7.開発プロセスの構築

|  |
| --- |
| Tom Cruise: Is this our waterfall? ― これ、僕らの滝かい?  Elisabeth Shue: No. ― いいえ、違うわ。  Tom Cruise: It's terrific. ― 素晴らしい。  (映画「カクテル」より) |

# ウォーターフォール型の開発モデル

前章で、実装は仕様書と設計書を書き終えてから始めるべきことを説明しましたが、これってウォーターフォール型の開発ではないかとお感じになった読者もおられるでしょう。しかし、そうではありません。本章ではいわゆるウォーターフォール型のソフトウェア開発について考察し、反復型開発のメリットを改めて明らかにします。

ウォーターフォールモデルとは、手戻りを許さないソフトウェア開発のモデルで、この原典は、1970年にウィンストン・ロイス博士が書いたManaging the Development of Large Software Systemsという論文であるとされています。この論文では、ソフトウェアの基本的な開発の流れは図7 - 1のようになるとしています。連続した段階では、繰り返しが発生していることに注目してください (これは手戻りではありません)。この論文は、連続した段階を繰り返しながらソフトウェアを開発できれば理想的 (だが、少々非現実的) としています。



図7 - 1 ロイス博士が示した、基本的なソフトウェア開発の流れ

この論文はわずか11ページしかなく、図も多く入っていて大変読みやすいものです。ロイス博士は、あくまで個人的な経験に基づいた見方であると前置きした上で、次の3点を述べています[[1]](#footnote-1)。

1. **ソフトウェア開発に基本的なステップとして、分析とコーディングがある。**
2. **ソフトウェア開発を制御するには、そのほかにシステム要求定義、ソフトウェア要求定義、プログラム設計、テストなどが必要である。**
3. **この基本的なフレームワークは危険で失敗を招きやすい。最後のテストの段階でさまざまな問題が明らかになるからだ。結局、要求を変更するか、設計を変更するしかない。**

つまりこの論文は、図7 - 1のモデルは危険で失敗を招きやすいとしているのです。現実には、連続した段階だけでなく、図7 - 2に示すように、離れた段階においてもこのような繰り返しが発生し、スケジュールやコストを倍にすることもあるとしています。



図7 - 2 設計の反復は  
連続した段階の反復からはみ出してしまう

しかし、ロイス博士はこれがソフトウェア開発への基本的なアプローチであるとしており、これを改善する手段として下記の5点を挙げています。

**Step 1: 最初にプログラム設計を行うこと**

**要求生成段階と分析段階の間に予備的なプログラム設計を行えば、早期に問題を明らかにできる。最終的な設計とコーディング、テストを開始する前に、要求と予備的設計を反復してやり直せる。**

**Step 2: 設計を文書化すること**

**初期段階では、文書が設計そのものだからだ。この文書がコミュニケーションの手段となるし、保守チームによる修正もこの文書でサポートできる。**

**Step 3: 同じことを2回行うこと**

**同じようなソフトウェアを開発した経験がないなら、最終的に顧客に渡すのは2番目のバージョンになるようにすべきだ。**

**Step 4: テストの立案、管理、監視を行うこと**

**プロジェクトのリソースを最も消費するのは、テストフェーズである。上記3つのステップはテストの前に実施して問題を明らかにし解決することを狙ったものだが、それでもテストでやるべきことはたくさん残っている。**

**Step 5: 顧客を巻き込むこと**

**ソフトウェアで何を実現したいのかは、予め合意した後でも解釈に幅が出やすい。最終的な引渡しより前の段階から、顧客に参加してもらうことが重要だ。**

この論文には、各ステップを実践した場合に、図7 - 2のモデル図がどのように変化するか、ステップそれぞれに改善されたモデル図を表示しています。そして、巻末にすべてのステップを実践した場合のモデル図を示して結んでいます。何とも示唆に富んだ内容で、Step 5などはXPのプラクティスのひとつ「オンサイト顧客」そのままです。今から40年も前に書かれたものとは思えないくらいです。

しかし、この論文で示された洞見は、事後の変更は法外に高くつく、手戻りは避けるべしというハードウェア製造の常識により曲解されてしまいます[[2]](#footnote-2)。つまり、後で変更する必要がないように、上の段階からきっちりと終わらせていく逐次的なプロセスとして、図7 - 3のように実践され、ウォーターフォールモデルという名前で世間に認知されました。



図7 - 3 誤って実践されたウォーターフォール型モデル

実は、この図もロイス博士の論文の先頭に (単にソフトウェア開発に必要となる作業を抽出して順に並べただけのものとして) 載っていますが、もちろんロイス博士はソフトウェアをこのように開発するべき、とは言っていません。また、このロイス博士の論文の中にはwaterfallという語は出てこないのですが、確かにこれはそう呼びたくなる形をしています。おそらく世界で最も大きいソフトウェア調達機関である米国防総省 (DoD; United States Department of Defense) が、このような開発モデルを1980年代に標準としてDOD-STD-2167に規定したことが、問題を大きくしました。2167は、イギリスのJSP-188やドイツのVモデルの策定時、さらにはオーストリアやスイスでの標準ソフトウェア開発モデルの策定時に参照され、大きな影響を及ぼしたといいます。このモデルによるソフトウェア開発プロジェクトを完了させるには、ロイス博士が示した5つのステップを追加で実施するよりも多くの費用がかかったことでしょう。ソフトウェアは反復的に開発すべきことを訴えたロイス氏の論文は、その意図とはまったく逆の影響を世界に及ぼすことになったのは皮肉なことです。

1987年の初めには、米国防総省はウォーターフォール型モデルを使うべきでないと注意を喚起し、反復型のモデルを推奨するという勧告を出しました。1994年には、反復型開発による標準が2167に代わって使われるようになっています。このあたりの事情については、書籍「初めてのアジャイル開発」に詳しく述べられています。

# ウォーターフォール型モデルの問題点

結局、図7 - 3に示したウォーターフォール型モデルは、この図を最初に書いたロイス博士によってその失敗が予言されていたようなものです。ロイス博士の指摘は、コンピュータの性能の向上により古くなってしまったものもありますが、その多くは今日のソフトウェア開発にもまだ有効です。

ウォーターフォール型の開発プロセスは、すべての詳細設計を紙上で完了してから一度に実装し、その後でまとめて統合してテストする、という流れですすみます。つまり、直前の段階がきちんと完了するまで、次の段階を開始しない (できない) としているのです。このようなアプローチをBDUP (Big Design Up Front) といいます[[3]](#footnote-3)。このモデルは簡潔でわかりやすく、かつソフトウェア開発の全体の進捗が管理しやすいと思われたので、広く使われるようになりました。しかも、これは常識的な感覚にまったく反しないものです。まず何が必要かを見つけだし、これをどのように実現するかを決め、次に実装し、正しく実装できたか確認し、最後にこれを運用する、という手順は、とても真っ当なものにみえます。

