エクストリームプログラミング

Kent Beck・Cynthia Andres 共著
角 征典 訳

Extreme Programming *Explained*

Embrace Change, 2nd Edition

Addison-Wesley
Ohmsha

推薦の言葉

『Extreme Programming Explained』の第2版は、ケント・ベックがXPの5年間の経験、成長、変更をまとめて、提示したものである。XPの改善の道をチームと一緒に歩き出す方法を真剣に知りたいならば、本書を読むべきだ。

▶ フランチェスコ・チリッロ（Francesco Cirillo）
XPLabs S.R.L. 社 CEO

第1版はXPが何かを教えてくれた。そして、多くの人たちのソフトウェア開発に対する考えを変えた。第2版では、それをさらに推し進めて、XPのさまざまな「なぜ」を教えてくれる。それは、プラクティスの背景にある動機や原則のことだ。素晴らしい内容である。「what」に加えて「why」を身に付ければ、プロジェクトをうまく実行する方法や、組織にアジャイル技法を導入する方法などの「how」に自信を持って取り組めるだろう。

▶ デイヴ・トーマス（Dave Thomas）
The Pragmatic Programmers LLC

本書はダイナマイトだ！ 数年前に登場した第1版は革命的だったが、新しい版も同様に深い内容になっている。料理の本のチェックリストのような内容を求めている人には、「主要プラクティス」という素晴らしい章が用意されている。だが、最初の章の冒頭にある「XPはソーシャルチェンジである」という言葉の意味を真剣に考えるところから始めてほしい。すべてのIT専門家とすべてのITマネージャー（最終的にはCIOに至るまで）の机の上に『Extreme Programming Explained』が置かれるように、あらゆる手段を尽くすべきだ。

▶ エド・ヨードン（Ed Yourdon）
作家、コンサルタント

XPには、ソフトウェアの設計、開発、テストのプロセスをシンプルにする強力なコンセプトが詰まっている。ミニマル主義とインクリメンタル主義だ。創造性と規律のバランスが求められる複雑な問題に対応する際に有用な原則である。

▶ マイケル・A・クスマノ（Michael A. Cusumano）
MIT スローン経営大学院教授、『ソフトウエア企業の競争戦略』著者

『Extreme Programming Explained』は、才能のある情熱的な職人の作品だ。ケント・ベックが、プログラミングとマネジメントに関する説得力のあるアイデアをまとめてくれた。これは最大の注目に値する。不満があるとすれば、我々の業界がこうした常識的なアイデアに「エクストリーム」とラベルをつけてしまうことだ……。

▶ ルー・マツケリー（Lou Mazzucchelli）
Cutter Business Technology Council 社フェロー

組織のソフトウェア開発のやり方を変える方法は 2 つある。一度にひとつずつ時間をかけて変えていくインクリメンタルな方法と、エクストリームプログラミングに飛び込む方法だ。エクストリームという名前に驚かないでほしい。決してエクストリームなものではない。そのほとんどが古きよきレシピや常識をまとめたものであり、長年にわたって積み重なってきた余計な部分をうまく削ぎ落としたものである。

▷ フィリップ・クルーシュテン（Philippe Kruchten）
ブリティッシュコロンビア大学

革命が独り歩きし始めると、それを生み出した革命家は取り残されてしまうことがある。本書では、ケント・ベックが今も時代の先端に残り、XP を次のレベルに導いていることが示されている。5 年間のフィードバックを取り入れ、優れたソフトウェアを短い時間と少ないお金で開発するために必要なことを再検討している。ここに銀の弾丸はない。あるのは実用的な原則だけだ。賢く使えば、ソフトウェア開発の生産性が劇的に向上するだろう。

▷ メアリー・ポッペンディーク（Mary Poppendieck）
『リーンソフトウエア開発』著者

ケント・ベックは、5 年以上かけて XP の適用や教育を行い、それをもとに名著を改定した。XP の道がどれだけ簡単で、困難なものかを示している。最初に使用するプラクティスはわずかであっても構わない。たとえわずかであったとしても、チームは今までよりも前進できるはずだ。

▷ ウィリアム・ウェイク（William Wake）
独立コンサルタント

ベックの名著の新しい版には、エクストリームプログラミングという技芸の新しい知見、経験から得られた知恵、明快な説明が書かれている。最高のソフトウェア開発という夢を多くの人が叶えることができるだろう。

▷ ジョシュア・ケリーエブスキー（Joshua Kerievsky）
『パターン指向リファクタリング入門』著者、Industrial Logic, Inc. 社創業者

XP は業界のソフトウェア開発に対する考え方を一変させた。見事なまでにシンプルで、実行に集中しており、推測ではなく事実にもとづいた計画にこだわっている。これはソフトウェアデリバリーの新しい標準だ。

▷ デヴィッド・トローブリッジ（David Trowbridge）
Microsoft 社アーキテクト

日本語新訳版への推薦の言葉

プログラマーの活動をエンジニアリングとソーシャルの両面から変えた設計手法、XP。ソフトウェアが世界と繋がった瞬間であり、個人が世界と繋がった瞬間。

▶ 平鍋健児
株式会社チェンジビジョン

エクストリームプログラミングは、ケント・ベックがクリストファー・アレグザンダーのパターンランゲージをソフトウェア開発に応用しようとした取り組みである。アジャイルソフトウェア開発の出発点として重要であるだけでなく、逆にここでの取り組みを建築や空間設計に活かすことができるだろう。この本には心地良い空間を設計するためのヒントが溢れている。ぜひソフトウェア開発に限らず、空間を設計する人にも届いてほしい。

▶ 江渡浩一郎
メディアアーティスト、国立研究開発法人産業技術総合研究所主任研究員、ニコニコ学会 β 実行委員長

この本には自分たちを自分たちで変化させる話が書かれています。はじめて『Extreme Programming Explained』を読んだとき、これはプラクティス集であると思いました。しかし第 2 版を読み返してそれが誤りであることに気づきました。紹介されているプラクティスはひとつの例にすぎず、その背景こそが XP でした。守破離の「離」まで含めて、XP の「守」だったんです。いま新しい翻訳の XPE を読むことができるのがとても楽しみです。入門者はもちろん、Scrum などの他のやり方に慣れている人にこの本を強く推薦します。

▶ 関将俊
プログラマ・アーティスト、『The dRuby Book』著者

ソフトウェア開発は本来とても楽しいものだ。新しい価値が世の中に提供されることに、自分自身も仲間も、時には競合相手でさえも興奮を隠せずワクワクしてしまう。しかしやり方を間違うと、関係者が次々と不幸になっていってしまうこともある。XP はこうした不幸を避け、人間関係を良好に保ち、それぞれが仕事を楽しみながら求められる成果を出していくための方法論だ。ソフトウェア開発の本来の楽しさを引き出し、ソフトウェアのよりよい未来を作っていくために、本書は必携の書だと言えるだろう。

▶ 小野和俊
株式会社アプレッソ 代表取締役社長、株式会社セゾン情報システムズ CTO

今やソフトウェアはビジネス価値を最大化するために不可欠なものとなった。ビジネスの変化のうねりを受け入れるにはソフトウェアが原動力となるということだ。では、ソフトウェア開発の現場はどうか？ 個々の能力を活かしているか？ ビジネス価値を、ソフトウェアそのものを探求しているか？ 本書には、そのギャップを埋め、あなた自身とあなたのチーム、顧客が考え、実践するためのきっかけが詰まっている。

▶ 長沢智治
アトラシアン株式会社 エバンジェリスト

もしもあなたがプログラミングを一生の仕事として取り組む価値が本当にあるのか、よりよい社会の構築に一介のプログラマでも貢献できることがあるのか、そんな懐疑的な気持ちになったことがあるなら、この本を読むことを勧めます。一人の優れたプログラマが同じ疑問に真正面から取り組み、肯定的な答えを出すべく苦闘した足跡は、きっと深い共感と確信をもたらしてくれるでしょう。

▶ 高橋征義
一般社団法人日本 Ruby の会代表理事、株式会社達人出版会

15 年前、XP は革命でした。ソフトウェア開発のあり方を変え、個とチームが価値と向き合う方法を教えてくれました。現在、XP を源流のひとつとしたアジャイルソフトウェア開発は「普通の、既にある」やり方になりました。するとそこからは「なぜ」が失われがちで、ふとレールを外れると足取りがおぼつかないことに気が付きます。失われたものを求めて原典に戻る必要があります。本書が、その原典です。

▶ 和田卓人
テスト駆動開発者、タワーズ・クエスト株式会社取締役社長

アジャイルプロジェクトを経験してきて、プロジェクト成功、すなわち「価値の高いソフトウェアを生み出す」カギは、プログラマの力だけではなく、チームに関わるすべてのメンバが互いにリスペクトする文化の醸成であると感じている。約 15 年前に本書で KB が示したように。

▶ 児玉公信
情報システム総研取締役副社長／モデラー、『UML モデリングの本質』著者、『リファクタリング』訳者

初版は世田谷の坂道を歩きながら読んだ覚えがある。見上げるとカラスが急降下してきた。この本はアジャイルの基本書というだけでなく、単独のパターンではなく体系としてのパタンランゲージの記述を建築以外の分野で初めて試み成功した貴重な実践例である。そして第 2 版でパタンランゲージは実践していく中で経験を取り込み書き換えられ進化していくということが学べる。変化を受け容れるということの自己適用が実践されている貴重な書。これを次にはもう一度、建築・まちづくりに持ち込み直すのが楽しみ。

▶ 羽生田栄一
株式会社豆蔵、UBrainTV

リスペクトをそなえた人と人とのつながり。これがプログラマーの仕事の価値を極限まで高めるための秘訣だ。XP が示すのは、リスペクトする/されるのに必要な考え方と振る舞いの道だ。読むだけでは「なあんだ」と思うかもしれない。だが道を知ることと、その道を歩むことは違う。だからケント・ベックは私たちを励ます。「プログラマーでも現実世界の一員になれる」。人が人のためにコードを書く、プログラミングという営みが時を超えられるかどうかは私たちにかかっている。あなたから、今日から始めよう。Social change starts with you.

▶ 角谷信太郎
『アジャイルサムライ』監訳者

著者はXPの5年間の実践を経て、顧客やビジネスと寄り添ったソフトウェア開発の価値・原則・プラクティスを、より洗練された「言葉」として整理し、それを組み合わせることで成功確率の高い仕事の進め方を記している。本書は、その言葉たちを大切に扱った新訳であり、実に読みやすい。しばしばアジャイル開発が誇大に評価されるような今日において、読者は本書の言葉に触れることでその本質と限界を理解し、言葉を用いて自らのよりよい仕事の進め方を考え、語り、形作ることができる。

▶ 鷲崎弘宜
早稲田大学グローバルソフトウェアエンジニアリング研究所所長

XPとの出会いは私の人生でもっとも大きな出来事でした。私はXPを実践していくうちに「自分たちのやり方を自分たちで考えて工夫していける」ということこそがXPの最大の魅力だと気づきました。この瞬間、私自身の仕事への向き合い方と周囲の人たちとの接し方が180度変化しました。本書をきっかけに多くの新たな出会いと変化が生まれることを祈ります。

▶ 木下史彦
株式会社永和システムマネジメント アジャイル事業部 事業部長

この10年間Scrumのファンだった。アジャイル開発に取り組み始めた頃、XPには何かハードルの高さのようなものを感じていた。多くの言語や環境でツールも整ったいま、本書が出版される。これから始める人や、全部を実践することが難しいチームにとっても、ずっとやさしい内容になっている。プラクティスがそのバックグラウンドと共に気持ちよく頭に入ってくる。これからはXPファンになりそうだ。

▶ 守田慥司
スクラム道スタッフ

XPは私の人生を変えた。本書は何度読み直してもそのたびに新しい発見のある稀有な一冊であった。本書は「XPとは何か」だけでなく「よりよい仕事・よりよい人生を送るための道」を記している。本書は「偉大な習慣を身につけた先輩プログラマー」から「これから偉大な習慣を身につけたいプログラマー」へのバトンである。新訳となり初版から15年の時を超えすべてのソフトウェア開発にかかわる人々にお勧めする。

▶ 懸田剛
Agile459代表、日本XPユーザーグループ創立スタッフ

■ XP シリーズの紹介

ケント・ベック（シリーズアドバイザー）

エクストリームプログラミング（XP）は、ソフトウェア開発ビジネスの規律であり、チーム全体が共通の達成可能なゴールに集中するためのものである。XP の価値と原則を使えば、チームは XP の適切なプラクティスを自分たちの状況に取り入れることができる。XP のプラクティスには、人間の創造性に刺激を与え、人間の弱さを受け入れるものが選ばれている。XP チームは、高品質なソフトウェアを持続可能なペースで生み出すことができる。

XP の目的のひとつは、ソフトウェア開発に説明責任と透明性をもたらし、その他のビジネス活動と同じようにソフトウェア開発を運営することである。もうひとつの目的は、目覚ましい成果を達成することである。つまり、現在の期待以上に効果的で、効率的で、欠陥の少ない開発だ。ソフトウェア開発に関係のあるすべての人たち（スポンサー、マネージャー、テスター、ユーザー、プログラマー）の人間としての欲求を受け入れ、それらを満たすことによって、XP はこうした目的を達成しようとしている。

XP シリーズは、XP を適用するときの無数のバリエーションを模索するために存在している。最初は社内プロジェクトで働く小さなチームのための方法論として作られた XP が、今では世界中のチームによって、パッケージソフト、組み込みソフト、大規模プロジェクトなどに使用されている。XP シリーズの書籍は、技術的な問題と社会的な問題の両方に対応しながら、さまざまな状況に XP を適用する方法を説明している。

ソフトウェア開発には変化が訪れた。変化は脅威ではなく、機会と見なすことができる。変化のための計画があれば、チームはこの機会をうまく活用できる。XP はそうした変化のための計画のひとつである。

- ケン・アウアー、ロイ・ミラー『XP エクストリーム・プログラミング適用編 —— ビジネスで勝つための XP』（ピアソン・エデュケーション）
- ケント・ベック、シンシア・アンドレス『エクストリームプログラミング』（オーム社）
- ウィリアム・C・ウェイク『XP エクストリーム・プログラミング アドベンチャー』（ピアソン・エデュケーション）
- ダグ・ウォレス、イザベル・ラゲット、ジョエル・アウフガング『XP エクストリーム・プログラミング Web 開発編』（ピアソン・エデュケーション）
- ロン・ジェフリーズ、アン・アンダーソン、チェット・ヘンドリクソン『XP エクストリーム・プログラミング導入編 —— XP 実践の手引き』（ピアソン・エデュケーション）
- ケント・ベック、マーチン・ファウラー『XP エクストリーム・プログラミング実行計画』（ピアソン・エデュケーション）
- Lisa Crispin、Tip House『Testing Extreme Programming』(Addison-Wesley Professional)

シンディへ

君がいなければ、部屋の隅に隠れたプログラマーに関する本になっていた。
君がいなければ、私もそうしたプログラマーのままだった。

プログラマーたちへ

プログラマーでも現実世界の一員になれる。XP によって、自分自身をテストすることができる。ありのままの自分になることができる。自分には何も問題はなく、付き合っていた仲間が悪かっただけということがわかる。

第2版への序文

わお。第2版だって。第1版からもう5年も経ったなんて信じられないね。第2版の序文を書いてほしいとケントから頼まれたときに、原稿の変更箇所を教えてくれと言ったんだけど、なんてアホなお願いをしてしまったんだろう。全部書き変わっているじゃないか！ 第2版では、ケントはXPを再検討して、XPのパラダイム（注意して、適応して、変更する）をXPそのものに適用している。あらゆる箇所を再検討して、キレイにして、リファクタリングして、数多くの新しい知見と統合したのである。その結果、ものすごくわかりやすくなった！

せっかくの機会だから、XPが私のソフトウェア開発に与えた影響を見ていこう。第1版が出てからすぐに、私はEclipseプロジェクトにかかわるようになった。今ではソフトウェアのエネルギーをすべて注ぎ込んでいる。Eclipseは純粋なXPで開発されているわけではない。その他のアジャイルプラクティスも使っているからだ。それでも、XPの影響を受けていることはすぐにわかるだろう。一番わかりやすいのは、XPのプラクティスをEclipseにそのまま組み込んでいるところだ。リファクタリング、ユニットテスティング、コードを入力したときの迅速なフィードバックといった機能は、今ではEclipseに欠かせない部分になっている。それから、我々は「ドッグフードを食べる」ので、こうしたプラクティスを自分たちの日々の開発にも使っている。興味深いのは、XPが我々の開発プロセスにも影響を与えていることだ。Eclipseはオープンソースプロジェクトであり、透明性のある開発を実践することが目的のひとつになっている。その根拠は単純だ。プロジェクトがどこへ進もうとしているのかがわからなければ、助けることもできないし、フィードバックを提供することもできないからである。XPのプラクティスは、こうした目的の達成を支援してくれている。

我々のプラクティスの適用方法を紹介しよう。

- ✧ **テスト：早めに、こまめに、自動化** —— 最新のビルドでチェックマークをグリーンにするには、21,000件以上のユニットテストをパスする必要がある。
- ✧ **インクリメンタルな設計** —— 設計に毎日時間を使っている。ただし、APIは安定させなければいけないという制約がある。
- ✧ **デイリーデプロイ** —— コンポーネントのコードを1日に最低1回はデプロイして、デプロイしたコードを踏まえて開発している。迅速なフィードバックを手に入れたり、問題を早期に発見したりするためだ。

- ✧ **顧客参加** —— 幸いにも我々には活発なユーザーコミュニティーがある。彼らは内向的ではなく、継続的にフィードバックを提供してくれる。こうしたフィードバックに耳を傾け、我々は最善を尽くしている。
- ✧ **継続的インテグレーション** —— 最新のコードを毎晩ビルドしている。ナイトリービルドによって、コンポーネントのインテグレーションに関する問題の知見が得られる。週に1回はインテグレーションビルドを実施して、すべてのコンポーネントの整合性を確保している。
- ✧ **短期間の開発サイクル** —— 我々の開発サイクルは、XPが提唱する週次サイクルよりも長い。だが、目的は同じだ。6週間の開発サイクルの終わりには、プロジェクトのリズムになるマイルストーンビルドを作っている。それは、進捗を外部に示すためであり（だからウソがつけない）、コミュニティーが本当に使用できて、フィードバックを提供できるだけの高品質のソフトウェアをデリバリーするためである（ますますウソがつけない）。
- ✧ **インクリメンタルな計画づくり** —— リリースが終わったら、未熟な全体計画を発展させている。全体計画はリリースサイクルのなかで成長させる。全体計画は早い段階からウェブサイトに掲載して、ユーザーコミュニティーが対話に参加できるようにしている。ただし、マイルストーンは例外である。これはプロジェクトのリズムを決めるためのものなので、最初のプランニングイテレーションで固定している。

我々はXPのすべてを取り入れているわけではないが、上記のようなXPのプラクティスから多くのことを手にしている。なんといっても開発のストレスが軽減されている！ 事前に計画したマイルストーンや出荷日を正確に達成する鍵は、こうしたプラクティスにある。高品質のソフトウェアを期日内に届けることを約束した強力なチームが、これらのプラクティスを実際に行っているのである。

ケントは、今でも私のソフトウェア開発の考え方に疑問を投げかけてくる。本書を読みながら、いつかやってみたいと思えるプラクティスをいくつも発見した。あなたも同じように、本書からXPの招待状を受け取り、ソフトウェア開発のやり方を改善して、素晴らしいソフトウェアを生み出してほしい。

2004年9月

エリック・ガンマ

第1版への序文

エクストリームプログラミング（XP）は、コーディングをソフトウェアプロジェクトの中心的な活動としている。こんなものがうまくいくはずがない！

私の開発の仕事を少しだけ見てみよう。私は「Just In Time Software[†1]」の文化のなかで働いている。リリースサイクルは圧縮されていて、技術的リスクも高い。生き延びるためには、仲間に変化をもたらさなければいけない。チームは地理的に分散している場合も多く、コミュニケーションはコードで行われている。我々はコードを読んで、新規または拡張中のサブシステムの API を理解する。複雑なオブジェクトのライフサイクルや振る舞いは、テストケースに定義されている。同様に、コードにも定義されている。問題レポートには、その問題を再現するテストケースがついている。同様に、コードにも問題が示されている。リファクタリングを使い、既存のコードを継続的に改善させている。我々の開発は、明らかにコードが中心だ。だが、期日内にソフトウェアをデリバリーできている。つまり、この方法はうまくいくのである。

ひたすらプログラミングすれば、ソフトウェアをデリバリーできる。そう結論付けるのは間違いだ。ソフトウェアのデリバリーは難しい。品質の高いソフトウェアを期日内にデリバリーするのはもっと難しい。それを実現するには、今までとは異なるベストプラクティスを規律正しく使う必要がある。そのためにケントは、示唆的な XP の本を書き始めたのである。

ケントは、Tektronix 社でリーダーたちと一緒に、複雑なエンジニアリングアプリケーションにおける Smalltalk による人間中心のペアプログラミングの可能性に気づいた。彼は、ウォード・カニンガムと一緒に多くのパターンムーブメントを引き起こし、私のキャリアに大きな影響を与えた。XP とは、成功を収めた多くの開発者たちのプラクティスを組み合わせ、開発アプローチとして表現したものである。こうしたプラクティスは、ソフトウェア手法やプロセスに関する膨大な文献に埋もれていたものたちだ。パターンと同じように、XP は、ユニットテスティング、ペアプログラミング、リファクタリングなどのベストプラクティスの上に成り立っている。XP では、これらのプラクティスを組み合わせて、相互に補完や制御ができるようになっている。さまざまなプラクティスの相互作用に焦点を当てたことが、本書の重要な貢献である。目的となるのは、適切な機能を備えたソフトウェアを期日ま

[†1] 訳注：当時エリック・ガンマが勤務していた Object Technology International（OTI）社のソフトウェア開発手法のこと。同社は 1996 年に IBM に買収されている。

でにデリバリーすることだ。OTI社の「Just In Time Software」プロセスは純粋なXPではないが、共通点も多い。

ケントとやりとりしながらJUnitと呼ばれるちょっとしたものを作り、XPを実際に経験できたのは楽しかった。彼の視点と手法は、私のソフトウェア開発の取り組みに常に疑問を投げかけてくる。従来の「作業規定[†2]」のアプローチにも間違いなく異論を唱えるはずだ。本書を読めば、あなたもXPを受け入れるか否かの選択をすることになるだろう。

1999年8月

エリック・ガンマ

[†2] 訳注：方法論（methodology）の頭文字が大文字の「M」になったもの。トム・デマルコ、ティモシー・リスター『ピープルウエア 第3版』（日経BP社）で「作業規定」と訳されている。

はじめに

エクストリームプログラミング（XP）の目的は、圧倒的なソフトウェア開発の実現である。ソフトウェアは、もっと安いコストで、もっと少ない欠陥数で、もっと高い生産性で、もっと高い投資効率で、開発することができる。現在苦戦しているチームであっても、仕事のやり方に目を向けて洗練したり、通常の開発プラクティスを極限（エクストリーム）まで推し進めたりすることによって、このような結果が成し遂げられる。

ソフトウェア開発の方法には、良いものと悪いものがある。良いチームというのは、異なるところよりも似ているところのほうが多い。良いチームだろうと悪いチームだろうと、改善は必ずできる。本書を改善の参考書にしてほしい。

本書は、良いソフトウェア開発チームの共通点を私なりにまとめたものだ。個人的にうまくできたことや、うまくできているところを目の当たりにしたことについて、私が考える最も純粋で最も「エクストリーム」な形で抽出している。こうした作業のなかで、私は自分の想像力の限界を思い知らされた。第 1 版が出版された 5 年前に「エクストリーム」だと思っていたプラクティスが、今ではありふれたものになっていたからだ。本書に登場するプラクティスも、おそらく 5 年後には控えめなものになっているだろう。

良いチームの活動のことばかり話していては、話が見えなくなってしまう。チームの活動は、仕事の状況によって違いがある。活動は川のさざ波だ。その水面下を見れば、卓越したソフトウェア開発の知性と直感の基盤が存在する。それこそが、私が抽出して文書化しようとしているものである。

第 1 版に対する批判は、プログラミングのやり方を強制しているというものだった。他人の行動をコントロールできるはずもないのだが、恥ずかしながらそれが当時の私の狙いだった。他人の行動をコントロールできるという幻想を捨て、一人ひとりが自分の選択に責任を持っていることを認識した。今回の版では、多くの人に伝わるように肯定的にメッセージを言い換えて、みなさんの道具箱に追加できるような実証済みのプラクティスを提供している。

- ✧ どんな状況でも必ず改善できる。
- ✧ どんなときでもあなたから改善を始められる。
- ✧ どんなときでも今日から改善を始められる。

謝辞

素晴らしいレビューアたちに感謝したい。かなりの時間をかけて原稿を読み、コメントをつけてくれた：Francesco Cirillo、Steve McConnell、Mike Cohn、David Anderson、Joshua Kerievsky、Beth Andres-Beck、Bill Wake。

Silicon Valley Patterns Group のみんなも草稿に対して貴重なフィードバックを提供してくれた：Chris Lopez、John Parello、Phil Goodwin、Dave Smith、Keith Ray、Russ Rufer、Mark Taylor、Sudarsan Piduri、Tracy Bialik、Jan Chong、Rituraj Kirti、Carlos Mc Evilly、Bill Venners、Wayne Vucenic、Raj Baskaran、Tim Huske、Patrick Manion、Jeffrey Miller、Andrew Chase。

ピアソン社の制作スタッフのサポートは素晴らしかった：Julie Nahil、Kim Arney Mulcahy、Michelle Vincenti。

編集者である Paul Petralia は、困難な時期をユーモアと理解で見守ってくれた。人間関係の大切さを学ばせてもらった。

ペアプログラミングのパートナーである Erich Gamma は、会話とフィードバックを提供してくれた。

Bluestone Bakery and Cafe のオーナーとスタッフは、ホットチョコレートとネット環境を提供してくれた。

Joëlle Andres-Beck は、原稿の校正とガベージコレクションをしてくれた。

子どもたち（Lincoln、Lindsey、Forrest、Joëlle）は、私が Bluestone カフェで校正しているときに何時間も付き合ってくれた。

Gunjan Doshi は、示唆に富んだ質問をしてくれた。

最後に、妻であり、デベロップメンタルエディタ[†3]であり、友人であり、知的な仲間である Cynthia Andres には、いくら感謝してもしきれない。

[†3] 訳注：書籍の内容や構成についてアドバイスする編集者。http://en.wikipedia.org/wiki/Developmental_editing

目次

第1章

XPとは何か

エクストリームプログラミング（XP）はソーシャルチェンジである。XPとは、以前はうまくいっていたかもしれないが、今では最高の仕事の邪魔になっている習慣やパターンを手放すことだ。XPとは、これまで自分たちを守ってきてくれたが、今では生産性の妨げになっているものを捨て去ることだ。何だか自分がさらけ出されたような気持ちになるかもしれない。

XPとは、自分たちのできることをオープンにして、それを実行に移すことだ。そして、そのことを他の人にも認めたり、期待したりすることだ。「自分は頭がいいんだから、ひとりで上を目指せばいい」などという未熟な思い込みを克服することだ。もっと広い世界で成熟した場所を見つけることだ。ビジネスや仕事も含めたコミュニティーのなかで、自分の居場所を見つけることだ。自己超越のプロセスのことだ。そのプロセスのなかで、開発者として最善を尽くすことだ。ビジネスのためになる優れたコードを書くことだ。

良好なビジネスは、良好な人間関係によってもたらされる。生産性や自信はコーディングや仕事だけでなく、仕事場の人間関係とも結び付いている。成功には、優れた技術力と良好な人間関係が必要だ。XPはその両方を扱っている。

成功に向けて準備しよう。前へ踏み出すこともせずに、自分から成功を遠ざけてはいけない。ベストを尽くし、その結果を受け入れよう。それがエクストリームだ。自分をさらけ出すのだ。そのことを怖いと思う人もいるかもしれない。逆にそれが日常的になっている人もいるだろう。XPに対する反応が人によって極端に違うのはそのためだ。

XPは、プログラミング技法、明確なコミュニケーション、チームワークなどを巧みに利用して、これまでに想像すらできなかったことを実現するためのソフトウェ

ア開発のスタイルである。XP には、以下のことが含まれる。

- ✧ コミュニケーション、フィードバック、シンプリシティ、勇気、リスペクトの価値にもとづいたソフトウェア開発の哲学。
- ✧ ソフトウェア開発の改善に有効であることが実証された複数のプラクティス。これらは相互に補完して、それぞれの効果を高める。プラクティスは、価値を表現するために選ばれたものたちである。
- ✧ 補完的な原則。価値をプラクティスに変換するための知的な技法。問題に対応できるプラクティスがないときに役に立つ。
- ✧ これらの価値やプラクティスを共有するコミュニティー。

XP とは、ソフトウェアを一緒に開発する人たちが高みに至るまでの改善の道である。他の方法論とは、以下の点で区別される。

- ✧ 開発サイクルが短期間である。それにより、迅速で具体的で継続的なフィードバックがもたらされる。
- ✧ 計画手法がインクリメンタルである。それにより、プロジェクトの期間中に発展していく全体計画をすばやく作成できる。
- ✧ 機能の実装スケジュールが柔軟である。それにより、ビジネスニーズの変化に対応できる。
- ✧ 開発の進捗状況を把握したり、システムを発展させたり、早期に欠陥を捕捉したりするために、プログラマー、顧客、テスターたちが書いた自動テストを信頼している。
- ✧ システムの構造や意図を伝えるために、口頭でのコミュニケーション、テスト、ソースコードを信頼している。
- ✧ システムが存在する限り設計を続けるために、進化的な設計プロセスを信頼している。
- ✧ 普通の才能を持った熱心で積極的な個人がお互いに密接に協力し合うことを信頼している。
- ✧ チームメンバーの短期的な動機とプロジェクトの長期的な利益の両方につながるプラクティスを信頼している。

第 1 版では、XP を「あいまいで急速に変化する要件に向き合った中小規模のソフトウェア開発チームのための軽量級の方法論」であると明確に定義した。この定義では、XP の起源と意図は示されているが、全体像が伝えられていない。第 1 版が出版されてから 5 年間で、さまざまなチームが当初の定義以上に XP を前進させてきた。XP は、以下のように説明できる。

- ✧ XP は、軽量である。XP では、顧客にバリューをもたらすために必要なことだけを実施する。多くの荷物を抱えたまま、すばやく移動することはできない。フリーズドライされたソフトウェアプロセスは存在しない。優秀なチームに必要な技術の知識は膨大であり、これからも増え続けるだろう。
- ✧ XP は、ソフトウェア開発の制約に対応するための方法論である。プロジェクトのポートフォリオ管理、プロジェクトの財務的な正当化、運用、マーケティング、営業などに対応するものではない。これらの分野にも関連しているが、直接的に対応するものではない。方法論というと「成功を保証するルール集」という意味で受け取られることも多いが、方法論はプログラムのようには動かない。人間はコンピューターではない。チームによって、XP のやり方も成功の度合いも違ってくる。
- ✧ XP は、あらゆる規模のチームに使える。5 年前はそのことをあまり主張したくなかった。だが、それ以降さまざまなプロジェクトに XP が導入され、プロジェクトやチームの規模を問わずに成功することがわかった。XP の背景にある価値や原則は、あらゆる規模に適用可能である。ただし、かかわる人数が増えたときには、プラクティスの追加や変更が必要になる。
- ✧ XP は、あいまいで急速に変化する要件に対応する。XP は今でもこうした状況に適している。これは好都合といえるだろう。現代のビジネス世界における急速な転換に適応するには、要件を変化させる必要があるからだ。ただし、移行プロジェクトのような要件が変化しにくいところであっても、チームは XP をうまく使うことができている。

XP は、私自身のソフトウェア開発の実践のなかで人間性と生産性を調和させ、その調和を共有しようとする試みである。自分や他人に思いやりのある接し方をすれば、生産性が高まることがわかってきた。成功の鍵は、個人の努力ではなく、「人と人」のビジネスに自分が携わっていることを受け入れることである。

技術も重要だ。我々は技術分野の技術人間である。働き方には、良いものと悪いものがある。開発においては、技術の高みを目指す行動が欠かせない。技術力は信頼関係につながる。作業を正確に見積もり、最初から品質の高いものを届け、高速なフィードバックループを構築すれば、あなたは信頼されるパートナーになれる。XP では、チームの目標達成に貢献できるレベルの高い技術を学ぶことを求めている。

XP は、仕事の古い習慣を捨て、現状に合わせた新しいやり方を導入するものである。若い頃に身に付けた習慣、態度、価値は、当時はうまくいったかもしれない。だが、チームによる現在のソフトウェア開発の世界では、必ずしも最善の選択とは

いえない。XPを使った開発の成功には、卓越した技術スキルだけでなく、健全で安全な社会的交流も必要だ。

たとえば、「脆弱性による安全性（*vulnerability is safety*）」という考えがある。安全性を確保するために、何かを犠牲にするという意味だ。こうした古い習慣はもはや機能しない。残しておいた20%の力が自分を守ってくれることはない。プロジェクトが失敗したときに、全力を尽くさなかったという事実が気持ちを楽にしてくれることはない。プロジェクトがうまくいかなかった挫折感から救ってはくれない。だが、全力でプログラムを書いていれば、たとえそれが受け入れられなかったとしても、自分自身に満足できる。どんな状況でも、こうした態度が気持ちを楽にしてくれる。自分が最善を尽くしたかどうかで感情が決まるのであれば、最善を尽くすことによって満足感が得られるのである。

XPチームは全力で目的達成に立ち向かい、その結果に対する責任を受け入れる。自尊心とプロジェクトが切り離されているので、いかなる状況でも自由に最高の仕事ができる。XPでは、失敗の準備はしない。XPチームには、人間関係の距離を置いたり、働きすぎ／働かなさすぎで全力を出さなかったり、フィードバックを遅らせて責任逃れをしたりする行為は存在しない。

チームには、時間、お金、スキルが足りていることもあれば、足りていないこともある。だが、常に十分にあるかのように振る舞うことが大切だ。この「充足の精神」については、人類学者のコリン・ターンブルが、2つの社会（資源の乏しいウソと裏切りの部族と、資源の豊富な愛と協調性の部族）を比較して、その様子を『ブリンジ・ヌガグ』（筑摩書房）や『森の民』（筑摩書房）に感動的に記録している。私は、ジレンマを抱える開発者に「時間が十分にあれば、どうしますか？」とよく質問している。制約があっても、最善を尽くすことはできる。制約に心を奪われると、ゴールから遠ざかってしまう。いかなる制約があろうとも、自分自身を明確にすることによって、最高の仕事が成し遂げられるのだ。

プロジェクトの期間が6週間だったとしよう。あなたがコントロールできるのは自分の行動だけだ。6週間で仕事は完了するのだろうか？ 他人の予測はコントロールできない。だが、プロジェクトについて自分が知っていることを伝えることはできる。そうすれば、他人の予測と現実が一致する可能性が出てくる。この教訓を学んだとき、私の期日に対する恐怖は消え去った。私の仕事は他人の予測を「管理」することではない。それは私以外の人の仕事だ。私の仕事は全力を尽くし、明確に情報を伝えることである。

XPはソフトウェア開発の規律であり、開発プロセスのあらゆるレベルのリスクに対応するものである。XPは生産的であり、高品質のソフトウェアを生み出すこ

とができる。そして、実践するのがとても楽しい。XP では、開発プロセスのリスクにどのように対応するのだろうか？

- ✧ **スケジュールの遅延** —— XP のリリースサイクルは短期間（長くても数か月）なので、遅延の範囲は限定されている。リリースの期間中は、1 週間のイテレーションで顧客が要求したフィーチャー[†1]を作り、進捗を細かくフィードバックする。イテレーションの期間中は、短時間のタスクで計画を立てるため、チームはそのサイクルのなかで問題を解決できる。XP では、優先順位の高いフィーチャーから実装するので、リリースできなかったフィーチャーの重要度は低い。
- ✧ **プロジェクトの打ち切り** —— XP では、ビジネスに意味をもたらす最小限のリリースをチームのビジネス担当に選択してもらう。それにより、デプロイ前に道を誤ることは少なくなり、ソフトウェアのバリューは最大化する。
- ✧ **システムの劣化** —— XP では、包括的な自動テストスイートを作成／保守する。変更が（1 日に何度も）発生するたびに自動テストが実行／再実行され、ベースラインとなる品質が確保される。XP では、システムを常にデプロイ可能な状態に保つ。問題が蓄積することは許されない。
- ✧ **欠陥率の高さ** —— XP では、プログラマーが書く機能ごとのテストと、顧客が書くプログラムフィーチャーごとのテストの両方の観点からテストする。
- ✧ **ビジネスの誤解** —— XP では、ビジネス担当をチームのファーストクラスのメンバーとする。プロジェクトの仕様を開発のなかで継続的に洗練するため、顧客やチームの学びをソフトウェアに反映することができる。
- ✧ **ビジネスの変化** —— XP では、リリースサイクルを短くしている。そのため、ひとつのリリースの開発期間で発生する変更は小さい。リリースの期間中に顧客は、まだ完成していない機能と新機能を自由に入れ替えることができる。したがって、新しい機能なのか数年前からある機能なのか気づかないまま、チームが作業することもある。
- ✧ **必要のないフィーチャーばかり** —— XP では、優先順位の高いタスクだけを扱う。
- ✧ **人材の流出** —— XP では、作業の見積りと完了の責任をプログラマーに委ねたり、見積りの精度を高められるように実際にかかった時間をフィードバックしたり、プログラマーの見積りを重視したりしている。誰が見積りをするのか、誰が見積りを変更するのかは、明確にルールで決められている。した

[†1] 訳注：フィーチャー（*feature*）とは、ソフトウェアを使う側の視点から記述した機能のこと。詳しくは『アジャイルな見積りと計画づくり』（毎日コミュニケーションズ）の p.29 を参照してほしい。

がって、明らかに不可能なことを依頼されて、プログラマーが不満を募らせるようなことは少ない。XPでは、チームのなかで人間関係を築くことが推奨されている。仕事に対する不満の多くは、孤独が原因だからだ。XPには、新しいチームメンバーが参加したときの明確なモデルが用意されている。新しいチームメンバーには、お互いに助け合いながら、既存のプログラマーの支援を受けながら、少しずつ責任を受け入れることが求められる。

XPでは、あなたがチームの一員であると自覚して、理想的には明確なゴールと実行計画を共有していることを前提とする。XPでは、あなたが誰かと一緒に働きたいと思っていることを前提とする。XPでは、コストをかけずに変化が引き起こせることを前提とする。XPでは、あなたが自身の成長、スキルの向上、人間関係の改善を望んでいることを前提とする。XPでは、これらのゴールを達成するために、あなたが変化を引き起こすつもりがあることを前提とする。

それでは、本章で提起した「XPとは何か」の問いに答えよう。

- ✧ XPとは、効果のない技術的／社会的な古い習慣を捨て、効果のある新しい習慣を選ぶことである。
- ✧ XPとは、自分が今日やるべきことを十分に理解することである。
- ✧ XPとは、明日をよりよくしようとすることである。
- ✧ XPとは、チームのゴールに貢献した自分を評価することである。
- ✧ XPとは、ソフトウェア開発で人間としての欲求を満たすことである。

本書の残りの部分では、こうした変化を引き起こすために何ができるかを見ていくことにする。それから、なぜそれが個人的および経済的に有効なのかを考えていく。本書は2つの部に分かれている。第I部は実践的な内容であり、ソフトウェア開発の方法や考え方を説明している。いずれも人間の欲求を前提にして、それらを満たそうとするものである。そこには人間関係の欲求も含まれている。第II部では、XPの哲学や歴史的なルーツに触れ、XPを現在の文脈で捉え直している。

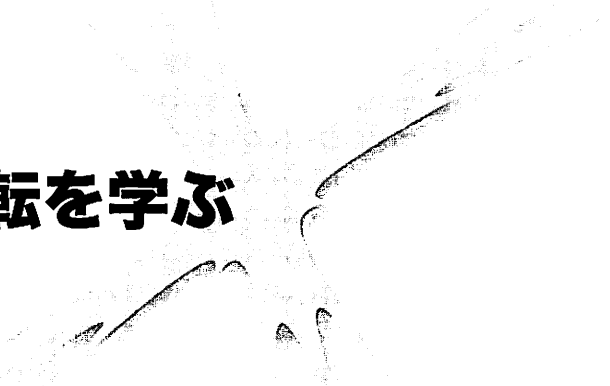
本書の読み方やXPの適用方法はさまざまである。暑い日に冷たいプールに入るのと同じだ。片足ずつ入る人もいれば、階段を着実に歩いて降りる人もいる。膝を抱えて勢いよく飛び込む人もいれば、競泳みたいに飛び込む人もいる。いずれもプールに入るという目的は同じだ。あなたがどれを選ぶかは、スタイル、スピード、効率性、あるいは恐怖によって決まる。どれが適しているかを決めることができるのは、あなただけだ。本書を読んだり適用したりするなかで、なぜ自分がソフトウェア開発に携わっているのか、どうすればこの仕事から満足感を得られるのかを深く理解してもらいたい。

第Ⅰ部

XPの探求

第2章

運転を学ぶ



車の運転をはじめて学んだ日のことをはっきりと覚えている。母親と私は、カリフォルニア州チコの近くにある州間高速道路5号線を走っていた。まっすぐで平らな道は、地平線まで続いていた。母は、助手席にいる私にハンドルを握らせた。ハンドルを動かすことで、車の方向が変わる感覚をつかませようとしたのである。それから私にこう言った。

「これが車の運転よ。車線の真ん中を走りなさい。地平線を目指すのよ」

私は慎重に道路の先に目を向けた。車線の真ん中をしっかりと意識して、きちんと車を走らせるようにした。うまくできた。だが、少しぼんやりしていると……。

車の足を砂利にとられ、我に返った。母が（その勇気に今も驚かされるのだが）車線の真ん中にそっと戻してくれた。私の心臓はバクバクしていた。母は私に車の運転を教えてくれた。

「運転というのはね、車を正しい方向に走らせることじゃないの。常に注意を払って、こっちに行ったら少し戻して、あっちに行ったら少し戻して、そうやって軌道修正していくものよ」

これがXPのパラダイムだ。注意して、適応して、変更する。

ソフトウェアはあらゆるものが変化する。要件も変化する。設計も変化する。ビジネスも変化する。技術も変化する。チームも変化する。チームメンバーも変化する。だけど、問題は変化ではない。変化はいずれにしても起きるものだ。問題はむしろ、我々が変化に対応できないことにある。

運転のメタファーは、2つの側面でXPに当てはめることができる。顧客はシステムの内容を「運転」する。チーム全体は開発プロセスを「運転」する。XPでは、

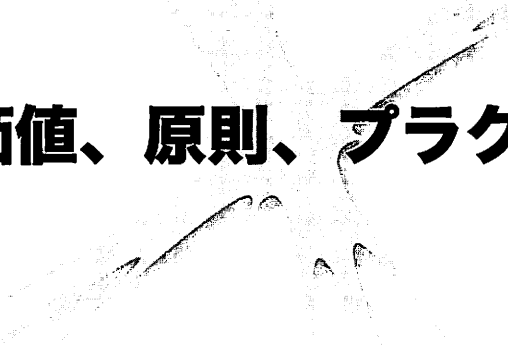
こまめに小さな軌道修正を加えながら、適応することができる。つまり、短い間隔でソフトウェアをデプロイしながら、ゴールに向かっていくことができる。道を間違えているかどうかが判明するまでに、時間がかかることはない。

