1. ソフトウェアの進化と寿命

|  |
| --- |
| The theory of evolution by cumulative natural selection is the only theory we know of that is in principle capable of explaining the existence of organized complexity.  自然淘汰による進化論は、組織化された複雑性の存在をおおむね説明できる理論としては、我々が知る唯一のものである。  Richard Dawkins (イギリスの進化生物学者) |

# 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。

**進化の形態・動機には大きく3種類ほどあります。**

**どこまで書くかということはありますが、赤下線の部分はどうも複数の種類の記述が混在しているようです。少なくとも書く側は意識的にどれかに統一した方が良いように思います。**

# 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。これは、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たいという動機づけとなります。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。このほか、売れない製品のブランチや、バグをいっぱい含んでいて保守にコストがかさむ製品のブランチにも廃棄したいという圧力がかかります。つまり、生物の進化と同じく、枝分かれした変異種の製品ブランチにも、厳しい生存競争が強いられるのです。これらの圧力に負けなかったブランチだけが絶滅から免れ、進化を許されます。この結果、一番多くのユーザーに使われていて、かつ健康的で保守しやすいブランチだけが、種 (製品) として長い時間生き残ることになります。この一番大事なブランチのことを、トランク (幹) とかメインライン、メインブランチなどとよびます。

**メインライン = 1[ブランチ]とは限りません。**

**緑の部分： 津田さんが言いたいことなのでしょうが、エンジニアリング的に言えばこの比喩は不適切です。どういうストリーム構成にするかor メインストリーム構成は、事前に組織の開発戦略の中で特定されるべきもので、淘汰うんぬんではありません。**

# 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ことがあります。なぜなら、どこかの世代からバグが少しずつ入り込み、製品を蝕んでいってしまうからです。バグの修正や新機能の追加を上手に行わなければ、いつの間にかコード (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り、いつの間にか深刻で修正が困難なバグが入り込んでしまいます。このような状態でバグを無理に治すと、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[[4]](#footnote-4)。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、ついには種としての寿命を迎えてしまうのです。そうなると、また最初から作り直すか、我慢してバグだらけの製品を使い続けるか、どちらかしかありません。

**「上手」と後段は違うレベルの話になっている**

場合によっては既存のコードをすべて捨てて、すべてをゼロから新しく書き直すというアプローチが優れた選択肢になることもあります。開発者は、同じ製品を何度か作り直すことで、それが対象とするドメイン[[5]](#footnote-5)をより良く理解し、より優れたソフトウェア・アーキテクチャを獲得することができるからです。このため、開発初期には捨てることを前提としてコードが書かれることがあります[[6]](#footnote-6)。しかし、優れた開発者を育て、ドメインに関するノウハウを蓄積するという可能性を考慮しないなら、コードを捨てることはその開発に投じた時間やお金を捨てることにほかなりません。

**これは「進化」とは別の話**

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るのです。そのような製品に対抗できる別の製品を手に入れるには、少なくとも同じだけのリソースを同じ時間だけ、かつ正しいやり方で投資し続けなければならないでしょう。あるいは、同様の製品を開発した経験をもった、そのドメインに明るいエンジニアを高給で引き抜くことも必要かもしれません。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。

**そういう場合もあるだろうけど、そうじゃない場合もある。というか、そうじゃない方が多いのでは？　話しの本質はイノベーション（理論）にかかわる**

# 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
* バグの除去と機能の追加は同時に行わない
* 新しくしたビルドを頻繁にリリースする
* 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[[7]](#footnote-7)。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。このように、ソフトウェアは非常に壊れやすいため、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治せます。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不可能になります。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。

**どっかで「非常に壊れやすい」本質的な理由・構造を説明してもらった方がいい**

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。ビルドを頻繁にリリースし、品質が安定したことを確認できた段階で、改めて新機能追加のための変更作業を再開します。複数人で開発するときには、実際にはこれを厳密に行うことは難しく、機能追加と安定化 (バグ修正) の作業はある程度並行して行うことになりますが、肝心なのは同時に多くをしすぎないことです。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れば良いのです。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日から1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。

**安定化 = バグ修正ではないのでは？**

**あと、安定化の概念自体も、先のテストと同様に問題**

**先のポリシーと矛盾している。**

**だったらポリシー自体をこのように変えるべき。**

**ポリシーは便宜的であってはならない**

**この考え方は古い**

**というか、こう考える自体問題**

**※ この本全体、この考え方に貫かれているようですが。**



図1 - 2 漸増的なアプローチを繰り返して行う

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。安定化の段階を省略して、多くの機能をがんがん作りこみ、その後でビッグバン・テスト[[8]](#footnote-8)のような手法でソフトウェアを一気に組み上げるのはうまくいきません。それはいわば、患者さんの全身に大手術をがんがん施して、その後でいきなり立たせてみるようなものです。もちろん立てるはずがありません。