しかし、ユーザーの要求が複雑になり、コンピュータの資源が潤沢になり、ユーザーインターフェイスが高度化し、プログラミング言語の表現能力が向上し、ネットワークの帯域幅が広がり、高度な並列処理が一般的になり、といった変化の中で、このモデルはほとんど使いものにならなくなってしまいました。最終的に要求される成果物としてのソフトウェア製品が複雑になるにつれ、ロイス博士が論文の中で表明した懸念がますます現実のものとして私たちの前に立ちはだかっています。

端的にいえば、ウォーターフォール型モデルの問題点は2つあると考えられます。ひとつめは、ソフトウェア開発に潜在するリスクがテストの段階まで表面化しないことです。これは特に、これまでに同様なソフトウェアを開発したことのない組織において顕著です。すでに「1回目」の開発を経験済みの組織では、そのリスクの多くを予期できるようになっているかもしれません。

ふたつめは、その極端なドキュメント駆動のアプローチにあります。どんなに丁寧に時間をかけて設計仕様書を書いても、複雑なソフトウェア開発に潜むリスクをすべて洗い出すことはできません。手戻りを恐れて上流段階を丁寧すぎるくらいに実施しても、やはり予期できなかったリスクが下流段階で表面化し、手戻りが発生します。しかも、上流段階の「丁寧に」という努力が、すぐに陳腐になってしまうドキュメントの品質を高めるという方向に向けられていたとき、後でそのドキュメントを修正するコストが莫大なものになり、取り返しがつかなくなります。例えば、プログラムと同じ程度に詳細な仕様書を自然言語 (日本語など) で書き、レビューミーティングを開催して時間をかけて「てにをは」を直すなどは笑止千万、お腹の片側が痛くなります[[4]](#footnote-4)。どうせ同じものを書くなら、始めから仕様書をJavaなどのプログラミング言語で書き、インデント[[5]](#footnote-5)をツールで自動的に直す方がよほど生産的です。

では、自然言語による仕様書は書くべきではないのでしょうか？いいえ、それも違います。確かに、書くべきドキュメントの量は以前より減りました。しかし、ロイス博士の論文が示唆するように、自然言語で書かれた仕様書は製品のビジョンを示し、チームを方向づけ、ユーザーとコミュニケーションを取るという重要な役割を担っています。

# コラム 仕様と実装を近づける試み

|  |
| --- |
| **コラム 仕様と実装を近づける試み**  ソフトウェア開発の話題で必ず出てくるのが、ドキュメントをどこまで書くべきかという問題です。最終的に欲しいものは実行可能なソフトウェアですから、実行できない仕様書は中間成果物でしかないという理屈も成立します。また、実行可能でないものは検証できないのだから、仕様書だけではその正しさも検証できないことになります。  しかし、実行可能なものが手元にあったとしても、それが全く保守できないものなら所有しているとは言えません。バグを治すこともできず、新機能を追加することもできないソフトウェアには、製品としての価値はほとんどないからです。  図7 - 4に、ソフトウェア開発における文書のモデルを示します。最初に人間に読みやすい仕様書で概要を記述し、これをコンピュータが解釈できる実行可能ファイルとして詳細化します (ただし、図7 - 4に示したテストケースには、コンピュータが直接実行できる自動化されたテストと、人の手で実行するマニュアルケースの両方が含まれます)。必要に応じて、仕様書は実行可能ファイルと整合性をとり、修正 (同期) していかねばなりません。  C:\idd\doc\spec.png  図7 - 4 設計書と実行可能ドキュメントの関係  このようなモデルで問題となるのは、次のような点です。   * 仕様書は実行不可能なので、その正しさを検証できない。 * 仕様書には概要しか書いていない。詳細な仕様を調べるには、実行可能ファイルを読む必要がある。 * 仕様書を最初に詳細にしすぎると、後でそれが間違っていたと判明したときの損害が大きい。 * 詳細な仕様書は、実行可能ファイルを変更したときに、それに合わせて変更する (同期を取る) のが大変。   そこで、自然言語による仕様書 (設計書) を自動的にプログラミング言語に変換 (翻訳) しようとか、自然言語による設計書そのものを直接実行可能にしてしまおうという試みがなされてきました。そのひとつがMDA (Model Driven Architecture; モデル駆動アーキテクチャ) です。MDAは、コンピュータに依存しない形でモデルを記述したら、これをさまざまなコンピュータのプラットフォーム上で実行できる形に自動変換するというコンセプトをもつソフトウェアの設計手法です。しかし、当初MDAの標準仕様を策定しようとしたOMGという団体の目論みは不成功に終わり、現在はさまざまなベンダがそれぞれのツールをリリースしており、使いにくい状況にあります。また、同じような取り組みとしてExecutable UML (実行可能UML) というのもありますが、やはり広く使われるには至っていません。  比較的うまくいっているものに、ラウンドトリップエンジニアリングがあります。モデル図からプログラムを自動生成するのをフォワードエンジニアリングといい、既存のプログラムからモデル図を自動生成するのをリバースエンジニアリングといいます。このふたつを双方向に行うのが、ラウンドトリップエンジニアリングです。フォワードエンジニアリングは、組み込み機器用のソフトウェアの状態遷移を管理するプログラムなどでうまくいっているようです。また、リバースエンジニアリングは、既存のプログラムの理解を助けることがあります。ただし、実際に双方向にラウンドトリップするアプローチがうまくいっている事例は、少なくとも私の周りでは聞きません。  私の主観 (偏見) ですが、既存のプログラミング言語 (テキスト) で表現できる単位あたりの情報量に対して、いわゆるモデル図 (絵) で表現できる単位あたりの情報量は少なすぎますし、モデル図で厳密な文法や意味を表現するのも困難ではないかと思います。そもそも図表が分かりやすいのは、良い意味で詳細を省けるからです。モデリングツールで実行可能なソフトウェアを構築したいという要求には、建築物をCADシステム[[6]](#footnote-6)で設計するスタイルにあこがれる気持ちがあるのかもしれません。しかし、ソフトウェアをCADシステムで作れるようになるとは私には思えません。  これに対して、文字で書かれたプログラムなら、厳密な文法に従えますし、ツールによるリファクタリングのサポートなども容易です。  成功する可能性が高いものとして、DSL (Domain Specific Language; ドメイン固有言語) をあげておきます。これは、開発したいソフトウェアのドメインに合わせて、特別にあつらえたプログラミング言語を使って、テキストで仕様を記述するというアプローチです[[7]](#footnote-7)。たとえば、正規表現[[8]](#footnote-8)は文字列検索に特化したDSLであると考えられます。ドメインを限定することは、コンピュータにはメリットが大きいのです。コンピュータは、文脈を推論するのがとても苦手だからです。例えば、自然言語の機械翻訳 (英語から日本語など) は、まだまだ品質の面で人間の翻訳者には遠く及びません。しかし、天気予報というドメインでは、機械翻訳は素晴らしく成功しているそうです。これは、天気予報という文脈に現れる文章がワンパターンで、誤解が少ないからです。  ただし、あるドメインに固有のDSLそのものを開発するには大きなコストがかかります。今のところは、JavaやC#のような汎用のプログラミング言語を使うのが現実的な解決でしょう。  ところで、文脈により自動的に振る舞いを変えるソフトウェアが身近にあります。それは、Windowsの右クリックによるポップアップメニューです。これは、クリックした場所 () によって表示するメニュー項目を自動的に変えるので、コンテキストメニューともよばれます。  実行可能なドキュメントで素晴らしくうまくいっているのは、3章で紹介したassert文です。assert文だらけのプログラムはとても魅力的です。 |

