情で、図1 - 1のように枝分かれしながら進化していきます。例えば、この製品のあるユーザーが、「この製品に、我々専用の特別な機能を追加してほしい」と言ってきたとします。するとこの製品は、正式版と、そのユーザー用にカスタマイズした版の2つに分けられて、それぞれが別の進化をたどることになります。たとえ、あなたの製品のユーザーが1社しかないとしても、いろいろな事情でを切り分ける機会が生じます[[3]](#footnote-3)。

しかし、ソフトウェアを開発する側からみると、このようなブランチが何本もあると、それだけで開発・保守・リリースなどの費用がより多くかかってしまいます。従って、やむをえず分けたブランチは早めに廃棄し、元のブランチと統合しなければなりません。例えば、あるユーザー専用に開発した機能を、ほかのユーザーに影響を与えない形で元のブランチに統合し、そのユーザーにはどこかのタイミングで元のブランチの製品に乗り換えてもらうように誘導するなどです。

生物の進化が自然淘汰によるものであるのに対して、ソフトウェアの進化は戦略的かつ計画的です[[4]](#footnote-4)。ソフトウェアの進化には、大きくわけて開発と保守の2種類があります。細かくいえば、開発によるものは主開発のほか、機能のバリエーションを追加したり、日本向けや欧米向けなど複数言語に対応するなど、適用範囲を広げていく開発に分けられるでしょう。この、主開発による進化のためのブランチのことを、トランク (幹) とかメインライン、メインブランチ[[5]](#footnote-5)などとよびます。保守によるものは致命的なバグだけを修正し、瑣末なバグについては(リグレッション[[6]](#footnote-6)を避けるために)修正しないメンテナンスブランチや、特定の顧客のための修正をするためのユーザーブランチなどがあります。ブランチの種別については、★3-21節で詳細に扱います。

# 種としてのソフトウェアの断絶

さて、このように自社製品を進化させていくことができれば幸せなのですが、残念ながらメインラインを含む全てのブランチの製品が死滅してしまう (保守ができなくなってしまう) ことがあります。

もちろん、バグがなく安定して動作しているので進化 (保守) するのも不要ということなら、それはとても良いことです。このようなソフトウェアのことを、枯れたソフトウェアといいます。この「枯れた」というのはソフトウェアの褒め言葉で、バグが少なく、安定していて安心して使えるという意味です。

しかし、良くない設計やその場しのぎの修正を積み重ねていくと、コード (プログラム) はスパゲッティのようにごちゃごちゃと絡まっていきます。このような状態でバグを修正したり、新機能を追加しようとすると、今動いている部分のどこかが動かなくなり、新しいバグを作りこんでしまう危険が大きくなります。こうなると、進化を続ける (バグを修正したり、新機能を追加したりして新しいバージョンの製品を構築する) ことができなくなってしまいます。つまり、世代を重ねるごとに製品は重い病気にかかって老化という不可逆的な変化をたどり、ついには保守ができなくなって、種としての寿命を迎えてしまうのです。このようなソフトウェアの特性を、メンテナビリティ(maintainability; 保守可能性)とか、サステナビリティ(sustainability; 持続可能性)といいます。私自身、保守ができなくなったため、それをゼロから新しく作り直すという開発案件をよく耳にしました。実際、生きているブランチと死んでいるブランチというのがあるもので、死んでいるブランチから製品をビルドすると、ビルドブレークしたり、スモークテストで煙を噴いたりします[[7]](#footnote-7)。

一般に、バージョンアップを繰り返しながら、機能と安定性を増しているソフトウェア製品を継続してリリースしている企業は、ソフトウェア開発の方法を良く知っているといえるでしょう。逆に、作ったシステムを数年ごとに何度も作り直しているような企業は、ソフトウェア開発の方法を理解していません。これまでに莫大な時間と人 (お金) を投じて継続的に開発・保守され、バージョンを重ねてきた製品は、それ自体が大きな資産となっています。例えばMicrosoft Excelのような長寿の製品は、これまでに甚大な投資がなされ、それに見合った大きな価値をもっています。強調しておきますが、いくら投資を続けても、数年で寿命が尽きてしまうような製品ばかり作っているソフトウェア会社は、その企業価値そのものを高めることもできないのです。

# ソフトウェアを育てる

では、ソフトウェア開発に対してどのようなアプローチを用いれば、ソフトウェアを病気にせずに継続して開発・保守できるのでしょうか。それには、各ビルドを健康に保てば良いのだと気づきます。具体的には、次のようなポリシーを適用します。

