ThoughtWorksアンソロジー

アジャイルとオブジェクト指向による ソフトウェアイノベーション

ThoughtWorks Inc.　著
(株)オージス総研 オブジェクトの広場編集部　訳

O'REILLY®
オライリー・ジャパン

The ThoughtWorks Anthology

Essays on Software Technology and Innovation

Roy Singham Martin Fowler Rebecca Parsons
Neal Ford Jeff Bay Michael Robinson
Tiffany Lentz Stelios Pantazopoulos
Ian Robinson Erik Doernenburg
Julian Simpson Dave Farley
Kristan Vingrys James Bull

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

推薦の言葉

本書はアメリカはいうにおよばず世界を代表するソフトウェア専門のコンサルティング会社であるThoughtWorks社のコンサルタントの皆さんのエッセイの集合体です。実は私の関係する会社もThoughtWorksとまったく同じ「オブジェクト指向とプロセス」という基本技術によって「ソフトウェア開発変革のお手伝いを実践的に行う」目的の会社として立ち上げられている点（残念ながらThoghtWorksの創業は1995年で豆蔵はそれに遅れること4年）でも非常に親近感を抱きます。

最初にThoughtWorksのことを知ったのは私が『UMLモデリングのエッセンス』として翻訳させていただいているUML Distilledの著者でありそれまで独立系コンサルタントであったMartin Fowler氏がThoughtWorksという聞いたことのない小さなコンサル会社のチーフサイエンティストになったらしいと2000年のOOPSLA会場で米国の知人から聞かされたときでした。そのときの印象は「思考の仕事」社という名称をすばらしいと思いやられたと感じたのを覚えています（こう感じたのは形容詞を名前に取り入れた「合理的なソフトウェア」社のとき以来です）。

まさに本書の内容は「思考の仕事」の結晶であり、どのエッセイも個性的なオリジナルのスタイルで書かれていますが、Fowler氏に代表される「理論と実践のバランス」がとれたpracticalな態度というものが一貫して感じられて快く、納得感をもって読まされてしまうものばかりです。

たぶん最初に置かれた創業者兼会長のRoy氏の『ビジネスソフトウェアのラストマイル』エッセイが全体の基調をなすように読者にThoughtWorks社の存在理由を呼びかけるような形で仕組まれているのですが、ここからは今後われわれIT業界に身をおくものとして何を解決していかねばならないのか、どのようなスタンスで今後のソフトウェアビジネスに関わっていくべきなのか、考えるきっかけを与えてくれます。その後に配置された他のエッセイもプログラマ・アナリスト・マネージャ・

テスターと異なる立場からこの「ラストマイル」問題への実践的な取り組みの戦果報告があくまでも中間報告としてなされています。自分がどのエッセイを読むべきかは、副社長 Mike 氏の「まえがき」に的確な要約があるのでぜひ参考にして下さい。

このようなさまざまな立場・視点・スタイルのエッセイの翻訳に当たられたオブジェクトの広場関係者のみなさんは大変だったのではないかと思いますが、今度はわれわれが読者から執筆者にならなければならないと強く思います。私の属する豆蔵も含めて日本からこのようなビジネス視点と技術視点がうまく織り込まれた「読むに値する」アンソロジーが発表できるようにぜひがんばろうという強い気持ちをいだきつつ、推薦の辞といたします。

2008 年 11 月 13 日
株式会社豆蔵 取締役
プロフェッショナル・フェロー
羽生田 栄一

賞賛の声

本書では、技術の深さ、詳細さ、斬新なアイデアや研究成果の量は各エッセイで異なります。しかし、すべてのエッセイはきわめて実践的だという点で共通しています。著者らは、「考えさせられる」にもかかわらず「すぐさま適用できる」という、長く目にしていない偉大な仕事を成し遂げています。

—— Stefan Tilkov（innoQ CEO）

IT業界では長い間、カスタムソフトウェアの開発はとても難しく、コストがかかると信じられてきました。このアンソロジーでは、あえてそれに取り組んでいる企業が有する、多様な視点と観点を垣間見ることができます。

—— W. James Fischer（Accenture 元CTO／シニアパートナー（引退））

例えばCruiseControlのようにとても成功したオープンソースのプロジェクトから、ブログやカンファレンスで公開されたアイデアまで、おそらく開発の現場でThoughtWorks社の影響力を実感したことがあるでしょう。とはいえ、外野の私たちには、ThoughtWorks社の中でどのような議論が行われているのかは知る由もありませんでした。しかし、本書を読めば、カーテンを開けて議論に参加することができます。一段上の開発者になれるのです。

—— Nathaniel T. Schutta（著述家、スピーカー、講師）

ソフトウェア開発はいろいろな意味で団体競技であり、リーダーがソフトウェア文化を形成します。成功している組織はそうした文化をたいていドキュメント化しないため、他の人はそれから何も得られません。ここに、ThoughtWorks社のリーダーたちのエッセイがあります。この興味深いエッセイ集を読むと、ThoughtWorks社

の文化をひと目見ることができます。

—— Dave Thomas（Bedarra Research Labs）

ソフトウェア開発に関する最高の知見は、現実に顧客のために問題を解決している人々からもたらされます。しかし、散在するブログをくまなく探索でもしないかぎり、そうした知見を手に入れるのはほぼ不可能でした。ThoughtWorker たちは、過去 10 年にわたり現実にある多くの問題を解決してきました。だからこそ、そこから得た英知の結集のスナップショットをこの手にして、私は本当にうれしく思います。

—— Gregor Hohpe（"*Enterprise Integration Patterns*" 共著者）

今日の厳しい IT 業界においてソフトウェアを開発するための、言語やツールの正しい使い方を取り上げた、実に素晴らしいエッセイ集です。本書の著者らこそが、ソフトウェア世界のベテラン中のベテランと呼ぶにふさわしい人たちです。

—— Terence Parr（サンフランシスコ大学 ANTLR プロジェクトリーダー）

ThoughtWorks 社は傑出した仕事をしました。エッセイ集をまとめ、定評ある経験と知恵の一部を私たちに提供してくれたのです。本書は、たびたび引き合いに出され、どのプロジェクトの本棚にも並べられるような書籍です。

—— Jeff Brown（G2One 北米担当役員）

まえがき

ThoughtWorks社は、カスタムアプリケーションと妥協のないコンサルティングを提供する、情熱的で意欲を持った、知的な人々からなる集団です。ThoughtWorkerに、この会社のどこが一番気に入っているかを尋ねてみてください。彼らは、他のThoughtWorkerと出会い、一緒に働き、彼らから何かを学べることだ、と答えるでしょう。私たちは、さまざまな文化、民族、教育のバックグラウンドを持ったギーク、マネージャ、アナリスト、プログラマ、テスト担当者、運用技術者で構成されています。こうしたバックグラウンドと視点の多様さとともに、私たちは新しいアイデアに対する情熱を共有しています。その2つが結びつくことで、非常に活発な議論が生み出されています。

私たちの作り上げた会社は、6か国に散らばり、1,000人近くの聡明で独自の意見を持った人々からなります。組織構造にはほとんどヒエラルキーがなく、そして熱狂的なまでに透明性にこだわっています†。私たちの会社は成功を収めています。もちろん、私たちが定義する成功とは、典型的な意味での成功とは異なります。成功とは、顧客を満足させ、ソフトウェア業界に影響を与え、そして社会にも影響を与えるものでなければいけません。私たちは、高い目標を掲げているのです。

ブロゴスフィアやカンファレンス会場、Web上、書籍で、多くのThoughtWorkerの声を聞くことができます。それというのも、これまでの自分たちの成果や手法について容赦なく批評することも、最高のものを目指そうとする私たちの姿

† 訳注：ThoughtWorks創業者のRoy Singhamは、CNNの取材に対して「中国オフィスの清掃員がCEOと同じくらい戦略的に価値があるような、フラットな会社を作りたい」と答えている。また、会社の透明性については、「全従業員の給料を公開することを検討したことがある」とも答えている（http://money.cnn.com/2008/03/14/technology/kirkpatrick_thoughtworks.fortune/index.htmより）。

勢の一部だからです。そうした批評によって、次なる改善を探究しているのです。私たちは、簡単には満足しようとしません。そして自分たちが学んだことは、ぜひ他の人にも伝えたいと思っています。

私たちの歴戦の経験は、ドメイン、テクノロジ、プラットフォームの異なる無数のプロジェクトから得たものです。私たちは自分たちの仕事について本当にたくさんのことを考えますが、その考えは数多くのソフトウェアを人々に提供してきた開発現場に根ざしたものです。ソフトウェア開発の仕事に対する純粋な思いから、私たちはこれまでソフトウェア開発に集中してきたのです。

IT コンサルタントは、新しい人事管理ポリシーの検討ミーティングに出席するために雇われるわけではありません。したがって、ほとんどの IT プロフェッショナルと比較して、私たちの仕事ははるかにソフトウェアの提供に関することに焦点が絞られています。その結果が、現実主義と技術的正確さを兼ね備えた私たちの姿勢につながっています。

このアンソロジーは、ThoughtWorker が取り組んでいる信じられないほど多様な IT の諸問題についての、優れたスナップショットを紹介するものです。私たちはこのアンソロジーを、単によりよいソフトウェアの作成方法を紹介するだけの書籍でなく、それ以上の内容にしようと努めました。組織の IT 投資に見合った、真のビジネス価値を実現するための課題に取り組んでいます。このアンソロジーは、基調テーマをなす Roy のエッセイにより幕を開けます。Roy は、総力を結集して、システムを稼働環境へ移行する「ラストマイル」に変化をもたらそうと呼び掛けます。Roy の計画は壮大で野心的です。それは、「ラストマイル」における運用やデプロイの課題を、要求定義やコーディングと同じように開発プロセスの中心課題に据えることに他なりません。プロジェクトの成功とは何かを思い出してください。単に、開発したコードが品質保証（QA：Quality Assurance）部門の審査を通り、壁の向こう側にいる運用チームに投げ渡されて、本番運用やデプロイなどをやってもらう用意ができた、ということではないはずです。それを思い出せば、ソフトウェアが稼働するのを見届けるまでは、ソフトウェア構築チームの仕事は「完了」していないことがわかります。そして、Roy の提唱は、プロジェクトの完了と成功の意味を鋭い切り口で定義し直すだけにとどまりません。彼は、利害関係者に参加してもらう方法とタイミングについても再考を求めるのです。これまでコーディングプロセスのツール（例えば自動ビルド、自動テスト、リファクタリングのツール）を改善するために考え出されてきた優れたノウハウは、「ラストマイル」問題に立ち向かう際にも応用できます。

本書を読み進めるにつれて、Roy の呼び掛けに対する返答に繰り返し出会うで

しょう。例えば、Jamesは性能テストで返答します。性能テストは、プロジェクトの後半になるまで忘れられ、先延ばしにされることが常態化しています。プロジェクトも後半になれば、非常に多くの設計判断がコード中に作り込まれてしまっています。性能上の問題を解決するために、苦労して作り上げた業務機能を壊さずにそうした設計判断を元に戻そうとするのは、まるで熱力学第二法則†に逆らおうとするようなものです。Jamesは、きわめて実践的なアプローチを採用します。前もって性能要求を定義しておくことが必要だ、などと当たり前な主張をする代わりに（それをわかっていない人などいるでしょうか）、利害関係者から有用な性能要求を引き出す方法について議論します。単に「テストを早くから開始せよ！」などと言う代わりに、実際には、どこで、どうやって性能テストを実施できるかを検討します。

JulianはAntリファクタリングで返答します。多数の標準的なリファクタリングをカタログ化し、それぞれのリファクタリングについて明快な例を提示します。Julianのエッセイは、日々成長し、進化するビルドスクリプトを扱っている人にとって、優れた参考書となります。Daveのエッセイは、Royによる巻頭のエッセイと対称をなす、素晴らしい巻末の結びを提供します。Daveは1クリックデプロイについて、その概念的枠組みの概略を示します。彼は、ビルドされた巨大で扱いにくいバイナリをいかに管理するか、多種混在するデプロイ環境の中でいかにシステムを統合していくかといった、大きな問題に取り組みます。ビジネスソフトウェア開発を効率化するテクニックはすべて、最終的にはデプロイツールの中に組み込まれるでしょう。Daveのエッセイは、それをいち早く実行に移すものです。

Steliosのエッセイは、プロジェクトの健康状態を伝えるコミュニケーション技術で返答します。彼はプロジェクトのメトリクスを、客観的なものから主観的なものまで、いくつか提案します。そして、関係者全員が毎日同じ「ダッシュボード」††を見ながら作業できるように、メトリクスを効果的に表示する方法を議論します。彼は、プロジェクトの生体情報を、可能なかぎり多くの利害関係者に見えるようにしようとしています。この試みは、もう1つの概念へとつながります。プロジェクト人類学とでも言うべき概念です。Tiffanyのエッセイは、まるでサモアでの発見

† 訳注：いわゆるエントロピー増大の法則で、熱は高温から低温へしか移らないという法則。閉じた系は、必ず秩序状態から無秩序状態に変化していくことを示す。

†† 訳注：ダッシュボードとは、本来は自動車の運転席前面のスピードメーターなどの計器類が配置されている場所をいう。ダッシュボードが自動車の状態をひと目で把握できることから、一般にものごとの状態を視覚的に把握するためのツールの意味にもなる。なお、7章では「情報発信器」という用語が使われる。

を報告するマーガレット・ミード†のようです。イテレーションマネージャというまったく新種のプロジェクトチームメンバと遭遇した彼女は、そのメンバがどのようにチームという種族の中に組み入れられていくのかを教えてくれます。彼女は、チームの組織方法を少し変えるだけで、チームの活動がもっと効果的になる可能性を見出します。そして、新しい方法で組織されたチームの働きを助ける、新たな役割を私たちに示すのです。Jeffによる「9つのルール」のエッセイは、プログラミング道の師範が弟子にその真髄を教える様子を想起させます。そのルールは簡潔で優雅でありながら、同時に苛立たしいほどに従うことが困難です（それは特に、ルールに従うにはプログラマが非常に多くの習慣を「忘却」しなければいけないからです）。Rebeccaのエッセイは、「言語論争」の問題に確固たる姿勢で歩み寄っていくかのようです。エッセイは、さまざまな言語を分類していく魅力的な読み物として始まります。初めのうちは、リンネ††が庭園をそぞろ歩きしながら、植物を眺めてその特性を調べ、そして彼がいずれ考え出すことになる植物の分類体系へと一般化させていくかのような印象を受けます。Rebeccaは、言語論争に対する優れた議論の基礎を打ち立てます。しかし、驚くのは最後です。このエッセイは、現在人気の言語をいくつか紹介する単なる概説講座ではありません。そうではなく、世の中にあるプログラミング言語がいかに多様かを示し、巷の「Java対.NET」言語論争は永遠に決着のつかない議論を繰り返しているだけにすぎないことを証明するのです。しかし重要なことは、どんな種類の問題を解こうとしているのか、そして問題に取り組むためにどんな種類のプログラミング言語を利用できるのか、を知ることです。Rebeccaの文章はまるで、散らかった工房に入ってレンチや金づちをきれいに選り分けていき、その用途がわかるようにラベルを貼って引き出しの中へ整理していくかのようです。

残りのエッセイはより技術的な話題になりますが、ここでもThoughtWorksの才能あふれる同僚たちがその多様性をいかんなく発揮しています。Ianは、顧客ではなく利用者駆動によるSOAの契約について検討するための、包括的なアプローチを説明します。変化するビジネス要求に長期にわたり適応しなければいけない共有サービスをいかに構築し、進化させていくか、特にそれを既存のサービス利用者に不自由を感じさせることなく行うにはどうすればいいか、というのは永遠の課題で

† 訳注：マーガレット・ミード（Margaret Mead）は米国の文化人類学者で、ジェンダー研究の先駆者として知られる。サモア諸島でのフィールドワークをまとめた著書『サモアの思春期』（蒼樹書房）が有名。

†† 訳注：カール・フォン・リンネ（Carl von Linné）は18世紀の植物学者。界・門・綱から始まる近代的な生物の分類体系の基礎を作り、「分類学の父」と呼ばれる。

す。彼のエッセイは、その課題に一石を投じるものです。そしてErikも、同様の問題を検討します。よく設計されたシステムでは、ドメインモデルはインフラストラクチャレイヤから分離されます。しかしそのためには、インフラストラクチャレイヤがドメインモデルの中のメタデータを活用しなくてはいけません。プログラム上の情報、例えばドメイン要素を表現するために選んだクラスの種類といったものから、暗黙的なメタデータをかき集めてくることも可能です。しかしそれだけでは、バリデーションのような本当に役立つことを行うのに十分な情報を提供できません。JavaやC#といった最近の言語では、アノテーションや属性といった形でより潤沢なメタデータを提供することができます。Erikはケーススタディを通して、そうした最新のメタデータを活用する方法を探究します。そしてMartinのエッセイは、邪悪な誇大妄想者のためにさまざまなDSLを駆使するという遊び心溢れるものですが、私は自分がはるか昔にCを学び始めた頃のことを思い出しました。当時の私の参考書はK&R（Kernighan & Ritchie）†でした。K&Rの中で、文字列コピーの関数が数回のイテレーションを経て簡潔で優雅な形へ書き替えられていくのを読み、私は感動しました。K&Rのおかげで、これから直面しそうなプログラミング上の障害がすべて解決されたかのように思われたものです。

エッセイとエッセイをつなぐ糸は、アンソロジーの至るところに張り巡らされています。これらのエッセイは、IT問題の生態系を深く探索しながらも、同時に、一見して明らかな形や予想もしない驚くべき形で互いに結びついているのです。本書が幅広い話題や多様な問題解決のアプローチに満ちていることは、執筆陣が所属する組織がアイデアを創出する健全な環境であることを反映するものです。私たちが生み出したアイデアの一端がこのような選集の形になるのを目にしながら、自分たちの可能性をもっと追究してみたいという思いに、私は強く駆られています。

2008年2月15日

Mike Aguilar（ThoughtWorks副社長）

† 訳注：邦訳は『プログラミング言語C ANSI規格準拠 第2版』（共立出版）。C言語の設計者であるBrian W. KernighanとDennis M. Ritchieによって書かれた、C言語の古典的な定番書籍。

本書の表記

本書では、以下の表記を使用しています。

太字（Bold）
重要な用語を示します。

等幅（Constant Width）
サンプルコード、コマンド、変数、属性、関数、クラス、名前空間、メソッド、モジュール、値、ファイルの内容、コマンドの出力などを示します。

等幅の斜体（*Constant Width Italic*）
ユーザが各自の環境に応じて入力する必要がある要素部分を示します。

等幅の太字（Constant Width Bold）
コードの重要な部分と、そのとおりに打ち込まなければならないコマンドやテキストを示します。

サンプルコード

サンプルコードは次のサイトからダウンロードできます。

http://www.pragprog.com/titles/twa/source_code（原書）

意見と質問

本書（日本語翻訳版）の内容については、最大限の努力をもって検証および確認していますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植に気づかれることもあるでしょう。本書を読んで気づいたことは、今後の版で改善できるように知らせていただければ幸いです。将来の改訂に関する提案なども歓迎します。

株式会社オライリー・ジャパン
〒160-0002　東京都新宿区坂町26番地27　インテリジェントプラザビル1F
電話　　03-3356-5227
FAX　　03-3356-5261
電子メール　japan@oreilly.co.jp

本書に関する技術的な質問や意見については、次の宛先に電子メール(英文)を送ってください。

info@nostarch.com

本書のWebページには、正誤表、サンプルコード、追加情報が掲載されています。以下のアドレスでアクセスできます。

http://www.oreilly.co.jp/books/9784873113890/
http://www.ogis-ri.co.jp/otc/hiroba/ogisbooks/TWAja/
http://oreilly.com/catalog/9781934356142/（原書）
http://www.pragprog.com/titles/twa/thoughtworks-anthology（原書）

オライリーに関するその他の情報については、次のオライリーのWebサイトを参照してください。

http://www.oreilly.co.jp

目次

推薦の言葉 v
賞賛の声 vii
まえがき ix

1章 ビジネスソフトウェアの「ラストマイル」を解決する 1
1.1 「ラストマイル」問題とは何か 1
1.2 問題の本質を理解する 2
1.3 「ラストマイル」問題を解決するには 4
1.4 人 5
1.5 自動化 6
1.6 非機能要求テストを自動化するための設計 7
1.7 設計を稼働環境から分離せよ 9
1.8 バージョンのないソフトウェアへ 10

2章 とある秘密基地とRubyによる20のDSL 13
2.1 秘密基地の例 13
2.2 グローバル関数の利用 17
2.3 オブジェクトの利用 20
2.3.1 クラスメソッドとメソッドチェイン 20
2.3.2 Expression Builder 23
2.3.3 さらなるメソッドチェイン 26
2.4 クロージャの利用 28

2.5 コンテキストの評価 29
2.6 リテラルコレクション 32
2.6.1 可変長引数メソッド 38
2.7 動的受信 39
2.8 まとめ 41

3章 言語の緑豊かな景観 43
3.1 イントロダクション 43
3.2 言語の標本 43
3.3 種類の多種性 47
3.3.1 パラダイム 48
3.3.2 型特性 49
3.3.3 実行時の振る舞い 50
3.3.4 実装モデル 51
3.4 言語の系統樹 52
3.5 どれも興味深いことであるが、どうして重要なのか 53

4章 多言語プログラミング 55
4.1 多言語プログラミングとは 56
4.2 Groovy の方法でファイルを読む 57
4.3 JRuby と isBlank 58
4.4 Jaskell と関数プログラミング 60
4.5 Java のテスト 63
4.6 多言語プログラミングの未来 65

5章 オブジェクト指向エクササイズ 67
5.1 ソフトウェア設計を改善する 9 つのステップ 67
5.2 エクササイズ 68
5.2.1 9 つのルール 69
5.2.2 ルール 1：1 つのメソッドにつきインデントは
1 段階までにすること 69
5.2.3 ルール 2：else 句を使用しないこと 71
5.2.4 ルール 3：すべてのプリミティブ型と文字列型をラップすること .. 73

5.2.5 ルール4：1行につきドットは1つまでにすること 73
5.2.6 ルール5：名前を省略しないこと 75
5.2.7 ルール6：すべてのエンティティを小さくすること 76
5.2.8 ルール7：1つのクラスにつきインスタンス変数は
2つまでにすること 76
5.2.9 ルール8：ファーストクラスコレクションを使用すること 78
5.2.10 ルール9：Getter、Setter、プロパティを使用しないこと 79
5.3 まとめ 79

6章 ところでイテレーションマネージャとは何だろうか 81
6.1 イテレーションマネージャとは何だろうか 81
6.2 よいイテレーションマネージャとは 82
6.3 何がイテレーションマネージャでないか 84
6.4 イテレーションマネージャとチーム 84
6.5 イテレーションマネージャと顧客 86
6.6 イテレーションマネージャとイテレーション 87
6.7 イテレーションマネージャとプロジェクト 88
6.8 まとめ 89

7章 プロジェクトバイタルサイン 91
7.1 プロジェクトバイタルサインとは 91
7.2 プロジェクトバイタルサイン vs. プロジェクトヘルス 92
7.3 プロジェクトバイタルサイン vs. 情報発信器 92
7.4 プロジェクトバイタルサイン：スコープバーンアップ 93
7.4.1 スコープバーンアップの情報発信器の例 93
7.4.2 スコープの測定単位を定義する 94
7.4.3 中間マイルストーンによってボトルネックを発見する 94
7.4.4 スコープバーンアップ図の詳細説明 95
7.5 プロジェクトバイタルサイン：提供品質 96
7.5.1 提供品質における情報発信器の例 96
7.5.2 バグカウント図の詳細説明 97
7.6 プロジェクトバイタルサイン：予算バーンダウン 98
7.6.1 予算バーンダウンの情報発信器の例 98

7.6.2 予算バーンダウン図の詳細説明 99
7.7 プロジェクトバイタルサイン：開発実況 99
7.7.1 開発実況における情報発信機の例 99
7.7.2 開発状態の定義 100
7.7.3 ストーリーボードとストーリーカードの詳細説明 101
7.8 プロジェクトバイタルサイン：チーム知覚 101
7.8.1 チーム知覚の情報発信器の例 101
7.8.2 チームムード図の詳細説明 102

8章 コンシューマ駆動契約——サービス進化パターン 103
8.1 サービスの進化：例 105
8.2 スキーマのバージョニング 106
8.2.1 拡張点 108
8.3 互換性に影響する変更 112
8.3.1 Schematron 113
8.4 コンシューマ駆動契約 115
8.4.1 プロバイダ契約 115
8.4.2 コンシューマ契約 118
8.4.3 コンシューマ駆動契約 120
8.4.4 契約の特性の要約 121
8.4.5 実装 121
8.4.6 メリット 122
8.4.7 コンシューマ駆動契約と SLA 122
8.4.8 課題 123
8.4.9 まとめ 124

9章 ドメインアノテーション 127
9.1 ドメイン駆動設計とアノテーション 127
9.1.1 ドメイン固有メタデータ 128
9.1.2 Java のアノテーションと .NET の属性 129
9.1.3 ドメインアノテーションとは 131
9.1.4 ドメインアノテーションの実例 132
9.2 ケーススタディ：リロイのトラック 134

9.2.1 ドメインモデル 134
9.2.2 データの分類 136
9.2.3 ナビゲーションのヒント 142
9.3 まとめ 149

10章 Antビルドファイルのリファクタリング 151
10.1 イントロダクション 151
10.1.1 リファクタリングとは。Antとは 151
10.1.2 いつリファクタリングをすべきか。あるいは、いつ逃げ出すべきか 152
10.1.3 build.xmlはリファクタリング可能か 153
10.2 Antリファクタリングカタログ 153
10.2.1 macrodefの抽出 155
10.2.2 ターゲットの抽出 157
10.2.3 宣言の導入 159
10.2.4 依存によるcallの置き換え 160
10.2.5 プロパティによるリテラルの置き換え 161
10.2.6 filtersfileの導入 162
10.2.7 プロパティファイルの導入 164
10.2.8 ターゲットのラッパービルドファイルへの移動 165
10.2.9 descriptionによるコメントの置き換え 166
10.2.10 デプロイ用コードのimport先への分離 167
10.2.11 要素のantlibへの移動 168
10.2.12 filesetによる多数のライブラリ定義の置き換え 171
10.2.13 実行時プロパティの移動 171
10.2.14 IDを用いた要素の再利用 173
10.2.15 プロパティのターゲット外部への移動 174
10.2.16 locationによるvalue属性の置き換え 176
10.2.17 build.xml内へのラッパースクリプトの取り込み 177
10.2.18 tasknameの追加 178
10.2.19 内部ターゲットの強制 180
10.2.20 出力ディレクトリの親ディレクトリへの移動 180
10.2.21 applyによるexecの置き換え 181

10.2.22 CI Publisher の利用 182
10.2.23 明確なターゲット名の導入 182
10.2.24 ターゲット名の名詞への変更 183
10.3 まとめ 185
10.4 参考文献 185
10.5 リソース 186

11章 1クリックデプロイ 187
11.1 継続的ビルド 187
11.2 継続的ビルドを越えて 188
11.3 ライフサイクル全体の継続的インテグレーション 189
11.3.1 バイナリの管理 189
11.4 チェックインゲート 191
11.5 受け入れテストゲート 193
11.6 デプロイの準備 194
11.7 その後のテスト 196
11.8 プロセスの自動化 197
11.9 まとめ 198

12章 アジャイルかウォーターフォールか
——エンタープライズ Web アプリケーションのテスト 199
12.1 イントロダクション 199
12.2 テストのライフサイクル 200
12.3 テストの種類 203
12.3.1 単体テスト 204
12.3.2 機能テスト 204
12.3.3 探索的テスト 205
12.3.4 統合テスト 206
12.3.5 データの妥当性確認 207
12.3.6 ユーザ受け入れテスト 207
12.3.7 性能テスト 208
12.3.8 非機能テスト 209
12.3.9 回帰テスト 209

12.3.10 本番検証 210
12.4 環境 211
12.4.1 開発統合環境 211
12.4.2 システム統合環境 212
12.4.3 ステージング環境 213
12.4.4 本番環境 213
12.5 課題管理 214
12.6 ツール 215
12.7 レポートとメトリクス 216
12.8 テストのロール 216
12.8.1 テスト分析 217
12.8.2 テスト記述 218
12.8.3 テスト実行 218
12.8.4 環境管理 218
12.8.5 課題管理 219
12.8.6 トラブルシューティング 219
12.9 参考文献 219

13章 実践的な性能テスト 221
13.1 性能テストとは何か 221
13.2 要求収集 222
13.2.1 何を測定するのか 222
13.2.2 どのように数値を設定するか 223
13.2.3 要求収集を通常のソフトウェア開発プロセスに
どのように適合させるのか 223
13.2.4 開発者も性能テストによって要求を持たないのか 224
13.2.5 性能要求の利害関係者を見つけられない場合はどうなるか 224
13.2.6 顧客があまり技術に詳しくなく、不可能と考えられる
ことを要求する場合はどうなるか 225
13.2.7 なぜビジネスアナリストにもこれらの要求収集に
参加させないのか 226
13.2.8 まとめ 226
13.3 テストの実行 226

13.3.1 どのようなテストを繰り返し行うのか 226
13.3.2 いつテストすべきか 227
13.3.3 どのような環境でテストすべきか 228
13.3.4 性能の低いテスト機でのテスト結果をどのように本番環境に関連づけるか 229
13.3.5 性能テストに適したデータベース容量はどのくらいか 230
13.3.6 サードパーティインターフェイスをどのように扱うか 231
13.3.7 何種類のテストケースが必要か 231
13.3.8 レスポンスタイムとスループットに数種類の測定方法をとるのはなぜか 232
13.3.9 システムのすべての機能をテストする必要があるのか 232
13.3.10 まとめ 233
13.4 コミュニケーション 233
13.4.1 知る必要があるのは誰か 233
13.4.2 つまり、単にレポートを作成すればよいということか 234
13.4.3 まとめ 234
13.5 プロセス 234
13.5.1 すべてをどのようにつなげるのか 235
13.5.2 遅れを防ぐためにはどうすればよいか 235
13.5.3 問題が発見されたとき、対策漏れがないようにするにはどうすればよいか 236
13.6 まとめ 236

参考文献 237
訳者あとがき 239
索引 243

1章 ビジネスソフトウェアの「ラストマイル」を解決する

Roy Singham ◎創業者兼会長
Michael Robinson ◎テクノロジプリンシパル

テスト駆動設計（TDD：Test-Driven Design）、継続的インテグレーション（CI：Continuous Integration）、ペアプログラミング、リファクタリングといったアジャイルプラクティスのおかげで、私たちは高品質のソフトウェアをすばやく提供できるようになりました。提供されるソフトウェアは、プロジェクトの最初期から確実に動作するばかりでなく、要求の進化に対応して修正が加えられていく中でも、確実に動作し続けるのです。

しかし、多くの課題も残っています。特に、ソフトウェア開発の「ラストマイル」が大きな課題です。この「ラストマイル」というのは開発プロセスにおけるフェーズの1つであり、ソフトウェアが機能要求を満たすところまで完成していながら、まだ稼働に入らず、企業に対して価値を提供し始めていない段階に相当します。

ソフトウェア開発者、特に納期へのプレッシャーを感じているソフトウェア開発者は、「ラストマイル」を軽視してしまいがちです。しかし、このフェーズはビジネスソフトウェアを提供する上で、最大のボトルネックになりつつあるのです。

1.1 「ラストマイル」問題とは何か

「ラストマイル」問題は、企業の成長にともなって発生する傾向があります。まずは、何もないところから斬新なビジネスモデルのアイデアを基に、小さなスタートアップ会社を起業するとします。システム、取引データ、顧客、収益がまったくない中で、必要なのはビジネスモデルの成長性を証明するためのシンプルなシステムだけです。もしそこでビジネスモデルが成功すれば、システムの機能と規模を拡大するためにさらなる投資をするでしょう。この時点では、資金にも限りがあるため、できるだけ早くシステムを稼働させて結果を出す必要があります。

本番稼働への迅速なデプロイを実現する上で、上記のシナリオは理想に近い状況です。ここには「ラストマイル」問題はありません。この場合、ビジネス要求さえ満たされれば、ソフトウェアはすぐにでも稼働に入ることができます。

時が経過し、スタートアップ会社は成功し、利益を生むようになりました。会社はたくさんの顧客を持っており、2年分の取引データを蓄積しています。会社は顧客情報管理（CRM：Customer Relationship Management）システムと会計パッケージシステムを購入し、両システムともに中核ビジネスシステムと統合しました。

最初の中核ビジネスシステムには何度か機能拡張が行われましたが、ここに至って新たなビジネスチャンスが到来しており、その実現のために2つめの中核ビジネスシステムを開発する必要が生じています。2つめのシステムも、既存のシステム群と統合する必要があります。

この時点で、システム開発の仕事は多少複雑化してきています。2つめの中核ビジネスシステムを稼働段階に移す前に、レガシーなシステムやデータに対する動作が信頼できるものかどうかをテストする必要があるからです。

さらに時が経過し、会社は株式公開され、多角経営で多国籍の大企業へと成長しました。多くの事業分野と市場を抱えています。会社には、何万人もの従業員と何百万人もの顧客がいます。収益の規模は大きく、いまだに成長を続けていますが、投資家や市場アナリストたちから厳しい視線も向けられています。いちばん最初の中核ビジネスシステムには、今や8年分もの取引データ履歴があり、12もある他のビジネス基幹システムと統合されています。システムは長年にわたり修正パッチと拡張が加えられ続けてきましたが、これ以上の取引量増加とビジネス要求の変化に対応するには限界が来ています。会社は、この古いシステムに代わり、最新の技術を活用した新しいシステムを一から再構築したいと考えています。

これが、「ラストマイル」問題を抱えたソフトウェアです。

1.2 問題の本質を理解する

企業が新しいソフトウェアの開発に投資するのは、そのソフトウェアが新たなビジネス価値をもたらすからです。しかし、以下のようなことが起こると、新しいソフトウェアが重大な財務上の問題を引き起こすこともあります。

- 新しいソフトウェアが、ビジネスモデルの実現に必要なだけのユーザ数や取引量に対応できない

- 新しいソフトウェアが、レガシーデータベース上のデータ整合性を破壊してしまう
- 新しいソフトウェアが突然ダウンしたり、接続しているレガシーシステムを不安定にしたりしてしまう
- 新しいソフトウェアから、機密情報が信頼できない個人や組織に漏洩してしまう
- 新しいソフトウェア上で、悪意のあるユーザが権限のない操作を実行できてしまう

新しいソフトウェアがもたらすはずのビジネス価値が、リスクにさらされるビジネス価値があることで簡単に吹き飛んでしまうことさえあります。そのため、企業が成長して大きくなればなるほど、新しいソフトウェアの導入には慎重にならなければいけません。それが原因となって、古いシステムや技術は新しいものに置き換えられずに、稼働環境にそのまま積み上がっていくのです。レガシーシステムが堆積して統合作業が困難なものになると、今度はそれが、新しいソフトウェアを導入するコストやリスクを押し上げることになります。これが延々と続いていくのです。

この悪循環が原因で、企業が成長し歴史を重ねるに従って、変わりゆくビジネスモデルの要求に適応することがますます難しく、コストのかかるものになっていきます。新しい企業はこのような技術的な負担がないおかげで、ある時点までは歴史のあるライバル企業に競り勝つことができます。しかし、いずれは同じ問題に直面することになるでしょう。

アジャイルソフトウェア開発は、変わりゆく要求にすばやく適応することを可能にします。情報技術、ユビキタスインターネット、グローバル化がビジネスモデルの革新を大きく加速させている今、変化に適応する能力は企業にとってますます魅力的なものになっています。

しかし、ビジネススポンサーの観点からすると、ソフトウェア開発にはエンドツーエンドなプロセスとしての意義しかありません。エンドツーエンドというのは、予算が承認されてから、ソフトウェアが実際にビジネスの中で稼働するまでのことを指します。ビジネススポンサーは、プロセスの途中がどうなっていようがたいして興味はありません。したがって、ビジネススポンサーにとってアジャイルプロセスといえるのは、ソフトウェアを稼働状態にまですばやく導入できるものだけなのです。

複雑なレガシーシステムをいくつも持つ歴史のある大企業では、新しいソフトウェアのリリースをインストールし、テストし、安定稼働させて、そのソフトウェ

アが安全だと十分に確かめるまでに、3、4か月またはそれ以上の時間がかかってしまうでしょう。こうした環境にあっても、アジャイル開発者はビジネス側から要望された新機能を1、2週間で実装できるかもしれません。しかし、企業が定めるリリースサイクルのタイミングや機能凍結の実施方法によっては、その機能がビジネスで実際に使われるようになるまで半年かかってしまうこともあります。

これを、アジャイル開発のサクセスストーリーと捉えたくなる気持ちもわかります。たったの2週間で完成してしまったのですから。そして、その後の6か月間の遅延は、他の誰かのせいにして片付けたくなるかもしれません。しかし、それは誤った見方といえます。

1.3 「ラストマイル」問題を解決するには

今日のアジャイルソフトウェア開発プロジェクトは、一般に以下のようなものになるでしょう。

1. ビジネススポンサーがソフトウェア開発の必要性を認識する
2. ビジネススポンサーがソフトウェア開発予算を承認する
3. 開発チームがストーリー†をリストアップする
4. 開発チームがストーリーを完成する
5. ビジネススポンサーがストーリーの完成を承認する
6. 開発チームが完成したコードを引き渡す

ステップ3、4、5が効率よく順調に進み、スケジュールどおりかそれよりも早くチームがステップ6までたどり着けば、プロジェクトは成功です。すべての受け入れテストに合格したソフトウェアが、プロジェクトの最終成果物です。受け入れテストに合格したソフトウェアは、次に、いわば壁の向こうに放り投げられます。そして何も問題がなければ、その数か月後にビジネススポンサーがソフトウェアを使い始めることになります。

もし何か問題があれば、ソフトウェアが壁の向こうから投げ返されてくることもあります。優れた開発者は、よいテスト設計やペアプログラミングなどによって、このような事態を未然に防ごうとします。そして多くの場合、防ぐことができるで

† 訳注：ストーリーとは、エクストリームプログラミング（XP）における要求定義の手法。XPでは、ユースケースの代わりにストーリーが用いられる。日本ではユースケースが主流だが、海外のアジャイル開発者の間ではストーリーのほうがよく使われているようである。

しょう。しかし、プロジェクトが上記のパターンに従うかぎり、「ラストマイル」のプロセスがネックとなって、ビジネス価値の提供に依然として大幅な遅れが生じます。たとえ壁の向こうに投げられたコードになんの瑕疵もなく、スケジュールよりも早く提供できたとしてもです。

「ラストマイル」問題は、アジャイルソフトウェア開発プロセスがエンドツーエンドのソフトウェア提供プロセスにならないかぎり、解決されないでしょう。開発プロセスのあらゆるステップにおいて、本番稼働へのデプロイに関する課題に取り組まなければいけないのです。まず手始めに、人と自動化の役割から再検討していきましょう。

1.4 人

アジャイルソフトウェアのムーブメントがもたらした最大の貢献の1つは、ソフトウェア開発は本質的に社会的活動である、という認識を広めたことにあります。対話の質を改善することで、ソフトウェアの質も改善されるのです。アジャイル開発プラクティスを採り入れるために費やされたこれまでの努力は、その大半が古い社会的な組織構造を解体し、より効率的なパターンとプラクティスに置き換えていくことでした。

しかし、これまでのところ、ソフトウェア開発者とソフトウェアユーザとの対話にほとんどすべての焦点が当てられてきました。両者の対話は要求を正しく定義し、ビジネス目標に対する共通理解を得ることにつながります。しかし、非機能要求についてはどうでしょうか。非機能要求を引き出すべき相手は誰で、いつ、どうすればその人々が対話に参加してくれるのでしょうか。こうした疑問には、まだ答えが示されていません。

「コードを壁の向こうへ」型のソフトウェア開発をやめる最も簡単な方法は、非機能的で横断的な要求に責任を持つ利害関係者をソフトウェア開発の社会的活動の中に巻き込むことです。プロジェクトの早い時期から彼らに対話に参加してもらい、対話の機会を頻繁に持つことです。繰り返しますが、そうした取り組みにはおそらく、古い社会的な組織構造を解体し、より効率的なパターンとプラクティスに置き換えていくことが求められるでしょう。

例えば、運用スタッフは、ソフトウェアがいったん完成したらそれをインストールし、設定する必要があります。また、稼働中のシステムを監視し、適切に動作しているかを確認する必要があります。異常が発生した場合に適切な動作に復元できるように、その手順がドキュメント化されている必要もあります。システムが最初

にインストールされるとき、それからシステムが拡張されていくときのために、メモリ、ディスク、電源、冷房設備などの物理的なインフラ要求についてあらかじめ計画しておくことも必要です。

サポートやヘルプデスク担当のスタッフは、システムから出力される有用なエラーレポートと、効果的な診断の手順を必要としています。ユーザが直面しそうな、簡単なシステム上のトラブルを解決する方法を知っておく必要があります。また、システムの深刻な問題を、いつ、どのように上に報告すべきかを知っておく必要もあります。

多くの業界で、法令順守の担当スタッフは、例えば法律で義務付けられている個人情報の保護やデータ保管の規定をシステムがきちんと実装しているか、といったことを確認する必要があります。また、会計監査上必須の要求にシステムが従っているかを確かめる必要もあります。

こうした利害関係者が持つ要求も、システムが満たさなければいけない正真正銘のビジネス要求です。開発プロセスの中でこれらの要求に早期に対処すればするほど、ソフトウェアが本番稼働へ移行できるのも早くなるでしょう。利害関係者が対話に参加するのが早ければ早いほど、その要求にもすばやく、効率的に対処できるでしょう。

1.5 自動化

現在、「ラストマイル」のリリースプロセスはほとんどが手動で行われており、非効率で誤りの起こりやすい作業になっています。また、手動であるために時間がかかり、コストの高い作業でもあります。「ラストマイル」にかかる時間を大幅に削減するには、可能なかぎり、あらゆる作業を積極的に自動化していかなければいけません。それには、ソフトウェアのビルド方法も変えていく必要があります。

チームが開発プロセスの自動化を始めるとすぐに気づくのは、自動化の仕組みそのものもソフトウェア開発と同じ問題をたくさん抱えているということです。ビルドスクリプトにもテストやデバッグが必要です。テストが失敗したら、修正しなければいけません。しかしながら、自動化のための成果物は「非ソフトウェア」と認識されてしまいがちです。そのため、苦労の末に得られた自動化の問題に対処するノウハウを、チームが重要な成果物として見ていないこともよくあります。

ビルドスクリプトやテストスクリプト、インストールスクリプト、設定ファイルはみな、最終的な本番システムに貢献するエンドツーエンドなコードの一部であり、そのように取り扱われるべきものです。「開発用」と「納品用」とを区別し、異なる

取り扱いをすべきではありません。どちらも、プロジェクトのバージョン管理リポジトリ上で管理すべきです。どちらも、わかりやすく整理し、一貫性を保つべきです。また、どちらもリファクタリングを行い、構造をシンプルにし、共通部分の再利用化を図るべきです。最終的には、オペレーティングシステムやアプリケーションサーバ、データベース、ファイアウォール、ストレージシステムなど、システムのあらゆる構成要素に効率的な自動テストが用意されるべきです。システムのアーキテクチャ全体が、自動テストの観点から設計され、定義されるべきなのです。

多くのレガシーシステムと統合が必要な環境では、システム統合ポイントのテストを完全に自動化することは非現実的かもしれません。そうした場合には、レガシーシステムとの統合ポイントをモックを使ってテストするだけでも、何もしないよりはましです。しかし、これからシステムを構築する場合には、必ず公開された統合ポイントのテストを自動化する仕組み（例えば、テスト前処理、テスト後処理、テスト結果のログ出力や検索、などの機能）を提供すべきです。

リリース直前のソフトウェアを検証するために役立つ技術としては、プレイバックテストがあります。このテストは、稼働中のシステムで一定期間に起きた実際のトランザクションをリリース前のシステム上で「プレイバック（再現）」し、それぞれの結果を比較するものです。これから構築するシステムでは、効率的な自動プレイバックテストをサポートする仕組みを用意すべきです。

1.6 非機能要求テストを自動化するための設計

非機能要求はシステムの機能的な仕様に含まれないかもしれませんが、それでもやはり正当なビジネス価値に基づいた要求であり、そのビジネス価値はたいてい非常に重要なものです。「ラストマイル」で実施されるテストでは、システムのレスポンスタイム、トランザクションのスループット、可用性、セキュリティといった非機能要求が満たされているかを確認することに、多くの時間が費やされます。しかし、よくありがちなことですが、非機能要求テストが開始されるのはソフトウェアが「完成」した後なのです。

非機能要求テストを開始する正しいタイミングは、コーディングを開始する前です。特に性能やリソースの使用に関する要求は、前もって特定し、詳細を分析しておく必要があります。また開発が進んでいくと、そうした要求をコードの中にどうやって実現すればいいかを示す実装サンプルも作る必要があります。

性能テストの一般的なアプローチは、次のとおりです。まずプロジェクトの開始時に、例えば「この操作に許容されるシステムのレスポンスタイムは5秒以下であ

る」と仕様を定めます。それから、開発者がソフトウェアを書きます。ソフトウェアが完成したら、運用スタッフがそれをステージング環境にインストールします。最後に、テスト担当者が時計を見ながら操作し、システムが応答に何秒かかったかを計測します。

このアプローチには、大きな問題が2つあります。1つめに、もし当の操作に5秒でなく5分もかかることが判明した場合、ソフトウェアが本番稼働に入るだろうと皆が期待しているまさにそのときに、システムを根本から書き直さなければいけなくなるでしょう。2つめに、もし開発チームがそうした問題を回避するために性能テストを自動化し、それを継続的インテグレーションのテストスイートに組み込んでおいたとしても、得られるテスト結果を信頼していいのは、テスト環境で計測された性能と本番環境で実際に発揮される性能とが一致する場合だけです。それは多くの場合、現実的ではないか、コスト面から不可能なことです。

しかし、次のようなテストならば可能です。「本番システムは、ディスクへのランダムアクセスを1秒間に500回までする性能がある。したがって、この操作を実現するために、2,500回を超えてディスクへランダムアクセスしてはいけない。」ほとんどのオペレーティングシステム環境は詳細な性能カウンタを提供しており†、自動テストに簡単に組み込むことができます。カウンタベースの性能テスト戦略が時間計測テストよりも優れているのは、テストが実行環境から独立している点です。環境の制約がなく、より多くのハードウェア上でテストを実行できれば、より頻繁に実行できるようになります。またカウンタベースのテストは、より適切な粒度で記述することができます。そのため、開発チームはソフトウェアを開発している最中に、性能やリソースの問題に気づく可能性が高まります。ソフトウェアを開発している最中こそ、そうした問題を最も低コストで容易に解決できるときなのです。

性能とリソース使用の実装サンプルは、エンドツーエンドなソフトウェア提供の一部分となります。実装サンプルは、バージョン付けし、保守し、さまざまな環境で動作するように調整する必要があります。適切な性能測定値を収集、参照するための基盤ライブラリを提供することで、開発者が容易かつ効率的に、自動化された性能テストにコードを統合できるようにする必要もあります。こうした実装サンプルとテストが用意されていれば、本番稼働前の最終テストは速やかに実行され、予期せぬ不幸なトラブルを回避できるはずです。

† 訳注：例えばWindowsにはパフォーマンスモニタというツールが、LinuxなどのUNIX系OSにはvmstat、iostat、netstatといったコマンドがある。どちらもプロセッサ、メモリ、ディスク、ネットワークなどのリソースの利用状況を計測できる。

1.7 設計を稼働環境から分離せよ

継続的インテグレーションとテスト駆動設計は、高品質のソフトウェアを短期間で構築するための非常に有益なツールとしての役割を果たしてきました。迅速なフィードバックが得られることで、開発チームは誤りを早期かつ低コストに排除することができます。そして、いかなるときにもシステムが期待どおりに機能しているという自信を持って、開発に取り組むことができます。

残念ながら、これまでのところ本番稼働デプロイのプロセスにおいては、継続的インテグレーションとテスト駆動設計からはほとんど恩恵を得てきませんでした。理想を言えば、他システムと完全に統合された本番環境においても、ユーザテストや運用テストを簡単に記述できるのが望ましいでしょう。そして、ソフトウェアを書いたらすぐに検証ができるように、継続的インテグレーションシステムからテストスイート全体をすばやく実行できるのが望ましいでしょう。しかし通常は、いくつもの障害がその実現を妨げています。

第一に、本番環境はたいてい大規模、複雑、高価であり、構築するのは大変です。2つめに、本番環境に対してエンドツーエンドなテストを設計し、構築し、検証するのは、多くの場合、困難な作業です。3つめに、たとえ本番環境を構築し、テストも用意したとしても、そのようなテストを実行するにはとても長い時間がかかるでしょう。テストスイートをすべて実行するのに、数日かかることさえあります。

3つめの問題を解決する簡単な方法は、複数のテストを並行して実行することです。一度に10テストを実行できれば、テストスイートを通常より10倍の頻度で実行できることになります。残念ながら、1つめの問題（本番環境は大規模、複雑、高価である）があるため、この解決方法はあまり現実的ではありません。開発チームがテストを実施するために10もの本番環境を調達し、維持できるというのは、想像もつかない贅沢です。

しかし、もしそのコストを低減でき、本番環境を即座に構築でき、そしてコストをいくつもの開発チーム間で共有することができれば、大規模なテストスイートを並行して実行することも現実的な選択肢の1つになります。ちょうどいいことに、最近の仮想化技術の進歩がそれを可能にしています。今日、本番環境を完全に仮想化し、多数の安価な普通のハードウェア上にすばやく保存し、復元できる製品があります。

もちろん、開発システムが仮想化されたテスト環境と現実の本番環境とで異なる振る舞いをしてしまうとすれば、このアプローチにとっては致命的な欠陥となります。自動テストの結果をまったく信頼できないことになり、結果として、開発やテ