# 反復型の開発プロセス

前節で述べたウォーターフォールモデルの2つの欠点を解決するには、どうしたら良いのでしょうか。「テストの段階までリスクが表面化しない」なら、「早い段階からテストをする」べきだと分かります。また、「詳細なドキュメントは修正が大変」なら、「初期段階では、ドキュメントを詳細にしすぎない」ことが、直接的な解決になるはずです。これを具体的に実施するには、なるべく早い段階で最初のビルドを獲得すること、これをテストしながら少しずつ機能を詳細化し実装することが必要です。これが反復型の開発モデルです。

反復型開発プロセス (iterative development process) とは、反復しながらソフトウェアを開発するというものです。これは決して新しいものではありません。1960年代には、すでに短い反復を繰り返しながら開発した事例がありました[[9]](#footnote-9)。一般に、反復型の「反復」とは、1ヶ月～数か月おきのマイルストーンで区切られた期間を指します。しかし、本書で示したように、ソフトウェア開発において繰り返すべき作業はたくさんあり、さまざまな反復が階層構造をつくっています。その中でも、ソフトウェア開発をスムーズに回すためには、「ビルドのリリース」を一番小さな単位として認識することが重要だと私は考えています。ビルドマシン上でつくる統合ビルドが、チームで行う反復の一番小さな単位だからです。今ほどコンピュータの処理速度が速くなかった頃から、ナイトリービルドやウィークリービルドといったアプローチが使われてきました。現在はPC資源が潤沢になったため、より頻繁にビルドを行うことができるようになっていますが、やはり定期的なビルドもちゃんと計画していただきたいと思います。定期的なビルドを使えば、タイムボックスの一番基本的な単位を簡単に構成できます。

反復型の開発プロセスの誤った説明に「ウォーターフォールを繰り返せば反復型開発になる」というのがあります。しかし、このような説明を聞いて、「反復型開発を実際に行うなんてできるはずがない」と考えた方も多いのではないでしょうか。反復の単位が1ヶ月もしくは1週間だとすれば、反復型開発は次のようなものになるからです。



図7 - 5 1ヶ月で反復する計画カレンダー??



図7 - 6 1週間で反復する計画カレンダー??

もちろん、こんなスケジュールでソフトウェア開発を前に進めることはできません。実際には、最終的に実現したいソフトウェアの機能をいくつかに分割し、それぞれで同じ手順を繰り返しながら並行して少しずつ機能を追加していくことになります。図7 - 7に、反復型開発で機能を追加していくようすを示します。



図7 - 7 機能を適切に分割し、少しずつ追加する

まず、機能仕様書を記述することで各機能を定義します。レビューを通したらスペックコンプリートとなり、実装を開始できる状態になります。機能を実装し、開発を終えることをコードコンプリートといいます。ただし、テストの開始を満たせば、コードコンプリートを待たずに並行してテストを開始できます。コードコンプリートの後も、テストと修正のサイクルが続きます。この段階を安定化といいます。図7 - 7では単純化して示していますが、機能の大きさにより開発にかかる期間は大きく異なります。3日で実装できる機能もあれば、数週間かかる機能もあるでしょう。

最初のビルドを獲得するまでにも、数週間かかることがあります。最初のビルドができたら、ビルドを毎日もしくは毎週といった頻度で定期的にリリースできるようになります。

# 反復の構築

反復型開発では、2週間から数ヶ月程度を開発を進捗させる単位とします。この期間を指して「反復」(iteration) といいます。これを「マイルストーン」(milestone) ということもあります[[10]](#footnote-10)。また、スクラムというソフトウェア開発手法では、この反復を指して「スプリント」(sprint) といいます。それぞれの反復の最後には、(機能は不完全だが) ある程度安定したソフトウェアをビルドすることをゴールとします。また、これをユーザーにリリースして評価してもらうこともあります。もちろん、この反復の最後だけでなく、期間中にも毎日もしくは毎週といった頻度で統合ビルドを作ります。

このような反復を計画するときには、それぞれの反復の開始日と終了日を明確に決めて、ソフトウェア開発計画書にのせます。例えば、6月の第1月曜を1回目の反復の入口 (entrance)、第4金曜をその出口 (exit) などとします。また、反復を開始するための開始条件 (enter criteria) と終えるための終了条件 (exit criteria) を設けます。もちろん、最初のソフトウェア開発計画書で、すべての反復のcriteriaを設定するのは難しい[[11]](#footnote-11)のですが、各反復を開始するときには、達成可能なゴールをその終了条件として設定しましょう。このように、期間を固定した範囲に区切ることをタイムボックスといいます。大きなタイムボックスを小さなタイムボックスに分割し、それぞれで見積もりや実績の管理をすることで、より確実な進捗を得られます。

反復を終えたとき、もしexit criteriaにmeetしていなかったらどうすればいいのでしょうか。いくつか選択肢があります。

1. 満足できなかったものは、残り作業として次の反復での実現をめざす。
2. スケジュールを延ばして、残りの反復計画も引きなおす。
3. プロジェクトの中断を宣言する。

つまり、その反復での実現をあきらめるか、設定した終了条件に達するまでその反復を続けるか、どちらかが現実的な選択肢となります。終了条件を満足していないのに、その反復を終えることには抵抗があるかもしれません。しかし、現在ではスケジュールを調整するよりも、スコープ (反復で実現する機能の範囲) を調整する方が優れたアプローチだといわれています。つまり、最初に設定した終了条件に無理があったのです。多くの場合、それは次の反復の終了条件をより適切にするためのインプットとして、その反復は予定通り終えるべきです[[12]](#footnote-12)。これがタイムボックスの考え方です。タイムボックスは、より現実的なトリアージとプリオリタイズをチームに要求します。また、チームの集中力を上げ、生産性を向上させます。