* コードを清潔に保ち、バグが入ったらすぐに取り除く
* 機能の追加は少しずつ、ソフトウェアが健康なときに限り行う
* 機能の追加とバグの除去は同時に行わない[[8]](#footnote-8)
* 頻繁にビルドしてリリースする
* 各ビルドをテストして、その健康状態をこまめにチェックする

よく、きれいなコード、クリーンなコードを書くことが大事と言われます。cleanとは、清潔で汚染されていないということなのです[[9]](#footnote-9)。ソフトウェアとは、本質的に複雑なものです。しかも、規模が大きくなるとその複雑さは急激に増加するため、どこかに手を入れると別のどこかに意図しない副作用 (side effect) があらわれたりします。ソフトウェアは非常に壊れやすいため、壊れていない状態を保ったまま、少しずつ進化させることが肝要です。これをインクリメンタル (漸増的) なアプローチといいます。もしバグが見つかっても、どの部分のコード修正により動かなくなったのかを突き止めることができれば、最悪でもその修正をすべて元に戻せばそのバグは治ります。しかし、たくさんの機能を同時に追加してバグが発生したら、それがどのコード修正により発生したのかを特定するのは困難です。また、たくさんの機能を少しずつ追加したとしても、ソフトウェアの健康診断を怠って、発生したバグをすみやかに発見・除去しなければ同じことが起こります。除去できなかったバグの上に新機能をどんどん追加してしまったら、そのバグは埋もれて見つかりにくくなりますし、単純に元に戻すこともできなくなります。仮にそのバグを取り除くことができたとしても、その上に実装された新機能 (の一部) も一緒に取り除かれてしまい、動かなくなってしまうかもしれません。



図1 - 2 漸増的なアプローチを繰り返して行う

ですから、機能を追加したら、正しく追加できたかどうかテストして確認します。もしバグが見つかれば、これを取り除いて品質を安定させます。ビルドを頻繁にリリースし、バグが除去できたかどうか、テストして確認します。そのためには、加えた変更が多くなりすぎる前に、ビルドをリリースしてテストする必要があります。リリースやテストにもコストがかかりますから、丁度よいビルドの頻度というのがあります。多くの場合、それは1週間から2週間くらいです。このように、ビルドを直してはリリースするという作業を定期的に繰り返すことをイテレーティブ (反復的) なアプローチといいます。このようにして、少しずつ着実に開発中のソフトウェアを理想に近づけていきます。



図1 - 3 クリエイティブな仕事にも、決まりきった手順が必要！

# ビルド中心の生活

UP (Unified Process; 統一プロセス)[[10]](#footnote-10) というソフトウェア開発プロセスには「アーキテクチャ中心」というコンセプトがあります。アーキテクチャとは、ソフトウェアの骨格のことです[[11]](#footnote-11)。しっかりした骨格がなければソフトウェアを正しく進化させることなどできないのだから、これをソフトウェア開発の中心に据えよう！というわけです。確かに、悪くない考え方です。しかし、ソフトウェア開発で「アーキテクチャ中心」を実践するには、高度な職人の技が必要です。もし、みなさんがまだ駆け出しのソフトウェア技術者なら、「アーキテクチャ中心」よりもまず「ビルド中心」という考え方を意識することが役に立つと思います。私は今、プライベートでは子ども中心の生活ですが、職場ではまさしくビルド中心の生活を送っています。いつもビルドの健康状態に気を配り、病気の兆候がないか見張っているのです。それは私だけでなく、私が所属している開発チーム全員がそういう気持ちで日々を過ごしています。