ストの効率がほとんど改善されないからです。

どんなビジネスソフトウェアであっても、デプロイされる環境に対して前提や依存を持っています。この前提や依存は、明示的なものもあれば暗黙的なものもあります。デプロイされる環境に対してシステムに暗黙的な前提や依存が多数あるときには、別の環境に向けて意味のあるエンドツーエンドのシステムテストを作成することは大変難しくなります。

そのため、仮想環境を活用してシステムテスト全体をすばやく実行できるようにするには、システム設計者は暗黙の前提と依存を見つけ出し、それらを顕在化させてテスト可能にしておく必要があります。またシステム設計者は、そうした前提や依存の数を、システム全体にわたって体系的に減らしていかなくてはいけません。

いったんシステムテストスイート全体の自動実行が現実的な選択肢の1つになってしまえば、そうしたテストスイートを作成するための投資は非常に魅力的なものとなります。最終的な目標は、継続的インテグレーションシステムによって、ソフトウェアがいかなるときにも本番稼働へデプロイできる状態にあるという自信を開発チームと運用スタッフが持てるようにすることなのです。

1.8 バージョンのないソフトウェアへ

アジャイルプロセスの価値は、あるビジネス要求が生じた時点からその要求を満たすソフトウェアが稼働段階に入る時点までの、エンドツーエンドの時間とコストを減らすことにあります。この目的を極限まで突き詰めると、ある機能の要求が発生してから、それが完成するやいなや稼働に入る、という状況を思い描くことができます。

こうした状況は、ある種の小規模で単純なWebアプリケーションでは、今日すでに実現されています。稼働中のアプリケーションに、1時間もかからずに新しい機能を追加できてしまうこともあります。この場合、開発者はわざわざバージョンを区切ってソフトウェアをリリースする必要がありません。単に機能が追加されるたびに、ソフトウェアをリリースしていけばいいからです。

しかし、細心の注意を必要とする大規模で複雑なレガシー環境では、バージョンのないソフトウェアというアイデアは、せいぜい遠いあこがれの目標でしかありません。「ラストマイル」のプロセスで発生するリリースコストが非常に大きいために、新機能の追加は大きな単位にまとめ、バージョンを区切って少ない頻度でリリースしなければいけないからです。この状況は、近い将来においても変わることはないでしょう。

しかし、現在「ラストマイル」にかかっているコストを、このまま見過ごし続けることはできません。無駄な作業によるビジネスへの直接的コストと、機会損失による間接的コストは莫大です。そして、そうしたコストがソフトウェア開発にかかる全コストのうちの大きな割合を占めており、その比率はますます増大しています。

バージョンのないソフトウェアを実現するのは不可能かもしれませんが、もっと上手にできることは確かです。簡単かつ明白で、今すぐに実行できる改善策はいくらでもあります。優秀でモチベーションに溢れた人々が取り組めば、さらに多くの改善策が発見されるのは間違いないでしょう。「ラストマイル」問題を、一夜にして解決することはできません。しかし、エンドツーエンドなソフトウェア開発プロセスにかかわるすべての人々がこの問題に注目し、積極的に協力し合い、一歩ずつ着実に取り組んでいけば、いずれ解決する日が来るでしょう。

2章
とある秘密基地とRubyによる20のDSL

Martin Fowler ◎チーフサイエンティスト

最近Rubyの人気が高まっている理由の多くは、内部ドメイン固有言語（DSL：Domain Specific Language）の記述に適していることにあります。内部DSLとは、ホスト言語の妥当なサブセットを使って書かれたDSLです†。現在、内部DSLを記述することがRubyコミュニティで再び脚光を浴びています。

内部DSL自体は古いアイデアで、特にLispコミュニティでよく知られています。Lispプログラマの多くは、RubyはDSLの領域に何も新しい技術をもたらしていないとして相手にしていません。しかし、Rubyが持つおもしろさの1つは、内部DSLを開発するために利用できるテクニックの幅広さです。Lispは素晴らしいメカニズムを提供してくれますが、多くの方法があるRubyと比べると、そのメカニズムの種類は比較的少ないと言えます。

このエッセイでは、1つの例題を基に多くの選択肢を紹介します。それにより、読者の皆さんがRubyの可能性を実感し、複数あるテクニックの中からどれが皆さんの役に立つか検討できるようにします。

2.1 秘密基地の例

ここからは、それぞれのテクニックを紹介するために簡単な例を使います。ここでは一般的で、興味深く、抽象的な構成の問題を例として用いています。この問題はあらゆる装置で見られるものです。例えば、Xというものが欲しいときに、一緒に互換性のあるYも必要になるというものです。コンピュータを買うとき、ソフト

† 訳注：一方で、著者は新しい言語の文法を定義し、その文法を処理する完全なパーサを実装する方法を外部DSLと呼んでいる。

ウェアをインストールするとき、他の多くの何気ない作業をするときに、この構成の問題を見かけるでしょう。

構成の問題を検討するために、ある会社について考えてみましょう[†]。その会社は、世界征服を企む邪悪な誇大妄想者たちへ、複雑な装置を提供することを専門にしています。そのような悪役が登場する映画の量から判断して、市場は巨大です。さらに、秘密基地は華やかなスパイによって爆破され続けているため、今後も成長していく市場と言えます。

そこでDSLを使い、悪役が秘密基地に設置するものの構成ルールを表現します。このDSLの例では、装置（Item）と資源（Resource）という2種類のものを扱います。装置とは、カメラや酸浴槽といった形のあるものです。資源とは、電気のように形がなく量で表されるものであり、装置に必要とされるものです。

この例では、電気（Electricity）と酸（Acid）という2種類の資源を扱います。資源は、多くのさまざまな属性を持つ可能性があり、それらの属性を適切な値にする必要があるとします。例えば、すべての装置が必要とする電力を、秘密基地内の発電機が供給できているかを確認する必要があります（悪役は公共インフラ設備を使いたがろうとしません）。各資源は属性が異なるので、この抽象的な構成の例では、資源ごとに別々のクラスを用意します。

秘密基地の問題では、資源を2つのカテゴリに分類します。1つめ（電力）は、少数の決められた属性を持つ単純なものです。そのため、属性はコンストラクタの引数として与えることができます。2つめ（酸）は、任意の属性を多く持っており、多くのセッターメソッドを必要とします。この例では、酸はたった2つの属性しか持ちませんが、実際にはたくさん存在すると想像してください。

装置には3つの特徴があります。資源を利用すること（use）、資源を提供すること（provide）、そして秘密基地内の他の装置に依存すること（depend on）です。

読者の皆さんの好奇心も湧き上がってきたところでしょうから、ここで抽象的な構成の実装をお見せしましょう。今後議論していくすべての例で、このモデルを利用します。

† 訳注：著者によると、本章の題材は特に想定しているような映画はなく、一般的なステレオタイプに基づいている。

lairs/model.rb

```
class Item
  attr_reader :id, :uses, :provisions, :dependencies
  def initialize id
    @id = id
    @uses = []
    @provisions = []
    @dependencies = []
  end
  def add_usage anItem
    @uses << anItem
  end
  def add_provision anItem
    @provisions << anItem
  end
  def add_dependency anItem
    @dependencies << anItem
  end
end

class Acid
  attr_accessor :type, :grade
end

class Electricity
  def initialize power
    @power = power
  end
  attr_reader :power
end
```

次の構成（Configuration）のオブジェクトの中に、すべての構成情報を保持します。

lairs/model.rb

```
class Configuration
  def initialize
    @items = {}
  end
  def add_item arg
    @items[arg.id] = arg
  end
```

```
    def [] arg
      return @items[arg]
    end
    def items
      @items.values
    end
  end
```

この章で使われる、いくつかの装置とルールを定義します。

- 酸浴槽(acid_bath)は12単位の電力、等級が5の塩酸(HCl)を使用する
- カメラ(camera)は1単位の電力を使用する
- 小型発電機(small_power_plant)は11単位の電力を提供し、秘密基地内の安全通気口(secure_air_vent)に依存する

上記の抽象的な構成に関するルールは、次のように表現できます。

lairs/rules0.rb

```
config = Configuration.new
config.add_item(Item.new(:secure_air_vent))

config.add_item(Item.new(:acid_bath))
config[:acid_bath].add_usage(Electricity.new(12))
acid = Acid.new
config[:acid_bath].add_usage(acid)
acid.type = :hcl
acid.grade = 5

config.add_item(Item.new(:camera))
config[:camera].add_usage(Electricity.new(1))

config.add_item(Item.new(:small_power_plant))
config[:small_power_plant].add_provision(Electricity.new(11))
config[:small_power_plant].add_dependency(config[:secure_air_vent])
```

このコードは構成を表現していますが、あまり流れるような(fluent)†表現にはなっていません。以降では、さまざまなコードの記述方法を模索しながら、ルールのよりよい表現を検討していきます。

† 訳注：著者は、後述するメソッドチェインなどを利用することで、読みやすく、よどみなく流れるように設計されたAPIを「流れるようなインターフェイス(Fluent Interface)」と呼んでいる。

2.2 グローバル関数の利用

関数は、プログラミングにおいて最も基本的な構造化メカニズムです。関数は、ソフトウェアを構造化し、プログラムに対象ドメインの名前を導入するための最も原始的な方法です。

したがって、最初に書くDSLはグローバル関数の呼び出しを並べたものになります。

lairs/rules8.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(acid)
acid_type(:hcl)
acid_grade(5)
uses(electricity(12))

item(:camera)
uses(electricity(1))

item(:small_power_plant)
provides(electricity(11))
depends(:secure_air_vent)
```

関数名は、そのDSLが扱うボキャブラリをプログラムに導入します。つまり、item関数は装置を宣言し、uses関数は装置が資源を利用することを示します。

上記のDSLで表現される構成ルールは、すべて関係についてのものです。カメラが1単位の電力を使うことを表現するには、カメラという装置と電力という資源の間にリンクが必要です。この最初の例では、関数呼び出しの並びから決定される文脈（コンテキスト）によって、リンクが設定されます。uses(electricity(1))の行は、カメラの宣言の直後にあるので、カメラに対して適用されます。関数呼び出しの並びによる逐次的な文脈によって、関係が暗黙のうちに定義されると言えるでしょう。

人にとってはDSLのテキストを読むことで、関数呼び出しの並びからその文脈を推測できます。しかし、コンピュータでDSLを処理するためにはもう少し工夫が必要です。文脈を追跡するために、DSLのロード時に特別な変数を用意します。当然のことですが、それをコンテキスト変数と呼びます。現在の装置を追跡するために、コンテキスト変数を1つ使います。

lairs/builder8.rb

```
def item name
  $current_item = Item.new(name)
  $config.add_item $current_item
end

def uses resource
  $current_item.add_usage(resource)
end
```

グローバル関数を使っているため、コンテキスト変数はグローバル変数にする必要があります。これはあまり褒められたものではありません。しかし、後で紹介するように、多くの言語にはグローバル変数の使用を回避する方法があります。確かに、グローバル関数の利用は理想からほど遠いものですが、スタート地点としては十分でしょう。

酸の属性を操作するときにも、同じ方法が利用できます。

lairs/builder8.rb

```
def acid
  $current_acid = Acid.new
end
def acid_type type
  $current_acid.type = type
end
```

逐次的な文脈では、装置とその資源の間のリンクを設定することができます。しかし、依存する装置の間にあるリンクに階層関係がない場合、うまく設定できません。そのために、装置同士を明示的に関係づける必要があります。明示的な関係を作成するには、装置を宣言する際に識別子を与え（item(:secure_air_vent)）、装置を後で参照する必要がある場合に、その識別子を使います（depends(:secure_air_vent)）。安全通気口に依存しているのが小型発電機であることは、逐次的な文脈を通して表現されます。

資源と装置の重要な違いは、資源はEvansが値オブジェクト（Value Object）†と

† 訳注：*"Domain-Driven Design: Tackling Complexity in the Heart of Software"* の著者であるEric Evansは、「色」や「量」のように、その属性だけが重要で、アイデンティティを考えることに意味のないオブジェクトを「値オブジェクト」と呼んでいる。逆に、装置のように、アイデンティティが同じならば、属性が異なっていても同一のものとして扱うものを「エンティティ（Entity）」と呼んでいる。

呼んでいるものであることです[Eva03]。その結果、資源は自分が属する装置以外のものから参照される心配はありません。しかし、装置自身は、依存関係を通してDSLのあらゆる箇所から参照される可能性があります。そのため、後で参照できるように、装置にはある種の識別子が必要です。

Rubyでこのような識別子を扱うには、:secure_air_ventのようなシンボルデータ型を使います。Rubyでのシンボルとは、コロンで始まる文字列（空白を除く）です。シンボルデータ型は、多くの主流の言語にはありません。シンボルデータ型を文字列のようなものだと考えることもできますが、本来は今回のように特定の目的で利用するものです。その結果、シンボルデータ型に対して通常の文字列操作の多くは適用できません。また、1つのシンボルデータ型が利用されているすべての箇所で、同じインスタンスを共有するように設計されています。その結果として、より効果的にインスタンスをルックアップできます。しかし、筆者がシンボルデータ型を利用する最も重要な理由だと考えているのは、そのシンボルをどう扱うかという意図を示すことができる点です。:secure_air_ventを文字列としてではなくシンボルとして用いています。そのように正しいデータ型を選ぶことで、筆者の意図を明確に示しています。

もちろん、変数を識別子として用いることもできます。筆者はDSLにおいて変数を使うことには消極的です。変数を使うことの問題は、それが可変であることです。同じ変数に異なるオブジェクトを代入できると、どの変数にどのオブジェクトが入っているかを追跡しなければいけなくなります。変数は便利な機能ではありますが、追跡するには不便です。そのため、DSLでは通常は変数を避けます。識別子と変数の違いは、識別子が常に同じオブジェクトを指す、すなわち変化しないということです。

識別子は依存関係を設定するためには欠かせないものです。しかし、識別子は逐次的な文脈を使う以外の代替手段として、資源の操作にも利用することもできます。

lairs/rules7.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(:acid_bath, acid(:acid_bath_acid))
acid_type(:acid_bath_acid, :hcl)
acid_grade(:acid_bath_acid, 5)
uses(:acid_bath, electricity(12))

item(:camera)
```

```
uses(:camera, electricity(1))

item(:small_power_plant)
provides(:small_power_plant, electricity(11))
depends(:small_power_plant, :secure_air_vent)
```

このように識別子を利用することで、関係を明示的に表現できます。さらに、グローバルなコンテキスト変数を使わずに済みます。これらは両方ともよいことです。筆者は明示的に表現することが好きであり、グローバル変数は好きではありません。しかし、その代償として、DSLがはるかに冗長になっています。そのため、DSLを読みやすくするために、なんらかの形で暗黙的なメカニズムを利用することは有用だと考えています。

2.3 オブジェクトの利用

前の節における関数の利用の主な問題点は、DSLのためにグローバル関数を定義しなければいけないことです。グローバル関数がたくさんあると管理が難しくなります。オブジェクトを使うことの利点の1つは、クラスによって関数を整理できることです。DSLコードを適切に書けば、DSL関数を一緒にまとめ、グローバル関数空間の外に出すことができます。

2.3.1 クラスメソッドとメソッドチェイン

オブジェクト指向言語において、メソッドのスコープを管理するための最も自明な方法は、クラスメソッドを使うことです。クラスメソッドは、メソッドの影響範囲を制限する際に役に立ちます。しかし、メソッド呼び出しのたびにクラス名を使わなければいけないため、繰り返しが増えてしまいます。以下の例のように、メソッドチェインとクラスメソッドを組み合わせることで、繰り返しの量を大幅に減らすことができます。

lairs/rules11.rb

```
Configuration.item(:secure_air_vent)

Configuration.item(:acid_bath).
    uses(Resources.acid.
      set_type(:hcl).
      set_grade(5)).
```

```
    uses(Resources.electricity(12))

Configuration.item(:camera).uses(Resources.electricity(1))

Configuration.item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```

DSLの各文は、クラスメソッドの呼び出しで始まります。クラスメソッドは、次の呼び出しのレシーバとして使われるオブジェクトを返します。そして、複数のメソッド呼び出しをつなげるために、繰り返し、次のメソッドへオブジェクトを返すことができます。ある程度までいくと、メソッドチェインがぎこちなくなってきます。そうしたら、再びクラスメソッドを利用します。

何をしているのか理解してもらうために、この例をもう少し詳細に掘り下げていきましょう。この例にはいくつか欠点があることを覚えておいてください。欠点については、以降の例で検討し、改善方法を示していきます。

それではitemメソッドを定義している箇所の冒頭から始めましょう。

lairs/builder11.rb

```
def self.item arg
  new_item = Item.new(arg)
  @@current.add_item new_item
  return new_item
end
```

このメソッドは新しい装置を生成し、クラス変数内の構成オブジェクトに加えて、装置を返します。ここで着目すべきなのは、新しく生成した装置を返すことで、メソッドチェインを実現している点です。

lairs/builder11.rb

```
def provides arg
  add_provision arg
  return self
end
```

providesメソッドは、単純にadd_provisionメソッドを呼び、自分自身のインスタンスを再度返しています。そうすることでメソッドチェインを継続します。他のメソッドも同様にメソッドチェインを継続します。

このようにメソッドチェインを使うことは、よいプログラミングのアドバイスの多くに反しています。多くの言語で、オブジェクトの状態を変化させるメソッドは、戻り値を何も返さないことが慣習となっています。この慣習は、コマンド・問い合わせの分離原則†に基づいています。そして、たいていの場合は従う価値があり、よく役に立つものです。あいにく、この原則は流れのある内部DSLに反しています。結果として、通常、DSLの作者はメソッドチェインをサポートするために、DSLコードの中ではこの原則を無視する判断をします。この例では、酸の種類と等級を設定するときにもメソッドチェインを使います。

さらに標準のコードガイドラインとの違いは、フォーマットに対して異なるアプローチをとっている点です。今回の場合、筆者はDSLが示している階層を強調するようにコードを配置しています。そのため、メソッドチェインを用いることで、メソッド呼び出しが新しい行にまたがるコードをよく見かけることになるでしょう。

この例は、メソッドチェインを説明すると同時に、資源を生成するためのファクトリクラスの使い方も説明しています。Electricityクラスにメソッドを加えるよりも、電力と酸のインスタンスを生成するクラスメソッドを持つ資源クラス(Resources)を定義します。そのようなファクトリは、適切なオブジェクトを生成するためのクラス(スタティック)メソッドのみを持つため、クラスファクトリもしくはスタティックファクトリと呼ばれます。クラスファクトリを使うことで、多くの場合DSLをより読みやすくすることができ、実際のモデルクラスに余計なメソッドを追加することを避けられます。

このことは、このDSLのコード片の問題を強調しています。つまり、コードを動作させるため、ドメインクラスになじまないメソッドを多く付け加えなければいけないのです。ほとんどのメソッドは、1つ1つの呼び出しで意味のあるものにすべきです。しかし、DSLメソッドは、DSL表現の中で意味があるように記述されます。その結果、コマンド・問い合わせ分離原則に反しているのと同じように、名前の付け方も通常のメソッドと異なるものになります††。さらに、DSLメソッドは非常に文脈固有です。オブジェクトを生成するときは、DSL表現の中でDSLメソッドを使うべきです。基本的に、よいDSLメソッドの原則は、通常のメソッドを効果的に動かすための原則とは異なります。

---

† 訳注:コマンド・問い合わせ分離原則とは、なんらかの処理を行うコマンドメソッドと、データを返却する問い合わせメソッドは別々のメソッドとして設計するという原則である。これにより、そのメソッドの利用者が混乱することを防ぐという効果がある。

†† 訳注:例えば、内部DSLとして有名なjMockでは、is、and、orのように通常のメソッドでは付けないような名前のメソッドが存在する。

2.3.2 Expression Builder

DSLと標準APIの間の衝突を避ける方法の1つは、Expression Builderパターン†を使うことです。このパターンの本質は、DSL内で使われるメソッドを別個のオブジェクト上に定義し、実際のドメインオブジェクトを生成するようにする、というものです。Expression Builderパターンは、2通りの方法で利用することができます。ここで示す1つめの方法は、前回と同じDSL言語を使い、その中でドメインオブジェクトの代わりにビルダオブジェクトを生成するというものです。

ビルダオブジェクトを生成するには、ドメインオブジェクトではなく装置ビルダ（ItemBuilder）のオブジェクトを返すように、DSLで一番最初に呼び出されるクラスメソッドの呼び出しを変更します。

lairs/builder12.rb

```
def self.item arg
  new_item = ItemBuilder.new(arg)
  @@current.add_item new_item.subject
  return new_item
end
```

装置ビルダはuses、provides、depends_onメソッドをサポートし、それらを実際の装置オブジェクトのメソッドへ変換します。

lairs/builder12.rb

```
attr_reader :subject
def initialize arg
  @subject = Item.new arg
end
def provides arg
  subject.add_provision arg.subject
  return self
end
```

もちろん、Expression Builderを使い始めると、完全にドメインオブジェクトのAPIを意識する必要がなくなり、より明確にDSLを書くことができます。次のコードを見てください。

† 訳注：Expression Builderパターンとは、流れるようなインターフェイスを通常のメソッドに変換するオブジェクトを用意する手法である。

ビルダオブジェクトを利用していないときは、依存性を設定するために、グローバル変数またはクラス変数から他の装置を探さなければいけませんでした。

最後の改善は、コードを読みやすくするために、酸ビルダ（AcidBuilder）のメソッドの名前を変えたことです。それは、酸ビルダによって背後にあるドメインオブジェクトとの名前衝突を心配する必要がなくなったためです†。

† 訳注：builder11.rbでは、Resources.acidメソッドがAcidオブジェクトをそのまま返しているため、すでにAcidに定義されているtype、gradeのアクセサとの衝突を避け、メソッド名をset_type、set_gradeとしていた。一方、builder13.rbでは、Resources.acidメソッドがAcidBuilderを返しており、AcidBuilderなら自由にメソッド名を付けられる。よって、Resources.acidメソッドの後にAcidBuilderのtype、gradeメソッドをチェインできるようになった。

lairs/rules14.rb

```
ConfigurationBuilder.
  item(:secure_air_vent).
  item(:acid_bath).
    uses(Resources.acid.
       type(:hcl).
       grade(5)).
    uses(Resources.electricity(12)).
  item(:camera).uses(Resources.electricity(1)).
  item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```


グラミングと呼ばれる「刑務所からの釈放」†カードが開発者に与えられたのです。

4.1 多言語プログラミングとは

多言語という言葉は、多くの言語を話すことを意味します。多言語プログラミングは、特定の問題を解くための特別な言語を使うために、Javaの（そしてC#も同様に）言語とプラットフォームの分離を利用します。現在、Java仮想マシンや.NETのマネージド・ランタイム上で動作する言語はたくさん存在します。しかし、開発者として、私たちはこの可能性を十分に活かすことができていません。

もちろん開発者は昔からさまざまな言語を使っています。多くのアプリケーションでは、データベースにアクセスするためにSQLを使いますし、Webページを対話的にするためにJavaScriptを使いますし、もちろんあらゆるものを設定するためのXMLもそこら中にあります。しかし、多言語プログラミングのアイデアはこれらと異なります。これらの例はすべて、JVMと直交しています。つまり、これらの言語はJavaの世界の外側で動作します。そしてそれが大きな頭痛の種になります。何十億ドルのお金がオブジェクトと、集合論に基づくSQLのインピーダンスミスマッチに費やされていることでしょうか。開発者は複数の言語を使っている点ですでに苦痛を味わっているので、インピーダンスミスマッチは開発者をいらいらさせます。しかし、多言語プログラミングはこれらの例とは異なります。多言語プログラミングは、インピーダンスミスマッチを取り除き、JVMの内部でバイトコードを生成する言語を活用することなのです。

多言語プログラミングを導入することの他の障害は、言語を変えることへの気持ちの問題です。昔は、言語を変更することはプラットフォームを変更することと同じであり、すべてのライブラリを書き直したくない開発者にとっては、明らかに悪い意味を持っていました。しかし、JavaとC#において、言語とプラットフォームを分離したことにより、もはやこのように苦闘する必要がなくなりました。多言語プログラミングにより、既存のすべての資産を活用し、目の前の仕事を片付けるためにより適切な言語を使うことができます。

目の前の仕事を片付けることに適しているというのは、どういう意味でしょうか。次の節では、多言語プログラミングを適用した例をいくつか紹介します。

† 訳注：モノポリーの刑務所脱出カード。モノポリーでは、刑務所に投獄されたユーザは50ドル払うか、ダイスを降ってゾロ目を出して釈放されるか、このカードを使って刑務所から脱出する。

2.3.2 Expression Builder

DSLと標準APIの間の衝突を避ける方法の1つは、Expression Builderパターン†を使うことです。このパターンの本質は、DSL内で使われるメソッドを別個のオブジェクト上に定義し、実際のドメインオブジェクトを生成するようにする、というものです。Expression Builderパターンは、2通りの方法で利用することができます。ここで示す1つめの方法は、前回と同じDSL言語を使い、その中でドメインオブジェクトの代わりにビルダオブジェクトを生成するというものです。

ビルダオブジェクトを生成するには、ドメインオブジェクトではなく装置ビルダ（ItemBuilder）のオブジェクトを返すように、DSLで一番最初に呼び出されるクラスメソッドの呼び出しを変更します。

lairs/builder12.rb

```
def self.item arg
  new_item = ItemBuilder.new(arg)
  @@current.add_item new_item.subject
  return new_item
end
```

装置ビルダはuses、provides、depends_onメソッドをサポートし、それらを実際の装置オブジェクトのメソッドへ変換します。

lairs/builder12.rb

```
attr_reader :subject
def initialize arg
  @subject = Item.new arg
end
def provides arg
  subject.add_provision arg.subject
  return self
end
```

もちろん、Expression Builderを使い始めると、完全にドメインオブジェクトのAPIを意識する必要がなくなり、より明確にDSLを書くことができます。次のコードを見てください。

† 訳注：Expression Builderパターンとは、流れるようなインターフェイスを通常のメソッドに変換するオブジェクトを用意する手法である。

lairs/rules14.rb

```
ConfigurationBuilder.
  item(:secure_air_vent).
  item(:acid_bath).
    uses(Resources.acid.
       type(:hcl).
       grade(5)).
    uses(Resources.electricity(12)).
  item(:camera).uses(Resources.electricity(1)).
  item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```

ここで、最初からビルダでメソッドチェインを利用しています。これで関数呼び出しの繰り返しを取り除くだけでなく、あまり洗練されていないクラス変数を避けることができます。最初に呼ばれている関数は、装置ビルダの新しいインスタンスを生成するクラスメソッドです。

lairs/builder14.rb †

```
def self.item arg
  builder = ConfigurationBuilder.new
  builder.item arg
end
def initialize
  @subject = Configuration.new
end
def item arg
  result = ItemBuilder.new self, arg
  @subject.add_item result.subject
  return result
end
```

構成ビルダ（ConfigurationBuilder）のインスタンスを生成し、すぐに新しいインスタンスのitemメソッドを呼び出しています。このインスタンスメソッドで新しい装置ビルダのインスタンスを生成し、さらなる処理のために返しています。インスタンスメソッドと同じ名前のクラスメソッドを使うという変わった例を示しています。これは混乱の元になるので、通常は避けます。しかし、より滑らかなDSLに

† 訳注：Rubyでは、self.itemメソッドのように特にreturnを明示しないと、最後に評価された文の結果が戻り値になる。

役立つので、筆者がよく使うAPIのルールを再び破ります。

装置ビルダは、以前示したものと同じように、装置の情報を設定するメソッドを持っています。さらに、新しい装置の定義を始めたいときに対応するために、装置ビルダ自身にもitemメソッドが必要になります。

lairs/builder14.rb

```
def item arg
  @parent.item arg
end
def initialize parent, arg
  @parent = parent
  @subject = Item.new arg
end
```

装置ビルダを生成するときに、構成ビルダを親ビルダとして渡している理由は、このitemメソッドで親ビルダを呼び出す必要があるからです。その他の理由としては、装置ビルダが依存性を設定する際に、他の装置を探せるようにするためです。

lairs/builder14.rb

```
def depends_on arg
  subject.add_dependency(configuration[arg])
  return self
end
def configuration
  return @parent.subject
end
```

ビルダオブジェクトを利用していないときは、依存性を設定するために、グローバル変数またはクラス変数から他の装置を探さなければいけませんでした。

最後の改善は、コードを読みやすくするために、酸ビルダ（AcidBuilder）のメソッドの名前を変えたことです。それは、酸ビルダによって背後にあるドメインオブジェクトとの名前衝突を心配する必要がなくなったためです†。

† 訳注：builder11.rbでは、Resources.acidメソッドがAcidオブジェクトをそのまま返しているため、すでにAcidに定義されているtype、gradeのアクセサとの衝突を避け、メソッド名をset_type、set_gradeとしていた。一方、builder13.rbでは、Resources.acidメソッドがAcidBuilderを返しており、AcidBuilderなら自由にメソッド名を付けられる。よって、Resources.acidメソッドの後にAcidBuilderのtype、gradeメソッドをチェインできるようになった。

ドメインオブジェクトごとにビルダオブジェクトを作る方法だけが、Expression Builderパターンの唯一の利用方法ではありません。もう1つの方法は、ビルダとして振る舞う単一のオブジェクトを使うことです。次のコードは、先ほど見たDSLと同じように動きます。

lairs/builder13.rb

```
def self.item arg
  result = self.new
  result.item arg
  return result
end
def initialize
  @subject = Configuration.new
end
def item arg
  @current_item = Item.new(arg)
  @subject.add_item @current_item
  return self
end
```

新しい装置ビルダのオブジェクトを作る代わりに、現在操作している装置を追跡するためにコンテキスト変数を利用します。これにより、親ビルダの転送メソッド†を定義する必要がなくなります。

2.3.3 さらなるメソッドチェイン

メソッドチェインはよいツールではありますが、いつでも使えるものでしょうか。資源ファクトリを取り除くことができるでしょうか。確かに、それは可能です。DSLコードは、以下のようになります（読みやすくするために、いくつか空行を追加していることに注意してください。Rubyは途中の改行や空行を無視します）。

lairs/rules2.rb

```
ConfigurationBuilder.
  item(:secure_air_vent).

  item(:acid_bath).
```

† 訳注：親ビルダの転送メソッドとは、装置ビルダのインスタンスを返却している構成ビルダのitemメソッドのこと（builder14.rb）。

```
uses.acid.
      type(:hcl).
      grade(5).
    uses.electricity(12).

  item(:camera).uses.electricity(1).

  item(:small_power_plant).
    provides.electricity(11).
    depends_on(:secure_air_vent)
```

メソッドチェインとパラメータのどちらを使うのか、という選択肢は絶えずつきまといます。grade(5)のようにパラメータがリテラルの場合、パラメータではとても簡単になる一方で、メソッドチェインを使うと非常に複雑になってしまうでしょう。筆者は複雑なものよりも簡単なものを好むため、この場合にどちらを選ぶかは単純です。注意が必要なのは、uses.electricity...とuses(Resources.electricity... のどちらを使うかです。

メソッドチェインを使えば使うほど、ビルダは複雑になります。これは、特に子オブジェクトを必要とし始めるときに起こります。上記の2つの選択肢は、ビルダが複雑になる際の格好の例になっています。資源は、後に続くusesメソッドかprovidesメソッドのどちらか2つの文脈で使われます。その結果、メソッドチェインを使う場合は、正しくelectricityメソッドの呼び出しに応答するために、どの文脈にいるかを追跡しなければいけません。

一方で、パラメータを利用する際の問題は、メソッドチェインでは可能だったスコープ管理ができなくなることです。そのため、パラメータ生成になんらかのスコープを提供する必要があります。この例では、ファクトリのクラスメソッドを利用することです。ファクトリ名を毎回引用するのはわずらわしいため、筆者は避けることを好みます。

パラメータを利用する際のもう1つの問題は、パラメータとメソッドチェインのどちらを使うかは、DSLの利用者には推測できないということです。これは、DSL表現の記述をさらに難しくします。

筆者は、読者の皆さんに指針を示せるだけの十分な経験を積んでいるわけではないため、ここでの助言は暫定的なものです。ただし、メソッドチェインはテクニックとして多くのサポートが利用できるので、メソッドチェインを必ず使いましょう。しかし、メソッドチェインを利用しているときに、メソッドチェインの扱いの複雑さに注意してください。漠然とした指標ではありますが、ビルダの実装が複雑になってきたら、パラメータを導入してください。後ほど、クラスファクトリの繰り

返しを回避する際に役に立つパラメータ導入のテクニックをお見せします。ただし、これらはDSLのホスト言語に依存します。

2.4 クロージャの利用

クロージャは、特に内部DSLによく使われる動的言語において、一般的な機能になりつつあります。クロージャを使うと、簡単に階層構造に新しいコンテキストを導入することができるため、DSLにとてもよく合います。以下は、クロージャを使った例です。

lairs/rules3.rb

```
ConfigurationBuilder.start do |config|
  config.item :secure_air_vent

  config.item(:acid_bath) do |item|
    item.uses(Resources.acid) do |acid|
      acid.type = :hcl
      acid.grade = 5
    end
    item.uses(Resources.electricity(12))
  end

  config.item(:camera) do |item|
    item.uses(Resources.electricity(1))
  end

  config.item(:small_power_plant) do |item|
    item.provides(Resources.electricity(11))
    item.depends_on(:secure_air_vent)
  end
end
```

この例の特徴は、はっきりとレシーバを指定して各メソッドを呼び出しており、メソッドチェインを使っていない点です。レシーバは、ホスト言語のクロージャ構文を使って構成されており、DSLによくある階層構造に非常に適した方法で、メソッド呼び出しのネストを簡単にしています。

このアプローチが見た目に直接与える利点は、ホスト言語の自然なネストがDSLコードのネストをそのまま表している点です。これにより、コードの配置が簡単になります。装置や酸などのDSLの言語要素を保持するために使われる変数(例えば、

itemやacid）は、ホスト言語の文法構造によって適切にブロック内に制限されます。

明示的なレシーバを使うことで、メソッドチェインを使う必要がなくなります。つまり、ドメインオブジェクト自身のAPIが使えるときに、ビルダの利用を避けることができるかもしれません。この例では、itemではビルダを使いましたが、acidでは実際のドメインオブジェクトを使いました。

このテクニックの制約の1つは、言語がクロージャを持っている必要があることです。一時変数を使えば、ある程度近いことはできますが、一時変数に関するあらゆる問題を引き起こします。なぜなら、スコープを別個に用意しないかぎり、変数のスコープが制限されないからです。追加のスコープがあろうとなかろうと、できあがるコードには筆者がDSLに求めるような流れはなくなり、間違いの起こりやすいものになります。クロージャにはスコープと変数定義があるため、この問題を回避できます。

2.5 コンテキストの評価

これまでの議論では、DSLコードが評価される全体のコンテキストについて触れてきませんでした。言い換えれば、レシーバなしに関数呼び出しやデータ項目を呼び出す場合、それはどのように解決されるのでしょうか。今までは、コンテキストはグローバルで、関数foo()はグローバル関数と仮定していました。異なるスコープにおいて関数を利用できるようにする方法として、メソッドチェインとクラスメソッドを用いる方法について説明してきましたが、DSLプログラム全体のスコープを変更してしまうことも可能です。

最も素直な方法は、プログラムテキストをクラス定義の中に記述することです。この方法では、コードはクラスに定義されたメソッドやフィールドを利用することができます。オープンクラス†をサポートしている言語では、DSLプログラムを既存クラスの中に直接記述することができます。オープンクラスをサポートしていない場合でも、サブクラスを作り、DSLプログラムを記述すればいいでしょう。

このスタイルの例は以下のようになります††。

† 訳注：メソッドやフィールドの定義を後から追加できるクラスをオープンクラスと呼ぶ。Rubyでは、StringやArrayなどの基本的なデータ型もオープンクラスとなっており、独自のメソッドやフィールドを自由に定義できる。

†† 訳注：Rubyではオープンクラスの機能を利用できるが、この例では、あえて上記のサブクラス化のアプローチを紹介している。

lairs/rules17.rb

```
class PrimaryConfigurationRules < ConfigurationBuilder
  def run
    item(:secure_air_vent)

    item(:acid_bath).
      uses(acid.
         type(:hcl).
         grade(5)).
      uses(electricity(12))

    item(:camera).uses(electricity(1))

    item(:small_power_plant).
      provides(electricity(11)).
      depends_on(:secure_air_vent)
  end
end
```

DSLテキストをサブクラスに配置することで、グローバルな実行コンテキストにおいて実行する際に不可能だったことが可能になります。もはやitemメソッドの連続呼び出しのために、メソッドチェインを利用する必要はありません。なぜなら、itemメソッドを構成ビルダのメソッドにできるからです。同様に、acidやelectricityを構成ビルダのメソッドとして定義し、スタティックファクトリを避けることができます。

この方法の欠点は、クラスやメソッドのヘッダとフッタがDSLテキストに追加されてしまうことです。

この例では、オブジェクトインスタンスのコンテキストでテキストを評価することを示してきました。オブジェクトインスタンスのコンテキストは、インスタンス変数を参照できるため便利です。クラスメソッドを利用して、クラスコンテキストに対しても同じことができます。筆者は、通常、インスタンスコンテキストのほうを好みます。それは、ビルダインスタンスを生成し、評価に使い、破棄することが可能になるからです。この方法では、DSLテキストの評価がインスタンスごとに分離しているので、データのゴミが残り、もう1つのインスタンスを台無しにするリスクを防いでいます（並行性が絡むと、特に厄介なリスクになります）。

Rubyは、コンテキストの変更と、ヘッダ・フッタの除去を同時に実現する大変便利な方法を提供しています。Rubyには、instance_evalというメソッドがあります。このメソッドは、文字列またはブロックとしてコードを取得し、オブジェクトイン

スタンスのコンテキストで評価します。これにより、ファイルの中にDSLテキストだけを残しても、評価コンテキストを調節することができます。この方法を使った例は以下のようになります。

lairs/rules1.rb

```
item :secure_air_vent

item(:acid_bath).
  uses(acid.
       type(:hcl).
       grade(5)).
  uses(electricity(12))

item(:camera).uses(electricity(1))

item(:small_power_plant).
  provides(electricity(11)).
  depends_on(:secure_air_vent)
```

言語によっては、複数のクロージャを組み合わせて使い、評価コンテキストを変えることができるものもあります。Rubyでは、クロージャで定義されたコードをinstance_eval関数に渡し、そのクロージャをオブジェクトインスタンスのスコープで評価することができます。これを利用することにより、DSLのテキストは次のように書けます。

lairs/rules18.rb

```
item :secure_air_vent

item(:acid_bath) do
  uses(acid) do
    type :hcl
    grade 5
  end
  uses(electricity(12))
end

item(:camera) do
  uses(electricity(1))
end

item(:small_power_plant) do
```

```
    provides(electricity(11))
    depends_on(:secure_air_vent)
  end
```

できあがったコードは非常に読みやすくなりました。クロージャを使って構成問題を表現しましたが、明示的なレシーバとしてブロック引数を繰り返し使うことを回避しました。しかし、これは注意が必要なテクニックです。ブロックコンテキストの切り替えは、混乱を招きやすくなります。各々ブロックにおいて、擬似変数であるselfが異なるオブジェクトを参照するため、DSLの利用者が混乱してしまいます。実際にブロック内の普通のselfへのアクセスする必要がある場合にも、ブロックコンテキストの切り替えがコードをぎこちなくします。

この混乱は、Rubyのbuilderライブラリ†の開発で実際に起こったことです。初期バージョンは、instance_evalメソッドを利用していました。しかし、実際に使われていく中で、混乱を招きやすく、利用が難しいことがわかりました。Jim Weirich（Rubyのbuilderライブラリの作者）は、このような評価コンテキストの切り替えは、プログラマ向けのDSLにとってよいアイデアではないと結論付けました。なぜなら、ホスト言語の想定に反しているからです（このことは、RubyでDSLを利用している他の開発者たちの経験とも一致していました）。もしDSLが非プログラマ向けであれば、それほど大きな問題ではありません。なぜなら、非プログラマはホスト言語の想定を知らないからです。筆者は、内部DSLがよりホスト言語に統合されれば統合されるほど††、利用者は言語の想定を逸脱するような処理を書くことをより嫌がるようになると考えています。構成の例のように、ホスト言語とそれほど同じに見えなくてもよいミニ言語にとっては、言語が読みやすいというメリットはより大きくなるでしょう。

2.6 リテラルコレクション

関数呼び出しの構文は、内部DSLを構築する重要なメカニズムです。多くの言語にとっては、唯一利用可能なメカニズムでもあります。しかし、いくつかの言語で利用可能で便利なメカニズムとしてリテラルコレクションがあります。リテラルコ

† 訳注：http://builder.rubyforge.org/

†† 訳注：著者は、ホスト言語とDSLの境界があいまいで、DSLプログラマがホスト言語の文法に頼りながらDSLを書ける状況を「統合する」と言っている。著者は、DSLとホスト言語との境界があいまいなDSLを設計するときは、プログラマが期待するホスト言語の動作に反しないDSLにすることを推奨している。

レクションを記述し、DSLの表現の中で自由に利用できる能力は有用です。しかし、多くの言語でこの能力は制限されています。なぜならリテラルコレクションを扱うための便利な構文がないか、もしくは利用したいと思うであろう箇所のすべてで、実際には利用できないからです。

リストとマップ（ハッシュ、ディクショナリ、連想配列とも言います）の2種類のリテラルコレクションが有効です。最新の言語のほとんどは、リテラルコレクションのオブジェクトと、それらを扱うための手頃なAPIを一緒にライブラリで提供しています。Lispプログラマならリストでマップを実装できると言うでしょうが、両方のデータ構造はDSLの記述に役立ちます。

ここで、acidの定義にリテラルデータ構造を利用する例を示します。

lairs/rules20.rb

```
item :secure_air_vent

item(:acid_bath) do
  uses(acid(:type => :hcl, :grade => 5))
  uses(electricity(12))
end

item(:camera) do
  uses(electricity(1))
end

item(:small_power_plant) do
  provides(electricity(11))
  depends_on(:secure_air_vent)
end
```

この例では、あいまいさがないときは括弧を使わなくてもよいというRubyの利点を活用し、関数呼び出しとリテラルコレクションを併用しています。acid関数は以下のようになります。

lairs/builder20.rb

```
def acid args
  result = Acid.new
  result.grade = args[:grade]
  result.type = args[:type]
  return result
end
```

リテラルハッシュを引数として使うことは、Rubyにおいてよくあるイディオムです（Perlの影響の1つです）。特に、任意の引数を多く持つ生成メソッドのような関数の場合に便利です。この例では、リテラルハッシュを使うことで、きれいなDSL構文を提供できるだけでなく、酸や電力のビルダを避け、直接必要とするオブジェクトだけを生成できます。

さらにリテラルを使ってみる場合どうなるでしょうか。uses、provides、dependes_onの関数呼び出しをマップで置き換えてみます。

lairs/rules4.rb

```
item :secure_air_vent

item :acid_bath,
  :uses => [acid(:type => :hcl,
                 :grade => 5) ,
            electricity(12)]

item :camera,
  :uses => electricity(1)

item :small_power_plant,
  :provides => electricity(11),
  :depends_on => :secure_air_vent
```

このアプローチの長所と短所の両方が現れています。小型発電機は構成が単純なため、この方法が非常によく合っています。酸浴槽のように少し構成が複雑な場合、コードがぎこちなくなっています。この例では、酸浴槽は2つの資源に依存しています。そこで、acidとelectricityのメソッド呼び出しをリストにして渡す必要があります。ただし、一度リテラルマップの中がネストし始めると、何をしているのか理解しにくくなり始めます。

メソッド呼び出しをリストにするというステップも、実装を複雑にします。itemメソッドは、名前とマップの両方を引数に持ちます。Rubyは、name引数と、その後に名前と値がペアになっている可変長引数が続く形になっているとして扱います。

lairs/builder4.rb

```
def item name, *args
  newItem = Item.new name
  process_item_args(newItem, args) unless args.empty?
```

```
    @config.add_item newItem
    return self
  end
```

process_item_args メソッドでは、各々のハッシュを処理するために、キーに対して case 文を利用する必要があります。さらには、ハッシュの値は単一要素の場合もあればリストの場合もあるため、それに対応する必要があります。

lairs/builder4.rb

```
  def process_item_args anItem, args
    args[0].each_pair do |key, value|
      case key
      when :depends_on
        oneOrMany(value) {|i| anItem.add_dependency(@config[i])}
      when :uses
        oneOrMany(value) {|r| anItem.add_usage r}
      when :provides
        oneOrMany(value) {|i| anItem.add_provision i}
      end
    end
  end
  def oneOrMany(obj, &block)
    if obj.kind_of? Array
      obj.each(&block)
    else
      yield obj
    end
  end
```

このような状況にあるとき、つまりハッシュの値が単一かリストのどちらの可能性もある場合には、すべてをリストで処理してしまうほうが簡単です。

lairs/rules21.rb

```
item :secure_air_vent

item :acid_bath,
  [:uses,
    acid(:type => :hcl, :grade => 5),
    electricity(12)]

item :camera,
  [:uses, electricity(1)]
```

```
item :small_power_plant,
  [:provides, electricity(11)],
  [:depends_on,  :secure_air_vent]
```

ここで、itemメソッドの引数は装置の名前とリスト（ハッシュではなく）です。リストの最初の項目はキーで、リストの残りの項目は値です（これはLispプログラマがリストをハッシュとして使う方法です）。このアプローチはネストを減らし、処理がより簡単です。

lairs/builder21.rb

```
def item name, *args
  newItem = Item.new name
  process_item_args(newItem, args) unless args.empty?
  @config.add_item newItem
  return self
end
def process_item_args anItem, args
  args.each do |e|
    case e.head
    when :depends_on
      e.tail.each {|i| anItem.add_dependency(@config[i])}
    when :uses
      e.tail.each {|r| anItem.add_usage r}
    when :provides
      e.tail.each {|i| anItem.add_provision i}
    end
  end
end
```

ここで重要なテクニックは、リストを番号が付いた列ではなく、ヘッド（head）とテイル（tail）からなるデータ構造として考えている点です。何もメリットはないのでハッシュを2要素のリストで置き換えないでください。代わりに、キーをリストのヘッドとして扱い、すべての値をテイルにつなげてください。そうすることで、1つのコレクションを別のデータ構造の中に埋め込む必要がなくなります。headメソッドとtailメソッドは、デフォルトではRubyのリストクラス（Arrayクラス）にありませんが、簡単に付け加えることができます。

lairs/builder21.rb

```
class Array
  def tail
    self[1..-1]
  end
  alias head first
end
```

リテラルコレクションの利用に関する話題を終える前に、それぞれの究極の形態について示しておくほうがよいでしょう。以下は、主にマップを用いて、必要に応じてリストを使った場合の全設定です。

lairs/rules22.rb

```
{:items => [
  {:id => :secure_air_vent},
  {:id => :acid_bath,
    :uses => [
      [:acid, {:type => :hcl, :grade => 5}],
      [:electricity, 12]]},
  {:id => :camera,
    :uses => [:electricity, 1]},
  {:id => :small_power_plant,
    :provides => [:electricity, 11],
    :depends_on => :secure_air_vent}
  ]}
```

そして、以下はリストのみを利用した例です。これはGreenspun形式†と呼んでもいいでしょう。

lairs/rules6.rb

```
[
  [:item, :secure_air_vent],

  [:item, :acid_bath,
    [:uses,
        [:acid,
            [:type, :hcl],
```

† 訳注：著者は、RubyでLispのようにリストのみを用いたコードを書いていることをPhil Greenspunの第十法則にちなんでGreenspun形式と呼んでいる。

```
    [:grade, 5]],
           [:electricity, 12]]],

      [:item, :camera,
        [:uses, [:electricity, 1]]],

      [:item, :small_power_plant,
        [:provides, [:electricity, 11]],
        [:depends_on, :secure_air_vent]]]
```

2.6.1 可変長引数メソッド

言語によっては可変長引数メソッドを定義できるものもあります。これは、関数呼び出しの中で、リテラルのリストを使う際に便利なテクニックです。以下のバージョンでは可変長引数メソッドを用いて、usesメソッドの呼び出しが1回で済むようにしています。

lairs/rules24.rb

```
item :secure_air_vent

item(:acid_bath) do
  uses(acid(:type => :hcl, :grade => 5),
       electricity(12))
end

item(:camera) do
  uses(electricity(1))
end

item(:small_power_plant) do
  provides(electricity(11))
  depends_on(:secure_air_vent)
end
```

このように、1回のメソッド呼び出しにリストをまとめて渡したい場合は、可変長引数メソッドを利用すると便利です。特に、リテラルリストを記述する場所に対して制約が厳しい場合に有用です。

2.7 動的受信

動的言語の利点の1つは、オブジェクトに定義されていないメソッドの呼び出しに適切に応答できることです。

例を交えて、その機能についてもう少し紹介します。今までは、秘密基地の資源の数は比較的限定されていて、資源を扱う具体的なコードを喜んで書けるぐらい十分限定されていると仮定してきました。もしそうした前提がなかったとしたら、どうなるのでしょうか。もし資源がたくさんあり、それらの資源を構成の一部としてその中に含めたいとしたら、どうなるのでしょうか。

lairs/rules23.rb

```
resource :electricity, :power
resource :acid, :type, :grade

item :secure_air_vent

item(:acid_bath).
  uses(acid(:type => :hcl, :grade => 5)).
  uses(electricity(:power => 12))

item(:camera).
  uses(electricity(:power => 1))

item(:small_power_plant).
  provides(electricity(:power => 11)).
  depends_on(:secure_air_vent)
```

この例では、まだビルダに対して electricity メソッドと acid メソッドの呼び出しが必要です。それらは、新しく定義した資源をメソッドの中で組み立てるために必要となります。しかし、資源から呼び出すメソッドを推測するべきなので、electricity メソッドと acid メソッドを定義したくはありません。

Ruby で未定義のメソッドに応答するには、method_missing メソッドをオーバーライドします。Ruby では、未知のメソッドが呼び出された場合、method_missing メソッドを実行します。これはデフォルトで Object クラスから継承されており、例外を投げるメソッドです。ここでのトリックは、もっとおもしろいことをするために、この method_missing メソッドをオーバーライドできることです。

最初の resource メソッドの呼び出しの中で、electricity メソッドや acid メソッドへの応答の準備をします。

lairs/builder23.rb