もし、最初に設定したcriteriaとあまりにかけ離れているようであれば、これは異常事態です。プロジェクトの中断を宣言し、今後をどうするのか、話し合わなくてはなりません。原因を明らかにできればそれを取り除き、計画を引きなおして再開します。あるいは、プロジェクトの中止も視野に入れて検討する必要があるかもしれません。

Exit criteriaとして適切なのは、もちろん実行可能なビルドで測れるものです。例えば、予定していたすべての機能の実装を終えるとか、深刻度が1のバグが0件になっているとか、これだけのパフォーマンスを実現するなどをexit criteriaとして設定すると良いでしょう。開発するソフトウェアのドメインによっては、より具体的なcriteriaを設定することもできると思います。各マイルストーンで設定すべき適切なexit criteriaを探してみてください。

# 反復型開発のメリット

本書では、ウィークリービルドによる1週間をタイムボックスとした反復型開発の実践について説明しました。改めて、この方法によるメリットをまとめておきます。

## 早期にリスクを発見する

早い時期に (機能は不完全だが) 出荷可能な製品をビルドすることは、予測できなかったリスクを早く発見させます。

## 最終段階での統合のリスクを軽減する

ウォーターフォール型では、「地獄の統合テスト」などという言葉で、このリスクが説明されます。反復型の開発モデルは、このリスクを低減します。

## プロジェクトを分割する

反復型開発は、大きくて複雑なプロジェクトを小さくて単純なプロジェクトに分割します。

## 頻繁なフィードバックを得る

ユーザーからのフィードバックにより、最終的にユーザーに引き渡される製品が、ユーザーの要求に沿ったものになる可能性を大きくします。

## 達成可能なゴールと達成感の両方を得る

プロジェクトのメンバーは、小さなゴールを継続して達成することで、成功体験と自信を得ます。また、ユーザーからは信頼を得ます。

## 正しい進捗を知る

ウォーターフォール型のモデルでは、解決すべきリスクが後になって表面化するため、特に初期段階では正しい進捗を得られません。

## 継続してプロセスを改善する

1週間おきのふりかえり (レトロスペクティブ) は、チームでの開発の手順を継続的に改善します。

## 納期通りに製品を出荷する

製品のスコープを調整することで、スケジュール通りに製品を出荷します。

## 変化を早く引き起こす

ソフトウェア開発に変化は避けられません。反復型開発は、そのような変化を早く引き起こすことを促します。

## 必要な機能だけを開発する

ソフトウェアに含まれる機能の多くは、実際には使われないことが多くあります。反復型開発を使えば、必要な機能だけを開発できます。

このように、反復しながらインクリメンタルにソフトウェアを進化させるアプローチは、正しいソフトウェアを正しく作るための有力な手段なのです。



図7 - 8 ソフトウェア開発プロジェクトに  
ありがちなイベントとタイムボックス (例)

# コラム ソフトウェア開発のメタファ⑦ お料理

|  |
| --- |
| **コラム ソフトウェア開発のメタファ⑦ お料理**  ソフトウェア開発管理の名著「人月の神話」では、プログラマをレストランのコックさんにたとえて、ソフトウェア開発のスケジュール変更がひどい結果になることを説明しています。コックさんが「2分でできます」と顧客に約束したオムレツが実際には出てこなかったとき、顧客には2つの選択肢があります。つまり、出てくるまで待つか、生焼けで食べるかのどちらかです。ソフトウェアの顧客も、同じ選択を迫られてきたといいます。また、コックさんは火を強くすることもできますが、それでは生焼けの焦げたできそこないのオムレツができるだけだとしています。  このほか、スコット・バークン氏の著書「アート・オブ・プロジェクトマネジメント」は、忙しい時間のレストランの厨房における複雑かつ迅速なタスクマネジメントには見習うべき点があるとしており、著名な料理人アンソニー・ボーデイン氏のノンフィクション「キッチン・コンフィデンシャル」を読むことを勧めています[[13]](#footnote-13)。これは、若かったボーデイン氏がさまざまな厨房で経験を積み、荒々しいとして一流になるまでの物語です。  私自身はレストランでアルバイトをしたことがないので、厨房の慌しさというのはあまり想像できないのですが、ドメインのを料理から家事全般に広げると、非常に納得できる部分があります。私は今、妻と共働きをしていますが、娘が産まれるまでは、朝がこんなに忙しいものだとは知りませんでした。どちらか片方は必ず娘を見ていなければならない、お風呂掃除とゴミ出しは私がする、離乳食作りは妻がする、洗濯物を取り込んで畳んだり、洗濯機を回して干したり、というのは娘を見ながらでもある程度こなせる、娘に食事を与えて自分たちも食事をとる、娘に食事を与える間はテレビを消しておかねばならない (娘の気が散って食べられない)、しかし可能なら朝の連続テレビ小説は見たい、顔を洗って歯を磨く、出勤の前に保育所に行って娘を預ける、といった多くの作業と付随する制約があるため、これらをパイプライン化して効率よく並列処理しなければ、とても出勤時間に間に合いません。また、家事においても、成功の秘訣はオートメーションであろうと思います。現代は、炊飯器、洗濯機、お風呂沸かし器、電気ポット、食器洗い機、トイレのウォシュレット、ルンバなどのツールにより、家庭でも自動化できる作業が大変多くなっていますね。これらのオートメーションツールを支える上下水道や電気などのインフラにも注目すれば、ソフトウェアの開発環境との類似性が見つかります[[14]](#footnote-14)。  このほか、2章で紹介したドッグフードも、料理のたとえといえば言えるかもしれませんね。 |

# アジャイルな開発プロセス

昨今、反復型開発をさらに一歩進めたものとして、アジャイルな開発プロセスが注目を集めています。Agileとは機敏で柔軟な、という意味です。2001年、XPやスクラムなどの軽量な開発プロセスを主導してきた17人が、アメリカのユタ州のスキー場で会合を開き、それぞれの開発手法の重要な部分を統合して、「アジャイルソフトウェア開発宣言」としてまとめました。この宣言自体は大変短いものですので、ぜひ一度読んでみてください[[15]](#footnote-15)。アジャイルソフトウェア開発宣言のそれぞれは、特に目新しいものではなく、ずっと以前から言われてきたことです[[16]](#footnote-16)。しかし、それらがひとつにまとめられたことで、ソフトウェア開発業界に新しいムーブメントを起こしました。