図1 - 4 進化するソフトウェアの4象限

# コラム ソフトウェア開発のメタファ① 旅行

|  |
| --- |
| **コラム ソフトウェア開発のメタファ① 旅行**  ウォーカー・ロイス氏は、著書「ソフトウェア開発のエコノミクス」に、反復型開発に関する深い知見をコンパクトにまとめています。この本の冒頭には、旅行の計画とソフトウェアの開発計画の類似性が指摘されています。旅慣れない旅行者は、旅先の言語や文化・習慣、インフラなどに疎いものですが、ソフトウェア開発に携わる人も同じだというのです。ソフトウェア開発でも、プログラミング言語や、それぞれの開発組織に文化として定着した習慣、インフラとして使っている開発ツールなどについて、旅に出るのと同様の準備が必要だからです。  また、詳細すぎる計画を最初に立ててしまうと、せっかくの旅行の品質が失われてしまいますが、ソフトウェア開発でも同じだといいます。例えば、見知らぬ土地での食事や買い物を分刻みで厳密に計画できるはずはなく、そんな緻密な計画にこだわっては、楽しい旅行を台無しにしてしまいます。旅慣れた旅行者は、まずスタート地点とゴール地点を決め、旅の概略を検討します。そして、時間的に近い事柄について、少しずつ計画を詳細化していきます。例えば、朝ホテルで目覚めたら、今日は1日どのように過ごすかを計画します。ロイス氏は、ソフトウェアの開発者も、旅慣れた人のようであるべきだといっています。つまり、最初にソフトウェア開発計画のアウトラインを決め、反復を繰り返しながら少しずつ仕様を詳細化していきます。最初にすべての仕様を詳細に決め、それに厳密に従おうとしても、台無しのソフトウェアしか出来上がらないのです。  私は、子どもが産まれる前に妻とバリに旅行したのですが、準備不足で大変なことになりました。旅行会社でツアーを申し込み、集合時間に間に合うように羽田空港に行きました。しかし、集合場所のはずの3Fには喫茶店しかありません。変だなと思ってツアーの説明書をよく見ると、なんと集合場所は羽田ではなく成田空港だと書いてあるのです。しばらく考えた末、これは隠し通せないと思い、妻に「ごめん、集合場所は成田だった」と打ち明けたところ、妻はその場に崩れ落ちたかと思うと起き上がり、私をぽかぽかと殴り始めました。  海外旅行に携帯電話は不要かと思ったのですが、念のため持ってきていたのが功を奏しました。成田への経路をウェブで調べたところ、羽田から成田へ直通の高速バスが出ているのを見つけて飛び乗り、ぎりぎりで飛行機の出発時刻に間に合わせることができました。しかし、間に合うかどうかは成田に着くまでわからず、私はバスの中でずっとぽかぽかと殴られる羽目に陥りました。ロイス氏の言うとおり、時間的に近い事柄の準備や計画を怠っては、旅行を台無しにするばかりか、羽田離婚に至る可能性さえあるようです。ソフトウェア開発も、最初にチームをへ正しく方向づけることが重要です。また、正しい方向へ向かっているか、定期的に観測することも必要であることがわかります。 |

# 本書で説明するツール

本書では、このようなソフトウェアの進化を支援するツールを4つ説明します。これらは、ソフトウェア開発を支える重要な基盤です。ここでは、これらのツールがソフトウェアの開発をどのように支援するか、その概要を示しておきます。

## 構成管理ツール

構成とは、複数のプログラムファイルやコンポーネント、種々の文書などの組み合わせのことです。構成管理とは、ビルド対象のソフトウェアが、どの組み合わせによって構築されるべきかを管理し、その一貫性を保つことです。これを支援するのが、ソフトウェア構成管理ツール (SCM; Software Configuration Management) です。構成管理ツールを使えば、各ファイルの変更を追跡し、いつ誰がどこを変更したのかを記録できます。よく使われる構成管理ツールには、SubversionやGit, Mercurialなどがあります。

構成管理ツールは、単にすべてのプログラムファイルの変更履歴を追跡するだけのものではありません。同じプログラムファイルを開発者全員で共有し、同時に編集できるようにする仕組みも提供します。また、プログラムを枝分かれさせることで、本章で説明したようなソフトウェアの進化を実際に可能にします。構成管理ツールは、みなさんのプログラムを育てる培養器であり、その起源を記録するタイムマシンであり、またプログラムが損傷しないように守る耐火庫でもあります。みなさんがソフトウェアを開発するときの作業場を提供する、とても大事なものです。構成管理ツールとコミットのプロセスについては、3章で説明します。

## ビルドツール

ソフトウェア開発における最終的な成果物は、ユーザーに価値を提供できる実行可能なソフトウェアです[[12]](#footnote-12)。この、実行可能なソフトウェアのことを「ビルド」といいます。また、プログラムファイル一式を統合し、実行可能なソフトウェアを構築することを「ビルドする」といいます。