顧客はシステムの内容を運転する。まずは（組織内または組織外の）顧客が、システムが解決すべき課題のアイデアを思いつく。だが、顧客はソフトウェアが何をすべきかは正確にはわからない。ソフトウェア開発が車の運転に似ているのはそのためだ。道路をまっすぐに走らせればいいわけではない。チームのなかにいる顧客は、チームが目指したい地平線を心にとどめておく必要がある。それと同時に、ソフトウェアが**次に**どこへ向かうかを毎週決める必要がある。

価値を表現するために行うことは、場所、時間、チームによって違う。システムの内容を運転しているのが顧客であるのと同様に、使用するプラクティスをはじめとした開発プロセスを運転しているのはチーム全体だ。チームで開発を続けていると、どのプラクティスがゴールを近づけ、どのプラクティスがゴールを遠ざけているかがわかるようになる。それぞれのプラクティスは、チームの能力、コミュニケーション、自信、生産性を向上させる実験だ。

第3章

価値、原則、プラクティス



ソフトウェア開発の新しい考え方や方法を明確に伝えるには、何が必要だろうか？ 造園の基本的な技術は、本ですぐに学ぶことができる。だが、それで造園家になれるわけではない。友人のポールは造園の専門家だ。私も土を掘ったり、植物を植えたり、水をやったり、除草したりはするが、造園の専門家ではない。

2人の違いは何だろう？ ポールは私よりも多くの技術を知っている。2人とも知っている技術については、彼のほうが私よりも詳しい。技術は重要だ。土を掘ったり、植物を植えたりできなければ、造園などできるはずもない。このレベルの知識や理解を**プラクティス**（*practices*）と呼ぼう。プラクティスは日常的な取り組みである。明確で客観的なので、具体的な形で表しておくと便利だ。たとえば、「コードを変更する前にテストを書く」というプラクティスは、先にテストを書くか書かないかのいずれかである。プラクティスには、何かを始めるきっかけになるという利点もある。ソフトウェア開発を深く理解していなくても、コードを変更する前にテストを書き始めることができるし、そこから恩恵を受けることもできる。

ポールの造園プラクティスをすべて知っていたとしても、やはり私は造園家ではない。ポールは造園の善しあしを判断する高度な感覚を持っている。彼は庭全体を見て、何がうまくいっていて、何がうまくいっていないかを直感的に把握できる。枝刈りがうまくできて私が得意気になっているとしたら、ポールは木全体の生え具合を見ているのである。それは、彼のほうが私よりも枝刈りがうまいからではない。庭に作用するフォース[†1]を彼が感じ取っているからである。彼にとっては簡単で明

[†1] 訳注：影響を与える力のこと。パターンランゲージの文脈では、問題解決における制約を意味する。XPはパターンランゲージの影響を受けているため「フォース」という言葉を選択した。

白であっても、私は懸命に取り組まなければいけない。

このレベルの知識や理解を**価値**（*values*）と呼ぼう。価値とは、ある状況における好き嫌いの根源にあるものだ。プログラマーが「自分のタスクを見積もりたくない」と言った場合、それは見積りの技法のことを話しているわけではない。すでに見積りは終わっているが、あとでそれが確固とした判断材料となり、自分が不利になることを恐れていて、その本音を明かしたくないのである。それなら見積りを3倍にしたほうがマシだ！ このことから、ソフトウェア開発における社会的な影響力の認識に深刻な問題があることがわかる。彼が見積りについて説明したがらないのは、おそらく不当に責められた経験があるからだろう。このプログラマーは、コミュニケーションよりも自己防衛に価値を置いている。価値とは、目にするものや考えていることなどを判断するための大きな基準である。

価値を明確にすることが重要だ。価値がなければ、プラクティスはすぐに機械的な作業になってしまう。活動そのものが目的となり、本来の目的や方向性が失われてしまう。プログラマーが欠陥を認めないとすると、それは技術ではなく価値の問題だろう。欠陥そのものは技術の問題かもしれないが、欠陥から学ぼうとしないのは、そのプログラマーが学習や自己改善に価値を置いていないことの表れだ。これでは、プログラム、組織、プログラマーのいずれの利益にもならない。プログラマーがプラクティス（ここでは**根本原因分析**）を効果的な時期に、正当な理由で実行できることが、価値とプラクティスが結び付いているという意味である。価値はプラクティスに目的をもたらしてくれる。

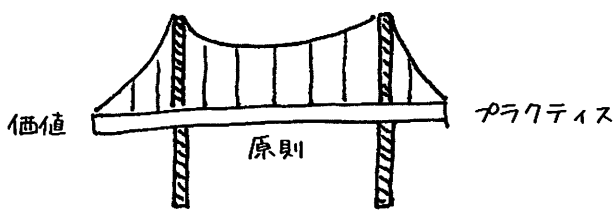
プラクティスは価値の証拠である。価値というのは、とても抽象的な表現なので、価値という名の下にどんなことでもできてしまう。たとえば、「この1,000ページのドキュメントを書いたのは、**コミュニケーションに価値を置いている**からです」と言うことも不可能ではない。それは正しいのかもしれないし、間違っているのかもしれない。だが、1日に15分間ほどの会話で、ドキュメントを書くよりも効果的にコミュニケーションができるとしたら、そのドキュメントはコミュニケーションに価値を置いていることにはならない。そのことを示すには、最も効果的なコミュニケーションの方法を実施しなければいけない。

プラクティスは疑う余地がない。私が朝のスタンドアップミーティングに参加したかどうかは、誰の目にもわかる。私がコミュニケーションに価値を置いているかどうかは、はっきりとはわからない。私がコミュニケーションを高めるプラクティスを継続しているかどうかは、具体的な事実だ。価値がプラクティスに目的をもたらすように、プラクティスは価値の説明責任を果たしているのである。

価値とプラクティスの間には、大きな隔たりがある。価値は普遍的なものである。

仕事で大切にしている価値は、普段の生活で大切にしている価値と同じであることが理想的だ。一方、プラクティスは状況によって大きく異なる。たとえば、プログラミングがうまくいっているかどうかのフィードバックが欲しいときは、ソフトウェアの継続的なビルドやテストに意味がある。だが、いつオムツを替えるべきかのフィードバックが欲しいときに、「継続的なビルドやテスト」があっても意味はない。2つの活動は関与するフォースが違うからだ。オムツ交換のフィードバックが欲しいなら、交換作業が終わってから赤ちゃんを持ち上げて、オムツがズレていないかを確認する必要がある。作業中に継続的に確認することはできない。オムツ交換とプログラミングの2つの活動では、「フィードバック」の価値の表現が大きく異なるのだ。

価値とプラクティスのギャップを埋めるのが**原則**（*principles*）だ（図1）。原則は、その分野に特化した活動の指針である。ポールの造園家としての知識は、原則という側面でも私を上回っている。イチゴの隣にマリーゴールドを植えることは私も知っているが、ポールは隣り合った植物がお互いの弱点を補う「コンパニオンプランティング（共生栽培）」の原則を理解している。マリーゴールドはイチゴを食べる虫を寄せつけない。これらを一緒に植えることはプラクティスである。コンパニオンプランティングは原則だ。本書では、XPの価値、原則、プラクティスについて説明している。



►図1

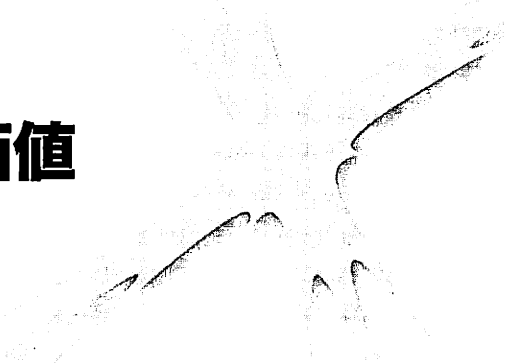
書籍で伝えられることには限界がある。きっかけにはなるが、XPを習得するには不十分だ。たとえ完璧な造園の本があったとしても、本を読むだけでは造園家にはなれない。まずは、実際に造園をやってみよう。それから、造園家のコミュニティーに参加しよう。そして、誰かに造園を教えてみよう。そうすれば、造園家だ。

XPも同じ。本書を読んでも、エクストリームプログラマーにはなれない。エクストリームなスタイルでプログラミングするには、XPの価値と少なくとも複数のプラクティスを共有したコミュニティーに参加して、自分の知っていることを誰かに伝えるしかない。

たとえXPの一部であっても、学習したり試したりすることは有益である。あなたの価値が何であっても、他にどのようなプラクティスを使っていても、コードの前にテストを書くことを学べばきっと役に立つ。ただし、私が手掛けた庭と専門家の造園に違いがあるように、XPを学ぶこととエクストリームにプログラミングすることには大きな違いがある。

第4章

価値



造園の専門家であるポールは、次に何を行うべきかの直感を持っている。何が重要であり、何が重要でないかを直感的にわかっている。私はまっすぐに植えることが重要だと考えていたので、どうにかしてまっすぐに植えられるように努力していた。するとポールがやって来て、このように言うのである。

「どうしてまっすぐに植えようと必死になっているんですか？ あなたに必要なのは堆肥ですよ」

私が大切だと考えていたものと、本当に大切なものは違っていた。それがムダを生み出していたのである。

ソフトウェア開発にかかわる人であれば、何が重要であるかの感覚を持っている。考えられるすべての設計を実装前に注意深く考えることが重要だと考える人もいれば、個人の自由が制限されないことが重要だと考える人もいる。

ウィル・ロジャース[†1]が言うように、「トラブルに巻き込まれる原因は、何かを知らないことではない。よく知らないのに、知っていると勘違いすることである」。ソフトウェア開発の「勘違い」のなかで、私が最も大きな問題だと思うのは、個人の行動に集中してしまうことだ。本当に重要なのは、個人としてではなく、チームや組織の一員としてどのように振る舞うかである。

たとえば、誰もがコーディングスタイルに夢中になる。確かに優れたスタイルや、よくないスタイルは存在する。だが、最も重要なのは、チームが共通のスタイルを目指すかどうかだ。独自のコーディングスタイルと、それが示す価値「何としても

---

†1 訳注：アメリカの俳優、コメディアン。「ウィル・ロジャース現象」として、その名を知られる。

個人の自由を守る」は、チームの成功にはつながらない。

全員がチームにとって大切なことに集中するとしたら、何に集中すべきだろうか？ XPでは、開発を導く5つの価値を採用している。コミュニケーション、シンプリシティ、フィードバック、勇気、リスペクトだ。

コミュニケーション（Communication）

チームによるソフトウェア開発で最も重要なのは、コミュニケーションである。開発中に問題が発生したときには、すでに誰かが解決策を知っていることが多い。だが、その情報は変更する権限のある人には伝わらない。情報があるのに伝わらないのは、自分の直感を無視するときの心の内面と同じだ。他人とコミュニケーションする必要があるときには、その影響の度合いはさらに悪化する。

コミュニケーションの欠如ではなく、情報の欠如によって問題が発生することもある。そのような問題は事前に対処できない。まだ何も知らないからだ。「ポーランドのWindowsは、Ctrl-shift-Sの割り当てが違うんだ[†2]。こんなの誰がわかるんだよ？」。予想外の問題に遭遇したあとであれば、コミュニケーションが解決につながる可能性もある。過去に同じような問題を経験した人に話を聞くこともできるし、問題の再発防止についてチームで話し合うこともできるからだ。

こんなことを言うと、みんなが「思いやりと分かち合い」のもとに集まって座り、コーヒーを飲みながら永遠にしゃべってばかりで、誰ひとり何もしようとしない状況を思い浮かべるかもしれない。チームはコミュニケーション以外にも価値を持っているので、このような状況になることはない。とはいえ、コミュニケーションを伴わない行動はうまくいかないものである。

問題に遭遇したときは、それがコミュニケーションの欠如によるものかどうかを自問してみよう。今から問題に対応するには、どのようなコミュニケーションが必要だろうか？ これからトラブルに巻き込まれないようにするには、どのようなコミュニケーションが必要だろうか？

コミュニケーションは、チーム感覚や効果的な協力関係を生み出すために重要なものである。だが、効果的なソフトウェア開発に必要なのは、コミュニケーションだけではない。

[†2] 訳注：通常は［別名で保存］のショートカットになるが、ポーランドのWindowsでは、アキュートと呼ばれるアクセント記号のついた「ś」の入力になる。

シンプリシティ（Simplicity）

シンプリシティは、XP の価値のなかで最も知的な部分である。システムをシンプルに保ち、今日の問題だけを優雅に解決するのは大変な作業だ。昨日シンプルだった解決策は、今日もシンプルのままかもしれない。だが、単純すぎたり、複雑すぎたりすることもあるだろう。シンプリシティを取り戻すために変更が必要なときは、現在地から目指すべき地点までの道筋を見つけ出さなければいけない。

私は「最もシンプルで、うまくいきそうなものは何ですか？」と質問するようにしている。批判する人たちは、質問の後半部分を見逃しているようだ。「我々には深刻なセキュリティや信頼性といった制約がありますので、システムをシンプルにはできません」などと返ってくる。私は、シンプルすぎてうまくいかないものについて質問しているわけではない。ムダな複雑性を排除するために、何ができるかを考えてもらっているのだ。セキュリティの観点からシステムを 2 台のマシンに分散する必要があれば、おそらくそれはシンプルであるといえるだろう。1 台のマシンでセキュリティを確保する解決策が見つからない限り、それが最もシンプルな解決策である。

シンプリシティは状況によって異なる。パーサージェネレーターを理解したチームとパーサーを作っているとしたら、パーサージェネレーターを使うことがシンプルだ。パーサーについて何も知らず、処理する言語が複雑ではないなら、再帰下降パーサーを使うほうがシンプルである。

価値は、お互いにバランスをとり合ったり、サポートし合ったりするものである。たとえば、コミュニケーションによって、現状の観点からは必要がない、あるいは時間的猶予のある要件を削除すれば、それがシンプリシティの達成につながる。シンプリシティを達成すれば、必要なコミュニケーションも少なくなる。

フィードバック（Feedback）

最初に決まった方向性が、長期間そのまま有効であることはない。それは、ソフトウェア開発の詳細であっても、システムの要件であっても、システムのアーキテクチャであっても同じだ。経験する前に方向性を決めてしまうと、すぐにダメになってしまう。変化は避けられないものであり、変化がフィードバックを必要にするのである。

デンマークのオーフスで行った 1 日がかりのプレゼンテーションが思い出される。最前列にいた観客の顔が次第に曇っていったのだ。彼はついに我慢できなくなり、「最初から正しくやるほうが簡単ですよね？」と言った。もちろんそうだ。ただ

し、以下の3つの場合は例外である。

- ✧ 「正しく」やる方法がわからない場合。これまでになかった新しい問題を解決するときは、うまくいきそうな解決策が複数考えられたり、そもそも明確な解決策が存在しなかったりする。
- ✧ 今日は正しかったとしても、明日は間違っている可能性がある場合。コントロールや予測ができる範囲を超えた変化が発生すると、昨日の決定はあっさり無効になってしまう。
- ✧ すべてを「正しく」やろうとして、時間がかかりすぎる場合。解決策が完成する前に環境が変わると、その解決策は無効になってしまう。

一時的な完成に期待するよりも、常に改善を続けていこう。そうすれば、フィードバックを使って、ゴールに近づくことができる。フィードバックには、以下のようなさまざまな形式がある。

- ✧ アイデアに対するあなたやチームメイトの意見
- ✧ アイデアを実装したときのコードの状態
- ✧ テストは書きやすいか
- ✧ テストは実行できているか
- ✧ 実現したアイデアがうまく機能しているか

XPチームはできるだけ早く、できるだけ多くのフィードバックを生み出そうとする。フィードバックのサイクルを数週間や数か月ではなく、数分や数時間に短縮しようとする。フィードバックが早く手に入れば、その分だけ早く適応できる。

ただし、フィードバックが多すぎる場合もある。チームが重要なフィードバックに気づかないようであれば、負担になっている可能性が高い。その場合は、反応できるようになるまで、フィードバックのペースを落とす必要がある。そうすれば、フィードバックが多すぎるという根本的な問題にチームが対応できるようになる。たとえば、四半期ごとにリリースしていたとして、次のリリースまでに対応できないほどの不具合レポートが突然出てきたとする。そのようなときは、不具合レポートに対応しながら、新機能の開発ができるようになるまでリリースを遅らせるべきだ。十分な時間を作り、なぜそれほど不具合が多いのか、なぜ不具合の特定に時間がかかったのかを解明しよう。根本的な問題が解決できれば、フィードバックの速度を上げて、四半期ごとのリリースを再開できるだろう。

フィードバックはコミュニケーションに欠かせない。「パフォーマンスは問題になるかな？」「わからないね。パフォーマンス測定用のプロトタイプを作って確認し

てみよう」。フィードバックはシンプリシティにも影響する。3 つの解決策のなかで、どれが最もシンプルになるだろうか？ そんなときは、3 つすべてを試して確認してみよう。同じものを 3 回実装するのはムダに思えるかもしれない。だが、シンプリシティが備わった納得できる解決策にたどり着くには、こうするのが最も効率的な方法だろう。それと同時に、システムがシンプルになれば、その分だけフィードバックを受け取ることも簡単になる。

勇気（Courage）

勇気とは、恐怖に直面したときの効果的な行動のことである。暗い出入り口を巡回する兵士の行動を表現する言葉なので、軽々しく「勇」の言葉を使うべきではないと主張する人もいる。兵士が勇ましく体を張っていることは認めるが、ソフトウェア開発の人たちも確実に恐怖を感じている。この恐怖に対処できるかどうかによって、チームの一員として効果的に働けるかどうかが決まる。

時として勇気は、行動の姿勢となって現れる。問題がわかっていれば、それに対応する行動をとるべきだ。時として勇気は、忍耐となって現れる。問題があることがわかっていても、何が問題かがわからなければ、明確にわかるまで勇気を持って待機すべきだ。

価値のバランスを考えずに、勇気を最優先の価値にするのは危険である。結果を考慮せずに行動していては、チームワークがうまくいっているとはいえない。恐怖を感じたときには、その他の価値にも注目して、チームワークを高めよう。

勇気のみでは危険だが、他の価値と合わせれば強力だ。勇気を持って真実を語れば（たとえそれが不愉快なことであっても）、コミュニケーションや信頼が強化されていく。うまくいかない解決策を捨てて、勇気を持って新しい解決策を見つければ、シンプリシティが促進される。勇気を持って現実の具体的な答えを求めれば、そこからフィードバックが生まれる。

リスペクト（Respect）

これまでの 4 つの価値は、水面下にあるもうひとつの価値を指し示している。それは、リスペクトだ。チームメンバーがお互いに関心がなく、何をしているかを気にもとめないようであれば、XP はうまくいかない。チームメンバーがプロジェクトを大切にしないのであれば、何をしたところで救えるはずもない。

ソフトウェア開発に関係している人は、人間として等しく重要である。他の人よりも本質的に重要な人などいるはずがない。ソフトウェア開発において人間性と生

産性を同時に高めるには、チームに対する個人の貢献をリスペクトする必要がある。私も重要であり、あなたも重要だ。

その他の価値

効果的なソフトウェア開発に必要な価値は、コミュニケーション、シンプリシティ、フィードバック、勇気、リスペクトだけではない。これらはXPの原動力となる価値である。組織、チーム、あなた自身が、その他の価値を選択しても構わない。最も重要なのは、チームの振る舞いをチームの価値に合わせることである。そうすれば、複数の価値を同時に維持するムダを最小化することができる。

その他の重要な価値としては、安全性、セキュリティ、予測可能性、生活の質などがある。チームがこれらの価値を共有すれば、XPの価値がプラクティスを作り出すのとは違ったやり方で、自分たちのプラクティスを作り出すことができるだろう。

価値は、ソフトウェア開発で**何をすべきか**といった具体的なアドバイスを提供するものではない。価値とプラクティスには隔たりがあるので、そのギャップを橋渡しする方法が必要だ。そこで必要となるのが原則である。プラクティスに進む前に、次の章ではXPの原則を紹介しよう。原則とは、XPの価値に調和したプラクティスを探すという分野に特化した指針である。

第5章

原則

価値は抽象的すぎるので、そのままでは振る舞いの指針にはならない。大量のドキュメントはコミュニケーションを目的としたものだが、会話も同じ目的を持っている。では、どちらがより効果的なのだろうか？ その答えは、文脈によって決まる部分もあれば、原則によって決まる部分もある。この場合は、人間の基本的な欲求である「つながり」が会話によって満たされること、その他の条件が等しければ、会話のほうが好ましいコミュニケーション方法であることが、人間性の原則によって示されている。ドキュメントによるコミュニケーションは本質的にムダが多い。多くの人に情報は伝わるかもしれないが、コミュニケーションは一方向である。会話であれば、明確化、迅速なフィードバック、一緒にブレインストーミングするなど、ドキュメントではできないことが可能になる。ドキュメントによるコミュニケーションは、事実として受け止められるか、真っ向から否定されるかのいずれかになりやすい。いずれにしてもコミュニケーションの増加にはつながらない。

ソフトウェア開発の指針となる原則は、ここに列挙するものだけではない。たとえば、セーフティクリティカルなシステムの開発では、トレーサビリティの原則が必要になる。つまり、完了した作業から、ユーザーが明確に表したニーズを追跡できなければいけない。作業そのものを目的とすべきではない。セーフティクリティカルなシステムに携わっているのであれば、システムの認証を得るためにもトレーサビリティの原則が重要になるだろう。ただし、これはすべてのソフトウェアに当てはまるものではないため、以下の一覧には含めなかった。チームのプラクティスの指針となる原則は他にもあるだろうが、これから紹介するものがXPの指針となる原則である。

人間性（Humanity）

人間がソフトウェアを開発する。このシンプルで逃れようのない事実によって、利用可能な方法論の多くがその効果を失っている。ほとんどの場合、ソフトウェア開発は人間の欲求を満たしていない。人間の弱さを認めていない。人間の強さを活用していない。ソフトウェアを人間が開発していないかのように振る舞えば、関係者にかかるコストが高くなり、人間の欲求を認めない非人道的な行為によって、人間性が失われてしまう。こうしたことは、高い離職率に伴うコストや組織の崩壊、クリエイティブな行動の機会損失など、ビジネスにとっても好ましいことではない。

優れた開発者になるには、何が必要だろう？

- ✧ **基本的な安全性** —— 空腹、身体的な危害、愛する人を危険にさらすものが存在しないこと。失業の不安はこうした欲求を脅かす。
- ✧ **達成感** —— 自分が所属する社会に貢献する機会や能力。
- ✧ **帰属意識** —— グループに所属して、承認や説明責任を受け取ったり、共通の目標に貢献したりすること。
- ✧ **成長** —— スキルや視野を広げる機会。
- ✧ **親密な関係** —— 他人を理解して、他人から深く理解されること。

XP のプラクティスには、ビジネスニーズと個人の欲求の両方を満たすものを選んでいる。人間の欲求には、休息、運動、社会化といったものもあるが、これらは仕事場で満たす必要はない。チームから離れているときに手に入れた活力や新たな視点は、チームに持ち帰ることもできる。労働時間を制限すれば、こうした人間の欲求を満たす時間を確保したり、チームと一緒にいるときの個人の貢献度を高めたりすることができる。

チームによるソフトウェア開発で難しいのは、個人の欲求とチームのニーズの両方のバランスをとることだ。個人の長期的な目標とチームのニーズが合っていれば、多少の犠牲は払うべきだろう。だが、常にチームのために個人の欲求を犠牲にしているようでは、うまくいくはずがない。プライバシーが必要であれば、チームに損害を与えないように、その方法を自分で探すべきだろう。優れたチームが素晴らしいのは、メンバーたちが信頼関係を築いてから一緒に働くことによって、みんなが**自分らしく**いられることである。

親密な関係になれば気分はいいが、それでも仕事は仕事である。私生活のことを詳しく語られると、チームのコミュニケーションに混乱をきたす。あるチームと話をしたことがある。そのチームには、チームになじんで以来、私生活のことを毎朝詳しく語りたがるメンバーがいた。彼の話を聞きたい人などいなかったが、誰も彼

に対抗する手段を知らなかった。結局、年長のチームメンバーが彼を呼び出して、私生活のことを話さないでほしいとお願いすることになったそうだ。

私は自分の生活を、妻とだけ話す私生活のこと、信頼できる人とだけ話す個人的なこと、誰とでも話せる一般的なことに分けている。どの話がどこに当てはまるのか、誰が信頼に値するのかを判断するのは簡単なことではない。だが、このようにうまく区別することができれば、仕事のコミュニケーションが効果的になり、人生のあらゆる場面で人間関係が重要なものとなる。

経済性（Economics）

誰かがお金を支払わなければいけない。経済性を認識していないソフトウェア開発は、「技術的な成功」という空虚な勝利に終わる危険性をはらんでいる。自分のやっていることが、ビジネスバリューにつながり、ビジネスゴールを達成し、ビジネスニーズを満たせるようにしよう。たとえば、最優先のビジネスニーズを最初に解決すれば、プロジェクトのバリューを最大化できる。

ソフトウェア開発に影響を与える経済性には 2 つの側面がある。貨幣のタイムバリューと、システムやチームのオプションバリューだ。貨幣のタイムバリューでは、今日のお金は明日のお金よりもバリューがあるとされる。ソフトウェア開発の場合は、早めにお金を稼ぎ、あとでお金を使うほうがバリューが高い。インクリメンタルな設計にすれば、設計の投資を最終責任時点まで遅らせて、あとからお金を使うことができる。利用都度課金にすれば、フィーチャーのデプロイ直後から収益を得ることができる。

ソフトウェア開発の経済性のもうひとつの源泉は、将来のオプションバリューである。たとえば、メディアのスケジューリングプログラムを開発したとしよう。それを他のスケジューリングのタスクにも再利用できたとすれば、本来の目的だけに使うよりもバリューが高い。貨幣のタイムバリューを気にかけながら、リスクの高い柔軟性に投資することなく、ソフトウェアとチームの両方のオプションバリューを高める。そのために、すべてのプラクティスが用意されているのである。

相互利益（Mutual Benefit）

あらゆる活動は、関係者全員の利益にならなければいけない。相互利益とは、最も重要であり、最も実行が難しい XP の原則である。いかなる問題であっても、ある人にとっては利益になるが、ある人にとってはコストになる解決策が常に存在する。絶望的な状況であれば、そのような解決策も魅力的に思えるだろう。だが、そ

れでは常に純損失が生じてしまう。そこから敵意が生まれ、大切にすべき人間関係を引き裂くことになるからだ。コンピュータービジネスとは、人間のビジネスであり、仕事上の人間関係を維持することが大切である。

ソフトウェアに関する大量の内部ドキュメントは、相互利益を破壊するプラクティスの一例である。将来の見知らぬ誰かがコードを保守しやすいように、現在の開発速度を大幅に下げろと言われているのと同じだ。ドキュメントが有効なものであれば、将来の人間の利益にはなるだろう。だが、現時点での利益はない。

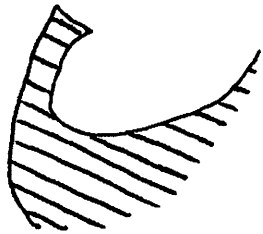
XPでは、このような将来とのコミュニケーションの問題を相互利益になるやり方で解決している。

- ✧ 現時点で設計や実装がうまくできるような自動テストを書く。また、テストは将来のプログラマーにも使えるように残しておく。こうしたプラクティスは、現在の自分と将来の保守担当者の両方の利益になる。
- ✧ 意図しない複雑性を排除するために、注意深くリファクタリングする。それにより、自分の満足感が得られ、欠陥が少なくなる。また、あとでコードに触れた人が理解しやすくなる。
- ✧ 一貫性のある明確なメタファーから名前を選ぶ。それにより、自分の開発速度が上がる。また、新しく参加したプログラマーにとってコードが明確になる。

アドバイスを聞き入れてもらうには、対象とする問題よりも多くの問題を解決する必要がある。XPの相互利益とは、現在の自分、将来の自分、顧客に対する利益を求めるプラクティスだ。Win-Win-Winのプラクティスは受け入れられやすい。差し迫った苦痛を取り除いてくれるからだ。たとえば、難しい欠陥に取り組んでいる人は、テストファーストプログラミングを学ぶ準備ができている。現在の自分の利益になるのであれば、他人や将来の役に立つことを受け入れやすい。

自己相似性（Self-Similarity）

サルディーニャ島の海岸を散歩したときのことだ。図2のような形状をした60センチくらいの小さな潮だまりを見つけた。目を上げてから気づいたのだが、散歩していた湾と形状がほとんど同じだった。こちらは1.6キロメートルくらいはあっただろうか。「なんて素晴らしい地質学のフラクタル性なんだ」と、ひそかに思った。この輪郭はサルディーニャ島の北西部の地図をトレースしたものだ。自然はうまくいった形状を見つけると、それをあらゆるところに使おうとする。



▶図2　自然に発生する形

ソフトウェア開発にも同じ原則が当てはまる。規模が違っていても、解決策の構造を新しい文脈にコピーしようとするからだ。たとえば、先に失敗するテストを書いてから、それを動かすという開発の基本的なリズムがある。このリズムはあらゆる規模で作用する。四半期単位では、扱いたいテーマを一覧にして、ストーリーを使って取り組んでいく。週単位では、扱いたいストーリーを一覧にして、ストーリーを表すテストを書いて、それを動かすようにする。数時間単位では、書かなければいけないテストを一覧にして、テストを書いて、動かして、別のテストを書いて、その両方を動かして、これをテストの一覧が終わるまで繰り返す。

ソフトウェア開発に有効な原則は、自己相似性だけではない。ある文脈でうまくいった構造をコピーしたからといって、他の文脈でうまくいくとは限らない。とはいえ、まずはそこから始めるのがいいだろう。同様に、解決策が独特だからといって、それが悪いとは限らない。その状況においては、独特な解決策が求められていた可能性もある。

第1版の週次サイクルのアドバイスは、先にコードを書いてから、動作確認のためにテストするという、まるでウォーターフォールのようなものだった。今考えれば、自己相似性に注目すべきだった。実装前にシステムレベルのテストを行えば、設計がシンプルになり、ストレスが減り、フィードバックが改善される。

改善（Improvement）

ソフトウェア開発においては、「完璧（*perfect*）」は動詞であり、形容詞ではない。完璧なプロセスは存在しない。完璧な設計は存在しない。完璧なストーリーは存在しない。だが、プロセスを「完璧にやる」ことはできる。設計を「完璧にやる」ことはできる。ストーリーを「完璧にやる」ことはできる。

「完璧は善の敵」という言葉がある。完璧を求めるよりも、平凡のほうが望ましいという意味だ。だが、この言葉はXPのポイントをつかめていない。XPのポイント

は、改善によってソフトウェア開発の高みを目指すことだ。改善のサイクルでは、明日をよりよくするために必要な気づきや理解を追い求めながら、今日できる最高のことをやる。完璧になるのを待ってから始めるわけではない。

改善の原則が見られるのは、すぐに行動に着手するが、時間をかけてその結果を改良していくようなプラクティスを価値から転換するときである。たとえば、**四半期サイクル**には、経験を踏まえながら長期計画を改善する可能性が示されている。**インクリメンタルな設計**によってシステムの設計を洗練していけば、改善をうまく機能させることができる。実際の設計は理想を完璧に反映したものには決してならないが、両者を日々近づけていくことならできる。

ソフトウェア開発技術の歴史を紐解くと、ムダな労力が少しずつ排除されてきたことがわかる。たとえば、シンボリックアセンブラによって、マシン命令を物理的なビット符号に翻訳するというムダで退屈な作業が排除された。「自動プログラミング」によって、抽象的なプログラム記述をアセンブリ言語に翻訳するというムダで退屈な作業が排除された。こうした流れは、ストレージを自動解放するところまできている。

テクノロジーの改善によってムダが排除されてきた一方で、開発組織においては硬直化や専門化が高まり、ますますムダが発生している。改善の鍵は、この2つを調和させることである。つまり、新しく発見された技術的な効率化を用いて、より効果的で新しい社会的な人間関係を可能にするのである。完璧を待たずに改善を機能させよう。出発点を見つけて、そこから始めよう。そこから改善していこう。

多様性（Diversity）

みんながよく似ているソフトウェア開発チームは、居心地はよいかもしれないが、機能的ではない。チームは、問題や落とし穴を見つけたり、問題を解決する方法を複数考えたり、解決策を実現したりするために、さまざまなスキル、考え方、視点を組み合わせる必要がある。チームには多様性が必要だ。

多様性には衝突がつきものである。これは「お互いに憎み合っているので前に進まない」という衝突ではなく、「これを解決するには2つの方法がある」という意味での衝突だ。あなたならどのように選ぶだろうか？

設計のアイデアが2つあれば、それは問題ではなく機会である。プログラマーは協力して問題に取り組むべきであり、どちらの意見も評価されるべきだ。それが、多様性の原則が意味するところである。

衝突の扱いがうまくないチームはどうすればいいのだろう？ 衝突が発生しないチームなど存在しないので、生産的に衝突を解決できるかどうかを考えてみよう。

みんなをリスペクトして、自分の言い分を主張すれば、ストレスのかかる状況であってもコミュニケーションは円滑になるはずだ。

多様性は**チーム全体**のプラクティスで表現されている。このプラクティスは、多種多様なスキルや視点を持った人たちをチームにまとめるというものだ。異なる視点を持った人たちが、さまざまな計画サイクルを使い、時間内に最もバリューの高いソフトウェアを作り出すという目的を達成するのである。

ふりかえり（Reflection）

優れたチームは単に仕事をしているだけではない。**どうやって**仕事をしているのか、**なぜ**仕事をしているのかを考えている。なぜ成功したのか、なぜ失敗したのかを分析している。自分たちのミスを隠そうとはしない。それを明らかにして、そこから学ぼうとするのである。偶然に優秀になれる人などいないのだ。

四半期サイクルや**週次サイクル**には、**ペアプログラミング**や**継続的インテグレーション**だけでなく、チームが過去をふりかえる時間も含まれている。ただし、ふりかえりは「公式」の機会に限定すべきではない。配偶者や友人との会話、休暇、ソフトウェアとは関係のない読書や運動といったものすべてが、今どのように仕事をしているのか、なぜそのように仕事をしているのかを個人で考える機会となる。食事やお茶を共にすれば、非公式な場で一緒にふりかえることもできる。

ふりかえりは、知力だけを必要とする行為ではない。データを分析することで知見を手に入れることもできるが、自分の直感から学ぶこともできる。恐怖、怒り、不安といった「否定的」な感情は、昔から何か悪いことが起こりそうなときの前触れだ。自分の感情が仕事について何を訴えているのか、それに耳を傾けるには努力が必要だが、知力に感情が伴えば、新たな知見の源泉になるだろう。

ふりかえりが度を超す場合もある。ソフトウェア開発の現場には昔から、**ソフトウェア開発のこと**を考えるのが忙しくて、ソフトウェアを開発する時間のない人たちがいる。ふりかえりは、行動したあとにやるものだ。そこから学習につながっていく。フィードバックを最大化するには、XP チームのふりかえりを行動と組み合わせるべきである。

流れ（Flow）

ソフトウェア開発の流れとは、すべての開発作業を同時に行い、バリューのあるソフトウェアの安定した流れを生み出すことである。XP のプラクティスは、不連続のフェーズよりも継続的な流れに適している。

ソフトウェア開発は、昔から大きなかたまりでバリューを届けてきた。「ビッグバン」インテグレーションは、こうした傾向を反映したものである。多くのチームは、ソフトウェアのデプロイやインテグレーションの頻度を下げることにより、バリューのかたまりをさらに大きくして、ストレスに対処しようとしている。そして、これが問題を悪化させている。フィードバックが減ると問題が悪化して、さらにかたまりが大きくなってしまう。遅れるものが増えると、かたまりがまた大きくなり、リスクが高まっていく。流れの原則では、改善のために小さなバリューを何度もデプロイすることを提唱している。

ソフトウェア開発のトレンドのなかには、大きなかたまりに反対しているものもある。たとえば、デイリービルドは流れを意識している。だが、それは流れにつながる小さな一歩にすぎない。毎日コンパイルしたり、リンクしたりするだけでは不十分である。1日1回（できれば数回）機能を正しく動作させるべきだ。

1週間ごとにデプロイしていたチームを訪問したことがある。そのチームはどんどん問題を抱えていき、ついには1週間かけて作ったソフトウェアを6日間かけてデプロイするまでになった。チームは2週間ごとにデプロイする道を選んだのである。これにより、インテグレーションとデプロイの問題が悪化することになった。流れから遠ざかったのであれば、元に戻さなければいけない。流れを滞らせる問題を解消して、できるだけ早く1週間ごとのデプロイに戻すべきだ。

機会（Opportunity）

問題を変化の機会と考えよう。ソフトウェア開発に問題がないというわけではない。「サバイバル」の姿勢では、その場を切り抜けるための問題解決しかできないということだ。高みに到達するには、問題を学習や改善の機会に変換しなければいけない。サバイバルだけでは不十分だ。

問題の対処方法がわからないこともあるだろう。何をすべきかを時間をかけて考えたいこともあるだろう。時間を求めてしまうのは、問題に対処した結果を目の当たりにする恐怖から自分を守る仮面になるからだ。とはいえ、しばらく我慢していれば、自然と時間が問題を解決してくれることもある。

問題を機会へと変換しよう。それを開発プロセスのあらゆる場面で実施しよう。それによって、強みを最大化し、弱みを最小化するのである。長期的な計画が正確にできない？ よろしい、ならば**四半期サイクル**で計画を改良だ。ひとりでやるとミスが多すぎる？ よろしい、ならば**ペアプログラミング**だ。人間が協力してソフトウェアを開発する。そのような普遍的な問題に取り組んでいるからこそ、これらのプラクティスは効果的なのである。

XP の実践を始めると、必ず問題に直面する。エクストリームになるというのは、それぞれの問題を個人の成長、人間関係の深化、ソフトウェアの改善などの機会に意識的に変換することでもある。

冗長性（Redundancy）

そう、冗長性だ。ソフトウェア開発の重要で困難な問題は、複数の方法で解決すべきである。ひとつの解決策が完全に失敗しても、その他の解決策で惨事を食い止めることができるからだ。冗長性にコストがかかるとしても、惨事から救われるならそのコストは支払うに値する。

たとえば、欠陥はムダの排除に有効な信頼関係を蝕む。欠陥は深刻で難しい問題である。XP では、ペアプログラミング、継続的インテグレーション、全員同席、本物の顧客参加、デイリーデプロイなど数多くのプラクティスで欠陥に対処する。パートナーがエラーを捕捉できなくても、周囲に座っている誰かが見つけてくれるだろう。あるいは、次のインテグレーションで捕捉できるだろう。同じ欠陥を捕捉しているという意味では、明らかに冗長なプラクティスも含まれている。

欠陥の問題は、ひとつのプラクティスだけでは解決できない。非常に複雑で多面的であり、完全に解決することなどできないだろう。したがって、チーム内の信頼関係と顧客との信頼関係を維持できるようになるまで欠陥を減らせれば、それでいいのである。

冗長性はムダにつながる可能性がある反面、目的の達成に必要な冗長性は排除しないように気を付けておかなければいけない。たとえば、開発終了後にテストフェーズを設けるのは冗長だと思うだろう。だが、テストフェーズを排除していいのは、何度かのデプロイで連続してひとつも欠陥が発見されず、テストフェーズが本当に冗長であることが実証された場合だけだ。

失敗（Failure）

うまく成功できなければ、失敗しよう。ストーリーを実装する 3 つの方法のうち、どれを選んだらいいかわからないだって？ それなら、3 つすべてを試してみよう。すべてが失敗に終わるとしても、重要なことが学べるはずだ。

失敗はムダではないのだろうか？ 知識が身に付けばムダではない。知識は貴重であり、獲得が難しいときもある。失敗は回避できるムダではない可能性もある。事前にストーリーの最適な実装方法がわかっていれば、その方法で実装すればいい。わかっていなければ、コストをかけずに見つける方法はないだろうか？

優秀な設計者が何人もいるチームをコーチしたことがある。彼らは、与えられた問題にそれぞれが2〜3通りの解決方法を考え出すことができた。何時間も座り込み、各アイデアについて順番に議論するのである。議論に疲れ果てるほどの時間があれば、すべてのアイデアを2回ずつ実装できたはずだ。プログラミングで時間をムダにしたくないがために、議論で時間をムダにしていたのである。

私はキッチンタイマーを買い与え、設計の議論を15分に制限してもらうようにした。タイマーが鳴ったら、誰か2人が何らかの実装をするのである。結局、数回しか使うことはなかったが、議論ではなく失敗することを思い出すために、彼らは常に手元にタイマーを置いていた。

このことは、よりよい方法を知っていながら失敗することを正当化するものではない。何をすべきかがわからないときには、失敗のリスクを受け入れることが成功につながる最短で確実な道である。

品質（Quality）

品質を犠牲にするのは、効果的なコントロール方法ではない。品質は制御変数ではない。低品質を受け入れることで、プロジェクトが速くなることはない。高品質を要求することで、プロジェクトが遅くなることもない。むしろ品質を高めることで、デリバリーが高速になることが多い。品質基準を下げてしまうと、デリバリーが遅くなり、予測できなくなってしまう。

第1版を出版してから最も驚いたのは、多くのチームが、欠陥数、設計品質、開発経験を使って、品質を計測できるようになったことだ。品質が高まるたびに、生産性や有効性などのプロジェクトの特性が改善されていくのである。品質による恩恵は計り知れない。限界があるとすれば、高品質を実現する方法を自分たちの能力が理解できるかどうかだけだ。

品質に求められるのは経済的要因だけではない。人間は自分が誇りに思える仕事を必要としている。以前、平均的なチームのマネージャーと話をしたことがある。彼は週末になると家に帰り、鍛冶職人のように鉄製の工芸品を作っていた。そこで品質に対する欲求を満たしていたのである。つまり、欲求を仕事以外で満たしていたのだ。

品質のコントロールによってプロジェクトを管理できないとしたら、どうすれば管理できるのだろうか？ 時間とコストは固定されていることが多いため、XPでは、プロジェクトの計画、追跡、運営の主な手段として、スコープを選択している。スコープは事前に正確に把握することはできないので、優れた制御レバーになる。スコープの追跡や選択は、**週次サイクル**や**四半期サイクル**のなかで行う。

品質が心配だからといって、何もしないことの言い訳にしてはいけない。きれいに実装する方法がわからなければ、できる限りのことをすればいい。きれいだが時間のかかる方法を知っていれば、今の時間でできる限りのことをすればいい。そして、あとからきれいな方法で完成させるのだ。こうしたやり方は、アーキテクチャを進化させるときにもよく使う。同じ問題を解決する２つのアーキテクチャを共存させ、少しずつ片方に移行していくのである。そうすれば、移行そのものが品質の実証になる。大きな変更を効率的に、安全で小さなステップで行うのだ。

ベイビーステップ（Baby Steps）

大きな変更は、大きなステップでやりたくなるものである。距離が長く、時間をかけずにそこまで行くとなれば、そうするしかないように思える。だが、重大な変更を一気に行うのは危険だ。変更を頼まれるのは人間である。変更は不安を伴う。不安を伴えば、人間は変更をすばやくやろうとする。