下記に、代表的なアジャイル開発手法とされるXPとスクラムについて紹介します。

## XP

XPとはExtreme Programmingの略[[17]](#footnote-17)で、「変化を受け入れる」というスローガンを掲げた開発プロセスです。ケント・ベック氏らによって提唱されました。技術的な知識と理解を「プラクティス」、全体を捉えて良し悪しを判断する根拠を「価値」、そのふたつの隔たりを埋めるものを「原則」とし、いくつかの価値と原則、そして多くのプラクティスを規定しています。ソフトウェア開発に良いと思われることは、すべてに実施するのがXPの態度です。たとえば、レビューは良いものだから、ペアプログラミングによってずっとレビューをし続けるべきだとしています。

初期のXPは、どちらといえば有益なプラクティス集といった趣きが強く、開発手法とはいえないという人もいました。また、「極端なプログラミング」という名前が示すように、XPには開発者が心地よく効率的に作業をするという側面が多分にあります。ただし、現在のXPには、ユーザーからの要求をストーリーカードと呼ばれる単位に分割すべしとしているなど、プログラミング以外のソフトウェア開発の側面についても規定する部分があります。これは、XP自身が常に変化を受け入れて進化し続けてきたからです。取り入れやすいプラクティスから、部分的・段階的にみなさんの組織に取り入れていくことが容易ではないかと思います。

## スクラム

スクラムは、ラグビーのスクラムチームにちなんだ開発手法です。スクラムという語が製品開発の文脈に初めて登場したのは、経営学者の野中郁次郎先生と竹内弘高先生が1986年に著した論文「The New New Product Development Game」においてです。これはソフトウェア開発に関する論文ではありませんでしたが、これに感銘を受けたケン・シュエイバー氏らがつくりあげたソフトウェア開発手法がスクラムです。文字通り、チームワークを重視したプロジェクト管理のフレームワークであり、人びとを継続的に統合する取り組みであると言えるかもしれません。

スクラムでは、スプリントとよばれる一か月間の反復をどのように扱うかに重点が置かれています。例えば、各スプリントにおいて、スプリント計画ミーティングやスプリントレビュー、レトロスペクティブなどの実施を規定しています。さらに、残りのタスク (バックログ) を、最終的な製品で実現すべきプロダクトバックログと、このスプリントで実現すべきスプリントバックログに分けて管理し、各タスクには厳密な完了 (“Done”) の意味を定義しておくことが必要だとしています。特に、スプリント中は製品に新機能の追加や変更を許さないこと、また各スプリントの最後には出荷可能な製品を作ることは、注目に値します。

## UP

UPとはUnified Process (統一プロセス) の略で、反復型開発プロセスのフレームワークです。テーラリング (カスタマイズ) されてさまざまな組織やプロジェクトに合わせることが意図されており、実際にUPを特殊化したものにはAUP (アジャイル統一プロセス)、BUP (基本統一プロセス)、EssUP (本質的統一プロセス)、EUP (エンタープライズ統一プロセス)、OpenUP (オープン統一プロセス)、OUM (オラクル統一メソッド)、RUP (ラショナル統一プロセス) など、多数あります[[18]](#footnote-18)。AUPはスコット・アンブラー氏により、EssUPはイヴァー・ヤコブソン氏により提唱されました[[19]](#footnote-19)。

UPでは、開発工程全体を方向づけ、推敲、作成、移行の4つのフェーズに分けています。これがウォーターフォール型モデルを想起させるのか、逐次的なプロセスであると誤解されることがありますが、実際には各フェーズは複数の反復から構成されています。

また、UPでは約50種類の作業成果物が定義されています。このため、UPは重たいとか、アジャイルでないという不当な批判を受けることがあります。しかし、これらの作業成果物にはリリースノートなど、ほかのプロセスを使ってもたいがい作成するようなものも含まれており、この中から必要なものだけを使えば良いとされています。このような共通の作業成果物が定義されていることは、用語の意味も統一されていることを意味します。これは、比較的規模が大きな開発では有利にはたらきます。

XPとスクラムは、ソフトウェア開発に注目する視点がかなり直交しています。XPはプログラマの視点に立ち、スクラムはマネージャの視点に立っています。それでいて、このどちらにも含まれるプラクティスもあります。たとえば、ベック氏はデイリースタンドアップミーティングをスクラムからXPに取り入れました。このため、両方を同時に実践することも難しくありません。UPは、より包括的なソフトウェア開発のメソッドを提供します。これらには、ソフトウェア開発に関する優れた知見が含まれています。ぜひ学習してみてください。

# プロジェクトをドライブする

ドライブ(drive)とは、駆動する、前へ進める、などの意味です。ドライバとは何かを駆動するもの、という意味ですが、何を駆動するかによって意味が異なります。まず、これらの意味を整理しましょう。

1. 運転手さん  
   ハンドルを握り、車を駆動させる係の人をドライバーといいます。
2. ねじ回し  
   ねじを回す道具を、スクリュードライバーといいます。
3. デバイスドライバ  
   パソコンの周辺機器(デバイス)を駆動させるためのソフトウェアをデバイスドライバといいます。
4. テストドライバ  
   単体テストをするときにテスト対象とするクラスを呼び出して動かすモジュールをテストドライバといいます。これは6章で紹介しました。
5. プロジェクトのドライバ  
   ソフトウェア開発プロジェクトを前へ進める係の人を指してドライバといいます。

本節で説明したいのは、最後のドライバです。プロジェクトマネージャが、ソフトウェア開発プロジェクトをドライブします。といってしまうと、「プロジェクトを前へ進める」というのがちょっと曖昧すぎて意味がわからないかもしれません。そこで、プロジェクトをドライブするためのヒントを与えましょう。端的にいえば、プロジェクトをドライブするとは、ソフトウェア開発に必要なイベントを特定し、それらをドライブする担当者を任命する、ということです。

例えば、本書ではソフトウェア開発に必要なイベントとして、ドッグフードやバグバッシュを紹介しました。ドッグフードのドライバは、criteriaとして設定した以上の人がドッグフードを行うように、ドッグフードプログラムへの参加方法をチームに案内したり、現在の参加人数をチームに通知するなどして、より多くの人がドッグフードをするように促します。バグバッシュのドライバは、スケジュールやテスト対象とするビルドの番号、バグのファイル方法などをチームに案内し、バグバッシュが終わればその結果を集計してチームと共有します。

このように、プロジェクトマネージャは、チームメンバーの中から、各イベントの運営を担当する人をドライバとして任命すると良いでしょう。持ち回り制にするのも良い考えです。何をドライブすればいいのかわからないなら、その何かを適切な単位(イベント)に分割すれば良いというわけです。もちろん、みんな忙しくて誰にも頼めそうにないなら、プロジェクトマネージャ自身が各イベントをドライブしてもいいでしょう。例えば、キックオフやポストモータムのようなイベントはプロジェクトマネージャ自身がドライブすべきです。また、イベントによっては、複数の人をドライバとして任命しても良いでしょう。任命された人は、必要に応じてステアリングコミッティ(運営委員会)を開催し、当該のイベントを正しく方向づけ、ドライブします。ステアリング(**steering**)とは、ハンドル操作とか舵取りという意味です。ステアリングコミッティとは、まさしくイベントをドライブするための会議というわけです。