```
def resource name, *attributes
  attributes << :name
  new_resource = Struct.new(*attributes)
  @configuration.add_resource_type name, new_resource
end
```

Rubyは、構造体（Struct）と呼ばれる匿名クラスを作る機能を持っています。資源が必要になったときに、それを構造体として定義します。resourceメソッドの第1引数で与えられた名前を設定し、第1引数に続く各引数をプロパティとして構造体に与えます。第1引数の名前もプロパティに加えます。最後に、構成オブジェクトの中にこれらの構造体を格納します。

次に、method_missingメソッドをオーバーライドして、メソッド名が新しい構造体の1つに対応しているかをチェックします。もし対応しているなら、メソッド呼び出しの引数に基づいて、対応する構造体を読み込みます。リテラルディクショナリが構造体の読み込みに役立ちます。

lairs/builder23.rb

```
def method_missing sym, *args
  super sym, *args unless @configuration.resource_names.include? sym
  obj = @configuration.resource_type(sym).new
  obj[:name] = sym
  args[0].each_pair do |key, value|
    obj[key] = value
  end
  return obj
end
```

筆者はmethod_missingメソッドを使うときにはいつでも、最初にメソッド呼び出し可能かを確認します。もし呼び出せない場合は、superメソッドを呼び出して例外を発生させます。

ほとんどの動的言語は、未知のメソッドを処理するハンドラをオーバーライドする機能を持っています。これは強力なテクニックですが、注意深く利用する必要があります。プログラムのメソッド判別システムを変更できるからです。深く考えずに利用すると、誰がプログラムを読んでも著しく混乱してしまうでしょう。

Jim Weirichが書いたRubyのbuilderライブラリは、method_missingメソッドの使い方を示す素晴らしい実例になっています。builderライブラリの目的はXML

マークアップを生成することです。そして、非常に読みやすいコードでXMLマークアップを生成するために、クロージャやmethod_missingメソッドを利用しています。

どれだけ素晴らしいか例をお見せしましょう。

lairs/frags

```
builder = Builder::XmlMarkup.new("", 2)
puts builder.person do |b|
  b.name("jim")
  b.phone("555-1234", "local"=>"yes")
  b.address("Cincinnati")
end
```

上記のコードは以下のマークアップを生成します。

lairs/frags

```
<person>
  <name>jim</name>
  <phone local="yes">555-1234</phone>
  <address>Cincinnati</address>
</person>
```

2.8 まとめ

2年前、Dave Thomasはブログ上で「コード＝カタ」という概念について語りました†。コード＝カタのアイデアは、異なる解決策の動作とそれらのトレードオフを検討するために、たくさんの方法で解くことができる単純な問題を提供するというものです。このエッセイは、これと似た取り組みになっています。このエッセイでは、最終的な結論は出しませんが、Rubyで内部DSLを作成する上で多くの方法を模索する手助けになるはずです。そして、これらの方法の多くは他の言語でも利用できるものになっています。

† 訳注：http://codekata.pragprog.com/

3章
言語の緑豊かな景観

Rebecca J. Parsons◎最高技術責任者

3.1 イントロダクション

植物学者は、植物でいっぱいの野原を歩き回りながら、その植物の多様性に驚き、そこで出くわすさまざまな植種を特定するでしょう。同様のことが、コンピュータ科学者にも当てはまり、コンピュータ言語の多様性に驚き、その言語特性に基づいて言語を分類します。言語に驚くべきほどの多様性があったとしても、言語特性を理解することは、素晴らしい言語の世界へこれから参入してくる言語を理解するための強力な基盤となります。

3.2 言語の標本

動植物に関する特性は、色、大きさ、葉の形状、花、実、トゲを扱います。それに対し、コンピュータ言語の特性は、利用できる文の種類や型の扱い方、言語自体の実装のされ方、プログラムの基本的な構成原理のような論点を扱います。動植物と言語の特性の種類におけるこの違いを考えれば、言語が多くのカテゴリに該当するのは驚くべきことではありません。その点では、植物の分類は、言語の分類よりもずっと単純なのです。このエッセイでは、言語の標本をいくつか調査し、言語の系統樹を作り上げていきます。古い言語も新しい言語も取り上げ、それぞれの言語の有名な特徴に特に注目していきます。

まず、長く使用されている言語 Fortran から見てみましょう。Fortran は、多くのバージョンを持ち、高度な科学技術計算に最も適する言語として昔から使用されてきた、非常にシンプルな言語特性を持つ古典的な言語です。代入文は、変数名を用いて記憶域の状態を操作します。代入以外の文は、記憶域の状態を参照し、計算を

行います。Fortranは、古典的な**命令型**プログラミング言語です。命令型言語は、文に対する基本的な構成機構がプロシージャであるため、**手続き型**言語と呼ばれることもあります。しかしながら、両者の言語特性は大きく異なるので、両者を区別しておくことは有用です。

Fortranは、**静的**言語でもあります。静的言語のプログラムは、コンパイルとリンクを行うことによって、ロードして実行できるようになります。コンパイラは、ソースプログラムを機械語に翻訳したり、最適化を行ったりします。さらに、プログラムが構文的に正しいかどうかを判定する責任もあります。

では、もう1つの古典的なプログラミング言語であるLispを見てみましょう。Lispは、人工知能の代名詞として見られていた時期もありましたが、人工知能よりも広い分野で用いられていました。LISPは「Lots of Insignificant Silly Parentheses(大量の無意味でくだらない丸括弧)」の頭字語であるとか、他にもいろいろなジョークが言われたりしますが、言語自体は注目すべき特性をいくつか備えています。Lispは**関数型**プログラミング言語ですが、関数型言語と手続き型言語を混同してはいけません。Lispプログラムの基本的な構成単位は、数学的な意味での関数です。純粋な関数は、パラメータとして渡されたデータを操作し、1つの値を返します。関数に同じ入力パラメータが与えられれば、返される値は必ず同じ値になります。そのため、純粋な関数は、ある呼び出しから次の呼び出しまで保存しておく記憶域や状態を持ちません。

Lispは**動的**言語でもあります。動的言語の特性は、ある特定の判断や計算が行われるタイミングに関するものです。動的言語は、コンパイラが行う機能の多くを実行時に行います。そのため、静的言語におけるプログラム開発の「コーディングして、コンパイルして、テストして、机を叩きつけて†、再度やり直す」というサイクルは、動的言語では「コーディングして、テストして、机を叩きつけて、再度やり直す」というサイクルになります。動的言語のプログラムは直接実行されるのです。CLRやJVMのような仮想マシンの普及によって厳密な区別がしづらくなってきていますが、動的言語という分類は重要な分類の1つです。

Lispは**動的型付け**の言語でもあります。動的型付け言語では、ある値の型は、文が実行されるまで決定されません。したがって、変数を特定の型として定義するという考え方がありません。変数の型は、値がなんであれ、現在保持している値の型になります。変数xは、あるときは整数値であるかもしれないし、次に参照したときはリストやブール値になっているかもしれません。動的言語と動的型付け言語は、

† 訳注:プログラムの実行結果が期待するものと違ったことを感情的なしぐさで表現している。

同じ言語の集合を表すものではありません。両者は異なる種類の言語特性をカバーしており、両者の実装に結びつきはありません。

では、もっと新しい言語であるJavaに移りましょう。Javaプログラムの主要な構成単位はオブジェクトであるため、Javaは**オブジェクト指向**（OO）プログラミング言語です。概念的には、オブジェクトは状態変数とメソッドの集合です。オブジェクトはクラスの一員であり、クラスはオブジェクトが持つ状態やメソッドを規定します。同じクラスのオブジェクトは互いに関連し合っていますが、それぞれは異なる実体です。実際に何がOO言語の必要十分条件かは議論のあるところですが、ほとんどの定義には、継承とカプセル化の特徴が含まれています。**継承**は、クラスひいてはオブジェクトを互いに関係づけるための手段です。サブクラスは、スーパークラスが持つすべての構成要素を受け継ぎます。サブクラスでは、新しいメソッドや状態を付加してスーパークラスのクラス定義を拡張したり、既存のメソッドのいくつかを再定義したりすることができます。**カプセル化**は、情報隠ぺいを実現するテクニックです。オブジェクトの実装の詳細は、実装への依存を減らすためにインターフェイスの背後に隠されるべきです。**ポリモフィズム**は、ある特定の関数がさまざまな型のオブジェクトを操作できるようにする機構です。ポリモフィズムはよくOO言語に必須の特徴だと思われていますが、OO以外の言語もポリモフィズムを備えています。同様に、カプセル化もOO言語だけの固有の特性ではありません。他の多くの言語でも、同様のアプローチが可能だからです†。

しかし、Javaは、CやC++のような波形括弧を使う言語と多くの構成要素を共有しているので、命令型の特性も持っています。そのため、Javaは純粋なOO言語ではありません。

Rubyは、今やIT雑誌で引っ張りだこの言語です。Rubyは、OO言語であり、動的言語であり、動的型付け言語でもあります。Rubyは、強力な拡張メカニズムを持っています。また、関数型プログラミングやさらには手続き型プログラミングまでもサポートしています。

ではHaskellを見てみましょう。Haskellはあまり知られていない言語ですが、徐々に認知されるようになってきています。Haskellは、Lispと異なり、命令型の構成要素をまったく持たないという意味で、純粋な関数型言語です。Haskellは**静的型付け**であり、**型推論**を用いることで明示的な型宣言の数を減らしています。また、Haskellは言語セマンティクスが**遅延評価**であるという点で、他の言語と大きく異な

† 訳注：例えば、関数型言語ではクロージャの機能を使ってカプセル化（情報隠ぺい）を実現することができる。

ります。遅延評価型言語とは、プログラムの最終結果を決定するのに必要でないものは何も評価しないという言語です。この遅延評価は、データ項目の各要素にも適用されます。つまり、リストの最初の要素が必要なときは、最初の要素しか計算されないということです（残りの要素は計算されません）。実行セマンティクスにおけるこの違いは、Haskellに関する最近の言及ではよく見落とされていますが、遅延評価セマンティクスがあるために、Haskellによるプログラミングは他の関数型言語とは大きく異なります。

まったく異なる言語の一例であるSQLを見てみましょう。SQLはリレーショナルデータベースのデータにアクセスするための一般的な問い合わせ言語です。SQLは宣言型言語です。宣言型言語のプログラムは、どうやって計算するかではなく、何を計算すべきかを指示します。SQLであれば、文はデータ検索の実行方法ではなく、目的とするデータの特性を定義します。また、Prologもよく知られた宣言型言語です。Prologのプログラムは、公理と推論ルールからなる論理的表明で構成されており、それによってシステムの状態を表現します。Prologプログラムの実行は、公理と推論ルールによって定義されるシステムの状態に対して、一連の表明が論理的に証明可能かという問いに答えることで構成されています。Prologの計算モデルは、実際の計算手順を記述する抽象的な計算機のモデルではなく、表明から結論を引き出すアルゴリズムなのです。

宣言型言語のこのような定義は役に立たないと、多くの人が思っています。けれども、その定義について考える1つの方法は、ある1つの文に着目し、起こりうる計算について決定できることは何かを考えることです。非宣言型言語の文では、どのような計算が起こるかを指定します。宣言型言語の文では、期待される結果のある側面を指定します。宣言型言語は、結果をどうやって決定するかという概念を持たないからです。確かに論点は、まだぼやけているところがあります。さらに、非宣言型言語の抽象度が上がり、またコンパイラが行うさまざまな最適化の程度が上がるにつれて、その違いがはっきりとしなくなり、関心も薄らいでいくことになるでしょう。

最後に、現在人気上昇中の言語であるErlangを見てみましょう。Erlangは関数型で正格評価の動的型付け言語で、並列計算を言語として明示的にサポートしているという特徴を持ちます。この節で取り上げた他のすべての言語は、逐次型の言語です。逐次型の言語でも、スレッドを使用したり、メッセージング層を導入したりすることで並列実行をサポートできますが、Erlangのプログラムでは、明示的に並列実行を記述して、明示的にメッセージを用いて通信を行います。

3.3 種類の多種性

これまでの言語を眺めてきただけで、言語の構成原理、型システム†、実行時の振る舞い、言語の実装のされ方に関するカテゴリが明らかになりました。しかし、汎用プログラミング言語に限ると、このリストからはとても重要なものが抜け落ちています。条件文と無限ループをサポートする言語はみな、チューリング完全だと言えるのです。チューリング完全な言語はすべて、有限個の入力を取り、有限個の出力を返す処理を有限時間内に完了するプログラムであれば、どんなものでも表現することが可能です。

なぜチューリング完全性を考える必要性があるのでしょうか。人は自分が推薦する言語をアピールすることに熱心ですが、Lisp、Ruby、Java、C#、C などを含むすべての主要な汎用プログラミング言語や、とっつきにくい枯れたアセンブラ言語でさえも、同じプログラムを表現できるのです††。そう、まったく同じプログラムです。

しかし、言語にかかわらずどんなプログラムでも書けるからといって、その言語でプログラムを書くことが必ずしも簡単であったり、適していたりするというわけではありません。問題が異なれば、求められる解決方法も異なります。言語が異なれば、抽象化の手法や技術的アプローチが異なり、それによって解決方法に違いが出ます。同じアルゴリズムを異なる言語で実装すると、まるで違うものに見えるのはもちろんですが、処理が大きく異なることもあるでしょう。例えば、ある言語では、そのアルゴリズムをもっと効率的に実行できるかもしれません。別の言語では、そのアルゴリズムをもっときれいに実装できるかもしれません。コンパイラによる最適化は、コンパイルされる言語によっては難しいこともあれば簡単なこともあります。言語特性の違いと言語がサポートしているプログラミングモデルを理解すると、特定のタスクに適した言語を選択できるようになります。では、このような多種性のさまざまな側面を取り上げ、その選択肢について見ていきましょう。

† 訳注：プログラミング言語が、型をどのように扱うかを定義したもの。型システムは、プログラミング言語で扱う型がどのような値を取りうるかだけでなく、型に対してどのような操作が行え、また型どうしの演算がどのような型に評価されるかなどの仕様を表す。

†† 訳注：チューリング完全な言語であれば、ある言語で記述できるプログラムの総体は、他の言語で記述できるプログラムの総体と同じになるということを表している。

3.3.1 パラダイム

プログラミングパラダイムには、一般的なものを挙げると命令型、手続き型、関数型、オブジェクト指向、宣言型、論理型があります。言語の中には、複数のパラダイムの側面を持ち合わせているものがあります。例えば、Common Lispは、関数型言語ですが、OOの概念もサポートしています。C++は、OOの特徴だけでなく手続き型言語の多くの特徴も持っています。次の表は、主要なプログラミングパラダイムを示しています。

分類	定義	例
命令型	一連の文は状態を操作する	アセンブラ、Fortran
手続き型	文をまとめたプロシージャで構成される	C、Pascal、Cobol
オブジェクト指向	オブジェクトで構成される	Smalltalk、Java、Ruby
関数型	状態を持たない関数で構成される	Lisp、Scheme、Haskell
論理型	解を求めるのに公理と推論ルールを用いる点が特徴である	Prolog、OPS5
宣言型	解を求める方法ではなく、解を記述する	XSLT、SQL、Prolog

この表は、共通性の少ないものが組み合わさっているように見えるかもしれません。実際のところ、この表には3つの異なる特性が入っています。

表の中には、構成、状態、スコープの概念があります。前述したように、言語パラダイムが異なれば、コードを構成する方法が異なります。オブジェクト、関数、プロシージャ、あるいは単なる文が、それぞれ基本の構成単位として用いられます。それから状態の概念があります。命令型言語は明示的に記憶域の状態を操作しますが、関数型言語は状態の操作を一切行いません。最後に、状態に対する可視性の概念があります。オブジェクト指向言語では、状態はオブジェクトの中にあり、状態はそのオブジェクト自身だけが見えるようになっています。命令型言語は、プログラムのあらゆるところから見えるグローバルな状態を持っています。関数型言語は、関数のクロージャ†の中で変数に値を結合しますが、この変数値を他の関数から変更したり見たりすることはできません。

† 訳注：クロージャとは、ある関数内で定義したコードブロックとそのコードブロック内で使用されている変数の状態（値）をカプセル化したもの。クロージャは、無名関数として定義されることが多く、そのクロージャを関数の引数に渡したり、関数の戻り値として返すことで、利用者が関数の挙動をカスタマイズできるようになる。クロージャの機能を持たないC言語やJava言語では、関数ポインタや無名クラスを用いることでクロージャと類似したものが実現できるが、そのコードブロックが定義された環境（変数の状態）を引き継がないという点で正式なクロージャとは異なる。

3.3.2 型特性

変数識別子の型は、その変数が保持できる適正な値を規定します。さらに、その変数に対して実行されうる操作も暗黙的に規定します。型システムにはいくつか種類があり、それぞれが異なる型特性を持ちますが、最も頻繁に議論される型特性は、変数の型が決定されるタイミングに関することです。静的型付け言語では、たいていコンパイル時に、変数に単一の型を割り当てます。動的型付け言語では、変数の型は、その変数に対して操作が実行されようとしているときに決定されます。動的型付け言語では、与えられた変数の型は、実行中に変わる可能性があります。このようなことは、静的型付け言語では発生しません。動的型付け固有のスタイルとして、例えばRubyが採用している**ダックタイピング**†があります。ダックタイピングでは、値の型に対する考え方が少し緩められています。値の型は、特定の型と一致する必要はなく、要求される操作を実行できさえすればいいのです。

もう1つの重要な用語であり、非常に乱用されている用語として、**強い型付け**があります。強い型付け言語に対する一般的な定義は、実行時に型エラーを起こす可能性があるプログラムはコンパイルを通さない、というものです。もちろん、コンパイルを通らなかった場合には型定義エラーとなります。ゼロによる除算なども型エラーとして考えることができますが、一般的には、型エラーは文字列や配列に対する算術演算といったものに限定されます。

型付けに関して最後に着目したい側面は、型推論です。型推論を使用すると、プログラムが正しい型付けとなるようにコンパイラが推論を試みます。型推論によって、強い型付けの言語がもっと書きやすいものとなります。しかし、完全に正しいプログラムであっても、型推論アルゴリズムが正しい型を推論できない場合もあります。

次の表は、普及している言語のいくつかを取り上げ、それらの言語がこれまで説明してきた型特性のどこに分類されるかを示しています。

† 訳注：ダックタイピングは、同じ操作を持っているオブジェクト同士は、それぞれがどのような継承階層にあろうとも、相互に交換可能であることを意味する型付けである。この名称は、「もし、ある鳥がアヒルのように見え、アヒルのように泳ぎ、アヒルのように鳴くならば、その鳥はたぶんアヒルである」と説明されるダックテストに由来している。

分類	定義	例
静的型付け	変数の型が固定されている（通常コンパイル時）	Java、C、Fortranなど
動的型付け	変数の型はアクセスされるときに決定される	Scheme、Lisp、Ruby
強い型付け	実行時に型エラーが発生する可能性がない	Haskell、C++（型キャストを無視した場合）
型推論	明示的に型定義をしなくても、型推論アルゴリズムによって変数に型が割り当てられる	Haskell、ML
ダックタイピング	必要な型の一部分だけがチェックされる	Ruby

3.3.3 実行時の振る舞い

振る舞いについては多くの分類がありますが、この節では2つの異なる分類について議論します。1つは、逐次型言語と並列型言語の違いです。もう1つは、遅延評価型言語と正格評価型言語の違いです。

クライアントサーバモデルの出現によって並列処理の概念がある程度導入されたものの、ほとんどのプログラマは並列計算についてあまり意識しません。こうしたことから考えて、ほとんどの言語が逐次型であることは驚くことではありません。逐次型言語の実行セマンティクスというのは、一度にプログラムの1つの文を実行することを前提にしています。しかしながら、何十年もの間、並列化されたマシンを利用してきていることもあり、多くのアプリケーションがタスクを実行するために並列計算や分散計算に頼るようになってきています。コンパイラが、並列処理可能な箇所を自動で推測することもあります。また、並列処理を取り入れるために、メッセージング層、タスク層、ロックやセマフォのライブラリが使用されることもあります。言語の中には、並列処理のサポートを言語の構成要素としてうたっているものもあります。以前は、学術調査や複雑な財務分析のアプリケーションに限定されていましたが、マルチコアプロセッサの一般化やアプリケーションに対するユーザの期待の膨らみによって、今では並列処理をサポートする言語が主流になりつつあります。

遅延評価は、業務アプリケーションでは検討すらされませんが、Haskellへの関心が高まるにつれて、遅延評価のパワーが理解されるようになるでしょう。ほとんどすべての言語が正格評価のセマンティクスを持ちます。正格評価というのは、簡単に言うとプログラムの終わりまで文が順番に実行されるというものです。遅延評価というのは、概念的に言うと、実行結果が何であるかを決定し、その実行結果を得るにはどの文を実行する必要があるかを判断することから実行を開始するというものです。最終結果を求めるのに直接必要のない文は、無視されます。架空のプログラミング言語で書かれた、次のような明らかにおかしなプログラムを考えてみましょう。

```
X = Y/0;
Y=6;
End(Y);
```

最後の文は、必要な最終結果がYの値であることを示しています。正格評価の環境下では、このプログラムはゼロ除算の例外で失敗するでしょう（ここでは、コンパイラの最適化がデッドコード†を除去するケースを考慮していません)。遅延評価の環境下では、コンパイラの最適化がなかったとしても、このプログラムは6という値を返すことになるでしょう。結果を得るのにXの値を決定する必要はないため、Xの値を決定する文は無視されます。遅延評価が意味するところは、単なる最適化だけにとどまりません。例えば、遅延評価は、無限のデータ構造を操作したり、定義したりすることを可能にします。しかし、この内容はこのエッセイの範囲を超えているためこれ以上は扱いません。

この節では表は必要ありません。Haskellとその同類であるHope言語††という目立った例外を除き、ほとんどの言語は正格評価だからです。特定のアプリケーションのために正格評価型言語で遅延評価を実現することは可能です‡が、その言語のセマンティクスは正格評価のままです。

3.3.4 実装モデル

このエッセイで取り上げる最後のカテゴリは、言語の実装のされ方に関係するものです。以前は、**コンパイル型言語**と**インタプリタ型言語**という2つの大きなカテゴリが広く知られていました。「昔」は、インタプリタ型言語は遅かったので、実業務となるとコンパイル型言語を用いる必要がありました。あるいは、少なくとも当時の通念上はそう考えられていました。信じられない人もいるでしょうが、当時でさえ、Lispを使って実システムを書いている人もいました。インタプリタ型言語は、お遊び言語や単なるスクリプト言語と考えられていました。仮想マシンの普及によって、多少その境界があいまいになってきています。例えばJavaのような言語では、プログラムはバイトコード表現にコンパイルされて、その後に仮想マシン上でインタプリタとして実行されます。しかし、まだ次の区別は役立つと思われます。

† 訳注：プログラムのソースコードに存在するが、決して実行されないコードを表す。到達不能コードとも呼ばれる。

†† 訳注：Hope言語は、パターンマッチングを最初に導入した関数型プログラミング言語である。高階関数、多相型、代数的データ型の特徴も持っている。

‡ 訳注：例えば、クロージャの機能を持つプログラミング言語であれば、クロージャを使うことで遅延評価が実現できる。

コンパイラはソースプログラムから何かを生成しますが、結果を得るにはそれを実行する必要があります。一方、インタプリタはソースプログラムを読み込み、そこから直接結果を生成します。

特に「スクリプト言語」というフレーズは、その言語がプログラムではなく単にスクリプトを書く用途にしか使えないものだ、という印象を与えるものでした。しかし、多くのシステムがスクリプト言語で実装されるようになってきたことから、そうした印象は無意味なものになりつつあります。

言語の中には、例えばSchemeのように、コンパイル型とインタプリタ型の両方のバージョンを持つものもあります。こうした言語がまた、コンパイル型かインタプリタ型かという特徴をあいまいにしています。

この節でも、表は意味を持ちません。コンパイル型とインタプリタ型の両方のバージョンを持つ言語がたくさんあり、またバイトコード形式の仮想マシンが使われることでその論点がさらに複雑になるからです。

3.4 言語の系統樹

次の表は、いくつかのよく知られている言語と、各言語がどのカテゴリに関係し位置するかを示しています。多くの言語が、構文は大きく異なるにもかかわらず、非常に似た特性を持っています。プログラム設計という観点からは、構文の違いはあまり問題になりません。言語の表現能力は、構文よりも言語特性によるところが大きいからです。しかしながら、一般にプログラマは構文にこだわりを持っています。これは、プログラマが言語の構文を直接扱わなければいけないことを考えれば、当然のことです。統合開発環境（IDE）の再流行でプログラマの構文的な負担は軽減されてきていますが、その負担がなくなってしまうことはありえません。

このカテゴリの一覧は、網羅的ではありません。言語の特徴については他にも多くの観点で検討することもできるでしょう。また、古い言語が備えている多くの特徴（クロージャや高階関数†など）が最近の言語に現れつつあります。言語は、新しい抽象化の手法やアプローチを取り入れることによってこれからも発展し続けるでしょう。

† 訳注：関数を引数にとる関数や関数を戻り値として返す関数のこと。関数型言語において多用される。

言語	パラダイム	型付け	実装
アセンブラ	命令型	動的型付け	アセンブル型、逐次型
Fortran	命令型	静的型付け	コンパイル型、並列処理のコンパイラを備えた逐次型
C	命令型、手続き型	ポインタを除けば静的型付け	コンパイル型、逐次型
C++	命令型、OO、手続き型	ポインタを除けば静的型付け	コンパイル型、並列処理のコンパイラを備えた逐次型
Java	命令型、OO、手続き型	静的型付け	コンパイル型、並行処理のスレッド機能あり
Lisp	命令型の一部も合わせ持つ関数型（CLOSにはOOが導入されている）	動的型付け	インタプリタ型とコンパイル型の両方あり、逐次型
Scheme	命令型の一部も合わせ持つ関数型	動的型付け	インタプリタ型とコンパイル型の両方あり、逐次型
Haskell	関数型、遅延評価型	静的型付け、型推論	インタプリタ型とコンパイル型の両方あり
Ruby	OO	動的型付け、ダックタイピング	インタプリタ型とコンパイル型の両方あり、逐次型
Prolog	宣言型	動的型付け	
Scala	関数型、OO	静的型付け	インタプリタ型とコンパイル型の両方あり、並列処理のサポートあり
Erlang	関数型	動的型付け	インタプリタ型とコンパイル型の両方あり、並行処理あり

3.5 どれも興味深いことであるが、どうして重要なのか

何十年にもわたって言語論争が激しく行われてきましたが、この論争はこれからも続くでしょう。中には、1つの完璧な言語が存在すると信じている人もいますが、これはまったくでたらめな考えだと筆者は思います。ドメイン固有言語（DSL）が今後も注目を集め続けることを認め、インテンショナルプログラミング†や言語ワークベンチ††のような新しいアプローチが現実によい結果をもたらすというのなら、唯一の完璧な言語が存在するという考えはさらに矛盾したものとなります。実際のところ、個々の言語が特定の種類のコンポーネントやプログラムを実装できるだけ

† 訳注：Charles SimonyiがMicrosoft時代に提唱したコンセプト。プログラマが頭の中で考えている意味論的レベルの情報（インテンションと呼ばれる）をソースコードに反映しながら開発が行える、拡張可能なプログラミング環境のこと。Microsoftは、インテンショナルプログラミングに関する研究開発を2000年代初期に中止したが、インテンショナルプログラミングの思想は、Microsoft退社後に立ち上げた会社で開発しているIntentional Softwareに引き継がれている。

の一連の機能を持ち、その機能に関する構文を持つべきです。言語の適用できる範囲が広がれば、それだけその言語が汎用目的の言語になるのは明らかです。もちろん、保守が必要な多くのアプリケーションを抱えたCIO（最高情報責任者）の観点から考えると、これしかないという言語が1つだけあれば、少なくとも表面上は、要員調達に関する課題を大幅に単純化することになるでしょう。JavaとRubyのどちらが優れているか、という議論はこれからも続くでしょう。これはJavaとRubyから他の言語に移ったとしても同じことです。筆者は、利用すべき言語だけを単に議論するのでなく、その問題にはどの言語を利用すべきかという議論ができる日が来るのをずっと待ち望んでいます。

艹 訳 注：Martin Fowlerが"*Language Workbenches: The Killer-App for Domain Specific Languages?*"の記事を執筆する際に考案したコンセプトで、DSLを使ったソフトウェア開発を支援する開発環境の総称のこと。開発環境内にユーザが自由に新しいDSLを定義でき、定義したDSLを使ってソフトウェアを開発できるところに特徴がある。言語ワークベンチというカテゴリに属するツールとして、Intentional Software、Meta Programming System、Software Factories、MDA（モデル駆動アーキテクチャ）などがある。

4章
多言語プログラミング

Neal Ford◎ミームラングラー

今後10年間、すべての人はSmalltalkで
プログラミングすることになるでしょう、
人々がそれをなんと呼ぼうとも。
——Glenn Vanderburg

1995年にJavaが現れたとき、Javaは、ポインタやメモリ管理、その他の難解な配管工事のような仕事と戦い疲れたC++プログラマを救済する言語として歓迎されました。Javaは、そうしたプログラマの難題を解決してくれました。しかし、Javaの成功のためには、当時の開発者にとって高位の聖職者ともいえるC++開発者にアピールする必要がありました。したがって、Javaの設計者が意図的にJava言語の見た目をC++に似せたことには意味がありました。もう一度基礎からすべてを学び直さなければならないとしたら、開発者を新しい言語に引きつけることはできませんから。

しかし、2008年現在、もはや下位互換性は意味がありません。新米のJava開発者が学ぶたくさんの奇妙な仕様は、本来片付けるべき仕事に関係のない単なる不思議なおまじないとなっています。多くのJava開発者が最初に出会う以下のコードについて考えてみましょう。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

このコードを理解するために新米の開発者に説明しなければならないことは、いくつあるのでしょうか。Javaは、0で始まる配列のようなC++主義や、プリミティブ型とオブジェクトの区別などの1995年には意味があったけれども最近の開発者の生産性に役立たないお荷物であふれかえっています。

幸運なことに、Javaの設計者はJavaを作成するときに素晴らしい決断を行いました。彼らはプラットフォームから言語を分離したのです。これにより、多言語プロ

グラミングと呼ばれる「刑務所からの釈放」†カードが開発者に与えられたのです。

4.1 多言語プログラミングとは

多言語という言葉は、多くの言語を話すことを意味します。多言語プログラミングは、特定の問題を解くための特別な言語を使うために、Javaの（そしてC#も同様に）言語とプラットフォームの分離を利用します。現在、Java仮想マシンや.NETのマネージド・ランタイム上で動作する言語はたくさん存在します。しかし、開発者として、私たちはこの可能性を十分に活かすことができていません。

もちろん開発者は昔からさまざまな言語を使っています。多くのアプリケーションでは、データベースにアクセスするためにSQLを使いますし、Webページを対話的にするためにJavaScriptを使いますし、もちろんあらゆるものを設定するためのXMLもそこら中にあります。しかし、多言語プログラミングのアイデアはこれらと異なります。これらの例はすべて、JVMと直交しています。つまり、これらの言語はJavaの世界の外側で動作します。そしてそれが大きな頭痛の種になります。何十億ドルのお金がオブジェクトと、集合論に基づくSQLのインピーダンスミスマッチに費やされていることでしょうか。開発者は複数の言語を使っている点ですでに苦痛を味わっているので、インピーダンスミスマッチは開発者をいらいらさせます。しかし、多言語プログラミングはこれらの例とは異なります。多言語プログラミングは、インピーダンスミスマッチを取り除き、JVMの内部でバイトコードを生成する言語を活用することなのです。

多言語プログラミングを導入することの他の障害は、言語を変えることへの気持ちの問題です。昔は、言語を変更することはプラットフォームを変更することと同じであり、すべてのライブラリを書き直したくない開発者にとっては、明らかに悪い意味を持っていました。しかし、JavaとC#において、言語とプラットフォームを分離したことにより、もはやこのように苦闘する必要がなくなりました。多言語プログラミングにより、既存のすべての資産を活用し、目の前の仕事を片付けるためにより適切な言語を使うことができます。

目の前の仕事を片付けることに適しているというのは、どういう意味でしょうか。次の節では、多言語プログラミングを適用した例をいくつか紹介します。

---

† 訳注：モノポリーの刑務所脱出カード。モノポリーでは、刑務所に投獄されたユーザは50ドル払うか、ダイスを降ってゾロ目を出して釈放されるか、このカードを使って刑務所から脱出する。

4.2 Groovyの方法でファイルを読む

テキストファイルを読み、それぞれの行の先頭に行番号を付けてテキストファイルの内容を出力する単純なプログラムを書くタスクがあります。Javaによる解決策を以下に示します。

./code/ford/LineNumbers.java

```
package com.nealford.polyglot.linenumbers;

import java.io.*;
import static java.lang.System.*;

public class LineNumbers {
    public LineNumbers(String path) {
        File file = new File(path);
        LineNumberReader reader = null;
        try {
            reader = new LineNumberReader(new FileReader(file));
            while (reader.ready()) {
                out.println(reader.getLineNumber() + ":"
                        + reader.readLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException ignored) {
            }
        }
    }

    public static void main(String[] args) {
        new LineNumbers(args[0]);
    }
}
```

JVM上で動作するスクリプト言語であるGroovyによる同じ解決策を以下に示します。

./code/ford/LineNumbers.groovy

```
def number=0
new File (args[0]).eachLine { line ->
    number++
    println "$number: $line"
}
```

単純なタスクでは、Javaは重々しく、必要以上に複雑になります。先の例では、問題を解決するためのコードよりも定型的な例外処理コード行のほうが多くなっています。Groovyは、例外処理のようなよくある下準備をほとんど処理します。その結果、すべてのJavaのおまじないを使うことなく、解決策をより簡単に見せています。このコードはJavaバイトコードにコンパイルされ、最終的にはJavaのコードとほとんど同じ結果になります。確かに、Groovy版のバイトコードは、効率的ではないでしょう。Groovyは言語をより動的にするために、Javaのバイトコードを使ったたくさんの魔法（プロキシオブジェクトを経由してJavaクラスを呼び出すような）を扱う必要があります。しかし、この状況にはもっと重要なことがあります。それは開発者の生産性か、実行効率かということです。行番号を追加するためにファイルを読み込み、一方では500ミリ秒かかり、他方では100ミリ秒かかったとして、誰が気にかけるでしょうか。コードを書く時間を節約することで、数千倍もの時間を節約します。時期尚早のパフォーマンスの最適化を行うことよりも、仕事により適したツールを使いましょう。

4.3 JRubyとisBlank

もちろん先ほどの例は、Javaがあまり適していない単純なスクリプティングでした。何かより一般的なもの、例えばWebアプリケーションにおいて、パラメータが空白でないことを検証するコードが必要ならどうでしょうか。

このJavaコードはApache Commonsプロジェクト由来のものです。Apache Commonsプロジェクトは、Javaで共通で使われ、インフラストラクチャとなるコードのリポジトリです。このコードでは、Webアプリケーションパラメータを取得する際にいつも起きること、つまり文字列が空白かどうかを判定できます。以下のコードは、StringUtilsクラスのisBlank()メソッドです。

./code/ford/StringUtils.java

```
public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
        return true;
    }
    for (int i = 0; i < strLen; i++) {
        if ((Character.isWhitespace(str.charAt(i)) == false)) {
            return false;
        }
    }
    return true;
}
```

このコードは、Java言語に組み込まれた多くの欠陥を明らかにしています。最初の欠陥は、StringUtilsと呼ばれるクラス自体にあります。なぜなら、Java言語はStringクラスの変更や拡張を認めていないからです。これは、「不適切なところにぶら下がったメソッド」（本来あるべきクラスに含まれていないメソッドをこう呼ぶ）の例です。次に、送られてきたオブジェクトがnullでないことを検証する必要があります。nullはJavaにおいて特別です（nullはプリミティブでもオブジェクトでもありません）。多くのJavaコードでは、nullに対するチェックが必要です。最後に、文字列の残りの文字がすべて空白か判定するために、文字列に対する繰り返し処理が必要です。もちろん、この判定を行うために、それぞれの文字のメソッドを呼ぶことはできません（なぜならJavaでは文字はプリミティブだからです）。つまり、Characterのラッパークラスを使う必要があります。

JRubyで同じ動作をするコードを示します。

./code/ford/blankness.rb

```
class String
  def blank?
    empty? || strip.empty?
  end
end
```

そしてそれを検証するテストです。

./code/ford/test_blankness.rb

```
require "test/unit"
require "blankness"

class BlankTest < Test::Unit::TestCase
  def test_blank
    assert "".blank?
    assert " ".blank?
    assert nil.to_s.blank?
    assert ! "x".blank?
  end

end
```

この解決策では何点か注目すべきことがあります。最初に、Rubyは直接Stringクラスにメソッドを追加することができるということです。今回のメソッドは適切な場所に落ち着いています。第二に、コードがとてもシンプルです。なぜなら、プリミティブ型とオブジェクトについて気にする必要がないからです。第三に、テストに注目すると、nilの場合を気にする必要がありません。Rubyのnilは、Ruby言語の他のすべてのものと同じようにオブジェクトです。そのため、nilを文字列として渡したとき、to_s()メソッド（Ruby版のtoString()メソッド）は空の文字列、つまり空白を返します。

JavaのStringクラスは、Javaの世界ではfinalクラス†なので、このコードをJavaに組み込むことはできません。しかし、JRubyでRuby on Railsを使っているのなら、Javaの文字列に対し同じコードを組込むことができます。

4.4 Jaskellと関数プログラミング

これまでに示したほとんどの例は、Javaの言語上の欠陥を扱っています。しかし、多言語プログラミングは、基本的な設計の決定も取り扱います。例えばスレッドは、JavaやC#のような命令型言語では実装が難しいのです。synchronizedを使ったときのニュアンスや副作用、共有データに複数のスレッドがアクセスするときの事情の違いを理解しなければいけません。

多言語プログラミングでは、Jaskell（HaskellのJavaバージョン）やScala（JVM用に書かれたモダンな関数型言語）のような関数型言語を使うことによって、この

† 訳注：Javaではfinalで装飾されたクラスは、サブクラス化できない。

問題を完全に避けることができます。

関数型言語には、命令型言語の多くの欠点がありません。関数型言語は、数学的な原則により厳格で忠実です。例えば、関数型言語における関数は、まさに数学の関数のように動作します。つまり、出力が完全に入力だけに依存しています。言い換えると、関数が動作するとき、関数は外部の状態を修正することができません。また、sin() のような数学関数を呼び出すとき、いくつかの内部状態が修正されたために次の sin() の呼び出しが偶然にコサインを返すかもしれないと心配する必要もありません。数学関数は、呼び出しと呼び出しの間で変更できる内部状態を持ちません。関数型言語における関数（とメソッド）はどれも同じように動作します。関数型言語の例は Haskell、O'Caml、SML、Scala、F# その他のものを含んでいます。

特に、関数型言語はステートフルを推奨しないので、関数型言語は命令型言語よりうまくマルチスレッドサポートを扱います。この議論の要点は、命令型言語より関数型言語は、堅牢でスレッドセーフなコードを書くことが簡単だということです。

それでは Jaskell について詳しく見てみることにしましょう。Jaskell は Java プラットフォームで動作する Haskell 言語の 1 バージョンです。言い換えると、Java バイトコードを生成する Haskell コードを書く方法です。

例を示します。配列の要素に安全にアクセスするクラスを Java で実装するとしましょう。以下のようなクラスを書くことができます。

./code/ford/SafeArray.java

```
class SafeArray{
    private final Object[] _arr;
    private final int _begin;
    private final int _len;

    public SafeArray(Object[] arr, int len){
        _arr = arr;
        _begin = 0;
        _len = len;
    }

    public Object at(int i){
        if(i < 0 || i >= _len){
            throw new ArrayIndexOutOfBoundsException(i);
        }
        return _arr[_begin + i];
    }
```

```
    public int getLength(){
        return _len;
    }
}
```

Jaskellでは、タプル†として同じ機能を書くことができます。タプルは本質的に連想配列です。

./code/ford/safearray.jaskell

```
newSafeArray arr begin len = {
  length = len;
  at i = if i < begin || i >= len then
           throw $ ArrayIndexOutOfBoundsException.new[i]
         else
           arr[begin + i];
}
```

タプルは連想配列として動作するので、newSafeArray.at(3) を呼び出すことは、タプルのat部を呼び出し、タプルの部分で定義されたコードを評価します††。Jaskellはオブジェクト指向言語ではありませんが、継承やポリモフィズムもタプルによって擬似的に実現できます。さらにいくつかの魅力的な振る舞い、例えばミックスイン‡のようにJava言語のコアだけを使って実現できないことが、Jaskellのタプルを使って実現できます。ミックスインは、インターフェイスと継承を組み合わせた機能を別の方法で提供します。ミックスインは、継承することなしに、クラスにシグニチャだけでなくコードを挿入することができます。本質的にミックスインは、継承することなしにポリモフィズムを提供します。

Haskellは関数の遅延評価を特色とします（したがってJaskellも同様です）。つまり、必要とされるまで関数の結果は決して評価されないということです。例えば、Haskellでは完全に正しい以下のコードですが、Javaでは決して動作しません。

---

† 訳注：タプルは、2つ以上の値をまとめたもの。異なる型やオブジェクトをまとめるために使う。

†† 訳注：ここでは、lengthとat関数をメンバに持つnewSafeArrayタプルを定義している。「.」でタプルのメンバ名を指定し、再参照できる。newSafeArray.at(3)は、newSafeArrayというタプルのメンバであるat関数に引数3を渡し、at関数を評価せよという意味。

‡ 訳注：ミックスインとは、実装を持つが単体で動作することはなくクラスに取り込まれて使用されることを前提としたクラス、もしくはその機能を指す。Javaのインターフェイスに実装が含まれているイメージに近い。

```
makeList = 1 : makeList
```

コードは「1つの要素を持ったリストを作りなさい。もしより多くの要素が必要なら、必要に応じてそれらを評価しなさい。」と読みます。この関数は本質的に、無限の1のリストを作成します。

ひょっとしたら読者の皆さんは、Javaでは1,000行のコードだが、Haskellではたった50行となるような、複雑なスケジューリングアルゴリズムを持っているかもしれません。Javaプラットフォームを活用して、そのタスクにより適した言語で書いてはどうでしょうか。やがては、プロジェクトにデータベースアドミニストレータを迎えるのと同様に、それぞれの専門に特化したコードを書くスペシャリストを迎えるようになるでしょう。

Jaskell単独でアプリケーション全体を書くことはありえません。しかし、より大きなアプリケーションでJaskellが本当に得意としている特長を活用してはどうでしょう。仮に、高度に並列スケジューリングが必要なWebアプリケーションがあるとしましょう。Jaskellでスケジューリング部分を書き、JRubyを使ってRailsもしくはGroovy on GrailsでWeb部分を書き、そして古いメインフレームと通信するために、既存のコードを利用しましょう。Javaプラットフォームのおかげで、それらをすべてバイトコードレベルで1つにつなげることができます。その結果、生産性が高まるでしょう。なぜなら解決する必要のある問題に適したツールを使っているからです。

4.5 Javaのテスト

複数の言語を使うことイコールすでに行っていることを捨て去る必要があるとはかぎりません。既存のインフラストラクチャでさえも、より適した言語で利用することができます。複雑なコードのテストなどは、誰もが共通で行う必要があるタスクでしょう。しかし、期待値のモックオブジェクトを作成するには、時間がかかります。なぜならJavaは、あるオブジェクトが他のオブジェクトをまねるときに必要となる柔軟性をあまり持っていないからです。なぜ、テストを（もっと言うならテストだけでも）、より適した言語で書かないのでしょう。

ここにJavaで人気のあるモックオブジェクトライブラリのJMockを使った、Orderクラス（実際にはクラスとインターフェイス）とWarehouseクラスの間の相互作用をテストする例があります。

./code/ford/OrderInteractionTester.java

```
package com.nealford.conf.jmock.warehouse;

import org.jmock.Mock;
import org.jmock.MockObjectTestCase;

public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new OrderImpl(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

}
```

このコードは、Order クラスが（そのインターフェイスを通して）Warehouse クラスと正確に相互作用していることを、適切なメソッドが呼び出され、結果が正しいことを確認することにより、テストしています。

以下に同じテストを実行する JRuby（と、Ruby 界のパワフルなモックオブジェクトライブラリである Mocha）を使った同じテストがあります。

./code/ford/order_interaction_test.rb

```
require 'test/unit'
require 'rubygems'
```

```
require 'mocha'

require "java"
require "Warehouse.jar"
%w(OrderImpl Order Warehouse WarehouseImpl).each { |f|
   include_class "com.nealford.conf.jmock.warehouse.#{f}"
 }

class OrderInteractionTest < Test::Unit::TestCase
  TALISKER = "Talisker"

  def test_filling_removes_inventory_if_in_stock
    order = OrderImpl.new(TALISKER, 50)
    warehouse = Warehouse.new
    warehouse.stubs(:hasInventory).with(TALISKER, 50).returns(true)
    warehouse.stubs(:remove).with(TALISKER, 50)

    order.fill(warehouse)
    assert order.is_filled
  end

end
```

Ruby は動的言語なので、このコードはとても簡潔です。JRuby は Java オブジェクトをプロキシクラスでラップするので、直接インターフェイス（この場合 Warehouse ですが）をインスタンス化できます。また注目するべきことは、先頭で単に require 'warehouse.jar' とすることで、該当するすべての Java のクラスをテストのクラスパスに置くことができることです。これを Java でもできるようにしたいと思いませんか。

多言語プログラミングは主流である必要がありません。つまり、仕事の仕方を土木工事のようにすっかり取り替える必要はありません。たいていの会社では、テストコードは「正式な」コードとして考えられていないので、テストを JRuby で書き始める許可を得る必要さえないかもしれません。

4.6 多言語プログラミングの未来

2007 年初頭、ThoughtWorks 社は Mingle と呼ばれる初の商用製品をリリースしました。それは、アジャイルなプロジェクトトラッキングツールです。Mingle をマーケットに対して早期リリースすることがとても重要だったので、筆者らは Ruby on Rails で Mingle を書くことに決めました。しかし、いくつかの既存のライブラリ、

例えばSubversionのサポートやJavaベースのチャートライブラリも利用したかったのです。それで、筆者らはRails on JRubyを使ってMingleを実装しました。そして、既存のJavaライブラリの使用と、Javaのデプロイの容易性という利点を得ることを可能にしました。Mingleは多言語プログラミングのアイデアの具体例です。つまり、基礎をなすプラットフォーム堅牢性と豊富さを活用する一方で、手元の仕事に対する最もよいツールを利用しています。

解決策を1つの言語に詰め込もうとしていた日々は、消えさろうとしています。私たちは、JavaとCLRという優れたマネージド・ランタイムを持っているので、よりよいツールを使ってそれらのプラットフォームを利用するべきです。多言語プログラミングは、重要な仕事をする既存のすべてのコードを捨てることなしに、解決策をミックスし適用することを可能にしてくれます。これらの2つの実績あるプラットフォームで、爆発的に新しい言語が開発されています。開発者として、仕事により適したツールを使って、よりよいコードを書くことができるように、この成長を活用する方法を学ぶべきです。

5章
オブジェクト指向エクササイズ

Jeff Bay ◎テクノロジプリンシパル

5.1 ソフトウェア設計を改善する9つのステップ

誰でも、理解やテスト、保守に苦労するようなひどいコードを目にしたことがあるでしょう。オブジェクト指向プログラミングによって、古い手続き型コードからの解放が約束され、再利用してインクリメンタルにソフトウェアを開発することが可能になりました。それにもかかわらず、これまでC言語で行ってきた複雑で結合度の高い設計を、同じようにJavaでも追求しようとしていることがあります。このエッセイは、新米プログラマがコードを書く過程でベストプラクティスを学ぶ、よいきっかけとなるでしょう。また、経験豊かな熟練プログラマは、ベストプラクティスの復習や、同僚に教える際の題材としてこのエッセイを利用できるでしょう。

優れたオブジェクト指向設計は、複雑さの排除に絶大な効果をもたらすものですが、簡単に習得できるものではありません。手続き型の開発からオブジェクト指向設計へ移行するには大きな思考の転換が必要であり、これは思っているよりずっと難しいのです。多くの開発者は優れたオブジェクト指向設計ができていると思い込んでいますが、実際には手続き型の習慣に無意識のうちにとらわれて、なかなか抜け出せていないのです。多くの手本やベストプラクティス（Sun JDKのコードでさえも）が、性能上の理由や単に歴史的な経緯から、質の悪いオブジェクト指向設計を助長しており、状況を悪くしています。

優れた設計を支える中心的な概念はよく知られています。例えば、凝集度（cohesion）、疎結合（loose coupling）、重複なし（zero duplication）、カプセル化（encapsulation）、テスト容易性（testability）、可読性（readability）、フォーカス（focus）の7つのコード品質特性は有名です†。しかし、これらの概念を実践に移すのは簡単ではありません。カプセル化とは「データ、実装、クラス、設計、実体化

の隠蔽」であると理解することと、カプセル化を適切に実現するコードを設計することは、まったく別のことなのです。そこで、優れたオブジェクト指向設計の原理を自分のものにして、実際に使えるようになるためのエクササイズを紹介します。

5.2 エクササイズ

これまで使ってきたものよりずっと厳しいコーディング標準に従って、簡単なプロジェクトを実施してみてください。このエッセイでは、経験に基づく9つのルールを紹介します。これは、ほぼ必然的にオブジェクト指向になるコードを書くように強制するものです。このエクササイズによって、日々の業務で問題に直面したときに、より適切な判断を下せるようになり、さらに選択肢の量と質が向上するでしょう。

まずは騙されたと思って、1,000行程度の小さなプロジェクトで9つのルールを厳密に適用してみてください。そうすれば、ソフトウェア設計に対するまったく異なるアプローチに気がつくでしょう。一度1,000行のコードを書き終えてしまえば、このエクササイズは終了です。今度はリラックスし、ルールを緩めてガイドラインとして使うことができます。

これはハードなエクササイズです。その主な理由は、ルールの多くが普遍的には適用できないからです。実際、クラスが50行を少し超えることもあります[††]。しかし、重複し散在してしまっている責務を移動して、その責務を持つ適切なファーストクラスオブジェクト[‡]へと作り替えるためには何をする必要があるか、を考え

† 訳注：これらのコード品質特性は、著者の経験と、『Code Complete 第2版〈上/下〉』（日経BPソフトプレス）や多くのXP指導者による資料などに基づくものである。また、著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）では、Alan Shallowayが7つの特性を提唱していることが書かれており、著者が彼のアイデアも参考にしていることがわかる。なお、7つの特性のうち「フォーカス」はそれほど知られていないが、これは概念の集中度を表す特性であり、かかわりのある概念が1つのクラスに集まっていれば高く、多くのクラスに散らばっているほど低くなる。「凝集度」はクラスやメソッドの中で扱われる概念の関連の強さを表すので、「フォーカス」は「凝集度」と直交する特性である。例えば、なんらかのかかわりを持つたくさんの概念を1つのクラスで扱うと、「フォーカス」は高くなるが、「凝集度」は低くなる。

†† 訳注：場合によっては、ルールが互いに矛盾することや、ルールを適用できないこともある。そのため、状況に応じて優先すべきルールを選択し、どのようにルールを適用するのかを判断しなければいけない、というのが著者の意図である。

‡ 訳注：プログラミング言語の文脈では「扱いに関してきわめて制限の少ない言語の構成要素」を表す用語として使われるが、本章では「プリミティブ型やコレクションなどをラップする独自のクラスが定義されたオブジェクト」を表す。具体例は、ルール3やルール8の説明で示されている。

ることに大きな価値があります。このエクササイズの本当の価値は、この種の考え方を身につけることなのです。したがって、これ以上できないという思い込みを捨てて、自分のコードについて新しい見方ができるようになっているかを意識するようにしてください。

5.2.1 9つのルール

以下に、エクササイズのルールを示します†。

1. 1つのメソッドにつきインデントは1段階までにすること
2. else句を使用しないこと
3. すべてのプリミティブ型と文字列型をラップすること
4. 1行につきドットは1つまでにすること
5. 名前を省略しないこと
6. すべてのエンティティを小さくすること
7. 1つのクラスにつきインスタンス変数は2つまでにすること
8. ファーストクラスコレクションを使用すること
9. Getter、Setter、プロパティを使用しないこと

5.2.2 ルール1：1つのメソッドにつきインデントは1段階までにすること

昔に書かれた、どこから手をつけてよいのかわからないほど大きいメソッドを見たことはありませんか。巨大なメソッドは凝集度が欠けています。メソッドの長さを5行までに制限するのもガイドラインの1つですが、500行もある化け物のようなメソッドでコードが散らかっていると、そうした変更は大変なものになるでしょう。代わりに、各メソッドが厳密に1つの仕事を行うこと、つまりメソッドごとに制御構造またはコードブロックを1つだけにすることを徹底しましょう。もし1つのメソッド内にネストされた制御構造があれば、複数の抽象レベルを扱っていることになり、それは1つ以上の仕事を行っていることを意味します。

厳密に1つの仕事を行うクラスで、厳密に1つの仕事を行うメソッドを書くようにすれば、コードが変わってきます。アプリケーションにおける処理単位が小さくなれば、再利用性が格段に上がります。責務を5つも持ち、100行に及ぶようなメ

† 訳注：著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）では、「ファクトリメソッド以外のスタティックメソッドは作らないこと」というルールもある。

ソッドを再利用できる機会はまずありません。さまざまな文脈で利用できるのは、ある文脈において1つのオブジェクトの状態を管理する3行のメソッドなのです。統合開発環境の「メソッドの抽出（Extract Method）」†機能を使って、例に示すようにメソッド内のインデントが1段階になるまで振る舞いを抜き出してください。

リファクタリング前††

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                for(int i = 0; i < 10; i++) {
                        for(int j = 0; j < 10; j++)
                                buf.append(data[i][j]);
                        buf.append("\n");
                }
                return buf.toString();
        }
}
```