私は「あなたができる最も小さなことで、正しい方向がすぐにわかるものは何ですか？」とよく質問している。ベイビーステップは、静止状態や変化の遅さを正当化するものではない。条件が整っていれば、人やチームは小さなステップを高速に繰り返し、まるで跳躍しているかのように見せることもできる。

大きな変更が失敗してチームがムダに後退するよりも、小さなステップのオーバーヘッドのほうが小さい。ベイビーステップは、テストファーストプログラミングや継続的インテグレーションなどのプラクティスで表現されている。テストファーストプログラミングでは、一度にひとつのテストを実行する。継続的インテグレーションでは、一度に数時間に相当する変更のインテグレーションやテストを行う。

責任の引き受け（Accepted Responsibility）

責任は割り当てるのではなく、引き受けることしかできない。誰かがあなたに責任を担わせようとしても、責任を持つかどうかを選ぶのはあなたである。

責任の引き受けを反映したプラクティスとしては、サインアップした人がその作業を見積もるというものがある。同様に、ストーリーを実装する担当者が、ストーリーの設計、実装、テストの責任を最終的に担うというものもある。

責任には権限が伴わなければいけない。このバランスがとれていないと、チームのコミュニケーションが破綻してしまう。たとえば、プロセスの専門家が仕事のやり方を命令し、その仕事の内容や結果に責任をとらないのであれば、権限と責任の

バランスがとれているとはいえない。そこには、改善に必要なフィードバックを確認したり、利用したりする知的な立場が存在していない。それから、バランスがとれていないことに耐えるための感情的なコストも必要になってしまう。

結論

プラクティスを深く理解したり、目的に適したプラクティスが見つからずに補助的なプラクティスを作ったりするときには、原則を使えばいい。プラクティスは(「コードを変更する前にテストを書く」のように）明確で客観的に説明されている。ただし、どのように自分の文脈に適用するかは明確ではないかもしれない。そのような場合は、原則がプラクティスの意図を教えてくれる。あらゆるソフトウェア開発の状況や文脈を網羅したプラクティスの一覧は存在しない。あなたも必要に応じて、新しいプラクティスを作ることになるだろう。原則を理解すれば、既存のプラクティスと全体のゴールに調和したプラクティスが作れるようになるはずだ。

第6章

プラクティス

これからXPのプラクティスを紹介する。プラクティスとは、XPチームが日常的に行うものである。だが、プラクティスだけでは味気ない。価値によって目的を与えなければ、機械的な作業になってしまう。たとえば、**ペアプログラミング**が「チェックボックスにチェックを入れる」行為であれば、何の意味もない。上司を満足させるためのものであれば、不満が募るだろう。それが、コミュニケーションであり、フィードバックを獲得するものであり、システムをシンプルにするものであり、エラーを捕捉するものであり、勇気のある行動を鼓舞するものであれば、**ペアプログラミング**には大きな意味がある。

プラクティスは状況に依存する。状況が変われば、条件に合わせてプラクティスを選ぶことになる。ただし、状況が変わっても、価値を変えてはいけない。分野が変わった場合には、新しい原則を採用することもあるだろう。

XPのプラクティスは絶対的なものとして記述している。私としては、あなたに完璧を目指す動機を与え、明確なゴールを提供して、そこに至るまでの実践的な方法を授けたいと思っているからだ。XPのプラクティスは、今いるところからXPを習得するところまでの軌道だ。あなたは理想的な開発の状態に向かって進んでいる。たとえば、1年に1回しかデプロイしないのであれば、**デイリーデプロイ**に意味があるとは思えないだろう。そこから何度もデプロイできるようにすることが改善であり、それにより次のステップに進む自信が得られるのである。

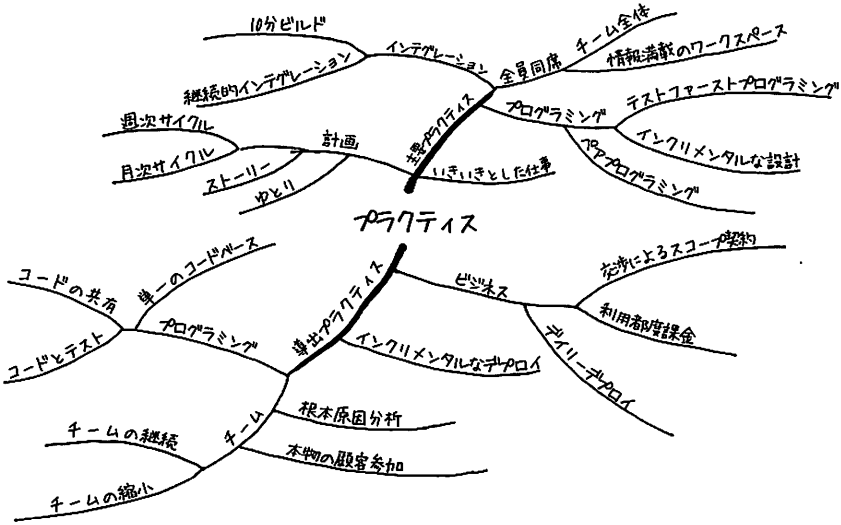
プラクティスを適用するかどうかは選択だ。プラクティスはプログラミングの効果を高めると私は考えている。プラクティスは単独でも機能するが、組み合わせたほうがさらにうまく機能する。これから紹介するのは、そうしたプラクティスのコ

レクションだ。プラクティスは過去に使用されたものばかりである。これらのプラクティスを仮説として使い、XPの実験をしてみてほしい。たとえば、こまめにデプロイすることを試してみて、役に立つかどうかを確認してみよう。

XPのプラクティスは、ソフトウェア開発の進化における頂点を表しているわけではない。あくまでも改善に至るまでの通過地点にすぎない。XPのプラクティスは組み合わせたほうがうまくいく。一度にひとつだけプラクティスを選んでも改善は見られるだろう。だが、組み合わせて使うようになれば、劇的な改善が見られるはずだ。プラクティスの相互作用が、その効果を増幅させるのである。

プラクティスは2つの章に分かれている。第7章「主要プラクティス」と第9章「導出プラクティス」だ。主要プラクティス（*primary practices*）は、あなたが他にやっているものとは無関係に役に立つものである。すぐに改善につながり、どれからでも安全に始められる。導出プラクティス（*corollary practices*）は、先に主要プラクティスを習得しておかなければ難しいだろう。プラクティスを組み合わせれば増幅効果が得られるので、できるだけ早くプラクティスを追加したほうが有利である。

図3はプラクティスをまとめたものだ。



▶図3　プラクティスのまとめ

第7章

主要プラクティス

本章では、XPを適用してソフトウェア開発を改善するときに、安全に始めるためのプラクティスを紹介しよう。どれを最初に使うべきかは、置かれている環境と、何を改善の機会とするかで決まってくる。たとえば、やるべきことがわからないので、計画づくりが必要な人もいるだろう。欠陥が多すぎて何が起きているかを把握できないので、品質関連のプラクティスが必要な人もいるだろう。

全員同席（Sit Together）

チーム全体が入れる十分な広さのオープンスペースで開発すること。近くに小さなプライベート空間を用意するか、別の場所でプライバシーを確保できるように労働時間を制限するなどして、チームメンバーのプライバシーや「自分だけの」スペースといった欲求を満たせるようにすること。

以前、低迷したプロジェクトのコンサルティングを依頼されたことがある。場所はシカゴの郊外だ。プロジェクトが低迷している理由は謎だった。そのチームには、会社のなかで最も優秀な人材が集められていた。私はオフィスを歩きまわり、彼らのコンピュータープログラムの問題を見つけ出そうとした。

数日後、あることに気づいた。歩いている時間がやたらに長いのだ。優秀な人材には、当然のことながら個室が与えられている。個室は頑丈な建物の各フロアの隅に配置されていた。チームは1日に数分程度しか、お互いにやりとりをしていなかった。そこで私は、全員同席できる場所を探してもらった。1か月後に再訪問すると、そのプロジェクトは快調だった。全員同席できる場所として見つかったのは、

マシンルームだった。彼らは、寒くて風の吹くうるさい部屋のなかで、1 日に 4〜5 時間も過ごしていた。それでも彼らは満足だった。成功していたからである。

この経験から 2 つの教訓を得ることができた。ひとつは、クライアントが主張する問題が何であろうと、それは常に人の問題であることだ。技術的な処置だけでは不十分である。もうひとつは、全員同席して、あらゆる感覚を使ってコミュニケーションすることの大切さだ。

全員同席は少しずつ実現していくこともできる。たとえば、部屋に座り心地のいい椅子を置いて会話しやすくする。会議室で半日プログラミングする。オープンワークスペースの実験として、会議室を 1 週間ほど使わせてもらう。こうしたことのすべてが、チームにとって効果的なワークスペースの発見につながる。

チームの準備ができる前に、パーティションの壁を外すのは逆効果である。チームメンバーの安心感が個人の小さなスペースと結び付いている場合は、それを集団で達成する安心感と置き換えない限り、怒りや抵抗を招いてしまう。きっかけさえ与えれば、チームは自分たちのスペースを作り出すだろう。物理的に近くにいれば、コミュニケーションが促進されることをチームはわかっている。コミュニケーションの重要性を学べば、いずれ自分たちのスペースを開放するだろう。

全員同席のプラクティスがあるということは、地理的に分散したチームは「XP を実行」できないのだろうか？ 第 21 章「エクストリームの純度」では、この疑問について詳しく掘り下げている。簡単に答えると、答えは「ノー」だ。分散したチームであっても XP を実行できる。プラクティスは理論であると同時に予測である。「全員同席」は、実際に会って話す時間を増やせば、人間性やプロジェクトの生産性が高まるだろうという予測になる。地理的に分散したプロジェクトがあり、すべてがうまくいっているのであれば、これからも続けていけばいい。何か問題があれば、全員同席の時間を増やす方法を考えてみよう。それは実際に足を運ぶことかもしれない。

チーム全体（Whole Team）

プロジェクトの成功に必要なスキルや視点を持った人たちをチームに集めること。これは昔からある「クロスファンクショナルチーム」の考えだ。このプラクティスは、その名前に目的が反映されている。チームの全体感。つまり、成功に必要なすべての資源の準備を整えることが、このプラクティスの目的である。プロジェクトの健全性のために綿密なやりとりが必要なところでは、機能単位ではなく、チーム単位でやりとりをすべきだ。

そのためには、以下のような「チーム」感が必要だ。

- ✧ 我々は、帰属している。
- ✧ 我々は、一緒の仲間である。
- ✧ 我々は、お互いに仕事、成長、学習を支え合っている。

「チーム全体」の構成要素は動的に変化する。スキルや考え方が重要なときには、それを身に付けた人をチームに迎え入れればいい。必要なくなれば、チームから外れてもらえばいい。たとえば、プロジェクトでデータベースに大量の変更が必要になったときには、データベース管理者をチームに入れる必要があるだろう。その後、データベースの変更が落ち着いたなら、データベース管理者は（少なくともその機能に関しては）チームから必要なくなるだろう。

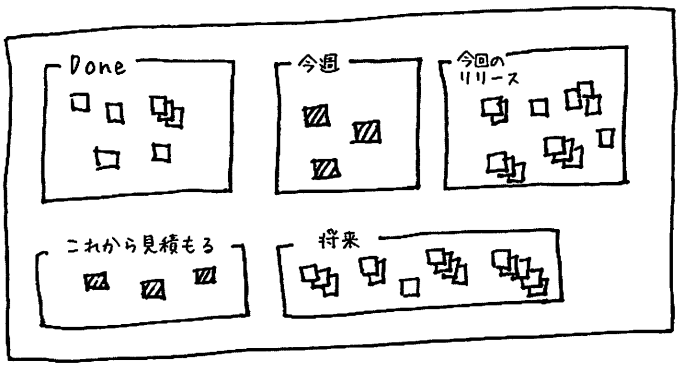
よく論点となるのは、チームの理想的な規模だ。マルコム・グラッドウェルの著書『急に売れ始めるにはワケがある』（ソフトバンククリエイティブ）では、チームの規模に 2 つの断絶があることが示されている。12 と 150 だ。軍隊、宗教団体、企業などの多くの組織では、これらの閾値を超えるとチームを分割している。12 とは、1 日に無理なくやりとりのできる人数だ。人数が 150 人を超えると、チーム全員の顔を覚えていられなくなる。これらの閾値を超えると、信頼関係の維持が難しくなる。コラボレーションには信頼関係が必要だ。大規模プロジェクトの場合は、複数のチームで解決できるように問題を分割する方法を見つけて、XP をスケールさせよう。

チームに小数点以下の人間を入れてしまう組織もある。たとえば、「こちらの顧客の開発作業に 40% の時間を使い、あちらの顧客の開発作業に 60% の時間を使う」といった具合だ。この場合、タスクの切り替え時間がムダになっているので、このプログラマーを完全にチームに参加させれば、すぐに改善が見られる。つまり、チーム単位で顧客のニーズに応えるのだ。そうすれば、プログラマーが思考の断絶から解放される。顧客は必要に応じて、チーム全体が持つ専門知識から恩恵を受けることができる。人間は承認と帰属を求めている。他のプログラマーたちと協力することなく、月曜日と木曜日はこのプログラム、火曜日と水曜日と金曜日はあのプログラムのように対応していたら、「チーム」感が失われるし、非生産的である。

情報満載のワークスペース (Informative Workspace)

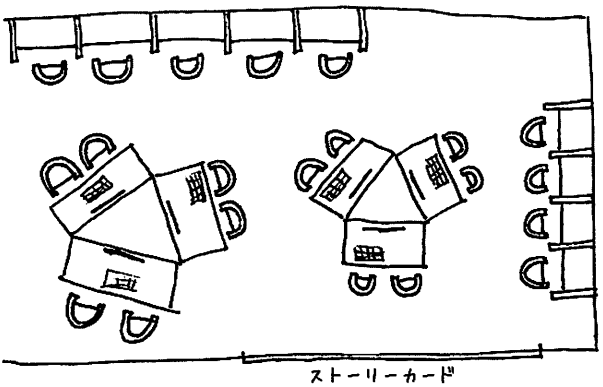
仕事の内容がわかるようなワークスペースを作ること。プロジェクトに関心のある人がチームのスペースを見たときに、15 秒で状況を把握できるようにすべきである。さらに近づいて見たときには、抱えている問題や潜在的な問題に関する詳しい情報を入手できるようにしておこう。

ストーリーカードを壁に貼り付けて、このプラクティスを部分的に実践しているチームも多い。カードを空間的に分類すれば、情報をすばやく伝達できる。作業が終わったことを示す「Done」のところにカードがなければ、チームは計画づくり、見積り、実行などを改善するために何をすべきだろうか？ 私なら顧客が関与できることはないかと考える。達成できないスコープがビジネスに及ぼす影響を最小化するためだ。図4は、空間的にストーリーを分類した理想的な壁を示したものである。



▶図4　壁に貼られたストーリー

ワークスペース（図5）では、人間のその他の欲求も満たす必要がある。たとえば、水やお菓子を用意しておけば、ポジティブな交流が生まれる。清潔に整理整頓しておけば、目の前の問題にとらわれずに自由に発想できる。パブリックスペースでプログラミングするにしても、人間にはプライバシーも必要だ。仕切りのある机を用意したり、労働時間を制限したりすれば、プライバシーを確保できる。



▶図5　チームのワークスペース

情報満載のワークスペースのもうひとつの実現方法は、大きな見える化チャートである。着実に進捗させなければいけない課題があれば、それをチャートにしてみよう。課題を解決できたり、チャートの更新が滞っていたりすれば、すぐに外してしまおう。そのスペースは重要な現在進行形の情報のために使いたい。

いきいきとした仕事（Energized Work）

生産的になれる時間だけ働くこと。無理なく続けられる時間だけ働くこと。意味もなく燃え尽きてしまい、次の2日間の作業が台無しになれば、あなたにとってもチームにとってもよろしくない。

なぜ長時間労働しようとするのだろうか？ よくXPのプラクティスの「科学的」な根拠を求められる。まるで科学がプロジェクトの成否の責任を担っているかのようだ。労働時間についても同じ質問をしてみたい。週に40時間働くよりも80時間働いたほうが、より高いバリューを生み出せるという科学的根拠はどこにあるのだろうか？ ソフトウェア開発は洞察力のゲームだ。洞察力は、準備の整った、休息のとれた、リラックスした精神から生み出される。

私の場合は、その他の方法では制御不能になったときに、制御を取り戻す手段として、長時間労働することが多いように思う。プロジェクト全体の状況は制御できない。プロダクトが売れるかどうかも制御できない。だが、自分が遅くまで残ることならできる。カフェインと糖分があれば、いつまでもタイピングできる。そして、プロジェクトのバリューを低下させていることに気づかないところまで行ってしまう。ソフトウェアプロジェクトのバリューはいとも簡単に失われる。だが、疲労し

た状態では、バリューを奪っていることさえ気づかない。

病気のときは休息と回復に努め、自分とチームをリスペクトしよう。静養こそがいきいきとした仕事に戻る近道だ。病気による生産性の低下からチームを守ることにもつながる。病気のまま仕事に来ても、コミットメントを示すことにはならない。そのような状態で仕事をしてもチームの役に立たないからだ。

労働時間についてはインクリメンタルな改善ができる。働く時間の長さは変えずに、時間の使い方を管理していくのである。たとえば、1日に2時間はコーディングの時間にすると宣言しよう。電話やメール通知を遮断して、ひたすら2時間プログラミングするのである。それだけで十分な改善になるだろう。今後、労働時間を減らすときの足がかりにもなるはずだ。

ペアプログラミング（Pair Programming）

同じマシンを前にした2人で、本番用のすべてのプログラムを書くこと。2人が並んで楽に座れるようにマシンを設置すること。入力しやすいようにキーボードやマウスを適宜移動させること。ペアプログラミングとは、2人でプログラミング（および分析、設計、テスト）とプログラムの改良を同時に行うやりとりのことである。ペアプログラミングでは、以下のようなことをする。

- お互いにタスクに集中する。
- システムの改良について意見を出し合う。
- アイデアを明確にする。
- パートナーがハマったら主導権を握り、相手の失望感を軽減させる。
- お互いにチームのプラクティスの説明責任を果たせるようにする。

ひとりでは考えられないから、ペアになるというわけではない。人間には、交流とプライバシーの両方が必要だ。ひとりでアイデアに取り組む必要があるなら、そうすればいい。そして、チームに戻ってから一緒に確認するのである。ひとりでプロトタイプを作れたとしても、ペアをリスペクトすることはできる。チームから離れて行動することの言い訳にはならない。調査が終わったら、コードではなくアイデアをチームに持ち帰り、パートナーと一緒にすばやく実装し直そう。そのほうが多くの人に理解されるだろうし、プロジェクト全体に恩恵をもたらすことになる。

ペアプログラミングは満足感はあるが、実際にやると疲れるプラクティスだ。多くのプログラマーは、1日に5〜6時間以上もペアを組んでいられない。平日がそのような状態なので、週末は仕事を離れてのんびり過ごすことになる。私がペアを組むときには、近くに水のボトルを置いている。健康にもいいし、休憩をとること

を忘れなくなる。休憩を挟めば、新鮮な気持ちを一日中保つことができる。

ペアのパートナーはこまめに入れ替えよう。タイマーを使って、60分ごと（難しい問題のときは30分ごと）に入れ替わることで、成果が得られたと報告しているチームもある。私は試したことはないが、あまりやりたいとは思わない。開発の区切りのいいところで交代しながら、数時間ごとに新しい相手とプログラミングするほうが私の好みである。

ペアとパーソナルスペース

ペアプログラミングで触れておかなければいけない問題として、密接な接触がある。人や文化によって、快適に感じる身体空間は異なる。たとえば、密接したコミュニケーションを得意とするイタリア人とペアを組むときと、数十センチ離れてパーソナルスペースを保つデンマーク人とペアを組むときとでは、まったく違う。この違いに気づかなければ、居心地が悪くなる可能性もある。うまくやるには、お互いのパーソナルスペースをリスペクトしなければいけない。

個人の衛生や健康も重要な課題だ。咳をするときは口を押さえること。病気のときは仕事を休むこと。パートナーが気になるような強めの香水は避けること。

一緒に効果的に働くのは気持ちがいい。職場によっては、新しい体験になるところもあるだろう。ただし、プログラマーの精神が成熟しておらず、仕事上の承認と私的な好意の区別がついていなければ、異性と働くことに性的な感情を抱く可能性もある。これでは誰の得にもならない。ペアになることでこうした感情が高まるのであれば、責任を持って感情に対応できるようになるまで、その人とペアを組むべきではない。お互いに私的な好意を持っていたとしても、感情のおもむくままに行動していては、チームに悪影響を与えてしまう。親密な関係になりたければ、どちらかがチームを離れたほうがいい。チームのコミュニケーションと性的な関係を混同することなく、私生活の部分で関係を築くべきである。仕事上の感情は、仕事に関係あるものにしよう。

ペアのときは個人差をリスペクトすることが重要だ。図6は、男性が女性に近づきすぎていて、女性が心地よく思っていない。たとえ不快感の原因に気づいていなくても、この状態では2人とも正しい技術的判断ができないだろう。

▶図6　ペアプログラミングとパーソナルスペース

チームにいる誰かとペアを組みたくないときは、信頼できる第三者に相談してみよう。たとえば、尊敬されているチームメンバー、マネージャー、人事部の担当者などだ。あなたが不満を感じていれば、チームも本来の力を発揮できない。それに、あなた以外にも不満を感じている人がいるかもしれない。

ストーリー（Stories）

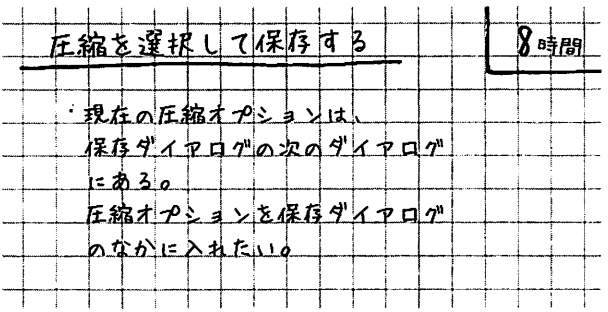
顧客に見える機能の単位を使って計画すること。たとえば、「これまでのレスポンスタイムで5倍のトラフィックを処理する」「よく使う番号に2クリックの短縮を提供する」などだ。ストーリーが書けたら、必要な開発工数をすぐに見積もること。

ソフトウェア開発は「要件」によって間違った方向に進んでいる。「要件」を辞書で調べると、「果たす必要があること、欠くことのできないもの」と書いてある。この言葉を聞くと、絶対的かつ永続的で、変化を受け入れてもらえないように思える。そもそも「要件」という言葉が間違っている。1,000ページの「要件」のなかから、10〜20%程度、あるいはわずか5%でもシステムにデプロイすれば、システム全体のビジネスメリットを認識できる。すると、残り80%は何だったのだろう？ それは「要件」ではない。果たす必要もなければ、欠くことのできないものでもない。

ストーリーは早めに見積もる。要件に関するその他のプラクティスとストーリーとの大きな違いはここだ。見積りがあれば、ビジネスと技術の観点からお互いにやりとりができる。そうすることで、アイデアの持つ可能性が最も高いときに、早めにバリューを作り出すことができる。チームがフィーチャーのコストを把握しておけば、そのバリューに合わせてスコープを分割、統合、拡張できる。

ストーリーには簡潔な記述や絵だけでなく、短い名前もつけること。ストーリー

はインデックスカードに書いて、人通りの多い壁に貼り付けること。図7はストーリーカードのサンプルだ。私のスキャナーにもこのプログラムが実装されていればよかったと思う。ストーリーをコンピューターで処理する試みをいくつも見てきたが、壁に貼ったカードほど扱いやすいものはない。組織に進捗を報告するときには、カードを規定のフォーマットに定期的に変換すればいい。



▶図7　サンプルのストーリーカード

XPスタイルの計画づくりの特徴は、ストーリーを書いたらすぐに見積もることだ。こうすることで、最小の投資で最大のリターンを得る方法を考えられる。たとえば、フェラーリとミニバンのどちらが欲しいかと聞かれたら、私はフェラーリと即答するだろう。絶対そのほうが楽しいからだ。だが、「15万ドルのフェラーリと2万5,000ドルのミニバンのどちらを買いたいですか？」と聞かれたら、よく考えてから判断する。「5人の子どもを乗せる必要がある」や「時速240キロは必要」といった制約が追加されれば、思い描く映像はさらに鮮明になるだろう。どちらの選択にも意味のある状況は存在する。イメージだけでは意思決定できない。車を賢く選択するには、費用と用途の両方の制約を考慮する必要がある。その他の条件がすべて同じなら、魅力的なほうを選ぶだろう。

週次サイクル（Weekly Cycle）

1週間分の仕事の計画をまとめて立てること。そして、週のはじめにミーティングを入れること。ミーティングでは、以下のことを行う。

1. 先週の進捗が期待していた進捗と合致しているかなど、これまでの進捗状況をレビューする。
2. 今週実装する1週間分のストーリーを顧客に選んでもらう。

3. ストーリーをタスクに分解する。チームメンバーはタスクにサインアップして、それぞれのタスクを見積もる。

週のはじめに、ストーリーを完成させたときに実行する自動テストを書くこと。そして、週の残りにストーリーを完成させて、自動テストをパスさせること。単にテストをパスさせるのではなく、仕事に誇りを持ったチームがストーリーを完全に実装しよう。週の終わりまでに、誰もが進捗を称賛するようなデプロイ可能なソフトウェアを作ることが目的である。

1週間は広く受け入れられた時間の尺度だ。本書の第1版では2〜3週間を推奨していたが、誰もが金曜日に向かって集中していることを考えると、1週間にしたほうが都合がいい。チーム（プログラマー、テスター、顧客）の仕事は、書いたテストを5日以内に実行できるようにすることである。水曜日になってもすべてのテストが動かず、完成していないストーリーが残ったままで、デプロイが間に合わないことが明らかになったとしても、最もバリューの高いストーリーを選んで完成させる時間はまだ残されている。

火曜日や水曜日に始めるのが好きな人もいるようだ。最初に知ったときは驚いたが、よくあることなので触れておきたい。あるマネージャーが「月曜日はおもしろくないし、計画もおもしろくありません。どうして一緒にしなきゃいけないんですか？」と言ってきたことがある。私は計画づくりがおもしろくないとは思わないが、プレッシャーをかけて週末にも働かせるようなことをしなければ、サイクルの開始日を変更しても構わない。週末にまで働くのは持続可能なことではない。見積りの甘さが問題であれば、見積りの改善に取り掛かろう。

計画づくりは必然的なムダのひとつである。それ自体はバリューを生み出さない。したがって、少しずつ計画づくりにかける時間の割合を減らすこと。あるチームでは、最初は1週間の計画を1日かけて作っていたが、次第に計画づくりのスキルを高めていき、最終的には1時間で終わるようになっていた。

私はストーリーをタスクに分解するのが好きだ。タスクは各個人が責任を持ち、見積りを行う。タスクを担当すれば、人間にとって必要な所有欲も満たされるだろう。他のやり方でうまくいくのを目にしたこともある。たとえば、タスクに分解する必要のない小さなストーリーを書くやり方がある。ただし、このやり方だと顧客の負担は増えてしまう。タスクをスタックに積んで、サインアップを廃止するやり方もある。新しいタスクを開始する準備ができたプログラマーから、スタックの一番上にあるタスクに着手するのだ。関心の高いタスクや得意なタスクを選択する機会は奪われてしまうが、すべてのプログラマーがさまざまなタスクを担当できるという利点がある（ペアを組むようにすれば、タスクを担当している人が誰であって

も、プログラマーが専門家としてのスキルを発揮する機会だ）。

週次サイクルは、チームや個人が実験をするための、手軽で頻繁で予測可能なプラットフォームでもある。「来週から時計の針が 12 になったらペアを交代しよう」「来週からプログラミングを始める前に毎朝 5 分間ジャグリングしよう」といった具合に、チームや個人が実験の場として利用できる。

四半期サイクル（Quarterly Cycle）

四半期分の計画をまとめて立てること。四半期に一度は、チーム、プロジェクト、進捗、大きなゴールの調整について、ふりかえること。

四半期の計画では、以下のことを行う。

- ボトルネックを特定する（特にチームの外側で制御されているもの）。
- 修正作業に着手する。
- 四半期のテーマを計画する。
- テーマに取り組むための四半期分のストーリーを選択する。
- プロジェクトを組織に適合させる全体像に集中する。

四半期は自然で、広く受け入れられている時間の尺度であり、プロジェクトの期間の区切りに使えるものである。四半期単位で計画すれば、その他の四半期単位のビジネス活動ともうまく同期できる。四半期は外部の供給者や顧客とのやりとりにも適した間隔である。

「テーマ」を「ストーリー」と区別しているのは、チームが全体像と今週のストーリーの調和を考えずに、今やっていることだけに集中する傾向があるからだ。テーマはマーケティングのロードマップを描くような規模の大きな計画にも適している。

四半期は、チームのふりかえりにも適した間隔だ。ふりかえりによって、有害だが気づいていないボトルネックを発見できる。それから、長期間の実験を四半期ごとに提案したり、評価したりすることもできるだろう。

ゆとり（Slack）

どのような計画にも、遅れたときに外せるような重要度の低いタスクを含めること。あとからストーリーを追加したり、約束より多くのストーリーをデリバリーしたりするのは、いつでもできる。不信感を抱かれたり、約束を破ったりしたときは、やるべきことをきちんと果たすことが重要だ。少しでもやるべきことを果たせば、人間関係の再構築につながるはずである。

アイスランドには、未開の土地を巨大なトラックで走り回るという冬のスポーツがある。トラックはすべて四輪駆動だが、快調なときには二輪駆動しか使わない。二輪駆動で立ち往生したら、四輪駆動で抜け出す。四輪駆動で立ち往生したら、もうそこからは抜け出せない。

2つの会話を思い出した。ひとつは、部下が100人いるミドルマネージャーとの会話だ。もうひとつは、そのミドルマネージャーの上司で、部下が300人いる経営幹部との会話だ。私がミドルマネージャーに提案したのは、自信のあるものだけサインアップするようにと、チームに助言することだった。オーバーコミットと乏しい成果が、長年にわたって常態化していたのである。

「それはできません。攻めの（つまり非現実な）スケジュールに合意しないと、私がクビになってしまいます」

ミドルマネージャーはそのように答えた。翌週、私は経営幹部と話をした。

「彼らはスケジュールどおりには終わりませんよ。ですが、それでいいのです。必要なところはすでに完成していますから」

そのときまでに私が目撃していたのは、習慣的なオーバーコミットによる、信じられないほどのムダだった。管理不能な大量の欠陥、士気の著しい低下、敵対的な人間関係。やるべきことを果たせば、それがどんなにささやかなものであっても、ムダの排除につながる。明確で正直なコミュニケーションが、緊張関係を和らげ、信頼を高めるのである。

ゆとりはさまざまな方法で生み出すことができる。たとえば、8週間のうち1週間を「ギーク週間」にしたり、1週間の予算の20%をプログラマーが選んだタスクにあてたりすればいい。あるいは、まずは自分からゆとりを持つ必要があるかもしれない。組織に正直で明確なコミュニケーションの準備が整っていなくても、タスクの所要時間を自分に言い聞かせ、その時間を自ら作り出すのである。

10分ビルド（Ten-Minute Build）

自動的にシステム全体をビルドして、すべてのテストを10分以内に実行させること。ビルドに10分以上かかるようだと使用頻度が減り、フィードバックの機会が失われてしまう。ビルドが短縮できれば、途中でコーヒーを飲むこともない。

物理学には、確固とした自然定数が存在する。海抜ゼロ地点では、物体は重力によって毎秒9.8メートルずつ加速する。重力は裏切らない。だが、ソフトウェアにはそうした確実なものがほとんど存在しない。10分ビルドはソフトウェアエンジニアリングに近い。自動ビルドと自動テストを導入したいくつかのチームは、全体のプロセスが10分を超えないようにしていた。10分以上かかった場合は、10分

以内になるまで誰かが最適化するのである。

10分ビルドは理想だ。その理想にたどり着くまでに、何をすべきだろうか？ プラクティスの説明にある「**自動的**にシステム**全体**をビルドして、**すべて**のテストを10分以内に実行させること」に3つのヒントが隠されている。プロセスが自動化されていなければ、まずはそこから手を付けよう。次に、システムの変更した部分だけをビルドできるようにしよう。最後に、変更によってリスクが高くなった部分のテストだけを実行できるようにしよう。

システムのどの部分にビルドが**必要**で、どの部分をテストする**必要**があるかを臆測で考えると、エラーのリスクが高まる。あなたが間違っていれば、予測できないエラーを見逃す可能性が高く、社会的および経済的な損失を被ってしまう。とはいえ、何もテストしないよりは、一部でもテストできたほうがはるかにマシだ。

自動ビルドは手動ビルドよりも有益である。手動ビルドの場合、普段のストレスが増えると回数が少なくなり、うまくいかなくなってしまう。その結果、エラーとストレスがさらに増える。プラクティスはストレスを減らすものでなければいけない。自動ビルドを使えば「間違えたかな？ ビルドして確認してみよう」といった具合に、大事なところでストレスを解放してくれる。

継続的インテグレーション (Continuous Integration)

数時間以内に変更箇所のインテグレーションとテストをすること。チームプログラミングとは、分割統治の問題ではない。分割、統治、統合（インテグレーション）の問題である。インテグレーションのステップは予測できるものではないが、プログラミングよりも時間のかかることが多い。インテグレーションに時間がかかれば、その分だけコストは上がり、予期しないコストも増えてしまう。

継続的インテグレーションの一般的なスタイルは非同期である。変更をチェックインすると、ビルドシステムがすぐに変更に気づき、ビルドとテストを開始する。問題があれば、メール、テキストメッセージ、あるいは（めちゃカッコいい）赤く光り輝くラバランプに通知が届く。

私は同期モデルのほうが好きだ。数時間ほどペアプログラミングしたら、パートナーと一緒にインテグレーションするのである。ビルドが完了して、すべてのテストスイートが実行され、手戻りがないことを確認してから、先へ進む。

非同期型インテグレーションによって、（特に自動テストのない）デイリービルドは大きく改善されるだろう。だが、同期型インテグレーションにあるふりかえりの時間がそこには存在しない。同期型インテグレーションであれば、コンパイラやテ

ストを待つ時間に、ここまで一緒に何をしたのか、うまくできたかどうかなどを自然に話し合うことができる。また、短時間の明確なフィードバックサイクルを生み出すポジティブなプレッシャーにもなる。たとえば、新しいタスクを開始してから30分後に問題に気づいたとしよう。何をしたのかを思い出し、その問題を修正して、先ほど中断したタスクに戻るまでに、多くの時間をムダにすることになる。

インテグレーションとビルドによって、完全なプロダクトを作り出すこと。CDを焼くことがゴールであれば、CDまで焼くこと。ウェブサイトにデプロイすることがゴールであれば、ウェブサイトにデプロイすること。それがテスト環境であっても構わない。システムの最初のデプロイが大変な作業にならないように、継続的インテグレーションを完全にしておくべきである。

テストファーストプログラミング (Test-First Programming)

コードを変更する前に、失敗する自動テストを書くこと。テストファーストプログラミングは、以下のような多くの問題を一度に解決する。

- **スコープクリープ** —— プログラミングに夢中になって我を忘れてしまうと、「念のため」に余計なコードを追加しがちである。プログラムのあるべき姿を明確に客観的に記述すれば、本来のコーディングに集中できる。どうしてもコードを追加したいなら、手元の作業を終えたあとに別のテストを書くようにしよう。
- **結合度と凝集度** —— テストを書くのが難しければ、テストの問題ではなく、設計に問題があるのだろう。疎結合で凝集度の高いコードは、テストしやすい。
- **信頼** —— 動かないコードの作者を信頼するのは難しい。動くきれいなコードを書いて、自動テストで意図を示せば、チームメイトから信頼を得られるはずだ。
- **リズム** —— コーディングすると何時間もさまよいがちである。テストファーストでプログラミングすれば、次に何をすべきか（別のテストを書けばいいのか、それとも失敗したテストを修正すればいいのか）が明確になる。そして、それは自然で効率的な開発のリズム（テスト、コード、リファクタ、テスト、コード、リファクタ）になっていく。

XPのコミュニティーでは、システムの振る舞いを検証するテストの代替案をあまり模索してこなかった。だが、静的解析やモデル検査のようなツールもテスト

ファーストと同じように使える。たとえば、システムにデッドロックがないことを「テスト」してから、すべてを変更したあとにデッドロックがないことをあらためて検証することができるだろう。とはいえ、このような使い方を想定した静的解析ツールをこれまでに見たことがない。実行速度が遅すぎて、分刻みのプログラミングサイクルに組み込むこともできない。だが、それは単にどこに集中するかの問題であり、本質的な制約ではないはずだ。

もうひとつの改良版テストファーストプログラミングは、継続的テスティングである。これは、デヴィッド・サフとマイケル・アーンストが、論文「*An Experimental Evaluation of Continuous Testing During Development*（開発における継続的テスティングの実験評価）」で最初に発表したものだ。私もエリック・ガンマとの共著『Eclipse プラグイン開発』（ソフトバンククリエイティブ）で詳しく述べている。継続的テスティングでは、プログラムを変更するたびにテストが実行される。ソースコードを変更するたびにインクリメンタルコンパイラが実行されるのと同じだ。テストが失敗すると、コンパイラのエラーと同じ形式で出力される。継続的テスティングでは、エラーを早期に発見することで、修正時間を短縮するのである。ただし、テストはすばやく実行できなければいけない。

テストファーストでコーディングすると、「この 2 つのオブジェクトはうまく協調できるか？」のように、どうしてもプログラムを局所的に見てテストを書いてしまう。経験を重ねていくと、こうしたテストに安心感を抱けるようになるだろう。スコープが限定されているので、高速に実行できることも多い。10 分ビルドの一環として、数千件のテストを実行することもできる。

インクリメンタルな設計（Incremental Design）

システムの設計に毎日手を入れること。システムの設計は、その日のシステムのニーズにうまく合致させること。最適だと思われる設計が理解できなくなってきたら、少しずつだが着実に、自分の理解できる設計に戻していくこと。

学校では「実装前に可能な限りの設計を詰め込みなさい。二度とチャンスはありません」と教えられた。上記の戦略とは正反対だ。学校で教えられた戦略の学術的な裏付けは、1960 年代の国防契約に関するバリー・ベームの研究にある。この研究では、欠陥の修正コストは時間の経過によって指数関数的に増加することが示されている。このデータが現代のソフトウェアにおけるフィーチャーの追加にも当てはまるとすれば、大規模な設計の変更コストは時間の経過によって劇的に増加するはずだ。もしそうであれば、大規模な設計判断は早期に行い、小規模な設計判断は遅らせることが、最も経済性の高い設計戦略になる。

ソフトウェア開発の世界では、この前提が何十年も正しいものとされてきた。したがって、時間経過による変更コストの増加は、ほとんど検証されることがなかった。この前提はもはや正しくない可能性がある。欠陥が増えるとコストが増加するように、変更が増えるとコストは増加するのだろうか？ 変更によってコストが増加するときとしないときがあるとすれば、変更コストが増加しない条件があるのだろうか？ 変更コストが大きく増加しないのであれば、ソフトウェアの最高の開発方法はどのようなものになるだろうか？

XP チームでは、ソフトウェアの変更コストを劇的に増やさない条件を作り出そうとしている。自動テスト、設計の継続的な改善プラクティス、明確な人間関係のプロセスのすべてが、変更コストを低く抑えることに貢献している。

XP チームは、将来の要件に設計を合わせられるという自信を持っている。そのため、すぐに何度も成功したいという人間の欲求を満たすことができるし、最終責任時点まで投資を遅らせる経済的なニーズも満たすことができる。第 1 版を読んで XP を適用したチームには、最終**責任**時点のメッセージがうまく伝わっていなかったようだ。彼らは設計に時間をかけずに、できるだけ早くストーリーを積み上げようとしている。設計に毎日注意を払わなければ、変更コストは急増するだろう。その結果、設計が貧弱で脆弱な変更しにくいシステムになってしまう。

XP チームは、短期的な設計の労力を最小化するのではなく、現時点までのシステムのニーズに合わせて設計すべきである。ここで疑問となるのは、設計をすべきかどうかではなく、いつ設計をするかだ。インクリメンタルな設計を取り入れるときは、経験を踏まえたあとに設計するのが最も効果的である。

小さくて安全なステップを踏むことが**設計の方法**だとしたら、次の疑問はシステムの**どの部分の設計**を改善するかである。私が発見したシンプルで役に立つ解決策は、重複を排除するというものだ。たとえば、同じロジックが 2 か所にあれば、どうすればひとつにできるかを考えながら設計に取り組めばいい。重複のない設計は変更しやすい。ひとつのフィーチャーを追加するために、複数の箇所にあるコードを変更するような状況に陥ることがなくなるはずである。

改善の方向性として、何も経験していないときに設計するのは最悪だと言っているわけではない。インクリメンタルな設計は、使用する直前に設計するのが最も効率的だと言っているのである。小さくて安全なステップで稼働中のシステムを変更する経験を積み重ねていくと、設計にかける労力を遅延させることができるようになる。そうすれば、システムはシンプルになり、進捗は早くなり、テストが書きやすくなる。システムが小さくなるので、チームのコミュニケーションも軽減できる。

チームが毎日設計に手を入れていくと、システムの目的に関係なく、変更の方法

がよく似てくる。このように繰り返し発生する変更パターンを体系化した設計の規律が「リファクタリング」だ。リファクタリングはさまざまな規模で登場する。一度決めたら変更できない設計判断はほとんど存在しない。結果として、システムは小さなところから始めて、膨大なコストをかけることなく、必要に応じて成長していけるのである。

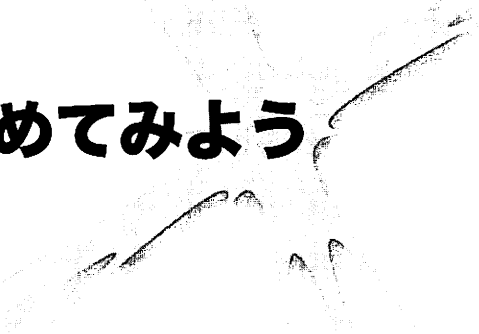
それから……

本章で紹介したプラクティスはXPのすべてではない。これらのプラクティスは、XPの価値であるシンプリシティと勇気を強化して、リスペクト、コミュニケーション、フィードバックの基礎を提供するものだ。チームメンバーはプラクティスによって自信と能力を高め、チーム内外との関係を構築できる。

プラクティスをうまく実践できるようになると、XPの大きな恩恵が受けられる。それから、仕事上の人間関係が改善され、ソフトウェア開発をもっと完璧にやれるようになる。そうした大きな前進につながっていく。

第8章

始めてみよう



あなたはすでにソフトウェアを開発している。もう始まっているのだ。XPは開発プロセスとそれに伴う経験の両方を改善する方法である。XPを実践するには、今いるところから始めて、プラクティスを追加しながら適応していくことになる。追加するプラクティスは、あなたの目的を達成するものであり、価値を表現したものである。プラクティスを追加すると、その相乗効果によって以前は想像もできなかったことが可能になる。そうすると、もっとプラクティスを追加したくなる。XPを完全に適用する頃には、以前よりも元気になり、自信がつき、活発なコミュニティーの一員となり、信じられないペースで仕事ができるようになり、ストレスも減っていくだろう。