# 次のステップへ

本書では、コミットのプロセス、ビルドと日次もしくは週次のリリースプロセス、バグの追跡プロセス、テストケースの追加プロセスなど、反復型のソフトウェア開発に重要なプロセスのいくつかを紹介しました。最後に、読者のみなさんがさらに前へ進むためのヒントを示し、本書を締めくくりたいと思います。

## FragileからAgileへ

Fragileとは、壊れやすいという意味です。本書では、ソフトウェアが壊れやすいことを前提にして、いかにソフトウェアを壊さずに進化させていくべきかを説明しました。本書で示したアプローチを使えば、プロジェクトリードが予期しない形で開発中のソフトウェアが壊れてしまうことはなくなるでしょう。このアプローチの上に、XPやスクラムなどのアジャイルといわれる開発手法を構築していくことは容易です。ぜひ、学習してみてください。

## ツール重視からコミュニケーション重視へ

本書では、ツールを重視したソフトウェア開発の手順を示しました。しかし、アジャイルソフトウェア開発宣言には、「プロセスやツールよりも個人と対話を価値とする」という一節があります。もちろん、これはツールを使わないという意味ではありません。何のためにプロセスやツールの運用ポリシーがあるのかを理解できないうちは、それに厳密に従うべきですが、それが腑に落ちれば、必要以上にツールの運用ポリシーに縛られる必要はありません。もし、あなたがツールに使われたり、規則に振り回されたりすると、本来果たすべきミッションを見失ってしまいます。ツールとプロセスは柔軟に使い、アジャイルな開発を実践できるようにしましょう。

## 手作業から自動化へ

ソフトウェア開発の歴史は、自動化の歴史といってもいいでしょう。アセンブラから始まり、コンパイラ、lint、ビルドツール、構成管理ツール、バグ追跡システム、ガベージコレクタによるメモリ管理、統合開発環境によるリファクタリングの支援、ラウンドトリップエンジニアリング、テストの自動化、ビルドサーバ、PCの仮想化による複数のテスト環境条件の構築など、利用できるツールは以前よりリッチになってきています。より広い範囲の作業を自動化すれば、人はより知的な作業に専念できるようになります。

トヨタ生産方式では「にんべんのついた自働化」といって、自動化には自働化という字をあてています。合理化を進めるあまりに従業員の人間性ややる気をそぐことはしないという考えからです。これには、アジャイルソフトウェア開発宣言と重なる部分があるように思います。

## 単能工から多能工へ

多能工とはトヨタ生産方式で推奨されるアプローチで、1人が複数の工程の作業をこなせるようにトレーニングすることです。これはソフトウェア開発にも有効です。本書では、ソフトウェア開発に必要な責務 (ロール) をいくつか紹介しました。これらの複数のロールを全員がこなせるようになれば、より適切な負荷分散ができます。ただし、多能工を実現しても、全員のロールが「何でも屋さん」に一本化されるわけではないことに注意してください。多能工の人は、今自分がどのロールを果たしているのか、しっかり意識しています。

## ビルド中心からアーキテクチャ中心へ

本書では、ビルド中心というアプローチを示しました。しかし、これは一歩間違えば、Build-and-Fixモデルというウォーターフォールよりも素朴で原始的なプロセスになりやすいのです。これは、Ad-hocモデル (いきあたりばったりモデル) としても知られています。



図7 - 9 ビルドして修正モデル[[20]](#footnote-20)

6章でも述べたように、テストとは、ソフトウェアの品質に関する情報を収集するプロセスであって、品質を作りこむためのものではありません。開発してビルドしてテストして、不具合を見つけたら治す、というプロセスは、ソフトウェア開発においては非常に基本的で重要なものですが、プロセスとしては非常に素朴で原始的なものです。ソフトウェアの品質を作りこむ責任を担うのは、あくまで開発者であり、アーキテクチャ中心という考え方です。

設計で品質を保証せずにテストに突入し、コーディングとテストを繰り返しながら開発をすすめる、そんなやり方は原始的です。ビルド中心というアプローチが腑に落ちたら、次はソフトウェアの高度な設計技術を身につけて、アーキテクチャ中心のソフトウェア開発を目指しましょう。

## より早い段階での不具合の検出へ

本書では、チームでの開発を支援するツールだけを取り上げ、テストの自動化のためのツールについては説明しませんでした。また、本書で説明したテストケース管理ツールは、一般に統合テストのためのテストケース (テストスクリプト) を管理するものがほとんどなので、本書でも統合テストの管理だけを扱いました。

しかし、実際には、もっと多くのテストを活用していく必要があります。テストは静的検証と動的検証に分けられます。統合テストは動的検証に分類されます。アーキテクチャ中心とも関連しますが、統合ビルドをビルドしてテストするというだけでは、やはり原始的なアプローチであろうと思います。動的検証は単体テスト、統合テスト、システムテストと階層化して管理し、静的検証についても仕様のレビューや種々のテストツールを活用していくのが、より現代的なアプローチです。

# まとめ

本章では、ウォーターフォール型モデルの出自を明らかにし、このモデルを示したロイス博士のソフトウェア開発に関する知見を紹介しました。また、ウィークリービルドよりも大きい単位のタイムボックスを示し、反復の構築方法を紹介しました。反復の構築方法についてより深く知りたい方は、スクラムを学習することを強く勧めます。本章の最後では、本書の締めくくりとして、先へ進むためのヒントを示しました。みなさんが本書を参考にして、それぞれの開発プロセスを構築されることを願っています。