リファクタリング後

```
class Board {
        ...
        String board() {
                StringBuffer buf = new StringBuffer();
                collectRows(buf);
                return buf.toString();
        }

        void collectRows(StringBuffer buf) {
                for(int i = 0; i < 10; i++)
                        collectRow(buf, i);
        }

        void collectRow(StringBuffer buf, int row) {
                for(int i = 0; i < 10; i++)
                        buf.append(data[row][i]);
                buf.append("\n");
        }
}
```

† 訳注：リファクタリング手法の1つであり、メソッドの一部を抜き出して別のメソッドを作り、それを呼び出すようにすること。

†† 訳注：このサンプルコードは、チェスや碁のようなボードゲームをイメージしたものである。Boardクラスは盤面を表しており、board() メソッドは盤面の状態を出力するメソッドである。

このリファクタリングには、もう1つメリットがあることに注目してください。個々のメソッドは小さくなって、実装がメソッド名と一致しました。たいてい、このような小さなコード片の中からバグを発見するのは非常に簡単です。

最後に、このルールを繰り返し適用すればするほどその強みが得られる、ということを付け加えておきます。初めてこの方法で問題を分解しようとするときは、やりにくい感じがして、なんの得があるのかわからないかもしれません。しかし、ルールを適用するにもスキルが必要で、これは次のレベルに進んだプログラマの技なのです。

5.2.3 ルール2：else句を使用しないこと

プログラマなら誰でもif-else構文を知っています。この構文は、ほとんどのプログラミング言語に組み込まれており、単純な条件ロジックなら誰でも簡単に理解できます。しかし、ほとんどのプログラマは、うんざりするほどネストされた追跡不可能な条件文や、何度もスクロールしなければ読めないcase文を見たことがあるでしょう。さらに悪いことに、既存の条件文に単に分岐を1つ増やすほうが、もっと適切な解決方法を考えるよりも楽なのです。また、条件文は重複の原因になることがよくあります。例えば、ステータスフラグはこの種の問題をよく引き起こします。

リファクタリング前

```
public static void endMe() {
        if (status == DONE) {
                doSomething();

        } else {
                ... その他のコード ...
        }
}
```

このコードをelse句を使わずに書き直す方法がいくつかあります。コードが単純な場合には、次のようにします。

リファクタリング後

```
public static void endMe() {
        if (status == DONE) {
                doSomething();
                return;
        }
        ... その他のコード ...
}
```

リファクタリング前

```
public static Node head() {
        if (isAdvancing()) { return first; }
        else { return last; }
}
```

リファクタリング後

```
public static Node head() {
        return isAdvancing() ? first : last;
}
```

このように、単語をたった1つ追加しただけで4行が1行になります。ただし、メソッドからの早期returnは、使いすぎるとわかりにくくなってしまうことに注意してください。ステータスに基づく分岐を避けるためにポリモフィズムを利用する方法については、GoFデザインパターン[GHJV95]のStrategyパターンを参照してください†。Strategyパターンは、ステータスに基づく分岐が複数箇所で使われている場合に特に有効です。

オブジェクト指向言語には、ポリモフィズムという、複雑な条件分岐を扱うための強力なツールがあります。単純なものであれば、ガード節††と早期returnに置き換えられます。また、ポリモフィズムを用いれば、可読性や保守性がもっと高く、意図をより明確に表す設計にすることができます。しかし、else句を使いこなしていると、この変更は簡単ではありません。そのため、このエクササイズの一環として、else句の使用を禁止します。NullObjectパターン‡も試してみてください。これが役に立つ状況もあるでしょう。同様にelse句を使わずに済ませるツールは他にもあります。選択肢をいくつか考えてみてください。

---

† 訳注：Stateパターンを想起させるような説明だが、実際、GoFのStrategyパターンに条件文を排除する例が掲載されている。

†† 訳注：ある条件を満たしていない場合に直ちにreturnするか例外を投げることで、以降の処理の事前条件を守るif文のこと。「ケント・ベックのSmalltalkベストプラクティス・パターン」（ピアソン・エデュケーション）で、「Guard Clauseパターン」として示されている。また、『リファクタリング』（ピアソン・エデュケーション）で、ネストされた条件記述をガード節に置き換える例が示されている。

‡ 訳注：通常の仕事をするオブジェクトと同じインターフェイスを持ちながら何もしない空オブジェクト（Null Object）を利用するパターン。『プログラムデザインのためのパターン言語』（ソフトバンククリエイティブ）に収録されている。

5.2.4 ルール3：すべてのプリミティブ型と文字列型をラップすること

int型は、それだけではなんの意味も持たない単なるスカラ値にすぎません。メソッドがint型をパラメータとして受け取る場合、メソッド名で意図を表現するしかありません。もし同じメソッドが「時間」オブジェクトをパラメータとして受け取るなら、それが何を指すのかずっとわかりやすくなります。このように小さなオブジェクトを使うことによってプログラムの保守性が高まります。なぜなら、「時間」オブジェクトをパラメータに取るメソッドに「年」オブジェクトを渡すことは不可能だからです。プリミティブ型の変数を使っていると、意味的に正しいプログラムかどうかをコンパイラは教えてくれません。たとえ小さくてもオブジェクトを使うことで、それが何の値でなぜそこで使うのかという情報をコンパイラとプログラマに伝えることができます。

「時間」オブジェクトや「金額」オブジェクトのような小さなオブジェクトを使うと、どこに振る舞いを配置すべきかが明確になります。そのようなオブジェクトがなければ、振る舞いはいろいろなクラスに散らばっていたでしょう。後で紹介するGetterとSetterに関するルール（ルール9）を適用して、小さなオブジェクトを厳密にカプセル化した場合に、これは特に当てはまります。

5.2.5 ルール4：1行につきドットは1つまでにすること

あるアクティビティに対する責務をどのオブジェクトが持つべきなのか、判断が難しいことがあります。複数のドットを使っているコードが何行かあれば、責務の配置を間違っている箇所がたくさん見つかるでしょう。1行の中に複数のドットがある場合、そのアクティビティは間違った場所で実行されようとしています。おそらく、そのオブジェクトは同時に2つの異なるオブジェクトを扱おうとしているのでしょう。この場合、そのオブジェクトは仲介役で、多くのオブジェクトを知りすぎています。そのアクティビティを他のオブジェクトに移すことを検討してください。

ドットがつながっているということは、オブジェクトが他のオブジェクトの中を深く掘り進んでいるということです。つまり複数のドットは、カプセル化に違反していることを示しています。自らオブジェクトの中をいじり回るのではなく、そのオブジェクトにしかるべき仕事をさせるようにしましょう。カプセル化の主な役割は、クラスの境界を越えて知るべきでない型にたどり着かないようにすることです。

デメテルの法則（「直接の友人にだけ話しかけよ」）†に従うことから始めるとよい

† 訳注：すべてのメソッドは、自分自身、パラメータのオブジェクト、自分で生成したオブジェクト、直接の関連を持つオブジェクトのメソッドのみを呼び出すべきであるという原則のこと。

でしょう。ただし、これを次のようにとらえてみてください。「遊んでよいのは、自分のオモチャ、自分で作ったオモチャ、誰かがくれたオモチャのみである。自分のオモチャのオモチャでは決して遊んではいけない。」

リファクタリング前†

```
class Board {
    ...

    class Piece {
        ...
        String representation;
    }
    class Location {
        ...
        Piece current;
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            buf.append(l.current.representation.substring(0, 1));
        return buf.toString();
    }
}
```

リファクタリング後

```
class Board {
    ...

    class Piece {
        ...
        private String representation;
        String character() {
            return representation.substring(0, 1);
        }

        void addTo(StringBuffer buf) {
            buf.append(character());
        }
```

† 訳注：このサンプルコードは、ルール1のサンプルコードと同様にボードゲームをイメージしたものである。Boardクラスは盤面、Pieceクラスはコマ、Locationクラスは盤面の1マスを表している。

```
    }
    class Location {
        ...
        private Piece current;

        void addTo(StringBuffer buf) {
            current.addTo(buf);
        }
    }

    String boardRepresentation() {
        StringBuffer buf = new StringBuffer();
        for(Location l : squares())
            l.addTo(buf);
        return buf.toString();
    }
}
```

この例では、アルゴリズム実装の詳細が拡散していることに注目してください。ひと目で理解するのが少し難しくなったかもしれません。しかし、コマ（Piece）を文字列表現に変換するためのメソッドを作り、character() という名前を付けただけなのです。このメソッドは名前と仕事が強く結びついていて、非常に再利用しやすいものになっています。また、プログラムの他の箇所で representation.substring(0, 1) が繰り返し登場する可能性が格段に下がりました。リファクタリング後のプログラムでは、メソッド名はコメントの代わりになります。名前を付けることに時間を割いてください。このような構造のプログラムを理解するのは、決して難しくありません。ただ、少し異なるアプローチが必要なだけなのです。

5.2.6 ルール5：名前を省略しないこと

クラスやメソッド、変数の名前を省略したくなることがよくあります。その誘惑に打ち勝ってください。省略は紛らわしくなりますし、もっと重大な問題を隠してしまいがちです。

なぜ省略したくなるのか考えてみてください。同じ単語を何度も何度も入力しているせいではありませんか。もしそうなら、そのメソッドは頻繁に使われすぎています。重複を避ける機会を見逃してしまっているのです。もしくは、メソッド名が長くなってきているせいではありませんか。もしそうなら、責務の配置を間違えているか、必要なクラスを抽出できていないのかもしれません。

クラスやメソッドの名前は、1つか2つの単語だけを使うように気をつけ、文脈

が重複する名前は避けてください。Orderというクラスなら、メソッドをshipOrder()とする必要はありません。単にship()と名付けてください。そうすることで、メソッド呼び出しはorder.ship()となります。何が起こるのかはっきりとわかりやすい表現です。

このエクササイズでは、すべてのエンティティの名前には1つか2つの単語だけを使い、省略しないでください。

5.2.7 ルール6：すべてのエンティティを小さくすること

これは、50行を超えるクラス、10ファイルを超えるパッケージは作らないという意味です。

たいてい、50行を超えるクラスは、複数の仕事をしています。それによって、理解や再利用が難しくなってしまいます。50行のクラスはスクロールせずに1画面で見ることができるので、ひと目で理解しやすくなるという利点もあります。

クラスを小さくするのが難しいのは、複数の振る舞いが集まって1つの論理的な意味を表すことがよくあるからです。そこで、パッケージが必要になります。クラスを小さくして責務を減らし、パッケージ内のファイル数も制限すると、パッケージがある目的のために協調して動作するクラスの集まりを表すようになることに、気がつくでしょう。パッケージもクラスと同様に、凝集度の高い、目的を持ったものにすべきです。パッケージを小さくすることで、パッケージが真の存在意義を持てるようになります。

5.2.8 ルール7：1つのクラスにつきインスタンス変数は2つまでにすること

ほとんどのクラスはただ1つの状態変数を扱うことだけに責任を持つべきですが、2つの変数を必要とするクラスも少しはあります。クラスに新しいインスタンス変数を1つ追加すると、途端にそのクラスの凝集度が低下してしまいます。通常、このエクササイズの9つのルールに従ってコーディングしていると、2種類のクラスがあることに気がつくでしょう。一方は、1つのインスタンス変数の状態を管理するクラス、もう一方は、2つの独立した変数を調整するクラスです。原則として、この2種類の責務を混ぜ合わせないでください。

鋭い読者の方は、ルール3と7は同類であることに気づいているかもしれません。一般的に言って、インスタンス変数をたくさん持つクラスが、凝集度の高い1つの仕事をしていることはほとんどないのです。

ここで読者の皆さんに、クラスの解剖実験に取り組んでもらいましょう。

リファクタリング前

```
class Name {
    String first;
    String middle;
    String last;
}
```

このコードは次のように2つのクラスに分解できます。

リファクタリング後

```
class Name {
    Surname family;
    GivenNames given;
}

class Surname {
    String family;
}

class GivenNames {
    List<String> names;
}
```

名前の分解について考える場合、ファミリーネーム（Surname）は多くの法律的な制約に用いられるので、他とは本質的に異なる種類の名前としてその関心事を分離できる、ということに着目してください†。名（GivenNames）オブジェクトは、名前のリストを持っています。これによって、新たなモデルは、ファーストネーム、ミドルネーム、その他の名前を持つ人々にも対応できるようになります。インスタンス変数を分解することで、関連する複数のインスタンス変数の共通点を理解できることがよくあります。また、関連する複数のインスタンス変数が、実際には同一ファーストクラスコレクション††の要素として収められることもあります。

属性をまとめて持つオブジェクトを分解して、協調するオブジェクトの階層構造に作り変えることは、有効なオブジェクトモデルの作成に直結します。筆者は、このルールを見いだす前、大きなオブジェクト全体にわたるデータの流れをたどるの

† 訳注：ファミリーネームは重要な概念であるため、リファクタリング前のコードのように最初（first）や最後（last）という順番で表すべきでないというのが著者の主張である。

†† 訳注：著者は、プログラミング言語で提供されているリストやマップなどのコレクションをプリミティブと見なして、それをラップしたクラスをファーストクラスコレクションと呼んでいる。具体的には、サンプルコードのGivenNamesのようなクラスがファーストクラスコレクションである。

に膨大な時間をかけていました。オブジェクトモデルをどうにか取り出すことはできましたが、関連する振る舞いのグループを理解し、その処理結果を確認するのは骨の折れる作業でした。それに比べて、このルールを繰り返し適用すると、巨大で複雑なオブジェクトから非常にシンプルなモデルへと速やかに分解できるようになりました。振る舞いは、インスタンス変数の後を追って自然と適切な場所に収まります。というのも、コンパイラとカプセル化のルールが、インスタンス変数と異なる場所に振る舞いを置くことを許さないからです。もし行き詰ってしまったら、オブジェクトを二分して関連する2つのオブジェクトを作るようにトップダウンで作業してみてください。もしくは、2つのインスタンス変数を取り上げて、それらからオブジェクトを作るようにボトムアップで作業してみてください。

5.2.9 ルール8：ファーストクラスコレクションを使用すること

このルールを適用するのは簡単です。コレクションを持つクラスには、他のメンバ変数を持たせないようにしてください。各コレクションをそれぞれ独自のクラスにラップすることで、コレクションに関する振る舞いをそのクラスに置くことができるようになります。フィルタ†がこの新たなクラスに含まれるようになるかもしれません。また、フィルタを独立した関数オブジェクトにしてもいいでしょう。さらに、この新しいクラスは、2つのグループの結合††や、グループの各要素に対するルールの適用‡といったアクティビティも扱うことができます。ルール8は、ルール7（インスタンス変数のルール）を少し拡張しただけのものですが、これだけで独立したルールにする価値のある重要なものです。なぜなら、コレクションは非常に便利ではありますが、実際には単なるプリミティブな型の一種にすぎないからです。コレクションは多くの振る舞いを持っていますが、後任のプログラマや保守担当者にプログラム上の意図やヒントをほとんど示せないのです。

† 訳注：コレクションからある条件に該当する要素だけを抽出する機能のこと。
†† 訳注：2つのコレクションをマージして、1つにまとめること。
‡ 訳注：コレクションの要素1つ1つに対して、なんらかのルールを適用すること。

5.2.10 ルール9：Getter、Setter、プロパティ†を使用しないこと

前ルールの最後の文は、このルールにほぼ直結します††。インスタンス変数の適切な集合をカプセル化してもまだ設計がぎこちないときは、もっと直接的なカプセル化の違反がないかをチェックしましょう。振る舞いがその場で簡単に値を求められるようになっていると、その振る舞いはインスタンス変数の後を付いてきません‡。カプセル化によって強固な境界を築く背景には、プログラマはオブジェクトモデルの中から振る舞いを配置すべき唯一の場所を見つけ、そこに振る舞いを配置すべきであるという考え方があります。これは後に、重複の大幅な削減や、新機能を実現するための修正の局所化、といった多くの効果をもたらします。

このルールは「求めるな、命じよ」として一般的に言われています。

5.3 まとめ

9つのルールのうち7つは、単純に、データのカプセル化というオブジェクト指向プログラミングの究極のテーマを明文化し、実現するための方法です。また、もう1つはポリモフィズムの適切な利用（else句を使わずに、条件ロジックを最小にする）を促すものです。さらに、もう1つは命名戦略で、一貫性がなく発音しにくい省略をなくし、明確でわかりやすい命名標準を促すものです。

このエクササイズ全体の目的は、コードレベル、あるいは概念レベルにおいて重複のないコードを作り上げることです。目標とするのは、日々扱っている複雑さを抽象化したシンプルでエレガントな概念を、簡潔に表現するコードです。

エクササイズを進めていく過程で、場合によってはルールが互いに矛盾したり、ルールを適用することでかえって悪い結果になったりすることが必ずあります。しかし、あくまでエクササイズだと思って、20時間かけて、ルールを100%適用した

† 訳注：C#などにおけるプロパティを含め、インスタンス変数を他のクラスに公開するあらゆる仕組みを使うべきでない、というのが著者の意図である。

†† 訳注：ここは原著の誤植である。著者のサイトで公開されている本章の草稿（http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf）から、実際はルール7の後半にある「振る舞いは、インスタンス変数の後を追って自然と適切な場所に収まります。」という文章を指していることがわかる。

‡ 訳注：インスタンス変数の値が他のオブジェクトから簡単に取得できるようになっていると、振る舞いはそのインスタンス変数とは違う場所に配置されてしまうということ。例えば、クラスAにインスタンス変数aがあり、クラスBでaを2倍した値が必要だとする。このとき、Getterなどによってクラスで簡単にaの値を取得できるようになっていると、aを2倍する処理はクラスBに書かれてしまう可能性がある。

1,000行のコードを書いてください。古い習慣から抜け出さなければならないことや、これまでのプログラミング人生でずっと使ってきたルールを変えなければならないことに気がつくでしょう。これまでなら通常は明確な（でも、おそらく正しくない）答えがあったのに、ルールに従うとその答えは使えない、という状況に直面するようなルールを選んであります。

努力してルールを守ることによって、これまでより確かな答えに必ずたどり着くことができます。そして、オブジェクト指向プログラミングをはるかに深く理解できるようになるはずです。ルールに従って1,000行のコードを書くと、出来上がったものが想像していたものとはまったく異なることに気がつくでしょう。ルールを守り、そして、結果を確認してください。それを続けることによって、特に意識しなくてもルールに従ったコードを書けるようになるでしょう。

最後に補足ですが、このエクササイズのルールを極端なもの、もしくは実際のシステムには適用できないものだと見る人もいるかもしれません。しかし、それは間違いです。この本が出版される頃、筆者はこの方法で書かれた100,000行を超えるシステムを完成させる予定です。このシステムを開発しているプログラマたちは、9つのルールに常に従っています。そして、真にシンプルであることを受け入れたときに開発がどれほど楽になるかを知って、とても喜んでいます。

6章 ところでイテレーションマネージャとは何だろうか

Tiffany Lentz◎プロジェクトマネージャ

IT業界が変化し、アジャイル、イテレーティブ、イテレーションといった、いわゆるバズワードがよりありふれたものになるにつれ、これまでにない、定義があいまいなイテレーションマネージャという役割が現れてきています。これは次世代のプロジェクトマネージャでしょうか。素晴らしいチームリーダーのことでしょうか。それとも新たなマネジメント層なのでしょうか。この謎に包まれたマネージャの正体はいったい何者なのでしょうか。

このエッセイでは、ソフトウェアチームの一員としてのイテレーションマネージャの役割と価値を明らかにします。また、イテレーションマネージャの責任の範囲を見るとともに、イテレーションマネージャが組織的、文化的な課題の中で、健全な環境を維持していくという重要な役割をいかにして果たすのかも見ていきます。

6.1 イテレーションマネージャとは何だろうか

一般的に、大規模なアジャイルプロジェクトにおいて、プロジェクトマネージャは、全体計画と特定のチームのイテレーションごとの状況に対して、同時に注意を払えないものです。2000年、ある1つのプロジェクトが、優先度の高い作業がどれであるかを探し出すのに苦労していました。そのとき、解決の糸口になったのが、提供チームに持続可能なペースで優先度の高い機能を継続的に送り込める人を決めることでした。そのときに決めた役割が、現在のイテレーションマネージャ（IM：Iteration Manager）へと成熟してきたのです。

イテレーティブ開発（反復型開発）の世界では、チームをサポートし、顧客との日々の会話をファシリテート†し、チームが最も優先度の高い活動へ集中し続けられるようにする人が必要です。ThoughtWorks社の上級アーキテクトであるFred

Georgeは、イテレーションマネージャとは「チームマネジメントにおける内側の役割††です。イテレーションマネージャはチーム内にスムーズなストーリーのフローを作ること‡に責任があり、これにはチームの適切な割り当てや、必要なスキルの変化にともなうスタッフの変更の助言も含まれています。」と言っています。

6.2 よいイテレーションマネージャとは

IMには、さまざまな異なるベーススキルを持った人がいます。技術力がある人（優れた対人能力も！）、分析力のある人（優れた対人能力も！）、ビジネスの知識が豊富な人（優れた対人能力も！）もいます。しかし、IMは常に前向き思考で、常にできるという態度で、かつ変化を抱擁する能力を持っていなくてはいけません。このようなチーム内ファシリテータは、提供チームのプロセスを最適化するために、日々その能力を活用するのです。

例えば、1回のイテレーションの作業負荷の合意が取れたら、IMはそのイテレーション期間中、チームの進捗を記録し、チームの中で実践的かつ能動的に開発プロセスの改善を行います。日々のスタンドアップミーティングで、IMが開発者から1日で完成すると見積られていたストーリーに3日間を費やしていることを聞いた場面を想像してください。IMはチームメンバの日々の活動とイテレーションの進捗に責任があるため、少なく見積られてしまったストーリーの詳細を徹底的に調査し始めるはずです。もしIMがそのストーリーの実際の状況をすばやく判断せず、イテレーションのスケジュールに変更があることをすぐ顧客に伝えないのであれば、チームはコミットメントを果たせないというリスクを負うことになるでしょう。そのようなときには、IMは次のような質問を自分に問いかけることから始めることができるでしょう。

- 担当の開発者は、そのストーリーのスコープを理解しているか
- 当初の見積りからそのストーリーのタスクが変更されているか、もし変更されているならどのように変わっているか

† 訳注：通常ファシリテーションとは、会議の場などで、発言を促したり、話の流れを整理したりしながら、相互理解を促進させ、会議での合意形成を導くことを指す。このエッセイでは、会議だけにとどまらず、物事を効率よく円滑に進めるという意味合いで使われている。

†† 訳注：プロジェクトマネージャ（PM：Project Manager）は顧客に対して、つまり外側に対して、プロジェクトをスムーズに進めることに責任がある役割。イテレーションマネージャはそれに対して、チームをうまく機能させるという内側に対して責任がある役割。

‡ 訳注：ストーリーのフローをスムーズに提供する（入れる）だけでなく、スムーズに消化させることも含む。

- 担当の開発者が、そのストーリーに対して要求されている最終状態をより深く理解するために、ビジネスアナリストや顧客の支援を必要としているか
- 担当の開発者は技術リーダーからの支援を必要としているか
- そのストーリーを完成させるにあたって、担当の開発者の妨げになっているものはないか(つまりハードウェア、ソフトウェア、インフラに問題はないか)
- 担当の開発者が、別のプロジェクトに割り当てられていないか、もしくはストーリーの完成を妨げるほど多くの雑多なミーティングに参加していないか

これらの問いかけは、IMがいかにしてチームをスケジュールどおり進めるか、いかにして日々の状況を顧客へ伝えるかという一例にすぎません。IMはチームの要求に毎日耳を傾け、それに応えなければいけません。その中でもIMの一番大きな責任は、プロジェクトの内側で、要求された機能を要求された品質で提供できるような、きちんとオイルの注してある機械を開発することなのです。

IMは技術的な素養とビジネスの知識をバランスよく持っているべきです。Mary PoppendieckとTom Poppendieckは著書†の中で、アジャイルなリーダーは、「顧客側の問題と技術的な問題、その両方に対する深い理解によって、彼らは尊敬を得る。」と書いています。コミュニケーションスキルが長けていることは必要不可欠です。IMは、顧客との関係において、またマネジメントとの関係において、チームの代弁者として働きます。

また、IMはチーム内においても、チームメンバの権利をファシリテートし、執行し、守らねばなりません。多くのアジャイルチームでは、これらの権利は開発者の権利章典に基づいています。これらの権利はチーム全体で合意されたもので、チームはそれらの権利が保障されるために、たいていIMの支援を必要とします。

このIMによる支援は、たいていチーム内コミュニケーションやチーム間コミュニケーションをファシリテートするという形をとります。開発者の多くは直接顧客と話をすることに慣れていませんし、顧客からの要求を直接聞き出し、ストーリー化するのにも慣れていません。IMは、たいていメトリクス、図表、グラフの例を使いながら、オープンなコミュニケーションをファシリテートする必要があります。

IMは顧客の権利も守らなくてはいけません。チームメンバが優先順位から外れた仕事に取り掛かる誘惑にかられるときには常に、IMは顧客の代弁者としてチームに介入します。顧客には望むとおりの優先順位で開発を進めてもらう権利があることを忘れていませんか。IMはどんなときでも中立の立場に立たなくてはいけま

† 訳注:邦訳は『リーンソフトウェア開発 — アジャイル開発を実践する22の方法』(日経BP社)。

せん。

6.3 何がイテレーションマネージャでないか

IMはプロジェクトマネージャ（PM）ではありません。IMは、PMとは違い、日々の活動において開発の現場の中でメンバと行動を共にします。もしIMを担当したら、予算編成、リソース管理、コンプライアンス、その他のごたごたはPMに任せ、ただチームに集中するのです。

それに、IMはチームメンバであって、スタッフやリソースの管理者でもないのです。IMはチームメンバの年単位の査定の報告書を書くような責任もありません。これをしてしまうと、IMの中心タスクである、顧客が最優先に考えている機能にチームを集中させながらもチームを守ることができる、という中立な立場を台無しにしてしまうでしょう。チームメンバはIMのご機嫌取りをするのではなく、必要なときに支援を求めるべきなのです。

IMは顧客でもありません。ストーリーの持つ性質や実際の顧客の都合によっては、チーム内のビジネスアナリストやアーキテクトが、顧客として振る舞ってもかまいません。しかし、IMは絶対に顧客として振る舞うべきではありません。もしIMが顧客として判断を下そうとしている場合、IMはチームと一緒に適切な問題解決活動をファシリテートすることはできません。

最後に、IMは通常、技術的な整合性や標準への準拠を保証したり、技術的なインフラ（例えばビルド、デプロイ、データベースなど）のサポートを提供したりもしません。複数のプロジェクトを調整したり、デプロイやロールアウトを調整したりといったプロジェクトの目標達成に向けた活動は、通常であれば技術リーダーやビジネスアナリストのリーダーの責任です。

6.4 イテレーションマネージャとチーム

イテレーションマネージャという役割は、明確には規定されていませんが、さまざまな日々の責任を含んでいます。そのいくつかは下記のとおりです。

- ストーリーに費やした時間の収集
- 提供プロセスにおけるボトルネックの可視化
- 顧客へのチーム状況の報告
- 日々のスタンドアップミーティングで取り上げた課題、不良、障害への対処

- 持続可能なペースを保つための、チームへ回ってくる作業のコントロールと作業の割り当てのマネジメント

個々のストーリーに費やした実際の時間を収集することで、さまざまなメトリクスが生成できます。これらの時間を収集し、他のさまざまなデータと比較することで、IM はチームの生産能力を明確にできるようになります。まず第一に、ストーリーを完成させるのに実際に費やした時間とイテレーション中に完成させたストーリーポイント数†を比較すれば、チームミーティングやその他のミーティングなどに費やされた時間に対する、ストーリーの提供に実際に費やされたチームの時間の比率が把握できます。第二に、ストーリーを完成させるのに実際に費やした時間と、プロジェクトのためにチームが確保した時間を比較すれば、IM はチームの許容作業量と、チームがプロジェクトのためにどれくらい稼動できるかに関する見解を得ることができます。最後に、ストーリーを完成させるのに費やした時間とストーリーの見積りを比較すれば、見積り精度がわかります。これらのメトリクスはすべて、別の環境でも有用ですし、チームが自分たちに合った提供ペース††を見つけるために使われるべきです。

このチームに合った提供ペースは、次回以降のイテレーションに向けてチームの許容作業量を計算するための基盤になります。テストが完全に完了している各イテレーションの成果物を把握し、チームメンバごとの今後の稼働率を把握することで、IM は実測値に基づいた提供に向けての許容作業量を計画できます。許容作業量は、チームへ押し付けるものでもなく、また期日に提供するという必要性にかられて決定されるものでもありません。この許容作業量はチームメンバがチーム自身を統率するために計算されるものです。そのペース（許容作業量）がビジネスのニーズに一致していない場合は、プロジェクトに関する許容作業量以外の要素を調整することで対応します。しかし、その場合でも、実際の生産能力から将来の許容作業量を継続して予測します。

さまざまなメトリクスや、ストーリーカードボード‡のストーリーカードを注意深く配置することによってボトルネックを明らかにすることができます。例えば

† 訳注：ストーリーごとに付けるポイントで、完成までにどのくらいの時間がかかるかを相対的に示す。

†† 訳注：原著では「consistent delivery pace」であるが、チームの持続可能な作業ペースから生み出される提供ペース、すなわちそのチームの許容作業量に見合った提供ペースという意味で、チームに合った提供ペースという訳語を当てた。

‡ 訳注:ストーリーカードを見やすいように貼っておくためのボード。7章で示す「図7-4 ストーリーボード」のようなもの。

1日で終了すると見積られたストーリーが3日経ってもストーリーカードボードの開発中カテゴリにあれば、それはチーム内で議論が必要なボトルネックを効果的に明らかにしています。フィンガーチャートはFred Georgeによって考案されたとても効果的なメトリクスの1つです。このチャートは積み上げ面グラフで表され、各領域は提供ライフサイクルの各フェーズに対応しています。ストーリーのステータスが日々更新されると、グラフの各領域が増加していきます。それによって、ストーリーが提供ライフサイクルの中をどのように進んでいくのかをチームで確認できます。すべての領域が比例して増加していくと、そのグラフは指の形に見えます。しかし、グラフの1つの領域の増加が他の領域と比例していない場合（つまり、開発待ちの領域が開発中の領域より広くなる場合）、ボトルネックが明らかになります。この時点で、チームは提供ペースを再安定させるために、どのようにボトルネックを減らすかを議論できます。

日々のスタンドアップミーティングの間、IMは余計な事柄を取り除き、過去24時間に何をしたか、次の24時間で何をするか、どんな障害があるかという、チームメンバの報告が順調に進むようにします。IMはチームメンバがストーリーカードを完成させるために必要な行動と、取り除くべき障害の報告に耳を傾けます。もし、誰かが報告以外のことで独占的に話し続けていたら、IMはチームを通常の報告に集中するように戻すことができます。これは通常、大きな問題を抱えている人に、ミーティングの後にその問題に取り組むことを促していることになります。

6.5 イテレーションマネージャと顧客

これまでに議論したように、メトリクスはIMがチームの持続可能なペースを把握するのに役に立ちます。持続可能なペースを把握することで、チームは定常的にコミットメントを行い、守り続けることができるのです。しかし、チームメンバがそのコミットメントを守るためには、IMはイテレーションの最中に顧客がストーリーを変更するのを防がなければいけません。IMはチームの窓口として振る舞うことで、顧客が次の仕事に優先度を付けられるようにし、優先度の頻繁な変更によってチームが混乱しないようにします。

IMはチームの窓口として、混乱からチームを守り、また不注意によるチームの生産性の悪化から顧客を守ります。イテレーションの外側では、顧客は優先度を頻繁に変えられますし、また変えるはずです。イテレーションが始まるまでは、意思決定にかかわるすべての要素は変化にさらされており、常に新しい情報が入ってきているのです。

プロジェクトにおけるジャストインタイムでの意思決定の考え方は、新しい考え方ではありません。トヨタ生産方式†を基に考案されたリーン開発では、これまでの長い間、集合ベース開発と呼ばれる技術を採用しており、成功し続けています。この集合ベース開発は「決定しなくてはいけないときまで注意深く決定をしないこと。そして、チームが解決策を決定することを目的として選択肢をすぐ閉じてしまうアプローチに比べて、チームが最も多くの情報を基に、できるかぎり決定を遅くでき、より早く最適な解決策にたどり着けるように、選択肢（の集合）の維持を懸命に行うこと。」と説明されています。

6.6 イテレーションマネージャとイテレーション

イテレーション固有の責任もあります。IMは次のような行動を通して、顧客とチームとともに各イテレーションを計画します。

- 顧客が行う優先度付けを支援する
- チームからの提案をファシリテートする
- 提供に向けたチームの許容作業量を計画する

IMはチームを導き、励まし、動機付けます。IMはチームの健康チェックを通してチームが正直であることを維持します。このチェックはチームがアジャイルという方法論で提案されている手法すべてに忠実であることを保証するためではなく、どの手法がチームによい影響を与えているかを明らかにするために行います。

IMに任されているイテレーション固有の最後の責任は、ミーティングをファシリテートすることです。IMは、イテレーション計画ミーティングやリリース計画ミーティングなどを主催し、運営します。イテレーション計画ミーティングやリリース計画ミーティングの適切なファシリテーションは、チームを成功に導きます。計画ミーティングでは、メトリクスや、うまく行っていること、うまく行っていないことだけでなく、許容作業量に関する計画も、正当かつオープンに議論されなければいけません。

リリース計画ミーティングでは、IMは自身の洞察力の強みを活かしながら、顧客とともに次のリリースで提供する機能の大まかな部分を計画します。その計画が

† 訳注：原語は「Toyota's Knowledge-Based Engineering」となっている。自動車製造を知識主導産業へと作り変えた、トヨタの継続的な生産プロセスのカイゼンの手法（暗黙知を形式知化していく手法）を指すと思われるが、読者の読みやすさを考慮し、その精神の詰まったトヨタ生産方式（Toyota Production System）と意訳した。

合意され、機能に対する変更の予測が立つと、IMはチームに見積りの概要（例えば、ストーリーポイント）を伝え、顧客に次のリリースで提供されるもののアイデアを提示します。

イテレーション計画ミーティングでは、イテレーションマネージャは提供するために必要な作業以上の作業にメンバがサインアップ†しないように守ることもたいていの場合に必要です。また、IMがメトリクスを評価することで、チームメンバがその評価結果を「活用」し、生産能力を改善できるようにしなければいけません。

最後にIMは、チームが「フェイルファスト」††を実践できるように、また次のイテレーションで必要な改善策を可視化できるように、振り返りミーティングをファシリテートします。IMは、そのイテレーションでうまく行っていること、うまく行っていないことに対して議論が行われるように導きます。その議論の最中に、うまく行っていないことの改善に集中させるチームメンバを割り当てる機会もあります。この活動によって、チームメンバそれぞれが成長しようとする責任感の輪を生み出せるのです。

6.7 イテレーションマネージャとプロジェクト

このエッセイで述べてきたように、IMはプロジェクトに関するいくつかの典型的な責任を持っていますが、時としてチーム内の文化的な目的に対して介入することも求められています。IMは顧客を満足させつつ、チームメンバが達成感と満足感を感じ、生産的で、互いに尊重し合えるような環境をファシリテートします。Fred Georgeは、「二次的な目的として、私は、プロジェクトの最後までにチームメンバを成長させられることをイテレーションマネージャに求めています。チームとは神聖な預り物であり、チームメンバのスキルの幅を広げることもイテレーションマネージャの仕事です。」と述べています。

IMはプロフェッショナルで責任感のあるチーム環境を作るように努力すべきです。そのような環境では、次のような適切な態度と習慣が現れます。

- 自分にも他人にも顧客にも、互いに尊重する態度を示す
- 成功を褒め称える

† 訳注：ストーリーカードに名前を書き込み、そのストーリーの実装を自分が担当することを宣言する行為。

†† 訳注：可能なかぎり早く失敗すること。ここでは、問題を早期発見し、早期に対応することを意図している。

- ミスを学習の機会と捉える

イテレーションマネージャは、チームのメンバが互いに成功も失敗も共有するような1つの結束したグループとなるように努力します。

6.8 まとめ

IMの終日の仕事は、きちんとオイルの注してある提供機械を作り、ストーリーカードを継続的に与え、その機械をチューニングすることです。明確なコミュニケーション、現実主義、変化を許容する能力は、身につけ上達させるのが難しいスキルです。2000年以降、ThoughtWorks社では、成功するアジャイルプロジェクトの数を増やし、また多くのプロジェクトを安定して成功させるために、IMを育成し、顧客へ派遣してきました。IMはアジャイルチームに繰り返し可能なプロセスを根づかせ、その責任の範囲内でプロジェクトを成功させ、チームの文化を改善します。この活動が満足感を持った生産性の高いチームメンバをもたらすのです。

開発者が、日々のコミュニケーションを持ち、余計な事柄を取り除き、顧客に最新の情報を提供していては、コードを書くための時間は細切れの時間ばかりになってしまいます。アジャイルチームにイテレーションマネージャがいなければ、チームは失敗してしまうでしょう。チームは目前のタスク（つまりストーリー）にだけ集中し続ける必要があり、余計な事柄はイテレーションマネージャに委ねます。

7章
プロジェクトバイタルサイン

Stelios Pantazopoulos◎イテレーションマネージャ

医療の分野では、医師や看護師が病室に入り、患者のカルテを見て、直近のバイタルサイン†を大まかに把握します。この情報によって、彼らは迅速に患者の健康状態を診断でき、是正措置が必要かどうかを判断できるのです。

ソフトウェア開発プロジェクトにおいても、医師たちと同じように、プロジェクトの直近のバイタルサインを把握できるカルテがあると素晴らしいと思いませんか。

このエッセイは、プロジェクトの直近のバイタルサインを把握し、チームメンバや利害関係者に効率よく伝えるための、シンプルで、実用的で、手間がかからない手法を提案します。チームはこの方法を使って情報を得ることで、プロジェクトの健康状態に関して確かな情報に基づいた見解を持ち、プロジェクトの健康問題の根本原因に対する是正措置を計画することができます。

7.1 プロジェクトバイタルサインとは

プロジェクトバイタルサインは定量的なメトリクス群であり、複数のメトリクスを同時に計測することでプロジェクトの健康状態をタイムリーに把握できます。

このエッセイでは次のプロジェクトバイタルサインを紹介します。

- **スコープバーンアップ**：納期までのスコープ†における提供プロセス††の状態
- **提供品質**：最終的に提供する製品の状態

† 訳注：生体情報のこと。
†† 訳注：提供しようとしている機能群を指す。
‡ 訳注：単に開発だけでなく、リリースから、サービスイン（カットオーバー）までを含む、いわゆる「ラストマイル」フェーズを含む一連の活動を指す言葉。

- **予算バーンダウン**：スコープの提供に関する予算の状態
- **開発実況**：システムの提供におけるリアルタイムな状況
- **チーム知覚**：プロジェクトの状態に関するチームの認識

7.2 プロジェクトバイタルサイン vs. プロジェクトヘルス

プロジェクトバイタルサインはプロジェクトヘルスとは独立したメトリクスで、それらを混同してはいけません。プロジェクトヘルスとはプロジェクト全体の状況についての見解で、プロジェクトバイタルサインを分析した結果です。ですから、プロジェクトヘルスは本質的に、主観的で計測不可能なものです。同じプロジェクトバイタルサインでも、2人のチームメンバはそれぞれプロジェクトヘルスに関する異なる結論に達するかもしれません。例えばマネージャは予算バーンダウンに比重を置くかもしれませんし、品質保証（QA）チームは提供されるソフトウェアの品質に比重を置くかもしれないからです。また、開発者はスコープバーンアップがより重要と考えるかもしれません。プロジェクトヘルスに関する意見はそれぞれのチームメンバの見方に大きく左右されるのです。いずれにせよ、どの見解もプロジェクトヘルスに関係しており、重要で、唯一無二のものです。

チームメンバ全員が、プロジェクトヘルスに関して確かな情報に基づいた見解を持つための最もよい方法は、プロジェクトバイタルサインを収集し公開することです。基準としてのプロジェクトバイタルサインがなくては、プロジェクトヘルスについての見解は推測の域を出るものではないからです。

各チームは独自にプロジェクトヘルスを定義する必要があります。プロジェクトヘルスについて合意するために、チームメンバは、必要な情報を得られるようなプロジェクトバイタルサインのリストも作成しなくてはいけません。いったん必要なプロジェクトバイタルサインを見つけたら、次にそのプロジェクトバイタルサインを周知する情報発信器†を開発する必要があるでしょう。

7.3 プロジェクトバイタルサイン vs. 情報発信器

情報発信器は、Alistair Cockburnによって考案されたもので、「通りがかりの人か

† 訳注：Alistair Cockburnによる『アジャイルソフトウェア開発』（ピアソン・エデュケーション）を参照のこと。

ら見える場所に情報を表示するもの」とされています。この情報発信器はプロジェクトバイタルサインを伝える有用なツールなのです。

プロジェクトバイタルサインにとって、絶対必要な情報発信器というのはありません。このエッセイでは有用だと実証されている、それぞれのプロジェクトバイタルサインに対応する情報発信器を提案します。ただし、ここで提案する情報発信器はそれぞれのプロジェクトバイタルサインを伝える唯一の方法ではないことに注意してください。

7.4 プロジェクトバイタルサイン：スコープバーンアップ

スコープバーンアップは、納期までのスコープにおける提供プロセスの状態を示します。メトリクスは、スコープの規模はどのくらいか、どの程度完成しているか、納期はいつかを示したものです。

7.4.1 スコープバーンアップの情報発信器の例

図 7-1 のスコープバーンアップ図は、システムがどのくらい完成していて、どのくらい完成していないのかを測定し、伝えるものです。

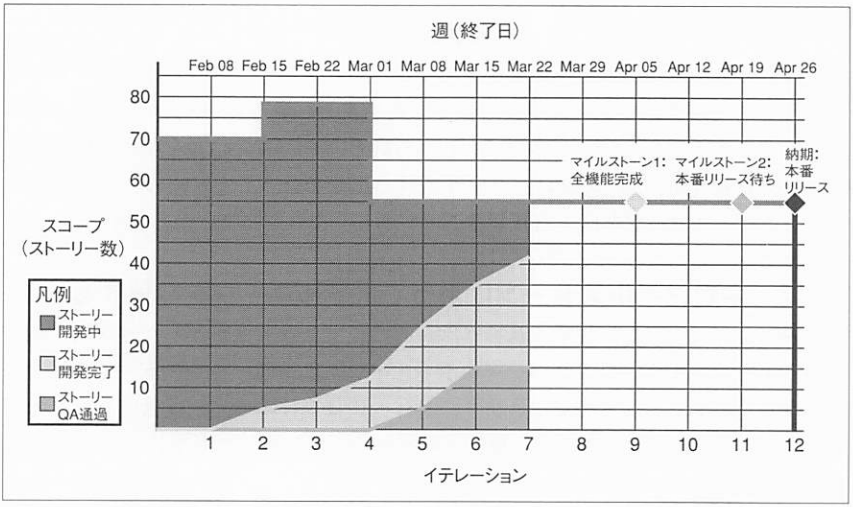

図 7-1 スコープバーンアップ図

この図は次のような情報を伝えています。

- スコープの測定単位(ストーリーの数)
- 各週末でのスコープの規模(3月22日では55ストーリー)
- スコープを無事に提供するために必要な中間マイルストーン(マイルストーン1、マイルストーン2)
- 週単位での中間マイルストーンに向けた進捗状況(3月22日では55ストーリー中15ストーリーが本番リリース待ち)
- スコープの提供の納期(イテレーション12の最後にあたる4月26日)

コミュニケーション、見やすさ、メンテナンスのしやすさを考慮するなら、この図は開発リーダーかイテレーションマネージャが管理し、全チームの近くにあるホワイトボードに貼るべきです。

7.4.2 スコープの測定単位を定義する

効果的にスコープバーンアップを把握するためには、スコープを定めるために、まずチームメンバで測定単位を合意しなくてはいけません。スコープの測定単位はプロジェクトごとに必ず違うものです。理想的には測定単位はプロジェクト進行中に変えるべきではありません。もし途中で変更した際には、それまでのスコープバーンアップデータはほとんどの場合利用できなくなります。

時間や日数は測定単位に使用しないでください。スコープは、どれだけのタスクがあるかを測定するべきで、時間の長さで測定するべきではありません(「どの程度」であるべきで、「どのくらいの期間」ではありません)。時間を測定単位にしていれば、見積り時間vs.実績時間という余計な比較を行うことになり、スコープを測定したり、スコープを伝えたりすることが難しくなってしまうからです。

7.4.3 中間マイルストーンによってボトルネックを発見する

中間マイルストーンに対する進捗率は、その提供プロセスがどれだけ順調に進んでいるかを示します。提供プロセスのボトルネックは中間マイルストーンに対する進捗率を比較することで発見できます。具体的には進捗率の差が提供プロセスのボトルネックを示します。例えば、QAフィードバックループによるボトルネックは、全機能完成マイルストーンに対する進捗率が、本番リリース待ちマイルストーンの進捗率を上回ったときに現れます。

7.4.4 スコープバーンアップ図の詳細説明

このスコープバーンアップ図に示されているプロジェクトにおいて、スコープの測定単位はストーリーです。

この事例のチームではプロジェクトを開始する前に、測定単位としてストーリーを使用することを決めました。

また、このチームではストーリーを次のように定義しました。

- ストーリーは1つ以上のユースケースの全部または一部の実装を示す
- 開発者は2～5稼働日でストーリーの実装と単体テストを完了できる
- QAチームがストーリーの受け入れテストを行い、要求を満たしていることを確認する

この事例では、プロジェクト開始時にスコープを実装するための、2つの中間マイルストーンが設定されていました。マイルストーン1は、全ストーリーのビルドと単体テストの期日であり、QAの実施前の状態です。マイルストーン2は全ストーリーのビルドと単体テストが完了し、QAも通過した状態の期日です。これらのマイルストーンの達成状況はバーンアップ図で直接追跡が可能になっていました。