XPは、改善の取り組みの新しい方向性や加速を意味するものなのかもしれない。だとしたら、どこからどのように始めればいいのだろうか？

あらゆる人に適切な開始地点というものは存在しない。問題に適した主要プラクティスならば、どれも安全で即効性がある。やるべきことが多すぎて圧倒されてしまう？ ならば、週次サイクルをひとりで始めよう。週のはじめに1週間で達成できそうなことをすべて書き出してみよう。やることが多すぎる場合は、チームのニーズに合わせて優先順位をつけよう。

変化に取り組むときは、一度にひとつずつ着手すると簡単だ。本書を読んでやろうと決めてから、すべてのプラクティスに挑戦し、すべての価値を受け入れて、すべての原則を新しい環境に適用しようとするのは難しい。XPの技術的スキルとその背景にある考え方の学習には時間がかかる。XPはすべてが一緒になったときに最も効果が現れる。だが、まずは開始地点が必要だ。

変化は必ずしもゆっくりとしたものではない。チームが熱心に改善を望んでいれば、すばやく先へ進める。変化が行き渡るのを長期間待ってから、次のプラクティスに取り組むようなことをしなくても構わない。ただし、あまりにも変化が速いと、古いプラクティスや価値に戻ってしまう危険性もある。そうなったときには、仕切り直そう。自分が維持したい価値を思い出そう。プラクティスを再検討して、なぜそれを選んだのかを思い出そう。新しい習慣の定着には時間がかかるものだ。

何を最初に変えるかをどのように決めるのだろうか？ まずは、今何をしているのか、何を達成したいのかを確認しよう。そして、そのために必要な最初のプラクティスを選択しよう。ひとつの選択肢としては、XP スタイルの計画づくりから始める方法がある。まずは、「ビルドを自動化する」「常にテストを先に書く」「ジョーと 2 時間ペアプログラミングする」のように、ソフトウェア開発プロセスを改善するストーリーを書いてみよう。それから、それぞれの時間を見積もって、プロセスの改善に必要な予算を把握しよう。それが終わったら、最初に取り組むストーリーを選択しよう。簡単なもの、バリューの高いもの、難しいものが見つかったら、状況に合わせて適応しよう。

XP の導入を計画している組織にこのプロセスを使ったことがある。あるチームは「勉強会を開催する」「パイロットプロジェクトを実施する」「経営幹部を教育する」などのストーリーを書いてくれた。誰もが不可能だとわかっていたが、スポンサーは即効性のある変化を求めていた。XP スタイルの計画づくりを変化のプロセスに適用すると、コミュニケーションが促進され、うまく優先順位をつけられるようになった。我々が何をしているかが、スポンサーにも見えるようになった。そして、現状に影響を与えることができた。

変化は気づきから始まる。変化が必要とされているという気づきは、感情、直感、事実、部外者からのフィードバックによってもたらされる。感情は重要だが、事実や信頼できる意見を用いて、クロスチェックする必要があるだろう。

メトリクスが気づきにつながることもある。メトリクスのトレンドを見れば、事態が悪化する前に変化の必要性を把握できる。以前、あるチームをコーチしたときに、開発後の欠陥を確認してもらったことがある。その結果、すべての欠陥が、単独でプログラミングしたときに生み出されたものであることが判明した。経験を正確にふりかえることができなければ、情報にもとづいてペアプログラミングの時間を決定することもできなかっただろう。

変化の必要性に気づけば、変化に着手できる。開発プラクティスを改善するには、主要プラクティスから始めるといいだろう。それぞれのプラクティスは、一連の振る舞いを表している。プラクティスごとに現在の自分の立ち位置を見極めよう。自

分が優先する変化に適した目的を持つプラクティスを選択しよう。プラクティスの最終段階に一歩近づこう。開発の人間性と有効性が改善するかを確認しよう。

たとえば、親密な技術的協力が必要になることもあるだろう。大きなインテグレーションが近づくと、設計やコードのレビューをどれだけやっても不安になる。どうすればもっと協力できるだろうか？

ペアプログラミングは技術的協力を扱うプラクティスだ。協力の度合いが極限まで高まれば、寿命の長いすべてのコードを 2 人で会話しながらプログラミングすることになる。とはいえ、コードの責任は個人に割り当てられているので、自分たちの時間を「諦めて」まで時間を割くつもりのないチームもあるだろう。それでも、不安に思っているコードを選び、1〜2 時間ほどペアになってもらえるように誰かに協力を求め、システムの他の部分とのインターフェイスに取り組めば、協力の度合いをさらに高めることができる。私ならシステムの該当部分の責任者をパートナーに選ぶだろう（相手にその意思があればの話だが）。

ペアを組めば、密接な技術的協力が、チームの目標達成の役に立つのか、阻害要因になるのかを評価できる。これでようやく、これから協力の度合いを強めるべきなのか、その方法は**ペアプログラミング**なのか、レビューなのか、それ以外の方法論なのかを確かな情報にもとづいて判断できる。

変化を起こしたあとに、改善された新しい働き方よりも、なじみのある古い働き方に戻りたくなることがある。自分が起こした変化であっても、関係者があまりにも抵抗すれば、新しい基準を維持せずに元に戻してしまうかもしれない。事態を悪化させているのは、新しいプラクティスに対する不安だ。プログラムと一緒に自動テストを書けば、プログラミングは速くなるのだろうか？ それとも遅くなるのだろうか？ 不安だからといって、不適切に元に戻すことは避けたい。

変化は常に自分のいるところから始まる。あなたが変えられるのは、あなただけだ。組織が機能していても、機能していなくても、あなたは自分に XP を適用することを始められる。チームの誰もが、自分の振る舞いを変えることを始められる。プログラマーはテストを先に書くことを始められる。テスターはテストを自動化することを始められる。顧客はストーリーを書いて優先順位を明確にすることを始められる。経営幹部は透明性を求めることを始められる。

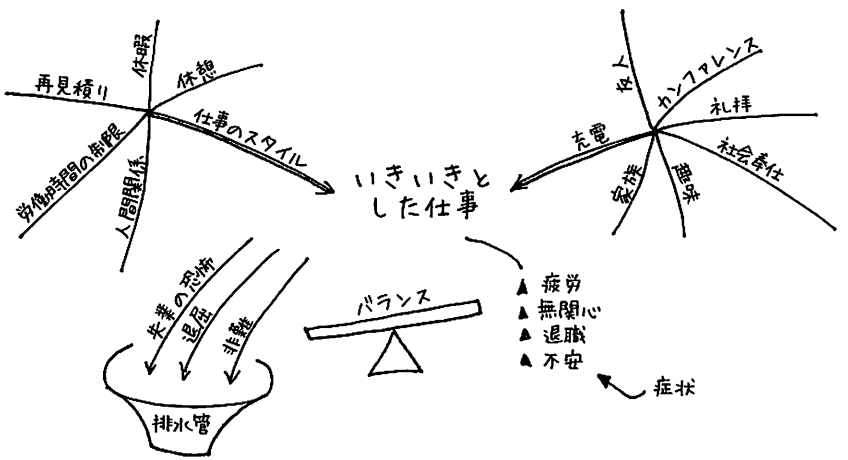
プラクティスをチームに強要すると、信頼関係を破壊して反感を買ってしまう。経営幹部にできるのは、チームに責任感や説明責任を促すことだ。チームが XP を使おうが、改良したウォーターフォールを使おうが、カオスになろうが、すべてはチームの責任である。XP を使えば、チームは欠陥、見積り、生産性に劇的な改善をもたらすことができる。改善に必要となるのは、密接な協力とチーム全体の取り組

みだ。説明責任とチームワークの期待値を高めて、変化に伴う避けられない不安をチームが乗り越えられるように支援しよう。

プラクティスのマッピング

プラクティスがあなたとチームにとって何を意味するかを見つけよう。これからそのためのエクササイズを行う。

図8は、プラクティス「いきいきとした仕事」のマップだ。中央にあるのがプラクティス。その下にあるのがプラクティスの目的だ。私は「仕事と生活のバランスの維持」が目的だと考えている。プラクティスに接続しているのは、影響を与えている要因だ。ここでは、プラクティスがうまくいかないときの症状を表している。プラクティスについて考えるときは、思い浮かぶ懸念点をすべてマッピングしよう。文字でも絵でも好きなやり方で構わない。



▶図8　「いきいきとした仕事」のマップ

このエクササイズに「正しい」答えはない。例として使ったのは、今の私の考えだ。チームメンバーやチームによって、プラクティスの解釈は異なるだろう。プラクティスに何を接続するかを議論することが、このエクササイズの有益な副作用だ。

これから起こしたい変化があるなら、実際に変化を起こしてみよう。事態をよくするアイデアがあっても、変化のエネルギーにつながらなければ意味がない。私も不平不満で変化のエネルギーを浪費する「くだを巻く」ような会議に数多く参加してきた。納得できる改善のアイデアがあれば、とにかくやってみればいい。チーム

でやれるともっといいだろう。チームでやれなくても、まずはひとりでやってみよう。何かを学習できたら、信頼できる誰かと共有してみよう。

半日かけてすべてのプラクティスを検討し、どのプラクティスを変化の構成要素とするかを一緒に決めよう。その結果をフリップチャートにマッピングして、チームの部屋に貼り付けておこう。ひとつの変化を実施したら、次に着手できる変化を確認し、新しい変化の取り組みを始めよう。

結論

あなたの肩書が何であれ、ソフトウェア開発との関係がどのようなものであれ、「ソフトウェア開発には現状よりずっと大きな可能性がある」ことを私はメッセージとして伝えたい。欠陥は見たことがないほど貴重な存在になるだろう。進捗の遅れが原因による大きなスコープ調整は、スケジュールの前半だけにしか登場しなくなるだろう。プロジェクト予算を少しだけ使ったあとで、すぐに最初のデプロイをするようになるだろう。チームは破滅的結果を招かずに、規模の拡大や縮小が可能になるだろう。XP はこれらを実現するための方法だ。開発プロセスのなかで人間性に取り組めば、飛躍的に効果が向上するはずである。

第9章

導出プラクティス

主要プラクティスを事前に実践していなければ、本章のプラクティスを導入するのは難しいだろう。あるいは危険だと思われる。たとえば、（ペアプログラミング、継続的インテグレーション、テストファーストプログラミングなどで）欠陥率をゼロに近づけないまま、デイリーデプロイを開始しようとすれば、大惨事を招くことになる。次に何を改善すべきかは、自分の嗅覚を信じよう。以下のプラクティスのいずれかが適していると思えば、試しにやってみよう。それでうまくいくかもしれないし、開発プロセスを改善する前にやるべきことが見つかるかもしれない。

本物の顧客参加（Real Customer Involvement）

あなたのシステムによって生活やビジネスに影響を受ける人をチームの一員にすること。明確なビジョンを持った顧客であれば、四半期や週単位の計画づくりに参加できる。そうした顧客は自由に使える予算を持っていることもある。予算は開発で利用可能なキャパシティーの一部だ。競合他社より半年も早く問題に気づくような顧客であれば、その顧客が必要とするシステムを構築することによって、あなたの競合他社に対する優位性にもつながる可能性がある。プロダクトが顧客にバリューをもたらすのであれば、きっとチームへの参加を申し出てくれるだろう。顧客参加のポイントは、ニーズを持つ人とそれを満たす人が直接やりとりをして、ムダな労力を減らすことである。

私が考えるチーム全体には、顧客も含まれている。だが、本物の顧客を参加させているチームをあまり見たことがない。本物の顧客がいれば、結果が違ってくる。

顧客はあなたが喜ばせる相手だ。本物の顧客がいなければ、あるいは本物の顧客の「プロキシ」しかいなければ、使われないフィーチャーを開発したり、本物の受け入れ条件を反映していないテストを仕様化したり、プロジェクトの多様な視点を持った人たちとの人間関係を構築する機会が失われたりと、さまざまなムダを招いてしまう。

顧客参加について私が耳にしたことのある反論は、参加した顧客は本当に必要なシステムが手に入るかもしれないが、参加していない顧客に適したシステムにはならない、というものだ。誰の問題も解決していないシステムを誰かに合わせるよりも、誰かの問題をうまく解決したシステムを一般化するほうが簡単である。システムが一般的にも便利に使えるようにするのは、チームのマーケティング担当者の仕事だ。通常、顧客のニーズと開発のキャパシティーの距離が近ければ近いほど、開発の重要性は高まっていく。

顧客参加に対するもうひとつの反論は「ソーセージ工場」である。これは「ソフトウェア開発がいかに混乱しているかを顧客に知られたら、信頼してもらえなくなる」というものだ。それでは、今は本当に信頼されているのだろうか？ ソフトウェアはそれを構築した組織を反映している。顧客はすでにソフトウェアを使っているので、開発の様子もかなり知っているはずだ。まだ知らなかったとしても、すぐに知られてしまうだろう。信頼できる行動をとり、何も隠さなければ、生産性は高まる（隠すことや取り繕うことに時間を費やす必要がないからだ）。見積りが正確であり、欠陥率が低下しているのであれば、顧客に開発プロセスに参加してもらうことで、さらに信頼関係が高まり、継続的改善が促進されるのである。

インクリメンタルなデプロイ (Incremental Deployment)

レガシーシステムをリプレースするときは、プロジェクトの初期段階から少しずつ引き継ぎをすること。最近、ある食料品チェーン店のシステムについて友人と話をした。手の込んだ内製ソフトウェアからパッケージへ移行するそうだ。今の機能をパッケージで再実装し、日曜日の夜にカットオーバーする計画だという。私はすぐに「それはうまくいかないだろ」と答えた。

大きなデプロイがうまくいかないわけではない。新機能を追加せずに「D デイ」に向けて何か月も準備をするのだろう。みんなで長時間働く。週末も働く。その努力が実を結び、新しいシステムがうまく稼働したとしても、全員が疲弊してしまい、数週間から数か月は生産的な開発に戻れない。努力が実を結ばなければ、新しいシステムの稼働は延期せざるを得なくなり、コストは一段と上がってしまう。大きな

デプロイはリスクが高く、人間的にも経済的にもコストが高い。

代替案は何だろう？ 今すぐに対応できるちょっとした機能や限定的なデータセットを見つけて、それをデプロイすればいいのである。もちろん新旧両方のプログラムを同時に実行する必要はあるだろう。ファイルを分割／マージしたり、ユーザーを教育したりする必要もある。このような技術的および社会的な足場を用意することは、移行のための保険料だと考えよう。

以前はインクリメンタルなデプロイのことを頭では信じていたが、心の底からは信じていなかった。9,000 件の契約情報を新しいシステムに移行する仕事をお手伝いしたときに、私は考えを変えることにした。移行を開始してから数か月後、契約情報の 80% をうまく処理できるようになった。だが、データの品質の問題が原因で、残りの 20% を問い合わせにうまく一致させることができなかった。試行錯誤に 6 か月かけて、古いシステムのエラーを再現するようなことまでして（浮動小数点数を丸めるために信じられない処理をしていた！）、なんとかすべての契約情報を処理できるようになった。だが、その矢先にマネージャーが優先順位を変更し、別の契約情報を変換するように依頼してきたのである。その年の終わりになっても、すべての契約情報を新しいシステムにデプロイすることはできなかった。結果として私は、新築の家に相当する額のボーナスをもらい損ねることになった。今ではインクリメンタルなデプロイを**心の底から**信じている。

チームの継続（Team Continuity）

優秀なチームは継続させること。大きな組織は、ヒトをモノに抽象化する傾向がある。互換性のあるプログラミングユニットだと考えているのだ。ソフトウェアのバリューは、みんなが知っていることや行っていることだけでなく、人間関係やみんなで一緒に成し遂げることによっても生み出される。要員計画の問題を単純化するためだけに、人間関係や信頼の大切さを無視するのは経済的ではない。

小さな組織には、こうした問題は存在しない。そこにはひとつのチームしかないからだ。チームが一致団結し、信頼関係を築けたならば、チームがバラバラになるのは大惨事が発生したときくらいである。一方、大きな組織はチームの重要性を無視して、「プログラミング資源」に分子や流体のメタファーを使っている。プロジェクトが完了すると資源が「プールに戻る」というわけだ。そのような要員計画の目的は、すべてのプログラマーを可能な限り有効利用することである。こうした戦略は、気心の知れた信頼できる人と一緒に働くことの重要性を無視して、プログラマーにキーボードを入力させ続けるという幻想の効率化を求めている。局所的には効率化できても、組織全体の効率性は低下するだろう。

一致団結したチームを継続させるといっても、チームを固定化するわけではない。確立されたXPチームの新規メンバーはすぐに貢献を始める。その早さに驚かされるほどだ。最初の週からタスクにサインアップするし、1か月もすると単独で貢献を始める。チームを継続しながら適度なローテーションを行えば、組織は「安定したチーム」と「知識や経験の継続的な広がり」の両方の利点が得られるだろう。

チームの縮小（Shrinking Teams）

チームの能力が高まったら、仕事量を維持しながら少しずつチームの規模を縮小すること。チームを離れた人は、また別のチームを作ることができる。チームメンバーが少なすぎる場合は、他の小さなチームと統合しよう。これはトヨタ生産方式で使われているプラクティスだ。私は実際に使ったことはないが、非常に納得できるものなので掲載しておきたい。より多くの仕事量をこなすために、チームの規模を拡大するような戦略も見てきたが、それではうまくいかない。他の方法を考えるべきだ。

私はこのプラクティスの経験がないので、比喩を使って説明しよう。たとえば、生産セルに5人いたとする。全員に等しく仕事を割り当てるよりも、仕事に専念する人をできるだけ多くしたい。その結果、5人目は30%の時間だけ仕事をするようになるかもしれない。だが、それでいいのである。チームメンバーは仕事をしながら、プロセスの改善方法についても考えている。5人目が不要になるまで、ムダを排除するアイデアを試しているのだ。全員を忙しい状態にしようとすると、チームに利用可能な資源が残されている事実が覆い隠されてしまう。

ソフトウェア開発でも同じことを試してみること。顧客が毎週必要とするストーリーを数えること。チームメンバーの誰かの手が空くまで、開発を改善していくこと。そうすれば、規模を縮小しながらチームを継続できるはずだ。

根本原因分析（Root-Cause Analysis）

開発後に欠陥が見つかるたびに、欠陥とその原因を取り除くこと。欠陥の再発防止だけではなく、同じ種類の過ちをチームが二度と犯さないようにすることが目的だ。

以下は、欠陥に対処するためのXPのプロセスである。

1. 欠陥を実証するシステムレベルの自動テストを書く。そこには期待する挙動も含めておく。これは、顧客、顧客サポート、開発者が行える。
2. 欠陥を再現する最小限のスコープでユニットテストを書く。

3. ユニットテストが動くようにシステムを修正する。これにより、システムテストもパスするはずだ。パスしなければ、2 へ戻る。
4. 欠陥を修正できたら、なぜ欠陥が生み出されたのか、なぜ発見できなかったのかを見極める。今後は同様の欠陥が再発しないように、必要な変更を加える。

大野耐一氏が、最後のステップのための簡単なエクササイズを用意している。それが「5 回のなぜ」だ。なぜ問題が起きたのかを 5 回質問するのである。たとえば、以下のような感じだ。

1. なぜ欠陥を見逃したのか？ 一晩で不均衡が生じる可能性があることを知らなかったからだ。
2. なぜ知らなかったのか？ クロスビーさんだけは知っていたが、彼女はチームの一員ではないからだ。
3. なぜ彼女はチームの一員ではないのか？ 彼女は古いシステムのサポートをしており、彼女以外にその方法を知らないからだ。
4. なぜ彼女以外に方法を知らないのか？ 他の人に教えることは経営上の優先事項ではないからだ。
5. なぜ経営上の優先事項ではないのか？ 経営者たちは 2 万ドルの投資で 50 万ドルを節約できることを知らないからだ。

「5 回のなぜ」が終われば、欠陥の中心に人の問題があることがわかる（たいていの場合、人の問題だ）。その中心になっている問題と、途中で出てきた問題に対応すれば、二度と同じ過ちを繰り返さなくて済むという安心感が得られる。

導出プラクティスには、単に追加でテストを書くのではなく、きちんとした回帰テストを書くことも含まれている。ほとんどのチームは欠陥が多すぎて、欠陥を解消するための大きな投資ができていないからだ。欠陥率が 1 週間に 1 つ、あるいは 1 か月に 1 つまで低下すれば、そのような投資も割に合う。それに、他の方法で改善するプラクティスも使えるようになる。そうすれば、チームの弱点を深く考察する準備ができるはずだ。

コードの共有（Shared Code）

チームの誰もが、システムのあらゆる部分をいつでも改善できる。システムに問題があり、その修正が現在作業中のスコープの範囲内なのであれば、遠慮せずに修正すべきである。

それぞれのコードに誰かが責任を持たなければ、誰もが無責任に行動するだろう。その場しのぎの修正をして、散らかったコードを次の人に残してしまうだろう。そのような反論を耳にしたこともある。コードの共有を導出プラクティスに載せたのは、こうしたリスクが発生するからだ。チームに連帯責任の意識がなければ、誰も責任をとろうとせず、品質もガタ落ちになる。チームに与える影響を考えずに、変更を加えてしまうだろう。

「自分本位」の他にもチームワークのモデルは存在する。チームメンバーは、ユーザーに届ける成果物の品質だけでなく、それを作る自分たちの仕事のプライドにも責任があると考えることができる。**ペアプログラミング**を実施すれば、チームメイトがお互いに品質に対する貢献を示すことができる。そして、何が品質につながるかの期待を一致させることができる。

継続的インテグレーションは、共同所有の重要な必要条件である。改善する機会が多ければ、わずか2時間のプログラミングセッションであっても、システムの多くの部分に触れることができる。ただし、2組のペアで広範囲に多くを変更すると、互換性のない解決コストの高い変更が増えてしまう。変更が多い場合は、インテグレーションの間隔を短くして、インテグレーションコストを下げるようにしよう。

コードとテスト（Code and Tests）

コードとテストだけを永続的な作成物として保守すること。その他のドキュメントについては、コードとテストから生成すること。プロジェクトの重要な履歴の維持については、社会的な仕組みに任せること。

顧客は、システムの今日の挙動と、チームが開発する明日のシステムの挙動に対してお金を支払っている。この2つのバリューの源泉に貢献する作成物は、それ自体にバリューがある。そして、それ以外はムダだ。

コードとテストは少しずつ取り組みやすいプラクティスである。チームのスキルが高まれば、複雑な5段階のドキュメント駆動のプロセス[†1]も少しずつ緩和されていくだろう。チームが**インクリメンタルな設計**をするようになれば、事前に行うべき設計判断は少なくなっていく。ビジネスの優先順位を決める**四半期サイクル**が明確になれば、その分だけ必要な要件仕様書は薄くなっていく。

過去数十年間、ソフトウェア開発のトレンドは正反対に進んできた。儀式はバリューの流れを滞らせる。ソフトウェア開発における重要な意思決定は「これから何をするのか？」「これから何をしないのか？」「どのようにやるのか？」だ。これ

[†1] 訳注：CMM（能力成熟度モデル）のこと。

らの意思決定をまとめて、お互いに補完できるようにすれば、バリューの流れがスムーズになる。時代遅れになった作成物を排除することで、こうした改善が可能になるのである。

単一のコードベース（Single Code Base）

コードの流れはひとつだけである。一時的なブランチで開発することもできるが、数時間以上も維持してはいけない。

複数のコードの流れは、ソフトウェア開発におけるムダの大きな原因である。コードの流れが複数あると、現在デプロイされているソフトウェアの欠陥を修正したときに、その修正を他のバージョンや開発ブランチにも反映しなければいけない。その修正が作業中の何かを壊していれば、修正をまた修正しなければいけない。それがずっと続くのである。

複数のバージョンを維持しなければいけない理由もあるだろう。だが、それは単なるその場しのぎの理由であり、全体的な影響を視野に入れていない局所的な最適化の可能性もある。複数のコードベースがあるなら、少しずつ減らしていく計画を立てよう。単一のコードベースから複数のプロダクトを作れるように、ビルドシステムを改良することもできるだろう。あるいは、バージョンの違いを設定ファイルに移動することもできるはずだ。いずれにしても複数のバージョンのコードが必要なくなるまで、プロセスを改善していこう。

あるクライアントでは、7社の顧客に対して7種類のコードベースが存在し、許容できないコストになっていた。開発は以前よりも時間がかかるようになっていた。プログラマーは以前よりも多くの欠陥を作り出すようになっていた。プログラミングは当初ほど楽しいものではなくなっていた。複数のコードベースのコストと、スケールが不可能であることを私が指摘すると、クライアントはコードを再統合する余裕がないと答えた。私はクライアントを説得できなかった。7種類あるバージョンを6種類に減らしてもらうことさえできなかった。次の顧客を既存のバージョンのバリエーションとして追加してもらうこともできなかった。

ソースコードのバージョンを増やしてはいけない。コードベースを増やすのではなく、単一のコードベースを妨げている根本的な設計の問題を解決すること。どうしても複数のバージョンを維持しなければいけない理由がある場合は、その理由は絶対的なものではなく、異議を唱えて挑戦していく前提だと考えること。根深い前提を解きほぐすには時間がかかるかもしれないが、それを解きほぐすことによって、改善の次のラウンドの扉が開かれるのである。

デイリーデプロイ（Daily Deployment）

新しいソフトウェアを毎晩プロダクションに反映すること。プログラマーの手元にあるものとプロダクションにあるものが違うのはリスクだ。プログラマーの意思決定につながる正確なフィードバックが得られない危険性がある。

デイリーデプロイが導出プラクティスなのは、必要条件が多いからだ。年間の欠陥率は数件程度でなければいけない。ビルド環境はうまく自動化されていなければいけない。デプロイツールは、インクリメンタルな展開と、失敗時のロールバックの機能も含めて、自動化されていなければいけない。そして、これが最も重要なことだが、チームや顧客との信頼関係が築かれていなければいけない。

ソフトウェアをこまめにデプロイする方向にトレンドに向かっていることは明らかだ。私のインスタントメッセージのプログラムは、数日ごとにアップデートを取り込んでいる。大規模なウェブサイトは、毎日気づかないところで更新されている。デイリーデプロイはこうしたトレンドから推測したものだ。

デイリーデプロイは、ある方向性を指し示すプラクティスの好例だ。1年に1回しかデプロイできないのであれば、デイリーデプロイは妄想のように思えるだろう。だが、1年に1回しかデプロイしていないと思っているチームでも、実際には1回のリリースと11回のパッチで、合計12回デプロイしていることもある。機能の小さなインクリメントを展開できる能力がチームにあっても、その能力を機会と考えずに、仕方なくそうしている恥ずかしいことだと考えていることもある。12回のリリースのほうが、11回のパッチよりもだいぶ響きがいい。

利用できるようになるまで、数週間や数か月かかるプロジェクトの場合は、どのようにしてデイリーデプロイを実施するのだろうか？ 大規模プロジェクトには、データベースの再構成、新しいフィーチャーの実装、ユーザーインターフェイスの変更など、さまざまなタスクがある。システムのユーザーエクスペリエンスを変更しない限り、それ以外のあらゆるものをデプロイして構わない。そして、最後の日にユーザーインターフェイスという「要石」を設置するのである。

こまめにデプロイするには多くの障壁がある。欠陥が多すぎることや、低コストのデプロイ方法の必要性などは、技術的な障壁である。デプロイプロセスにストレスがかかりすぎて何度も実施したくない、という心理的あるいは社会的な障壁もある。こまめなリリースに対して料金を請求する方法がない、というビジネス上の障壁もある。いずれにしても、障害を取り除き、こまめなデプロイを可能にすれば、当然の結果として開発が改善されるはずだ。

交渉によるスコープ契約 (Negotiated Scope Contract)

ソフトウェア開発の契約では、期間、費用、品質を固定すること。システムの明確なスコープについては、継続的に交渉を求めること。長期的なひとつの契約ではなく、短期的な契約をいくつも結ぶようにして、リスクを減らすこと。

あなたはスコープを交渉する方向に進むことができる。長くて膨大な契約書は、2分の1から3分の1に分割できる。付随的な部分については、両関係者が合意した場合だけ行使すればいい。「変更要求」のコストが高い契約は、事前に決めるスコープが少なく、変更コストが安い契約に書き直せる。

交渉によるスコープ契約は、あくまでもソフトウェア開発に関するアドバイスだ。供給側と顧客の利益を一致させる仕組みであり、コミュニケーションやフィードバックを促進して、今日正しいと思えることをみんなが勇気を持って取り組めるようにするものである。契約にあるからという理由だけで、効果のないものに取り組むことはない。ビジネスや法律のことを考えると、今はまだバカげた話に聞こえるかもしれない。それでも、スコープを交渉する方向に進むことで、改善につながる情報源が得られるのである。

利用都度課金（Pay-Per-Use）

利用都度課金システムがあれば、システムが利用されるたびにお金を請求することができる。お金は究極のフィードバックだ。お金には実体があり、これから自分で使うこともできる。お金の流れをソフトウェア開発に直接接続すれば、改善を推進するための正確でタイムリーな情報が得られるはずだ。

多くのソフトウェアは、すでに利用都度課金になっている。電話交換システム、電子証券取引システム、航空座席予約システムなどは、すべて取引ごとに料金を徴収している。ビジネス的には、利用都度課金には長所も短所もある。だが、それが生み出す情報は、ソフトウェア開発の改善に役立つ。

私がこれまでに見たことのある究極的な利用都度課金は、とあるメッセンジャーで使われていたものだ。そのプロダクトでは、ユーザーはメッセージを送信するたびにお金が請求されていた。開発中は、より多くのメッセージを送りたくなるようなストーリーが意図的に選択されていた。たとえば、新しい携帯電話をサポートするときは、コストと収益の両方の見積りを必ず行うようになっていた。そのチームは、システムの利用状況を分析して、収益の見積りの正確性を高めるフィードバックを手に入れていたのである。こうした情報を利用して、コストと利益の両方を最

適化していたのだ。

現在の典型的な取り決めでは、ソフトウェアのリリースごとに顧客がお金を支払わなければいけない。こうしたリリース都度課金は、供給側と顧客の両方の利益に反している。供給側は、顧客にお金を支払わせるだけの最低限の機能しか含まれないリリースを自分勝手に量産するだろう。顧客としては、リリースの回数が少なく、1回のリリースに多くのフィーチャーが含まれているほうが望ましい（アップグレードが面倒だからだ）。この2つの利益の対立関係が、コミュニケーションとフィードバックを低下させているのである。

利用都度課金にできなくても、サブスクリプションモデルに移行することはできるかもしれない。これは、毎月あるいは四半期ごとにソフトウェアを購入するというものだ。サブスクリプションモデルでは、チームは自分たちの行動の状況を把握する情報源として、少なくとも定着率（サブスクライブを継続する顧客数）を見ることができる。ビジネスモデルの変更をさらに小さくしたいなら、事前に得られる収益を減らして、サポート料の比重を増やすような契約にするといいだろう。

利用都度課金に対する反論は、顧客は予測可能なコストを求めているというものである。だが、価格優位性が十分に高ければ、おそらく顧客は気にしないだろう。ライセンス収益のフィードバックだけを頼りにしているチームよりも、利用都度課金の情報を使っているチームのほうが効果的な仕事ができるはずだ。

結論

ソフトウェア開発を成功させるために必要なのは、主要プラクティスと導出プラクティスだけではない。とはいえ、これらは私が観察によって、ソフトウェア開発チームが高みに至るために重要だと信じられるようになったものたちだ。これらのプラクティスが網羅できない問題を抱えているのであれば、価値と原則の部分を読み返して、あなたのチームにふさわしい解決策を考えよう。

第10章

XPチーム全体

効果的なソフトウェア開発には、多くの人の視点が注ぎ込まれなければいけない。XPのプラクティスのチーム全体とは、さまざまな人が相互に関連しながら一緒に仕事をして、プロジェクトをさらに効果的にするものである。各自が成功するためにも、グループとして一緒に働かなければいけない。XPチームにいる全員が、仕事の領域で運命を共にしているのである。XPは、プロジェクトにおけるプログラマーの効果的な振る舞いを規定するところから始まった。ここでは、その冒頭の部分にあたるXPチームのメンバーのための処方箋を紹介しよう。

流れの原則では、頻度の低い大規模なデプロイよりも、スムーズで安定したソフトウェアの流れによって、多くのバリューが作り出されるとしている。さまざまな種類の仕事をうまく体系化してソフトウェア開発につなげるには、このような流れが特に重要になる。だが、適用が難しいこともある。以前、ソフトウェアの開発方法を計画する1日がかりの会議に参加したことがある。最初から会議に参加していたのは、プログラマーと経営幹部だった。そこにさまざまな専門家の代表者が参加して、それぞれの観点からの必要な開発スタイルについて意見を述べるのだ。それを1日かけて行う予定になっていた。

最初にプログラマーが、リスクマネジメント、早期の収益、フィードバック、フェーズよりも活動を好む理由などの観点から、XPについて説明した。参加者たちはうなずいていた。おかしな点は何もなかった。

次にアーキテクトの番になった。そして、プログラミングのところはXPで問題ないが、プロジェクトの初期フェーズで自分たちがアーキテクチャを設計するほうが、すべてがスムーズに進むと説明した。すると、流れに賛成する意見から集中

砲火を浴びることになった。流れに従うならば、必要最低限なアーキテクチャから開始して、継続的にアーキテクチャを洗練させるべきだと説明された。アーキテクトたちは、その方法でやることを不本意ながらも承諾した。それでも、アーキテクチャのフェーズで設計したほうがうまくいくと考えていた。

次にインタラクションデザイナーの番になった。そして、プログラミングのところはXPで問題ないが、プロジェクトの初期フェーズで自分たちがインタラクションを設計するほうが、すべてがスムーズに進むと説明した。すると、流れに賛成する意見を持ったプログラマー、経営幹部、アーキテクトたちから反論されることになった。インタラクションデザイナーたちは、必要最低限なインタラクションから開始して、継続的にインタラクションを洗練させていくこともできるが、プロジェクトの最初に作業したほうがうまくいくと考えていた。

インフラ設計者がプロジェクトの最初に意思決定をしたいと提案する頃には、その会議はお笑いの場になっていた。インクリメンタルに仕事を進めることを不本意ながらも承諾してもらうまでに、それほど時間はかからなかった。

この話にハッピーエンドはない。人は自分の意思に背いたまま納得することはできないのである。どのグループも自分たちをもっと大きなグループの一部であるとは考えていなかった。このようなストレスを受けた結果、自分たちの仕事をすべて事前にやろうとする状態に戻ってしまった。

異なる視点を持つ人たちがロープに結ばれて氷河の上を歩いているときに、誰が先頭になるかを議論したいだけのように思えた。本当に重要なのは、誰が先頭になるかではない。**チーム全体**が見失っていたのは、全員がロープに結ばれているという感覚だ。あるグループが先頭になって他のグループを追従させるよりも、全員で足並みをそろえて歩いたほうが、ずっと先まで進めるのである。

テスター

XPチームのテスターは、システムレベルの自動テストの選択や記述について開発前に顧客を支援したり、プログラマーにテスト技法をコーチしたりする。XPチームのなかで、つまらないミスを捕捉する責任の大半を引き受けるのはプログラマーである。**テストファーストプログラミング**からは、プロジェクトを安定させるテストスイートがもたらされる。つまり、テスターの役割は開発の初期段階に移行しているのだ。機能が実装し終わる前に、システムの受け入れ可能な機能を構成する要素の定義や仕様化を支援するのである。

週次サイクルでは、選択した**ストーリー**をシステムレベルの自動テストに変換するところから始める。この部分は強力なテストスキルを活用できるところだ。顧客

は自分が求める一般的な振る舞いについてはよく考えているかもしれないが、テスターはそうした「ハッピーパス」を視野に入れながら、そこから外れたときに何が起きるかを質問するのが得意である。たとえば、「わかりました。それでは、ログインが 3 回失敗するとどうなりますか？ 何が起きればよいでしょうか？」と聞くことができる。ここでは、テスターがコミュニケーションを増幅する役割を担っている。システムレベルのテストは、ストーリーが完全に実装され、デプロイの準備が整ったときにはじめて成功する。それを確実にするのが、テスターだ。

その週のテストを書いて、失敗することを確認したあとは、実装によって新しく仕様化が必要なところが明らかになってきたときに、引き続き新しいテストを書いていく。テストの自動化や調整などを行うこともできる。プログラマーがテストで行き詰まっていたら、ペアになって問題解決を支援することもできる。

インタラクションデザイナー

XP チームのインタラクションデザイナーは、システム全体のメタファーを選んだり、ストーリーを書いたり、デプロイされたシステムの利用状況を評価して、新しいストーリーの機会を見つけたりする。最終的なユーザーの課題に取り組むことが、チームの優先事項である。ペルソナやゴールといったインタラクションデザインのツールは、本物の人間との会話の代わりにはならないが、チームがユーザーの世界を分析したり理解したりするのに役立つ。

インタラクションデザイナーに対するアドバイスの多くは、開発のフェーズモデルに準拠している。フェーズモデルでは、求められるシステムをインタラクションデザイナーが特定してから、プログラマーがそれを実現することになる。フェーズはフィードバックを減らし、バリューの流れを制限する。開発をフェーズに分割しなければ、インタラクションデザイナーとその他の XP チームのメンバーとの相互利益が成立するだろう。

XP チームでは、インタラクションデザイナーは顧客と一緒に働いて、ストーリーの記述や明確化を支援する。ここでは、インタラクションデザイナーがいつも使っているツールを利用できる。それから、システムの実際の利用状況を分析して、次にシステムに求められるものを決定する。インタラクションデザイナーは、事前に少しだけ仕様化してから、プロジェクト期間全体でユーザーインターフェイスを洗練させていく。

アーキテクト

XPチームのアーキテクトは、大規模リファクタリングの調査や実施をしたり、アーキテクチャにストレスを与えるシステムレベルのテストを書いたり、ストーリーを実装したりする。アーキテクトはプロジェクト期間中に、少しずつ専門知識を適用する。プロジェクトのアーキテクチャを進化させながら、方向付けをしていく。小さなシステムのアーキテクチャは、大きなシステムのアーキテクチャと同じであってはいけない。システムが小さければ、それにあった大きさのアーキテクチャにしなければいけない。システムの成長に合わせてアーキテクチャを追従させるのがアーキテクトの仕事だ。

アーキテクチャの大幅な変更を小さくて安全な手順で行うことは、XPチームの挑戦である。権限と責任のバランスを保つ原則[†1]では、意思決定の権限を誰かひとりに与え、その決定に他の人たちが当事者意識を持たずに従うのは、よくない考えだとしている。したがって、アーキテクトも他のプログラマーと同じようにプログラミングのタスクにサインアップする。ただし、アーキテクトは大きな見返りのある大きな変更についても目を光らせておく。

テストによってアーキテクチャの意図を伝えることもできる。主要なクレジットカード処理システムのアーキテクトと話をしたときに、このような負荷の高い環境では、邪魔になりかねないアーキテクチャは不要であると言っていた。彼のチームには、高機能なストレステストの環境が用意されていた。アーキテクチャを改善したいときには、まずはシステムが壊れるまでストレステストを実施してから、ストレステストが実行できるようにアーキテクチャを改善するのである。

この戦略を他の会社のアーキテクトに提案したことがある。そのアーキテクトは、仕様書を書いて開発者に説明することにすべての時間を使っていた。彼は、そのことに対して不満を漏らしていた。もはやコードを書く時間がないことに苛立っていた。そこで私は、テストインフラを用意して、仕様書や説明の代わりにテストを使ってみてはどうかと提案した。設計に欠陥があれば、それを指摘するための失敗するテストを書くべきだ。そのアーキテクトを納得させることはできなかったが、このアイデアは今でも有効だと思っている。

XPチームのアーキテクトのもうひとつの仕事は、システムの分割だ。分割は事前に行う一度きりのタスクではない。XPチームは統治分割を行う。分割統治ではない。つまり、小さなチームで小さなシステムを作ってから、自然な切れ目を見つけ、比較的独立した単位に分割して、システムを拡張するのである。アーキテクト

[†1] 訳注：責任の引き受けの原則

は、グループが小さなセクションに集中しているときにシステム全体に目を配りながら、全体像を心に描いて最適な切れ目の選択を支援するのである。

プロジェクトマネージャー

XP チームのプロジェクトマネージャーは、チーム内のコミュニケーションを円滑にしたり、顧客、サプライヤー、その他のチーム外の組織とのコミュニケーションを調整したりする。プロジェクトマネージャーは、チームの歴史学者となり、チームに進捗状況を思い出させる。プロジェクトの情報をまとめて、経営幹部や同僚にプレゼンするために、クリエイティブでなければいけない。正確性を保つために、プロジェクトの情報を頻繁に変更することになる。したがって、プロジェクトマネージャーには変更をうまく伝えることが求められる。

XP の計画づくりは活動であり、フェーズではない。プロジェクトマネージャーには、計画と現実を同期し続ける責任がある。プロジェクトマネージャーは、計画プロセス自体の改善を推進する立場にいることが多い。週のはじめに 1 日かけて計画しているチームもあるだろうが、継続的に改善していけば、より短い時間でよりよい結果が得られるようになる。ベストなチームは週の作業をわずか 1 時間で計画する。だが、ここまで効率的にやるには練習が必要だ。

情報の流れはチームに対して双方向に出入りする。プロジェクトマネージャーは、顧客、スポンサー、サプライヤー、ユーザーからチームへ向かうコミュニケーションを円滑にしなければいけない。コミュニケーションを円滑にするためには、必要に応じてチーム内の適任者をチーム外の適任者に紹介して、コミュニケーションのボトルネックにならないようにする必要もあるだろう。それから、チーム内のコミュニケーションを円滑にして、一体感や信頼関係を築くようにしなければいけない。重要な情報の管理者になるよりも、効果的なファシリテーターになるほうが、得られる力は大きい。

プロダクトマネージャー

XP のプロダクトマネージャーは、**ストーリー**を書いたり、**四半期サイクル**のテーマやストーリーを選択したり、**週次サイクル**のストーリーを選択したり、実装によって明らかになったストーリーのあいまいな部分の質問に答えたりする。プロジェクトの開始時にストーリーの選択が終わったら、あとは椅子に座って何もしなくてもいいわけではない。XP の計画とは、これから**何が起こるか**を予測したものではなく、**何が起こり得るか**を示したものである。

チームがオーバーコミットしていたら、想定していた要件と現実の違いを分析して、チームが優先順位をつけられるように支援する。プロダクトマネージャーは、今実際に起きていることにストーリーやテーマを適応させるのである。

ストーリーの順番は、技術ではなくビジネスの理由で決めるべきだ。最初の週からシステムを動作させることが目的である。プロダクトマネージャーは、必ずしも最初から最後まで仕事をする必要はない。以前、ある計画ツールのプロダクトマネージャーと話をしたことがある。彼は最初に編集機能を試したいと考えていたが、プログラマーはユーザーが情報を入力できない状態で編集するのは合理的ではないと考えた。そこでプロダクトマネージャーは、プログラマーが手作業で入力できるようにダミーデータを用意した。編集機能がプロダクトのなかで最もバリューの高い部分だったからだ。手作業でデータを入力した結果、全員が編集機能の必要性を理解することができた。こうして**チーム全体**がプロダクトの中心部分に早めに目を向け、時間をかけて洗練できるようになった。