**安定化とビッグバンテストは対立概念ではない**



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)[[9]](#footnote-9) 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」というコンセプトをもっていますがあります。アーキテクチャとは、ソフトウェアの骨格のことです[[10]](#footnote-10)。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、最適なアーキテクチャをはじめから構築するのはとても難しいのです。多くの場合、これは洗練と改善を繰り返しながら成長させていく必要があります[[11]](#footnote-11)。たとえば、これまで絶滅せずに生き残っている動物たちの骨格は、厳しい生存競争に耐え続けた結果、長い時間をかけてそのような骨格に進化しました。いろんな骨格を試した結果、環境に不適だったものは絶滅し、良い骨格をもった生物が生き残ったのです。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。さらに、そのアーキテクチャが本当に優れたものであるかを検証するには、ある程度の時間の試練に耐えることも必要です。

ソフトウェア開発初級者のみなさんには、「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してほしいと思います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。

**違いは？**



**この部分の名称、しっくりこない**

**どっかで説明した方がいいのでは？**

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。  **オモシロイ！！**  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離陸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。準備や計画を怠りすぎるのも、旅行を台無しにしてしまう可能性があるようです。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があるようです。  それにしても、羽田と成田を間違える人が本当にいるとは自分でも驚きです。しかし、そのようなニーズに対応するため、羽田・成田間を高速バスが結んでいることには、もっと驚きます。世の中、いろんな人がいるのですね。 |

# 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。

**これは変更管理等、構成管理の一部。**

**構成管理ツールの構成要素については、もうちょっとちゃんと説明した方が良いのでは？**

## 構成管理ツール

一般に構成管理とは、その製品を構成する要素や部品の変更を管理し、その一貫性を保つことです。いつ、誰が、どこを、なぜ、どのように変更したのか、その履歴をすべてとっておきます。さもないと、製品にバグが出たとき、そのバグがどこからやってきたのか突き止められませんし、それを取り除くことも困難です。

**？？**

ソフトウェア製品は、プログラムによって構成されます。ですから、ソフトウェア開発における構成管理とは、プログラムの変更履歴をすべて取っておき、任意の時点のプログラムを簡単に取り出せるようにしておくことです。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。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

**テスト自動化ツールはとりあげないの？**

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。そう、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、簡単に累計で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決めます割り当てます。。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、段階を踏んで解決します。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。

**対応は計画するのが基本**

**バグ票だけでやりとりするのは素人**

## 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。実行可能なソフトウェア (ビルド) を正しい方向に進化させていくことで、その最終形を手に入れることができます。その進化が正しい方向に向かっていることは、テストで検証します。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。実行可能であれば、その正しさを検証することができます。仕様書から実行可能なビルドと実行可能なテストを生成し、両者を突き合わせることで、そのどちらもが正しいことを検証します。テストケースは手作業で実行するマニュアルケースと、PCが実行する自動化されたテスト (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プロセスについては、6章で説明します。

**おかしい**

**？？**

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、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い、とんちんかんなことをして周りに迷惑をかけてしまいます。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。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マネージャとプロジェクトリードを兼務し、ある人がデブリードと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QAリードとQA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QAを兼ねるときには、この感覚が非常に重要です。開発者とQAでは、その役割が正反対だからです。下記に、それぞれのロールについて説明しましょう。

## 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。チーム全体を方向づけし、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。チーム内に生じたボトルネックを検出して解決を計画するなど、チームの全体最適を考慮した活動を行います。複数のプロジェクトを担当することもあり、その場合は複数のプロジェクトリードから報告を受けます。プロジェクトが佳境に入ってお祭り騒ぎになったときは、音頭をとってプロジェクトを鎮火するのも、プロジェクトマネージャの責務です。

**役割の違い、いまいちはっきりしない**

**デブリードも**

## プロジェクトリード

ユーザーとの折衝を担当します。また、各作業を整理して開発チームとQAチームに依頼します。開発やテストに必要な工数の見積もりも、プロジェクトリードから各メンバーに依頼して、数字を取りまとめます。作業 (開発・テスト) 全体の進捗と各人の作業負荷を見張って、作業量が誰かに大きく偏らないように配慮するのもプロジェクトリードの仕事です。

**この見積りは最悪のやり方**

## 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、開発上のリスクを早期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、最も技術的に明るく、経験の長い開発者が担当します。

## 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。開発者は、当該のプログラミング言語や技術に専門性が求められます。いかにバグを出さないようにするかを工夫することが求められます。

## 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。この図では、開発者の1人がビルドマスターを担当していますが、QAの1人がビルドマスターを担当することもあります。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。

**QAの役割**

## QAリード

QAチームを統括します。QAに指示を出し、作業を割り当てます。

## QA (テストエンジニア、QE、テスター)

**せまい**

**QAの役割**

テストを担当する人ですテストを計画し、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QAあるいはテストエンジニアといいます。また、QEということもあります。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。QAとはQuality Assuranceの略、QEはQuality Engineerの略です。QAとテスターには、いかにバグをたくさん出すか (漏らさず発見するか) を工夫することが求められます。