# 付録 ソフトウェア開発で使われる用語

|  |
| --- |
| **付録 ソフトウェア開発で使われる用語**  ほかのソフトウェア開発の入門書にはあまりのっていませんが、中級～上級書には説明のないまま使われる語を集めてみました[[21]](#footnote-21)。参考にしてください。  **Structured Programming (構造化プログラミング)**  処理の流れを構造化したプログラミングのことです。プログラムの処理の流れは、「順接 (sequence)」「分岐 (selection)」「反復 (iteration)」の3つだけで構造化できることが、1966年に2人の数学者コラド・ベーム先生とジュゼッペ・ヤコピーニ先生によって証明されました。1967年には、エドガー・ダイクストラ先生の「Go To Statement Considered Harmful (goto文は有害である)」という有名な論文が発表されましたが、当時はfor文のような制御構文をサポートするプログラミング言語はなく、大論争を巻き起こしたといいます。ドナルド・クヌース先生は1974年に「Structured Programming and Goto Statements」という論文を著し、goto文は構造化プログラミングと一緒に便利に使えるとしました。この流れにより、徐々にループ構文などをサポートする言語が増え、goto文をサポートする言語は減っていきました。  **OOP (Object Oriented Programming; オブジェクト指向プログラミング)**  プログラムは、データ構造と処理の2つからできています。しかし、以前のプログラムでは、あるデータ構造を、誤って別のデータ構造の処理を意図した関数 (手続き)に渡すことで、致命的なエラーが発生することがありました。そこで、あるデータ構造とそれを専用に処理する関数を密に結び付けるという考え方がうまれました。このような考え方を支援するプログラミング言語が、オブジェクト指向プログラミング言語 (OOPL) です。この、データ構造と処理を結びつけたものをクラスといいます。さらに、OOPLの支援により、既存のクラスを拡張して再利用することも容易になりました。Java言語では、extendsというキーワードを使ってクラスを拡張できます。この「拡張」を「継承」とか「派生」などということもありますが、同じ意味です。現代的なプログラミング言語の多くが、OOPをサポートしています。  **OOAD (Object Oriented Analysis and Design; オブジェクト指向分析設計)**  オブジェクト指向言語は、大変な成功を収めました。プログラミング (コーディング) とは、設計とほぼ同義ですから、最初からクラスという考え方を取り入れて分析・設計をしてしまおう、というのが素朴なOOADの動機です。これも、仕様と実装を密に近づけようという試みのひとつといえます。さらにさまざまな開発手法を加えて、オブジェクト指向開発方法論としたものも多く提唱されています。私見では、OOADを学習するには、まずJavaやC#のようなオブジェクト指向言語に精通するのが早道であろうと思います。  **Generic Programming (ジェネリックプログラミング)**  オブジェクト指向というパラダイムが、どちらといえばデータ構造の再利用を主としたアプローチであるのに対し、ジェネリックプログラミングはアルゴリズムの再利用を主としたアプローチで、総称的プログラミングと訳されますこともあります。Generic (一般な) とは、specific (固有な) という語に対するものです。つまり、specific programmingとは型に囚われたプログラミングであり、generic programmingとは型に囚われないプログラミングです。型を総称的に扱うことから、総称的プログラミングなどと訳されることもあります。オブジェクト指向は、データ構造とアルゴリズムを強固に結びつけてクラスという型にしますが、ジェネリックプログラミングではデータ構造とアルゴリズムをきれいに切り離し、その可能な組み合わせの数を極大化することで、アルゴリズムの再利用を促します。Smalltalkのような動的な型づけを行う言語では、多くの言語でサポートされています。しかし、静的な型づけを行う言語でジェネリックプログラミングを強力にサポートしているのはC++だけです。C++によるジェネリックプログラミングは、STLやBoost、ATLなどのライブラリにより大成功していることが認知されています。JavaやC#がサポートするジェネリックな機能もそれなりに強力ですが、C++のテンプレートによるジェネリックな機能とは質が違います。  **Coupling and Cohesion (結合度と凝集度)**  ソフトウェアの品質を測る指標のひとつで、ラリー・コンスタンティン氏らが導入しました。例えば、以前のソフトウェア製品からある機能を取り出して再利用したいとします。ところが、再利用したい部分を取り出そうとすると、関係ない部分もくっついてきてしまいます。これが、結合度が高い状態です。逆に、ある機能を取り出したいのに、それを実現するコードがソフトウェアシステム全体に散らばっているために取り出せません。これが、凝集度が低い状態です。一般に、一緒に変化 (変更) しそうなものは同じモジュールとして凝集度を高く (高凝集) し、そうでないものは別のモジュールとして結合度を低く (疎結合) することが求められます。特に、ログ出力のためのコードは凝集度が低くなりやすいといわれています。また、「カプセル化」は、結合度を弱めるための技術のひとつです。  **Divide-and-Conquer (分割して統治)**  Divide-and-Ruleとしても知られ、マケドニアのフィリップ二世王 (紀元前382年～紀元前336年) がギリシャを統治するために使った戦略だったといわれています。これは分割統治法 (D&C algorithm) として、コンピュータ科学に応用されています。そのままでは解決できない問題を小さな問題に分割することで、最終的に問題を解決しようとするアルゴリズムです。また、プログラムの複雑性を管理するにも、分割統治するしかないことは以前から常識となっています。問題はどのように分割すべきかですが、これはずっと議論されてきました。このひとつの解が、高凝集性と疎結合性です。最近は、プロジェクト管理などの話題でも見かける語となっています。  **AOP (Aspect Oriented Programming; アスペクト指向プログラミング)**  プログラムにある側面 (aspect) を簡単に追加できるようにしようとするものです。特に、ログ出力などの凝集度がどうしても低くなるようなものをaspectとして、これをプログラムに簡単に追加したり、取り外したりできるようにします。以前は、特別な表記法とツール (コンパイラ) が必要になることがあり、少々使いにくいものでした。しかし、JavaやC#でアノテーションという機能が標準の言語仕様に追加され、アスペクトをプログラム中に埋め込むことが容易になってきています。  **Separation of Concerns (SoC; 関心事の分離)**  関心とは、プログラムの中で興味のある機能や振る舞いのことです。結合度を下げ凝集度を高める考え方のひとつで、ダイクストラ先生が1974年に提唱されました。通信プロトコルが階層構造に分離されていたり、HTMLとCSSが文書の抽象的な構造と見た目を分離しているなどは、SoCのよい例です。  **Granularity (粒度)**  分割の度合いや詳細度をあらわします。小さい単位に分割したものを、粒度が小さいとか、粒度が細かいなどといいます。プログラムやドキュメントを書くときには、適切な粒度を心がけることが重要です。  **Pattern (パターン)**  パターンとは、黄金のワンパターンに名前をつけてカタログ化し、暗黙知を形式知に変換しようという試みです。デザインパターンの大成功によりソフトウェア業界に認知されました。デザインパターンは、ありがちなクラス間の関係や構造に名前をつけてカタログ化したものです。デザインパターンについては、p.★のコラムを参照してください。デザインパターンのほかにも、アーキテクチャパターンやアナリシスパターンなどが提唱されています。  **UML (統一モデリング言語)**  UMLとはUnified Modeling Languageの略で、ソフトウェアの構造を説明する図面のセットです。UMLの初期バージョンを策定したジェームズ・ランボー氏、グラディ・ブーチ氏、イヴァー・ヤコブソン氏の3氏は、スリー・アミーゴスと敬称されています[[22]](#footnote-22)。UMLは、言語といっても、日本語やプログラミング言語のようなものではありません。ソフトウェアのさまざまな側面に注目し、クラス図、状態遷移図、シーケンス図など、複数の図を使い分けてシステムを描写する、ビジュアルな言語です。また、図中の注釈は自然言語 (日本語など) で書かれます。  **Static and Dynamic (静的と動的)**  静的とは変化しない、動的とは変化するという意味です。プログラムをコンパイルしたときに決まり、以後は変化しないことを静的、プログラムの実行時にも変化することを動的とよんで区別します。これらは、UMLのクラス図とオブジェクト図に対応します。クラス図は静的なもので実行時には変化しませんが、オブジェクト図は動的に変化します。注目しているパラメータが静的なものか動的なものかを識別することは、ソフトウェア設計においてとても基本的なことです。  **Hungarian notation (ハンガリアン記法)**  プログラムを書くとき、変数名にその変数の種類を示す接頭文字をつける表記法です。ハンガリー出身のプログラマ[[23]](#footnote-23)が考案したので、ハンガリアン記法といわれるようになりました。例えば、(行数+行数) や、(行数×列数) という演算には意味がありますが、(行数+列数) のような演算は一般に意味をなしません。そこで、行数を表す変数の名前の先頭にrwを、列数を表す変数の先頭にはcolをつけることで、誤った演算を発見しやすくします。異なる単位 (通貨など) の値や、異なる座標系の値を格納した変数を区別するのに、とても便利な表記法です。このほか、変数のスコープや、変数の型を表す接頭文字を変数名につけることもあります。 |