次にスコープがどのようにマネジメントされたかを簡潔に紹介します。

1. プロジェクト開始時点で、スコープの規模は70ストーリーだった
2. イテレーション2の途中、8ストーリーがスコープに追加され、合計で78ストーリーになった
3. イテレーション4の途中、プロジェクトの全利害関係者が集まってバーンアップ図の傾向分析をした結果、予算内かつ期日どおりに望まれた品質でマイルストーン1、2を達成する見込みはないという見解に達した。後の会議で合意した対応策では、スコープの規模縮小が決定され、結果として23ストーリーを今後のリリースへ延期し、スコープを55ストーリーへ縮小した
4. イテレーション5で、スコープは55ストーリーに縮小した
5. プロジェクトは現在イテレーション8中で、スコープは55ストーリーのままである。チームメンバは、マイルストーン2を期日どおりに達成できるかどうか確信がないが、この時点では是正措置の議論を控えようと決めていた

次のデータはスコープバーンアップ図用に測定した、スコープメトリクスの元データです

イテレーション	スコープ	「開発待ち」あるいは「開発中」	ビルド済(QA待ち)	ビルド済(深刻なバグ有)	ビルド済&QA通過
1	70	70	0	0	0
2	78	73	2	3	0
3	78	71	1	6	0
4	78	66	3	9	0
5	55	25	9	11	10
6	55	20	8	12	15
7	55	13	10	17	15

7.5 プロジェクトバイタルサイン：提供品質

提供品質は、提供される最終製品の状態を表します。このメトリクスは、チームがどのくらいの品質でスコープを提供できるかを示すものです。

7.5.1 提供品質における情報発信器の例

バグカウント図（図7-2）は、システムにおける欠陥数を重要度別に測定し、伝えるものです。

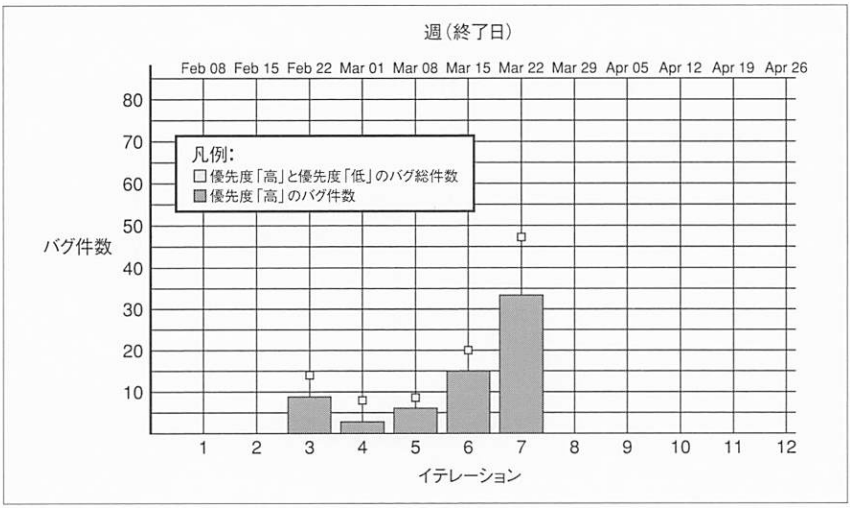

図7-2 バグカウント図

この図は次のような情報を伝えています。

- 未解決のバグの総件数(イテレーション7の最後にあたる3月22日の週では47件)
- リリースまでに修正しなくてはいけないバグの件数（イテレーション7の最後にあたる3月22日の週では、47件中33件が優先度「高」）
- 今後のリリースに修正を延期できるバグの件数（イテレーション7の最後にあたる3月22日の週では、47件中14件が優先度「低」）
- 週ごとのバグ件数の推移（イテレーション1、2の最後では0件、イテレーション3の最後では14件、イテレーション4の最後では8件、イテレーション5の最後では9件、イテレーション6の最後では20件）

コミュニケーション、見やすさ、メンテナンスのしやすさを考慮するなら、この図はQAチームが管理し、全チームの近くにあるホワイトボードに貼るべきです。

7.5.2 バグカウント図の詳細説明

QAチームはバグレポートを登録するとき、重要度「低」、「中」、「高」、「致命的」のいずれかを割り当てます。重要度「致命的」のバグが出た場合はQA作業をストップし、直ちにバグを修正しなくてはいけません。重要度「高」のバグは本番リリースまでに修正されなくてはならず、重要度「中」のものは本番リリースまでに修正することが望ましく、重要度「低」のものは修正できれば素晴らしいが必須というわけではありません。

月曜日の朝に、QAチームが先週の金曜日までに報告されたバグの件数をバグカウント図に反映します。図中の優先度「高」は重要度が「致命的」か「高」で未解決のバグで、優先度「低」は重要度が「中」か「低」で未解決のバグです。

このプロジェクトの例では、バグカウントは最初の2週間は0件になっています。これはQAチームがまだ組織されておらず、誰もストーリーの実装に対する受け入れテストを、それまで実施しなかったからです。

次のデータはバグカウント図用に測定された、バグカウントメトリクスの元データです。

イテレーション	「致命的」バグ数	「高」バグ数	「中」バグ数	「低」バグ数
1	0	0	0	0
2	0	0	0	0
3	0	9	4	1
4	0	3	4	1
5	0	6	2	1
6	0	15	3	2
7	3	30	10	4

7.6 プロジェクトバイタルサイン：予算バーンダウン

予算バーンダウンはそのスコープの提供に関する予算の状況を表したものです。そのメトリクスは、プロジェクトがどれだけの予算を持ち、どのくらいのペースで予算を消化し、あとどのくらいの期間予算を持たす必要があるのかを示したものです。

7.6.1 予算バーンダウンの情報発信器の例

予算バーンダウン図は、プロジェクト予算の消化額、残額、消化のペースを測定し伝えるための情報発信器です。図7-3に例を示します。

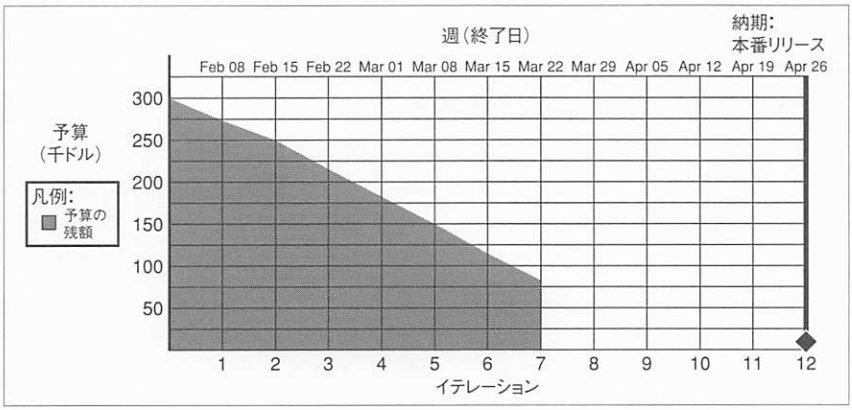

図7-3 予算バーンダウン図

この図は次のような情報を伝えています。

- 予算の測定単位(千ドル)
- 予算総額(プロジェクト開始時は30万ドル)
- 予算の消化額(イテレーション7の最後までで22万ドルの消費)
- 予算の残額(イテレーション7の最後で8万ドル)
- 週ごとの予算消化額(イテレーション1,2では平均2万5千ドル。イテレーション3以降では平均3万3千ドルに増加)
- そのスコープの提供の納期(イテレーション12の最後にあたる4月26日)

コミュニケーション、見やすさ、メンテナンスのしやすさを考慮するなら、この図はプロジェクトマネージャが管理し、全チームの近くにあるホワイトボードに貼るべきです。

7.6.2 予算バーンダウン図の詳細説明

イテレーション1、2の間は、費用の発生するチームメンバが8名いて、各メンバ用のワークステーションと、ビルド兼ソース管理用の共用サーバをリースしていました。週の予算バーンダウンは、費用が発生するチームメンバ、リースしたワークステーションとサーバで2万5千ドルでした。

イテレーション3で、費用の発生するメンバ2名と、2台のリースのワークステーション、1台の受け入れテスト用のリースのサーバがプロジェクトに追加されました。費用が発生するチームメンバ、リースのワークステーションとサーバが増えた結果、週の予算バーンダウンは3万3千ドルに増加しました。

7.7 プロジェクトバイタルサイン：開発実況

開発実況は、システムの提供プロセスのリアルタイムな状況を表します。メトリクスは、スコープ内の各アイテム†の提供に関するリアルタイムな状態を示したものです。

7.7.1 開発実況における情報発信機の例

ストーリーボード（図7-4）とストーリーカード（図7-5）は対象システムの開発実況を測定し、その情報を伝えるための情報発信器です。ストーリーボードやストーリーカードは次のような情報を伝えています。

- スコープに含まれる全アイテム（全55ストーリーが存在している。各ストーリーは灰色のストーリーカードで表されている）
- アイテムが取り得る開発状態（開発待ち、分析、開発中、QA、バグ、完成）
- スコープ内の各アイテムの現在の開発状態（各ストーリーカードは開発状態の下に置かれている）
- 開発状態ごとのアイテム件数（開発待ち4件、分析中5件、開発中4件、QA 10件、バグ17件、完成15件）
- アイテムの開発に現在割り当てられている開発者（ストーリーカード上に貼られている、開発者名が書かれた黄色い付箋紙。ストーリー35にはJohnが割り当てられている）

† 訳注：スコープを構成するもの。このエッセイでは例として、「ストーリー」を扱っている。アイテムの他の例としては、非機能テスト、保守担当者が参照するドキュメント、システム変更履歴に関するドキュメントなどが挙げられる。

コミュニケーション、見やすさ、メンテナンスのしやすさを考慮するなら、この図はアナリスト、開発者、QAチームなどが管理し、全チームの近くにあるホワイトボードに貼るべきです。

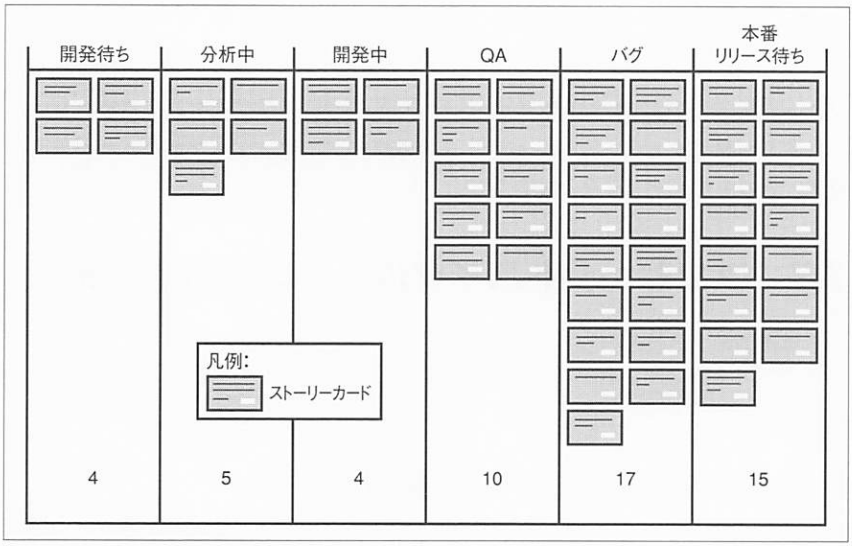

図7-4 ストーリーボード

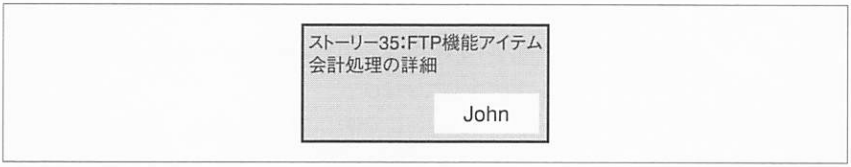

図7-5 ストーリーカード

7.7.2 開発状態の定義

アイテムが取り得る開発状態は各プロジェクトで個別なものです。図7-4の開発状態は必ずしもすべてのプロジェクトに通用するとはかぎりません。すべてのチームメンバは決められた開発状態についてはっきりと理解し、同意するべきです。

プロジェクト中に開発状態の定義を変更してもかまいません。当初から定義していた開発状態がプロジェクト中に不適切と判断され、後になって開発状態を再度定義し直さければいけないことは多くあります。

7.7.3 ストーリーボードとストーリーカードの詳細説明

このストーリーボードはイテレーション 8 の火曜日午後 3 時 14 分のものです。以下は各開発状態の詳細説明です。

開発状態	定義
開発待ち	ストーリーの分析、実装ともに未着手
分析中	ストーリーの分析作業中
開発中	ストーリーの実装、単体テスト中
QA	ストーリーの実装・単体テストが終了し、QAチームのレビュー準備完了
バグ	QAチームがストーリーをレビューし、その開発の問題を発見した
本番リリース待ち	QAチームがストーリーの実装をレビューし、発見した問題が解消された

7.8 プロジェクトバイタルサイン：チーム知覚

チーム知覚は、プロジェクトの状態に関するチームの認識を収集したものです。メトリクスは、プロジェクトの提供プロセスのある側面に対するチームの意見を示すものです。

7.8.1 チーム知覚の情報発信器の例

チームムード図（図 7-6）は、プロジェクトの進捗についてチームメンバがどのように認識しているかを測定し、伝えるためのツールです。

この図は次のような情報を伝えています。

- 毎週のイテレーション振り返りミーティングでチームメンバに行った質問（図 7-6 では「～の確信がありますか？」）

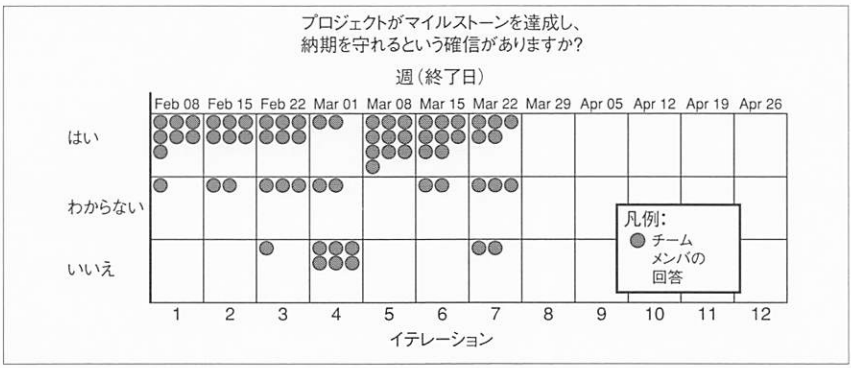

図 7-6 チームムード図

- その質問に対する回答の選択肢（はい、わからない 、いいえ）
- 各チームメンバの質問に対する回答（イテレーション6の振り返りミーティングでは、10人中8人が「はい」を回答）

コミュニケーション、見やすさ、メンテナンスのしやすさを考慮するなら、この図は全チームメンバが管理し、全チームの近くにあるホワイトボードに貼るべきです。他のチームメンバからの影響や威嚇を避けるために、匿名で回答させるべきです。

チームへの質問は1つでなくてもかまいません。

7.8.2 チームムード図の詳細説明

質問に対する各チームメンバの回答を表すために緑色の点を使います。チームメンバは毎週の振り返りミーティングのときに匿名で質問に回答します。

このプロジェクトではプロジェクト開始以降、チームメンバの人数が変化しました。イテレーション1、2では8名のチームメンバがいて、イテレーション3以降では10名のメンバがいました。

8章
コンシューマ駆動契約
——サービス進化パターン

Ian Robinson ◎アーキテクト†

サービス指向アーキテクチャ（SOA：Service-Oriented Architecture）は、組織の俊敏性を向上し、変更の総コストを削減します。高い価値を持つビジネス機能群を別々の再利用可能なサービスとして配置することで、中核的ビジネスプロセスの実現に欠かせないサービス間の接続やオーケストレーションは、より容易に実現できます。サービス間の依存を減らして、変更や不足の事態に対応するためのすばやい再構成とチューニングをできるようにすることで、よりいっそうの変更コストの削減を実現できます。

しかし、企業がこのようなメリットを十分に実感できるのは、SOAによってそれぞれのサービスが互いに独立して進化できる場合にかぎります。サービスの独立を可能にする一般的な方法は、メッセージの型ではなく、契約を共有してサービスを構築することです。サービスプロバイダ（サービスの提供者）は、サービス、送受信するメッセージ、エンドポイント、利用可能なコミュニケーション手段を記述した契約を発行します。これにより、コンシューマ（サービスの利用者）は、サービスが維持する内部ドメイン表現とサービスの基盤となっているプラットフォームと技術のどちらにも制約を受けずに、契約を実装し、サービスを利用することができます。

このエッセイでは、サービス契約、あるいは、そのありがちな実装法と利用法が、どのような場合に過度に結合してしまうのかについて検討します。契約に基づいて

† この章の執筆の準備にあたって協力いただいた、Ian Cartwright、Duncan Cragg、Martin Fowler、Robin Shorrock、Joe Walnesに感謝いたします††。

†† 訳注：本章は、以下に掲載されている記事を基として、本書のために加筆修正されたもの。
http://martinfowler.com/articles/consumerDrivenContracts.html
http://msdn.microsoft.com/en-us/library/bb286659.aspx

サービスを開発すると、たいていはプロバイダと同じペースでコンシューマを進化させなければならなくなります。コンシューマは、内部ロジックの中でドキュメントスキーマ全体を特別な考慮もなく表現してしまうことでプロバイダと結合する傾向があるためです。スキーマ拡張点の追加と、受信メッセージの「ほどほどの」バリデーションは、このような結合の問題を軽減する有名な戦略です。

サービス契約はプロバイダ中心で書かれるため、サービスは互いに密結合する傾向にあります。プロバイダ契約に個々のコンシューマの期待や要望が反映されにくいのは、まさにその性質によるものです。筆者は、サービス提供の目的をコンシューマの要望、すなわちコンシューマの契約に合わせることで、契約におけるこのような性質を補正できると考えています。コンシューマ契約は、サービスが表すビジネス機能に対してコンシューマが持つ、正当な期待を表すものです。

コンシューマとメッセージをやりとりする間、インポートしたコンシューマ契約を守り続けるサービスは、プロバイダ契約から派生した、コンシューマ駆動契約と呼ばれるものを実装します。コンシューマ駆動契約は、後の節で紹介するアサーションベース言語を用いて、プロバイダに自分のコンシューマに対する義務についての洞察を浸透させます。そして、サービスの進化の関心を、コンシューマの要求する主要なビジネス機能の提供に集中させます。

コンシューマ駆動契約パターンの適用対象は、主に、コンシューマを特定でき、支配下に置くことのできるサービスコミュニティ、言い換えると、1つの企業の境界内におけるサービス群です。このパターンには明らかな制約があります。特に、ツールがサポートされない、メッセージ処理パイプラインに影響を及ぼす、複雑さが増大する、サービスコミュニティに導入したプロトコルに依存するといったものが挙げられます。しかし、このパターンが適切な文脈で採用された場合、そのようなデメリットをはるかに上回るメリットを得ることができるはずです。サービス間のコミュニケーションは一見複雑になりますが、組織の俊敏性に必要な、ある種のきめの細かい洞察や迅速なフィードバックが促進されるという点に限っていえば、間違いなくアジャイルなパターンです。ただし、サービス間の結合をゼロにすることはできません。むしろ、ある程度の結合の存在は、望ましいと言えます。コンシューマ駆動契約パターンは、そのような結合を明らかにして、定量化し、分析の対象とするのに役立ちます。さらにこのパターンは、開発、デプロイ、運用といったシステムのライフサイクルの各フェーズ間の橋渡しを行い、軽量なバージョニング戦略の構築を可能にし、サービスの進化のコストとその影響を予測可能にします。そして、システム所有の総コスト（TCO：Total Cost of Ownership）についても十分に考慮し、達成すべき義務にも貢献します。

8.1 サービスの進化：例

サービスが進化する際に直面する問題の例として、コンシューマのアプリケーションが製品カタログを検索するための、シンプルな製品（Product）サービスについて考えます。

例えば、検索結果として得られるXMLドキュメントは、以下のようになります。

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns="urn:example.com:productsearch:products">
  <Product>
        <CatalogueID>101</CatalogueID>
        <Name>Widget</Name>
        <Price>10.99</Price>
        <Manufacturer>Company A</Manufacturer>
        <InStock>Yes</InStock>
  </Product>
  <Product>
        <CatalogueID>300</CatalogueID>
        <Name>Fooble</Name>
        <Price>2.00</Price>
        <Manufacturer>Company B</Manufacturer>
        <InStock>No</InStock>
  </Product>
</Products>
```

このProductサービスは、現在のところ、2つのアプリケーションから利用されています。1つは自社内におけるマーケティング用アプリケーションで、もう1つは、社外の小売業者のWebアプリケーションです。いずれも、受け取ったXMLドキュメントを処理する前にXSDバリデーション†を行っています。社内のアプリケーションでは、カタログ番号（CatalogueID）、製品名（Name）、価格（Price）、製造者（Manufacturer）の各フィールドを利用しています。それに対して、社外のアプリケーションでは、CatalogueID、Name、Priceの各フィールドを利用しています。在庫あり（InStock）フィールドは、どちらのアプリケーションにも利用されていません。InStockフィールドはマーケティング用アプリケーションでの利用を想定して実装されたフィールドでしたが、開発ライフサイクルの早い段階で、不要になりました。

サービスの進化が起こる最も一般的な理由の1つは、単一または複数のコンシュー

† 訳注：W3C XML Schema（http://www.w3.org/XML/Schema）.

マのための、ドキュメントへのフィールドの追加です。しかし、プロバイダとコンシューマの実装方法によっては、そのようなシンプルな変更であっても、その企業およびパートナーに、コスト面で多大な影響を及ぼすことになるでしょう。

この例では、稼働開始後しばらくして、第2の小売業者がProductサービスの利用を検討していることがわかりました。この小売業者は、各Productメッセージに備考（Description）フィールドを追加するよう依頼しています。この変更は、コンシューマの設計方法が原因となって、プロバイダだけでなく既存のコンシューマにも深刻な影響を及ぼし、そのコストは、各コンシューマの変更をどう実装するかによって、大きく変動します。サービスコミュニティのメンバの間で変更のコストを負担する範囲を広げる可能性のある方法が、少なくとも2つ存在します。1つは、検索結果のメッセージに正しくバリデーションを適用するために、オリジナルのスキーマを修正し、各コンシューマの持つスキーマのコピーの更新を要求する方法です。システムの変更コストを負担する範囲は、このような変更要求に直面するプロバイダ（常にこの手の変更を行う必要があるでしょう）と、更新された機能にはまったく興味のない既存のコンシューマに及びます。もう1つは、新しいコンシューマ用に第2の操作とスキーマを公開しながら、既存のコンシューマのために、オリジナルの操作とスキーマを維持するという方法です。この場合、変更コストは当面の間プロバイダだけに限定されますが、サービスはより複雑になり、将来の保守コストを増加させるという負担をともないます。

このような単純な例からでも、一度サービスプロバイダとコンシューマが稼働を開始すると、プロバイダは即座に、コンシューマに提供している契約のどのような要素の変更に対しても、慎重なアプローチしか採用できなくなることが明らかになります。このような事象が発生する理由は、コンシューマが契約をどう実現するかを、プロバイダが予測し、洞察することができないことにあります。SOAで実装する契約の機能と役割に対する深い理解がなければ、サービスは必然的に、誰も体系的に存在を指摘することのできない「隠れた」結合の形式を受け入れます。サービスコミュニティによる契約の採用方法をプログラムで認識できる手段がないことと、プロバイダとコンシューマによって選択される契約駆動の実装に課すべき制約が不足していることが重なり、SOAによって企業が手にできるメリットが損なわれます。要するに、企業にとって、サービスの利用が負担となります。

8.2 スキーマのバージョニング

まず、Productサービスを破たんさせてしまう契約と結合の問題について、スキー

マのバージョニングという観点から調査を始めてみましょう。W3Cのテクニカルアーキテクチャ部会（TAG：The W3C Technical Architecture Group）は、今までに数多くのバージョニング戦略†を発表しており、それらは、結合の問題を軽減し、サービスにおけるメッセージスキーマを進化させる手助けとなり得ます。

ここでの戦略は、極端に自由放任的なバージョニングなしから、極端に保守的なビッグバンまでを範囲としています。バージョニングなし戦略の場合、サービスは、スキーマのバージョンの違いを区別してはならず、すべての変更を許容しなければなりません。ビッグバン戦略の場合、想定されたバージョン以外のメッセージを受信した場合、サービスの異常終了を要求します。

この2つの極端な戦略はいずれ、ビジネス価値の提供を妨げ、システムにおけるTCOを悪化させるという問題につながります。バージョニングなし戦略を採用するシステムは、それを明言するかどうかにかかわらず、インタタラクションが予測不可能となり、壊れやすく、下流における変更のためのコストが増大する性質を持ったものになります。

それに対して、ビッグバン戦略は、プロバイダとコンシューマを通じてスキーマの変更が波紋のように広がり、稼働中のサービスを中断させ、進化を妨げ、収益を生む機会を減少させるといった、密結合のサービスランドスケープ††で観察できる弊害の原因となります。

例におけるサービスコミュニティでは、事実上、ビッグバン戦略を採用しています。もし、システムにおけるビジネス価値の向上とともにコストも上昇するようであれば、前方互換スキーマと後方互換スキーマの利用による、より柔軟なバージョニング戦略（TAG研究報告では互換性戦略と呼ばれています）の導入は、明らかにプロバイダとコンシューマの双方にメリットとなります。サービスの進化という文脈では、後方互換スキーマは、コンシューマが新しいスキーマを用いて、より古いスキーマから生成されたインスタンスを受け入れることを可能にするものです。後方互換リクエストの新しいバージョンを扱えるように作られたサービスプロバイダは、古いスキーマでフォーマットされたリクエストを、従来どおり受け入れることができます。それとは反対に、前方互換スキーマは、コンシューマが古いスキーマを用いて、より新しいスキーマから生成されたインスタンスの処理を可能にするものです。ただし、例では、Productサービスの既存のコンシューマが障害となって、

† TAG研究報告（勧告案）："*Versioning XML Languages [editorial draft]*"（http://www.w3.org/2001/tag/doc/versioning), November 16, 2003.

†† 訳注：プロバイダとコンシューマの相互作用の全体像を上から眺めている様子を表すために、著者が用いている言葉。

これらの導入は困難です。もし、稼働開始時に、検索結果のスキーマが前方互換に作られていたら、コンシューマは、処理の失敗や変更要求もなく、新しいバージョンのスキーマのインスタンスを適切に扱えていたことでしょう。

8.2.1 拡張点

後方互換かつ前方互換であるスキーマの作成は、よく知られた設計作業の1つです。そしてそれを最適に表現したものが、拡張性における Must Ignore パターン† です。Must Ignore パターンでは、スキーマに拡張点を含めることを推奨しています。拡張点とは、型への拡張要素の追加と、各要素への属性の追加を可能にするものです。また、このパターンは、コンシューマが拡張部分を処理する方法を指定するための、XML 言語による処理モデルの定義も推奨しています。最もシンプルな処理モデルは、認識不可能な要素を無視するようにコンシューマに要求することから、Must Ignore パターンという名前が付いています。同様に、このモデルは「Must Understand」フラグを持つ要素を必ず処理するよう、そして、そのフラグを理解できない場合は異常終了するよう、コンシューマに要求します。

以下は、検索結果の XML ドキュメントに基づいて作成したスキーマの最初のバージョンです。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                      name="Product" type="Product" />
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
```

† David Orchard: "*Extensibility, XML Vocabularies, and XML Schema*" (http://www.pacificspirit.com/Authoring/Compatibility/ExtendingAndVersioningXMLLanguages.html).
Dare Obasanjo: "*Designing Extensible, Versionable XML Formats*" (http://msdn.microsoft.com/en-us/library/ms950793.aspx).

```
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Price" type="xs:double" />
        <xs:element name="Manufacturer" type="xs:string" />
        <xs:element name="InStock" type="xs:string" />
      </xs:sequence>
  </xs:complexType>
</xs:schema>
```

時間を遡って、スキーマに前方互換性に着目した修正を加えてみましょう。拡張性を考慮したスキーマは、以下のようになります。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="urn:example.com:productsearch:products"
        id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
                    name="Product" type="Product" />
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Product">
      <xs:sequence>
        <xs:element name="CatalogueID" type="xs:int" />
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Price" type="xs:double" />
        <xs:element name="Manufacturer" type="xs:string" />
        <xs:element name="InStock" type="xs:string" />
        <xs:element minOccurs="0" maxOccurs="1"
                    name="Extension" type="Extension" />
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Extension">
      <xs:sequence>
        <xs:any minOccurs="1" maxOccurs="unbounded"
                namespace="##targetNamespace" processContents="lax" />
      </xs:sequence>
  </xs:complexType>
</xs:schema>
```

このスキーマは、各 Product 要素の末尾に、任意の Extension 要素を含んでいま

す。このExtension要素は、対象名前空間に属する要素を1つ以上保持することができます。

これで、各製品に備考を追加したいという要求があっても、プロバイダは、拡張コンテナに挿入されたDescription要素を含む新しいスキーマを発行することで、対応できるようになりました。これにより、Productサービスは、製品に対する備考を含めた検索結果を返すことができ、新しいスキーマを使うコンシューマは、XMLドキュメント全体にバリデーションを適用することができます。

古いスキーマを利用しているコンシューマは、備考を処理しないにもかかわらず、適切に動作することでしょう。Productサービスの新しい検索結果は、以下のようなXMLドキュメントになります。

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns="urn:example.com:productsearch:products">
  <Product>
        <CatalogueID>101</CatalogueID>
        <Name>Widget</Name>
        <Price>10.99</Price>
        <Manufacturer>Company A</Manufacturer>
        <InStock>Yes</InStock>
        <Extension>
          <Description>Our top of the range widget</Description>
        </Extension>
  </Product>
  <Product>
        <CatalogueID>300</CatalogueID>
        <Name>Fooble</Name>
        <Price>2.00</Price>
        <Manufacturer>Company B</Manufacturer>
        <InStock>No</InStock>
        <Extension>
          <Description>Our bargain fooble</Description>
        </Extension>
  </Product>
</Products>
```

改訂されたスキーマは、以下のようになります。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
```

```
        id="Products">
    <xs:element name="Products" type="Products" />
    <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                      name="Product" type="Product" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
          <xs:element name="Name" type="xs:string" />
          <xs:element name="Price" type="xs:double" />
          <xs:element name="Manufacturer" type="xs:string" />
          <xs:element name="InStock" type="xs:string" />
          <xs:element minOccurs="0" maxOccurs="1"
                      name="Extension" type="Extension" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Extension">
        <xs:sequence>
          <xs:any minOccurs="1" maxOccurs="unbounded"
                  namespace="##targetNamespace"
                  processContents="lax" />
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Description" type="xs:string" />
</xs:schema>
```

この拡張スキーマの最初のバージョンは、2番目のバージョンと前方互換であり、2番目のバージョンは、最初のバージョンと後方互換であることに注意してください。ただし、この柔軟さは、複雑さの増大という代償をともないます。拡張スキーマにより、XML言語に予測不可能な変更が持ち込まれてしまう可能性もあります。さらに言うと、拡張スキーマが想定している要求は、まず発生する可能性のないものかもしれません。このようなアプローチを選択することで、シンプルな設計に由来する表現力を不明瞭にしてしまい、ドメイン言語へのメタ情報用のコンテナ要素の導入により、ビジネスの情報として意味のある表現を台無しにしてしまいます。

ここで、スキーマの拡張性についてこれ以上検討するつもりはありません。拡張点により、サービスのプロバイダとコンシューマを破たんさせることなく、ドキュメントとスキーマに対して後方互換と前方互換な変更を行えることを、十分に説明できたと思います。しかしながら、明らかに互換性に影響する変更が契約に必要と

なる場合、スキーマ拡張は役に立ちません。

8.3 互換性に影響する変更

付加価値を追加するため、Productサービスは、検索結果に製品の在庫状況を表すフィールドを含めています。このサービスは、そのフィールドを、呼び出しコストの高いレガシー在庫システムを用いて、メッセージ内に埋め込んでいます。この在庫システムに対する依存関係を維持するために、高額な保守費用を負担しています。サービスプロバイダは、この依存関係を断ち切って、設計を見直し、システムの総合的な性能を改善したいと思っています。そして、できれば、コンシューマに変更のコストを負担させたくないとも考えています。幸運なことに、実際には、どのコンシューマのアプリケーションも、この値を利用していませんでした。高コストにもかかわらず、余計なものでした。

これは良いニュースでしたが、悪いニュースもあります。現在の設定では、拡張スキーマが要求しているコンポーネント（このケースでは`InStock`フィールド）をスキーマから削除してしまうと、既存のコンシューマが動作しなくなります。プロバイダを修正するためにはシステム全体を修正する必要があります。プロバイダから機能を削除して新しい契約を発行する場合、各コンシューマのアプリケーションは、新しいスキーマを用いて再デプロイされる必要があります。また、サービス間のインタラクションについて、十分なテストを行う必要もあります。このような点で、Productサービスは、コンシューマから独立して進化することができません。すべてのプロバイダとコンシューマは、いっせいにジャンプしなければなりません。

この例におけるサービスコミュニティは、サービスの進化について不満を持っています。各コンシューマが、プロバイダの契約全体を特別な考慮もなくコンシューマの内部ロジックに反映することで「隠れた」結合形式を実装しているからです。コンシューマは、XSDバリデーションと、ドキュメントスキーマから派生させた静的言語との束縛を通じて、コンポーネントの一部を処理したいという欲求とは無関係に、プロバイダ契約全体を暗黙のうちに受け入れています。

David Orchard氏は、この問題を回避する手段として、「一般論として、実装は、送信の振る舞いについては厳格であり、受信の振る舞いについては寛容であるべきだ」と、インターネットプロトコルの堅牢性原則[†]を暗に示すことで、いくつかの手がかりを提示しています。

サービスの進化という文脈では、この原則を、メッセージの受信者は「ほどほどの」バリデーションを実装せよ、と言い換え、補強することができます。すなわち、

メッセージの受信者は、自分が実装しているビジネス機能に貢献するデータだけをバリデーションの対象とするべきです。そして、受信したデータに対しては、暗黙的で適用範囲を定めない場合に行う、XSDの処理としてもともと備わっている「全か無か」のバリデーションではなく、個別に対象を指定できる、または、範囲を限定することができるバリデーションを適用するべきです。

8.3.1 Schematron

コンシューマ側のバリデーションを改善する1つの方法として、おそらくSchematron[††]のようなツリー構造パターン検証言語を用いて、受信したメッセージのXMLドキュメントのツリー構造の軸[‡]に沿うパターン表現式によるアサーション[‡‡]を行うものがあります。Schematronを利用すると、Productサービスの各コンシューマは、検索結果に含まれていると期待されるものに対して、プログラムによるアサーションを行うことができます。

```
<?xml version="1.0" encoding="utf-8" ?>
<schema xmlns="http://www.ascc.net/xml/schematron">

  <title>ProductSearch</title>
  <ns uri="urn:example.com:productsearch:products" prefix="p"/>

  <pattern name="Validate search results">
       <rule context="*//p:Product">
         <assert test="p:CatalogueID">Must contain
                                        CatalogueID node</assert>
         <assert test="p:Name">Must contain Name node</assert>
         <assert test="p:Price">Must contain Price node</assert>
       </rule>
  </pattern>

</schema>
```

† 訳注：「受信するものについては寛容であれ、送信するものについては保守的であれ」(be conservative in what you do, be liberal in what you accept from others)
David Orchardの言及しているインターネットプロトコルの堅牢性原則（Robustness Principle）は、一般的にはポステルの法則（Postel's Law）として知られている。ポステルの法則は、Jon Postelが、1981年にRFC 793（TCP）のsection-2.10中に記したもの（http://tools.ietf.org/html/rfc793）。

†† *"Schematron：A Language for Making Assertions About Patterns Found in XML Documents"*（http://www.schematron.com）.
訳注：JIS X 4177-3 (ISO/IEC 19757-3) 規則に基づく妥当性検証 -- Schematron

‡ 訳注：軸は、XPathなどで用いられる表現で、コンテキストノードとの相対関係を表す。

‡‡ 訳注：アサーションは、要素の存在や値に対する期待の表明を行うこと。

Schematronの実装は、通常、このようなSchematronスキーマをXSLTスタイルシートに変換します。メッセージ受信者は、妥当性を判断するために、このスタイルシートをXMLドキュメントに適用します。

このSchematronスキーマのサンプルでは、コンシューマのアプリケーションが必要だと考えていない要素に対して、アサーションを行っていないことに注意してください。この方法では、検証言語はある範囲の必須の要素の集合をバリデーションの対象とします。対象とするドキュメントのスキーマに行った変更が、もともとは必須であった要素を除去する拡張であったとしても、Schematronスキーマに記述された明示的な期待が裏切られないかぎり、バリデーションのプロセスで検出されることはありません。

これが、契約と結合の問題に対する比較的軽量な解決策です。これなら、XMLドキュメントへの不明瞭なメタ情報用要素の追加要求もあがりません。再度時間を遡って、この章の始めのほうで紹介したシンプルなスキーマに戻してみましょう。ただし今回は、コンシューマは、受信の振る舞いについては寛容であれ、と主張したいと思います。これは、コンシューマは、自分の実装したビジネス機能に必要な情報だけに対して、(XSDではなく、Schematronスキーマを利用して)受信メッセージのバリデーションと処理を行うべきだ、という意味です。プロバイダに対して、各製品に備考を追加したいという依頼があった場合、サービスは、既存のコンシューマを阻害せずに、修正したスキーマを発行できます。同様に、どのコンシューマもInStockフィールドをバリデーションも処理もしていないことがわかった場合に、サービスは各コンシューマの進化の速度に影響を及ぼすことなく、検索結果のスキーマを改訂することができます。

最終的に、Productサービスの検索結果のスキーマは、以下のようになります。

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="urn:example.com:productsearch:products"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          targetNamespace="urn:example.com:productsearch:products"
          id="Products">
  <xs:element name="Products" type="Products" />
  <xs:complexType name="Products">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
                      name="Product" type="Product" />
        </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="Product">
        <xs:sequence>
          <xs:element name="CatalogueID" type="xs:int" />
          <xs:element name="Name" type="xs:string" />
          <xs:element name="Price" type="xs:double" />
          <xs:element name="Manufacturer" type="xs:string" />
          <xs:element name="Description" type="xs:string" />
        </xs:sequence>
  </xs:complexType>
</xs:schema>
```

8.4 コンシューマ駆動契約

前の節における Schematron の利用は、プロバイダ - コンシューマ間の契約について、XML ドキュメントのバリデーションの範囲を超える意味を持つ、いくつかの興味深い観察結果を提供しています。この節では、その得られた見識のいくつかを一般化して、コンシューマ駆動契約パターンという言葉で表現します。

まず最初に、ドキュメントスキーマは、サービスプロバイダがコンシューマに提供しなければならない情報の、ほんの一部にすぎないということに注意してください。本章では、サービスにおいて外部に向けて宣伝されるものの総和を、プロバイダ契約と呼ぶことにします。

8.4.1 プロバイダ契約

プロバイダ契約とは、サービスプロバイダが提供するビジネス機能のケイパビリティを、その機能をサポートする上で必須となる、外部に公開可能な要素の集合を用いて表現したものです。サービスの進化という観点では、契約は、外部に公開可能なビジネス機能を要素とする集合のためのコンテナです。その要素の不完全なリストには、以下のようなものが含まれます。

ドキュメントスキーマ

ドキュメントスキーマについては、すでに詳細にわたって検討しました。プロバイダ契約においては、サービスが進化する際、インターフェイスの次に変更が発生しやすい部分です。しかしながら、その変更の多さゆえに、拡張点や XML ドキュメントのツリー構造パスを利用したアサーションなど、サービス進化戦略の浸透を最も多く経験できる項目かもしれません。

インターフェイス

最もシンプルな形式だと、サービスプロバイダのインターフェイスは、コンシューマが自分のメリットとなるようにプロバイダの振る舞いを起動させることのできる、外部に公開可能な操作とシグニチャの集合で構成されます。メッセージ指向システムでは、通常、比較的シンプルな操作とシグニチャが公開され、交換されるメッセージの中にビジネス知識が詰め込まれます。メッセージ指向システムでは、受信されたメッセージは、メッセージヘッダまたはペイロードに埋め込まれたセマンティクスを利用して、エンドポントの振る舞いを起動させます。それに対して、RPCのようなサービスでは、メッセージ指向インターフェイスよりも多くのビジネスに関するセマンティクスが、操作のシグニチャに埋め込まれます。いずれにせよ、ビジネス価値の実現のためには、コンシューマはプロバイダのインターフェイスに少なからず依存することとなり、それゆえに、サービスランドスケープが進化する際には、インターフェイスの利用に対して責任を負わなければなりません。

対話

サービスプロバイダとコンシューマは、1つ以上のメッセージ交換パターンにより構成される対話の中で、メッセージのやりとりを行います。対話の過程で、コンシューマは、プロバイダが送受信するメッセージが、インタラクションに関するいくつかの状態を外部化することを期待します。例えば、対話の開始時に部屋の予約を行い、続いてメッセージで予約の確認と決済を行う、ホテルの予約サービスがあるとします。ここで、もしこのやりとりがこの後も繰り返し行われるのであれば、予約プロセスの各ステップで対話全体を繰り返すよりも、サービスが予約の詳細を「覚えておく」ことをコンシューマが期待するのはもっともなことです。サービスが進化するにつれ、プロバイダとコンシューマが利用できる対話のきっかけは、変化することでしょう。このように、対話は、プロバイダ契約の一部として考慮されるべき候補の1つです。

ポリシー

ドキュメントスキーマ、インターフェイス、対話といったものの公開とは別に、サービスプロバイダは、契約の残りの要素の使い方を管理するための特別な利用要求を定義し、それを強制するかもしれません。最も一般的なものは、コンシューマがプロバイダの機能を利用する際のセキュリティとトランザクションコンテキストに関する要求です。Webサービススタックは、通常、WS-Policy[†]汎用モデルに、WS-SecurityPolicy[††]のようなドメイン固有のポリシー記述言語を追加したものを利用するポリシーフレームワークにより、そういった要求を

表現します。しかし、ポリシーをプロバイダ契約に含める候補として考えるという文脈では、その定義は、仕様または実装に依存しないものであるべきです。

サービス品質特性

サービスプロバイダとコンシューマが開拓するビジネス価値の潜在的な可能性は、可用性、遅延時間、スループットといった、サービス品質の特性という文脈において評価されることがあります。このような特性も、プロバイダ契約の構成要素として考慮されるべきであり、サービス進化戦略に基づいて責任を負うべき項目です。

ここでいう契約は、普段サービスについて議論をしたときに耳にする契約と比較すると、若干広い意味のものですが、問題領域に影響を及ぼす重大な制約を、サービスの進化という観点から、使いやすいように要約したものとなっています。プロバイダ契約に含まれる可能性のある要素の種類について、この定義は完全ではありません。単純に、サービス進化戦略に含まれる候補となる外部に公開可能なビジネス機能群の、論理的な集合にすぎません。論理的には、この候補となる要素の集合は、変更に対して開かれており、自由に要素を追加することができます。しかし実際には、相互運用性に関する要求やプラットフォームの制約のような内部または外部の要因により、契約に含めることのできる要素は制約を受けます。例えば、WS-Basic Profile ‡に準拠しているサービスの契約には、おそらく、ポリシー記述用の要素は含まれていないでしょう。

このような制約にもかかわらず、契約のスコープは、単に、メンバとなる要素の凝集性により決定されます。ビジネス機能のケイパビリティを表現していさえすれば、契約は、広範なスコープを持ち、多くの要素を含めることができ、また、狭いスコープで2個または3個程度の要素だけに焦点を合わせることもできます。候補となる要素をプロバイダ契約に含めるかどうかは、どのように判断すればよいのでしょうか。要素におけるビジネス機能のケイパビリティが、サービス稼働期間にずっと保証され続けることを期待するかどうか、コンシューマに質問してみるとよいでしょう。コンシューマが、サービスによって公開されたドキュメントスキーマの中のある部分だけに関心を示す方法と、契約の要素に関するコンシューマの期待

† WS-Policy (http://www-128.ibm.com/developerworks/library/specification/ws-polfram).

†† Web Services Security Policy Language (http://www-128.ibm.com/developerworks/webservices/library/specification/ws-secpol).