最初の週次サイクルの終わりには、システムは「完全」でなければいけない。たとえば、画像処理を計画していたら、最初の週の終わりに画像を処理できるようになっていなければいけない。プロダクトマネージャーは、それを実現するために必要なストーリーを選択するのである。

プロダクトマネージャーは、顧客とプログラマーのコミュニケーションを促進する。顧客の最も重要な課題がチームに伝わり、きちんと対処されるようにしなければいけない。チームが**本物の顧客参加**を実践していれば、ストーリーを選択した顧客やマーケット全体のニーズを満たせるように、システムの成長を促さなければいけない。これがプロダクトマネージャーの責任だ。

経営幹部

経営幹部は、XPチームに勇気、自信、説明責任を提供する。共通のゴールに向かって一緒に進んでいくXPチームの強みは、弱みにもなり得る。チームのゴールが会社のゴールと合っていなかったらどうなるだろう？ 成功のプレッシャーと興奮によって、ゴールを見失ってしまったらどうなるだろう？ 大きなゴールの明確化と維持は、XPチームのスポンサーや監督を務める経営幹部の仕事である。

XPチームのスポンサーや監督となる経営幹部のもうひとつの仕事は、改善の監視、促進、円滑化である。XPの目的は、優れたソフトウェア開発を標準にするというものなので、経営幹部はチームが作り出す優れたソフトウェアだけでなく、継続的な改善についても目を配らなければいけない。

経営幹部は、XPプロジェクトのあらゆる側面について自由に説明を求めることが

できる。説明は筋が通ったものでなければいけない。筋が通っていなかったなら、経営幹部はチームに対して考察と明確な説明の提供を求めるべきである。

経営幹部はチームのあらゆる意思決定のプロセスにおいて、選択肢に関する正直で明確な説明を求めるべきだ。経営幹部は問題に直面しても、全体像を見失ってはいけない。スコープを削減する必要性に迫られたとしても、組織の真のニーズやプロジェクトの要件に集中する。意思決定が必要になったときには、こまめでオープンなコミュニケーションによって、すでに必要な情報を手に入れていなければいけない。

私は、XP チームの健全性を計測する 2 つのメトリクスを信頼している。まずは、開発後に発見された欠陥数だ。XP チームであれば、最初の開発後に欠陥数が劇的に減り、そこから急速に進歩しているはずである。改善の道を数年間歩んできた XP チームのなかには、年間の欠陥数がごくわずかしかないチームもある。許される欠陥などないのだが、欠陥のおかげでチームは学習や改善の機会が得られるのである。

私が使っているもうひとつのメトリクスは、アイデアに投資してから収益が生まれるまでのタイムラグだ。小さな組織であっても投資を回収するまでに通常は 1 年以上かかる。この投資回収期間を少しずつ短縮すれば、チーム全体が利用できるフィードバックの量やタイミングが改善されていく。

開発後の欠陥数と投資回収期間は、いずれもチームが優秀であることを示している。速度計が速度を示すのと同じだ。速度計の針をつかんで動かしても車の速度は上がらない。速度を上げたければ、アクセルを踏んで、その効果が現れているかを速度計で確認するしかない。それと同じように、メトリクスにもとづいてゴールを設定できたとしても、チームはその根本にある問題に取り組む必要がある。数字の「ゲーム」に挑戦しても数字以外は何も改善されない。むしろプロジェクトにおける透明性が台無しになってしまう。

XP チームが組織の期待にそわない可能性もある。チームのことを積極的に組織にプレゼンすることも経営幹部の仕事の一環だ。チームは自分たちの功績だけで成否を評価されたいと思っている。経営幹部は批判を受けても勇気を持って前進しなければいけない。チームが新機能をこまめにデプロイし始めたら、バリューの流れのボトルネックは組織の別のところに移動するだろう。経営幹部は、この移動がポジティブなものであるとあらかじめ会社に認識させておく必要がある。制約が移動したことによって、他の部門に問題があるかのように見える可能性があるからだ。それによって、ソフトウェア開発を元の状態に戻せと言われるかもしれない。それでも経営幹部は、毅然とした態度を貫く準備をしておかなければいけない。

XP チームの評価を決める人たちは、優秀なチームがどのようなものかを理解す

べきである。それまでに見たことのあるチームとは違っているかもしれない。たとえば、XPチームは会話しながら仕事をする。にぎやかな話し声は健全である証拠だ。静寂はリスクがたまっている音色だ。経営幹部がXPチームを理解し、自身の経験や視点をうまく適用するには、新しい経験則を学ぶ必要があるだろう。

テクニカルライター

XPチームにおけるテクニカルパブリケーション[†2]の役割は、フィーチャーのフィードバックを早期に提供したり、ユーザーとの密接な関係を築いたりすることである。ひとつめの役割があるのは、最初にフィーチャーを目にするのがテクニカルライターだからだ。この段階ではまだ概略しかないという場合も多い。チームの部屋に同席していれば「これはどのように説明すればいいですか？」と質問できる。優れた説明方法があるかもしれないし、そんなものはないかもしれない。いずれにしてもチーム全体として学んでいけばいい。文章と図を使ったシステムの説明は、チームにフィードバックをもたらす要素のひとつである。

もうひとつの役割は、ユーザーとの密接な関係を作り出すことである。ユーザーがプロダクトを学習できるように支援したり、ユーザーからのフィードバックを受け取ったり、ユーザーが混乱しないように発表資料や新しいストーリーを追加したりする。発表資料の形式には、チュートリアル、リファレンスマニュアル、テクニカルオーバービュー、動画、音声などがある。テクニカルライターは顧客の声に耳を傾けて、ユーザーが実際にプロダクトを使用する段階で起きる類いの誤解を把握しなければいけない。コミュニケーションが不十分であれば、今後の発表資料で補足する。プロダクトの欠点は計画プロセスのインプットになる。たとえば、ユーザーから批判を受けたら、その「ユーザーエラー」を解決できるようなストーリーを選択する。

XPにおけるテクニカルパブリケーションでは、システムをデプロイするときの変化の速度が課題となる。昔は開発の初期段階に仕様を凍結していたので、ライターが仕様をマニュアルに書き換えると同時に、プログラマーが仕様をコードに落とし込むことができた。だが、XPの世界では、ゲームの終盤になるまで全体の詳細な仕様は凍結されない。チームが優秀になればなるほど、終盤になっても大きな変更を受け入れるようになる。そして、テクニカルパブリケーションは常にそれを追いかけることになる。

だが、その距離をもっと縮めることはできる。今週のストーリーのドキュメント

[†2] 訳注：マニュアルやウェブサイトなどの技術的な刊行物全般を指す。

作成を翌週のタスクにすれば、ストーリーが完成した1週間後にドキュメントを完成させることができる。これが可能になれば、ストーリーと同じ週にドキュメントを完成させることにも挑戦できるだろう。

完璧なドキュメントはどのようなものだろうか？ それは、記述されたフィーチャーが完成すると同時にできあがるドキュメントだろう。改善の必要性があるとわかったときに、簡単に更新できるドキュメントだろう。コストのかからないドキュメントだろう。基本的な開発サイクルの時間を増やさないドキュメントだろう。ユーザーにとってバリューのあるドキュメントだろう。そのようなドキュメントを書くことは、チームにとって重要なことである。

XPの哲学とは、現状から始めて理想に向かって進むことだ。現状から少しでも改善できるだろうか？ 紙のマニュアルを使用することで、デプロイサイクルが6週間余計にかかっているとしたら、すべてを電子化することはできないだろうか？ デプロイの6週間後に紙のマニュアルを送付することはできないだろうか？ すでに電子化されていて、フィーチャーの2か月後にドキュメントが完成している場合は、作業時間を調整して2週間後に完成させることはできないだろうか？ 同じ週に完成させることはできないだろうか？

XPチームは実際の利用状況からフィードバックを得るべきだ。マニュアルをサイトに掲載しているなら、利用状況を監視できる。ユーザーがドキュメントを見ていなかったら、その部分を書くのはやめよう。そして、空いた時間をもっとうまく活用しよう。プロダクトに利用状況の追跡機能があり、ドキュメントを一緒にデプロイしているのであれば、ドキュメントの利用状況も追跡してもらおう。そうすれば、さらに多くのカスタマーバリューの提供につながるはずだ。

ユーザー

XPチームのユーザーは、開発中にストーリーの記述や選択の支援をしたり、専門領域の意思決定をしたりする。構築中のシステムと類似したシステムに関する幅広い知識や経験を持っていたり、システムを実際に利用するユーザーコミュニティーとの強い関係性を持っていたりすれば、そのユーザーは非常に大切な存在だ。ユーザーはコミュニティーの代表者であることを忘れないようにしなければいけない。コミュニティーの人たちと会話するまでは、影響範囲の大きな意思決定は遅らせるべきだ。チームには、その間に他のストーリーに取り組んでおいてもらおう。

プログラマー

XPチームのプログラマーは、ストーリーやタスクを見積もったり、ストーリーをタスクに分解したり、テストを書いたり、フィーチャーを実装するコードを書いたり、退屈な開発プロセスを自動化したり、システムの設計を少しずつ改善したりする。プログラマーは技術的に密接に協力しながら一緒に働く。たとえば、プロダクションコードにはペアで取り組む。つまり、プログラマーは社交性や人間関係のスキルを身に付ける必要がある。

人事

チームがXPを適用し始めるとき、人事評価と雇用という2つの課題が発生することが報告されている。人事評価の課題が発生するのは、XPはチームのパフォーマンスに集中しているのに、実際の人事評価や昇給は個人の目標や達成度に対して行われているからだ。プログラマーがペアになって半分の時間を誰かと一緒に過ごしている場合、個人のパフォーマンスはどのように評価できるのだろうか？ 個人のパフォーマンスで評価されるとしたら、他人を助けるインセンティブはどれだけ残されているのだろうか？

XPチームのメンバーを個人として評価するといっても、XP適用前の評価の仕方を大きく変える必要はない。以下は、XPにおける重要性の高い従業員だ。

- ✧ リスペクトを持って行動できる。
- ✧ 他人とうまくやれる。
- ✧ イニシアチブをとれる。
- ✧ 約束したものをデリバリーできる。

人事評価の課題は、2つの方法で解決できる。このまま個人ベースの目標、評価、昇給を続けるか、チームベースのインセンティブや昇給に移行するかである。XPの透明性は、個人の評価に必要な情報をマネージャーに与えている。チームメンバーは、毎週みんなの目の前で、顧客が要求した仕事に直接関係のあるタスクにサインアップし、そのタスクを見積もり、実際に遂行している。一方、利他的行動が必要になるため、個人ではなくチーム全体として昇給させているところもある。それ以外のアイデアとしては、個人の評価や昇給に加え、優秀なチームワークにはボーナスを出すなどして、2つの方法をミックスするというものもある。

XPチームでの雇用は、既存の雇用方法とは異なる可能性がある。XPチームはチームワークとソーシャルスキルを重要視している。ズバ抜けたスキルを持ってい

るが孤独を好むプログラマーと、そこそこ優秀でソーシャルスキルのあるプログラマーがいれば、XP チームは常にソーシャルスキルのある候補者を選ぶ。チームと一緒に 1 日働いてもらうことが、最高の面接方法だ。**ペアプログラミング**は、技術スキルとソーシャルスキルを評価できる素晴らしい試験方法である。

役割

成熟した XP チームにおける役割は、固定化された不変のものではない。全員がチームの成功に最善を尽くして貢献することが目的である。とはいえ、最初は役割を固定化しておいたほうが、新しい習慣を身に付けるのに役立つだろう。たとえば、技術側の人間には技術的な意思決定をしてもらい、ビジネス側の人間にはビジネス的な意思決定をしてもらうのである。チームメンバーが相互にリスペクトした新しい人間関係を築くことができれば、こうした役割の固定化は全員が最善を尽くすという目的の妨げになるだろう。そのときは、プログラマーでも**ストーリー**を書けばいい。プロジェクトマネージャーでもアーキテクチャの改善を提案すればいい。

これまでにさまざまな役割が XP チームに貢献できると述べたが、単純に 1 人がひとつの役割を担うとは限らない。プログラマーがアーキテクトの役割を果たすこともある。ユーザーがプロダクトマネージャーに成長することもある。テクニカルライターであってもテストはできるだろう。抽象的な役割を果たすことではなく、チームメンバーがチームに最善を尽くして貢献することが目的だ。

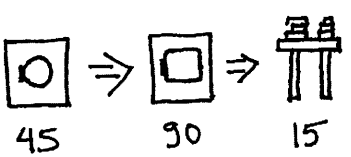
チームが成熟しても、権限と責任のバランスを保つことを忘れないようにしたい。チームの誰もが変化を提案できるが、その懸念を行動で裏付けられるように準備しておくべきである。

第11章

制約理論

まずは、どの問題が開発の問題かを見極めるところから、ソフトウェア開発の改善の機会を発見しよう。XPは、マーケティング、営業、マネジメントの問題を解決するものではない。ソフトウェア以外のボトルネックは重要ではないということではない。XPがそこまで網羅していないだけだ。その他の分野にもツマミを10にする方法はあるだろうが、それはXPのスコープ外である。XPはソフトウェア開発の問題に対処するための価値、原則、プラクティスをまとめたものだ。我々には、ソフトウェア開発の全体像を見る方法が必要である。

システム全体のスループットを見る方法のひとつに「制約理論（*Theory of Constraints*）」がある。洗濯室の例を使って、この理論を説明してみよう（図9）。洗濯機が衣類を洗濯するのに45分かかり、乾燥機が衣類を乾燥するのに90分かかり、衣類を畳むのに15分かかるとする。



▶図9　洗濯プロセスの現状

このシステムのボトルネックは乾燥だ。洗濯機が2台あっても、洗濯がすべて完了した衣類が増えるわけではない。洗濯だけが終わった衣類は一時的に増えるかもしれないが、濡れたままの衣類が至るところに山積みになり、その対応をしなけれ

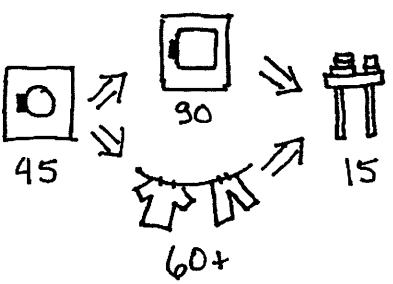
ばいけなくなる。その結果、洗濯がすべて完了した衣類の数は少なくなってしまうだろう。より多くの衣類の洗濯をすべて完了させたければ、乾燥をどうにかする以外に選択肢はない。

制約理論では、あらゆるシステムに同時に 1 つ（場合によっては 2 つ）の制約が存在すると考える。システム全体のスループットを改善するには、最初に制約を見つけなければいけない。次に、その制約が最大限に稼働していることを確認する。そして、制約のキャパシティーを増やすか、制約以外の負荷を下げるか、制約を完全に排除するかのいずれかの方法を探す。

では、システムの制約をどのように発見するのだろうか？ 仕掛品が山積みになっているところが制約である。これから畳まなければいけない乾燥した衣類は山積みになっておらず、これから乾燥する濡れた衣類が山積みになっている。

つまり、乾燥機が制約だ。乾燥機を最大限に稼働できるように、乾燥が終了したときに耳障りなブザーを鳴らすようにしよう。ブザーが鳴ったら、衣類を乾燥機から移動させる。洗濯機にはブザーは不要である。乾燥機から衣類を取り出したときに、次の洗濯を開始すればいいからだ。作業は実際の需要によってシステムからプルするものであり、需要を予測してシステムにプッシュするものではない。

洗濯する衣類を増やす必要があれば、大型の乾燥機を購入するか、排気口の詰まりを取り除いて乾燥速度を上げるなどして、乾燥のキャパシティーを増やさなければいけない。また、脱水機能に優れた新しい洗濯機を購入して、乾燥時間を短縮し、負荷を下げることもできるだろう。あるいは、図 10 のように衣類を屋外の太陽の下に干すこともできる。

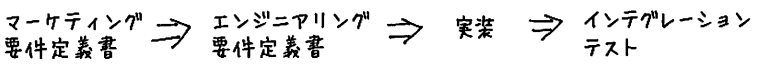


▶ 図 10　衣類を屋外に干してスループットを上げる

これで制約は、洗濯機か衣類を畳むテーブルに移動した。制約理論では常に制約があると考える。制約を排除すれば、また別の制約が作り出される。部分的な最適化では不十分だ。結果を改善するには、何を変えるかを決める前に、全体的な状況

に目を向けなければいけない。

ソフトウェアでは、開発システムの制約を特定する必要がある。図 11 は、あまり効果がないことでおなじみのウォーターフォール型の開発スタイルだ。



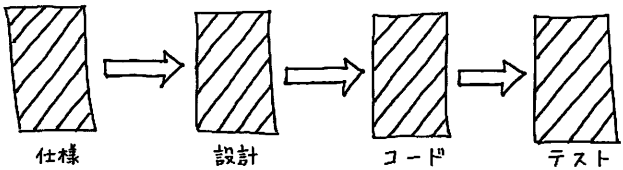
▶図 11　ウォーターフォール型プロセス

それぞれの工程にかかる時間を把握しても、ボトルネックを特定することはできない。ボトルネックを見つけるには、作業が山積みになっているところを探そう。たとえば、ER 図が多くのフィーチャーを網羅しているにもかかわらず、やることが多すぎて実装から外されている場合は、実装プロセスが制約になっているのだろう。多くのフィーチャーの実装が終わっているにもかかわらず、インテグレーションやデプロイが待機中になっている場合は、インテグレーションプロセスが制約になっているのだろう。

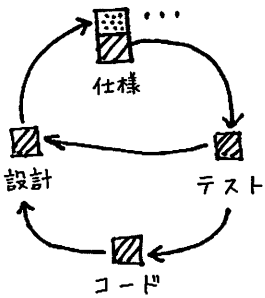
インテグレーションが制約になっているときに私が最初に確認するのは、インプットと環境に対して、可能な限りスムーズにインテグレーションが進んでいるかどうかである。おそらく実装からインテグレーションに人を移動できるはずだ。以前、20 人のプログラマーの作業を忙しそうに 1 人でデプロイしている人を見かけたことがある。何人かのプログラマーにお願いしてデプロイを手伝ってもらえば、全体的なスループットは上がるはずだ。

インテグレーションがスムーズになったら、負荷を上流に移動する方法を探す。たとえば、実装中にテストを自動化するのである。それによって実装の速度が下がる可能性はあるが、システム全体のスループットは改善される（むしろ実装速度は上がるのだが、それはまた別の章で触れよう）。

作業はシステムからプルすると言ったが、これはソフトウェアにも当てはまる。開発の「プッシュ」モデル（図 12）では、処理する要件を山積みにして、実装する設計を山積みにして、統合してテストするコードを山積みにして、最終的にその名もズバリの「ビッグバン」インテグレーションを行う。一方、XP では「プル」モデル（図 13）を使っている。ストーリーは実装直前に詳細に仕様化する。テストは仕様からプルする。プログラミングインターフェイスはテストのニーズに合わせて設計する。コードはテストやインターフェイスに合わせて書く。設計は書いたコードに合わせて洗練させる。これによって、次に仕様化するストーリーが決まる。その間、残りのストーリーは実装の対象に選ばれるまで壁に貼り付けておく。



▶図12　開発のプッシュモデル



▶図13　開発のプルモデル

制約理論では、組織全体が部分最適ではなく全体的なスループットに集中することが前提となる。その他の組織改革の理論と同様だ。誰もが自身の役割を制約と見なされることを拒んでいたら、何も変化は起きない。開発者が「自動テストを書くのは『よいこと』ですが、私の仕事の速度が落ちますし、今は負荷が高すぎます」と言っていたら、いくら組織やそこに属する個人に有益であっても、何も変化は起きないだろう。変化を貫き通すには、報奨制度や文化を個人の生産性ではなく、全体的なスループットに合わせる必要がある。

制約理論の弱点は、それがモデルであるという点だ。モデルは地図であり、現地ではない[†1]。ソフトウェアを開発するのは人だ。人はモデルのなかの箱ではない。効率的だが一見すると混沌としたコミュニケーションに組織が近づいていくと、制約理論の世界にマッピングして戻ってきたときの正確性が失われていく。ソフトウェア開発は工場ではなく人間のプロセスである。それでも制約理論は、自分のプロセスを認識する優れた方法だ。現在のプロセスを連結された一連の活動として描き、どこで作業が山積みになっているかを探してみよう。

[†1] 訳注：アルフレッド・コージブスキーの言葉「地図は現地ではない」に由来する。

制約理論はボトルネックの発見に役立つが、ボトルネックがソフトウェアと関係ない場合はどうすればいいのだろうか？ 私のクライアントには、セーフティクリティカルではないソフトウェアを飛行機にデプロイしているところがある。彼らの最初のパートナーは、飛行機が整備中のときにしかデプロイできなかったので、完成したソフトウェアを４か月後に飛行機にデプロイしていた。その後、毎日デプロイできるインフラを持った新しいパートナーを選び、クライアントはその制約を排除した（そして、制約はマーケティングに移動した）。ボトルネックがソフトウェア開発の外側に存在していれば、その答えはソフトウェア開発の外側に存在するのである。

XP のスコープを広げて、ビジネスパートナーの選択のような課題まで扱いたいと思うことがある。幅広く問題を網羅しなければ、XP が重要だと見なされないのではないかと不安になることもある。気分が晴れているときは、明確で限定されたスコープのビジネスにも XP は確実に重要であると信じている。

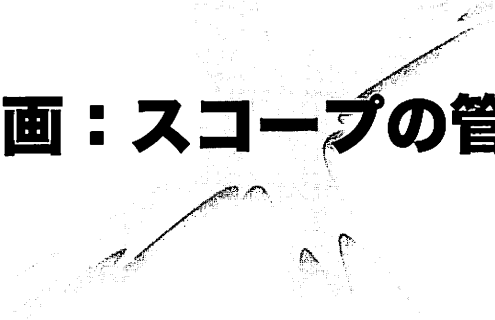
XP は、ソフトウェア開発の外側に制約を移動させることもできる。したがって、XP を適用する組織全体に波及効果をもたらす。たとえば、チームのビジネスに集中した人たち（プロダクトマネージャーや顧客など）が、その週に追加するフィーチャーを選ぶための週次ミーティングを開始したとしよう。すると、プログラマーの学習速度が速すぎて、プロダクトマーケティングがフィーチャーを仕様化できない可能性が出てくる。制約がソフトウェア開発の外側に移動したのである。

開発チームが XP の適用を開始して、品質や生産性が劇的に向上しても、チームは解散させられ、リーダーは解雇され、残りのチームメンバーはバラバラになってしまうという悲しい話が何度も繰り返されている。なぜこのようなことが起きるのだろうか？ 制約理論の視点から考えると、チームのパフォーマンスが改善された結果、制約が組織の別の場所に移動したのである。新しい制約（たとえば、必要とする機能を十分な速度で決定できないマーケティング）は、そうしたスポットライトを浴びたくない。組織全体のスループットを本気で気にしている人は誰もいない。その結果、騒動の「原因」である XP が非難され、排除されるのである。

経営幹部の支援やチーム外の人たちとの強い人間関係は、XP の適用に不可欠である。XP によってソフトウェア開発を整備すると、ソフトウェア開発以外の組織の仕事の構造も変化させてしまうからだ。経営幹部の支援がない場合は、たとえ評価や保護が得られなくても、自分自身でよりよい仕事をする覚悟を持たなければいけない。

第12章

計画：スコープの管理



共有された計画の状態は、その計画によって影響を受ける人たちの人間関係の状態を示すヒントになる。たとえば、現実離れした計画は、不明確で不安定な人間関係を表している。お互いの合意の上に成り立っており、現実の変化を反映して調整された計画は、リスペクトに満ちた相互に重要な人間関係を表している。

計画づくりによって、ゴールと方向性が明確でわかりやすいものになる。XP の計画づくりでは、現在のゴール、前提、事実を検討するところから始める。現在の明確な情報があれば、何をスコープに入れるのか、何をスコープから外すのか、次に何をするのかの合意に取り組むことができる。

XP の計画づくりは、食料品の買い物のようなものだ。ポケットに 100 ドルを入れて、食料品店に買い物に行くとしよう。棚にある品物にはそれぞれ値札がついている。必要なものもあれば、必要ではないものもある。欲しいと思うが、予算に合わないものもある。101 ドル分の食料品を抱えてレジに向かったとしたら、何かを戻さなければいけない。買い物におけるあなたの仕事は、100 ドルを賢く使うことだ。必要なものはきちんと購入し、欲しいものもできるだけ購入する。

XP では、食料品がストーリーである。値札はストーリーの見積りだ。予算は利用可能な時間である。希望するデプロイ日程は、通常はプロジェクトの初期に設定されるので、ストーリーにどれだけ費やせるかは事前に判明している。たとえば、ポケットに 200 ペア時間（*pair-hours*）を持っていて、カートに 400 ペア時間分のストーリーが入っているとすれば、バリューの高いストーリーから 200 ペア時間分を選択していくのである。ここで選択しなければ、予算を超えてカートに入れていることが誰の目にも明らかだ。

計画づくりとは、すべての可能性のなかから次に何をやるかを決めることである。ストーリーのコストやバリューの見積りは不確実なので、計画づくりは難しい。意思決定に使った情報は変化する。そのため、フィードバックを使って見積りを改善したり、可能な限り質の高い情報が使えるように、意思決定をできるだけ遅らせたりする。計画づくりが、XPの毎日、毎週、四半期ごとの活動なのはそのためだ。新しい事実が登場すれば、それに合わせて計画を変更できる。

計画は未来の予測ではない。明日起こりそうなことについて、今日知っていることを表したものにすぎない。計画の不確実性は、計画の重要性を否定するものではない。計画は他のチームとの調整に役立つ。計画によって出発点がわかる。チームの全員が、チームのゴールに合致した選択ができるようになる。

若いソフトウェアエンジニアだった頃、私はプロジェクト管理の3つの変数を学んだ。3つの変数とは、速度、品質、価格である。スポンサーは、3つのうち2つを固定しようとする。そして、チームは3つめを見積もることになる。計画が受け入れられなければ、そこから協議が始まる。

このモデルは実際にはうまくいかない。時間と費用はプロジェクトの外側で決まる。操作できる変数として残されているのは、品質だけである。品質を下げても、作業がなくなるわけではない。自分の責任だとわからないように、作業を後回しにしているだけだ。錯覚の進捗を生み出すことはできるかもしれない。だが、満足度の低下と人間関係の崩壊という代償を支払うことになる。満足度は質の高い仕事から生まれるのである。

このモデルから外されている変数はスコープだ。スコープを明確にすれば、以下のようになる。

- ✧ 状況に合わせて適応する安全な方法が得られる。
- ✧ 協議の方法が得られる。
- ✧ 非常識で不必要な要望を制限できる。

以下の4つのステップを使い、あらゆるタイムスケールで計画しよう。

1. 完成させる必要がありそうな作業項目を一覧にする。
2. 項目を見積もる。
3. 計画サイクルの予算を設定する。
4. 予算内で完成させる作業に合意する。協議のときには見積りや予算を変更しない。

チームにいる全員の意見を聞く必要がある。計画づくりはチームが要望を認識す

る会議の場だが、専念するのはニーズだけだ。

この手順は、ペアになったプログラマーにも使える。満たしたいテストケースや改善したい設計を数時間かけて計画するのである。また、チームがその日の活動を計画するのにも使える。それから、チームが１週間や四半期を計画するのにも適している。その場合は、一覧にある作業項目をストーリーにして、それぞれをきちんと見積もり、数時間から数日かけて計画する。

計画づくりは共同作業だ。それには協力が必要である。計画づくりとは、規定の時間内に耳を傾け、意見を出し合い、ゴールの認識を合わせるものである。チームメンバーにとっては重要なインプットになる。計画づくりをしなければ、場当たり的な関係性と有効性を持った個人の集まりになってしまう。お互いに調和をとりながら計画づくりや仕事するときは、全員がチームだ。

チームの全員が計画づくりに参加すべきである。XP チームのなかには、チームに参加している顧客に非公式に集まってもらい、翌週の予算について議論してもらっているところもある。プログラマーをゼロサムゲームから遠ざけて、オーバーコミットさせないようにするためだ。だが、これでは**相互利益**の機会が失われてしまう。**チーム全体**が一致団結したほうが、一見すると発散的な顧客のニーズを満足させる方法が見つかるだろう。ニーズやそれに関係する問題をチームがすべて把握しておかなければ、ビジネス的にも技術的にも優れた選択をすることはできない。

次に実装するストーリーを選択するときは、複数の方法で並び替えよう。ストーリーを空間的に並べれば、ストーリーの関係性が新たに見えるようになり、選択プロセスがスムーズになる。リスクの高いストーリーを左に配置したり、バリューの高いストーリーを上に配置したりすることもできる。パフォーマンスチューニングのストーリーを机の隅に集めたり、新機能のストーリーを別の隅に集めたりすることもできる。計画づくりで行き詰まったときは、すべてのストーリーを机から取り除き、シャッフルしてから新規に並べ直すといいだろう。

ストーリーを見積もるときには、類似したストーリーについて知っていることをすべて考慮した上で、ペアでストーリーを完成させるのに何時間あるいは何日かかるかを想像する。「完成」とは、デプロイの準備ができていることを意味している。したがって、テスト、実装、リファクタリング、ユーザーとの議論なども含まれる。類似したストーリーの知識が増えれば、それだけ見積りの精度も高まる。見積りは普通のペアがストーリーに取り組んだときを想定する。優秀なペアもいれば、そうではないペアもいるだろう。だが、全員で見積りをすれば、平均的な値に落ち着くはずだ。

これらの見積りは、最初は大幅に間違っている可能性がある。経験にもとづいた

見積りのほうが正確だ。見積りを改善するには、できるだけ早くフィードバックを手に入れることが重要である。プロジェクトの詳細な計画に1か月間使えるとしたら、1週間の開発イテレーションを4回実施して、見積りを改善していけばいい。計画に1週間使えるとしたら、1日のイテレーションを5回実施しよう。フィードバックサイクルによって、正確な見積りに必要な情報や経験が手に入る。こうした経験をできるだけ早く積めば、見積りが改善されるはずだ。

これで、食料品（ストーリー）と価格（見積り）がわかった。それでは、予算（納期とチーム規模）はどのように設定すればいいのだろうか？ まずは、生産的なプログラマーが平均的な週に作業できる時間を計測しよう。ペアで仕事をするので、その時間を2で割る。たとえば、チームに6人のプログラマーがいて、1日に1人4時間プログラミングできるとすれば、1日12ペア時間で1週間を計画する。ペアに対する反論として、プログラミングの時間が半分になるというものがある。私の経験からすると、ペアになると2倍以上の効果がある。実際に私がタスクを完了させるのに必要な時間を比較したところ、デバッグの時間も含めて、単独でやったときのほうがペアでやったときよりも2倍以上の時間がかかった。つまり、ペアでやるほうが完全でクリーンなコードが手に入るのである。ペアと個人を比較するときは、デプロイ可能なコードにかかった時間と生産性の両方を考慮する必要がある。時間内かつ予算内にバリューのあるソフトウェア開発を成し遂げることが目的だ。計画の数値は重要だが、それはすべてこの目的のためである。計画づくりにおいては、関連する数字をすべて計算に含める必要がある。

第1版では、もっと抽象的な見積りモデルを紹介した。ストーリーのコストに1、2、3という「ポイント」をつけていたのである。大きなストーリーは計画する前に分割する必要があった。ストーリーの実装が始まったときに、1週間で達成できるポイント数をすばやく把握するのである。だが、今では実時間で見積もるほうが私の好みだ。そのほうができるだけ明確で、直接的で、透明性のあるコミュニケーションがとれるからである。

1日に行える作業量には限界がある。この限界に注意を払うことにより、効果的に計画したり、うまくデリバリーしたりすることが可能になる。プログラマーは2倍の作業をすべきだ、と言ってもうまくいくはずがない。スキルや能力を伸ばすことはできるが、作業できる量を需要に応じて増やすことはできない。机に向かう時間を増やしても、クリエイティブな仕事の生産性が高まるわけではない。

実時間とポイントのどちらの単位を使っても、現実と計画が合わない状況への対応は必要になる。実時間で見積もっている場合は、まだ完成していないストーリーの見積りを経験を踏まえて修正する。ポイントで見積もっている場合は、今後のサ

イクルで予算を修正する。先週の実績とまったく同じ作業量で計画するのがシンプルな方法だ。マーティン・ファウラーはこれを「昨日の天気」と呼んでいる。新しい情報が得られたらすぐに計画を調整しよう。

サイクルの途中で計画よりも進捗が遅れることもあるだろう。そのときは計画を再調整する方法を見つけよう。あまり重要ではない課題に気をとられていないだろうか？ 役に立つプラクティスなのに、手を抜いていないだろうか？ プロセスの変更によって、計画と実績のバランスが回復できないなら、顧客に最初に完成させたいストーリーを選択してもらおう。そして、そのストーリーだけに集中しよう。あらためて計画づくりをすることになるので、余計に時間はかかってしまうが、それによってチームがデプロイの期日に向かって取り組むときの調和や効率性が高まる。ここで調整しなければ、ウソをついたまま仕事をすることになる。誰もがそのことをわかっていて、自分自身を守るために責任逃れをすることになる。これでは優れたソフトウェアを完成させ、デプロイすることなどできない。そして、チームや個人の信頼が損なわれていく。

物事がうまくいかないときこそ、価値と原則を忠実に守り、プラクティスを修正して、有効性をできるだけ維持する必要がある。不正確な見積りは情報不足が原因であり、価値や原則が不足しているわけではない。数字が間違っていれば、数字を修正して、その結果を伝えよう。

ストーリーをインデックスカードに書いて、人目につきやすい壁に貼り付けよう。このステップを省略して、いきなりコンピューターでストーリーを扱おうとするチームが多い。だが、それでうまくいった試しはない。コンピューターで管理しているからといって、ストーリーのことを強く信じるようになる人はいない。みんなで話し合いをするからこそ、ストーリーは重要なのである。カードはツールだ。ゴールに向かって話し合いや調整を行うことが、ストーリーの共通認識であり、それが最も重要な部分である。人間関係は自動化できない。誰もが信じられる計画を作り、そのために邁進できるようにすることが目的だ。

プロジェクトには力のバランスがある。仕事をうまく完成させる人と、完成した仕事が必要な人がいる。その両方が、信じられる計画づくりに必要な情報を持っている。壁に貼ったカードは、透明性の実践だ。それぞれのチームメンバーのインプットを大切にして、リスペクトするものである。

プロジェクトマネージャーには、チーム外の組織が期待する形式にストーリーを翻訳する仕事がある。彼らに壁の読み方を教えてもいい。隠すものは何もない。誰もが閲覧できて、誰もが利用できる。それこそが、最も有益なソフトウェア開発につながる人間関係を反映した計画だ。

第13章

テスト：早めに、こまめに、自動化

欠陥は効果的なソフトウェア開発に必要な信頼をぶち壊す。顧客はソフトウェアを信頼できなければいけない。マネージャーは進捗報告を信頼できなければいけない。プログラマーはお互いに信頼できなければいけない。欠陥はこうした信頼をぶち壊す。信頼できなければ、誰かが間違いを犯す可能性から自分の身を守ることに時間の大半を費やすだろう。

だが、すべての欠陥を取り除くことは不可能である。平均故障間隔を1か月から1年に伸ばすには、膨大なコストがかかるだろう。1世紀に伸ばすには、スペースシャトルを飛ばすのに必要なコードのように、天文学的なコストがかかる。

ここにソフトウェア開発のジレンマがある。欠陥はコストだが、欠陥を取り除くにもコストがかかる。だが、多くの欠陥のコストは、それを防止するコストを上回る。欠陥が発生すると、それを修正する直接コストと、人間関係の崩壊、ビジネスの損失、開発時間の喪失といった間接コストの両方がかかる。XPのプラクティスは、明確なコミュニケーションを目的としている。最初から欠陥が発生しないようにするためだ。欠陥が発生したときには逆にそれを利用して、今後発生する同様の問題の回避方法をチームが学べるようにする。

欠陥は常に存在する。想定外の状況も発生する。これまでにない不測の事態に陥ると、ソフトウェアが作者の意図に反する挙動を行う可能性がある。

許容可能な欠陥レベルはさまざまである。開発におけるひとつの目的は、経済的に持続可能なレベルまで欠陥の発生を抑えることだ。このレベルはソフトウェアの種類によって異なる。たとえば、世界最大のウェブサイトでは、1秒間に100件のエラーが発生していたとしても、99.99%のページは正しく表示されているため、

経済的には持続可能なレベルだろう。閲覧したすべてのユーザーが、このウェブサイトは信頼できると思えるはずだ。一方、命にかかわるスペースシャトルでは、ソフトウェアの欠陥は1世紀に1つまでに制限されているはずである。

開発におけるもうひとつの目的は、チームの信頼が適度に成長するレベルまで欠陥の発生を抑えることだ。欠陥削減に対する投資は、チームワークに対する投資としても意味がある。ひとりのプログラマーが間違いを犯せば、他の全員の仕事が困難になってしまう。ひとりのチームメンバーが他のメンバーに影響を与える間違いを犯せば、チームの時間、エネルギー、信頼が損なわれる。優れた仕事や優れたチームワークが、モラルと信頼を築くのである。同僚をリスペクトして信頼できれば、生産的になり、仕事をもっと楽しめるようになる。自己防衛のために失敗を隠すのは、それが必要なときもあるだろうが、膨大な時間とエネルギーのムダだ。信頼は関係者をいきいきとさせる。物事がスムーズに進んでいると気持ちがいい。実験や失敗は安心してできなければいけない。害のないことを示すために、実験の説明責任を果たすような試験を行う必要がある。

最近までほとんどチームは、欠陥と共存することを選んできた。欠陥のない（1か月から1年に欠陥が1つ程度の）コードは、不可能であると考えられてきた。欠陥率を半分にするだけでも、金銭的にも時間的にもコストがかかりすぎると思われてきた。**ペアプログラミング**などのXPのソーシャルプラクティスの多くは、欠陥を削減してくれる。XPのテストは、欠陥に直接対処する技術的な活動だ。XPでは、テストのコストパフォーマンスを向上させる2つの原則を適用している。「ダブルチェック」と「欠陥コスト増加」だ。

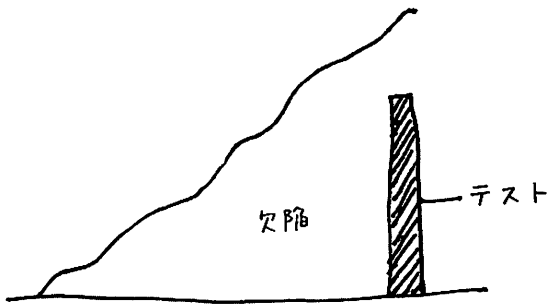
たとえば、縦に並んだ数字を上から順番に足していくとしよう。おそらく計算ミスが多発して、合計を間違えると思う。だが、足し算の順番を上からと下からの2方向にすれば、両方の合計が一致していたときに答えが合っている可能性が高い。2回とも同じ間違いが発生する可能性は低いからだ。

ソフトウェアテストはダブルチェックだ。テストを書くときに、コンピューターにやってほしいことを伝える。その処理を実装するときには、同じことをまったく別の言い方で伝える。2つの処理の表現が一致していれば、**コードとテスト**が調和して、正しくなる可能性が高い。

欠陥コスト増加（DCI：Defect Cost Increase）は、テストのコストパフォーマンスを高めるためにXPが適用しているもうひとつの原則である。DCIは、ソフトウェア開発において経験的に裏付けされた数少ない真実だ。欠陥を発見するのが早ければ、それだけ修正するコストは安くなるのである。たとえば、欠陥の発見が開発の10年後だとしたら、コードの当初の意図、その前提が間違っていた箇所、残

りの（正しいと思われる）プログラムが影響を受けないようにどこを修正すべきか、などを解明するために、膨大な歴史や文脈を再構築する必要がある。同じ欠陥であっても、作成直後に捕捉すれば、その修正コストは最小限になる。

DCI は、フィードバックループの長いソフトウェア開発スタイルはコストが高く、多くの欠陥が残ることを示している（図 14）。欠陥の発見や修正の予算は限られている。発見や修正のコストが高ければ、デプロイされたコードにそれだけ多くの欠陥が残ることになる。



▶図 14　時期が遅くコストの高いテストは多くの欠陥を残してしまう

XP は DCI を逆に利用して、欠陥の修正コストとデプロイされた欠陥数の両方を削減している。プログラミングのインナーループに自動テストを持ち込むことで、XP はコストをかけずに早めに欠陥を修正しようとしているのである（図 15）。このことは、同時代の標準と比べて欠陥が非常に少ないソフトウェアを安価に開発する機会を XP チームにもたらしている。

▶図 15　こまめなテストはコストと欠陥数を下げる

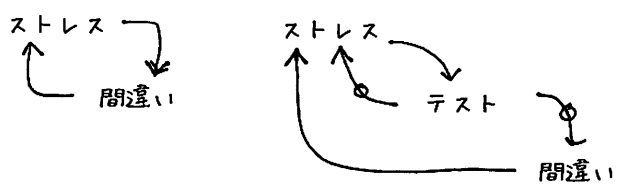
こまめなテストには複数の意味がある。そのひとつは、間違える可能性のある本人がテストを書かなければいけないというものだ。欠陥の作成から検出までの間隔が数か月であれば、それぞれの活動を別々の人が担当してもおかしくない。間隔が数分だとしても、プログラマーごとに専任のテスターがいたとしても、お互いが期待していることを伝達するコストは膨大になるだろう。いずれにしてもプログラ

マーがテストを書かない限り、間隔を短縮することで得られるバリューを調整コストが上回ってしまう。

プログラマーがテストを書く場合も、システムを別の視点から見る必要があるだろう。システムの機能という単一の視点からコードとテストを見ているのであれば、たとえペアを組んでいたとしても、ダブルチェックの重要性の一部が失われてしまう。2つの異なる思考プロセスが同じ答えにたどり着いたときに、ダブルチェックの効果が最もよく現れる。したがって、計算結果に期待する値として計算結果をコピーするのは危険だ。それでは一度しか考えていない。別の視点を手に入れるために、手動で計算したほうがはるかにマシだ。

ダブルチェックの恩恵を完全に手に入れるために、XPには2セットのテストが用意されている。ひとつは、プログラマーの視点で書かれたテストだ。システムのコンポーネントを徹底的にテストするものである。もうひとつは、顧客やユーザーの視点で書かれたテストだ。こちらはシステム全体の操作をテストするものである。これらのテストはお互いにダブルチェックしている。プログラマーのテストが完璧であれば、顧客のテストでエラーが捕捉されることはないはずである。

XPではすぐにテストを行うが、それはテストを自動化しなければいけないという意味でもある。自動テストと手動テストに関する難解な議論を読んだことがあるが、XPでは議論の余地はない。チームは設計を改善したり、開発ツールをカスタマイズしたりすることで、時間をかけてテストを自動化するコストを下げていく。そして、最終的にはすべてのテストを自動化する。自動テストはストレスのサイクルを断ち切るのである（図16）。



▶図16　ストレスのサイクル

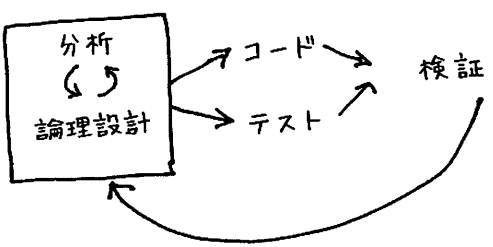
手動テストでは、チームのストレスが増えると、コーディングとテストの両方で間違いが増える。自動テストでは、テストを実行することがストレスの解消になる。したがって、チームのストレスが増えると、実行するテストが増える。テストによって、プログラマーによる検出を逃れるエラーの数も減る。