# 参考文献

## Managing the Development of Large Software Systems

<http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>

## Waterfall model

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

## Big Design Up Front

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

## ソフトウェアプロジェクト管理

Walker Royce (著) 日本ラショナルソフトウェア (株) (監訳)

<http://www.amazon.co.jp/dp/4894715597>

## 初めてのアジャイル開発

Craig Larman (著), 児高 慎治郎, 松田 直樹, 越智 典子 (訳)

<http://www.amazon.co.jp/dp/4822281914>

## Software Engineering Economics

Barry W. Boehm

<http://www.amazon.com/dp/0138221227>

## Production of Large Computer Programs

Herbert D. Benington

<http://portal.acm.org/citation.cfm?id=41799>

## The Software Process

Stephen R. Schach

<http://www.mhhe.com/engcs/compsci/schach5/ppt/schach5-chap03-14%5B1%5D.ppt>

## Structured Design

Larry L. Constantine

<http://www.amazon.com/dp/0138544719>

## Divide and Rule

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

## Divide and conquer algorithm

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

## Exit-criteria

<http://en.wikipedia.org/wiki/Exit-criteria>

## アジャイルソフトウェア開発宣言

<http://www.agilemanifesto.org/iso/ja/>

1. 書籍「ソフトウェアプロジェクト管理」から要点を抜粋して引用しました。この著者は、ウィンストン・ロイス博士の息子さんであるウォーカー・ロイス氏です。 [↑](#footnote-ref-1)
2. もちろん、ソフトウェアにおいても同様です。 [↑](#footnote-ref-2)
3. Up frontとは、「前もって」という意味です。 [↑](#footnote-ref-3)
4. そのようなドキュメントを、開発初期に要求するユーザーにも問題があろうかと思います。 [↑](#footnote-ref-4)
5. プログラムの「てにをは」のようなものです。インデントがおかしくても、プログラムの意味は変わりません (Pythonみたいな言語は除きます) が、インデントのつけ方はコーディング規約で統一すべきです。多くのIDEがインデントを揃える機能をもっているので、活用すると良いでしょう。 [↑](#footnote-ref-5)
6. CADとはComputer Added Designの略で、製図作業を支援するシステムのことです。この意味では、Eclipseのような開発統合環境こそが、ソフトウェアの設計支援システムであるように私には思われます。似たような語に、CAE; Computer Added Engineering (工学支援システム)、CAI; Computer Added Instruction (教育支援システム) などがあります。 [↑](#footnote-ref-6)
7. ビジュアルなDSLというのも提案されていますが、これはMDAの一種と考えられます。 [↑](#footnote-ref-7)
8. 置換行為をはたらくときは、正規表現が便利です。(すいません) [↑](#footnote-ref-8)
9. 書籍「初めてのアジャイル開発」は、反復型開発をとりいれた最初のプロジェクトは少なくとも1950年代後半にさかのぼるとしています。 [↑](#footnote-ref-9)
10. 個人的には、マイルストーンという方が馴染みがあります。 [↑](#footnote-ref-10)
11. このような項目には、とりあえずT.B.D.と書いておきましょう。To Be DescribedもしくはTo Be Determinedの略で、後で書くとか、後で決めるという意味です。私は、T.B.D. と書くとき、いつも映画「バックトゥザフューチャー」のエンディングを思い出します (あれはT.B.C.でしたね)。 [↑](#footnote-ref-11)
12. もちろん、最終リリースを出荷するときのcriteriaを満たしていなければ、そのまま出荷することはできません。スケジュールを伸ばすか、安定化の反復をもう1回繰り返すかしてcriteriaにmeetさせるしかありません。 [↑](#footnote-ref-12)
13. 邦訳が文庫で読めます。おもしろいです。 [↑](#footnote-ref-13)
14. 本書で説明したSCMやBTSなどは、ソフトウェア開発に必須のインフラです。 [↑](#footnote-ref-14)
15. http://www.agilemanifesto.org/iso/ja/ [↑](#footnote-ref-15)
16. ジェラルド・ワインバーグ氏は、著書「パーフェクトソフトウェア」の中で、50年前にテストファーストのプラクティスを使ったことがあると述べています。 [↑](#footnote-ref-16)
17. eXtreme Programmingと表記する人もいますが、ベック氏はこれをExtreme Programmingとつづっています。 [↑](#footnote-ref-17)
18. アルファベット順に示しました。 [↑](#footnote-ref-18)
19. アンブラー氏はIBMのアジャイル開発のリーダーであり、「アジャイルモデリング」や「データベース・リファクタリング」など、多数の著作で知られています。また、ヤコブソン氏はUMLの設計者の1人としても有名です。 [↑](#footnote-ref-19)
20. 書籍「Object-Oriented and Classical Software Engineering」より引用しました。 [↑](#footnote-ref-20)
21. などと尤もらしいことを書いていますが、実は独立したコラムにしそこなったものをかき集めただけのものです。大目に見てください。 [↑](#footnote-ref-21)
22. iThree Amigos!といえば、コメディ映画「サボテン・ブラザーズ」の原題ですね。同様の映画「ギャラクシー・クエスト」も合わせてどうぞ。 [↑](#footnote-ref-22)
23. Charles Simonyiという人で、Microsoft Excelの開発を指揮しました。インテンショナル・プログラミングの創始者でもあります。すごい資産家で、宇宙旅行に2回行ったそうです。インテンショナル・プログラミングについては、拙訳「ジェネレーティブプログラミング」をどうぞ。 [↑](#footnote-ref-23)