‡ 訳注：WS-I (The Web Services Interoperability Organization) が制定した、Webサービスの相互運用性のためのプロファイル：(http://www.ws-i.org/)

を満足させ続けるためのアサーションの方法は、すでに例で見てきました。このように、ドキュメントスキーマは、プロバイダ契約の一部だと言えます。

プロバイダ契約は、以下のような特性を持ちます。

変更に対して閉じており、完全

プロバイダ契約は、サービスにおけるビジネス機能のケイパビリティを、コンシューマが利用可能かつ、外部に公開可能な要素の完全な集合を用いて表現したものです。システムが利用可能な機能については、プロバイダ契約は閉じており、完全です。

唯一かつ正式

プロバイダ契約は、システムが利用可能なビジネス機能をプロバイダが表現したものであるという点で、常に1つだけしか存在せず、正式なものです。

限定された安定性と不変性

プロバイダ契約は、期間、場所、またはその両方を境界として、その範囲内で、安定かつ不変なものです†。プロバイダ契約は、通常、バージョンを利用して境界の異なる契約のインスタンスを識別します。

8.4.2 コンシューマ契約

もし、公開したスキーマに関するコンシューマの期待に対して責任を負うことを決意したら（そして、その期待がプロバイダにとって知る価値のあるものだと考えられるなら）、プロバイダは、コンシューマの期待をインポートする必要があります。前述の例における Schematron によるアサーションは、プロバイダがクライアントに約束した条件を満たし続けることを保証するのに役立つ、プロバイダに与えられたある種のテストのように見えます。このようなテストを実装することにより、プロバイダは、サービスコミュニティを破たんさせることなく、メッセージ構造を進化させる方法について、より理解を深めることができます。コンシューマから提案された変更が破たんしていたとしても、プロバイダはすぐにその問題に対する洞察を得るでしょう。そして、彼らの要求に対応したり、ビジネス要因に応じてコンシューマに変更を促すインセンティブを提供しながら、当事者間で関心を持ち合ってその問題を解決することができます。

---

† Pat Helland の記事（http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp）の「Validity of Data in Bounded Space and Time」と、「Data on the Outside vs. Data on the Inside：An Examination of the Impact of Service Oriented Architectures on Data」を参照してください。

前述の例では、すべてのコンシューマにより生成された一式のアサーションは、コンシューマの親アプリケーションのためのアサーションが妥当となる期間にやりとりされるメッセージが満たすべき構造を表現しています。もし、プロバイダがこの一式のアサーションを得られるならば、プロバイダの送信するすべてのメッセージは、すべてのコンシューマにとって妥当であることを保証できるようになります(ただし、そのアサーション一式が妥当かつ完全である場合にかぎりますが)。

この構造を一般化することで、プロバイダとコンシューマの関係ごとに固有の、個々の契約上の義務と、プロバイダ契約を区別することができます。筆者は、その契約上の義務のことをコンシューマ契約と呼ぶことにしています。プロバイダが、コンシューマが表現した正当な期待を受け入れ、それを採用した場合、コンシューマ契約が有効となったと言えます。

コンシューマ契約は、以下のような特性を持っています。

変更に対して開かれており、不完全

コンシューマ契約は、システムが利用可能なビジネス機能について、開かれていて、不完全です。コンシューマ契約は、プロバイダ契約に対するコンシューマの期待という側面から見た、システムのビジネス機能が持つケイパビリティのサブセットを表現しています。

複数かつ非正式

コンシューマ契約は、サービスのコンシューマの数に比例して、単数、または複数存在します。さらに、契約上の義務の合計がプロバイダに存在しているという点で、個々のコンシューマ契約を取り出しても、それだけでは正式なものになりません。コンシューマからプロバイダへと拡張される関係の非正式な性質は、SOAと他の分散アプリケーションアーキテクチャを区別する、主要な特徴の1つです。サービスコンシューマは、サービスコミュニティ内の他のコンシューマが、自分とはまったく異なった方法でプロバイダを利用している可能性を考慮する必要があります。他のコンシューマは、進化の速度が異なっているかもしれませんし、システム内の他の構成要素が持つ期待や依存関係を破たんさせる可能性があるような、プロバイダの変更を要求するかもしれません。いつ、どのようにして、他のコンシューマがプロバイダ契約を破たんさせてしまうのかということを、コンシューマは予測できません。SOAではない分散アプリケーションのクライアントには、このような懸念はありません。

限定された安定性と不変性

コンシューマ契約は、プロバイダ契約と同様に、特定の期間、特定の場所、またはその両方において有効となります。

8.4.3 コンシューマ駆動契約

コンシューマ契約は、プロバイダが稼働しているどの時点においても開拓され続けているビジネス価値に関して、再考を促します。プロバイダ契約への期待を表現し、主張することによって、そのプロバイダ契約のどの部分がシステムにより実現されるビジネス価値を現在サポートしているのか、あるいはしていないのかを、コンシューマ契約は効果的に定義します。この点から筆者は、コンシューマ契約に関して、サービスコミュニティは最初に定められたコンシューマ契約からメリットを得ることができるはずだと考えるようになりました。この観点においては、プロバイダはコンシューマの期待と要望を満たすように契約します。契約の新たな取り決めが持つ派生的な性質を受けて、このようなプロバイダ契約をコンシューマ駆動契約、または派生契約と呼ぶことにします。

このコンシューマ駆動のプロバイダ契約が持つ派生的な性質は、サービスプロバイダとコンシューマとの関係に他律的な側面を追加します。すなわち、プロバイダは、自分の境界の外に起源がある責任も負うことになります。とはいえ、その実装における基本的な自律性に影響することはありません。単純に、サービスの成功は利用されることによって決まる、という事実が明確になります。コンシューマ駆動契約は、以下のような特性を持ちます。

変更に対して閉じており、完全

コンシューマ駆動契約は、既存のコンシューマに要求される機能の完全な集合であるという点で、閉じていて、完全です。この契約は、コンシューマの親アプリケーションにとって期待が有効な期間において、彼らの期待をサポートするために必要な外部に公開可能な要素の必須の集合を表しています。

唯一かつ非正式

コンシューマ駆動契約は、システムが利用可能なビジネス機能の表現という点では、常に1つだけ存在するものですが、既存のコンシューマの期待の和集合からの派生であるため、非正式なものです。

限定された安定性と不変性

コンシューマ駆動契約は、コンシューマ契約の特殊な集合であるという点で、安定かつ不変なものです。つまり、特定のコンシューマ契約の集合と比較して、

時間と空間における契約の前方互換および後方互換な性質が事実上限定されるので、コンシューマ駆動契約の有効性を理解できるでしょう。契約の互換性は、特定のコンシューマの契約と期待の集合にとっては安定かつ不変となります。ただし、期待は移り変わるものであるため、変更の発生は避けられません。

8.4.4 契約の特性の要約

次の表は、この章で説明した3種類の契約の特性を要約したものです。

契約の種類	開放性	完全性	存在する数	正式版	境界
プロバイダ契約	閉じている	完全	唯一	存在する	時間/空間
コンシューマ契約	開かれている	不完全	複数	存在しない	時間/空間
コンシューマ駆動契約	閉じている	完全	唯一	存在しない	コンシューマ

8.4.5 実装

コンシューマ駆動契約パターンは、コンシューマ契約とコンシューマ駆動契約の両方を用いるサービスコミュニティの構築を推奨しています。ただし、このパターンは、コンシューマ契約とコンシューマ駆動契約がともに適用すべき書式、構造を明示しません。また、コンシューマの期待をプロバイダに伝える方法と、プロバイダの稼働中にその期待を主張する方法も、規定されていません。

契約は、いくつかの手段で構造化され、表現されます。最もシンプルな形式だと、コンシューマの期待は、スプレッドシート、または似たようなドキュメントで記録され、プロバイダのアプリケーションにおける設計、開発、テストの各フェーズで実装されます。少し進んで、それぞれの期待に対するアサーションを行うユニットテストが導入されると、契約の記述と強制は、繰り返されるビルドごとに、自動的に保証されるようになります。さらに洗練された実装では、期待は、Schematronのような、あるいはWS-Policyのようなアサーション言語によって表現され、サービスのエンドポイントの入出力パイプラインにおいて、実行時に評価されるようになります。

契約の構造化と同様に、プロバイダとコンシューマの間で期待を伝達する手段にも、いくつかの選択肢があります。コンシューマ駆動契約パターンは実装非依存であるため、組織が適切に設計されているのであれば、口頭または電子メールにより、他のチームに期待を伝えることができるはずです。期待の数、コンシューマ数、またはその両方があまりにも大きくなりすぎて、メールまたは口頭による管理が難しくなった場合は、契約サービスのインターフェイスとその実装を、サービスのインフラストラクチャの一部に導入することを検討してください。それがどんなメカニ

ズムであろうと、コミュニケーションは帯域外†で行われ、システムのビジネス機能を実行するどんな対話よりも優先されるはずです。

8.4.6 メリット

サービスの進化に際して、コンシューマ駆動契約は、2つの大きなメリットを提供します。第1に、この契約は、主要なビジネス価値要因に基づいて、サービス機能の仕様とその提供に焦点を合わせます。サービスは、どれだけ利用されたかによってのみ、その価値が決まります。コンシューマ駆動契約は、外部に公開可能なサービスコミュニティの要素の値、言い換えると、プロバイダにコンシューマの業務を行わせるために必要なものに対してアサーションを行うことで、サービスの進化とビジネス価値を結びつけます。結果として、プロバイダは、コンシューマの土台を支えるビジネスのゴールと明確に結びついた、スリムな契約を公開できます。コンシューマがプロバイダに対する期待を修正することにより、サービスの進化は、ビジネスニーズが明確に表現される場所で起こります。

もちろん、最小限の要求の集合からスタートし、コンシューマの期待どおりサービスを進化させることができるということは、管理可能かつ効率的な方法でサービスの進化と運用を行える地位にある人を前提としています。そのような立場にある人は、サービスが利用される現場周辺の期待を把握することができないため、プロバイダ契約は、変更の影響を監視し評価するための、別のメカニズムによって補完される必要があります。それに対してコンシューマ契約は、プロバイダに、ナレッジの蓄積と、システムのライフサイクルにおける運用フェーズで引き出すことのできる、フィードバックのメカニズムを浸透させます。コンシューマ駆動契約に由来するきめの細かい洞察と迅速なフィードバックの利用により、変更の計画と、稼働中のアプリケーションに対する影響の評価が可能となります。実際に、これは個々のコンシューマにとってインセンティブとしてはたらき、結果として、後方互換性、前方互換性、またはそのいずれもない現状での変更は一切認めないとする、彼らの期待を放棄させることができます。

8.4.7 コンシューマ駆動契約とSLA

これまで、コンシューマとコンシューマ駆動契約がビジネス価値を表現する方法について検討してきました。しかしながら、WS-Agreement や Web Service Level

† 訳注：帯域外（out of band）とは、通常の伝送用チャネルとは別に、独立した伝送用チャネルが存在していること。

Agreement（WSLA）のように、表現方法が似ているように見える仕様も存在しています†。このような仕様があるにもかかわらず、今までコンシューマ駆動契約でサービスレベル合意（SLA：Service Level Agreement）について触れていない点について、ここで明らかにしようと思います。WS-Agreement と WSLA は、サービス品質（QoS：Quality of Service）とリソースの可用性に関して、コンシューマに対する保険を提供するべきだという動機に基づいています。WSLA の場合はさらに、サービスの提供と動的なリソースの割り当てについても考慮されています。コンシューマ駆動契約パターンは、サービスそれ自体にはビジネス価値がない、ということを議論の前提としています。サービスの価値は、利用によって決まります。コンシューマが実際に利用する方法でサービスの仕様を決定することによって、そして、サービスの進化を制御可能な方法を用いてビジネス価値を開拓することによって、組織の俊敏性を提供することを目的としています。

そうは言っても、WS-Agreement と WSLA はともに、自動化された契約のプロトコルとそのインフラストラクチャがどのようなものであるかを示す例として役に立ちます。どちらの仕様も、合意条件の表現と監視のためのいくつかのアサーション言語で構成される、合意のテンプレートについて言及しています。合意は、サービス固有の表現に依存していない Web サービスのインターフェイスを通じて確立され、サービスのパイプラインにハンドラを注入することで監視されます。

8.4.8 課題

これまで、サービスランドスケープにコンシューマ駆動契約を導入する動機を識別し、それがどのようにサービスの進化の決定に関する強制力となるかについて述べてきました。このパターンが適用可能なスコープについて、コンシューマ契約、コンシューマ駆動契約の実装時に発生する問題とあわせて検討し、このエッセイを終わりたいと思います。

コンシューマ駆動契約パターンは、単一の企業内、もしくは、よく知られたサービスで構成される閉じたコミュニティ、より具体的には、コンシューマがプロバイダとの契約を定める方法に関して、プロバイダがある程度の影響力を行使できる環境に適用できます。コミュニケーションのためのメカニズムや、期待と責任を表現するためのメカニズムがどれほど軽量であっても、プロバイダとコンシューマは、一連のチャネルと規約について知り、受け入れ、適応しなければなりません。これ

† 訳注：WS-Agreement (http://www.ogf.org/documents/GFD.107.pdf)
Web Service Level Agreements (WSLA) Project (http://www.research.ibm.com/wsla/)

は、すでに複雑なサービスのインフラストラクチャに、さらに1つ複雑な層とプロトコルへの依存関係を追加しなければならないということです。契約を記述し、実装し、運用するためのツールと実行環境のサポートがないことが、問題を悪化させています。

これまでに、コンシューマ駆動契約を考慮して構築されたシステムは、契約の互換性に影響する変更をより管理しやすくするということを示してきました。しかし、このパターンが、そのような変更の問題に対する万能薬であると考えているわけではありません。今までに述べてきたことがすべて行われたとしても、互換性に影響する変更は発生します。しかし、筆者は、このパターンの導入により、実際にその変更が発生した場所で、多くの洞察を得ることができると強く信じています。単純化すると、互換性に影響する変更とは、実在するコンシューマの期待を満足させることのできないすべてのものが該当します。その識別を手助けすることによって、このパターンは、サービスのバージョニング戦略の基礎としての役割を果たしてくれるでしょう。さらに、前述のとおり、このパターンを実装するサービスコミュニティは、サービスの進化の影響をより予測しやすくなり、システムの健康状態に影響が及ぶ前に、潜在的な互換性に影響する変更を特定することができます。とりわけ、開発チームと運用チームは、互換性のために特定の期間だけ契約に盛り込まれる要素の廃止と、新バージョンの契約への移行に反対しているコンシューマを対象としたインセンティブの提供によって、より効果的に、進化に関する戦略を立案できるようになります。

コンシューマ駆動契約を用いても、サービス間の依存が必ず削減されるとはかぎりません。スキーマ拡張と「ほどほどの」バリデーションは、コンシューマとプロバイダの間の結合を緩めるのに役立ちますが、疎結合されたサービスにもある程度の結合が残ります。サービス間の依存の緩和には直接貢献しなかったとしても、コンシューマ駆動契約は、プロバイダ-コンシューマ間のよりよい交渉と管理が可能となるように、いくつかの残存する「隠れた」結合を発見し、明らかにします。

8.4.9 まとめ

SOAによる組織の俊敏性の向上や変更コストの削減は、各サービスが進化を独立して行える場合にかぎり、実現可能です。サービス間の過度の結合は、コンシューマが特別な考慮もなくプロバイダ契約を実装してしまった結果です。スキーマ拡張点のMust Ignoreパターンと、Schematronのアサーションを用いて実装された「ほどほどの」スキーマのバリデーション戦略は、コンシューマとプロバイダの間の結合を緩め、コンシューマにメリットとなります。反対にプロバイダは、コンシュー

マとプロバイダのコミュニケーションの結果であるコンシューマ契約の集合から派生した契約を用いて、実行時におけるコンシューマの責任に対する洞察とよりいっそうのフィードバックを得ます。コンシューマ駆動契約は、サービスのライフサイクルにおける運用フェーズでのサービスの進化をサポートし、主要なビジネスゴールと、サービスの機能における仕様と提供を、より密接に提携させます。

9章
ドメインアノテーション

Erik Doernenburg ◎テクノロジプリンシパル

9.1 ドメイン駆動設計とアノテーション

ここ10年以上もの間、ソフトウェア開発プロジェクトにかかわった多くの人々は、アプリケーションにおける本当の複雑さがソフトウェアの扱う問題領域の中に存在することに気がつき始めています。そのため、ドメイン駆動設計†として知られているアプローチは、以下の2つの前提に従っています。

- ほとんどのソフトウェアプロジェクトでは、ドメインとドメインロジックに最も注力すべきである
- 複雑なドメインの設計はモデルをベースにすべきである

つまり、ドメイン駆動設計においては、ドメインのオブジェクト指向モデルがソフトウェアシステムの中核となります。これが、ドメイン駆動設計がドメインという言葉を使っている理由です。多くの場合、データはリレーショナルデータベースに格納されます。しかし、データは主にドメインオブジェクトの観点から捉えられるべきものであり、テーブルやストアドプロシージャとして捉えられるべきものではありません。中核となるドメインロジックはドメインモデルの中にあり、アプリケーションのユーザインターフェイスやサービスレイヤの中に散らばることはありません。

ドメイン駆動設計に従えば、ドメインモデルとそれ以外のアプリケーションイン

† 訳注：ドメイン駆動設計（Domain Driven Design）は、Eric Evansが"*Domain-Driven Design: Tackling Complexity in the Heart of Software*"（Addison-Wesley）で提唱しているドメイン中心の設計思想。

ターフェイス、インフラストラクチャコードをきれいに分離したソフトウェアシステムを作ることができます。ドメインモデルは通常、長い寿命を持ち、比較的、安定しています。一方、アプリケーションインターフェイスとインフラストラクチャコードは寿命が短く、オブジェクトリレーショナルマッパやWebフレームワークのような特定のテクノロジに結びついています。難しいのは、ドメインモデルとインフラストラクチャコードの両方を再利用できるように、この分離を維持し続けることです。1つのドメインモデルをさまざまなアプリケーションやサービスで使うことが可能でなければなりませんし、新しいテクノロジスタックへのアップグレードも可能でなければなりません。その一方で、インフラストラクチャコードはさまざまなドメインモデルにおいて、簡単に使えるものでなければなりません。最終的なゴールは、もちろん、インフラストラクチャコードの商用製品化、あるいはオープンソース化です。これにより、アプリケーション開発者が問題領域に専念することができます。

9.1.1 ドメイン固有メタデータ

ドメインモデルに基づくアプリケーションは、できるだけ簡単に、かつ自動化を最大限に活用して開発すべきです。また、インフラストラクチャコード、あるいはドメインモデルのコードの変更が必要であってはなりません。そのためには、リフレクションとジェネリック型を多用する必要があります。しかし、その一方でインフラストラクチャコードはドメインモデルの持つメタ情報も有効に活用します。

メタデータは、モデルの実装の一部としてもたくさん存在しています。例えば、部門クラスと社員クラスの関連を表現するために使われている型から、部門と社員の関連が1対多であるということが推測できます。部門クラスが社員を保持するためにコレクションを使っている場合、この関連は対多であることがわかります。そして、社員クラスが通常のオブジェクトの参照として部門を保持していれば、この関連は1対多であることが明らかです。この情報を元にして、ユーザインターフェイスのフレームワークは、部門の画面を自動生成するときに、社員を選ぶためのリストボックスのようなウィジェットを選択することができます。

暗黙的なメタデータ、つまり実装の一部として表現されるメタデータは自動化に大きく貢献します。しかしながら、多くのアプリケーションにとっては、明示的なメタデータのほうが役に立ちます。特にバリデーションにおいてこれは顕著です。先ほどの例で説明しましょう。メンテナンス画面を自動生成するフレームワークは、社員の存在しない部門が正しい状態であるかどうかを知りません。しかし、もし、社員を保持するためのコレクションに、この関連が1対多であることを表すメタ

データを追加すれば、社員の存在しない部門をユーザが保存することを、フレームワークで防ぐことができます。1対多であることを表現するために、特別なコレクションクラスを使うこともできます。しかし、最近の開発プラットフォームは、このような目的のためのより優れた仕組みを提供しています。

9.1.2 Javaのアノテーションと.NETの属性

抽象化を行い、結合度を弱めるメタデータの力は、プログラミング言語の設計者の心を捉えて離しませんでした。Microsoftの.NETプラットフォームと共通言語ランタイム（CLR：Common Language Runtime）は、開発者が任意のメタデータを作成し、ほとんどすべての言語要素に付加することのできる仕組みを、その最初のリリースから提供しています。CLRではこの概念を属性（Attribute）と呼んでいます。属性は他の型と同じように定義でき、クラスのようにデータや振る舞いを持つことができます。しかし、属性を他の言語要素に付加する場合には、角括弧（[]）を使う特別な文法を使用します。

以下の例†は、C#における属性の使い方を示しています。この属性は、プロパティ値の最大長を指定し、それをバリデーションするためのものです。

```
[AttributeUsage(AttributeTargets.Property)]
public class MaxLengthAttribute : Attribute
{
    private int maxLength;

    public MaxLengthAttribute(int maxLength)
    {
        this.maxLength = maxLength;
    }

    public void validate(PropertyInfo property, object obj)
    {
        MethodInfo method  = property.GetGetMethod(false);
        string propertyValue = (string)method.Invoke(obj, new object[0]);
        if(propertyValue.Length > maxLength)
        throw new ValidationException( ... );
    }
}
```

† 訳注：C#には、属性を定義する場合、名前の末尾をAttributeで終わらせる習慣がある。ただし、この属性を使うときにはAttributeを省略することができる。この例では、属性をMaxLengthAttributeという名前で定義し、MaxLengthという名前で使っている。

ここではAttributeの継承についての説明は省略します。CLRによって定義されているAttributeUsage属性を属性の定義に付加することによって、この属性がクラスやフィールドではなくプロパティに対してのみ使うことができることを指定しています。実はこの実装はプロパティの型が文字列型であることを仮定していますが、この制約を属性の定義の一部として表現することはできません。validateメソッドの中で行っているように、疎結合を実現するためにリフレクションを使うことは属性においては一般的です。注意してほしいのは、validateメソッドはアプリケーションコードによって実行する必要があるということです。CLRは対象のプロパティがアクセスされたときにコードを実行する仕組みを提供していないのです。

属性は以下のように使います。

```
[MaxLength(50)]
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Javaにも同じような仕組みがJava 5で追加されました。Javaでは、この概念をアノテーションと呼んでいます。アノテーションのほうがこの概念をうまく表現していますし、ソフトウェア開発の文脈において他の意味と誤解されにくいでしょう。したがって、これ以降、この概念の実装すべてを指して、アノテーションという言葉を使うことにします。

Javaにおいても、言語要素にアノテーションを付加するための特別な文法、@文字があります。しかしながら、.NETとは大きく異なるところもあります。Javaでアノテーションを定義するためには、継承の代わりに新しいキーワード、@interfaceを使います。@interfaceは、属性の使用箇所については指定しません†。代わりに、開発サイクルのどの段階までアノテーションを保持するのかを指定しま

† 訳注：実際には、@Targetでアノテーションの使用箇所を、また@Retentionでアノテーションが保持される段階を指定することができる。例えば、MaxLengthアノテーションをJavaで記述すると以下のようになる。

```
Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface MaxLength {
    int value();
}
```

す。最も重要な違いは、Javaのアノテーションはコードを含むことができないということです。つまり、クラスというよりはインターフェイスに近いといえるでしょう。したがって、.NETではデフォルト値をコンストラクタで定義していましたが、Javaのアノテーションは特別なdefaultキーワードを使って定義します。

もっと重要な違いは、Javaではバリデーションの振る舞いを別のクラスに書く必要があることです。これ自体はひどい仕様というわけではありませんが、カプセル化の原則には違反しています。.NETでは、アノテーションは最大長の値をプライベートに保持し、バリデーションのロジックも含んでいました。Javaでは、値はパブリックに保持して、別のクラスのバリデーションメソッドにアノテーションを渡します。コードの不吉な匂い†によれば、これは「属性、操作の横恋慕」††です。そして、この結果、パラレル継承‡が発生してしまいます。ただし、1つ付け加えておくと、通常、このようにアノテーションに振る舞いを含ませることはありません。したがって、ほとんどのケースでは、Javaのアノテーションと.NETの属性は同じように使われます。

9.1.3 ドメインアノテーションとは

ドメインに関する情報を表現するアノテーションがドメインアノテーションです。ドメインアノテーションは3つの明確な特徴を持っています。

- ドメインアノテーションは、ドメインオブジェクトの言語要素にのみ付加される。つまり、クラスやドメイン固有のパブリックメソッド、あるいはそのようなメソッドの引数などのみに付加することができる
- ドメインアノテーションは、ドメインオブジェクトと同じパッケージや名前空間のドメインモデルの中に定義される
- ドメインアノテーションは、アプリケーションの複数の領域で使用することのできる情報を提供する

† 訳注：不吉な匂いは、Martin FowlerとKent Beckが『リファクタリング』（ピアソン・エデュケーション）の中で紹介しているリファクタリングをするかどうかのガイドライン。

†† 訳注：属性、操作の横恋慕はコードの不吉な匂いの1つ。あるクラスのメソッドが、自分のクラスより他のクラスの属性や操作に関心を持っている状態。

‡ 訳注：パラレル継承もコードの不吉な匂いの1つ。新しいサブクラスを定義すると、別の継承ツリーにもサブクラスを追加しないといけない状態。

最初の項目は本当に決定的な特徴だといえます。ドメインモデルではないオブジェクトに付加されたアノテーションは明らかにドメインの範囲外です。ドメインオブジェクトに特定のメソッドを実装することを要求する環境も存在します。実装する必要のあるメソッドには、同値判定やハッシュコードのような単純なものから、シリアル化のような複雑なものまで存在します。このようなメソッドにドメインアノテーションを使う意味はまったくありません。同様に、プライベートメソッドは明らかにオブジェクトの内部実装ですから、これらに関する情報を提供する必要はないはずです。

もちろん、ルールには例外が存在します。前の節の1対多の例では、メタデータがさまざまなドメインにおいて有用であることを説明しました。EJB 3の仕様が汎用的な@OneToManyアノテーションを提供しているのはこのためです。しかしながら、このようなアノテーションをドメインモデルではなくインフラストラクチャフレームワークに定義することは、2番目のルールと矛盾しています。そして、このアノテーションを使うと、ドメインモデルとEJB 3の仕様は結合してしまいます。さらに問題なのは、他のインフラストラクチャコードがアノテーションの情報を必要とする場合は、それらのコードからEJB 3の仕様への依存が発生することです。ドメインモデルのインフラストラクチャコードに対する依存をなくすことと、汎用的なインフラストラクチャが特別なアノテーションの使用を要求することは明らかに矛盾しています。いつものごとく、いつどちらを使うべきかという絶対的な回答は存在しません。

9.1.4 ドメインアノテーションの実例

先に述べたようにアノテーションはメタデータを提供します。そして、CLRの属性の場合は振る舞いも提供します。しかし、これらは、決してアノテーションが付加された要素の実行パスの一部には成りえません。つまり、アノテーションを使用するためには別のコードが必要だということです。追加のコードが必要であるにもかかわらず、アノテーションを使うと、コード全体の量は少なく、実装はきれいになります。例えば、最大長のバリデーションを行う別の方法を考えてみてください。規約や明示的なインターフェイスを使うことによって、バリデーションメソッドをドメインオブジェクトに追加することができます。この場合、以下のようなJavaコードが必要になります。

```
public void validate()
{
    if(getName().length > 50)
```

```
            throw new ValidationException("Name must be 50 characters or less");
        if(getAddressLineOne().length > 60)
            throw new ValidationException(
                            "Address line one must be 60 characters or less");
        /* 他のオブジェクト状態に対するバリデーションは省略 */
    }
```

このようなメソッドにはいくつかの関心が混在しているので、大きな問題があることは一目瞭然です。例えば、後でバリデーション時に複数のエラーを報告しなければならなくなったことを想像してみてください。このような問題を解決するために、多くの開発者は、以下のようにバリデーションエラーを報告するメソッドを抽出します。

```
    public void validate()
    {
        if(getName().length > 50)
            validationError("Name", 50);
        if(getAddressLineOne().length > 60)
            validationError("AddressLineOne", 60);
        /* 他のオブジェクトの状態に対するバリデーションは省略 */
    }
```

抽象化のレベルをさらに1段階あげるために、値を取得し、チェックを行うメソッドを作ります。このメソッドの実装にはリフレクションを使います。すると、validate メソッドに残るコードは以下のようになります。

```
    public void validate()
    {
        validate("name", 50);
        validate("addressLineOne", 60);
    }
```

この例の場合、抽象化可能なすべてのコードをチェックを行うメソッドの中に移動したので、validate メソッドに残ったのはメタデータのリストだけです。このような情報は明らかにアノテーションとして保持すべきものです。

```
    @MaxLength(50)
    public String getName()
    {
        /* 実装は省略 */
    }
```

アノテーションを使うことにより†、getNameメソッドの近くに名前の最大長に関する情報を置くことができます。そして、より重要なのは、メソッド名を表す文字列の使用を避けることができることです。アノテーションを使わないバージョンと比べて追加する必要のあるコードは、すべてのメソッドの中からMaxLengthアノテーションを持つメソッドを探すループ処理だけです。

9.2 ケーススタディ：リロイのトラック

デザインパターンは聡明な人が発明するものではなく、既存のコードの中から発見されるものです。ドメインアノテーションにも同じことがいえます。この節では、私たちが初めて使ったドメインアノテーションによく似た2つのドメインアノテーションを紹介します。

この例では、商業上の理由から、私たちが実際に使ったアノテーションやオリジナルのソースコードは使用していません。それらは、ThoughtWorks社が顧客に提供したプロジェクトの成果物の一部だからです。

リロイのトラックは、同僚のMike Royleと筆者が、ドメインアノテーションの使用方法を説明するために作った小規模なサンプルアプリケーションです。このアプリケーションは、先に述べたThoughtWorks社のプロジェクトにおける経験を元にしたものです。そして、オリジナルのプロジェクトと同様に問題領域は物流と出荷です。コードはスマートクライアント上で動作するように書かれています。これは、サーバ上のデータをメンテナンスするWindowsのアプリケーションなのですが、非接続状態でも動作します。オリジナルのアプリケーションと同様に、この例でも実装言語としてC#を使用しました。ただし、コードをわかりやすくするためにC#のバージョン2.0を使っています††。

9.2.1 ドメインモデル

このケーススタディでは、2つの領域のドメインモデルを扱います。1つは倉庫間での商品の配送に関するドメインモデルです。そして、もう1つはシステムのユー

† 訳注：Javaには、このようなアノテーションを使ったバリデーションフレームワークとして、Hibernate Validator（http://validator.hibernate.org/）が存在する。また、JSR 303: Bean Validation（http://jcp.org/en/jsr/detail?id=303）として仕様化が進められている。.NETには、Microsoft Enterprise Library Validation Application Block（http://msdn.microsoft.com/en-us/library/dd140088.aspx）が存在する。

†† 訳注：C#のバージョン2.0でGenericsが導入された。

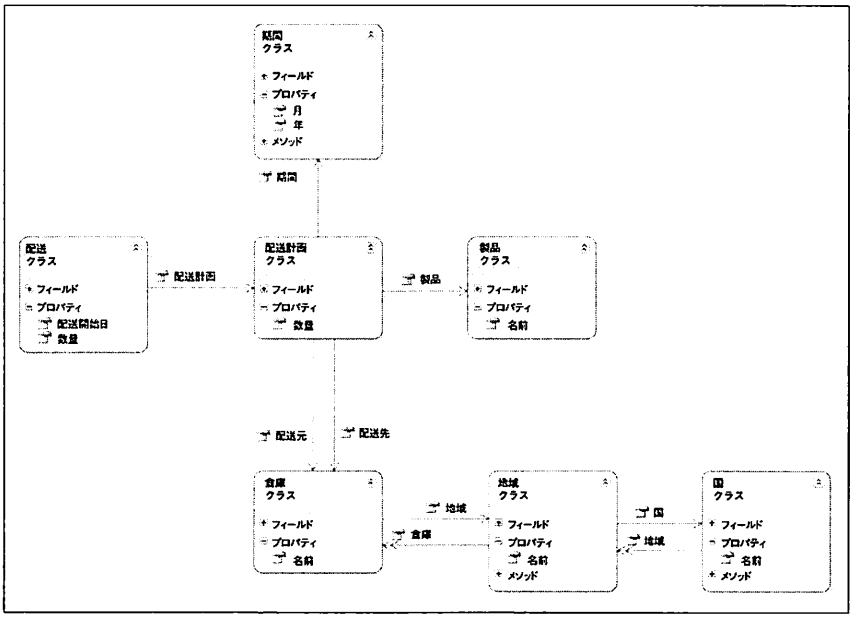

図 9-1 配送

ザとそのロールに関するドメインモデルです。

配送モデルの中心は、配送計画（PlannedTransfer）ドメインオブジェクトです。このオブジェクトは、指定された年月における、配送元（Origin）の倉庫から配送先（Destination）の倉庫への、ある数量（Quantity）の製品の配送計画を表現しています。配送実績は、配送（Transfer）ドメインオブジェクトで表現されます。このオブジェクトは、対応する配送計画への参照を持ちます。配送は、実際の配送開始日（PickupDate）と配送された製品の数量も保持します。配送開始日は、計画時の配送期間内の日付である必要はありません。

また、これ以外に場所に関係するドメインオブジェクトが存在しています。配送元と配送先の倉庫（Warehouse）は、ある国（Country）のどこかの地域（Region）に存在します。

これらすべてのドメインオブジェクトには、2つのドメインアノテーションの使い方を説明するためのデモに必要最低限のプロパティしか持たせていません。リロイのトラックの元になった実際のモデルにおいては、製品は7つのプロパティを持っていました。また、配送は種類ごとに6つのクラスでモデル化され、配送モードのようなプロパティや契約への参照を持っていました。このモデルには別のアプ

ローチで十分だと思うことがあるかもしれませんが、これは単純化されたモデルなのです。

ユーザドメインオブジェクトはシステムのユーザを表現しています。この単純化されたモデルにおいて、ユーザは名前、働いている国への関連を持っています。そして複数のロールを持つことができます。ロールには、計画者（Planner）、各国の管理者（Country Admin）、グローバル管理者（Global Admin）があります。計画者は配送計画を作ったり修正したりします。各国の管理者は、自分の国の倉庫、地域、ユーザデータをメンテナンスします。グローバル管理者は、新しい国を登録し、その国の管理者を任命することができます。

9.2.2 データの分類

最初のドメインアノテーションはデータを分類するためのものです。先ほどのドメインモデルの説明において、すべてのドメインオブジェクトは、ドメインの概念を表現するオブジェクトとしてまったく同じように扱われていました。しかし、考えてみてください。倉庫、地域、国は場所に関するデータです。つまり、これらのオブジェクトはなんらかの共通点があるエンティティだということです。また、ロールには、ユーザはロールに応じて異なるクラスのデータを操作できるという違いがありました。

明らかに、ドメインの中には現在のドメインオブジェクトが表現している以上の情報が存在しています。そして、もうおわかりかもしれませんが、この情報を表現するためにアノテーションを使うのです。

リロイのトラックで使用するアノテーションは、DataClassificationアノテーションです。そして、このアノテーションは、以下の4つのカテゴリのうちのどれかにドメインオブジェクトを分類するために使われます。

- **参照データ**：国、地域、倉庫、製品、ユーザ
- **トランザクションデータ**：期間、配送、配送計画
- **設定**：ロール
- **監査**：監査記録

この実装では、4つのカテゴリを表現するためにenumを使っています。そして、属性はクラスの分類を保持する以上の機能を持ちません。つまり、Javaでも同じような実装になるということです。

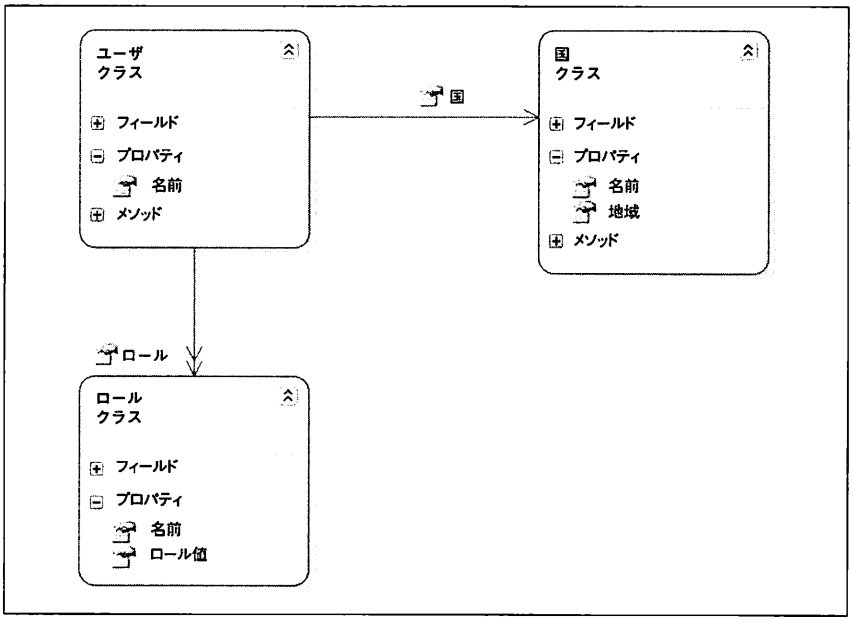

図9-2 ユーザ

```
namespace LeroysLorries.Model.Attributes
{
    public enum DataClassificationValue
    {
        Reference,     // 参照データ
        Transactional, // トランザクションデータ
        Configuration, // 設定
        Audit          // 監査
    }

    [AttributeUsage(AttributeTargets.Class)]
    public class DataClassificationAttribute : Attribute
    {
        private DataClassificationValue classification;

        public DataClassificationAttribute(
                                DataClassificationValue classification)
        {
            this.classification = classification;
        }
```

```
        public DataClassificationValue Classification
        {
            get { return classification; }
        }
    }
}
```

指定されたドメインオブジェクトの型に該当するカテゴリの問い合わせは頻繁に使われるので、ヘルパークラスに以下のようなちょっとしたメソッドを作るとよいでしょう。

```
public static DataClassificationValue GetDataClassification(Type classToCheck)
{
    return
       GetAttribute<DataClassificationAttribute>(classToCheck).Classification;
}

private static T GetAttribute<T>(Type classToCheck)
{
    object[] attributes = classToCheck.GetCustomAttributes(typeof(T), true);
    if (attributes.Length == 0)
        throw new ArgumentException( ... );
    return (T)attributes[0];
}
```

属性クラスにパブリックなスタティックメソッドを作るのもよいでしょう。そうすれば、API を 1 箇所にまとめることができます。この場合、汎用的な処理をヘルパークラスのパブリックメソッドとして作り、そのメソッドを再利用できるようにすべきです。

9.2.2.1 他の方法

アノテーションを使う以外にもデータを分類する方法はあります。すぐに思いつくのは、継承を使うことです。すべての参照データのための共通の基底クラス、たぶん、ReferenceDataObject のようなクラスを作り、Country、Region などがそれを継承するようにすることができます。ドメインオブジェクトの別のクラスに対しても同様の方法を使うことができるでしょう。しかしながら、Java と標準の .NET 言語†は多重継承をサポートしていません。つまり、継承は一度しか使うことので

† 訳注：例えばEiffel .NETはコンパイル時のコード生成によって多重継承をシミュレートしている（http://www.ddj.com/architect/184414864）。

きない技なのです。私たちは、別の問題を解決するために継承が必要になるかもしれないと考えました。

ドメイン駆動設計には、データの分類に継承を使うことに対する、理論的ではありますが強力な反対意見が存在します。ドメイン駆動設計においては、ドメインの専門家と技術者の間でモデルを共有する必要があります。継承はis-a関係を表現するものです。「地域は場所です」と言うことに反対する人はいないでしょう。しかし、もし、分類のために地域と国の親クラスを作ると、「地域は参照データオブジェクトです」ということになりますが、これは現実世界ではまったく意味を持ちません。要するにドメイン駆動設計においては、ドメインオブジェクトの継承は現実世界の分類をモデル化する場合にのみ使うべきだということです。

おそらく、分類のためのデータをドメインオブジェクトに付加する最も簡単な方法は、メソッドを持つインターフェイスを使うことです。

```
interface DataClassification
{
    DataClassificationValue GetDataClassification();
}
```

すべてのドメインオブジェクトは、以下のようにこのインターフェイスを実装し、ハードコードされた値を返します。

```
public DataClassificationValue GetDataClassification()
{
    return DataClassificationValue.Transactional;
}
```

この方法を使うと、ドメインオブジェクトにコードを追加する必要があります。そして、すべてのインスタンスのメソッドの戻り値が同じ定数の場合、ドメインオブジェクトのメソッドとしては不適切です。なぜなら、これはメタデータであり、データではないからです。

この問題には、スタティックメソッドのほうが適しています。しかし、JavaとC#においては、インターフェイスはスタティックメソッドを持つことができません。そして、スタティックメソッドではポリモフィズムを使うことができません。

この他のデータを分類する方法として、マーカーインターフェイスの使用があります。マーカーインターフェイスは、メソッドを持たないインターフェイスです。アノテーションを持たない言語では、これを次善の策として使うことができます。しかし、マーカーインターフェイスの使用は、ある目的のために設計された言語要素を、別の目的のために使うことを意味しています。インターフェイスは、ポリモ

フィックなメソッドを宣言するためのものであり、メタデータを格納するためのものではありません。

その上、もし、このようにインターフェイスを使うと、例えば、基底インターフェイスとしてDataClassificationを作り、さらに分類の値それぞれに対応する4つの派生インターフェイスを作ってしまうかもしれません。これにより、ReferenceDataであるかどうかだけでなく、その基底インターフェイスであるDataClassificationであるかどうかをオブジェクトに尋ねることができるようになってしまいます。ドメインオブジェクトの分類と分類のためのデータの格納場所を混同してしまうと、このような設計を行いがちです†。

9.2.2.2 監査役におけるドメインアノテーションの使用

DataClassificationアノテーションを最初に使うのは、以下のビジネスルールを実装する監査ロジックの中です。

- 参照データが変更された場合のみ、監査記録を作る

このルールは、監査役（Auditor）クラスの中に実装します。Auditorクラスは、実際に監査記録を作る責務を持っています。このクラスは、先に述べたヘルパーメソッドを使用します。

```
private bool ShouldAudit(Type type)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(type);
    return classification == DataClassificationValue.Reference;
}
```

監査記録を作るかどうかを判断するために必要な情報は、アノテーションとしてドメインオブジェクトに格納されます。しかし、この情報に基づくロジックはAuditorクラスに実装します。一般的には、データと振る舞いを一緒にするのはよいことです。しかし、ドメインアノテーションの場合は異なります。同じデータが複数の目的で使われるため、データと振る舞いを一緒にすると、あるアノテーションに関係するすべての振る舞いを一箇所にまとめることになってしまいます。次の

† 訳注：DataClassificationのサブインターフェイスとしてReferenceDataを作り、そのインターフェイスをCountryクラスが実装したとする。すると、「国（Country）はデータ分類（DataClassification）です」ということになってしまう。

節以降では、同じアノテーションのまったく異なる使い方を紹介します。それらを見れば、ドメインアノテーションに関していえば、データと振る舞いを分離するのがよいアイデアだということがわかるでしょう。

9.2.2.3 権限チェック係におけるドメインアノテーションの使用

以下のビジネスルールも、同じDataClassificationアノテーションを使って実装します。

- グローバル管理者のみがすべての参照データを変更できる

先に説明したリフレクションによるヘルパーメソッドを使うことによって、権限チェック係（PermissionChecker）クラスのメソッドの実装が非常に簡単になり、ビジネスロジックの実装に専念することができます。このメソッドはオブジェクトが変更可能かどうかを判断します。

```
public bool CanChangeObject(User user, object anObject)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(anObject.GetType());
    switch(classification)
    {
        case DataClassificationValue.Reference:
            return user.HasRole(RoleValue.GlobalAdmin)
        default:
            return true;
    }
}
```

前のコードとの違いは、アノテーションの値をじかに判断しているのではなく、値によって適切なルールを選択しているところです。

9.2.2.4 ローダにおけるドメインアノテーションの使用

はじめに述べたように、リロイのトラックはサーバと接続していない状態でも動作するスマートクライアントアプリケーションです。つまり、オフラインになる前に、データのワーキングセットがクライアントにダウンロードされていなければなりません。そして、このワーキングセットは、サーバの負荷とダウンロードするデータ量を減らすために、できるかぎり小さくなければなりません。

ここでは、以下のルールを実装するためにDataClassificationを使います。

● トランザクションデータは、計画者の場合のみロードできる

ローダ（Loader）の実装は権限チェック係のものとほとんど同じです。しかし、ドメインオブジェクトの具象メソッドと比較してアノテーションを使用する利点が際立っています†。このオブジェクトを生成するかしないかを判断するメソッドが呼ばれるときは、まだインスタンスが存在していないので、オブジェクトの型を渡しています。オブジェクトを生成するかどうかは、このメソッドが判断します。

```
private bool ShouldLoad(Type type, User user)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(type);
    if(classification == DataClassificationValue.Transactional)
        return user.HasRole(RoleValue.Planner);
    return true;
}
```

リロイのトラックの元になったアプリケーションは、この例よりも複雑なルールと多くの条件分岐があったので、データの分類にアノテーションを使わざるをえませんでした。

9.2.3 ナビゲーションのヒント

このエッセイで紹介する2つめのドメインアノテーションは、ドメインモデルにおける間接的な関連にかかわるものです。例えば、倉庫と国に直接的な関連がなくても、倉庫から地域へ、地域から国への関連をたどっていくことにより、倉庫がどの国にあるのかはわかります。間接的な関連は、倉庫や国のような同じ分類に属するオブジェクトだけにかぎりません。例えば、ドメインモデルを見ると、配送も国に関係しています。しかも2つの国と関係しています。これは配送計画が2つの倉庫に関連しているからです。

リロイのトラックでは、最終的に取得したいオブジェクト、例えば国オブジェクトにたどり着くためにたどる必要のあるプロパティに印を付けるためにアノテーションを使いました。プロパティをたどるとは、再帰的に検索を行うために、関連するドメインオブジェクトを取得し、そのオブジェクトから同じアノテーションが

† 訳注：継承を使った具象メソッドは、インスタンスが生成されていない時点では使うことができない。だからといって、スタティックなメソッドにするとポリモフィズムが使えないので抽象化できなくなる。

付加されているプロパティの検索を繰り返すことです。

アノテーションの値が必要ないので、実装は先ほどの分類の例より簡単です。つまり、Java で実装してもほとんど同じになるということです。

```
namespace LeroysLorries.Model.Attributes
{
    [AttributeUsage(AttributeTargets.Property)]
    public class CountrySpecificationAttribute : Attribute
    {
    }
}
```

次のコードは、リロイのトラックの Warehouse クラスから倉庫の名前に関するコードを取り除き、関連を表すプロパティに属性を付加したものです。

```
namespace LeroysLorries.Model.Entities
{
    [DataClassification(DataClassificationValue.Reference)]
    public class Warehouse
    {
        private Region region;

        [CountrySpecification]
        public Region Region
        {
            get { return region; }
            set { region = value; }
        }
    }
}
```

アノテーションと一緒に、汎用的なパスを検索するクラスを使えば、検索対象のオブジェクトを取得することができますし、指定した型から検索対象をたどっていくためのパスを表す文字列の配列を取得することもできます。以下の 2 つの例を見れば、パス検索クラスの使用方法がわかるでしょう。

```
Warehouse warehouse; // どこかから取得する
PathFinder<Country> finder = new PathFinder<Country>();
Country country = finder.GetTargetObject(warehouse);
```

Country を取得するための PathFinder クラスを生成した後、指定された Warehouse

の存在するCountryを取得するためにPathFinderを使っています。特筆すべきなのは、PathFinderがどの属性をたどるのかを決定するために命名規則を使用していることです。

型の名前、この例ではCountryの後にSpecificationAttributeを付加し、その名前を持つ属性をAttributes名前空間から探しています。汎用のPathFinderが必要な理由は、このエッセイの後半で説明します。

```
PathFinder<Country> finder = new PathFinder<Country>();
string[] path = finder.GetPath(typeof(Warehouse));
```

2つめの例では、PathFinderは、たどっていくパスを「Region」と「Country」という要素の格納された文字列の配列として返します。この文字列の配列は、それぞれWarehouseとRegionクラスのプロパティ名を表しています。このメソッドは、インスタンスではなく型を引数にとるので、ドメインオブジェクトのインスタンスが生成される前に使用することができます。

```
public class PathFinder<T>
{
    private static string NAMESPACE = "LeroysLorries.Model.Attributes.";
    private Type attrType;
    public PathFinder()
    {
        string typeName = NAMESPACE + typeof(T).Name + "SpecificationAttribute";
        if((attrType = Type.GetType(typeName)) == null)
            throw new ArgumentException( ... );
    }

    public T GetTargetObject(object anObject)
    {
        Type objectType = anObject.GetType();
        if(objectType == typeof(T))
            return (T)anObject;
        PropertyInfo propInfo = ReflectionHelper.GetPropertyWithAttribute(
                                                    objectType, attrType);
        object nextObject = ReflectionHelper.GetPropertyValue(
                                                anObject, propInfo.Name);
        return GetTargetObject(nextObject);
    }

    public string[] GetPath(Type type)
    {
```

```
        List<string> path = new List<string>();
        if(BuildPath(type, path) == false)
            throw new ArgumentException( ... );
        return path.ToArray();
    }

    private bool BuildPath(Type type, List<string> path)
    {
        if(type == typeof(T))
            return true;
        PropertyInfo prop = ReflectionHelper.GetPropertyWithAttribute(
                                                        type, attrType);
        if(prop == null)
            return false;
        path.Add(prop.Name);
        return BuildPath(prop.PropertyType, path);
    }
}
```

9.2.3.1 他の方法

データ分類の例と同様に、アノテーション以外の方法を使うこともできます。最も一般的な方法は、すべてのドメインオブジェクトに対して、間接的な関連を持つクラスのプロパティを実装し、関連を直接ハードコードすることでしょう。

例えば、以下のプロパティを Warehouse に追加することができます。

```
public Country Country
{
    get { return region.Country; }
}
```

Country 用の PathFinder は、CountrySpecification の付加されたプロパティを検索するのではなく、単に Country 型のプロパティを呼び出すだけになります。この程度であれば、うまくカプセル化されたオブジェクト指向のコードといえるでしょう。しかしながら、ドメインモデルをメンテナンスするには、多くの修行†と作業が必要です。

より自動化されたアプローチをとる場合、グラフ探索を使うことになるでしょう。自動化アプローチを使えば、ドメインモデルに余分なコードを追加する必要がなく

† 訳注：この方法を使う場合、コードのコピー、ペーストが発生しやすいので、膨大な数のクラスを間違いのないように修正を行うための修行が必要となる。

なります。ドメインオブジェクトの型と関連は有向グラフとして考えることができます。ドメインオブジェクトから標準的な深さ優先検索、あるいは幅優先検索を開始すれば、検索アルゴリズムを使い、目的のオブジェクトを発見することができます。この方法は型だけでなくインスタンスに対しても有効です。したがって、この方法でPathFinderを完全に実装することもできます。

検索アルゴリズムによって発見されたパスをキャッシュすることにより、このアプローチはよりうまく機能します。ただし、これはパスが一意に決まり、ドメインロジックが必要ない場合のみです。残念ながら、リロイのトラックはこのケースに当てはまりません。配送計画には、配送元倉庫と配送先倉庫のプロパティがあります。もし、これらの倉庫が異なる国にある場合、検索アルゴリズムを使うためにはたどる倉庫を決定するための情報が必要です。アノテーションを使う場合、配送計画は配送元の国ごとに一覧としてまとめられるというドメインの知識を使い、配送元のプロパティにCountrySpecificationを付加することができます。

9.2.3.2 権限チェック係におけるドメインアノテーションの使用

話を前の節で説明した権限チェック係に戻しましょう。以下のビジネスルールを実装するために、CountrySpecificationを使います。

- 各国の管理者は、自分の国の参照データのみを変更できる

これを実装するためにCanChangeObject()メソッドを拡張します。switch文の参照データのcaseの中に新しいルールを追加したコードは次のようになります。

```
public bool CanChangeObject(User user, object anObject)
{
    DataClassificationValue classification =
        ReflectionHelper.GetDataClassification(anObject.GetType());
    switch(classification)
    {
        case DataClassificationValue.Reference:
            if(user.HasRole(RoleValue.GlobalAdmin))
                return true;
            if(user.HasRole(RoleValue.CountryAdmin))
                return FindCountry(anObject) == user.Country;
            return false;
        default:
            return true;
    }
}
```

新しいルールを追加した後も、グローバル管理者はすべての参照データを変更することができます。しかし、各国の管理者の場合、参照データの属している国が管理者の国と一致している必要があります。コードをきれいに保つために、実際にPathFinderを使用するコードはヘルパーメソッドとして抽出します。

```
private Country FindCountry(object anObject)
{
    return new PathFinder<Country>().GetTargetObject(anObject);
}
```

この例では、個々のアノテーションのさまざまな使い方だけでなく、これらのアノテーションを一緒に使うことによって、関心事の分離を犠牲にすることなく、ビジネスルールをわかりやすく簡潔に実装できることを紹介しました。

9.2.3.3 ローダにおけるドメインアノテーションの使用

計画者は国ごとに割り当てられ、その国の配送のみを計画することができます。同様に、各国の管理者は自分の国のデータだけをメンテナンスすることができます。クライアントアプリケーションにダウンロードされるデータの量を考慮すると、以下のルールに従ったデータの削減による最適化は必須です。

- ユーザがグローバル管理者ではない場合、ユーザの属する国のデータのみがダウンロードされるべきである

CountrySpecificationとPathFinderを使うと、それぞれのドメインオブジェクトクラスからCountryオブジェクトをたどるためのパスを決定することができます。この後、メモリ上にフェッチされるオブジェクトを制限する問い合わせ条件にこのパスを含めるのは、オブジェクトリレーショナルマッピング・テクノロジの役目です。

以下のコードは、このアイデアを抽象化した実装です。

```
private Query CreateQuery(Type type, User user)
{
    QueryBuilder builder = new QueryBuilder(type);
    if(!user.HasRole(RoleValue.GlobalAdmin))
    {
        PathFinder<Country> finder = new PathFinder<Country>();
        string[] path = finder.GetPath(type);
        builder.AppendCondition(path, QueryBuilder.EQUALS, user.Country);
    }
```

```
        return builder.GetQuery();
    }
```

引数の type と user に対して、前の節で説明した ShouldLoad() メソッドが true を返した場合にのみ、このメソッドが呼ばれてオブジェクトがロードされます。

リロイのトラックにおいては、場所に関する情報は重要な側面の1つにすぎません。期間と計画者の月次の作業も同様に重要です。というのも、過去の履歴を参照する場合を除いて、計画者は前月、今月、来月の3か月間のデータにしか関心がありません。このことから、何年分ものデータをロードするのではなく、以下のルールと、それ以外のデータの遅延ロードを実装しました。

- トランザクションデータは、前月、今月、来月のものだけロードされなければならない

このルールは、計画者の場合のみトランザクションデータはロードされなければならないという先のルールへの追加条件となります。

PathFinder の汎用的な実装と、SpecificationAttribute の名前と検索の規則を使うことにより、以下の属性を作成し、配送と配送計画を表現するドメインオブジェクトに付加することができます。

```
namespace LeroysLorries.Model.Attributes
{
    [AttributeUsage(AttributeTargets.Property)]
    public class PeriodSpecificationAttribute : Attribute
    {
    }
}
```

これにともない、先に説明した CreateQuery() メソッドを以下のように拡張しました。

```
private Query CreateQuery(Type type, User user, Period period)
{
    QueryBuilder builder = new QueryBuilder(type);
    if(!user.HasRole(RoleValue.GlobalAdmin))
    {
        PathFinder<Country> finder = new PathFinder<Country>();
        string[] path = finder.GetPath(type);
        builder.AppendCondition(path, user.Country);
```