ソフトウェア開発においては、その製品を何度もビルドしなければなりません。こういう繰り返しの作業は、コンピュータが最も得意とする分野です。つまり、ソフトウェア製品のビルド作業は、コンピュータで自動化すべきものです。この自動化を支援するのが、ビルドツールです。ビルドツールには、Make, Ant, NAnt, MSBuildなどがあります。また、より高度なビルドを実現するビルドサーバというソフトウェアもあります。

ビルドを自動化するには、そのためのプログラムのようなものを書きます。これをビルドスクリプトといいます。ビルドスクリプトをビルドツールで実行することで、開発中のソフトウェアをビルドできます。ひと言にビルドといっても、この中には多くの作業が含まれます。できるだけ多くの作業を自動化し、かつ単純で保守しやすいビルドスクリプトを記述することが重要です。それが健康的なソフトウェアを構築することにつながるからです。ビルドツールとビルドのプロセスについては、4章で説明します。

## バグ追跡システム

ソフトウェアの開発では、ビルドとリリースを繰り返しながら、ソフトウェア製品を少しずつ進化させていきます。各ビルドをテストして、もしバグを発見したら、これをその後のビルドで修正し、正しく修正できたことを確認します。この作業をスムーズに行うために、発見したバグをすべて文書化し、それらを正しく修正できたか追跡していかなければなりません。開発中のソフトウェアで発見されるバグの数は、累計で簡単に数百から数千という数になりますから、それらを紙に書いて管理するのは非効率です。これも、コンピュータによって自動化すべきタスクです。これを支援するのが、バグ追跡システム (BTS; Bug Tracking System) です。これには非常に多くの種類がありますが、現在はTracやJira、Redmineなどに人気があるようです。

BTSにしたバグ報告票は、各チームメンバーのタスクリストとして扱います。つまり、バグ報告票ごとに担当者を割り当てます。これにより、誤って複数人が同じ作業をしてしまうことを防ぎます。また、バグ報告票に優先順位をつけたり、忙しい人がそうでもない人にバグ報告票を渡したりできるので、効率的に作業をすすめられます。ただし、バグを1人だけで処理することはできません。チーム内のさまざまな立場の人の間でバグ報告票をやりとりしながら、段階を踏んで処理します。このやりとりの順序 (ワークフロー) を変更することで、私たちの業務を改善し、最適化を図ることもできます。さらに、現在どれだけのバグが残っているのか、すでに解決できたバグはどれだけあるか、といった指標を使って、ソフトウェア開発プロジェクトの現在の進捗を計ることもできます。多くのバグ追跡システムは、このような作業を支援するための機能を実際に備えています。バグ追跡システムとバグ追跡のプロセスについては、5章で説明します。

## テストケース管理ツール

ソフトウェア開発とは、仕様を定義し、それを実行可能なものに変換するプロセスであると定義できます。正しく変換できたかどうかは、テストで検証します。テストの手順書 (テストケース) も、ビルドと同じく実行 (実施) 可能なドキュメントであることに注意しましょう[[13]](#footnote-13)。テストケースは手作業で実行するテストと、PCが自動実行するテスト (オートメーション) に分類できます。本書は、オートメーションの開発手法については触れませんが、テストとはどのような性質をもつのかを説明し、テストケースとビルドの関係を明らかにします。テストケース管理ツールとテストケースを記述する手順については、6章で説明します。

これらのツールは、ソフトウェア開発における多くの作業をソフトウェアで自動化します。まさしくソフトウェア開発に必須のインフラであり、製品の健康を守るライフラインなのです。これらを利用しないソフトウェア開発プロジェクトは、よく「紺屋の白袴」などと揶揄されます。自分たちの仕事さえコンピュータ化できないのに、ユーザーの業務をコンピュータ化してさし上げましょうなどとは、おこがましいというものです。

# 開発プロジェクトに必要な役割 (ロール)

ツールを使って自動化できる部分を自動化すれば、人間はより高度で知的な作業に集中できます。ただし、すべての作業を完全に自動化することはできません。ツールはツール (道具) であって、ロボットではないからです。各ツールは、異なる役割をもった別の担当者らにより、それぞれの責務を果たすために操作されます。ですから、みなさんがツールを使うときも、自分の役割や立場をよく理解しておく必要があります。さもないと、何のためにツールを使うのかという大事な目的を見失ってしまいます。そこで本節では、ソフトウェア開発プロジェクトに必要な責務 (役割) を紹介しておきます。