ベータテストは、テストプラクティスが貧弱であり、顧客とのコミュニケーションが不足している兆候である。だが、早めにこまめにテストできるようになるまで

は、現在のテストプラクティスをそのままにしておいたほうが賢明だ。チームの目的は、開発後のすべてのテストを排除して、テストの資源を開発ライフサイクルの最もレバレッジの高い部分に移動することである。なお、開発が「完了」したあとに欠陥を発見するストレステストやロードテストなどは、開発サイクルに取り入れておいて、継続的に自動的に実行しよう。

静的検証はダブルチェックの有効な形式だ。特に動的に再現するのが困難な欠陥に有効である。静的チェックの重要性を高めるには、もっと高速にして開発のインナーループの一部にすればいい。静的チェッカーは現在のインクリメンタルコンパイラと同じように、プログラムの変更後数秒以内にフィードバックを提供するものでなくてはいけない。現状では、相当量のプログラムの静的検証に数日間もかかってしまうが、プログラムの並行性について自信が得られるならば、それでも有益である。その他のテストと同様に、プログラムをダブルチェックする必要があるときに、静的検証のステートメントを少しずつ書いていけばいい。

DCI は、テスティングとコーディングを近づけることは教えてくれるが、正確にいつテストするかまでは教えてくれない。実装後のテストは実にわかりやすい。物理的な耐性のように、作る前にテストができない部分もあるからだ。ここは「テスティング」の言葉の背景にある物理的なメタファーが誤解を招いているところである。ソフトウェアはバーチャルな世界なので、事前にしろ事後にしろ「テスティング」に違いはない。鋳型に合わせたコードを書くこともできるし、コードに合わせて鋳型を作ることもできる。より利益のあるほうを選べばいい。最終的には両者を組み合わせ、一致するかどうかを確かめるのだ（図 17）。



▶図 17　コードとテストの順番はどちらでも構わない

コードとテストはどちらを先に書いても構わない。XP では、できるだけ実装前にテストを書く。テストを先に書く利点はいくつかある。ソフトウェア開発では、インターフェイスは実装に大きな影響を受けてはいけないとされている。テストを先に書くのは、インターフェイスと実装を分離する具体的な方法だ。また、テストは

確信を求める人間の欲求を満たす。失敗すると考えられるすべてのテストを書き、それらをすべてパスさせれば、自分の書いたコードが正しいと確信を持てる。失敗するかどうかまだよくわからないテストについても、あらかじめテストを書いておけば、少なくともシステムの挙動を示すことができる。

テストによって進捗を計測することもできる。10 件のテストが失敗していて、そのうちの 1 件を修正したとする。そうすれば、その分だけ進捗していることになる。ただし、私は失敗するテストが 1 件だけになるようにしている。テストファーストでプログラミングするときは、失敗するテストを 1 件だけ書いて、それを動くようにしてから、次の失敗するテストを書いている。私の経験からすると、最初のテストを動かしたときに、次に書くテストの情報が得られることが多い。検証していない仮定を前提にしてテストを書くと、仮定が間違っていたときにすべてを修正しなければいけなくなる。システムレベルのテストがあれば、週の終わりにシステム全体が動いているという確信が持てるようになる。

XP では、テスティングはプログラミングと同様に重要なものである。テストを記述して実行すれば、チームは自分たちの仕事に誇りを持つ機会が得られる。予期しなかった方向にすばやく進むときでも、テストを実行することでチームの確信がしっかりと裏付けられる。テストによってチーム内や顧客との信頼関係が強化され、開発にバリューがもたらされるのだ。

第14章

設計：時間の重要性

インクリメンタルな設計は、機能を早期に届ける方法であり、プロジェクトの全期間にわたって毎週継続して機能を届ける方法である。XP は設計のインクリメンタル主義を極限まで高めている。設計が日常業務の一環になれば、プロジェクトがもっとスムーズに進むというものだ。本章では、インクリメンタルな設計を受け入れる技術的、経済的、人間的な理由について深く掘り下げる。

パーマカルチャーとは、バランスのとれたエコシステムにおける持続可能な生活の哲学および実践である。パーマカルチャーのように「各要素が有益に関連したシステム」が設計であると私は考えている。この定義は、それぞれの言葉に意味が込められている。「要素」は、システムは全体だけで理解されるわけではないことを意味している。「関連」は、設計の要素は単独で存在しているわけではなく、相互につながっていることを意味している。このように設計を要素に分割したり、関連を作ったりというのは、善かれあしかれ設計者が意識的あるいは無意識的に行っていることである。「有益に」は、関連によって要素が強化されることを意味している。それぞれの要素は単独でいるときよりも強力になり、システムは要素を個別に検討したときよりも優れたものになる。

設計はソフトウェアのバリューを高めるものである。物理的な世界とは異なり、ソフトウェアではコストをかけることなく無限に要素を複製できる。要素間の有益な関連を新しく作れば、それを既存のすべての要素の関連に広げることができる。ソフトウェアはレバレッジゲームだ。つまり、ひとつの優れたアイデアが何百万ドルものコストを削減したり、何百万ドルもの収益を生み出したりするのである。

残念ながらソフトウェアの設計は、物理的な設計活動のメタファーにとらわれて

いる。たとえば、50階建ての超高層ビルを持っているとしよう。すべての空間をすでに貸し出しているからといって、そこに別の50階を付け足すことはできない。巨大なビルを持ち上げて、基礎を強固なものに置き換える方法など存在しない。

これに相当する変更は、ソフトウェアにおいては日常的に発生している。たとえば、ある分散システムで、通信技術として最初にリモートプロシージャコールが採用されたとしよう。システムの経験が増えていくと、チームはCORBAに移行してシステムを改善する方法を発見する。1年後には、CORBAをメッセージキューに置き換える。この間、システムはずっと稼働している。段階を踏むごとに次の段階に移行する経験が得られる。たとえるなら、犬小屋から開始して、少しずつ部品を置き換えながら、基本的な構造はそのままにして、最終的に超高層ビルにするようなものである。物理的な世界ではバカげた話だが、ソフトウェア開発においては実践的でリスクの低い方法だ。

物理的な世界では、移行のコストがかかりすぎるため、インクリメンタルな設計はうまくいかない。中間段階のバリューが少なすぎたり、変更コストが高すぎたりするからだ（再建プロセスで部品が破壊されるので複製コストが高い）。だが、スチュワート・ブランドの『How Buildings Learn』で述べられているように、次の設計段階の情報を推測ではなく既存の構造の経験から得ることができれば、物理的な構造でもインクリメンタルな設計と建築は可能である。

ソフトウェアの世界でインクリメンタルな設計が重要なのは、アプリケーションをはじめて書く機会が多いからだ。似たようなテーマの繰り返しであっても、ソフトウェア設計には常に優れたものが存在する。設計には大きな影響力があり、設計のアイデアは経験によって改善される。したがって、ソフトウェア設計者が持つべき最も重要なスキルのひとつは忍耐だ。フィードバックが十分に得られる分だけ設計を行い、そこから得られたフィードバックを使って、次回のフィードバックが十分に得られる分だけ設計を改善する。そうした技能が求められるのである。

ソフトウェアでは、建築のメタファーがよく使われる。たとえば、スティーブ・マコネルは『Code Complete 第2版』（日経BP社）のなかで、ソフトウェアコンストラクションとして建築のメタファーを使っている。スミス大学で演劇を学ぶベス・アンドレス・ベックは、このメタファーには根本的な欠陥があると指摘している。建築の世界では、逆方向に進めることが極めて難しい。フロアプランを変更したければ、ペグや紐を動かすだけで無料で変更できる。2日後、基礎工事が終わったあとに同じ変更をしようとすると、1万ドルのコストがかかる。こうしたコストの非対称性が、建築における活動やその関係性を形作っている。

一方、ソフトウェアにおいては、1日の作業を逆方向に進めるのは大したことで

はない。失われるのはその日の作業だけだ。こうした根本的な違いがあるため、建築に適した一連の活動はソフトウェアには適していない。論点となるのは、設計を**するかどうか**ではなく、設計を**いつするか**である。

マコネルは以下のように書いている。

> 10 年間で「すべてを設計する」ことから「何も設計しない」ことへと、振り子は大きく揺れた。だが、BDUF（Big Design Up Front：前もって大部分を設計すること）に取って代わるのは、「何も設計しないこと」ではなく、LDUF（Little Design Up Front：前もって少しだけ設計すること）、または ENUF（Enough Design Up Front：前もって十分に設計すること）だ。[†1]

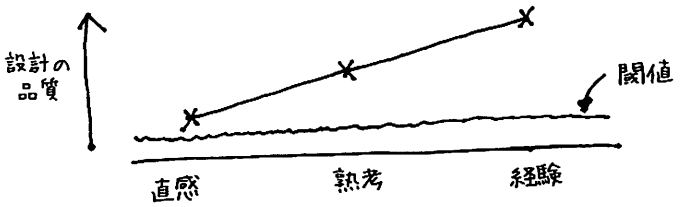
これは藁人形論法である。実装前の設計に取って代わるのは、実装後の設計だ。前もった設計は必要だが、最初の実装ができるだけで十分である。それ以上の設計は、実装後に設計の本当の制約が明らかになってから行えばいい。XP の戦略は「何も設計しない」ではなく「常に設計する」である。

これから紹介するグラフはいつ設計すべきかを可視化したものである。グラフの縦軸は設計の品質だ。成功に必要な最低限の品質を波線で描いている。ソフトウェアが成功するための設計はいくつもある。これがソフトウェア設計のおもしろいところだ。設計の品質は成功を約束するものではないが、設計の失敗は確実に失敗につながる。

グラフには 3 つの点を描いている。それぞれ、直感による設計、真剣に熟考したときの設計、経験にもとづいた設計である。3 つの点の関係性と最低品質の閾値の位置は、事前設計が選択肢になり得るのか、それともインクリメンタルな設計にすべきなのかを示している。

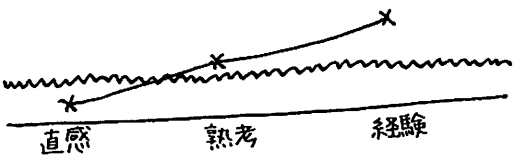
図 18 は、直感だけでも十分であることを示したグラフだ。どんな設計でも大丈夫。思い切って今日から設計しても成功するだろう。

[†1] 訳注：『Code Complete 第 2 版 上』(日経 BP 社) p.144 を参考にして、原文に合わせて調整を加えた。BDUF の訳語については、該当箇所の直前の段落から引用した。



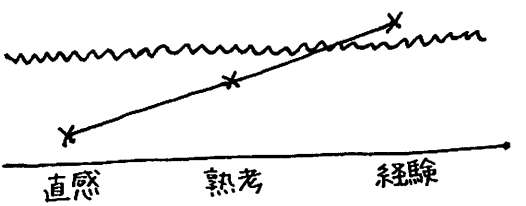
▶図18　どんな設計でも大丈夫

図19は、いつ設計をするかの答えが明確ではない場合を示している。慎重な熟考によってその答えを導き出せるが、経験にもとづいたほうが優れた答えになる。何も考えずに設計すると失敗するので、何も設計しないという選択肢はない。この場合、現時点で設計に大きく投資すべきだろうか？ それとも経験が得られるまで待つべきだろうか？



▶図19　設計の熟考や経験が必要

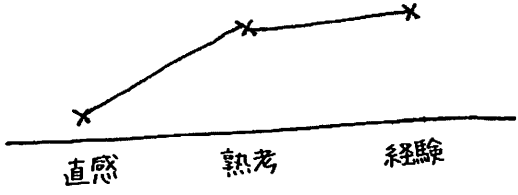
図20は、インクリメンタルな設計が必然的な場合だ。経験がないままいくら熟考しても、不十分な設計になってしまう。経験によってのみ、優れた設計を生み出すための十分な理解が得られる。



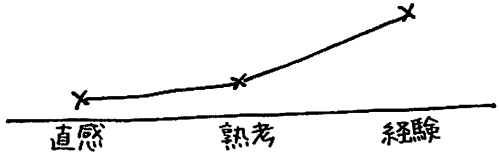
▶図20　熟考するだけでは不十分

いつ設計するかを決めるときに考慮に入れるべき要因のひとつは、それぞれの戦略によってもたらされるバリューである。熟考によってフィードバックがなくても大きなバリューを生み出せるのであれば（図21）、すぐに設計するほうが理に適っ

ている。経験が大きなバリューを生み出しているのであれば（図 22）、現時点では必要な分だけ設計をして、経験を踏まえてからあとで本格的に設計したほうがいい。



▶ 図 21　経験が役に立たない



▶ 図 22　経験から多くを学ぶ

考慮に入れるべきもうひとつの要因はコストである。早めに設計すれば、設計の初期コストはそれに費やした時間だけである。間違いが多ければ、プロジェクト期間中に修正や対応が必要になるため、総コストはもっと高くなるだろう。経験にもとづいた設計の場合は、最小限の初期設計の時間に加えて（事前設計よりもコストは低い）、稼働中のコードや実データにあとから設計判断を組み込むコストが必要になる。XP の多くのプラクティスは、進行中の設計コストを低下させるためのものである。

設計の問題で特に重要なのは、データベースの設計だ。ThoughtWorks 社のピラモド・サダラージが、以下のようなインクリメンタルなデータベース設計の洗練された戦略を考え出している。

- ✧ 空のデータベースから始める。
- ✧ 自動スクリプトでテーブルやカラムを追加する。必要に応じて既存のデータを移行する。
- ✧ スクリプトには連番をつけておく。スクリプトを実行することにより、後期のデータベースに追従することができる。

特定のバージョンのコードは、特定のバージョンのデータベース設計を前提とし

ている。新しいバージョンのシステムをデプロイするには、新しいコードを公開して、データベースの設計とデータ移行のスクリプトを実行する。停止時間を短くするために、デプロイは小さくこまめに行うべきである。

私が知る最も強力な設計方法は「Once and Only Once（一度、ただ一度）」である。これは、データ、構造、ロジックなどは、システムのひとつの場所に存在すべきであるというものだ。重複を見つけたら、設計を改善する必要があることがわかる。重複した表現をひとつにまとめる方法を思いつくまで、設計の改善に取り組んでいく。こうした設計の改善が共通してたどり着く先が、パターンだ。パターンとは、重複をほとんど必要としないコードの構造のことである。重複は悪だ。ひとつの概念を変更したときに、複数の場所にあるコードを変更する必要があるからだ。重複のないコードには、ひとつの概念を変更したときに、ひとつのコードだけを変更すればいいという特性がある。コードにこのような特性が備わっていれば、継続的に変更を加えても、コストを抑えることができる。

インクリメンタルな設計に対する反論は、「できない」と「するつもりはない」のいずれかに当てはまる。「できない」については、稼働中のコードや実データを使った設計に必要なスキルを学べば解決できる。「するつもりはない」については、少し難しい。設計の改善がいくつも必要だと気づいていながら、時間をかけて実現することができなかったシステムに私もかかわったことがある。ほとんどの場合、システムにフィーチャーを追加するという問題に集中しているため、設計を改善するまでには至らなかった。チーム全体の生産性と今日のタスクのバランスをとろうとしなかったのだ。適切ではないところにフィーチャーを押し込むことが、どれだけ気分が悪いことかを考えていなかったのだ。本当に優れた仕事をしたときのモチベーションや満足度が、どれほどのものかを考えていなかったのだ。

大きな泥だんご（*big ball of mud*）[†2]に直面しても、設計の改善を始めることはできる。まずは、自分が歩いているところをランプで照らしてみよう。コードを修正しながら、少しずつきれいにしていこう。設計の改善をやりすぎてしまう誘惑に打ち勝とう。現在に影響を与えている設計を改善する習慣を身に付けよう。時間をかけて取り組む改善の公開リストを作ろう。いつも変更しているコードの設計がよくなったことがすぐにわかるはずだ。システムのなかで手を入れていない部分に触れたときは、古い方法で設計されていることに驚くだろう。

設計の改善を1週間で終わらせながら、引き続き新機能をデリバリーすることができないこともあるだろう。インクリメンタルな設計には、設計の改善の実施

[†2] 訳注：でたらめに構築されたソフトウェアシステムのこと。設計のアンチパターンのひとつ。http://en.wikipedia.org/wiki/Big_ball_of_mud

方法を考えることも含まれている。本章の冒頭にあるリモートプロシージャからCORBAに移行する例で考えてみよう。最初からどれだけ慎重に設計しても、リモートプロシージャコールを使用する前提が、通信とは無関係のコードに影響してしまうことがある。既存のコードに触れるときには、こうした前提をできるだけ局所化しよう。最終的には、新しい通信プロトコルに置き換える仕事が1週間以内に終わるようになるはずだ。

このように時間をかけて変更していくと、開発を中断して一気に変更するよりもコストがかかると思うかもしれない。XPにおいては、ソフトウェアの設計はそれ自体では完結しない。設計とは、技術側の人間とビジネス側の人間の信頼関係を築くためのものである。要求された機能を毎週デリバリーすることが、信頼関係の構築には欠かせない。理論的に最善の設計方法が何であるかは重要ではない。チームのなかでバリューを生み出す多様な関係性を維持することに比べたら、設計者の利便性は優先順位が低いのである。

まとめると、XPスタイルの設計に移行するには、設計の意思決定の時期を移動すればいい。経験を踏まえて決定することができ、その決定をすぐに利用できるようになるまで、設計を遅らせるのだ。チームは、以下のことが可能となる。

- ✧ より早くソフトウェアをデプロイできる。
- ✧ 確信を持って意思決定できる。
- ✧ 誤った意思決定を無理に許容する必要がない。
- ✧ 当初の設計の前提が変わっても開発ペースを維持できる。

この戦略の代償は、バリューのある新機能の流れを維持するために、プロジェクトが続く限り継続的に設計に投資を行い、大きな変更を小さなステップで行うという規律が求められることだ。

シンプリシティ

XPチームはできるだけシンプルな解決策を好む。設計のシンプリシティを評価する4つの基準を紹介しよう。

1. **対象者に適している。**設計がいかに見事で洗練されているかは重要ではない。その設計を使うべき人たちが理解できなければ、それはシンプルではない。
2. **情報が伝わりやすい。**伝えるべきすべてのアイデアがシステムに表現されている。システムの要素は、用語集の単語と同じように、未来の読者に情報を

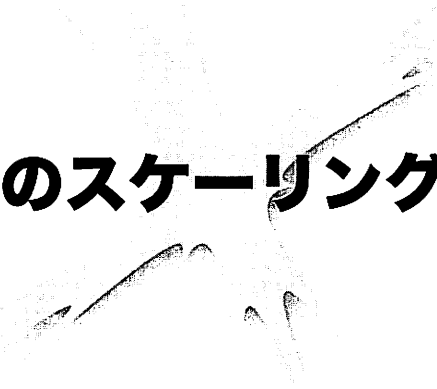
伝えるものである。

3. **うまく分割されている。**ロジックや構造の重複は、コードの理解や修正を困難にする。
4. **最小限である。**上記の 3 つの制約を守った上で、システムの要素はできるだけ少なくする。要素が少なければ、その分だけ必要なテスト、ドキュメント、コミュニケーションが少なくなる。

プロジェクトがシンプルな方向に進んでいけば、ソフトウェア開発の人間性と生産性の両方が改善されるだろう。

第15章

XPのスケーリング



XPがどのようにスケールするかをよく質問される。100人が週一の会議で仕事を詳細に計画することはできない。だが、100人でもコミュニケーション、フィードバック、シンプリシティ、勇気、リスペクトの精神で一緒に働くことはできる。12人のコミュニティーよりも、100人のコミュニティーを作ったり維持したりするほうがはるかに難しいが、それでも常に行われていることだ。

ソフトウェア開発においては、プロジェクトの人数だけがスケールを決める尺度ではない。ソフトウェア開発は以下のさまざまな要素でスケールする。

- ✧ 人数
- ✧ 投資
- ✧ 組織の規模
- ✧ 期間
- ✧ 問題の複雑さ
- ✧ 解決策の複雑さ
- ✧ 失敗の重大さ

人数

スケーリングについて話をするときには、ほとんどの人がこの要素のことを考えているだろう。私がこれまでに訪問した中小企業は、いずれも昔の懐かしい思い出を持っていた。以前ならすぐに解決していた問題が、ある時点からプログラマーが

2人だったときの方法ではうまく解決しないことに気づくのである。以前の解決策は「スケールしなかった」のだ。確かに開発者が2人だったときと同じように50人の開発者が振る舞えるはずはないのだが、よくある厳格なコントロールと書類の承認だけが解決策というわけではない。

私が大きな問題に直面したときは、以下の3つのステップを使っている。

1. 問題を小さな問題に分割する。
2. 簡単な解決策を適用する。
3. 問題が残っていたら、複雑な解決策を適用する。

大きなチームが明らかに必要だとしても、このステップに従えば小さなチームでも解決できるだろう。開発者が50人から300人まで増えて、その後も2,000人まで増やそうとしている組織を見たことがある。成長するたびに問題は増えていき、全体的なスループットは失われていった。従業員をとにかく増やそうとする組織は、最初の50人のほうが問題をうまく解決できるとは考えたくないようだ。

小さなチームでうまくいかない場合は、プログラミングの大きな問題を小さな問題に分割し、小さなチームでも解決できるようにする。まずは、問題の小さな部分を小さなチームで解決しよう。それから、システムを自然な切れ目にそって分割し、複数のチームで作業を始めよう。分割したものを統合するときにうまく合わないリスクがあるので、できるだけこまめに統合するようにして、チームごとに異なる前提を調整しよう。これは分割統治戦略ではなく、統治分割戦略である。次の章で取り上げるSabre Airline Solutions社は、この戦略を広範囲に利用している。

統治分割の目的は、複数のチームをそれぞれ単独であるかのように管理して、調整コストを制限することだ。それでもシステムはこまめに統合する必要がある。チームが単独であるかのように錯覚すると、例外が発生する可能性もあるが、それは例外として処理すればいい。例外が日常化してチームの調整時間が増えた場合は、システムを調整してチームを単独に戻す方法を考えよう。これが失敗したときにだけ、大規模プロジェクトマネジメントを適用すればいい。

まとめると、大きなチームが必要に見えても、小さなチームで問題を解決できないかを確認してみよう。うまくいきそうになければ、小さなチームでプロジェクトを開始してから、自律的な複数のチームで作業を分担しよう。

投資

XP プロジェクトにおける大きな投資の会計処理についてよく質問される。たとえば、XP スタイルの開発は費用なのか、それとも資本投資なのかを何度も聞かれる。ソフトウェア開発を費用で処理している企業は、デプロイ後のプログラムの継続的な保守作業として XP を正当化できる。ソフトウェア開発を資本投資で処理している企業は、プロジェクトの正確なスコープを事前に詳細に決めることはできないにしても、特定の問題を対象とした開発費として**四半期サイクル**や年間サイクルのなかで承認することができる。

問題は、XP そのものにあるのではなく、ソフトウェア開発の会計処理にある。工場や製品の世界の会計モデルを何も考えずにソフトウェア開発のような活動に当てはめると、必然的に会計に歪みが生じてしまう。会計とソフトウェア開発の相互利益となる関係について再考することは、XP のスコープ外にある興味深い課題だ。

XP スタイルのソフトウェア開発を大規模にするのであれば、早い段階で財務面の協力者を見つけておこう。上記の問題をうまく切り抜ける支援をしてくれるはずだ。ソフトウェアの会計処理は企業によって少しずつ違っている。

組織の規模

組織の大部分が変わらない場合に、XP を組織の一部に適用するにはどうすればいいだろうか？ チームは、より多くの、より正確な情報をすばやく作り始めるべきだが、その情報を気が進まない人たちに押し付けると、チームに必要な支持者たちが敵になってしまう。ここでの目的は、チームの新しい取り組みを隠すことでもなければ、他人を強制的に変えることでもない。組織が慣れている形でコミュニケーションを維持するようにしよう。

ここでは、スキルの高いプロジェクトマネージャーが XP チームの役に立つ。大きな月例会議で規定のフォーマットのスライドが求められているのであれば、XP プロジェクトマネージャーがきちんとそれを用意する。プロジェクトマネージャーは、組織が受け入れられる形で情報を提示するのである。それでも壁に貼られたストーリーカードが「真実」であることに変わりはない。読み方を知りたければ、誰もが自由に来て、見て、質問できる。プロジェクトマネージャーは、組織の期待に応えなければいけない。XP チームの状況は他のチームとは大きく異なるため、これは大変な作業になるだろう。組織の人たちをリスペクトしよう。新しく得た知識や力を自分の利益のために他人に押し付けてはいけない。

組織の期待に応えると同時に、XP のよいところを維持するには、創造力が必要に

なることもある。私のあるクライアントのところでは、プロジェクトの詳細な四半期計画が必要とされていた。それは、XPや週単位のスコープ交渉とは相容れないように思われた。だが、上司の上司がとても頭のいい人で、四半期計画の目的を把握し、それがいつ読まれるのかを特定したのである。それは、四半期の終わりに開催されるエグゼクティブレビューのときだった。チームが責任を持って行動したかどうかを実績と比較して、確認するための計画だったのだ。

そのプロジェクトマネージャーは、毎週金曜日にXPチームのところにやって来て、チームメンバーにその週にやったことを聞くようにした。そして、情報を四半期計画の書式に記入した。その結果、そのチームの計画には組織のなかで最も正確な見積りが含まれていることが、四半期の終わりのレビューで認められた。

チームのなかにウソをつこうとする人はいなかった。すべてが正々堂々と行われた。チームで何が起きているかをすべてのマネジメント層が把握していた。このチームは、正しい書式によって組織が期待する字義を満たし、責任のある時間の使い方によって四半期計画の精神を満たしたのである[†1]。

期間

長期間のXPプロジェクトはうまくいく。テストのおかげで、よくある保守の間違いを防げるし、開発プロセスのストーリーがうまく伝わるからだ。期間のスケーリングで最もシンプルな課題は、チームがプロジェクトで継続性を維持できるかどうかである。自動テストやインクリメンタルな設計のおかげで、システムを維持したまま、さらに成長させることが可能である。10年前に私がこれらの技法を使ってコーチしたチームは、それ以来ずっと安定して機能を追加している。欠陥はほとんどない。進捗は地味だが安定している。小さなチームだとしても、大規模で洗練されたシステムを10年間で育てるのに劇的な進捗は必要ない。

頻繁に開始と停止を繰り返すプロジェクトがあり、停止するたびにチームがバラバラになっていると、時間をかけてプロジェクトを維持することが難しい。このような場合は、プロジェクトが停止する前に、XPチームで「ロゼッタストーン」を書くことが多い。これは、将来の保守のための簡単なガイドであり、ビルドとテストのプロセスを実行する方法や、システムについて学習しやすい出発点を記述したものである。ビルドにはテストが含まれているため、保守担当者がシステムを学習するときに落とし穴にハマることがない。

---

†1 訳注：法律や条約などを指すときに使う言葉。文面を指す「字義（*letter*）」とその意味を指す「精神（*spirit*）」で対になっている。

問題の複雑さ

XP は、専門家同士の密な連携が必要とされるプロジェクトに最適だ。そのようなプロジェクトで最初に課題となるのは、お互いの専門分野を少しずつ学びながら、全員を協力させることである。たとえば、私は生命保険のプロジェクトに参加したことがある。保険数学の学習を始めたときには、アクチュアリー（保険数理士）が辛抱強くペアになってくれた。1 か月後、私はわずかな間違いを発見できるようになった。数か月後、役に立てることが増えてきた。その後、私はアクチュアリーになったわけではない。それでも、私とアクチュアリーが別々に、それぞれユーザーインターフェイスとシステムの特定の機能に取り組むよりも、はるかに強力なシステム（とチーム）ができたと思う。

解決策の複雑さ

システムが成長して複雑になり、解決する問題に対して不釣り合いになることがある。ここでの挑戦は、問題をさらに悪化させないことだ。欠陥を修正するたびに新しく 3 つの欠陥が生み出されるような状況では、そのままチームで対応を続けるのは難しい。そこで、XP である。

あるクライアントは、ビルドプロセスを制御するところから手を付けた。チームでビルドを改善した結果、専用マシンで 24 時間かけてマニュアル片手にやっていたものが、どのマシンでも 1 時間で自動的にビルドが完了するようになった。その後、ストーリーとストーリーボードを作成して、誰が何をどのくらい時間をかけてやっているかを誰でもわかるようにした。チームが着実に改善を進めた結果、2 年間でコスト（エンジニアの人数）が 70 人から 20 人に 60% 削減、欠陥の修正時間が 66% 短縮、メジャーおよびマイナーリリースにかかる時間が 10 週間から 2 週間に 75% 短縮された。墓穴を掘る状況を脱してからは、余計な複雑さを排除しながら欠陥を修正することで、チームは這い上がり始めたのである。

余計な複雑さを扱うための XP の戦略は常に同じだ。継続的にデリバリーしながら、少しずつ複雑さを削り取っていくのである。自分のいる一隅を光で照らそう。欠陥を修正するときは、その部分をきれいにしよう。この「追加」の作業に時間がかかりすぎるという反論もある。だが、欠陥の修正のために作業を中断されて、チームはすでに時間をムダにしているのだ。きれいにすることで作業の負荷も軽減される。計画づくりを見える化すれば、時間がかかっている場所を全員が把握できるので、適切に作業を行うために必要な見積りが受け入れられやすくなる。

失敗の重大さ

安全性やセキュリティが重要なプロジェクトでは、XPをどのように使えばいいのだろうか？ 安全性やセキュリティが最も重要になるため、いくつかのルールを変更することになるだろう。病院のソフトウェアを担当しているチームは「我々が間違いを犯せば、赤ちゃんが死んでしまいます」と言っていた。人命にかかわるような状況では、XPだけでは不十分である。

セキュリティの原則を守って構築し、専門家によって審査を受けるまでは、システムの安全性は保証されない。XPとうまく共存させながら、上記のプラクティスをチームの日常業務に組み込む必要がある。たとえば、リファクタリングはシステムの振る舞いだけでなく、セキュリティについても保たなければいけない[†2]。

航空システムや医療システムでは、デプロイの許可を得る前に審査を受ける必要がある。XPの流れの原則からすると、審査がプロジェクトの最後のフェーズになってはいけない。審査は早めにこまめに行うべきである。DO-178B[†3]などの審査要綱では、厳格なウォーターフォール以外のソフトウェア開発ライフサイクルが明確に認められている。以下は、FDA（アメリカ食品医薬品局）の「FDA審査官および業界向けのガイダンス：医療機器に含まれるソフトウェアのための市販前申請内容に関するガイダンス[†4]」から抜粋したものだ。

> ライフサイクルモデルには、ウォーターフォール、スパイラル、進化型、インクリメンタル、トップダウン機能分割（あるいは段階的詳細化）、形式的変換などさまざまなものがある。選択したモデルに適切なリスク管理活動とフィードバックプロセスを組み込めば、いずれかのモデルもしくはその他のモデルを用いて、医療機器に含まれるソフトウェアを作ることができる。

審査官と継続的な関係を築けば、審査の成功率が高まる。

XPにはトレーサビリティ（システムの変更箇所と変更理由を結び付ける能力）が組み込まれている。ただし、情報を記録する手順が決まっているわけではない。トレーサビリティを実現するには、こうした情報を物理的に記録するだけでいい。たとえば、「**この**コード行を変更しているのは、**あの**テストを書いたからだ。これは**あの**システムレベルのテストの一部である。それは、5/24に計画を立てて、5/28に

[†2] 訳注：リファクタリング（名詞）の定義は「外部からの見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変化させること」。

[†3] 訳注：航空システムと機器認証でのソフトウェアに関する考慮事項。アメリカ連邦航空局から安全性の認証を得るときの標準とされる。

[†4] 訳注：*Guidance for FDA Reviewers and Industry Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices*

デプロイする準備ができていた**あの**ストーリーから来たものだ」という情報があれば、あとは審査官が情報を記録する書式を教えてくれるだろう。

結論

意識してきちんと適応すれば、XP はスケールする。問題をシンプルにすれば、小さな XP チームでも簡単に対処できるようになる。シンプルにできない問題については、XP を拡張する必要がある。基本的な価値や原則は、あらゆるスケールに適用できる。プラクティスは状況に合わせて修正すればいい。

第16章

インタビュー

これから紹介するのは、Sabre Airline Solutions 社の航空製品開発シニアバイスプレジデントであるブラッド・ジェンセンのインタビューだ。

Q： XP を使い始めた理由を教えてください。
A： 13 あったプロダクトチームをひとつの組織にまとめて、アーキテクチャやルックアンドフィールを統一する仕事のためでした。XP は私の戦略の一環です。チームには「プロセスは XP だ！」と言いました。

Q： 組織の人数は？
A： 航空製品部門には 300 人が所属しています。内訳は、開発者が 240 人、マネジメントと事務職が 25 人、テストと構成管理の担当者が 35 人です。

Q： どのように XP を導入されましたか？
A： 13 のプロダクトグループに Object Mentor 社の 1 週間のトレーニングを受けてもらいました。13 グループありますから、全部で 13 週間かかりました。XP を使い始めてからは、チームにコーチをつけました。

今やるとしたら、まずはひとつのチームで始めて、XP が本当に「自分のもの」になったことを確認してから、そのチームに次のチームを指導させると思います。

Q： XP のすべてを使っていますか？
A： 状況によって目盛りを調整しています。XP に適したプロジェクトの場合は、すべてを使っています。XP に最適なプロジェクトは、意欲的なチームがいる Java の新規開発です。C++ のレガシーコードを拡張するときは、ウォーターフォール

型のプロジェクトにXPを組み込む必要があります。その場合は、要件定義や設計の作業を事前に行うことになります。正式なテストフェーズも用意して、それが終わったらデプロイして顧客に届けます。

レガシーなプロジェクトでは、設計をシンプルに保つことができません。そもそも最初からシンプルではないからです。設計が複雑になっているため、テストは十分に書かれていませんし、それによって多くのバグが存在します。C++にはツールが不足しているため、リファクタリングも困難です。プロジェクトの開始時と終了時にウォーターフォールのフェーズが必要なのはそのためです。

Q： XPの利点は何でしょうか？
A： 純粋なXPプロジェクトでは、欠陥がほとんど見られません。2年間の運用後、欠陥がひとつも見つからなかったプロジェクトもあります。レガシーのXPプロジェクトの場合は、コード1,000行あたり1〜2個です。こちらも他と比べて、欠陥率は低いといえます。たとえば、CMM（能力成熟度モデル）レベル5を取得した10の組織で構成されるBangalore SPINは、コード1,000行あたりの欠陥数は平均8個であると報告しています。それから、生産性も向上しました。XPの導入前後で比較したところ、生産性が40%も向上したプロジェクトもあります。

Q： XPの導入は簡単ではなかったと思います。
A： そうですね。最初は、3分の1の人が懐疑的です。その他の3分の1はすぐに取り入れてくれて、残りの3分の1は様子を見ています。最終的には、80〜90%が取り入れてくれます。10〜20%は、嫌々XPを使います。3〜5%は、絶対に取り入れてくれません。ペアを組もうとしなかったり、コードの所有権を主張したりするプログラマーがいたら、解雇する勇気を持つべきです。残ったチームがあなたを助けてくれるはずですから。

Q： オンサイト顧客[†1]について教えてください。
A： 多くのことが顧客次第だと思います。我々の顧客は、会社のプロダクトマネージャーたちです。彼らにはチームと同じオープンスペースに同席してもらっています。プロダクトによっては、100社の航空会社を対象にしています。すべての航空会社の利益を代表するのが、プロダクトマネージャーの仕事です。プロダクトマネージャーは、ユーザー会議、顧客フォーラム、デザイン協議会で、要件を取りまとめます。特にデザイン協議会は重要です。年に一度、プロダクトの上顧客を何名かご招待して、今後のリリースに望まれているものを教えていただくのです。1回

[†1] 訳注：第1版で登場したXPのプラクティスのひとつ。第2版では「本物の顧客参加」に相当する。

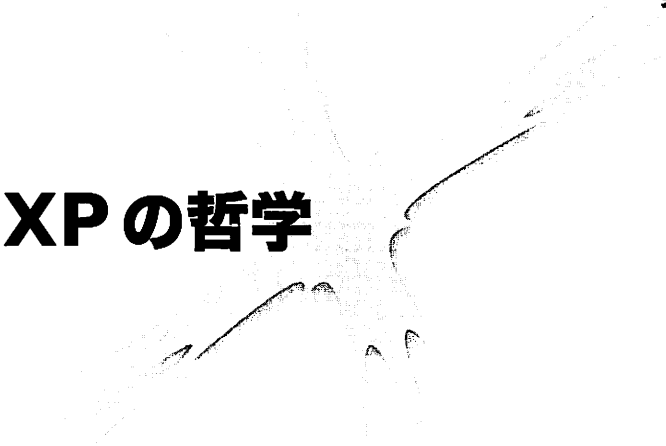
のデザイン協議会で、1〜2 年分のストーリーが手に入ります。シニア開発者も同席しますが、フィーチャーの最終決定者はプロダクトマネージャーです。

オンサイト顧客は XP で最も重要な部分ですが、多くの問題が発生する部分でもあります。スコープの管理ができるのは素晴らしいことです。スケジュールの 4 分の 1 が過ぎた時点で、完成できるかどうかがわかるのも素晴らしいです。ですが、気を付けておかないと、スコープを管理できずにスコープクリープが発生してしまいます。

顧客のなかには素晴らしい方々もいます。彼らはストーリーを書いてくれます。受け入れテストの条件を書いてくれます。テスターが受け入れテストを書くのを手伝ってくれます。ですが、なかにはそうではない顧客もいます。彼らは抽象的なストーリーは書いてくれるのですが、受け入れテストの条件を書くことに興味を持ってくれません。その場合は、ドメイン知識のある経験豊富な開発者に書いてもらいます。見積りを低く抑えるために要件を隠しておいて、あとからフィーチャーのすべてを要求するという、顧客の立場を悪用した事例も何件かありました。

Q： XP の導入を検討している経営幹部にアドバイスをお願いします。
A： ぜひ、XP を使ってください。そして、フィーチャー単位で計画してください。計画の項目は、顧客が関心のあるフィーチャーでなくてはいけません。四半期ごとにリリースを計画してください。イテレーションはそれよりもっとこまめに計画しましょう。我々は 2 週間のイテレーションにしています。イテレーションの期間は絶対に固定しましょう。顧客をチームと同席させてください。チームをオープンスペースで働かせてください。XP をウォーターフォールスタイルの開発に組み込む必要があるとしても、それによってさまざまな効果が得られるでしょう。

第II部



XPの哲学

ここまでは、XPの適用方法について見てきた。これからは、XPにつながるその他の領域のアイデアについて、私がどのように考えているかを紹介する。XPに直接影響を与えたアイデアもあれば、XPとの類似性を明確にするアイデアもある。

このようなアイデアをひとつでも学べば、XPのことをすばやく理解して、部品がどのように組み合わさっているかがわかるようになる。以前、非常に懐疑的なCFOのいるグループにXPについて説明したことがある。午後になると、彼にひらめきの光が射し込んだ。

「これはリーン生産方式をソフトウェアに適用したものですね！ 前職でリーン生産方式への移行を経験したことがあります。これならあなたの言っていることがよくわかります」

他の分野との類似性は慎重に利用すべきだろう。製造業の作業現場でうまくいったことが、プログラミングの開発現場ではうまくいかないこともある。逆もまた然りだ。だが、その他の分野では、組織変革などの人間の難しい問題に何十年も取り組んでいる。こうした教訓を学べば、XPの適用を加速できる。

第17章

はじまりの物語

広まったアイデアには、それにまつわる「はじまりの物語」がある。こうした物語はアイデアを定着させ、聞き手が理解しやすい文脈に持ち込んでくれる。それでは、XPのはじまりの物語の話をしよう。

すべては1本の電話から始まった。それは、クライスラー社が開発した給与計算システムのパフォーマンスを見てくれないか？ という内容だった。私はSmalltalkのパフォーマンスチューニングに関する執筆や講演をしていたので、そのような電話があっても驚きはしなかった。だが、予備調査の質問に返ってきた答えには、少しだけ驚かされた。なかでも以下の答えが印象的だった。

「変更後に何も壊れていないことを確認するテストはありますか？」

「それどころか、まだ正しい答えの計算ができていないのです」

正しい答えを計算する必要がなければ、いくらでもシステムを速くできる。それまでのコンサルティングの経験から、単なるパフォーマンスチューニングよりも大きな問題の臭いがした。私は好奇心から現場に行ってみることにした。

到着したのは1996年の「懺悔の火曜日[†1]」だった（イースターの時期である）。チームと一緒にポンチキを食べたので、日付を正確に覚えている。ポンチキとは、「懺悔の火曜日」にだけ作られるポーランドのジャムドーナツだ。

プロジェクトがトラブル状態になっていることがすぐにわかった。

- ✧ 本番稼働まで2か月の状態が、5か月間続いていた。
- ✧ 私の質問に答えるときに、誰が聞いているかを確認していた。これはチーム

[†1] 訳注：1996年は2月20日。イースターの47日前（四旬節の前日）に懺悔をしていたことに由来する。

メイトから自分の身を守っている証拠だ。

✧　誰もが疲れていて不機嫌だった。

コンサルティングではよくあることだが、クライアントはすでに答えを知っていた。コンサルタントとしての私の仕事は、その情報を持っている人を見つけて、それを権限を持つ人に伝えることである。初日の朝にコーヒーを飲みながら、誰かがこんなことを言った。

「ハードディスクを消去しちゃえば、こんなプロジェクトは終わるのになぁ」

デプロイまで2か月しかないというのに、すべてのコードを捨て去るなんて。誰もがそんな発想に苦笑いした。

2日後、クライスラー社のCIOが出席する会議に参加した。私は彼女に3つの選択肢を提示した。いつデプロイできるかわからないまま現状を維持するか、プロジェクトを中止してすべての専門知識を失うか、小さなチームで最初からやり直すかである。彼女は、私をチームに入れて最初からやり直すことを選択した。

大きな変化を成功させるには、多くの要素が必要になる。運もそのひとつだ。家に帰ると、ロン・ジェフリーズからメールが届いていた。

「仕事を探しているんだ。810.555.1234まで電話してくれ」

彼に電話して「810ってデトロイトの市外局番でしたよね」と聞いた。

「いや、アナーバーだ。十分近いがね」

こうしてロンは、最初のフルタイムのXPコーチになった。

マーティン・ファウラーは、このプロジェクトで分析に関するコンサルティングを担当していた。我々が新しいソフトウェア開発のアイデアを試しているときでも、彼は冷静さを保っていた。私はそのことを知っていたので、彼に分析やテストを無理やり手伝ってもらうことにした。最終的には、マネジメントから支持を得ることができた。最初は違和感を感じるであろう開発スタイルにチームも付き合ってくれることになった。

2週間後、私はプロジェクトのキックオフのために戻ってきた。まずは（縮小した）チームのメンバーにインタビューすることにした。彼らが何に貢献できると思っているかを把握したかったのだ。まずは、プロジェクトマネージャーのボブ・コウからインタビューを始めた。雑談が終わると、彼のほうから質問してきた。

「それで、このプロジェクトをどのように進めるんですか？」

聞かれるんじゃないかと思っていたが、聞かれたくない質問だった。どう答えればいいか、自分でも正確にわかっていなかったからだ。

「ええと……3週間の……あー……イテレーションがあります。イテレーションでは、いくつかの……えー……ストーリーを実装します。マリー（給与計算の専門

家）がそれを選びます。我々がストーリーを見積もり、イテレーションにいくつのストーリーが収まるかを計算します。イテレーションを合計すれば、全体のスケジュールの長さがわかります」

「了解しました」

次のインタビューでは、こう言った。

「3 週間のイテレーションがあります。イテレーションには、いくつかのストーリーが含まれます。実際の給与が正しく印字されるように、えー……あー……最初のイテレーションで、ストーリーを完成させるようにします」

このような説明を 1 日かけて繰り返し、そのたびに少しずつプロジェクトの体制に慣れ、そのたびに少しずつ詳細を加えていった。

このプロジェクトのスタイルを整えるときに考えていたのは、私がソフトウェアエンジニアリングで重要だと思うすべてのことを取り入れ、その目盛りを 10 にすることだった。必要不可欠なことはすべて考えられる限り熱心に行い、それ以外はすべて無視する。他に追加するものがあれば、あとで追加することにした。

その日の終わりには、XP の基礎を整えた。システムは常にデプロイ可能であり、一定の間隔で、顧客が選んだフィーチャーを追加して、自動でテストを行う。だが、私が目盛りの 10 だと思っていたものは、実際には 8、あるいは 6 だった。それは、その後の経験でわかったことだ。

私が「基礎」を整えたと言ったのは、こうした抽象的な概念を実際のプロジェクトに適用したのは、チームだったからである。そこから作業の手順を決め、ツールを作成し、スキルを身に付け、人間関係を築いて深めていく必要があった。私は、ソフトウェアの新しい開発方法のビジョンを提示しただけだ。彼らがその方法を使い、もっとうまくソフトウェアを開発したのである。

我々の最初のタスクは、プロジェクト完了までの期間を見積もることだった。チームはストーリーを作成して、それぞれを見積もった。1 日がかりの計画会議を開き、すべてのストーリーをレビューした。そして、最初にデプロイする最小限の機能を顧客に選択してもらった。それから、見積りを合計して、IT 部門と支援組織のトップマネージャーに日付を伝えた。その日の終わりに、ひとりのプログラマーが私にこう言った。

「はじめて本当に信じられる見積りでした」

その後、チームは 3 週間ごとにストーリーの実装を始めた。

チームは余裕を見せていた。私も自信過剰になっていた。スコープを交渉すれば、常に予定どおりにデリバリーできるものだと思っていた。だが、ソフトウェア開発に保証はついていない。なかでも忘れられない出来事がある。システムを

フィーチャーに分割して、顧客にフィーチャーをコントロールしてもらえば、ソフトウェアは常に予定どおりにデプロイできるものだと完全に確信していた。確実に期日（1997年1月）までにシステムを本番稼働できるはずだった。だが、11月の中旬に間に合わないことが明らかになった。給与の計算はうまくできるようになっていたが、出力ファイルに数字が正しく保存されていなかった。テストが数字の計算処理だけを見ていたのである。本来ならば、その結果を正しく出力するところまでテストするべきだった（この惨事から「エンドツーエンドは思っているよりもはるかに遠い[†2]」という格言が生まれた）。

この状況にうまく対応できたと言いたいところだが、対応できなかった。私はパニック状態だった。残作業すべてを時間内に終わらせるようにチームに命令しようとした。チェット・ヘンドリクソンが、ゆっくりとした口調で私を止めた。

「これまで完成日の見積りをするときには、分割して見積もってから、合計していたでしょう。今回と何が違うんですか？」

ボーナスのことも心配だったが、二度とデプロイを遅らせたくなかった。その後、事態は好転した。3月に動くことを確認し、4月に本番稼働した。

システムは3年間稼働して、2000年に役目を終えた。その背景には、技術的な課題と資金的な課題があった。2000年までにオブジェクトテクノロジーの優位性を確立していたのは、Smalltalkではなく Javaだった。クライスラー社は、あまり使われていない言語で書かれたシステムをいつまでも保守する気がなかった。他にも同じ状況のシステムがあり、それが厄介でコストのかかるものだったのだ。このプロジェクトは、オブジェクトテクノロジーを使ったパイロットプロジェクトとして、IT部門から予算が出ていた。後期フェーズの予算も申請されたが、財務部門に却下された。そして、プロジェクトは中止されたのである。

このシステムは、信頼性が高く、コストもかからず、保守や拡張も簡単だった。アーキテクチャは柔軟で適切な状態を保っていた。欠陥率は同様のプロジェクトに比べると非常に低かった。チームは新しいメンバーをすぐに受け入れられるようになっていた。チームメンバーが去る場合も、チームの有効性は大きく損なわれなかった。プロジェクトは最後まで技術的に成功していたのである。

それから、ビジネス的にも成功していた。オブジェクトテクノロジーが大規模なデータ処理プロジェクトに適していることを証明したのである。その後、ビジネスニーズは大きく変わった。ソフトウェア開発の予算は政治的に管理されるようになった。ソフトウェア開発の予算の優先度は、技術的あるいはビジネス的な手法が変わることによって、急速に変化するのである。

[†2] 訳注：End-to-end is further than you think.