## SCMエンジニア (Configuration Manager; 構成管理担当者)

SCMサーバを管理する役割を担います。図1 - 5では、QAチーム内に構成管理者を配していますが、開発チーム内の開発者もしくはビルドマスター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。その場合は、チーム内にSCMエンジニアを配置する必要はありません。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。

## バックアップエンジニア

このロールの責務の重要性は、理解しやすいものと思います。ファイルサーバやSCMサーバ内にあるデータを失うことがないように、定期的にバックアップを取ります。図1 - 5ではQAチーム内にこのロールを配していますが、開発者がこのロールを兼務することもありますし、チーム外の社内のIT部門がバックアップのサービスを各プロジェクトに提供することもあります。この責務は、ソフトウェア開発企業の資産を直接的に守るという非常に重要なものです。もし、チーム外のIT部門からバックアップのサービスを受けられないなら、きちんとこのロールと必要なハードウェアをチーム内に準備しましょう。

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 (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。

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[[13]](#footnote-13)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。

**そうなんだけど、それ以前に、計画できちんと役割分担は特定されていなければならない**

ところで、このように、役割分担は最初の計画で特定されているべきですが、だからといって「この仕事は、自分の責務に含まれないから、私はやりません」といった態度は良くありません。そのような考え方は、ソフトウェア開発プロジェクトを失敗に導きます。誰が、どのロールの帽子をかぶっても良いことを思い出してください。あなたの価値は、あなたが周りにどれだけ貢献したかで決まります！

# 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**  ソフトウェア開発プロジェクトを開始するときは、全員でキックオフミーティングをします。サッカーのキックオフと同じく、さあやるぞ、という気合いが入るときです。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するほか、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ます。また、このプロジェクトで初対面の人がいれば、ここで自己紹介をします。  **多くはプロジェクト登録時につけられる**  キックオフミーティングでは、ぜひそのプロジェクトにコードネームをつけて下さい[[14]](#footnote-14)。チームメンバーで話し合って、みんなが好きな言葉をコードネームとします。これにより、チームメンバーは自分たちがプロジェクトを所有するという意識をもち、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かけすぎないようにしましょう。  LotusのAmi Pro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プロジェクト名としていました。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[[15]](#footnote-15)を提供できるように、との願いをこめたものです。あるいは、Lotusのほかの製品には、その製品とは全く関係なく、バージョンが変わるごとに別のカレーの名前をコードネームとした製品や、別のカクテルの名前をコードネームとした製品もありました。  また、Lotus Domino Serverというサーバ製品は、エンジニア達がドミノピザを食べながら夜遅くまで働いて開発したので、ドミノというコードネームがつきました。面倒だからそのまま製品名にしてしまえということで、製品名もドミノサーバーになりました。その後、ドミノピザから商標侵害で訴えられ、賠償金を払って和解したそうです。コードネームは何をつけてもいいのですが、製品名はそうはいかないようです。みなさんも気をつけましょう。 |

# まとめ

本章では、ソフトウェア開発を生物の進化に例えて、これをどのように発展させていくべきかを示しました。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、ソフトウェアを壊さないように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。

最後に、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、QA、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?page=2>

## 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、いくつかを3章で詳細にとりあげます。 [↑](#footnote-ref-3)
4. このようなバグをリグレッション (退化) といいます。リグレッションについては、5章で扱います。 [↑](#footnote-ref-4)
5. ドメインとは業種や分野のことです。ソフトウェアには、財務や会計、携帯電話、自動車、家電、ゲーム、医療など、多様なドメインがあります。 [↑](#footnote-ref-5)
6. この場合でも、すべてを捨てることを前提とすべきではありません。現代は、OSやデータベースなど、ソフトウェアのレイヤ化やコンポーネント化がすすんでおり、すべてを捨てる必要はなくなっています。 [↑](#footnote-ref-6)
7. 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-7)
8. 開発の最終段階になってからシステムを統合してテストする開発手法をビッグバン・テストといいます。現在では、このようなアプローチはほとんどうまくいかないことが知られています。 [↑](#footnote-ref-8)
9. UPについては、7章で紹介します。 [↑](#footnote-ref-9)
10. 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-10)
11. 多くのソフトウェア開発に使われることで、すでに時間の試練に耐えたアーキテクチャもあります。これを再利用すれば、あなたのソフトウェア開発における技術的なハードルを下げることができます。このように、再利用しやすい形をしたソフトウェアの骨組みをフレームワークといいます。例えば、Webアプリケーションのフレームワークには Struts や Ruby on Rails などがあります。 [↑](#footnote-ref-11)
12. このほかにも、ユーザーに納品しなければならない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. 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。 [↑](#footnote-ref-13)
14. キックオフミーティングに先立って、プロジェクトの登録時につけられることもあります。そういうときは、キックオフミーティングでは非公式なコードネームをつけてしまいましょう！ [↑](#footnote-ref-14)
15. 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-15)