一般に、ソフトウェアを開発するプロジェクトチームには、次のような責務を担う人員を配置します。



図1 - 5 ソフトウェア開発プロジェクトチームに登場する役割 (例)

これらの役割を指して、ロールといいます。ロールプレイングゲーム (役割を演じて遊ぶゲーム) のロールと同じ意味です。また、このようなロールを指してディシプリン(discipline; 規律)ということもあります。ロールは帽子のようなものなので、状況に応じて別の帽子をかぶり分けることができます。つまり、1人が複数のロールを担当しても構いません。少人数でソフトウェアを開発するときは、ここで示したロールのいくつかを1人で担当します。例えば、ある人がプロジェクトリードとデブリードを兼務し、ある人がアーキテクトと開発者とビルドマスターを兼務し、ある人がテストリードとテストエンジニアとテスターを兼務すれば、3人で開発プロジェクトチームを構成することもできます。ただし、同時にかぶることができる帽子は1つだけなので、別の帽子をかぶりたいときは、いまかぶっている帽子を一時的に脱がなければなりません。開発作業に従事するときは、いま、自分がどの帽子をかぶって開発に参加しているのかを意識しておく必要があります。特に、同じ人が開発者とテスターを兼ねるときには、この感覚が非常に重要です。開発者とテスターでは、その役割が正反対だからです[[14]](#footnote-14)。下記に、それぞれのロールについて説明します(かっこ内は別名)。

## プロジェクトリード (プロジェクトマネージャ)

プロジェクトチーム全体を統括する棟梁です。ユーザーと折衝してチーム全体を方向づけ、チームメンバー全員が気持ちよく安心して働くことのできる環境を供出する役割を担います。チーム内で作業が滞っていれば、その原因を見つけて解決し、作業がスムーズに流れるようにするなど、チームの全体最適を考慮した活動を行います。プロジェクトが佳境に入ってお祭り騒ぎ (てんやわんやの大忙し) になったときは、音頭をとってプロジェクトを鎮火するのも、プロジェクトリードの責務です。

## デブリード

開発チームを統括します。開発者に指示を出して、作業を割り当てます。また、技術的なリスクを早期に発見して対策を立てます。

## アーキテクト

ソフトウェアのアーキテクチャの構築に責任をもつ開発者です。開発対象のソフトウェアについて、最も技術的に明るい開発者が担当します。デブリードがアーキテクトを兼務することもよくあります。

## 開発者 (ソフトウェアエンジニア、デベロッパー、デブ)

ソフトウェアを開発する人です。文字通り、ソフトウェア開発における主役を果たします。開発者には、当該のプログラミング言語や技術、当該のドメイン (ユーザー業務などに関する知識) などに専門性が求められます。

## ビルドマスター (ビルドエンジニア、ビルド担当者)

ソフトウェアのビルドを担当します。ビルドマスターは開発プロセスを学習しやすいため、新人のエンジニアにこのロールを (も) 担当してもらうことも多いでしょう。製品のブランチが非常に多かったり、製品の規模が大きいためにビルドに半日から1日程度の時間がかかるといった場合には、開発チームの内部 (あるいは外部) に専任のビルドチームを配置し、ビルドとリリースの作業を担当してもらうこともあります。

## テストリード

テストチームを統括します。テストエンジニアとテスターに指示を出し、作業を割り当てます。多くの場合、テストエンジニアを兼務します。

## テストエンジニア、テスター

テストを計画し、実装し、実施する役割を担います。一般に、比較的難しい作業 (テスト仕様書の記述、テストケースの設計、統合ビルドのテストの自動化など) をする人を指してテストエンジニアといいます。これに対し、比較的簡単な作業 (マニュアルテストの実施) を行う人をテスターということが多いようです。

## 構成管理者 (コンフィギュレーションマネージャ、SCMエンジニア)

構成管理とは、広義には当該の製品の部品 (ソースコードやドキュメントなど)の組み合わせの一貫性を保つことですが、本書では狭義にSCMサーバの管理についてのみ扱います。図1 - 5では、独立して構成管理者を配していますが、開発チーム内のひとりが構成管理者を兼ねることもあります。また、プロジェクトチーム外部の、社内のIT部門がSCMサーバを管理し、このインフラを複数の開発プロジェクトに提供することもあります。メールサーバと同じく、SCMはソフトウェア開発企業にとって非常に重要なインフラだからです。SCMエンジニアには、当該の構成管理ツールに対する深い知識が求められます。