これが、私の XP のはじまりの物語だ。それ以来、多くのチームの考えや実践によって、アイデアが洗練されている。それから、XP の価値、原則、プラクティスの関係性の探求も行われている。

第18章

テイラー主義とソフトウェア

フレドリック・テイラーは、世界初の生産技術者（*industrial engineer*）である。それ以前から工場の効率性を研究する人たちはいたが、テイラーは情熱とカリスマ性を携えてこの分野に乗り込み、インダストリアルエンジニアリングという新しい分野を生み出した。彼と弟子たち（特にフランク・ギルブレス、リリアン・ギルブレス、ヘンリー・ガント）の成果により、工場の生産性を体系的に向上させる、厳密で説得力のある論拠が示されたのである。彼は、自分の学説に「科学的管理法」という強力なメタファーを選んだ。

説明的な名前を選ぶときは、反対の意味が魅力的ではないものにするといい。「非科学的」管理法に賛同する人などいるだろうか？ テイラーが「科学」と言ったのは、工場の生産性を高めるために科学的手法を適用しようとしたからだ。つまり、観察、仮説、実験である。彼は、作業している作業者を観察して、作業を遂行する代替方法をいくつか考案して、それらを観察し、最もうまくいくものを選択した（彼のモットーは「唯一最善の方法[†1]」だった）。あとは生産性の向上を保証するために、工場全体で作業を標準化するだけだった。

これが「テイラー主義」と呼ばれるまでに、技術的、社会的、経済的に影響を与えたことについて、すべてを説明している余裕はない。巻末の参考文献を参照してほしい。テイラー主義には好ましい効果もあるが、いくつかの深刻な欠点もある。これらの欠点は、以下の 3 つの単純化された仮定によるものだ。

- ✧ 通常、物事は計画どおりに進む。

[†1] 訳注：The One Best Way

- ✧ 局所最適化は全体最適化につながる。
- ✧ 人はほぼ代替可能であり、何をすべきかを指示する必要がある。

テイラー主義はソフトウェアエンジニアリングにとって重要なのだろうか？ クリップボードとストップウォッチを持って、ソフトウェアの開発現場を歩きまわっている人などいない。ソフトウェア開発で問題となるのは、テイラー主義に伴う仕事の社会構造だ。テイラーが提唱した「時間・動作研究」の儀式や道具こそなくなっているが、我々は無意識にこの社会構造を継承しているのである。その社会構造は、ソフトウェア開発にまったく適していない。

テイラー主義によるソーシャルエンジニアリングの手順その1は、計画立案と実行作業の分離だ。作業方法や作業期間を決定するのは、教育を受けたエンジニアである。作業者は、与えられた作業を、与えられた方法で、与えられた時間内に、忠実に、実行しなければいけない。そうすれば、すべてがうまくいく。だが、作業者は機械の歯車ではない。自分のことを歯車だと思いたい人などいない。

ソフトウェア開発においても、権限のある人が他人の作業の見積りを作成したり、変更したりする。そこにテイラーの影響が見られる。「エリート」のアーキテクチャグループやフレームワークグループが誰かの仕事を指示するときにも、同じくテイラーの影響が見られる。

テイラー主義によるソーシャルエンジニアリングの手順その2は、独立した品質部門の設置だ。テイラーは、作業者はとにかく「怠ける」（目立たない程度に仕事のペースや品質を落とす）と考えていた。彼は、品質管理部門を設置することで、作業者に適度なペースかつ規定の方法で作業させるようにして、適切な品質レベルを保ったのである。

多くのソフトウェア開発組織は、品質部門を別に設置しているという意味で、まぎれもなくテイラー主義である（むしろそれを誇りに思っている）。品質部門を別に設置すれば、エンジニアリングにおける品質の重要性が、マーケティングや営業における品質と同等だというメッセージを送ることになる。エンジニアリングで品質に責任を持つ人がいなくなる。すべては他の誰かの責任だ。開発組織のなかにQA部門を設置したとしても、エンジニアリングと品質保証が別々の並行した活動であるというメッセージを送ることになる。品質とエンジニアリングを組織的に分離すれば、品質部門の仕事は建設的なものではなく、懲罰的なものとなってしまう。

ソフトウェア開発において、品質、アーキテクチャ、フレームワークが重要ではないと言っているわけではない。むしろ逆だ。あまりにも重要なので、テイラー主義の社会構造に委ねるべきではないのである。変化している世界のなかで、実際に動く、柔軟性の高い、安価なソフトウェアを作り出すために必要不可欠なのは、コ

ミュニケーションとフィードバックだ。テイラー主義の社会構造によって、これらの流れが滞ってしまうのである。次の章では、比較的新しい製造業のアイデアを見ていこう。生産性と品質の目標を達成するために、テイラー主義とは異なる社会構造を提供している。

第19章

トヨタ生産方式

トヨタは最も利益性の高い自動車メーカーのひとつである。素晴らしい製品を作り、急速な成長を遂げ、利益性が高く、多くのお金を稼いでいる。トヨタがこのような目標を達成できているのは、ムリをしているからではない。車を製造するプロセスのすべての工程で、ムダな労力を削減しているからだ。ムダを十分に排除すれば、単に速く進もうとするよりも、ずっと速く進むことができる。

本章では、トヨタ生産方式（TPS：Toyota Production System）の一部に注目する。これは、実際の車の生産に使われているものだ。TPSの製品開発に関する部分がソフトウェアによってバリューを届ける作業にどれだけ重要であるかは、メアリー・ポッペンディークとトム・ポッペンディークが詳細に記述している。

従来と異なる仕事の社会構造が、トヨタの成功には絶対不可欠である。すべての作業者が、生産ライン全体に責任を持つ。欠陥を見つけたら、紐を引っ張って、ラインを止める。そして、ラインの全員で問題の根本原因を発見し、それを修正するのである。アメリカ人の作業者には、最初は信じられなかったようだ。チェット・ヘンドリクソンが、ケンタッキー州のトヨタの工場で働いている義理の兄弟の話をしてくれた。欠陥のあるドアが目の前を通り過ぎたときのことだ。同僚が「紐を引っ張れ」と言った。だが、彼はトラブルに巻き込まれたくなかった。再び欠陥のあるドアが目の前を通り過ぎた。それからもうひとつ。ついに彼は紐を引っ張った。その結果、真実を伝え、欠陥を発見したことを褒められた。「ラインの最後」で品質に責任を持つ人がいる大量生産ラインとは異なり、後工程の品質保証が必要ないくらいにラインの品質を保つのがTPSである。このことはつまり、全員が品質に責任を持つことを意味している。

TPSでは、作業者個人が作業のやり方や改善について多くの意見を述べる。ムダは、カイゼン（継続的改良）イベントで削減していく。まずは、作業者がムダ（品質問題や非効率）の源泉を特定する。そして、率先して問題を分析して、実験して、その結果を標準化する。

TPSでは、テイラー主義の工場で見られた厳格な社会的成層が排除されている。生産技術者は、ラインの作業からキャリアを始め、常に工場で膨大な時間を費やしている。日常的なメンテナンスはエリート階級の技術者ではなく、普通の作業者が行っている。独立した品質部門は存在しない。組織全体が品質部門である。

作業者は自分の作業の品質に責任を持っている。そこで作った部品は、ラインの次の工程ですぐに使用されるからだ。作業工程がこのように直接結び付いていることを最初に知ったとき、自分の直感に完全に反していると私は思った。大量生産の工場をスムーズに稼働させるには、工程間に大量の在庫の部品が必要だと考えていたからだ。在庫があれば、たとえ前工程で機械が停止しても、後工程の機械はバッファにある部品を使い続けることができる。

TPSはこうした考えをくつがえした。「仕掛品」の在庫があれば、個々の機械はスムーズに稼働しているかように思える。だが、工場全体として見れば、うまく機能していないのだ。部品をすぐに使えば、部品自体のバリューだけでなく、前工程の機械が正常に稼働しているという情報も受け取ることができる。部品は単なる部品ではなく、部品作りの情報でもある。このような考えは、ラインのすべての機械をスムーズに稼働させ続ける強制力となり、機械をスムーズに稼働させるために必要な情報を提供するものである。

TPSの精神的指導者である大野耐一氏は、最大のムダは「つくりすぎのムダ」だと言っている。何かを作り、それが売れないとなると、作った労力の行き場がない。生産ラインで何か作り、それをすぐに使わないとなると、情報のバリューが消えてしまう。保管コストもかかってしまう。たとえば、倉庫に運搬しなければいけないし、保管中の管理をしなければいけないし、再び取り出したときにサビを磨いて落とさなければいけない。それから、まったく使われないというリスクもある。その場合は、お金を払って撤去しなければいけない。

ソフトウェア開発には「つくりすぎのムダ」が満ちあふれている。すぐに陳腐化する分厚い要求文書、まったく使われない精巧なアーキテクチャ、数か月も放置され、インテグレーション、テスト、本番環境での実行がまったくなされないコード、不適切で誤解を招くようになるまで誰にも読まれないドキュメント。これらの活動は、すべてソフトウェア開発にとって重要なものである。だが、ムダを排除するために必要なフィードバックを手に入れるには、アウトプットをすぐに利用する必要

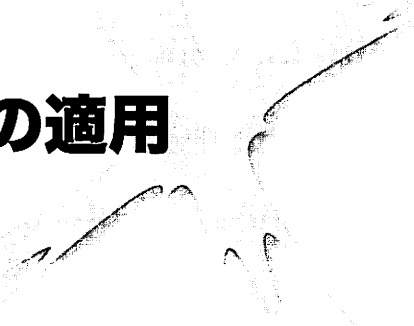
がある。

たとえば、要件収集を改善するには、要件収集のプロセスを念入りに行うのではなく、詳細な要件の作成とソフトウェアのデプロイの間隔を短縮すればいい。詳細な要件をすぐに利用すれば、要件収集は静的なドキュメントを作成するフェーズではなく、開発で必要になった詳細な情報を作り出す活動になる。

TPS の多くの側面は、ソフトウェア開発との強い類似性がある。たとえば、作業者の多能工化、工場のセル化、顧客とサプライヤー間の利益分配契約の作成などは、いずれも役に立つ考え方だ。興味があれば、まずは大野耐一氏の著書『トヨタ生産方式』（ダイヤモンド社）を読んでみるといいだろう。

第20章

XPの適用



5年前、自分がプログラミングの仕事をうまくやれば、みんなが気に入ってくれて、私を見習ってくれるだろうと思っていた。だが、XPの適用はそんなに簡単に割り切れるものではなかった。XPは、複雑な社会的状況で実施するものであり、技法を適用するだけでは、組織どころかプロジェクトさえも管理できない。

XPを適用してから、劇的な結果が見られるまでには時間がかかる。数週間ではなく、数年単位である。最初の数週間や数か月は大きな改善が見られるだろうが、こうした改善は将来の大きな飛躍のための準備段階にすぎない。ソフトウェア開発にはムダが多い。これらのムダの根幹は、我々が行っていることよりも、信じていることや感じていることにある。自分たちの信じていることや感じていることに気づき、それらに対応するには、相当な時間と経験が必要だ。

ソフトウェア開発のスタイルを「採用する」という言葉には、誤った意味が含まれている。ソフトウェア開発のスタイルを取り入れることで、既存の問題が隠れたり、排除されたりすることはない。これまでのあなたの問題は、これからもあなたの問題である。XPを使えば、問題の解決に取り組むための新しい文脈が手に入る。XPは問題を解決するものではない。問題は、自分のやり方で、自分の都合に合わせて、XPであれ何であれ、自分が使っているプロセスの文脈のなかで、あなた自身が解決するものだ。数年以内に実現したい開発方法のビジョンを持っているかもしれないが、そこにたどり着くまでは、自分のスタイルで開発することになるだろう。あなたのスタイルはXPなどのビジョンに大きく影響を受けるかもしれないが、それでもそれはあなた自身のスタイルだ。

XPを採用するのは、子どもを養子にする[†1]というよりも、プールに入るようなものである。プールに入るには多くの方法がある。つま先だけ入れることもできるし、端に座って足を入れることもできるし、階段を下りて水に浸かることもできるし、競泳のように力強く滑らかに飛び込むこともできるし、膝を抱えて勢いよく飛び込んで、大きな音を立てながら周囲の人たちに水をかけることもできる。水に入る正しい方法はひとつではない。

チームがXPの適用を始めると、古いやり方に戻ってしまう危険性が常に存在する。知識のあるプログラマーは、失敗するテストを先に書かずにコードを変更してしまう。知識のあるマネージャーは、明確で正直なコミュニケーションの利点をわかっていても、誰もが可能だと思っている以上のことをチームに要求してしまう。劇的な改善を目撃した組織は、ストレスがかかったときに、古い習慣やムダに戻ることを選んでしまう。どれだけ効果の高いものであっても、過去の振る舞いに戻ってしまうのはよくあることである。

組織的な後戻りは対処が難しい。チームのコントロール外だからだ。XPチームは、素晴らしい仕事をする。余計なストレスや時間外労働のない状態で、欠陥を最小限にして、期日を正確に守りながら、当初の要求よりも多くの機能をデプロイする。その他のチームは、徹夜で作業しながら、期日間際に無理やりスコープを削減して、欠陥まみれの状態でデプロイする。だが、組織の規模が縮小したときには、XPチームから解雇される。その他のチームのほうが「献身的」な姿勢を見せているからだ。XPチームの働き方は他とは違っているので、社会的および政治的に否定的な影響を与える可能性のある目立ち方をしてしまう。チームは、組織の目標達成に尽力することを強調して、自分たちの働き方がどのように目標を支えるのかを示す必要がある。

組織内でXPを擁護してくれる経営幹部のスポンサーを見つけよう。そうすれば、新しい仕事のスタイルに移行するときに、会社とのやりとりがスムーズになる。自分たちを守ってくれる人には説明責任を果たそう。組織の上層部から支援が得られなければ、チームの現在のラポール（信頼関係）や生産性で満足せざるを得なくなってしまう。

組織変革を始める場合も、自分から始めることになる。これはあなたがコントロールできる変化だ。まずはスキルを磨いて、それを実際に使ってみよう。自ら見本を示すことは、強力なリーダーシップの方法である。「達人プログラマー」のデイヴ・トーマスとアンディ・ハントは、この戦略の素晴らしい例である。

最近、ある技術リーダーと話をした。彼は、プログラマーが自動テストを書くの

[†1] 訳注：「採用する」の *adopt* という単語には「養子にする」という意味もある。

を全面的に支援していると言った。そこで、彼に聞いてみた。

「素晴らしいですね。JUnit は試してみましたか？」

「いえ、自分ではテストを書いたことはないんです。素晴らしいアイデアだとは思います」

自分で試すつもりのないことを他人に期待すれば、相手をリスペクトしていないことになる。それでは効果的ではない。自分で引き受けるつもりのないリスクを他人に負わせれば、人間関係が損なわれることになるし、チームの結束力も破壊されてしまう。このように権限と責任のバランスが崩れれば、不信感が生まれる。自分の学習、フィードバック、自己改善の機会も失われてしまう。

スキルを学び、それを実際に使ってみる戦略は、さまざまなスケールで機能する。

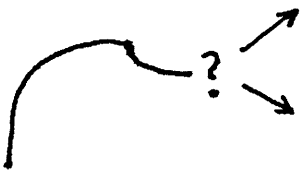
- テストファーストプログラミングを学んでから、チームと共有する。
- チームでストーリー単位の見積りと開発を学んでから、組織内の顧客にストーリーを選んでもらう。
- 組織内で安定したソフトウェアを予測どおりにデプロイすることを学んでから、組織外の顧客に計画づくりに加わってもらう。

いずれの場合も、やることは同じだ。自分を変えてから、その変化の成果を他の人にも提供する。どちらのステップもバリューを生み出している。自分を変えるときは、改善の方法が見つかったからである。顧客に新しいスキルを提供するときは、その利点が相手に伝わっているのである。

「継続的」改善というと、絶え間なく連続的に改善を続けているように聞こえるかもしれないが、そうではない。継続的改善とは、継続的に、注意して、フィードバックに反応して、積極的に改善を受け入れる、という意味だ。つまり、改善する方法がわかったときに、改善するのである。何かを変更して、その効果を観察して、変化の意味を飲み込んで、それを習慣として定着させる。いずれ停滞期に達するだろうが、そこからさらにフィードバックを吸収して、次の機会を見つけることができる（図 23）。

▶図23　「継続的」学習は連続的ではない

新しいプラクティスを試したら、パフォーマンスが落ちることもあるだろう（図24）。



▶図24　学習は直線的ではない

改善が不可能なわけでも、試したプラクティスが悪かったわけでもない。プラクティスに必要な経験が足りなかったのかもしれない。隠れた前提条件が不十分で、新しいプラクティスの準備ができていなかったのかもしれない。変更を元に戻して、根本的な問題に取り組もう。そのプラクティスが再び改善リストの一番上に登場したときには、すでに準備ができているはずだ。

改善の道筋は、順調でも予測可能でもない。開始地点の文脈と改善の道筋そのものに大きく影響される。あるチームでは魔法のようにうまくいったプラクティスが、他のチームでは大惨事になることもある。

改善の道筋は常に険しくて曲がりくねっている、と言いたいわけではない。数週間で好転させたチームをいくつも見たことがある。急速な方向転換を円滑にする条件を以下に挙げよう。

- ✧ **価値の統一。**チームと組織が、XPの価値を受け入れて仕事をする意思がある。
- ✧ **苦痛。**チームが、解雇やデプロイの失敗などの喪失感を最近味わっている。苦痛の記憶が明確であれば、人々は劇的な変化を求める。

このような改善の道筋は、波瀾万丈で非連続的であるかのように思える。だが、実際には通常のカーブが短期間に連続しているだけである。それが可能なのは、チームに変化を受け入れる力があるからだ（図25）。

▶図25　高速な学習

急速な変化のために、上記のような条件を強制するのは倫理的ではない。だが、このような状況に置かれていたら、奇跡のような変化をもたらす絶好の機会である。奇跡といっても魔法ではない。通常の出来事が高速に発生しているだけである。

コーチの選択

「コーチ」という言葉には、チームの一員になることと、独立した視点を持つことのバランスをとる、という意味が込められている。最初にコーチは、改善する箇所を特定して、それに対処するための実験をリードする。コーチは経験と視野を持っていて、日常的なグループダイナミクスに巻き込まれることはない。

経験豊かなコーチがいなくても、XPはうまく適用できる。多くのチームはそのようにしている。その場合、経験や視野を持ったコーチから恩恵を受けることはできないが、自分たちの経験から学ぶことができる。ただし、XPを適用するには、チーム内もしくはチーム外のリーダーシップが不可欠だ。

XPの価値、原則、プラクティスは、例を使って学ぶのが最適だ。間違いを犯した人から学ぶこともできるし、自分で間違いを犯して学ぶこともできる。私はその両方から学んできた。私とペアプログラミングした人は「テストファーストプログラミングのことは知っていましたが、今**本当**の意味で理解できました。実際に**すべて**の変更の前に**小さな**テストを書くんですね」と言ってくる。私もこまめなリリースを理解していると思っていたが、毎日デプロイしているチームと一緒に仕事をしたときに「**本当**に**毎日**新しいソフトウェアをデプロイしているんだ」と思ったことがある。先ほどと同じようなアハ体験だ。言葉の意味を理解したからといって、そのことを本当に理解しているわけではない。コーチがいれば、学びをうまく促進さ

せることができる。

コーチは、コミュニケーションのボトルネックを発見して対応する。コーチは、チームが不安を感じているときに、単純なことをやるべきだと思い出させる。コーチは、チームにプラクティスを使う気にさせる。たとえば、「あのテストはもう書きましたか？」と質問する。コーチは、効果的な価値やプラクティスの見本を示す。コーチは、プロセス全体に責任を持ち、チームが持続的なペースで働き、継続的な改善を行えるようにする。コーチは、自分の理解したことをチームが問題に対処できるような方法で伝える。

コーチの選択は、重要かつ困難なものである。コーチは、既存の価値を無視するわけにはいかないが、XPの価値にもこだわり、その方向にリードし続ける必要がある。コーチは、みんなが自力では簡単に学習できないことを教えられるだけの技術スキルを持つべきである。そして最後に、これが最も重要なことだが、依存ではなく自立を促すべきである。優れたコーチというのは、あなたが準備ができたと思う少し前に離れていく。そして、持続可能な、利益性の高い、安定した、高速な、楽しいソフトウェア開発の道を自分たちで確実に見つけることのできるチームを残していく。

XPを使うべきではないとき

XPを使っても効果がないのは、実際の価値とXPの価値が相容れない組織である。ここで「実際の価値」と言ったのは、多くの組織は見せかけの価値を持っているからだ。それは行動によって明らかになる価値とは違っていたり、矛盾していたりする。XPには、対応する価値を表現したり、強化したりするプラクティスがある。組織の実際の価値が「秘密、複雑、孤立、臆病、不遜[†2]」の場合、新しいプラクティスによって突然正反対の価値がもたらされると、改善よりもトラブルを招くことになるだろう。

[†2] 訳注：XPの5つの価値と正反対になっている。

第21章

エクストリームの純度

「私のチームはエクストリームですか？」と何度も聞かれる。グラフや指標を組み合わせて「エクストリーム」を計測しようとする人たちがいる。ふりかえりのツールとして使うのであれば納得できる。だが、10点が5点の2倍優れているというような計測方法はバカげている。「私のチームはエクストリームですか？」の質問に対して、バイナリな（客観的な二者択一の）答えや数値的な答えを期待するのはおかしな話だ。

私はテックスメックス料理[†1]が好きだ。テックスメックス料理の看板が出ているレストランを見かけたら、肉が多くて、豆を使った、スパイシー料理を期待する。そこで、トマトソース、スイスチーズ、ピクルスが挟まれたフニャフニャしたトルティーヤが出てきたら（これは私の娘が以前、チューリッヒで経験したことだが）ガッカリするだろう。この場合に「この料理はテックスメックス料理か？ **本当に**テックスメックス料理か？」と問いかけることは重要である。それがこれから食べるものに対する期待になるからだ。

「私のチームはエクストリームですか？」という質問もチームに対する期待を設定する。だが、その質問にはバイナリな答えは存在しない。誰かが「チームがボストンにも分散しています。XP以外のあらゆるプラクティスを実施しています。それでもXPでしょうか？」と質問してきたら、私は肯定もしないし、否定もしない。私の判断は重要ではない。「チームメンバーが納得できることを持続可能な方法でやっているでしょうか？」。これこそが質問だ。ただし、質問に答えられるのは、

[†1] 訳注：テキサス風のメキシコ料理。

チームメンバーだけである。

XPは、全員同席したほうがよりよい結果が得られるという理論を持っている。だが、すでにチームが十分によい結果を収めていることもある。本物のXPか否かに関係なく、うまくやっているのだ。そうしたチームがさらに改善したければ、一部またはすべての時間に全員同席して、フェイスツーフェイスのコミュニケーションを増やしたり、すでに使用しているプラクティスを強化したり、XP以外の分野（ユーザビリティ、チームワークやコミュニケーション、人事、マーケティング、営業）のプラクティスに挑戦したりすることができるだろう。

あらゆるソフトウェア開発スタイル（もしくはそれらを使わないこと）が「エクストリーム」だと言っているわけではない。XPが価値、原則、プラクティスを用意しているのは、ガイダンス、チャレンジ、説明責任を提供するためだ。たとえば、「仕事の協力がうまくいかないのであれば、人間関係やパフォーマンスが改善するかを確認するために、こうこうこういう理由で、これらのことを試してほしい」と伝えることができる。単にドキュメントを書きたくなくて、その言い訳としてXPを使うのであれば、周囲からはドキュメントを書くことを求められるだろう。好戦的に「我々はエクストリームですから、ドキュメントを書くつもりはありません」と答えれば、コミュニケーションを軽視していることになる。それではコミュニケーションの価値を受け入れているとはいえない。

個人やチームがエクストリームかどうかをバイナリで判定するテストは存在しない。多くのチームがそれぞれの価値、原則、プラクティスを持ち、ソフトウェア開発のなかでバリューを生み出すことに成功している。XPチームで失敗するくらいなら、純粋なウォーターフォールのチームで成功するほうがいい。目的は人間関係とプロジェクトの成功であり、XPクラブの会員になることではない。

チームがエクストリームだと言えば、コミュニケーションの取り方、開発プラクティス、成果のスピードや品質に対して、相手は期待を抱くはずだ。

認証と認定

同じく一般的なテーマの話題として、個人またはチームの認証（*certification*）の話がある。認証プロセスにおける認証機関は、自らの信用をかけて認証した人の適正を保証し、その人に対する一定の責任を受け入れている。たとえば、専門医資格を持った医者が不適正だとわかれば、その認証機関の信頼性は失われてしまう。

コンピューティングの分野には、そこまで踏み込んだ認証は存在しない。他人の開発の選択について、法的な責任を負いたいと思う人がいるだろうか？ 認証機関が自分たちの認証に責任を持たないのであれば、それは単なる証明書の発行と金儲け

をしているだけだ。

それでも、XP のスキルがあると主張する人に本当にスキルがあるのかを知りたいという声はある。非公式な人脈は用意されているが、誰に聞けばいいのかわからない。

ラ・レーチェ・リーグ[†2]では、母乳で子どもを育てたいお母さんたちに情報を提供する「集い」をリーダーが開催している。このリーダーの認定（*accrediting*）モデルが魅力的だ。このモデルでは、両者が自らの振る舞いに全面的な責任を担っている。自己申告した内容にお互いが同意したことを公に認めるのである。母親たちをサポートするという目的のために、リーダーの価値と組織の価値が統一されている。

ラ・レーチェ・リーグのアプリカント（リーダー志願者）は、最初に既存のリーダーから勧誘を受ける。そのリーダーとボランティアの認定者がメンターとなり、アプリカントの知識やスキルを評価する。このプロセスには、以下のことが含まれる。

- ✧ 「集い」をリードするために必要なアプリカントの技術的、社会的、組織的なスキルの知識の評価。
- ✧ アプリカントと一緒に「集い」の運営とそのパフォーマンスの批評。
- ✧ アプリカントが自らの母乳育児の経験をふりかえって作成した文書のレビュー。
- ✧ 他のリーダーたちとの交流。
- ✧ 地域コンファレンスでグループに紹介。
- ✧ 奨励とサポート。

このモデルは、現在コンピューティングの分野で行われている認証モデルよりも、はるかに実りのあるやり方だ。認証の手続きや厳密性の確保にどのようなものを使ったとしても、その意味するところは必然的に主観的なものとなる。結局、XP の価値を大切にして、XP の原則を日々のプラクティスに取り入れていれば、あなたは XPer だ。XPer は集団のなかでお互いを認識できる。このプロセスが形式化できれば、XP の明確な存在感が確立され、雇用者と被雇用者の期待を一致させることができ、お互いにリーダーとして指導や支援をする方法を身に付けられるだろう。そうすれば、世間のソフトウェア開発に対する考え方を一変させる大仕事に取り掛かることができる。

[†2] 訳注：詳しくは、ラ・レーチェ・リーグ日本の公式サイト（http://www.llljapan.org/）を参照してほしい。

第22章

オフショア開発

XPの価値、原則、プラクティスは、小さなチームが全員同席しているところが「スイートスポット」になるが、そのスイートスポット以外にXPを適用するケーススタディーが、オフショア開発である。

私は「オフショア」という言葉が持つ政治的かつ人種差別的な意味合いが好きではない。高賃金の白人が低賃金の有色人種を利用して、「奴ら」が「我々」の仕事を奪ったと文句を言うような意味合いだ。「オフショア」には、権限の不均衡が伴う。権限の不均衡は、ソフトウェア開発を簡単に狂わせてしまう。ここでは、オフショアの代わりに「マルチサイト（複数拠点）」という言葉を使うことにしよう。XPは地理的に分散したチームにも同じように適用できるからだ。

プロジェクトをマルチサイトで運営する理由はいくつもある。給料の違いはその理由のひとつにすぎない。たとえば、データベースの担当者がトロントにいて、通信の担当者がデンバーにいることもある。マルチサイト開発の理由がどのようなものであれ、最終的にはビジネスの意思決定によって決まる。全員同席していないことで生まれるムダを上回るだけの利点があるかを考慮すべきである。

XPの価値は、全員同席しているチームだけでなく、マルチサイト開発にも適している。ただし、距離が離れていることで自然と孤立が生まれるため、より密度の高いフィードバックを受け入れる必要がある。五感を使ったフェイスツーフェイスのやりとりができないため、こまめにコミュニケーションをとらなければいけない。複雑すぎるところを偶然発見する機会が少なくなるため、シンプリシティの実現に向けて熱心に取り組まなければいけない。勇気はどんな状況でも重要だ。文化やライフスタイルが違うため、分散チームの全員をリスペクトすることも重要である。

マルチサイトプロジェクトのために修正が必要なプラクティスもあるはずだ。たとえば、計画づくりは週単位よりもこまめに行う必要があるだろう。それは会話の感覚を維持するためでもあるし、ある拠点から他の拠点にやるべきことを命令するような状況を避けるためでもある。難しそうという理由だけでプラクティスを諦めてはいけない。単一のコードベースは、全員同席しているときよりもマルチサイト開発のときのほうが、お互いの接点として重要になる。いかなる技術的な障害があろうともその解決に取り組み、単一のコードベースを実現してほしい。そうすれば、共通のプログラムにチームが協力して取り組み続けることができる。

相互利益の原則は、マルチサイト開発のすべての問題に最も影響を与える。仕事の拠点を移動するときは特にそうだ。関係者全員に最も有益な成果というのは、あらゆる拠点のプログラマーが（相対的に）給料のいい仕事をすることと、ソフトウェアのバリューを高めて顧客を満足させ、より多くのソフトウェアにお金を支払ってもらうことである。仕事は賃金の安いところに流れるわけではない。誠実性と説明責任があるところに流れるのである。誠実性と説明責任がもたらされるのであれば、タイムゾーンが大きく離れた会社であっても、顧客はコミュニケーションの難しさを克服するために必要な投資を行い、その会社に仕事を依頼するはずだ。ソフトウェア業界が、安いコストで多くのバリューを生み出すことを学べば、一時的に失業が増えても、需要の増加によって埋め合わせされるだろう。

コストの高い地域で生き残るには、効率性、誠実性、説明責任を高めることが不可欠である。100人規模の高価な請負業者が、説明責任を果たさずにプロジェクトに取り組むような時代は終焉を迎える。そして、効率的で説明責任を果たす10人のプログラマーが、代わりにそのプロジェクトに取り組むだろう。あるいは、プロジェクトそのものが他の場所に移動してしまうだろう。コストの高い地域で技術職の雇用を守るには、劇的な改善が必要となる。コストの高いチームは、直接的なコミュニケーションが重要になるレバレッジの高いプロジェクトに集中する必要があるだろう。チームの効率性を高め、他の地域で同様のプロジェクトを行うときと同程度にコストを抑えなければいけない。

コストの低いチームは競争のために、生み出すバリューを高める必要がある。CMMのようなテイラー主義の改善プログラムで技術やマーケティングの優位性を高めるやり方は、近いうちに消滅するだろう。大量の人員を投入できる余裕があるからといって、それが最も有益な問題解決方法であるとは限らない。大量の人員に依存している組織は、スループットを高めながら少しずつチームの規模を縮小する必要がある。

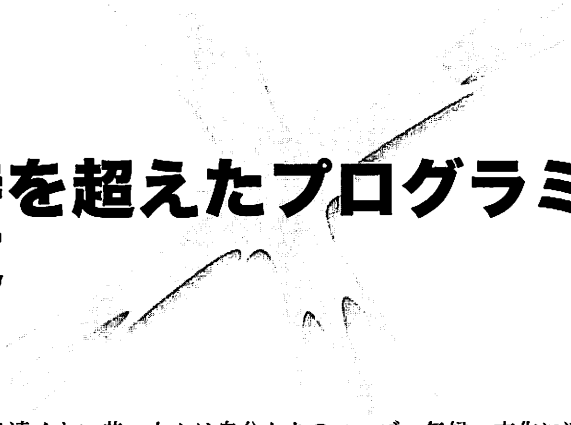
今後のグローバルなソフトウェア開発のシナリオは2つある。ひとつめのシナ

リオは、コストの高い国々が時計の針を止めようとするものだ。今後も同じように開発が続けられるように、政治的な力を利用するのである。これではコストの低い国々は改善のインセンティブを持てない。そして、ソフトウェア開発は停滞する。

もうひとつのシナリオは、世界中のプログラマーがソフトウェア開発のさまざまな形のムダを排除するというものだ。信頼性や効果の高い安価な新世代のソフトウェアは、ビジネスにおいて新たな使用法をいくつも発見されていく。そして、ソフトウェアのグローバル市場が活気づく。あらゆる国で10年前よりも多くのプログラマーが雇用されるだろう。

改善していくことが当然というわけではない。現状を維持しながら、ゆっくりとソフトウェア開発を続けていく道もあるだろう。だが、劇的な改善をしなければ、ソフトウェアのグローバル市場は停滞し、製造業やバイオテクノロジーに魅力的な投資が流れていく。今後50年にかけてソフトウェア開発の技能やビジネスを促進していくためにも、あらゆるプログラマーがバリューの高いソフトウェアを生み出す挑戦を受け入れてほしいと願っている。これから効率性が高まり、マルチサイト開発が行われることによって、どこかの地域で損失が発生したとしても、それを補ってあり余るほどに市場が拡大していくものと私は信じている。

第23章



時を超えたプログラミングの道

そう遠くない昔、人々は自分たちのニーズ、気候、文化に適した空間の設計と構築の方法を知っていた。建築家のクリストファー・アレグザンダーは、そのように著している。子どもの頃、大工だった曽祖父の話を聞いたことがある。家族で新しい町に引っ越すと、すぐに臨時仕事を始めて、お金を節約したそうだ。そして、お金がたまったら、材木を購入して、家を造った。建築家としての訓練を受けたわけでもないのに、家族に適した家の設計方法を知っていたのである。

アレグザンダーは、建築家の自己中心的な関心事は、施主の関心事と一致していないと指摘している。建築家は仕事をすぐに終わらせて、賞を獲得したいと思っているが、重要な情報を見逃している。それは、施主がどのように生活したいかという情報だ。アレグザンダーの夢は、生活に最も大きな影響を受ける人の手のなかに、空間を設計する力を取り戻すことだった。

アレグザンダーはそのための手段として、建築のパターンを収集し、家の設計や構築で繰り返し発生する問題の解決策をまとめた。パターンはそれ自体を目的としたものではなく、設計の専門家とその空間で生活や仕事をする人たちとの力のバランスをとる手段である。

アレグザンダーのビジョンには、建築家の役割も残されている。あらゆるプロジェクトには、うんざりするほどありきたりな問題もあるが、そのプロジェクトに特有の問題もある。いきいきとした空間を設計する鍵は、その空間を使う人の好みや社会的な人間関係を深く理解して、建築家の技術に対する深い理解と結び付けることである。この2つの観点を組み合わせ、どちらかを優位にすることなくうまく調和させることができれば、人間の欲求を満たしながら、雨を遮断できる空間の設

計と建設が可能となる。

ソフトウェア開発の仕事を始めたときに、私は力の不均衡を目の当たりにした。それは、アレグザンダーが建築の世界で戦っていたのと同じものだ。私はシリコンバレーで育った。そこでは、エンジニアリングが王様だった。「あなたに必要なものを与えよう。必要かどうかは知らなくても構わない」。よく引き合いに出されたモットーだ。このようにして作られたソフトウェアは、技術的には優れていたが、役に立たないものが多かった。

経験を積んでいくと、正反対の不均衡を目にするようになった。ビジネスの関心事が開発を支配する世界だ。ビジネス上の理由だけで期日やスコープを設定すると、チームの誠実性を維持できない。ユーザーやスポンサーの関心事は重要だが、開発者のニーズにも正当な理由がある。ユーザーとスポンサーと開発者の三者で、相互に情報交換をすべきだろう。

XP はもともとプログラマーに向けて書いたものだった。私自身がプログラマーだったからだ。チームのなかで共感できたのもプログラマーだった。だが、高みに到達することが目的であれば、ソフトウェア開発は「プログラマーとその他大勢」で成立するものではない。それが、過去 5 年間で私が学んだことだ。関係者全員の関心事のバランスがとれていなければ、開発に貢献できない人が出てくる。チームが成功するには、そのような人の視点が重要だ。私の今後の目的は、チームが技術とビジネスの関心事に常に調和をもたらせるように支援することである。

調和とバランスは XP の狙いだ。テストを書くというのは、それ自体もよいことではあるが、もっと大きなタスクの準備にすぎない。それは、ソフトウェアでお金を稼ぐために集まったさまざまな人たちの絆を強くするというものである。気持ちの変化が伴わなければ、世界中のあらゆるプラクティスや原則を使ったとしても、短期的なわずかな利益しか得られない。あなたとチームは運命を共にしている。そのように行動すれば、そのことを信じられるようになるだろう。

空間の設計者と利用者の力のバランスをとるアレグザンダーの試みは、最終的に失敗した。建築家は自らの力を手放そうとしなかったし、施主は自らの要求を伝えることさえ知らなかった。だが、プログラムは建物ではない。ソフトウェア開発は建築ではない。我々の構成要素は、建築の材料とは違う。我々の社会構造は、数千年も変わらない人間関係に縛られてはいない。我々は、ソフトウェア開発において新たな社会構造を作り出し、そのなかで技術的な高みとビジネスのビジョンを結び付け、独自のバリューを持ったプロダクトやサービスを生み出す機会を持っている。これが我々の強みだ。