```
        }
        if(ReflectionHelper.GetDataClassification(type) ==
        DataClassificationValue.Transactional)
        {
            PathFinder<Period> finder = new PathFinder<Period>();
            string[] path = finder.GetPath(type);
            builder.AppendCondition(path, period);
        }
        return builder.GetQuery(n);
    }
```

最後の例では、ドメインアノテーションとそれを使用した汎用のアルゴリズムによって、異なる関心事をうまく分離する方法を紹介できたと思います。ドメインの知識を使用するデータレイヤにおける最適化は、きれいにドメイン部分に分離されています。つまり、ドメインモデルの中に実装されているのです。そして、データアクセスロジックが、データレイヤ以外の場所に漏れ出していることもありません。

また、この例ではケーススタディの3つのアノテーションすべてを使いました。これにより、アノテーションを使って横断的関心事を実装できること、そして、それをコードベースの1箇所にまとめることができるということがわかったと思います。

9.3 まとめ

Javaのアノテーションと C#の属性は、メタデータを表現する仕組みをプログラミング言語に追加しました。この仕組みを使うと、メタデータを明確かつ拡張可能なかたちで表現することができます。ドメイン駆動設計のプロセスに従うと、アプリケーションの対象ドメインに属するメタデータは、アノテーションとしてドメインオブジェクト上に表現することができます。私たちは、このようなアノテーションをドメインアノテーションと呼ぶことを提案します。ドメインアノテーションはドメインモデルの中だけに存在し、通常、ドメインモデルと同じパッケージや名前空間に定義されます。そして、通常、複数の用途に使われます。

ドメインアノテーションを使用すると、ドメイン固有のコードとインフラストラクチャコードを簡単に分離でき、それぞれのコードを別々に再利用することができます。この分離により2つの恩恵を受けることができます。ドメインに関するすべての知識はドメインモデルの中に表現され、異なるインフラストラクチャテクノロジと一緒に再利用することができます。そして、インフラストラクチャコードを、

商用製品化、あるいはオープンソース化することができます。これにより、アプリケーション開発者はアプリケーションの対象ドメインに注力し、長期にわたり価値を持つものを作ることができます。

10章
Antビルドファイルのリファクタリング

Julian Simpson @ビルドアーキテクト

ビルドファイルを構築するとき、保守するとき、
コードに対する完全主義者の理想は消え失せるように思える。
——Paul Glover

10.1 イントロダクション

「悪い」ソフトウェアビルドシステムは積極的に変更を阻害するように見えるものです。ちょっとした変更のせいで、同僚の仕事を妨げてしまうことがあります。そういう認識をしてしまうと、いずれはやらなくてはいけないとしても、ビルドを改善する試みから人々を遠ざけてしまうでしょう。このエッセイでは、そのような痛みを和らげて変更を可能にする方法として、Antビルドファイルに適用する簡潔なリファクタリングをいくつか紹介します。各リファクタリングは、リファクタリング前と後をビフォアアフター形式のコード例（間に変換を表す矢印があります）で示します。読者の皆さんは、いくつかの具体的なツール、Antビルドファイルを小さく読みやすくして、そしてその修正を楽にしてくれるツールを手に入れることができるでしょう。

10.1.1 リファクタリングとは。Antとは

リファクタリングは、コードの読みやすさ、明確さ、保守の容易さを改善するために、システムに小さな変更を加える技法です。リファクタリングは機能面の変更をしません。コードの内容を整理するために用いられます。Antは多くのJavaベースのソフトウェアプロジェクトで利用されるビルドツールです。Antは、XMLがソフトウェアのさまざまな問題に対するソリューションだと考えられていた頃に作られたツールです。そのこともあって、プロジェクトの依存関係は、通常build.xmlと呼ばれるXMLファイルに記述します。これはJavaにおけるMakefileと言えます。Antはオープンソースのプロジェクトとしてとても成功しています。しかし、ソフトウェアプロジェクトが大きくなり、柔軟性を求められ、より複雑になるに従って、

ほとんどの場合、build.xmlは保守が困難なものになってしまいます。

10.1.2 いつリファクタリングをすべきか。あるいは、いつ逃げ出すべきか

各リファクタリングの詳細に入る前に、リファクタリングのエクササイズ全体にわたる背景と目的を設定しましょう。私たちは今、稼動するソフトウェアを顧客に提供するためにここにいます。そのためにはソフトウェアをビルドする必要があります。時には、ビルドを変更する必要があるでしょう。もし皆がビルドの変更を嫌がれば、ソフトウェアを提供できません。それでは困ったことになります。

そのため、このエッセイではビルドの変更のコスト（または痛み）を下げようと試みています。もちろん、全体像を考慮する必要があります。ビルドが痛みの主な原因なのでしょうか。ビルドがソフトウェアの提供を遅らせていますか。プロジェクトに関する問題のほうがビルドの問題よりも大きくないですか。すぐに逃げ出したほうがよくありませんか。

あなたがまだここにいるなら、ビルドの問題も依然として残っています。問題はどれほど大きいのでしょうか。もし、現実にスパゲティモンスター†のようなビルドファイルがあるなら、このエッセイはあなたの役に立ちます。けれども、とてもとても用心深く進めてください。最初は削除可能なものを探しましょう。おそらく、使われないターゲットがいくつかあるはずです。そこから手を付けていきます。筆者は、プロジェクトでSimian similarity analyzer††を使って成功しています。このツールを使って非常に害のある重複箇所を探し、その後、「macrodefの抽出」もしくは「ターゲットの抽出」リファクタリングを適用します。

多くのリファクタリングは共存可能です。前述の「抽出」のリファクタリングは、状況によっては互いに排他的になることがあります。ただ、効果という点ではどちらも違いはないでしょう。リファクタリングの動機には幅があります。誰でも理解できるコードにするために「descriptionによるコメントの置き換え」リファクタリングを適用することもあるでしょう。意図しない方法でビルドされるのを防ぐために「内部ターゲットの強制」リファクタリングを適用して隠蔽することもあるでしょう。

† 訳注：http://www.venganza.org/
†† 訳注：http://www.redhillconsulting.com.au/products/simian/

10.1.3 build.xmlはリファクタリング可能か

リファクタリングのエクササイズ中に、外部から見たときのビルドの振る舞いを定義するのは簡単です。一般的にそれは与えられたソースファイルから、なんらかの成果物（生成されたコード、コンパイル済みコード、テスト結果、ドキュメント、WARファイルのようにデプロイ可能なもの）を得る、という振る舞いになります。

Antビルドファイルは、ビジネスアプリケーションのコードと同様にリファクタリングする価値があるものです。ただ、プログラミング言語に比べてエラーへの耐性がありません。一部のエラーは、それが生じてもすぐにビルドを壊すようなことはありません。しかし、後々、思いがけない形でビルドは失敗します。例えば、プロパティの設定に失敗しても、JavaやRuby、Pythonにおける変数の未定義時と同じようには、ビルドは停止しないでしょう。テストによるセーフティネットやIDEツールなしで、規律あるリファクタリングを適用することが課題です。

リファクタリングは通常、コードに対して機能的な影響を与えていないと保証するために、単体テストを必要とします。しかし、Antビルドファイルのリファクタリングでは、変更の影響を確認するツールはありません。JavaのJUnitやRubyのTest::Unitのようによく目にするテストフレームワークはなく、スタブを使ってテストする部分を切り出す便利なツールもありません。しかし、たとえそのようなツールがあったとしても、その効果は疑問でしょう。テスト駆動しなければいけないほど複雑なビルドシステムが必要なのでしょうか。

さらに悪いことに、Antはコンパイル時に型チェックされない静的型付け言語です。ビルドファイルはXMLファイルなのですが、決められたDTDがないために決して検証されません。そのため、整理されていないビルドファイルの変更は、たいてい高いリスクを抱えています。1つの変更が生産性に打撃を与えかねないからです。リファクタリングプロセスの開始時点では、変更のテストをチェックイン前に部分的に行うのは困難かもしれません。とても小さな単位での変更とテストを頻繁に行う必要があるでしょう。ビルドの内部構造がシンプルになるにつれて状況は好転し、積極的にリファクタリングできるようになるはずです。

10.2 Antリファクタリングカタログ

それぞれのリファクタリングには簡潔な要約になるような名前が付いていて、実際にリファクタリングしている例も付いています。最初のコード断片がオリジナルで、次のコード断片がリファクタリング後のものです。長い説明がコード断片の後にあり、またいくつかのリファクタリングではコラム記事が追加されています。

リファクタリング名	説明
macrodefの抽出	Antのコードブロックを小さな単位で取り出し、適切な名前のmacrodefの定義に置き換える
ターゲットの抽出	大きなターゲットの一部を取り出して独立したターゲットとして宣言し直し、元のターゲットに依存関係を宣言する
宣言の導入	ターゲットにその依存関係を宣言する
依存によるcallの置き換え	antcallを使った呼び出しを、ターゲットのdependsで置き換える
プロパティによるリテラルの置き換え	繰り返し用いるリテラルをプロパティに置き換える
filtersfileの導入	ネストしたfilter要素の代わりにfilterset内でプロパティファイルを使用する
プロパティファイルの導入	build.xml内のproperty要素を、フラットな外部プロパティファイルに移動する
ターゲットのラッパービルドファイルへの移動	開発者が利用しないターゲットは、上位のラッパービルドファイルに移動し、そこから開発者用のビルドを呼び出す
descriptionによるコメントの置き換え	要素への注釈は、XMLコメントではなくdescription属性内に記述する
デプロイ用コードのimport先への分離	デプロイ用のコードを外部ファイルに分離し、importする。それにより、ビルド時に適切なデプロイ用コードを選択できる
要素のantlibへの移動	複数のプロジェクトで頻繁に使われるタスクはantlibを使って共有する
filesetによる多数のライブラリ定義の置き換え	ライブラリのパスを個別に指定するのではなく、globによりライブラリを発見するよう、filesetを使用する
実行時プロパティの移動	コードのビルドに用いるプロパティと、実行時の設定に関するプロパティを明確に分離する
IDを用いた要素の再利用	filesetのようなデータ型を宣言したら、他の場所から参照して利用することにより、重複を排除する
プロパティのターゲット外部への移動	プロパティはbuild.xmlのボディの直下に定義することにより、これらのプロパティがターゲット内のみのスコープであるという勘違いを防ぐ
locationによるvalue属性の置き換え	Antが正規化できるよう、location属性を用いてファイルシステムのパスを表す
build.xml内へのラッパースクリプトの取り込み	入力の検証やクラスパスの操作をbuild.xml内に入れて、クロスプラットフォームなAntスクリプトに含める
tasknameの追加	taskname属性を追加することにより、そのタスクが実行時に何を意図しているかを明示する
内部ターゲットの強制	コマンドラインから内部ターゲットを実行できないようにする。
出力ディレクトリの親ディレクトリへの移動	1つのディレクトリ以下のディレクトリツリーに、すべてのアウトプットを生成する
applyによるexecの置き換え	外部実行の入力情報にはarg要素のリストではなくパス類似構造を渡す
CI Publisherの利用	ビルドに対してタグを付け公開する場合は、ビルド実行中にタグを付けるのではなく、ビルドが完了してから外部ツールを使って行う
明確なターゲット名の導入	可読性を向上するために、targetとpropertyには異なる命名規則を使用する
ターゲット名の名詞への変更	ターゲットの名称は、それが行うプロセスではなく、出力する成果物から命名する

10.2.1 macrodefの抽出

概要：macrodefにより複雑なビルドファイルを小さな部分に分割して整理することができる

refactoring_before.xml

```
<target name="build_and_war_foo.war">
    <javac srcdir="src/foo" destdir="classes/foo" />
    <copy todir="${classes.dir}">
        <filterset>
            <filter token="ENV" value="${environment}" />
        </filterset>
        <fileset dir="config" />
    </copy>
    <war destfile="foo.war">
        <fileset dir="${classes.dir}" />
    </war>
    <move todir="archives" file="foo.war" />
</target>
```

refactoring_after.xml

```
<macrodef name="build_code">
    <attribute name="component" />
    <sequential>
        <javac srcdir="src/@{component}" destdir="classes/@{component}" />
        <copy todir="${classes.dir}">
            <filterset>
                <filter token="ENV" value="${environment}" />
            </filterset>
            <fileset dir="config" />
        </copy>
    </sequential>
</macrodef>

<macrodef name="make_war">
    <attribute name="component" />
    <sequential>
        <war destfile="@{component}.war">
            <fileset dir="${classes.dir}" />
        </war>
        <move todir="archives" file="@{component}.war" />
```

```
    </sequential>
</macrodef>

<target name="foo.war" >
    <build_code component="foo"/>
    <make_war component="foo"/>
</target>
```

巨大なターゲットは、オブジェクト指向言語における巨大なメソッドと同じ匂いを持っています。これらは脆弱でテストやデバッグがやりづらく、再利用も困難になります。

このようなターゲットを変更するのは困難です。なぜなら、それぞれの行が他のターゲットに対する暗黙の依存関係を持っているかもしれないからです。長いターゲットは、ビルドファイルの作成者の意図を伝えられず、読み手を混乱させてしまうでしょう。

macrodefタスクはタスクのコンテナとなります（逐次もしくは並列なタスクをラップしたり、再利用したいタスクを含めることができます）。ビルドファイルのどこからでも、属性を指定した上で呼び出し可能です。属性にはデフォルトの値を与えることも可能で、これは使い方が何種類もあるような特殊なmacrodefに対して有効です。

macrodefは再利用への道も開きます。先ほどの例ではターゲットが多くのことを行いすぎていますが、ターゲットの一部を取り出して小さなXMLに分解することができます。次のステップをターゲットに導入します。

refactoring_before.xml

```
<target name="bar.war">
    <war warfile="bar.war" basedir="classes/bar"/>
</target>

<target name="baz.war">
    <war warfile="baz.war" basedir="classes/baz"/>
</target>
```

refactoring_after.xml

```
<macrodef name="war">
    <attribute name="name"/>
```

```
    <sequential>
        <war warfile="'@{name}.war" basedir="classes/@{name}"/>
    </sequential>
</macrodef>
```

ビルドファイルは重複しやすいものです。過去のバージョン（Ant 1.6より前のバージョン）では再利用のためにantcallタスクが用いられていましたが、macrodefはそれに代わる優れた方法となります†。先ほどの例でも重複を含んでいましたが、異なる属性を渡して共通のmacrodefを何度も呼び出す形に置き換えます。

macrodefをどの程度書くべきなのでしょうか。もちろん、複雑なことをする必要があれば、大きなmacrodefを書かなければいけません。しかし、ターゲットで記述すればよいことにはmacrodefを用いるべきではありません。XML言語は大きく複雑なことをするには不向きですので、Antタスクを Javaや動的言語で書くことを検討しましょう。scriptdefタスク††を使ってRubyやPythonといった動的言語でタスクを書けば、XMLと動的言語の両方の長所を活かすことができます。コンパイルなしに、テスト可能なコードを書けるようになります。ただし、Antオブジェクトモデルの学習にいくらか時間を費やす必要があることに注意してください。

10.2.2 ターゲットの抽出

概要：異なる種類のタスクを実行しているように見えるターゲットは、2つもしくはそれ以上のターゲットに分解する

refactoring_before.xml

```
<target name="test" >

    <javac srcdir="${test.src}" destdir="${test.classes}">
        <classpath refid="test.classpath"/>
    </javac>

    <junit failureproperty="test.failure">
        <batchtest todir="${test.results}">
            <fileset dir="${test.results}"
```

† 訳注：antcall呼び出しに対するmacrodefの利点としては、パラメータにデフォルト値を設定可能な点、パス類似構造などのデータ型やタスクそのもののようなXML断片をパラメータとして渡せる点、などがある。

†† 訳注：ビルドファイルの中にスクリプトを直接記述することができる。

```
                includes="**/*Test.class"/>
        </batchtest>
    </junit>
</target>
```

refactoring_after.xml

```
<target name="compile_tests" depends="compile_code">
    <javac srcdir="${test.src}" destdir="${test.classes}">
        <classpath refid="test.classpath"/>
    </javac>
</target>

<target name="unit_tests" depends="compile_tests">
    <junit failureproperty="test.failure">
        <batchtest todir="${test.results}">
            <fileset dir="${test.results}"
                includes="**/*Test.class"/>
        </batchtest>
    </junit>
</target>
```

長いAntのターゲットは理解しにくく、トラブルシューティングや拡張が困難です。ついついやってしまうことですが、既存のターゲットにいろいろ追加する変更を安易に繰り返していると、そうなってしまいます。このような悩ましい機能を正しい依存関係で複数のターゲットに分割すれば、ビルドファイルはクリーンに保たれメンテナンスしやすくなります。

どんなときにmacrodefの抽出を使い、どんなときにターゲットの抽出を使うべきでしょうか。依存関係のあるコードブロックの場合は、ターゲットにします。コマンドラインから実行したい場合（例えば、開発者個人用のデータベーススキーマのdropなど）もターゲットにします。実際、ほとんどの場合でターゲットを作成したくなるでしょう。パスや入力が異なるだけで、コードブロックが重複しているような箇所は、異なる属性で呼び出せるように、macrodefの抽出を行います。大きなプロジェクトにおけるコンパイルや単体テストでは、複数のソースツリーやいろいろな種類のテストがあるため、それぞれを実行するmacrodefを呼び出すターゲットにするのがお勧めです。

一方、antcallを使いたい状況でも、macrodef呼び出しを使うというテクニックがあります。実際の例としては、antcallをビルド状況のチェックなどに使ってい

るときなどです。antcallは内部でprojectオブジェクトを毎回作成するせいで低速になることもあるので、このテクニックを利用します†。

10.2.3 宣言の導入

概要：デバッグしにくくなるif条件分岐の代わりに、Ant組み込みの宣言的ロジックを利用する

refactoring_before.xml

```
<target name="deploy">
    <if>
        <equals arg1="${j2ee.server}" arg2="was" />
        <then>
            <antcall target="was_deploy"/>
        </then>
        <else>
            <antcall target="weblogic_deploy"/>
        </else>
    </if>
</target>
```

refactoring_after.xml

```
<property name="j2ee.server" value="was" />
<import file="${j2ee.server}.build.xml" />
<!-- there is now a an appropriate target named
        deploy depending on the version of the app server -->
```

この例では、適切なバージョンのターゲットに分岐するロジックを使っています。これは一見、自然に見えますが、XMLベースの言語ではわかりやすく表現しにくいものです。XMLは本来、データを表現することを意図しています。ロジックの表現には向きません。さらにAntは宣言型の言語††です。タスク実行時の細かい制御は

† 訳注：antcallは呼び出しのたびに、projectから始まるオブジェクトモデルを、まるごと子プロジェクトとして再生成し実行する。そのためターゲットの中で何回も呼び出すような場合は非常に重くなる可能性があり、細かい問題も多い。

†† 訳注：プログラムとして、処理の手順ではなく問題や出力結果が持つべき性質や制約を記述するプログラミング言語。純粋関数型言語や制約論理型言語などがこの範疇に含まれる。またDSLなどの記述にも利用されることがある。

できませんので、ちょっとしたヒントを与えて、Ant自身が正しい順序でタスクを実行するようにします。Antビルドファイルに宣言したターゲットには、depends属性を利用して依存関係を宣言します。もしもなんらかの分岐するロジックが必要になったら、それはビルドファイルの再検討が必要になる兆しです。

ビルドファイルの中にたくさんのif-else要素があるなら、このテクニックは特に有用です。大量の分岐構造を取り除き、とても明確なビルドファイル群に変えられます。オブジェクト指向のコードに慣れた開発者なら、Antが、コンテキストにより同じ名前で違う振る舞いをするターゲットを持てる点は、ポリモフィズムの例に見えるかもしれません。

Antはバージョン1.6からimportタスクを利用できるようになりました。プロパティと一緒にこの機能を利用することで、必要な振る舞いを持ったファイルをimportすることができます。

ant-contrib

ant-contribプロジェクトの「if」要素†により、Antにスクリプティング能力を持たせることができます。しかしAntのオリジナル開発者とメンテナたちは、Antがスクリプティング言語になるべきではないと考えています。注意して利用してください。

10.2.4 依存によるcallの置き換え

概要：明示的な呼び出しの代わりに、Antが管理する依存関係の仕組みを利用する

refactoring_before.xml

```
<target name="imperative_build">
    <antcall target="compile"/>
    <antcall target="test"/>
</target>
```

† 訳注：http://ant-contrib.sourceforge.net/tasks/tasks/if.html

refactoring_after.xml

```
<target name="declarative_build" depends="test, publish "/>
<target name="test" depends="compile"/>
```

Antのような依存関係ベースのビルドツールはすべて、ターゲットが何回も実行されるのを防ぐようになっています。antcallは、タスクの強制的な実行によって、Ant本来の宣言的な性質を壊してしまいます。タスクを再利用するために、異なるパラメータを渡して呼び出す形にすることはよくあります。しかし、antcallを使っている場合、特にdependsとantcallを同時に利用している場合は、ターゲットが2回もしくはそれ以上実行されてしまうことが、容易に起きます。

この場合の正解は、deployターゲットがcompileとtestターゲットに依存している、と宣言することです。testターゲットは、それ自身もコンパイルに依存しています。Antは、このような依存関係のツリーを解決して、正しい順序で実行するよう設計されています。宣言した順序どおりに実行させようとしてもうまくいきません。なぜなら、依存関係を満たすために実行順序が調節されるからです。

10.2.5 プロパティによるリテラルの置き換え

概要：ビルドファイルでは、繰り返し用いるリテラルをプロパティに置き換える。外部環境の値を取得するには、組み込みのJavaプロパティやAntプロパティを利用する

refactoring_before.xml

```
<target name="deploy_to_tomcat">
    <copy file="dist.dir/webapp.war" todir="tomcat.webapps.dir"/>
</target>
```

環境変数

よくある「不吉な匂い」として、Antがインポートした環境変数からユーザや実行しているOSの名前を取得している、というものがあります。これは動きはしますが、ビルドシステムが外部に依存することになり、システムを脆弱にしてしまいます。Javaのシステム変数は、デフォルトでAntビルドファイルの名前空間に取り込まれます。環境変数をインポートして見つけたエントリから情報を取得する代わりに、user.nameやos.nameといった組み込みプロパティを使ってください。

refactoring_after.xml

```
<property name="dist.dir" location="${build.dir}/dist"/>
<property name="tomcat.webapps.dir" location="/opt/tomcat5/webapps"/>
<target name="deploy_to_tomcat">
    <copy file="${dist.dir}/webapp.war" todir="${tomcat.webapps.dir}"/>
</target>
```

ビルドファイルの中の静的および動的な文字列はプロパティを使って表現することが必要です。クラスをコンパイルするディレクトリはめったに変わらないでしょうが、変更が生じたときには、5箇所も修正するのではなく1箇所の修正で済むようにしなければいけません。大雑把に言って、3回同じ文字列をタイプしたことに気づいたら、プロパティにするべきです。「location による value 属性の置き換え」リファクタリングに、プロパティ内でファイルシステムのパスを表現する方法があります。Antではプロパティを変更できないことを常に覚えておいてください。つまり、最初に指定した値がプロパティにずっと紐付きます。これは、最後に評価されるようにしたプロパティファイルの中でデフォルト値を設定しておくことで、ビルドファイルの外側でプロパティをオーバーライド†できることを意味します。

10.2.6 filtersfileの導入

概要：filter要素のセットではなく、置き換えたい要素と値のマッピングをプロパティファイルに記述し、テンプレートからはプロパティを直接参照する

refactoring_before.xml

```
<target name="filter">
    <copy todir="${build}" file="${src}/config/config.xml">
        <filterset>
            <filter token="APP_SERVER_PORT" value="${appserver.port}"/>
            <filter token="APP_SERVER_HOST" value="${appserver.host}"/>
            <filter token="APP_SERVER_USERID" value="${appserver.userid}"/>
        </filterset>
```

† 訳注：Antには一般的な意味のオーバーライドはないが、評価順序の機構を利用して似たことができる。例えば起動時引数でプロパティを設定できるが、起動時引数は最初に評価されるため、それ以降のbuild.xml内の同一プロパティ宣言は無視される。これでbuild.xmlのプロパティ宣言がオーバーライドされているように「見える」。デフォルト用、カスタマイズ用のプロパティファイルを用意した場合は、カスタマイズ用を最初に読み込むようにする。

```
    </copy>
</target>
```

refactoring_after.xml

```
<target name="filtersfile">
    <copy todir="${build}" file="${src}/config/config.xml">
        <filterset filtersfile="appserver.properties"/>
    </copy>
</target>
```

appserver.properties

```
appserver.port=8080
appserver.host=oberon
appserver.userid=beamish
# END filtersfile
```

ビルドファイルはすぐに読みにくくなってしまいます。プロパティをプロジェクトチームのどのメンバが読んでも理解できるプレインテキストで記述するのが、最もよい方法になることがあります。例では、このテクニックをfiltersetに適用しています。多くのビルドシステムでは、複数の異なる環境に対応する必要があるときに、テンプレートのトークンを特定の値で置換します。IDなどを使ってfiltersetを再利用していない場合、ビルドファイルの大部分がトークンの記述で占領されてしまいます。このアプローチには2つのメリットがあります。トークンを使わずに直接プロパティの名前を指定できること、プロパティファイルとして分離することでXMLファイルを壊さずに誰でも編集できることです。例にあるようなコピー要素の子要素としてfiltersfileを1つ以上指定することもできます†。Antには最初に指定されたものが有効になるという原則があるため、デフォルト値を持たせることもできます。フィルタファイルは通常のプロパティファイルなので、ビルドの別のところでも利用することができます。

† 訳注:複数のfiltersfileを指定するにはfiltersetをcopyなどの子要素として複数定義する。

どの程度プロパティを定義すればよいか？

少ししか定義しなかった場合、最後には不恰好に連結されたプロパティを多用することになり、DRYの原則†に反することになります。たくさん定義した場合、重複するプロパティができてしまったり、プロパティを覚えきれなかったりしてしまいます。ビルドファイルを複数に分離すれば、プロパティをファイルごとの範囲に収めることができるでしょう。

10.2.7 プロパティファイルの導入

概要：あまり変更されないプロパティはメインのビルドファイルから別ファイルへ移動する

refactoring_before.xml

```
<property name="appserver.port" value="8080" />
<property name="appserver.host" value="oberon" />
<property name="appserver.userid" value="beamish" />
```

refactoring_after.xml

```
<property file="appserver.properties" />
```

appserver.properties

```
appserver.port=8080
appserver.host=oberon
appserver.userid=beamish
# END filtersfile
```

Antのビルドファイルでは「定数」のようなものは存在しません。なぜなら、そもそもプロパティが常にイミュータブルなので意味がないからです。いわゆる定数との主な相違点は、値が固定されているプロパティだけではなく、Antの実行中もしくはタスクの結果などから動的に生成されるものがある点です。静的な方のプロパティは、定数に限りなく近いものです。さらにビルドファイルからプロパティを

† 訳注："Don't Repeat Yourself!"（同じことを繰り返して記述しない！）ということ。

分離し、プロパティファイルを介して評価することもできます。それによりビルドファイルは、プロパティが直接見えなくなるのと引き換えに、ずっと読みやすくなります。

10.2.8 ターゲットのラッパービルドファイルへの移動

概要：継続的インテグレーション（CI：Continuous Integration）用のターゲットは、ビルドファイルから引き上げて、分離する

refactoring_before.xml

```
<target name="build">
        <!-- developer build-->
</target>
<target name="functest">
        <!-- functional tests-->
</target>

<target name="cruise" depends="update,build,tag"/>
<target name="functional_cruise" depends="update,build,functest,tag"/>
```

refactoring_after.xml

```
<target name="build">
    <!-- developer build-->
< /target>
<target name="functest">
    <!-- functional tests-->
</target>
```

ccbuild.xml

```
<project name="cruise" default="tag">

    <target name="tag" depends="build">
        <!-- code to tag the files you have checked out -->
    </target>

    <target name="build" depends="update">
        <ant buildfile="build.xml"/>
    </target>
```

```
    <target name="update" >
        <!-- code to update from your scm system-->
    </target>

</project>
```

CIは、ソフトウェア開発におけるインテグレーションの痛みを和らげてくれるテクニックです。開発者がソースコードの変更を構成管理リポジトリにコミットするたびに、CIサーバは最新バージョンのソースコードをチェックアウトし、コンパイル、テストしてくれます。ステータスの変更は、いろいろな方法でチームに通知されます（同じ場所で作業しているチームには、音声を使った警告が効果的です）。それは、ビルドを問題のない状態もしくはテストにパスした状態に保つための、強い文化的な圧力となるはずです。仮に、2人の開発者がコードの統合をしていなかったら、すぐにチーム全体が知ることになります。

CI関連のオペレーション（例えば、構成管理のタグ付け、更新など）は、ビルドと非常に密接に結びつくことが時々あります。理想的には、CIシステムは開発者とまったく同じビルドを実行するべきなので、ラッパー用Antファイルに拡張ターゲットを作り、そこからAntタスクを使って開発者のビルドを呼び出すようにします。同じコードベースに対してCI関連のビルドを数種類実行する場合は、それぞれを別のccbuildファイルで保守でき†、CIの仕組みすべてをそれらのファイルに置いて危害を受けないように守ることができます。

10.2.9 descriptionによるコメントの置き換え

概要：インラインのコメントではなく、description属性を使用する

refactoring_before.xml

```
<target name="distribute">
    <!-- copy the compiled classes -->
    <copy todir="${dist}">
        <fileset dir="${classes.dir}"/>
    </copy>
</target>
```

† 訳注：ccbuildは継続的インテグレーションビルドツールのCruiseControl（http://cruisecontrol.sourceforge.net/）から呼び出すbuildファイルに付ける一般的な名称。

refactoring_after.xml

```
<target name="dist">
    <copy todir="${dist}" description="copy compiled classes">
        <fileset dir="${classes.dir}"/>
    </copy>
</target>
```

たいていのAntビルドファイルは、コメントだらけになっています。コメントすること自体はよいことですが、ビルドのメカニズムを覆い隠してしまうものでもあります。ほぼすべてのタスクで、description属性を使用することができます。これでタスクの周辺にコメントを書くよりも、ダイレクトに注釈を付けることができます。ユーザに実行時に何が起きているか伝える手段として、taskname属性を利用することもできます。私はタスク名を短くするのが好みなので、長い説明はdescriptionに記述します。

Good likeness（そっくりさん）

とはいえ、デプロイの仕組みはなるべく（開発用のものと）そっくりにするべきでしょう。そのためには軽量コンテナ†をしっかりと選んでください。

10.2.10 デプロイ用コードのimport先への分離

概要：コードをデプロイするためのターゲットは、開発者が使うターゲットとは別のファイルに分離する

refactoring_before.xml

```
<target name="deploy_to_weblogic" >
    <!-- insert WL task or similar -->
    <sshexec host="${deploy.host}" username="dev" command="restart_container"/>
</target>
```

† 訳注：Spring等の軽量コンテナと標準的なServletコンテナの組み合わせなど、製品依存がなるべく少ない仕組みを構築する。

refactoring_after.xml

```
<import file="deploy.xml" />
<target name="test_in_container" depends="deploy_to_weblogic"/>
```

もし、あらゆるプロジェクトが1つのマシン上の1つのアプリケーションサーバしか使わないなら、1つのビルドファイルでシンプルになるでしょう。しかし、一般的には、プロジェクトはより多くのマシンとアプリケーションサーバを使います。例えば、開発者のマシンでは軽量サーバを、データセンターではエンタープライズ用のサーバを使うなどです。

関連するコードを抽出して別ファイルに分離すれば、うまく関心事の分離ができます。必要なファイルはどれでもimportできますし、デプロイに関する詳細はすべて隠されます。また、Antがビルドファイルをパースしたときに生成する、最終的なプロジェクトオブジェクトをとても簡素化できます。そのため、エンタープライズ用サーバにデプロイしなければいけないときに、ローカル用ビルドの依存関係にミスがあっても、事実、問題になりません。

10.2.11 要素のantlibへの移動

概要：複数のプロジェクトで何度も使われるAntの要素はantlibを使って配布する

ccbuild.xml

```
<project name="cruise" default="tag">

    <target name="tag" depends="build">
        <!-- code to tag the files you have checked out -->
    </target>

    <target name="build" depends="update">
        <ant buildfile="build.xml"/>
    </target>

    <target name="update" >
        <!-- code to update from your scm system-->
    </target>

</project>
```

```
<!-- END ccbuild  -->

<project  default="update" basedir="."
        xmlns:my="antlib:com.thoughtworks.monkeybook">
    <target name="update" depends="build">
        <my:svn_up/>
    </target>
</project>
<!-- END antlibccbuild -->
```

ccbuild.xml

```
<project  default="update" basedir="."
        xmlns:my="antlib:com.thoughtworks.monkeybook">
    <target name="update" depends="build">
        <my:svn_up/>
    </target>
</project>
<!-- END antlibccbuild  -->
```

antlib.xml

```
<antlib>
    <macrodef name="svn_up">
        <attribute name="svn.exe" default="/usr/bin/svn" />
        <sequential>
            <echo message="${basedir}" />
            <exec failonerror="true" executable="@{svn.exe}">
                <arg value="update" />
            </exec>
        </sequential>
    </macrodef>
</antlib>
<!-- END antlib-->
```

Antを使った複数のプロジェクトで、同じXMLコードを再三繰り返すことがあります。これはMavenプロジェクトが始まった要因の1つでもあります。

> 個々のビルドを超えた共通のテーマがあるのに、それぞれのコミュニティが独自のビルドシステムを作成し、プロジェクトをまたがるビルドロジックの再利用がまったく行われなかった [Casey]

それぞれのライブラリを何のために使うのか？

もしランタイム用、ビルド用、といったライブラリのためのサブディレクトリをそれぞれ作っていたら、これはよいスタート地点になります。ビルドやデプロイに何が必要かわかっていることは、リファクタリングをする上で非常に役に立ちます。

antlibは<antlib>という名前のルート要素を持ったXMLファイルです。このファイルがクラスパスにあり（おそらく$ANT_HOME/libディレクトリの中のJARファイルに含まれているでしょう）、ビルドファイルでXML名前空間を指定すると、定義された要素を直接利用することができます。実際の例を示します。

例えば、大規模なプロジェクトは、最終的にはCruiseControlによって作成される多数の小さなプロジェクトの集合体になるでしょう。それぞれ以下のようなことをしなければいけません。

- 構成管理リポジトリから最新の更新を取得する
- 開発者用ビルドを実施する
- ビルドに成功したらタグを付ける

それぞれのビルドのための短いビルドファイルがあります（おそらくcc-build.xmlのような名前で呼ばれます）。これらは開発者用のビルドを呼び出す前に動作します。antlibは、デフォルトのクラスパスを通じて、データ型、タスク、macrodefを公開できるようにしてくれます。この例のプロジェクトでは、SVNタスクやmacrodefを定義して、それを$ANT_HOME/libに置くだけで、誰でもどこからでも使えるようになっています。ただ、他のチームが使うためには、パッケージ化する作業をする必要はあります。

```
mkdir -p com/thoughtworks/monkeybook/
cp ~/workspace/monkeybook/content/antlib.xml com/thoughtworks/monkeybook/. jar
cvf antlib.jar com
cp /tmp/antlib.jar apache-ant-1.6.5/lib/.
```

一度JARファイルをプロジェクトのクラスパスに入れれば、先に示した例のようにmacrodefを使うことができます。

10.2.12 filesetによる多数のライブラリ定義の置き換え

概要：手作業でpathelementを指定して苦労するのではなく、pathの定義の子要素でfilesetを利用する

refactoring_before.xml

```
<path id="build.path">
    <pathelement location="${lib}/build/junit.jar"/>
    <pathelement location="${lib}/build/crimson.jar"/>
    <pathelement location="${lib}/build/emma.jar"/>
</path>
```

refactoring_after.xml

```
<path id="build.path">
    <fileset dir="${lib}/build" />
</path>
```

ほとんどのプロジェクトでは、ライブラリは構成管理ツールで管理されています。ライブラリが変更されるときに、それらへのリファレンスを更新するのは退屈な作業となります。この例ではパスにバージョン番号が含まれていませんが、あるリリースの時点でライブラリをアップグレードする必要があれば、ビルドファイルの修正が必要になります。Antがライブラリを発見するようにすれば、あなたの多くの仕事を肩代わりしてくれます。しかし注意してください。依然として、あなたはコードがどのようなライブラリを使っているか理解して、正しい方法で組み合わせる必要があります。

10.2.13 実行時プロパティの移動

概要：アプリケーションの再設定を簡単に行えるように、実行時のプロパティをビルドのプロパティと明確に区別する

refactoring_before.xml

```
<property name="runtime.smtp.server" value="foo.thoughtworks.com"/>
<property name="web.service.endpoint" value="bar.thoughtworks.com/axis"/>
<target name="war">
    <echoproperties destfile="${build}/war/lib/myapp.properties"/>
    <war destfile="${dist}/myapp.war" basedir="${build}/war"/>
</target>
```

refactoring_after.xml

```
<target name="war">
    <copy file="${src}/runtime.properties"
        tofile="${build}/war/lib/myapp.properties"/>
    <war destfile="${dist}/myapp.war" basedir="${build}/war"/>
</target>
```

異なる環境にデプロイするために、アプリケーションを再パッケージし、再コンパイルする必要があるでしょうか。リリース候補をコンパイルできるようにし、後の段階でデプロイするためにそれらをリポジトリに置くべきです。これは、いつでも、テスト済みのコードと同一のコードをデプロイしていることを保障します。実際にあった別の問題として、アプリケーションが実行時に利用するちょっとしたプ

設定情報を自由に変更できるようにする

ビルドに時間がかかるようになったり、リリースが迫ってきたりした場合には、コードから設定情報を分離することを検討しましょう。例えば、システム実行時に使用するプロパティファイルをファイルシステム上に配備することができます。必要があればこのファイルを編集できるので、この仕組みがうまく機能します。ただ本番環境ではプロパティファイルに対するセキュリティを確保する必要があります。LDAPはまた別の選択肢になります（それ以上のこともしてくれます†）。アプリケーションから利用する、LDAPとは別のサービスを導入してもよいでしょう。

† 訳注：LDAPのエントリに設定情報を格納する。

ロパティを変更するために、全体の再ビルドが必要になったことがありました。「Webサービスのエンドポイント URL が変わったって？ ビルドし直して」

ビルドファイルの中には大きく違う2種類のプロパティがあります。ビルド時（どこにコンパイルするか、どのリポジトリに発行するか）と実行時（データベース接続情報や外部サービス情報など）のプロパティです。これらは、簡単に混じって、混乱してしまいます。もちろん問題にならないときもあります。あなたが、環境が変化したときに複数のブランチの間を行ったり来たりしながらプロパティをマージしたり、たった1つのプロパティの変更を施したリリース候補を20分で自動機能テストしてパスさせたりする必要がないかぎり、ですが。

ソフトウェアプロジェクトのライフサイクルの間中まったく管理しなければ、プロパティをコントロールできなくなります。自分たちのためにも、デプロイのために必要とされるプロパティと、アプリケーションを走らせるときに必要とされるプロパティを別の入れ物に分離するようにしましょう。ランタイムプロパティを独自のリポジトリに移動して、コード、プロジェクト、チーム特有のビルドから独立させてください。

10.2.14 IDを用いた要素の再利用

概要：重複するpathのようなエレメントを単一エレメントへのリファレンスによって置き換える

refactoring_before.xml

```
<target name="copy_and_filter">
    <copy todir="${build}/content">
        <fileset dir="${html}" />
        <filterset>
            <filter token="AUTHOR" value="${author.name}" />
            <filter token="DATE" value="${timestamp}" />
            <filter token="COPYRIGHT" value="${copyright.txt}" />
        </filterset>
    </copy>
    <copy todir="${build}/abstracts">
        <fileset dir="${abstracts}" />
        <filterset>
            <filter token="AUTHOR" value="${author.name}" />
            <filter token="DATE" value="${timestamp}" />
            <filter token="COPYRIGHT" value="${copyright.txt}" />
        </filterset>
    </copy>
```

```
    </target>
```

refactoring_after.xml

```
    <filterset id="publishing_filters">
        <filter token="AUTHOR" value="${author.name}" />
        <filter token="DATE" value="${timestamp}" />
        <filter token="COPYRIGHT" value="${copyright.txt}" />
    </filterset>

    <target name="copy_and_filter">
        <copy todir="${build}/content">
            <fileset dir="${html.content}" />
            <filterset refid="publishing_filters"/>
        </copy>
        <copy todir="${build}/abstracts">
            <fileset dir="${abstracts}" />
            <filterset refid="publishing_filters"/>
        </copy>
    </target>
```

path や fileset のようなトップレベルの要素はリファレンスによって呼び出すことができます。重複するような path 定義の代わりに、一度だけ定義して ID を割り当てることによって、それを build.xml ファイルのどこからでも参照することができます。忙しいプロジェクトにおいて巨大な build.xml に対処するときには、これは非常に便利です。たくさんの行が積み重なったファイルは理解しづらいものです。そんな path や fileset の定義を消して 1 行に変えてしまうのは、やる気が出る作業でもあります。

誰かが、大きな build.xml で fileset のインスタンスをすべてアップデートし忘れた、といったバグを見つけることがあるかもしれません。

10.2.15 プロパティのターゲット外部への移動

概要：ターゲット内で宣言されているプロパティを、ビルドファイルのボディの直下に移動する

refactoring_before.xml

```
<target name="distribute">
    <property name="dist_file" value="widget-1.0.tar"/>
    <tar destfile="${dist_file}">
        <tarfileset dir="${build}/dist"/>
    </tar>
    <gzip src="${dist_file}"/>
    <scp file="${dist_file}.gz"
        todir="${appserver.userid}@${appserver.host}:/tmp"/>
 </target>
```

refactoring_after.xml

```
<property name="dist_file" value="widget-1.0.tar"/>
<target name="distribute">
    <tar destfile="${dist_file}">
        <tarfileset dir="${build}/dist"/>
    </tar>
    <gzip src="${dist_file}"/>
    <scp file="${dist_file}.gz"
        todir="${appserver.userid}@${appserver.host}:/tmp"/>
</target>
```

私たちの多くは、変数をローカルに定義するには、コードのブロックの中に定義すればよいと思っています。これはレキシカルスコープと呼ばれるものですが、Antには存在しません。プロパティの定義をすると、プロジェクトの名前空間内にあるどのターゲット、タスクからも使えるようになります。プロパティはそれを使うところで宣言するというのは、魅力的なアイデアなのですが、それでプロパティのスコープをターゲット内に限定できるとは思わないでください。

同じ名前のプロパティが複数あり、しかも、ターゲットの実行順序によってプロパティの値が変化する†場合に、同僚は本気で混乱し始めるでしょう。また、ant-contribライブラリ††を用いることで、プロパティの根本的な性質であるイミュー

† 訳注：プロパティは宣言したタイミングで設定されて、それ以降は同じ名前のプロパティ宣言が実行されても無視される。したがって、どの宣言が先に実行されるかによって、値が違ったものになる。

†† 訳注：http://ant-contrib.sourceforge.net/tasks/tasks/variable_task.htmlで、後から変更可能なプロパティとしてvariableというデータ構造が提供されている。

タブルを壊すことも可能です。しかしこれをしてしまうと、結果として、プロジェクトはビルドツールではなくスクリプト言語としてAntを使っていることになってしまいます。

10.2.16 locationによるvalue属性の置き換え

refactoring_before.xml

```
<property name="libdir" value="lib" />
<property name="libdir.runtime" value="${libdir}/runtime" />
```

refactoring_after.xml

```
<property name="libdir" location="lib" />
<property name="libdir.runtime" location="${libdir}/runtime" />
```

あなたのAntビルドは、特定のディレクトリや特定の環境変数の存在を前提として動作しているかもしれません。また、あなたはwikiか何かで、新人の開発者向けにセットアップ方法や環境の作成方法を公開しているかもしれません。

しかし理想的には、build.xmlがビルドに必要なものを自分で発見できない場合は、何が必要なのかをユーザに通知するべきです。property要素でlocation属性を利用することで、頑強なビルドシステムに向かっての最初の一歩を踏み出すことができます。開発者がスクリプトやツールを誤用することがあります。ユーザを原因とする不具合が発生した場合、一般的な対処法はすみやかに適切なメッセージとともにプログラムを終了することです。間違ったディレクトリ（デフォルトではbuild.xmlがあるディレクトリが${basedir}として設定されます）で動かしたとき、Antもこのような正しい動作をします。Antはlocation属性に${basedir}ディレクトリからの相対パスを設定してプロパティを構成しますが、動作するときは相対パスのままではなく、build.xmlファイルが存在するディレクトリから正しい絶対パスを割り出して設定し直します。多くのビルドはvalue属性を利用しているため、脆いものになっています†。

location属性は他の要素、特にexecuteなどのタスクに渡されるarg要素で利用

† 訳注：-dオプションを付けてAntを実行すると、location属性に書いた相対パスが、実行時には絶対パスとして再設定される様子がわかる。

可能です。タスクに正確なパスを渡したい場合に、同じように動作します。

10.2.17 build.xml内へのラッパースクリプトの取り込み

概要：スクリプトでパスやオプションを設定している部分をbuild.xml内に入れる

go.bat

```
rem START:push_down_wrappers
@echo off

set CLASSPATH=%CLASSPATH%;lib\crimson.jar
set CLASSPATH=%CLASSPATH%;lib\jaxp.jar
set CLASSPATH=%CLASSPATH%;lib\ojdbc14.jar

cd build
ant -f build.xml
rem END push_down_wrappers
```

refactoring_after.xml

```
<classpath id="classpath" description="The default classpath.">
    <pathelement path="${classpath}"/>
    <fileset dir="lib">
        <include name="jaxp.jar"/>
        <include name="crimson.jar"/>
        <include name="ojdbc14.jar"/>
    </fileset>
</classpath>
```

多くのプロジェクトでは、結局Antビルドファイルをラップする DOS のバッチファイルや Unix シェル、Perl スクリプトなどを作ることになります。これらは普通、プロジェクト固有の情報やいくつかのオプションを含んでいます。複雑なプロジェクトのビルドでも、誰もがコードをビルドできるように、使いやすいフロントエンドを提供するためです。また、さらに他のスクリプトでラップされることもあります。これらはデバッグしにくくいらだたしいものです。そのため、可能なかぎり使用を避けるか、必要であってもワンライナー程度に収めるべきです。また、スクリプトによるラッピングは、プロセスの戻り値が返ってこないという意図しない結果を招く可能性があります。そのため、自動デプロイプロセスがAntスクリプト

を呼び出し、その後に他の処理を実行する場合に、何か問題を隠してしまうこともあります。

ビルドシステムを頑強なものにするため、失敗する要素を事前にチェックし、正しくプロパティが設定されていることを確認できます。ビルドが失敗したときに、available タスク†を使うことにより、ファイル、classpath の内容、JVM のリソースが Ant プロセスから利用可能であるということを、プロパティを通じて検知することができます。また、コマンドラインで -D オプションを使って、必要とするプロパティを作成することもできます。

もし、コマンドラインのタイピングにうんざりしたり、チームのメンバからクレームが出たりしたら、go.bat のようなラッパースクリプトを作成します。しかし、これは 1 行か 2 行に保つべきです。OS のシェルから Ant を起動する程度で十分です。例外的に Windows では、ビルドの classpath を汚染しないように CLASSPATH 環境変数の消去が必要になるかもしれません。Windows インストーラパッケージの中には CLASSPATH 環境変数に内容を追加するものがあり、一部のマシンで予期しない結果となることがあるからです。

しかしときには、Ant のタスクを実行するためのライブラリを使えるようにする必要があるでしょう。Ant のマニュアルには必要な依存関係がリストアップされています。

このライブラリの依存関係を解決するために、ライブラリのリストを classpath に追加するラッパースクリプトを使うか、もしくはライブラリを Ant の lib ディレクトリに置くことができます。Ant が動作するときは、ランチャー用クラスが呼び出されて、見つかったライブラリからクラスパスを作成します。これは、ライブラリの依存関係を解決するには、とても便利な方法です。Ant 自体を構成管理ツールにチェックインするのもよいアイデアです。開発者は自分のマシンをプロジェクト用にセットアップしなくても、チェックアウトしてビルドを実行することができます。

10.2.18 taskname 属性の追加

概要：Ant に意味のある出力をさせることにより、タスクのコードの意図を理解できるようにする

† 訳注：available タスクはリソースが利用可能だった場合に指定したプロパティに値（デフォルトは true）を設定する（http://www.jajakarta.org/ant/ant-1.6.1/docs/ja/manual/CoreTasks/available.html）。

Antの呼び出し

Ant自身を実行するスクリプトでビルドが失敗したときに、Unixシステムではゼロ以外のコードを返します。Windows上では、同じ動作を期待しないでください。いくつかのバージョンのAntではant.batファイルがWindowsにおけるエラー値を返さないからです。いくつかの動的言語でもAntをラップするものがあります。これらに対しては、ビルドが失敗したときにデプロイスクリプトが動き続けないことを、しっかりテストして確認したほうがよいでしょう。

refactoring_before.xml

```
<target name="copy_config">
    <copy tofile="${output}/style.xsl" file="${src}/xsl/style.xsl" />
    <copy todir="${output}">
        <fileset dir="${xml.docs}"/>
    </copy>
    <copy todir="${output}/images">
        <fileset dir="${common}/images"/>
    </copy>
</target>
```

refactoring_after.xml

```
<target name="copy_config">
    <copy tofile="${output}/style.xsl"
        file="${src}/xsl/style.xsl" taskname="copy xsl stylesheet"/>
    <copy todir="${output}" taskname="copy xml docs to output">
        <fileset dir="${xml.docs}"/>
    </copy>
    <copy todir="${output}/images" taskname="copy images to output">
        <fileset dir="${common}/images"/>
    </copy>
</target>
```

ビルドファイルを書いていると、同じタスクがずらっと並んでしまうことがあります。よい例はcopyタスクです。これはビルドを走らせているユーザにとって、この状況でビルドが何をしているか把握するのを難しくする原因になります。tasknameプロパティをタスクに追加することは、そこで何を実行しようとしているかを明

確にする上で、大きな助けになります。

10.2.19 内部ターゲットの強制

概要：内部ターゲットの最初にハイフンを1文字加えることにより、コマンドラインから呼べないようにする

refactoring_before.xml