このほか、チーム内外のプロジェクトに関する利害関係者のことをステークホルダー (stakeholder) といいます。Stakeとは仕事上の利害、あるいはゲームの賭け金という意味で、stakeholderとはまさしくソフトウェア開発というゲームに自分の虎の子を賭けている人たちのことです[[15]](#footnote-15)。

このほかにも、開発スタイルによって別の担当者を配置することもあります[[16]](#footnote-16)が、ここではどのドメインのソフトウェア開発にも一般的なものを紹介しました。

「これは自分の仕事ではないので、やりません」といった態度は良くありません。しかし、優秀な人がなんでもかんでもやる、といったやり方はもっと良くありません。ソフトウェア開発に必要な役割や責務をきちんと特定し、それぞれに責任者を決めておくことが、ソフトウェア開発のプロセスをスムーズに動かすための前提条件です。

# コラム 開発中の催し① キックオフミーティング

|  |
| --- |
| **コラム 開発中の催し① キックオフミーティング**  キックオフミーティングは、サッカーのキックオフと同じく、プロジェクトの開始を宣言するときです。プロジェクトの目的や開発するソフトウェアの概要、スケジュールなどを共有し、チームのオリエンテーション(orientation; 方向付け)をします(オリエンテーションは、学校の遠足などでもお馴染みですね)[[17]](#footnote-17)。チーム構成やレポートライン、チーム運営のポリシーなどを説明したり、このプロジェクトで初対面の人がいれば自己紹介をしたりします。  社内でプロジェクトが登録されるときに、プロジェクトにはコードネームがつけられることが多いと思います。しかし、もしみなさんの組織にコードネームに関する特段のルールがなく、まだコードネームがなければ、キックオフミーティングでつけてしまいましょう。みんなでアイデアを出して決めます。何でもいいのですが、チームをうまく方向づける名前が見つかると良いです。これにより、チームメンバーの一体感や結束力、責任感を高めることができます。友達同士でバンドを組むときに、バンド名を決めるのと同じような効果があるのです。ただし、コードネームを決めることに時間を使い過ぎないようにしましょう。  パッケージ製品では、メジャーバージョンが変わるごとに、同じ製品でも別のコードネームを使う習慣が一般的です。昔の話ですが、Lotus のAmiProというワープロソフトでは、バージョンが変わるごとに別の万年筆のブランド名をコードネームにしていました。ユーザーに、素晴らしい万年筆を使うときと同じユーザーエクスペリエンス[[18]](#footnote-18)を提供できるように、との願いをこめたものです。また、プレゼンテーションソフトのFreelanceにはDaVinciやRemingtonなど画家の名前が、データベースソフトのApproach JにはVindalooやJalfreziなどのカレーの名前が、コードネームとして使われました。1-2-3 R3はGodivaで、その日本語版の1-2-3 R3JはGlicoでした(バージョンを通してお菓子メーカーの名前で統一されていたわけではありませんが)。  また、Lotus Domino Serverというサーバ製品は、コードネームがそのまま製品名になりました。表向きは、「リンクを動的につないでいくWebサーバだから、数字をつなぐゲームのドミノにあやかった」という説明がされました。が、本当は「アプリケーションサーバ市場を席巻(dominance)する」という意味が込められていたそうです。  最近の話では、Microsoftの検索エンジンBingのコードネームがKumoでした。クラウド(雲)やウェブ(蜘蛛の糸)を連想させる、楽しいコードネームです。このほか、Microsoft Windows Phone 7のコードネームはMangoです。競合製品を、より甘い製品でやっつけようという気持ちが、チームをひとつにします。 |

# まとめ

本章では、ソフトウェアが枝分かれしながら進化するものであることを紹介しました。進化論のことを、英語ではThe theory of evolutionといいますが、このevolutionという語には「進化」のほかにも「発展」「案出」「漸近的な変化」などの意味があります。Evolutionという言葉は、などではなく、直接的にソフトウェア開発を意味する言葉なのです。

そして、ソフトウェアを壊れていない状態に維持したまま、進化させるためのユーティリティとして、構成管理ツール、ビルドツール、バグ追跡システム、テストケース管理ツールの4つを紹介しました。これらの詳細は、続く章で説明します。

また、ソフトウェアを正しく進化させるための組織と役割 (ロール) を紹介しました。これらのロールの中でも、開発者、テスター、ビルドマスターは、続く章にも頻繁に登場します。ソフトウェアを開発するうえで、彼 (女) らがどのようなやりとりを必要とするのか、本書の残りでさらに詳しく見ていきましょう。