XP の成功は、信頼できるソフトウェアのすばやい見積り、実装、デプロイができ

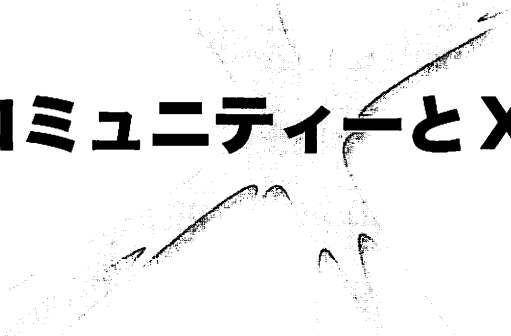
る優秀なプログラマーの増加にかかっている。このようなプログラマーがいれば、チームのビジネス担当者に意思決定を任せることができるだろう。チームのなかで権限と責任を共有するのは、非現実なことに思えるかもしれない。このバランスをとるには、相互のリスペクトが前提となる。絶対的な権限は存在しない。権限を悪用すれば、XP のパワーが失われてしまう。見積りをごまかし、プライドを捨てて仕事を片付けていると、チームは潜在的な力を発揮できない。XP は、経営幹部、マネージャー、顧客も含めたチームのメンバー全員が、自分のできることを全力で貢献するかどうかにかかっている。チームのみんなが協力して仕事をすれば、個々のメンバーの力を合わせた以上のことが成し遂げられる。権限の共有とは、実践主義的なものであり、理想主義的なものではない。

ソフトウェアの可能性を認識するには、チームワークが必要となる。コンピューティングが誕生してから最初の半世紀は順調に進んできた。だが、次の世紀はコンピューティングではなく、バイオロジーの世紀になると予測している講演を何度か聞いたことがある。コンピューティングは脇役に格下げされるのだろう。ソフトウェアがこれまでどおりのビジネスを続けていくのであれば、その予測は現実になるだろう。ソフトウェアのツールや技法がゆっくりと進んでいくのであれば、バイオロジーが社会的および経済的な変化の担い手として、すぐにソフトウェアを追い越していくだろう。

ツールや技法は何度も変化するが、大きく変化するわけではない。一方、人はゆっくりとだが、深く変化する。XP の課題は、このような深い変化を促し、個人の価値と相互の人間関係を新しいものにして、ソフトウェアに次の 50 年間の居場所を用意することだ。人間の精神の可能性を発揮することが、まだ想像もできないコンピューティングの未来につながっていくのである。

第24章

コミュニティーとXP



サポートしてくれるコミュニティーの存在は、ソフトウェア開発の偉大な資産である。ここで言うコミュニティーとは、今のチームかもしれないし、共通の考えを持った地域のソフトウェア開発者の集まりかもしれないし、グローバルなコミュニティーかもしれない。いずれにしても偉大な資産である。コミュニティーは、自分の問題を声に出したり、みんなと経験を共有したりする安全な場を提供してくれる。コミュニティーは、自分に共感してくれる人を見つけたり、誰かの声に自分の耳を傾けたりする絶好の場である。

誰もがサポートを必要とするときがある。そのためにコミュニティーは重要だ。人と人との関係は、安全に実験ができる安定した場を提供してくれる。新しい経験を誰かと一緒に評価すれば、変化に対する平常時の不快感がどれだけあるかを把握できる。コミュニティーの誰かが新しい視点を必要としていれば、あなたが耳を傾けてあげることができる。求められたときにだけ、自分の意見を提供すればいい。

コミュニティーでは、自分から話すよりも耳を傾けるスキルのほうが重要だ。オープンで正直なコミュニケーションによって、コミュニティーの参加者は安心感や理解を得られなければいけない。自分の話を聞いてもらいたいときもあるだろう。そのときに、文句、罵倒、自分の頭の良さを自慢するなど、呼び方は何でも構わないが、頼みもしないアドバイスばかりが返ってきたら、そのコミュニティーは安全ではない。心の底から話したいことがあれば、それを安全に話すことができ、受け入れてもらえるという確信を持てるようにすべきである。

コミュニティーは一緒に勉強する場にもなる。XPには、練習が必要なスキルも数多く含まれている。地域のXPグループでは、参加者が興味のある話題を勉強し

ている。近くになければ、自分で始めてみてはいかがだろうか。企業内のグループも役立つだろう。ただし、さまざまな視点を自分の経験に取り入れることが重要だ。できれば、毎日働いている会社の文化にはない視点を取り入れたい。

コミュニティーは説明責任の場でもある。つまり、コミュニティーでは、自分の発言に責任を持たなければいけない。今日仲間の誰かに何かを説明すれば、明日仲間の誰かが何かを説明してくれるだろう。説明責任は変化を引き起こす上で特に重要だ。近道してみたり、別の道に後戻りしたりしたくなることもある。だが、自分の行動を仲間に説明しなければいけないとしたら、そのことを正当化するのは難しい。説明責任を果たすためにも、コミュニティーには安全性が必要である。相手の秘密を大切にしたり、求められたときにだけアドバイスしたり、よく考えてから判断したりすることは、すべてが安全性につながっている。

コミュニティーは質問や疑問を投げかける場だ。コミュニティーには、個人の意見が重要である。衝突や意見の不一致は、共に学習するための下地となる。衝突を抑えこんでしまうのは、コミュニティーが弱い証拠だ。本当に重要なアイデアならば、そのようなことで重要性が失われることはない。コミュニティーのメンバーは、常に意見を一致させる必要はない。お互いをリスペクトしながら、意見の不一致に対応すればいい。安全なコミュニティーの参加条件に、無理に意見を合わせなければいけないという条件はない。

XPには活発なオンラインコミュニティーがあり、気軽に参加することができる。最も活発なのは、Yahoo!にホストされているコミュニティーだ[†1]。英語以外の言語にもXPのメーリングリストがあるので、オンラインで検索してほしい。地域のユーザーグループもオンラインで見つかるはずだ。月に1回開催されているところがほとんどだが、毎週開催されているところもあるようだ。

地域や世界のコミュニティーに参加してほしい。最高の自分になれるコミュニティーを見つけよう。コミュニティーが見つからなければ、自分で始めてみよう。難しい課題に取り組んでいるのは、あなたひとりではないはずだ。ひとりのときにはできなかったようなことでも、コミュニティーなら成し遂げられる。

[†1] http://groups.yahoo.com/group/extremeprogramming

第25章

結論

私はプログラマーの生活をよくするためにXPを体系化した。体系化しているうちに、XPは世界における私のあり方を決めるものになった。けれどもそれは、私自身の価値について考えさせ、それに合わせた行動を求めるようなあり方だ。最初に私自身を改善しなければ、何も改善されない。そのことを私は発見した。

XPの鍵は誠実性（*integrity*）だ。本当の価値と調和のとれた行動をすることだ。誠実性を目標にした途端、私は実際の自分の価値が世界から持っていると思われたい自分の価値ではないことに気づいた。この5年間は、実際の自分の価値を自分の持ちたい価値に変える旅だった。

この旅は、完璧からはかけ離れていた。理想からほど遠いことを痛感させられることもある。だが、すべてがうまくまとまることもある。価値と理想が一致して、そこから自然と行動が流れてくるような瞬間だ。そのときは旅を続けてよかったと思える。

XPによって、私は他人からリスペクトされ、他人をリスペクトする人間になりたいと思っている。自ら全力を尽くし、常に改善を目指そうとしている。自分が誇りに思える価値を胸に抱き、その価値と調和のとれた行動をしている。

XPの価値は、ビジネスの世界で実践すべきものだ。快適に暮らせる生活を送るだけでなく、関係者全員がお互いにリスペクトした人間関係によって、新しい働き方を開発するのである。活発に積極的に、世界に貢献できるソフトウェアを作り出すのである。創造的にいきいきと働くのである。

XPを適用した人たちが、ソフトウェア開発や自分たちの生活に新たな希望をもたらす様子を目にしてきた。あなたはすでに、XPを始められるだけの知識を持っ

ている。今すぐに始めよう。自分の価値について考えてみよう。その価値と調和した生き方を意識的に選択しよう。最初のプラクティスを選び、自分の道を歩き出そう。快適に過ごせる世界を求めているのであれば、バランスのとれた生活と順調な仕事をしよう。XPとは、あなたが自分の理想について考え、その理想にもとづいて行動するための方法だ。

注釈付き参考文献

さまざまな分野の書籍を読んだことで、私の理解は深められた。ここでは、XP に関連するアイデアについて書かれた興味深い書籍を紹介しよう。

哲学

- Sue Bender, Plain and Simple: *A Woman's Journey to the Amish*, HarperCollins, 1989; ISBN 0062501860.
『プレインアンドシンプル —— アーミッシュと私』(鹿島出版会)

  シンプリシティと明確さの重要性について調査している。

- Leonard Koren, *Wabi-Sabi: For Artists, Designers, Poets, and Philosophers*, Stone Bridge Press, 1994; ISBN 1880656124.

  侘び寂びとは、いびつさや機能性を賛美する美意識のことである。

- Richard Coyne, *Designing Information Technology in the Postmodern Age: From Method to Metaphor*, MIT Press, 1995; ISBN 0262032287.

  モダニストとポストモダニストの考えの違いを議論している。メタファーの重要性に関する素晴らしい議論も含まれている。

- Philip B. Crosby, *Quality Is Free: The Art of Making Quality Certain*, Mentor Books, 1992; ISBN 0451625854.
『クオリティ・マネジメント —— よい品質をタダで手に入れる法』(日本能率協会マネジメントセンター)

  時間、スコープ、コスト、品質の 4 つの変数を使ったゼロサムモデルからの脱却。品質を下げてもソフトウェアをすばやく手に入れることはできない。むしろ品質を上げたほうが、すばやくソフトウェアを手に入れることができる。

- George Lakoff and Mark Johnson, *Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Thought*, Basic Books, 1998; ISBN 0465056733.
『肉中の哲学 —— 肉体を具有したマインドが西洋の思考に挑戦する』(哲学書房)

  メタファーと思考について優れた議論が行われている。メタファーを組み合わせる方法についても記述がある。古くからあるソフトウェアのメタファーは、土木工学や数学の分野から引用してきたものだが、ゆっくりとソフトウェアエンジニアリング特有のメタファーに変化している。

- Bill Mollison and Rena Mia Slay, *Introduction to Permaculture*, Ten Speed Press, 1997; ISBN 0908228082.
『パーマカルチャー —— 農的暮らしの永久デザイン』(農山漁村文化協会)

西洋諸国における大量消費は、一般に搾取や消耗と結び付いてきた。パーマカルチャーとは、よく考えられた農業の学問体系であり、シンプルなプラクティスの相乗効果によって、持続可能な土地の利用を目指すものである。XP との類似点がいくつも存在する。たとえば、成長は要素の相互作用によって発生するとされている。パーマカルチャーでは、さまざまな植物をスパイラル状に植えたり、池の形を不規則にしたりするなどして、相互作用を最大化する。XP では、オンサイト顧客やペアプログラミングなどを利用して、相互作用を最大化する。

態度

- Christopher Alexander, *Notes on the Synthesis of Form*, Harvard University Press, 1970; ISBN 0674627512.
『形の合成に関するノート』(鹿島出版会)

  アレグザンダーは、デザインは制約の対立を解消する決定であり、それが残された制約を解消する決定につながると考えるところから始めた。

- Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979; ISBN 0195024028.
『時を超えた建設の道』(鹿島出版会)

  クリストファー・アレグザンダーの建築や建設に対する考えの概要を示したもの。建築物の設計者および建設者と利用者の関係は、プログラマーと顧客の関係とほとんど同じだ。

- Ross King, *Brunelleschi's Dome: How a Renaissance Genius Reinvented Architecture*, Penguin Books, 2001; ISBN 0142000159.
『天才建築家ブルネレスキ —— フィレンツェ・花のドームはいかにして建設されたか』(東京書籍)

  エクストリームな建築と建設。ブルネレスキは、問題に屈することを拒んだ。XP の「機会」の原則の好例である。

- Field Marshal Erwin Rommel, *Attacks: Rommel*, Athena, 1979; ISBN 0960273603.

  明らかに絶望的な状況で前進する事例。

- Dave Thomas and Andy Hunt, *The Pragmatic Programmer*, Addison-Wesley, 1999; ISBN 020161622X.
『達人プログラマー』(ピアソン・エデュケーション)

  デイヴとアンディは、技術的スキルと私が「エクストリーム」と呼んでいる態度を結び付けている。

- Frank Thomas and Ollie Johnston, *Disney Animation: The Illusion of Life*, Hyperion, 1995; ISBN 0786860707.
『ディズニーアニメーション —— 生命を吹き込む魔法』(徳間書店)

  ビジネスや技術の変化に対応するために、ディズニーのチーム体制が長年にわたり、どのように進化したかが描かれている。ユーザーインターフェイスデザイナーのヒントになることが満載。素晴らしい絵も収録されている。

- *Office Space*, Mike Judge, director, 1999; ASIN B000069HPL.
  映画「リストラ・マン」

  パーティションのなかの人生観。

- *The Princess Bride*, Rob Reiner, director, MGM/UA Studios, 1987; ASIN B00005LOKQ.
  映画「プリンセス・ブライド・ストーリー」

  「私たちは助からないわ」
  「バカな。今まで誰もやったことないからそう言ってるだけだ[†1]」

創発的プロセス

- Christopher Alexander, Sara Ishikawa, and Murray Silverstein, *A Pattern Language*, Oxford University Press, 1977; ISBN 0195019199.
  『パタン・ランゲージ —— 環境設計の手引』(鹿島出版会)

  創発的特性を生み出すためのルール体系の例。このルールがうまくいくかどうかは議論の余地があるが、ルールそのものは興味深く読める。作業場のデザインに関する簡潔だが素晴らしい議論も含まれている。

- James Gleick, *Chaos: Making a New Science*, Penguin USA, 1988; ISBN 0140092501.
  『カオス —— 新しい科学をつくる』(新潮社)

  カオス理論のやさしい入門書。

- Stuart Kauffman, *At Home in the Universe: The Search for Laws of Self-Organization and Complexity*, Oxford University Press, 1996; ISBN 0195111303.
  『自己組織化と進化の論理 —— 宇宙を貫く複雑系の法則』(日本経済新聞社)

  カオス理論の少しだけ難しい入門書。

- Roger Lewin, *Complexity: Life at the Edge of Chaos*, Collier Books, 1994; ISBN 0020147953.
  『コンプレクシティへの招待 —— 複雑性の科学』(徳間書店)

  これもカオス理論。

- Margaret Wheatley, *Leadership and the New Science*, Berrett-Koehler Pub, 1994; ISBN 1881052443.
  『リーダーシップとニューサイエンス』(英治出版)

  自己組織化システムをマネジメントのメタファーに使っている。

[†1] DVD の字幕では「これまでの通説を破ってみせる」。

システム

- Albert-Laszlo Barabasi, Linked: *How Everything Is Connected to Everything Else and What It Means*, Plume Books, 2003; ISBN 0452284392.
『新ネットワーク思考 —— 世界のしくみを読み解く』(NHK 出版)

  プログラミング、ソーシャル、技術などの分野の多くのネットワークは、本書で説明されているように「スケールフリー」である。

- Eliyahu Goldratt and Jeff Cox, *The Goal: A Process of Ongoing Improvement*, North River Press, 1992; ISBN 0884270610.
『ザ・ゴール —— 企業の究極の目的とは何か』(ダイヤモンド社)

  「制約理論」とは、システムの理解とスループットの改善の方法である。

- Gerald Weinberg, *Quality Software Management: Volume 1, Systems Thinking*, Dorset House, 1991; ISBN 0932633226.
『ワインバーグのシステム思考法』(共立出版)

  システムを要素の相互作業として考えるためのシステムと表記法。

- Norbert Weiner, *Cybernetics*, MIT Press, 1961; ISBN 1114239089.
『サイバネティックス —— 動物と機械における制御と通信』(岩波書店)

  深くて難しいシステムの入門書。

- Warren Witherell and Doug Evrard, *The Athletic Skier*, Johnson Books, 1993; ISBN 1555661173.
『アスレチックスキーヤー』(ストーク)

  相互に関連したスキーの法則もシステムである。最後のいくつかの法則を導入すれば、大きな改善が見られる。バランスが保たれていることと、バランスが少し崩れていることには、大きな違いがあるからだ。

人々

- Tom DeMarco and Timothy Lister, *Peopleware*, Dorset House, 1999; ISBN 0932633439.
『ピープルウエア 第 3 版 —— ヤル気こそプロジェクト成功の鍵』(日経 BP 社)

  ワインバーグの『プログラミングの心理学』に続いて、人の書いたプログラム、特にチームで書いたプログラムについて、実践的なやりとりを展開している。XP の「責任の引き受け」の原則の情報源である。

- Tom DeMarco, *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*, Broadway, 2002; ISBN 0767907698.
『ゆとりの法則 —— 誰も書かなかったプロジェクト管理の誤解』(日経 BP 社)

  ソフトウェア開発にゆとりの概念を適用している。

- Carlo d'Este, *Fatal Decision: Anzio and the Battle for Rome*, HarperCollins, 1991; ISBN 006092148X.

  エゴが明快な思考の妨げになる例。

- Robert Kanigel, *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*, Penguin, 1999; ISBN 0140260803.

  テイラーの仕事の有効性と彼の思考の限界を示した伝記。

- Gary Klein, *Sources of Power*, MIT Press, 1999; ISBN 0262611465.
  『決断の法則 —— 人はどのようにして意思決定するのか？』（トッパン）

  経験豊かな人が困難な状況でどのように意思決定しているかを説明したシンプルで読みやすい文章。

- Alfie Kohn, *Punished By Rewards: The Trouble with Gold Stars, Incentive Plans, A's, Praise, and Other Bribes*, Mariner Books, 1999; ISBN 0618001816.
  『報酬主義をこえて』（法政大学出版局）

  適切な報酬を与えれば他人をコントロールできると思い込んでいた私の幻想を打ち砕いた本。

- Thomas Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 1996; ISBN 0226458083.
  『科学革命の構造』（みすず書房）

  どのようにしてパラダイムは支配的なパラダイムになるのか。パラダイムシフトは予測可能な結果を伴う。

- Patrick Lencioni, *The Five Dysfunctions of a Team: A Leadership Fable*, Jossey-Bass, 2002; ISBN 0787960756.
  『あなたのチームは、機能してますか？』（翔泳社）

  チームでうまくいかないことや、それについて何ができるのかについて、読みやすく書かれている。

- Scott McCloud, *Understanding Comics*, Harper Perennial, 1994; ISBN 006097625X.
  『マンガ学 —— マンガによるマンガのためのマンガ理論』（美術出版社）

  最後の何章かで、なぜ人はマンガを描くのかについて触れている。これを読んで私は、なぜ人はプログラムを書くのかを考えた。マンガの技法と芸術を結び付ける優れた情報も載っている。それはプログラミングの技法（テスティングやリファクタリング）と芸術の関係にもよく似ている。間隔を使って情報を伝える方法や、狭い空間に雑然とさせずに情報を配置する方法など、ユーザーインターフェイスデザイナーに役立つ優れた情報も載っている。

- Geoffrey A. Moore, *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*, HarperBusiness, 1999; ISBN 0066620023.
  『キャズム —— ハイテクをブレイクさせる「超」マーケティング理論』（翔泳社）

  ビジネスの観点からのパラダイムシフト。新しいアイデアを受け入れるときの障壁のなかには予測可能なものもあり、それらに対応するための簡単な戦略もある。

- Marshall Rosenberg and Lucy Leu, *Nonviolent Communication: A Language of Life: Create Your Life, Your Relationships, and Your World in Harmony with Your Values*, PuddleDancer Press, 2003; ISBN 1892005034.
『NVC 人と人との関係にいのちを吹き込む法』(日本経済新聞出版社)

  NVC (非暴力コミュニケーション) の狙いは、観察と評価を引き離し、より深い要求に耳を傾け、自分の要求を明確に伝えることである。

- Frederick Winslow Taylor, *The Principles of Scientific Management, 2nd ed.*, Institute of Industrial Engineers, 1998 (1st ed. 1911); ISBN 0898061822.
『[新訳] 科学的管理法』(ダイヤモンド社)

  「テイラー主義」を生み出した本。専門化と厳格な分割統治によって、多くの車が安価に作り出せるようになった。だが、ソフトウェア開発の戦略としては、ビジネス的にも人間的にも意味がない。

- Barbara Tuchman, *Practicing History*, Ballantine Books, 1991; ISBN 0345303636.

  思慮に富んだ歴史家が、歴史研究の方法について考えていること。『マンガ学』と同様に、自分の行動の理由をふりかえるのに最適。

- Colin M. Turnbull, *The Forest People: A Study of the Pygmies of the Congo*, Simon & Schuster, 1961; ISBN 0671640992.
『森の民』(筑摩書房)

  豊富な資源のある社会は、精神的にも満ち足りている。それによって、相互利益のある人間関係や豊かな生活がもたらされる。

- Colin M. Turnbull, *The Mountain People*, Simon & Schuster, 1972; ISBN 0671640984.
『ブリンジ・ヌガグ —— 食うものをくれ』(筑摩書房)

  資源の乏しい社会。飢餓は恐ろしい結末を招く。

- Mary Walton and W. Edwards Deming, *The Deming Management Method*, Perigee, 1988; ISBN 0399550011.
『デミング式経営 —— QC 経営の原点に何を学ぶか』(プレジデント社)

  デミングは、恐怖心がパフォーマンスを阻害することを明確に示した。デミングの著書の多くは統計的品質管理を中心に扱っているが、この本では人間の感情の影響について書かれている。

- Gerald Weinberg, *Quality Software Management: Volume 3, Congruent Action*, Dorset House, 1994; ISBN 0932633285.
『ワインバーグのシステム行動法』(共立出版)

  言っていることとやっていることが違うと、悪いことが起きる。この本は、自分の言動を一致させる方法、他人の言動不一致を認識する方法、その対処法について書かれている。

- Gerald Weinberg, *The Psychology of Computer Programming*, Dorset House, 1998; ISBN 0932633420.
『プログラミングの心理学』(日経 BP 社)

  人が人のためにプログラムを書いていることを最初に認識すべきだ。

- Gerald Weinberg, *The Secrets of Consulting*, Dorset House, 1986; ISBN 0932633013.
『コンサルタントの秘密 —— 技術アドバイスの人間学』(共立出版)

  変化を取り込むための戦略。

プロジェクトマネジメント

- David Anderson, *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*, Prentice Hall 2004; ISBN 0131424602.
『アジャイルソフトウェアマネジメント』(日刊工業新聞社)

  制約理論をソフトウェア開発に適用している。各イテレーションでシステムの制限を取り除く。制限を取り除かない作業はムダだ。

- Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, 2000; ISBN 0201710919.
『XP エクストリーム・プログラミング実行計画』(ピアソン・エデュケーション)

  XP の計画プロセスに関する技術的な詳細。ポイント法よりも実時間見積りのほうが、精度の高い情報であり、レベルの高い説明責任をもたらす。

- Fred Brooks, *The Mythical Man-Month, Anniversary Edition*, Addison-Wesley, 1995; ISBN 0201835959.
『人月の神話』(丸善出版)

  4 つの変数について考えさせられる話。新装版には有名な「銀の弾などない」に関する興味深い議論が収録されている。

- Mike Cohn, *User Stories Applied: For Agile Software Development*, Addison-Wesley, 2004; ISBN 0321205685.

  フィーチャー単位でプロジェクトの計画や追跡をする方法。

- Brad Cox and Andy Novobilski, *Object-Oriented Programming--An Evolutionary Approach, Second Edition*, Addison-Wesley, 1991; ISBN 0201548348.
『オブジェクト指向のプログラミング —— ソフトウエア再利用の新しい方法』(トッパン)

  ソフトウェア開発の電気工学的なパラダイムについて解説している。

- Ward Cunningham, "Episodes: A Pattern Language of Competitive Development," in *Pattern Languages of Program Design 2, John Vlissides, ed.*, Addison-Wesley, 1996; ISBN 0201895277 (also `http://c2.com/ppr/ episodes.html`).
『プログラムデザインのためのパターン言語 —— Pattern Languages of Program Design 選集』（ソフトバンククリエイティブ）収録「エピソーズ：競争力のある開発のパターン言語」

  短いサイクルでのプログラミングに関する議論。

- Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982; ISBN 0131717111.
『品質と生産性を重視したソフトウェア開発プロジェクト技法 —— 見積り・設計・テストの効果的な構造化』（近代科学社）

  ソフトウェアプロジェクトを計測するためのフィードバックの作成方法や活用方法の例。

- Tom DeMarco and Tim Lister, *Waltzing with Bears: Managing Risk on Software Projects*, Dorset House, 2003; ISBN 0932633609.
『熊とワルツを —— リスクを愉しむプロジェクト管理』（日経 BP 社）

  XP ではリスク管理のさまざまな機会を提供しているが、それでもなおリスクを管理しなければいけない。この本には、状況を把握しながらリスクを負うための多くのアイデアが掲載されている。

- Tom Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988; ISBN 0201192462.

  小さなリリース、継続的なリファクタリング、顧客との密接な対話など、進化型デリバリーを強く主張している。

- Ivar Jacobson, Magnus Christerson, Parik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Case Driven Approach*, Addison-Wesley, 1992; ISBN 0201544350.
『オブジェクト指向ソフトウェア工学 OOSE —— use - case によるアプローチ』（トッパン）

  ストーリー（ユースケース）から開発を駆動することの情報源。

- Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999; ISBN 0201571692.
『UML による統一ソフトウェア開発プロセス —— オブジェクト指向開発方法論』（翔泳社）

  UP（統一プロセス）には、短いイテレーション、メタファーの強調、ストーリー駆動の開発が含まれている。UP はドキュメント駆動であり、テスト手順はあまり厳密ではない。

- Philip Metzger, *Managing a Programming Project*, Prentice Hall, 1973; ISBN 0135507561.
『プログラミング・プロジェクト管理』（近代科学社）

私が発見できたなかで最も古いプログラミングプロジェクト管理の本。貴重なものではあるが、考え方は純粋なテイラー主義である。200 ページのなかで保守に触れている部分は、わずか 2 段落しかない。

- Charles Poole and Jan Willem Huisman, "Using Extreme Programming in a Maintenance Environment," in *IEEE Software*, November/December 2001, pp. 42-50.

  あるチームでプロダクトの保守に XP を適用した結果、コストが 60% 削減、欠陥の修正時間が 66% 短縮された。

- Mary Poppendieck and Tom Poppendieck, *Lean Software Development*, Addison-Wesley, 2003; ISBN 0321150783.
  『リーンソフトウェア開発』(日経 BP 社)

  リーン生産方式とリーン製品開発の考えをソフトウェアに適用している。

- Jennifer Stapleton, *DSDM, Dynamic Systems Development Method: The Method in Practice*, Addison-Wesley, 1997; ISBN 0201178893.

  DSDM は、高速なアプリケーション開発の利点を諦めずに、厳格に制御するためのひとつの考え方。

- Hirotaka Takeuchi and Ikujiro Nonaka, "The new new product development game", *Harvard Business Review* [1986], 86116:137-146.

  プロジェクトの規模の拡大を伴う進化的なデリバリーのための合意指向のアプローチ。

- Jane Wood and Denise Silver, *Joint Application Development, 2nd edition*, John Wiley and Sons, 1995; ISBN 0471042994.

  JAD のファシリテーターは、指示をせずにファシリテートに専念し、意思決定の方法を最もよく知っている人に権限を与え、最終的にはその場から姿を消す。JAD では、開発者と顧客が合意できる要件文書を作る。

プログラミング

- David Astels, *Test Driven Development: A Practical Guide*, Prentice Hall, 2003; ISBN 0131016490.

  テスト駆動開発のチュートリアル。

- Kent Beck, *JUnit Pocket Guide*, O'Reilly, 2004; ISBN 0596007434.

  JUnit テスティングフレームワークの紹介と使い方。

- Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1996; ISBN 013476904X.
  『ケント・ベックの Smalltalk ベストプラクティス・パターン —— シンプル・デザインへの宝石集』(ピアソン・エデュケーション)

  小規模な設計とコードによるコミュニケーションのパターン。

- Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002; ISBN 0321146530.
『テスト駆動開発入門』(ピアソン・エデュケーション)

  テストファーストプログラミングとインクリメンタルな設計の入門書。最大の特徴は TDD パターンの一覧。

- Kent Beck and Erich Gamma, "Test Infected: Programmers Love Writing Tests," in *Java Report*, July 1998, volume 3, number 7, pp. 37-50.

  xUnit テスティングフレームワークの Java バージョンである JUnit を使って、自動テストを書いている。

- Jon Bentley, *Writing Efficient Programs*, Prentice Hall, 1982; ISBN 0139702512.
『プログラム改良学』(近代科学社)

  「速度が出ない」問題の治療薬。

- Edward Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976; ISBN 013215871X.
『プログラミング原論 —— いかにしてプログラムをつくるか』(サイエンス社)

  数学としてのプログラミング。ダイクストラはプログラミングを通じて美を探求している。

- Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003; ISBN 0321125215.
『エリック・エヴァンスのドメイン駆動設計』(翔泳社)

  ビジネスの人間と技術の人間が明確なコミュニケーションをするための実践的なロードマップが説明されている。

- Brian Foote and Joe Yoder, "Big Ball of Mud," *Pattern Languages of Program Design 4*, edited by Neil Harrison, Brian Foote, and Hans Rohnert, Addison-Wesley, 2000; ISBN 0201433044.

  インクリメンタルな設計に十分に投資しなかったときに起こること。

- Martin Fowler, *Analysis Patterns*, Addison-Wesley, 1996; ISBN 0201895420.
『アナリシスパターン —— 再利用可能なオブジェクトモデル』(ピアソン・エデュケーション)

  分析の意思決定をするための共通語彙。アナリシスパターンは、ソフトウェア開発に影響を与えるビジネス構造を理解するためのものである。

- Martin Fowler, ed., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999; ISBN 0201485672.
『新装版 リファクタリング —— 既存のコードを安全に改善する』(オーム社)

  リファクタリングの参考書の決定版。

- Martin Fowler and Pramod Sadalage, "Evolutionary Database Design," January 2003, http://www.martinfowler.com/articles/evodb.html.
「データベースの進化的設計」http://www.objectclub.jp/community/XP-jp/xp_relate/evodb-jp

  データベースのインクリメンタルな設計のシンプルな戦略。

- Erich Gamma, Richard Helms, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995; ISBN 0201633612.
『オブジェクト指向における再利用のためのデザインパターン』(ソフトバンククリエイティブ)

  設計の意思決定をするための共通語彙。

- Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004; ISBN 0321213351.
『パターン指向リファクタリング入門 —— ソフトウエア設計を改善する 27 の作法』(日経 BP 社)

  デザインパターンとリファクタリングのギャップを埋めるもの。インクリメンタルな設計について学ぶときに役立つ。

- Donald E. Knuth, *Literate Programming*, Stanford University, 1992; ISBN 0937073814.
『文芸的プログラミング』(アスキー出版局)

  コミュニケーション指向のプログラミング手法。文芸的プログラミングは、保守するのは難しいが、伝えるべきことがどれだけ多いかを思い出させてくれる。

- Steve McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition*, Microsoft Press, 2004; ISBN 0735619670.
『Code Complete 第 2 版(上・下) —— 完全なプログラミングを目指して』(日経 BP 社)

  プロの開発者になるために知っておくべきこと。コーディングにどれだけ注意を払うことができるかを考察している。

- Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997; ISBN 0136291554.
『オブジェクト指向入門 第 2 版 原則・コンセプト』『オブジェクト指向入門 第 2 版 方法論・実践』(翔泳社)

  契約による設計は、ユニットテスティングの代替または拡張である。

- David Saff and Michael D. Ernst, "An Experimental Evaluation of Continuous Testing During Development," in *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis* (Boston, MA, USA), July 12-14, 2004, pp. 76-85.

  継続的テスティングによって、プログラミングの最中に継続的なフィードバックがもたらされる。

その他

- Barry Boehm, *Software Engineering Economics*, Prentice Hall, 1981; ISBN 0138221227.

  ソフトウェアのコストとその理由について考えるための標準的な参考書。

- Stewart Brand, *How Buildings Learn: What Happens After They Are Built*, Penguin Books, 1995; ISBN 0140139966.

  強固な構造のように見えても、成長や変化を遂げている。

- Malcolm Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference*, Back Bay Books, 2002; ISBN 0316346624.
  『急に売れ始めるにはワケがある —— ネットワーク理論が明らかにする口コミの法則』(ソフトバンククリエイティブ)

  いかにしてアイデアが広まるか。

- Larry Gonick and Mark Wheelis, *The Cartoon Guide to Genetics*, HarperPerennial Library, 1991; ISBN 0062730991.
  『漫画 分子遺伝学が驚異的によくわかる』(白揚社)

  情報通信の媒体として、図解がいかに優れているかを示している。

- John Hull, *Options, Futures, and Other Derivatives*, Prentice Hall, 1997; ISBN 0132643677.
  『フィナンシャルエンジニアリング —— デリバティブ取引とリスク管理の総体系』(金融財政事情研究会)

  オプション価格決定の標準的な参考書。

- Nancy Margulies with Nusa Mall, *Mapping Inner Space: Second Edition*, Zephyr Press, 2002; ISBN 1569761388.

  自分の考えをグラフィカルに表現する方法。直線的に思考する人と非直線的に思考する人のコミュニケーションを強化する。

- Taiichi Ohno, *Toyota Production System: Beyond Large-Scale Production*, Productivity Press, 1988; ISBN 0915299143.
  『トヨタ生産方式 —— 脱規模の経営をめざして』(ダイヤモンド社)

  科学的管理法の原則とは対照的な興味深い内容。ビジネスを成就させるという大野氏の感動的な宣言は、関係者全員をリスペクトするという哲学によって達成されたものである。

- Edward Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1992; ISBN 096139210X.

  図を使って数値情報を伝える技法が満載。たとえば、メトリクスのグラフに何を使うべきかが理解できる。それに、本自体も美しい。

訳者あとがき

本書は、"Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley, 2004" の全訳である。訳語の選定については「ソフトウェアテスト標準用語集[†1]」を参考にした。

XP はソーシャルチェンジ

本書は印象的な言葉から始まる。

> エクストリームプログラミング（XP）はソーシャルチェンジである。

この一文が本書のすべてを物語っていると言っても過言ではない。「ソーシャル（*social*）」という言葉は日本語にするのが難しい。辞書で引けば「社会的」という言葉が出てくるが、「社会」という言葉から受けるイメージは「世間」のように広い。XP は必ずしも「社会」を変えるための活動ではない。隣の席にいるプログラマーとうまくコミュニケーションしたり、顧客と密接にやりとりをしたり、ユーザーが安心して使えるソフトウェアを届けたりする。そうした「人間関係」を扱う活動こそが、エクストリームプログラミングである。

なぜ人間関係を扱うのかといえば、成功には「優れた技術力と良好な人間関係が必要」だからだ（第 1 章参照）。XP の大きな特徴は、**この両方を同時に満たそうとしている**ところにある。そのために XP は「価値、原則、プラクティス」という 3 つのレベルの活動指針を用意した（第 3 章参照）。そして、日常的な取り組みであるプラクティスのなかに「ソーシャル」な要素を盛り込んだのだ。

わかりやすいのは**ペアプログラミング**だろう（第 7 章参照）。このプラクティスを実践すれば、欠陥率の減少や生産性の向上といった技術的な改善が期待できる。それと同時に、誰かと一緒に仕事をするという人間の社会的欲求も満たされるのだ。

XP は当たり前

だが、そのようなことを考えていなくても、すでに XP のプラクティスを実践している組織も多いだろう。本書でも述べられているように、今では「ありふれたもの」になったプラクティスも少なくない（「はじめに」参照）。

[†1] http://jstqb.jp/syllabus.html

たとえば、継続的インテグレーションに異論を唱える人はもはやいないだろう。デイリーデプロイどころか、1日に何度もデプロイする世の中になっている。プログラマー以外も当たり前のようにGitなどを使いこなし、コードの共有を実現している。多くの人がコードとテストを書いており、テストファーストプログラミングを実施している人も少なくない。リモートワークが台頭した結果、全員同席のように実現が難しくなったものもあるかもしれないが、チーム全体が大切であることに変わりはなく、多くのプラクティスが今でも有効である。そして、誰もがいきいきとした仕事を求めている。ここに「エクストリーム」なものは存在しない。

XPはパターンランゲージ

すでにXPのプラクティスを実施しているのであれば、「私のチームはエクストリームですか？」と聞きたくなるかもしれない（第21章参照）。ここがXPの難しいところであり、魅力でもあるのだが、XPが提唱する「価値、原則、プラクティス」は、絶対に守らなければいけない「決まりごと」ではない。したがって、何も考えずに、本書に書かれたとおりに機械的にプラクティスを実施していては、XPとはいえないのである。それはむしろケント・ベックが否定する「計画立案と実行作業の分離」に近い（第18章参照）。XPの本来のあり方とは、「価値、原則、プラクティス」のそれぞれを言葉にして、場合によっては利用者であるあなた自身が自分の言葉を生み出しながら、利用することに他ならない。

こうした考え方は、クリストファー・アレグザンダーの「パターンランゲージ」を参考にしたものだ（第23章参照）。XPのプラクティスがすべて命令形（〜すること）で書かれていることからも明らかである。パターンランゲージはXPのみならず、アジャイル開発手法を理解する上で重要なものなので、詳しくは本書の「注釈付き参考文献」に掲載されたアレグザンダーの書籍を読んでもらいたい。また、そこからXPへとつながる流れは、江渡浩一郎氏の『パターン、Wiki、XP』（技術評論社）に詳しく書かれている。

XP（の作者）の現在

最後に、XPの現在……というよりも、作者であるケント・ベックの現在について触れておきたい。現在の彼は、2011年からFacebook社でサーバーサイドの開発（C++/Thrift）を担当している。近況を知りたければ、Facebookでフォロー（もしくは友達申請）するのが一番だろう。TDDやパターンについて、いくつかエントリを書いているので、興味のある人は読んでみてほしい。

Twitterにもアカウントがあるので、こちらもフォローしておこう。それから、Quoraという質問サイトでもよく姿を見かけるので、何か質問があれば問いかけてみるのもよいかもしれない。いずれも名前で検索すれば出てくるはずだ。

また、各種カンファレンスで講演も行っている。たとえば、RailsConf 2015では、クロージングの基調講演を務めていた。リーンスタートアップのカンファレンスにも過去数回登壇している（2010年には「新アジャイルソフトウェア開発宣言」なるものを提唱している）。2014年のAgile Singapore Conferenceでは「Ease at Work」という講演を行っている。それから、DHH（Railsの作者）のブログ記事「TDD is dead. Long live testing.」に端を発する鼎談（ケント・ベック、DHH、マーティン・ファウラー）も記憶に新しいだろう。いずれも動画を検索すれば出てくるので、ぜひともチェックしてほしい。

謝辞

翻訳にあたっては、Re:VIEW[†2]、github、Emacs、xyzzy、PDIC、英辞郎のお世話になりました。特にRe:VIEWについては、翻訳から組版までのすべての工程で活躍しています。関係者のみなさんに感謝します。

日本語版の推薦の言葉を名だたる方々に書いていただきました。どうもありがとうございました。

恩田伊織さん、木下史彦さん、鈴木則夫さん、田島暁雄さん、細谷泰夫さん、的場聡弘さん、菅野洋史さん、髙橋征義さん、懸田剛さん、森雄哉さん、内山滋さん、小芝敏明さん、原田巌さん、髙江洲睦さん、守田憲司さん、木村卓央さんには、翻訳レビューにご協力いただきました。この場を借りて、お礼申し上げます。

編集を担当されたオーム社のみなさんには、大変お世話になりました。いつものプロの仕事を見せていただきました。

最後になりますが、大いなる先輩であり、友人であり、本書を翻訳する機会を作ってくれた角谷信太郎さんに感謝します。

2015年6月

角　征典

[†2] http://reviewml.org

索引

Q

R

S

T

U

W

X

ア

イ

ケ

コ

サ

シ

ス

セ

ソ

タ

チ

テ

ト

ヘ

ホ

マ

ミ

ム

メ

著者と訳者について

＜著者＞

Kent Beck

ケント・ベックは、パターン、テスト駆動開発、エクストリームプログラミングなどのアイデアを提唱して、常にソフトウェアエンジニアリングの定説に挑戦している。現在は、Three Rivers Institute と Agitar Software に所属。Addison-Weskey が出版する多くの書籍の著者である。(原書執筆当時)

Cynthia Andres

シンシア・アンドレスは、心理学の理学士号を持っており、組織行動学、意思決定分析、女性学などの分野で大きな功績を残している。エクストリームプログラミングの誕生のときからケントと一緒にソーシャルな側面に尽力。ケントと同じく Three Rivers Institute に所属している。(原書執筆当時)

＜訳者＞

角 征典（かど まさのり）

1978 年山口県生まれのプログラマー。アジャイル開発の導入支援に従事。訳書に『リーダブルコード』『Team Geek』『Running Lean』『Lean Analytics』『ウェブオペレーション』(オライリー・ジャパン)、『7 つのデータベース 7 つの世界』『アジャイルレトロスペクティブズ』(オーム社)、『エッセンシャルスクラム』(翔泳社)、『プログラマの考え方がおもしろいほど身につく本』『Software in 30 Days』『サービスデザインパターン』『Clean Coder』『メタプログラミング Ruby』(アスキー・メディアワークス)、『Fearless Change アジャイルに効く アイデアを組織に広めるための 48 のパターン』(丸善出版) などがある。

エクストリームプログラミング

2015年6月25日　第1版第1刷発行
2021年5月10日　第1版第3刷発行

著　者　Kent Beck・Cynthia Andres
訳　者　角　征典
発行者　村上和夫
発行所　株式会社 オーム社
郵便番号　101-8460
東京都千代田区神田錦町3-1
電話　03(3233)0641(代表)
URL https://www.ohmsha.co.jp/

印刷・製本　日経印刷
ISBN978-4-274-21762-3　Printed in Japan

好評関連書籍

アジャイルサムライ
達人開発者への道

Jonathan Rasmusson 著／
西村直人・角谷信太郎 監訳／
近藤修平・角掛拓未 訳

A5判 336頁 本体2600円【税別】
ISBN 978-4-274-06856-0

アジャイルプラクティス
達人プログラマに学ぶ現場開発者の習慣

Venkat Subramaniam and
Andy Hunt 著／
角谷信太郎・木下史彦 監訳

A5判 220頁 本体2400円【税別】
ISBN 978-4-274-06694-8

アジャイルレトロスペクティブズ
強いチームを育てる「ふりかえり」の手引き

Esther Derby and
Diana Larsen 著／
角 征典 訳

A5判 192頁 本体2400円【税別】
ISBN 978-4-274-06698-6

リーン開発の現場
カンバンによる大規模プロジェクトの運営

Henrik Kniberg 著／
角谷信太郎 監訳／
市谷聡啓・藤原 大 共訳

A5判 208頁 本体2400円【税別】
ISBN 978-4-274-06932-1

情熱プログラマー
ソフトウェア開発者の幸せな生き方

Chad Fowler 著／
でびあんぐる 監訳

A5判 200頁 本体2400円【税別】
ISBN 978-4-274-06793-8

実践 反復型ソフトウェア開発

津田義史 著

A5判 288頁 本体2800円【税別】
ISBN 978-4-274-06898-0

レジデント初期研修用資料
医療とコミュニケーションについて

medtoolz 著

A5判 248頁 本体2000円【税別】
ISBN 978-4-274-06836-2

ペーパープロトタイピング
最適なユーザインタフェースを効率よくデザインする

Carolyn Snyder 著／
黒須正明 監訳

B5変判 384頁 本体3200円【税別】
ISBN 4-274-06566-9

◎本体価格の変更、品切れが生じる場合もございますので、ご了承ください。
◎書店に商品がない場合または直接ご注文の場合は下記宛にご連絡ください。
TEL.03-3233-0643 FAX.03-3233-3440 http://www.ohmsha.co.jp/