# 参考文献

## Is Design Dead?

Martin Fowler

<http://martinfowler.com/articles/designDead.html>

(邦訳)

<http://www.objectclub.jp/community/XP-jp/xp_relate/isdesigndead>

## リファクタリング―プログラムの体質改善テクニック

<http://www.amazon.co.jp/dp/4894712288>

## アジャイル開発者の習慣－acts\_as\_agile 第5回 設計を進化させる

角谷信太郎

<http://gihyo.jp/dev/serial/01/agile/0005>

## Lead programmer

<http://en.wikipedia.org/wiki/Lead_programmer>

1. ソフトウェアの病気ときいて、コンピュータウィルスを想像する読者もいるでしょう。しかし、ここでいう「ソフトウェアの病気」とは、意図した通りに動かなかったり、大事なデータを意図せず消してしまったりするなどのソフトウェアの不具合のことです。 [↑](#footnote-ref-1)
2. ビルドという用語の正確な定義は後で示します。 [↑](#footnote-ref-2)
3. これらの事情については、3章で扱います。 [↑](#footnote-ref-3)
4. 3つのチームに同じものを作らせて、一番優れたものを作ったチームの成果物を製品としてリリースする、という手法もあります。また、市場に出たソフトウェアも、自然淘汰によるモデルでその寿命を説明できるかもしれません。しかし、本書で取り上げている構成管理ツールのブランチング (枝分け) は、計画的に行われるものであり、自然淘汰のモデルでは説明できません。 [↑](#footnote-ref-4)
5. サブシステムごとにメインライン (メインブランチ) を準備することがあるので、メインブランチが1本だけになるとは限りません。 [↑](#footnote-ref-5)
6. リグレッションとは、動いていたはずの機能が動かなくなってしまうバグのことです。デグレード、デグレなどということもあります。リグレッションについては★5-10節で後述します。 [↑](#footnote-ref-6)
7. ビルドブレークやスモークテストについては、後の章で説明します。 [↑](#footnote-ref-7)
8. 「同時に行わない」とは、「同時にコミットすべきでない」という意味です。ソフトウェア開発の段階としては、バグ修正をせずに機能追加だけを行う期間を準備することはなく、機能追加とバグ修正は並行して実施するのが一般的です(ただし、開発の後期には、機能追加をせずにバグ修正だけを行う期間を設けることはあります)。コミットについては3章で扱います。 [↑](#footnote-ref-8)
9. コードの不衛生な匂いとその除去方法については、書籍「リファクタリング―プログラムの体質改善テクニック」が参考になります。 [↑](#footnote-ref-9)
10. UPについては、7章で紹介します。 [↑](#footnote-ref-10)
11. 特に、プロジェクトの初期に確立した基本的なアーキテクチャのことを、ベースラインアーキテクチャといいます。 [↑](#footnote-ref-11)
12. このほかにも、ユーザーに納品すべき成果物がありますが、それは個々のソフトウェア開発プロジェクトで異なります。本書では、最終的な成果物として、実行可能なソフトウェアに注目します。 [↑](#footnote-ref-12)
13. 厳密にいえば、実施可能なテストの手順書はテストスクリプトといいます。端的にいえば、テストケースを実施可能な段階まで詳細化したものがテストスクリプトです。テストケース管理ツールは、実際にはテストスクリプト管理ツールとして使われることが多いようです。 [↑](#footnote-ref-13)
14. 端的にいえば、開発者の役割はバグが出てこないようにすること、テスターの役割はバグを出すことです。 [↑](#footnote-ref-14)
15. ステークホルダーという語は、ソフトウェア開発以外の文脈でも使われます。 [↑](#footnote-ref-15)
16. 例えば、スクラムという開発手法にはスクラムマスターやプロダクトオーナーといったロールがあります。詳細はスクラムの専門書を参照してください。 [↑](#footnote-ref-16)
17. 余談ですが、オブジェクトプログラミングは英語でobject-oriented programmingといいます。 [↑](#footnote-ref-17)
18. 使いやすいだけでなく、使って楽しいとか、所有する満足感を抱かせる製品の価値を指してユーザーエクスペリエンスといい、UXと略されます。ユーザー体験などと訳されることもあります。 [↑](#footnote-ref-18)