```
<target name="init">
    <mkdir dir="build"/>
</target>
```

refactoring_after.xml

```
<target name="-init">
    <mkdir dir="build"/>
</target>
```

誤ったターゲットが呼び出された場合、ビルドに思わぬ結果を招きます。特に、コマンドラインから呼び出されることを意図していなかったターゲットが呼び出された場合です。Javaのprivateメソッドのようにtaskをプライベートで宣言する方法が存在しないために、このようなことが起こるのです。しかし、誰かが、Antを騙してターゲット名をコマンドラインオプションに見せかけるという、シンプルですが賢いアイデアを思いつきました。シェルはAntラッパースクリプトにパラメータを引き渡します。そこでAntはパラメータを解析するのですが、先頭にハイフンが付いたパタメータはオプションだと解釈します。つまり、-create_databaseのようなタスクのことを、(存在しないものでも) オプションとして解釈するため、コマンドラインから実行する手段がありません。しかし、-propertyfileや-logger†のような内部ターゲットを宣言してしまうと、混乱することがあるかもしれません。

10.2.20 出力ディレクトリの親ディレクトリへの移動

概要：ビルドによっていろいろな場所に作成されるコンポーネントを1つのディレクトリに集約する

† 訳注：Antの実行時に有効なコマンドラインオプションと同じ名前。

```
project/
|-- build
|-- dist
|-- docs
|-- src
'-- testresults
```

```
project/
|-- build
|   |-- dist
|   |-- docs
|   '-- testresults
'-- src
```

複数のディレクトリがあって、それらがコードにより動的にアップデートされていると、混乱が生じます。最初からビルドをやり直すためには、いくつかのディレクトリを掃除する必要がありますが、そのとき別のディレクトリに退避するものを慎重に選ぶ必要があり、とても面倒なことになってしまうでしょう。生成先のディレクトリを1つ作り、確実に構成管理ツールの管理外にし、そして生成されるものすべてをそこに移動してください。プロパティファイルが正しい場所を示していれば、このディレクトリの変更は1行修正するだけで済みます。

10.2.21 applyによるexecの置き換え

refactoring_before.xml

```
<exec executable="md5sum" output="md5sums.txt">
    <arg value="${dist.dir}/foo.dll"/>
    <arg value="${dist.dir}/bar.dll"/>
</exec>
```

refactoring_after.xml

```
<apply executable="md5sum" output="md5sums.txt">
    <fileset dir="${dist.dir}" includes="*.dll"/>
</apply>
```

execは利用できる引数のセットに制限があり、まさに単純なコマンド向けのものです。applyはAntのデータ型を受け入れることができます。それはfilesetを引き渡したり、refidと一緒に渡したり、といったことができることを意味します。よりわかりやすくしていきましょう。

10.2.22 CI Publisherの利用

概要：ビルドを壊す誤ったタグ付けをしない。それは開発者へのフィードバックを妨げる

refactoring_before.xml

```
<target name="cruisecontrol" depends="developer_build, functional_tests, tag">
```

refactoring_after.xml

```
<target name="cruisecontrol" depends="developer_build, functional_tests">

<target name="tag"
    description="this will fail unless run from a cruise publisher">
    <fail unless="logdir"
        message="${logdir} property missing -
        are you running this from a cruisecontrol publisher"/>
</target>
```

10.2.23 明確なターゲット名の導入

概要：ターゲットとプロパティに対して同じ命名規則を使わない

refactoring_before.xml

```
<property name="build.dir" location="${basedir}/build"/>
<property name="lib.dir" location="${basedir}/build"/>
<target name="test.unit" >
    <junit haltonerror="false" haltonfailure="false">
        <!-- details excluded -->
     </junit>
</target>
```

refactoring_after.xml

```
<property name="build.dir" location="${basedir}/build"/>
<property name="lib.dir" location="${basedir}/build"/>
<target name="unit-test" >
    <junit haltonerror="false" haltonfailure="false">
        <!-- details excluded -->
    </junit>
</target>
```

> ... Antが持つ性質か何かのせいで、人はビルド対象のコードとはまったく違った形でAntを扱う。一貫性、テスト、保守性、そして良識といったものに関するルールが、すっかり消えてなくなるように思える。[Newman]

XMLは必ずしも読みやすいものでありません。build.xmlを読みやすくに保つには、大変な労力が必要です。実際、これがbuild.xml向けのスタイルに執着すべき理由になります。読み手が間違いやすいスタイルとして、ターゲットもプロパティも同じように単語をドットで分割するというものがあります。属性がプロパティを参照しているか、値を参照しているかが、はっきりしないことがあります。プロパティの単語を分けるにはドットを使いましょう。これはJavaのプロパティがドットを使っていて、Antのプロジェクトでも Javaのシステムプロパティの名前空間を取り込んでいるからです。ターゲットの名前は、アンダースコアもしくはダッシュ記号で分割すればよいでしょう。より詳しくは "The Elements of Ant Style"[†]を参考にしてください。ただし、いくつかのアドバイスはもはや必要ではありません。例えば最近のIDEでは、ターゲットを特定するために、コメントのブロックなどを使わずに、ナビゲートしてくれます。また、例えばGrandなどのツールは、ビルドの依存関係をツリー表示してくれます（「10.5 リソース」を参照してください）。

10.2.24 ターゲット名の名詞への変更

概要：ターゲット名は、プロセスではなく、ターゲットが作成する成果物の名前を付ける

† http://wiki.apache.org/ant/TheElementsOfAntStyle

refactoring_before.xml

```
<target name="foo-build-webapp">
    <war destfile="foo.war">
        <fileset dir="${build.dir}/frontend/dist"/>
    </war>
</target>
```

refactoring_after.xml

```
<target name="foo.war">
    <war destfile="foo.war">
        <fileset dir="${build.dir}/frontend/dist"/>
    </war>
</target>
```

> 何を作るかを表現した名前を選ぶことをお勧めします。例えば、classes、test もしくは report です。[Williams]

Ant の build.xml のイディオムとしては「compile」や「test」といったターゲットがあります。これは疑う余地のないものです。中にはビルドの状態を表現していて、その状態のときに実行するターゲットもあります。このようなターゲットは、ready to deply、ready to publish、またはそれに類似したものになるでしょう。成果物を生成するターゲットでは、その成果物に注目することができます。実行する必要がないターゲットをスキップするために uptodate タスク†を使って、皆の時間を節約できます。また、成果物を名前にすることで、ビルドがとても明確になります。

make やその親戚を知っていれば、このような動作を見たことがあるかもしれません。一般的な make のスタイルでは、ターゲットの名前を作成する成果物から付けます。これは使いやすいのですが、残念ながらプラットフォーム独立性を失います††。

---

† 訳注:uptodateはターゲットファイル（成果物）とソースファイル（成果物の基になるもの）、ファイルセットの最終更新日を比較し、ターゲットファイルよりもソースが新しい場合のみプロパティを設定する。このプロパティでtargetの実行可否を制御することができる。

†† 訳注：「foo.exe」や「bar.so」といったプラットフォーム依存のファイル名をターゲット名に用いた場合、ビルドのプラットフォーム独立性をなくす可能性がある。

10.3 まとめ

このエッセイでは、Antビルドファイルのリファクタリングを紹介しました。いくつかのリファクタリングは、リファクタリングについて初めて書かれた書籍[FBB+99]から抜き出してAntビルドファイル用に変換したものです。その他のリファクタリングは、開発現場から得たものです。既存のビルドシステムを変更しなければいけない状況に直面したとき、それに取り掛かるのはつらく厳しいことかもしれません。しかし、努力は報われるものです。顧客に提供するためにソフトウェアをビルドするという事実を見失わないでください。常にその認識の下でビルドの重要性を評価していれば、きっと素敵な週末を過ごせるでしょう。

10.4 参考文献

[Hunt, Thomas] Andrew Hunt, David Thomas: "*The Pragmatic Programmer*", Addison-Wesley, 1999.
(邦訳は『達人プログラマー — システム開発の職人から名匠への道』村上 雅章訳、ピアソン・エデュケーション)

[Newman] http://www.magpiebrain.com/blog/2004/12/15/ant-and-the-use-of-the-full-stop/

[Casey] Better Builds with Maven (http://code.google.com/p/opencookbook/downloads/detail?name=BetterBuildsWithMaven.pdf&can=2&q=)

[Fowler, Foemmel] http://martinfowler.com/articles/continuousIntegration.html

[Loughran, Hatcher] Steve Loughran, Erik Hatcher: "*Java Development with Ant*", Manning Publications, 2002.

[Williams] http://dogbiscuit.org/mdub/weblog/Tech/Programming/Java/AntBuildTips

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts: "*Refactoring: Improving the Design of Existing Code*", Addison Wesley Longman, 1999.
(邦訳は『リファクタリング — プログラミングの体質改善テクニック』児玉 公信+友野 晶夫+平澤 章+梅澤 真史 翻訳、ピアソン・エデュケーション)

10.5 リソース

- Grand (http://www.ggtools.net/grand)
- The Elements of Ant Style (http://wiki.apache.org/ant/TheElementsOfAntStyle)

11章
1クリックデプロイ

Dave Farley◎テクノロジプリンシパル

11.1 継続的ビルド

アジャイル開発プロジェクトのコアプラクティスの1つは、継続的インテグレーション（CI：Continuous Integration）です。CIは、ビルドと大規模な自動テストスイートからなるプロセスです。ビルドと自動テストスイートは、ソフトウェアをバージョン管理システムにコミットするときに実行されます。

CIは何年にもわたってプロジェクトで実践されてきました。このプラクティスは、開発中のソフトウェアがどの時点でもビルドに成功し、単体テストスイートをパスするという、大きな安心を提供します。これにより、ソフトウェアが目的のサービスを提供するということがより確実になります。多くの（ほとんどのという人もいます）プロジェクトにとって、これは最終的に提供されるソフトウェアの品質と信頼性における大きな一歩です。

しかし、複雑なプロジェクトでは、コードがコンパイルされ単体テストをパスしても、まだ問題が生じる可能性があります。

単体テストのカバレッジがどれほど高くても、それは、単体テストの性格上、要求を満たしたというよりもコーディングが終わったという意味に近くなります。状況によっては、単体テストは、そのコードが要求を満たしていることではなく、開発チームが想定したとおりになっていることしか証明しません。

コードを作り上げたら、それをデプロイしなければいけません。複数チームで開発するほとんどの現代的なソフトウェアでは、デプロイは1つのバイナリファイルをファイルシステムへコピーするような単純なものではありません。むしろ、ソフトウェアそれ自体に加えて、Webサーバ、データベース、アプリケーションサーバ、

キュー†といった技術要素をまとめてデプロイし、設定する場合がほとんどです。

通常、このようなソフトウェアは、本番稼動に至るまでにさまざまな環境にデプロイされる、とても複雑なリリースプロセスを経なければいけません。ソフトウェアは、最終的に本番稼動にたどり着くまでに、開発マシン、品質保証（QA）環境、性能テスト環境、そしてステージング環境にデプロイされるでしょう。

ほとんどのプロジェクトでは、これらのステップのすべてではなくても大部分で、大量の手作業があるでしょう。人が手作業で設定ファイルを管理し、デプロイ先の環境に合わせて手作業でさまざまな調整をしているでしょう。そして、いつも何かを忘れます。「テンプレートファイルの保存場所が開発環境と本番環境とで違うとわかるのに2時間かかったよ。」

「継続的インテグレーション」はよいものですが、今最も一般的に行われているものをそう呼ぶのは誤りです。それは「継続的ビルド」と呼ぶべきものです。それが真にリリースプロセス全体に適用されたらどうなるでしょうか。それが真にシステム全体にわたる継続的インテグレーションであったならどうなるでしょうか。

11.2 継続的ビルドを越えて

筆者が働いているチームは、過去2、3年の間ずっとこのことを重く受け止めて、エンドツーエンド††のCIリリースシステムを構築してきました。このシステムにより、大きく複雑なアプリケーションを任意の環境に、ボタンのクリック1つでデプロイできます。このアプローチの結果、リリース時のストレスが激減し、問題の発生がとても少なくなりました。また、エンドツーエンドのCI環境を確立する過程で、筆者らはとても汎用的な抽象ビルドプロセスを発見しました。この抽象ビルドプロセスによって、プロジェクトの開始時から本格的に仕事に取り掛かることができ、とても洗練されたビルドシステムを速やかに構築できます。

そのプロセスは、開発中のソフトウェアのリリース候補が一連のゲートを通過しながら進歩していくというアイデアに基づいています。各ゲートを通過するとき、リリース候補の信頼が高まります。このアプローチの目的は、本番稼動へ向けたリリースの準備が整っていると証明されるレベルにまで、リリース候補の信頼のレベルを高めることです。アジャイル開発の原則どおり、各チェックインでこのプロセスが開始され、各チェックインで（問題がないなら）それ自体が実行可能なリリー

† 訳注：メッセージキューのこと。
†† 訳注：1章参照。

ス候補となります。

リリース候補がプロセスを進むときに通過するゲートには、ほとんどのプロジェクトで必須のものもありますし、個別のプロジェクトのニーズを満たすよう調整されるものもあります。

筆者らは普通、このプロセス ——ゲートの連続のこと—— を、**ビルドパイプライン**、**パイプライン化ビルド**、もしくは**継続的インテグレーションパイプライン**と呼びます[†]。また、**段階的ビルド**と呼んだこともあります。

11.3 ライフサイクル全体の継続的インテグレーション

図 11-1 では、一般的なライフサイクルの全体像を見ることができます。これが CI パイプラインで、このアプローチの本質を表しています。多くの異なるプロジェクトでの経験を通じて、このプロセスはとても汎用的であるとわかりました。とはいえ、すべてのアジャイルプラクティスがそうであるように、当然、個別のプロジェクトのニーズに合わせて調整、変更されます。

このプロセスは、開発者がソースコードリポジトリに変更をコミットすることから始まります。変更をコミットすると、CI システム（通常、筆者らのプロジェクトでは CruiseControl[††]）が CI プロセスを起動します。このとき、CI プロセスによってコードがコンパイルされ、コミットテスト[‡]が実行されます。そして、それらをすべてパスしたら、コンパイル済みコードのアセンブリ[‡‡]が作成され、記憶装置の管理領域にこれらのアセンブリがコミットされます。

11.3.1 バイナリの管理

このアプローチの基本は、フルリリースまで順にプロセスの各ステップを実行し、リリース候補を進めていくことです。そうする主な理由の 1 つは、誤りがプロセスの中に紛れ込む機会を最小にすることです。

---

† 訳注：CPU のパイプライン処理をメタファにしている。ビルドプロセスを複数のステージに分割し、それぞれを独立して並行に実行できるようにしているところは、まさにパイプラインである。

†† 訳注：http://cruisecontrol.sourceforge.net/index.html

‡ 訳注：著者は、コミット時に実行されるテストを総称してコミットテストと呼んでいる。

‡‡ アセンブリはコンパイル済みコードをグルーピングしたもので、いくつかの種類があります。例えば、.NET アセンブリ、JAR ファイル、WAR ファイル、EAR ファイルです。大切なことは、設定に関する情報がこれらのアセンブリに含まれないことです。バイナリは、変更することなくどの環境でも実行可能であるべきです。

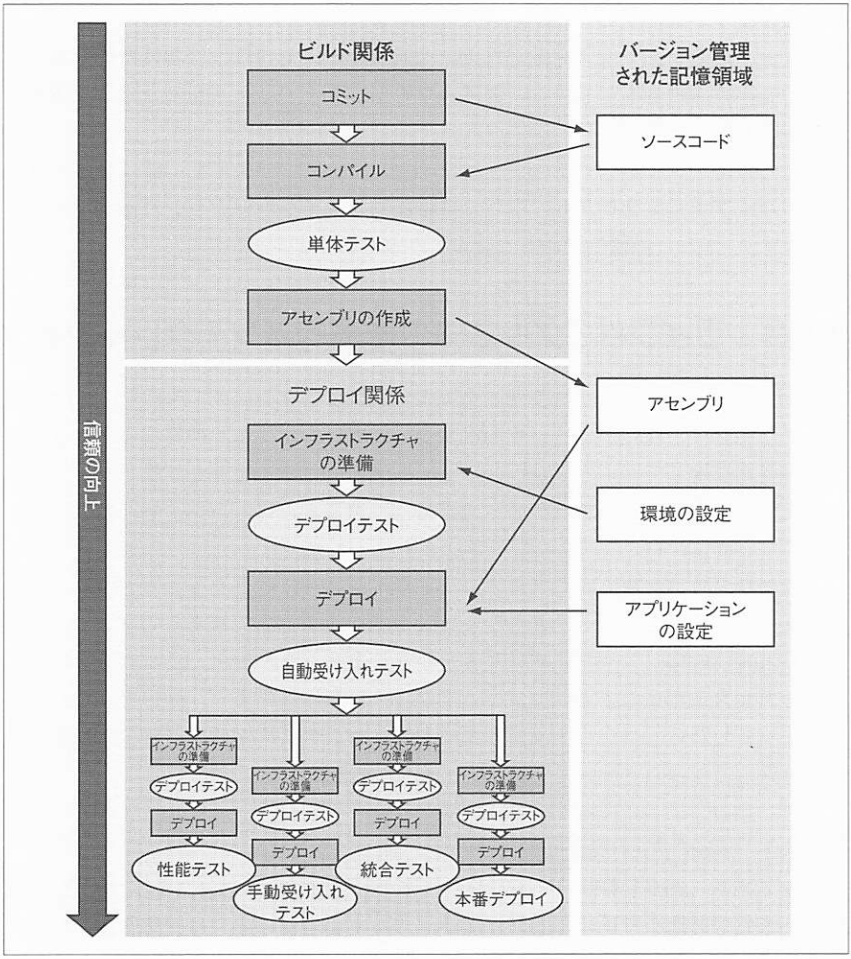

図 11-1 継続的インテグレーションパイプライン

例えば、リポジトリにソースコードのみを保存した場合、デプロイの必要があるたびにそのソースコードを再コンパイルしなければいけません。そのため、仮に、性能テストのためにまずソースコードの再コンパイルが必要な状況であれば、性能テスト環境の何かが違っているというリスクを冒します。おそらく、うっかりして異なるバージョンのコンパイラを使ったことや、異なるバージョンのライブラリとリンクしたことがあるでしょう。ビルドパイプラインのコミットテストや機能テストの段階で発見すべき誤りが不注意で紛れ込む可能性は、できるかぎり取り除かな

ければいけません。

プロセス内での作業のやり直しを防ぐという考えを貫き通すことで、次のような副次的な効果が生じます。プロセスの各ステップで用いるスクリプトがとてもシンプルに保たれやすくなり、また、環境固有のものと環境に依存しないものがきれいに分離されるようになります†。

しかし、このアプローチでバイナリを管理するときには注意が必要です。ごく稀にしか使用しないバイナリで、大量の記憶領域をすぐに消費してしまいます。そのオーバーヘッドに見合うだけのメリットはないため、たいていは妥協して、バイナリをバージョン管理システムに保存するのを避けます。代わりに、筆者らは共有ファイルシステムの専用領域を使っています。この専用領域はローリングバイナリリポジトリ††として管理され、バージョン名を付与した圧縮イメージにバイナリをアーカイブします。これまでのところ、筆者らはこのリポジトリを使うためのスクリプトを自作しており、これ以外のすぐに使える適当な代替手段をまだ見つけていません。

バイナリのイメージは、それの元となったソースコードのバージョン情報でタグ付けされます。ビルドタグをリリース候補の識別子とするのが一番楽です。すべてのソースコード、バイナリ、設定情報、スクリプト、その他システムのビルドとデプロイに要するものが互いに関連していることを示すために、この識別子が使われます。

このような方法で「管理されたバイナリ」の集まりは、最近の数回のビルドをキャッシュしたものにあたります。ある期間が過ぎると、筆者らは古いバイナリを削除します。削除してしまったバージョンに戻す必要があると後になって決めた場合は、バージョン管理システムに安全に保管されているソースコードのタグを取り出し、ビルドパイプライン全体を再実行しなければいけません。しかし、これはとても稀なことです。

11.4 チェックインゲート

自動ビルドパイプラインはリリース候補の作成によって開始されます。リリース候補は、開発者がバージョン管理システムに変更をコミットしたときに暗黙的に作

† このプロセスにより、デプロイ先の環境に特化したバイナリの作成は暗に抑制されます。そのようなデプロイ先に特化したバイナリは柔軟なデプロイとは正反対のものですが、それにもかかわらず企業システムでは一般的に見られます。

†† 訳注：ローリングとは一定量やある期間で中身を順次入れ替えること（ログのローリング処理などがある）。その方法でバイナリファイルを管理するリポジトリを指して、筆者はローリングバイナリリポジトリと呼んでいる。

成されます。

このとき、ソースコードがコンパイルされ、一連のコミットテストが実行されます。コミットテストには、普通すべての単体テストが含まれます。加えて、選抜した少数のスモークテスト†やその他のテストが含まれます。これらのテストは、チェックインにより作成されるリリース候補が実際に実行可能であり、次の段階に進んでよいと実証するものです。

これらのコミットテストの目的はフェイルファスト††です。チェックインゲートは対話式で、開発者はコミットテストをパスするのを待ってから次のタスクに取り掛かります。そのため、コミットテストに時間がかかってしまうと開発プロセスの効率を保てません。

一方で、失敗がCIパイプラインの後になればなるほど、その修正のコストは高くなります。そのため、必須である単体テスト以外のテストもコミットテストスイートへ追加しますが、その選択を慎重に行うことがチームの効率を保つ上でたいてい重要となります。

コミットテストをすべてパスした時点で、チェックインゲートを通過したと見なします。開発者は自由に他のタスクに取り掛かれるようになります。この際、ビルドパイプラインのこれより後の段階をまだ実行していなくてもかまいません。テストをパスしていなくてもよいのは言うまでもありません。

これはプロセスレベルの最適化に他なりません‡。すべての受け入れテスト、性能テスト、統合テストを瞬時に実行できる理想の世界では、CIプロセスをパイプラインで実行していく利点はありません。しかし、現実においては、これらのテストの実行にはいつも時間がかかります。そのため、すべてのテストが成功するまで作業を進められなければ、開発チームの生産性が大幅に下がってしまいます。

コミットテストをチェックインゲートとすることで、チームは自由に新しいタスクに取り掛かれるようになります。しかし、チームは、チェックイン以降のリリース候補のテスト結果を、残りのライフサイクルを通じて念入りにモニタリングしな

† 訳注：コードに加えた変更が開発者の想定どおりであり、システム全体を不安定にしないことを確認するテストのこと。由来はハードウェアの通電テストで、回路が火を噴いて煙を出さなければ合格とすることからこの名前で呼ばれる。

†† 訳注：可能なかぎり早く失敗すること。問題を早期発見し、早期に対応する。

‡ 訳注：CPUのパイプライン処理では、ある命令の実行がすべてのステップを完了するのを待つことなく、次々に命令を並列実行していくことで全体での処理速度を向上させている。それと同様、あるストーリーがすべてのテストをパスするのを待つことなく、次々に並行してテストを進めていくことでプロセス全体での生産性を向上させている。また、コミットテストをパスした時点で開発者が別のタスクに取り掛かれるようにすることで、無駄な待ち時間をなくしている。

ければいけません。チェックインゲートの目的は、長い時間がかかるテストと並行して開発者が他の仕事をどんどん進められるようにしながらも、チームができるかぎりすばやく誤りを見つけてそれらを修正できるようにすることです。

このアプローチは、コミットテストのカバレッジがほとんどの誤りを見つけるのに十分なときにのみ受け入れられます。もしほとんどの誤りがパイプラインの後の段階で発見されるのであれば、それは、コミットテストの質を高めるときが来た、というよい警告です。

開発者ならばいつでも、コミットに要する時間を最短にしたいと言うでしょう。実際にはこのニーズは、チェックインゲートの、最もありふれた誤りを特定する能力とバランスを取る必要があります。これは試行錯誤を通じてのみうまくいく最適化のプロセスです。

コミットテストスイートの設計は、すべての単体テストの実行から始めてください。その後、パイプラインの後の段階でよくある失敗を検出するテストを個別に追加してください。

11.5 受け入れテストゲート

単体テストはどのアジャイル開発プロセスでも必要不可欠なものですが、単体テストだけではまったく不十分です。アプリケーションがすべての単体テストをパスしても、そのアプリケーションが対象とするビジネスの要求を満たしていないことは、珍しくありません。

単体テストと少しテストを追加したコミットテストに加えて、筆者のチームは自動受け入れテストをとても重視します。このテストは開発するストーリー†の受け入れ基準となり、コードが受け入れ基準を満たすことを証明します。

受け入れテストは機能テストでもあります。つまり、システムの端から端までテストします。ただ、たいていは自分たちの管理下にない外部システムとの相互作用の部分を除きます。このような場合、受け入れテストスイートの目的に合うよう、システム外部の接続ポイントのスタブを用意するのが一般的です。

筆者らは、受け入れテストの作成と保守を開発プロセスの中心に据えるのを好みます。開発者によって作成され、受け入れテストスイートに追加された自動テストスイートで受け入れ基準がテストされるまでは、ストーリーが完成したとは見なし

† 訳注：ストーリーとは、エクストリームプログラミング（XP）における要求定義の手法。XPでは、ユースケースの代わりにストーリーが用いられる。日本ではユースケースが主流だが、海外のアジャイル開発者の間ではストーリーのほうがよく使われているようである。

ません。筆者らは、技術に明るくない人でもすらすら読めるテストになるよう気を配ります。しかし、読みやすさの確保はここで議論しているCIプロセスの要点ではないでしょうから、このエッセイの範囲外とします。

受け入れテストは、専用の管理環境で実行され、CI管理システム（通常はCruiseControl）によってモニタされます。

受け入れテストゲートは、リリース候補のライフサイクルにおける2つめのキーポイントです。自動デプロイシステムは、すべての受け入れテストをパスしたリリース候補に限ってデプロイします。つまり、すべての受け入れ基準が満たされることなしに、リリース候補がこの段階を越えて本番稼動に進むことは不可能だということです。

11.6 デプロイの準備

状況によっては、アプリケーションと関連するものすべてのデプロイを自動化するのもうなずけるのですが、これは大規模なエンタープライズアプリケーションでは稀なことです。しかしながら、もしインフラストラクチャ全体の管理と設定を自動化できれば、多くの誤りが取り除かれるでしょう。特に、誤りの要因となっているのは、エンタープライズシステムの規模でよく見られる手動でのデプロイと設定です。このような状況ですので、自動化の試みには価値があり、たとえ部分的にしか成功しなくても、よくある誤りの多くの原因をたいていは取り除けるでしょう。

筆者らはこの問題に実用的なアプローチを採用していて、アプリケーションサーバ、メッセージブローカ、データベースなどからなる標準構成のサーバのイメージをよく用います。これらのイメージは、基本の設定でインストールし、設定したデプロイ済みシステムのある種の「スナップショット」です。

そのようなイメージは、プロジェクトが必要とするか使いやすいものであれば、さまざまな形式にすることができます。筆者らはよく、初期スキーマを構築するデータベーススクリプトと、そこに投入するデータのダンプを用意します。標準構成のインストール済みOSやアプリケーションサーバを用意することもあります。これらは、デプロイ先と決めたどのサーバででも、その環境構築プロセス†の一環として用いることができます。それはファイルシステムへのフォルダツリーのコ

† 訳注：OS、基盤となるサポートソフトウェア（Java実行環境、RDBMS、Webサーバなど）、アプリケーションのコンポーネントなど、システムが稼動するために必要なソフトウェア群のインストールと設定、テストを行うこと。

ピーと同じくらい単純なので、常に決まった位置に同じ構造が作られます。

「イメージ」がどのような性質のものであっても、差分をまとめて適用することで後の変更を行えるよう、共通のベースライン構成を構築するのがその目的です。

筆者らはよく、ヒューマンエラーが入り込む余地を極力減らし、新しい環境をすばやくセットアップできるよう、そのようなイメージをひとまとまりにして保守します。

ソフトウェアのデプロイごとに、このようなまっさらなインフラストラクチャをデプロイするわけではありません。ほとんどの場合、それはある新しい環境を構築するときにデプロイされ、以後めったにデプロイされることはありません。

一方で、アプリケーションをデプロイするときには、毎回、できるかぎり基本の状態に近づけるためにインフラストラクチャを再設定します。デプロイの残りの作業を、問題がないとわかっている状態から始めるためです。

ベースラインのインフラストラクチャの準備ができたら、簡単なデプロイテストスイートが役に立つでしょう。デプロイテストは、基本的なインフラストラクチャが整っていること、アプリケーションの個別のニーズと特定の環境に合わせて設定する準備ができていることを確認します。通常、これらのテストはとても単純で、基本的な配管工事が適切になされているかのアサーションとなります。例えば、DBMSが稼動していることやWebサーバがリクエストに応答することの保証です。

もしこのテストが失敗したら、イメージもしくはハードウェアに何か問題があるとわかります。

このテストをパスしたら、アプリケーションをデプロイする準備が整っているとわかります。プロセスのコミット段階で作成、テストされたアセンブリを管理領域から正しい場所へコピーするため、アプリケーション特化のデプロイスクリプトを実行します。

筆者らのスクリプトは、バイナリの単純なコピーに加え、アプリケーションサーバやWebサーバの起動と停止、データベースの適切なスキーマ定義やデータ更新、時にはメッセージブローカの設定なども必要に応じて行います。

次に示すように、本来、デプロイは5段階のプロセスですが、個々のデプロイは（最初の1つを除いた）4段階のプロセスとなります。

- 使用可能なサーバにサードパーティ†のインフラストラクチャを（たいていはイメージから）インストールする。**新しいサーバの環境構築時にのみこれを実行する。**

† 訳注：自分たちが開発しているソフトウェア以外のソフトウェアを指す。

- 問題がないとわかっている初期状態にまでインフラストラクチャを整理する
- デプロイテストを実行し、ソフトウェアのためにインフラストラクチャの準備が整っていることを確認する
- アプリケーションアセンブリをデプロイする
- アプリケーションのニーズに合わせてインフラストラクチャを設定する

筆者らは、ソフトウェアを適切に分割するのと同様に、ビルドやデプロイのスクリプトをシンプルにするために小さく分割します。それぞれのスクリプトは特定のタスクにフォーカスされ、その入力は可能なかぎり明確に定義されている必要があります。

11.7 その後のテスト

前述のように、受け入れテストゲートはプロジェクトのライフサイクルにおける重要なマイルストーンです。このゲートを通過したリリース候補は、どんな環境にもデプロイ可能です。もしリリース候補がこのゲートの通過に失敗したら、事実上、膨大な手作業を行わないかぎりはデプロイできないということです。これは望ましいことです。なぜなら、十分にテストされ、自動テストスイートが実行できる範囲で動作が保証された場合のみ、コードをリリースできるという規律が守られるからです。

CIパイプラインのこの段階までは、完全に自動化されています。もしリリース候補がプロセスの前段階をパスしていれば、リリース候補は次の段階へ進み、その段階が実行されます。

ほとんどのプロジェクトで、プロセスの残りの段階では順番どおりにゲートを通過していくアプローチは意味をなしません。代わりに、次に実行する段階を選択できるようにします。受け入れテストをパスしたリリース候補は、手動でのユーザ受け入れテスト、性能テストのためにデプロイするか、もしくは、実際に本番稼動へ向けてデプロイするかのいずれかを選択できます。

これらのデプロイごとに、クリーンデプロイ†を保証するため、「11.6 デプロイの準備」で示したステップを実行します。リリースが本番稼動へ到達する頃には、何度か同じテクニックを使ってデプロイに成功してきたはずです。そのため、失敗

† 訳注：クリーンインストールやクリーンビルドと同様、過去のデプロイの影響を受けない形でデプロイすること。

する不安はほとんどありません。

筆者の最近のプロジェクトでは、アプリケーションをホストする各サーバに、使用可能なリリース候補の一覧を表示する簡単なWebページを用意しました。そして、そのサーバの環境において、機能テストスイートと性能テストスイートの両方またはいずれか一方を選択的に再実行できるようにしました。これにより、プロセスに不注意で誤りが紛れ込む恐れはほとんどなく、望むときにはいつでも、望むところへはどこへでも、システムをとても柔軟にデプロイすることができました。

このプロセスはとても汎用的なのですが、受け入れテスト後の段階を進める際にどの程度自動化するのか、またはしないのかという点に関しては、最も変化しやすい部分だと言えます。性能テストを常に自動実行の範囲に含めるべきプロジェクトもあれば、そうではないプロジェクトもあるでしょう。受け入れテスト後の各段階の関係や、それらを手動で選択するのか自動で実行するのかといった細かいことは、CIプロセスを管理するスクリプトがきれいに分割されているならば、実際には問題ではありません。

11.8 プロセスの自動化

図11-2では、CIパイプラインの自動化に使われるスクリプトの簡単なチャートを見ることができます。各矩形はプロセスの段階を表します。矩形の中の各行は個々のスクリプトかビルドスクリプトのターゲットを表し、それぞれ、その名前が表す機能を実行します。

このアプローチを用いるほとんどのプロジェクトでは、最初の2つのプロセスゲート（チェックインゲートと受け入れテストゲート）はCruiseControlなどのCI管理アプリケーションによって開始されます。

ビルドスクリプトを体系化するアプローチの重要なメリットの1つは、各スクリプト、もしくはスクリプトの要素が、1つの複雑なステップでビルドプロセス全体を管理しようとするのではなく、それよりずっとわかりやすい1つの仕事にフォーカスするようになることです。ビルドプロセスを管理しやすくし、プロジェクトの発展と成熟にともなう変化を受け入れやすくするためにも、これはとても重要です。

これらのスクリプトの詳細はこのエッセイの範囲外としますが、実際のところ、プロジェクトに強く依存しすぎていて、あまり興味を持ってもらえないでしょう。しかし、筆者らはいくつもの異なるプロジェクトにおいて次のことを発見してきました。この種の高度な構造をビルドプロセスに適用するとき、デプロイは、確実で、繰り返し可能で、信頼のおけるものになります。それにより、以前であれば数日か

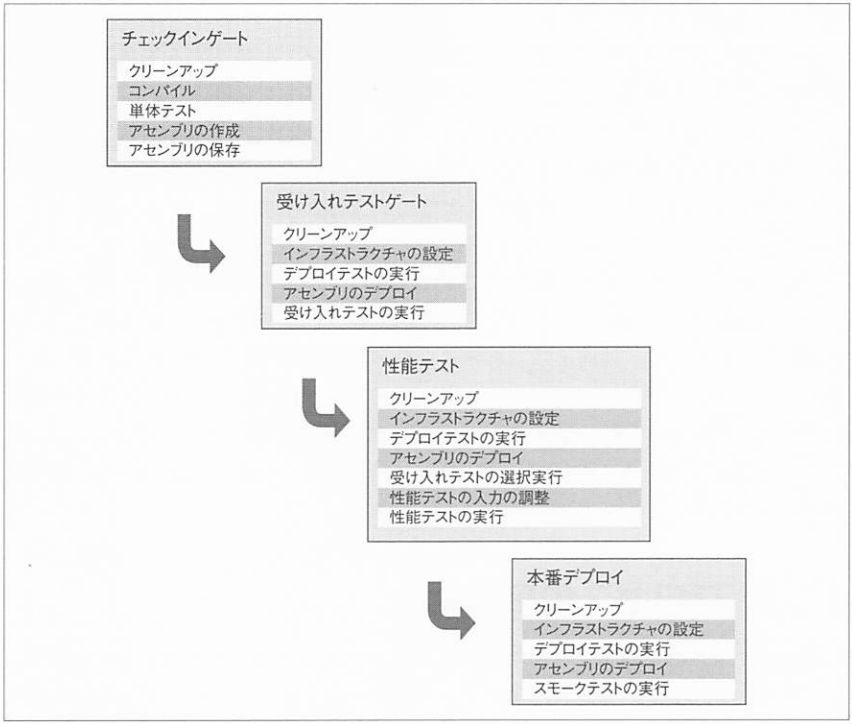

図 11-2 プロセスステップの例

かっていた（その上、たいてい悲惨な週末になる）デプロイは、数秒もしくは数分で終わるようになります。

11.9 まとめ

もし、あなたの組織がビルドにCIアプローチをまだ使っていないなら、明日にでも始めてください。CIアプローチは、私たちが知る中で、システムの信頼性を改善する最も効果的な方法です。

ヒューマンエラーの原因を効果的に可能なかぎり多く取り除くためにCIの適用範囲を広げることは、生産性だけでなく、成果物の品質向上にも本番リリースのストレス軽減にも、とても大きな利益をもたらします。

12章

アジャイルかウォーターフォールか
―エンタープライズWebアプリケーションのテスト

Kristan Vingrys ◎QAコンサルタント

12.1 イントロダクション

エンタープライズWebアプリケーションにおいて、アジャイルなプロジェクトと、ウォーターフォールのプロセスで開発するプロジェクトでは、テスト戦略にどのような違いがあるのでしょうか。どちらのテストも、アプリケーションの振る舞いを顧客に伝えることを目的としています。また、アプリケーションが本番環境に導入された後に障害が発生するリスクを取り除くことも目的の1つです。主な違いは、実行されるテストそのものではなく、いつ誰によってテストが実行されるかです。テストフェーズは、システムが存在すればいつでも開始できるものであり、前のテストフェーズが完了するまで待つ必要はありません。

このエッセイは、これまでアジャイルなプロジェクトにかかわったことのない人や、アジャイルなプロジェクトを始めたばかりでなんらかのアドバイスを求めている人を対象としています。これから紹介する情報は新しいものではありませんが、よりアジャイルなプロセスを取り入れるのに役立つようなものをまとめています。

アジャイルなプロジェクトのテストフェーズは、通常はウォーターフォールのプロジェクトと同じです。アジャイルなプロジェクトでも各フェーズに終了基準を設けることが可能ですが、アプリケーションが完成するまで各テストフェーズの開始を待つ必要はもうありません。代わりに、次のテストフェーズを開始するのに十分な程度のアプリケーションの完成を待てばよいだけなのです。リリースまで待たずに、完成した機能に対してテストを実行するため、テストフェーズは並行して†継

† 訳注：ある機能に対しては機能テスト中であり、別の機能に対しては統合テスト中であるというように、同じ時期に複数のテストフェーズが実施されうるということ。

続的に実施されます。この結果、数多くの回帰テストが必要となり、テストの自動化が不可欠になります。また、アジャイルなプロジェクトは環境をより早い段階でより頻繁に必要とするため、テスト環境の利用法やテストにかかわる人材も重要な関心事です。

「フェイルファスト」†がアジャイルなプロジェクトのモットーです。これは、アプリケーションが今のままではビジネスの要求を満たせないということを、できるだけ早く明らかにしようという意味です。これを実現するには、今のソリューションがビジネスニーズを満たしていることを継続的に確認する必要があります。また、ニーズを満たしていないときには、できるだけ早く問題を修正する必要があります。アジャイルなプロジェクトチームは、開発者やテスト担当者、アーキテクト、ビジネスアナリスト、ビジネスの代弁者††で構成され、この全員が、できるだけ早い時期にビジネス価値を提供したいと考えています。したがって、テストはすべてのチームメンバが考えるべきことであり、もはや単なるテスト担当者の責務ではありません。

12.2 テストのライフサイクル

テストのライフサイクルは、ウォーターフォールのプロジェクトとアジャイルなプロジェクトで大きく異なるところです。ウォーターフォールのプロジェクトでは各フェーズに厳格な開始基準と終了基準があり、前フェーズが完了したときのみ次のフェーズに移ります。アジャイルなプロジェクトはテストフェーズをできるだけ早く開始し、複数のフェーズを並行して実施可能です。アジャイルなプロジェクトにも終了基準などの決まりごとが多少あるものの、厳格な開始基準は存在しません。

図12-1を見ると、アジャイルなプロジェクトとウォーターフォールのプロジェクトにおける、テストのライフサイクルの違いがすぐにわかるでしょう。アジャイルなプロジェクトでは、ビジネスアナリスト、テストアナリスト、およびビジネスの代弁者が、今検討中のアイデアについて議論します。議論の内容は、そのアイデアによって何が実現できるか、そのアイデアを全体に対してどう適応させるか、期待どおりの仕事が行われていることをどのような手法で立証するのか、というものです。このような分析が、機能テストやユーザ受け入れテスト、性能テストの基礎に

† 訳注：可能なかぎり早く失敗すること。問題を早期発見し、早期に対応する。
†† 訳注：プロジェクトによって異なるが、ビジネスの特定分野の専門家など、ビジネスを説明できる担当者を表す。

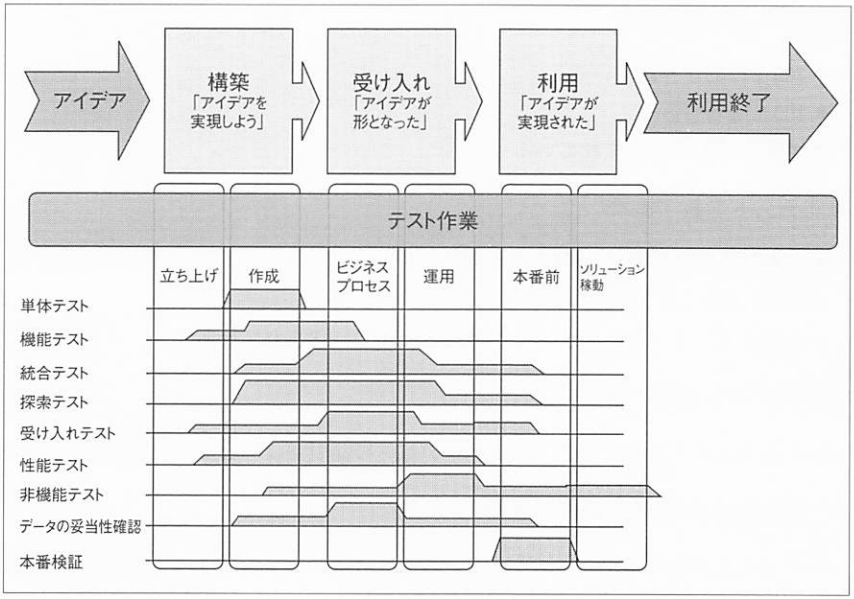

図 12-1 アジャイルなプロジェクトとウォーターフォールのプロジェクトにおけるテストのライフサイクル

なります。機能の開発が行われるのはこの分析の後であり、このときに単体テスト、統合テスト、探索的テスト、非機能テスト（必要な場合はデータの妥当性確認も）が始まります。本番検証は、システムが本番稼動する直前にのみ行われます。

テストフェーズに厳格な開始基準を設けないということは、適切なときにいつでもテストを開始できるということです。すべてのテストフェーズがアプリケーションの品質確保にとって重要であるため、各フェーズの分析をできるだけ早く行うことが重要です。早い段階でのテスト分析は、アプリケーション設計を方向付けて問題を明らかにすることに役立ち、これによりプロジェクトの後半で相当多くの時間を節約できます。アジャイルなプロジェクトの終了基準の例は次のとおりです。

単体テストの終了基準

- 100%自動化されている
- 100%成功する
- コードカバレッジが90%を超えている
- 継続的ビルドに含まれている

統合テストの終了基準

- 100%自動化されている
- 100%成功する
- 継続的ビルドに含まれている

機能テストの終了基準

- 自動化の割合が90%を超えている
- 100%成功する
- 自動化されたテストのすべてが継続的ビルドに含まれている

探索的テストの終了基準

- テストアナリストから、アプリケーションの品質がよいという信頼を得る

ユーザ受け入れテストの終了基準

- ビジネスの代弁者から、アプリケーションがニーズを満たしているという合意を得る
- ユーザから、アプリケーションが使えるものであるという合意を得る

性能テストの終了基準

- 100%自動化されている
- ビジネス側から、アプリケーションがビジネス上の性能要求を満たしているという合意を得る
- 性能テストが繰り返し実行可能である

非機能テストの終了基準

- ビジネス側から、非機能要求を満たしているという合意を得る
- 運用側から、非機能要求を満たしているという合意を得る

データの妥当性確認テストの終了基準

- データを正確に移行したと確信できる

本番検証の終了基準

- アプリケーションを本番環境に正常にインストールしたと確信できる

ウォーターフォールのプロジェクトにおけるテストのライフサイクルでは、前のフェーズが完了するまで、テストフェーズに入ることができません。理論的には、あるフェーズの完了にそれ以降のテストフェーズが依存することになるため、この制限は妥当なものです（ある機能が正しく動作しない状態で、その性能をわざわざテストしたくないでしょう）。しかし、すべての機能が正しく動作するようになるまで待ってから性能テストを始める必要はありません。アジャイルなプロジェクトでは各テストフェーズを適切なタイミングで始めるため、結果として問題が早い段階で発見され、チームは問題を修正するための時間をより多く得ることができます。しかし、アジャイルなプロジェクトのテストフェーズの終了は、やはりウォーターフォールのプロジェクトと同じです。機能に対する性能テストは、その機能が正しく動作するようになってからでないと、完了したとは見なされません。

12.3 テストの種類

アジャイルなプロジェクトで行われるテストの種類は、ウォーターフォールのプロジェクトとほとんど変わりません。主な違いは、テストの焦点と各テストフェーズの実施時期にあります。アジャイルなプロジェクトは単体テストと機能テストを重視するため、これらのテストは後のテストフェーズのための高い品質のコードを生み出します。この結果、後の方のテストフェーズでは、それ以前のフェーズで発見できたはずの不具合が見つかるということがなくなり、今テストしたい領域に集中できます。このような問題は、ウォーターフォールのプロジェクトでは一般的です。ウォーターフォールのプロジェクトでは、プロジェクト後半のテストフェーズで行う作業の焦点が、もっと早く発見できたはずの不具合を見つけることになります。この結果、不具合の修正に対してより多くの費用が発生し、テスト作業が重複し、テストの焦点がぶれてしまいます。

ウォーターフォールのプロジェクトとアジャイルなプロジェクトでのもう1つの大きな違いは、テストの自動化です。アジャイルなプロジェクトは、テストのすべての領域で100%のテストの自動化を目指します。テストは継続的ビルドシステムに統合されるため、コードの変更時はその変更が自動的に検出され、アプリケーションがビルドされ、その後すべてのテストが実行されます。

テスト駆動開発（TDD：Test-Driven Development）は、アジャイルなプロジェクトで一般的に利用されるアプローチです。このアプローチは、コードを書く前にテストケースを記述することを意味します。テスト駆動開発を利用することで、通常はコードや機能に対するテストケースが事前に作成されるようになります。自動化

されたテストを使って開発し、重複を除去することで、どれほど複雑であっても、バグのない信頼できるコードをあらゆる開発者が書くことができます。テスト駆動開発は一般的に単体テストで使われますが、機能テストや統合テスト、ユーザ受け入れテスト、性能テストでも利用できます。

12.3.1 単体テスト

単体テストは、ホワイトボックステストとしても知られ、モジュールの開発中に行うテストも含まれています。ウォーターフォールのプロジェクトはこのテストフェーズを重視せず、行うとしてもたいていの場合はアドホックに行います。アジャイルなプロジェクトは単体テストと、すべての単体テストの自動化を重視します。自動化された単体テストはアジャイルなプロジェクトの土台となり、継続的インテグレーションやリファクタリングを支援します。

単体テストについて考慮すべき点は、次のとおりです。

- 外部のインターフェイスへの依存をなくすためにスタブやモックを利用する
- コードを書く開発者がテストを記述する
- テストを自動化し、開発用の継続的ビルドに含める
- 単体テスト間の依存をなくし、各単体テストを単独で実行できるようにする
- すべての開発者が自分のマシン上でテストを実行できるようにする
- コードカバレッジを利用して、単体テストが存在しないコードを明らかにする
- 単体テストが100%成功してから、変更したコードをチェックインする

テストの失敗はビルドの失敗です。

12.3.2 機能テスト

機能テストは、一般的にシステムテストとも呼ばれ、アプリケーションの機能のテスト（無効となる条件や境界上の条件に対するテストを含む）を対象とします。ウォーターフォールのプロジェクトでは、テストチームの作業は機能テストから始まるのが一般的です。開発者がすべての機能を完成し、単体テストを成功させて機能テストのフェーズに移るまで、テストチームのメンバが待つことになります。アジャイルなプロジェクトでは機能を複数のストーリー†に分割し、いくつかのスト一

† 訳注：ストーリーとは、エクストリームプログラミング（XP）における要求定義の手法。XPでは、ユースケースの代わりにストーリーが用いられる。日本ではユースケースが主流だが、海外のアジャイル開発者の間ではストーリーのほうがよく使われているようである。

リーが1回のイテレーションで開発されます。すべてのストーリーにはいくつかの受け入れ基準があります。受け入れ基準は、通常ビジネスアナリストやテストアナリストが作成するものであり、テスト条件と同様のものと考えることができます。その後、テストアナリストがこの受け入れ基準の観点から、完成したコードの振る舞いを実証するためのテストケースを作成します。ストーリーがコーディングされ単体テストされた後は、受け入れ基準を満たしているかどうかを判断するために、機能面がテストされます。これはつまり、アジャイルなプロジェクトでの機能テストが、最初の機能がコーディングされたときに始まり、プロジェクトのライフサイクル全体にわたって続いていくことを意味しています。

機能テストについて考慮すべき点は、次のとおりです。

- テストを自動化し、開発用の継続的ビルドに含める（テストの実行時間が長い場合は、開発用の継続的ビルドには一部のテストのみを含め、システム統合用の継続的ビルドのほうにすべてのテストを含めることもできる）
- ソースコードを書く前に、そのテストの目的を書く。また、ソースコードが完成した後に、テストの実装を完了できる†
- すべての機能テストを成功させてはじめて、ストーリーが完成したと考えてもよい
- アプリケーションが他の環境（ステージング環境や、可能ならば本番環境でも）にインストールされたときは、機能テストを実行する

テストの失敗はビルドの失敗です。

12.3.3 探索的テスト

探索的テスト††は、アドホックテストとしても知られています。ウォーターフォールのプロジェクトではテスト戦略にこの種のテストを含めませんが、ほとんどのテスト担当者はこのテストをある程度は実施します。探索的テストは、アジャイルなプロジェクトにとっては重大なテストフェーズです。自動化されたテストのカバレッジを確かめて、アプリケーションの品質に関する全般的なフィードバックを得

† 訳注：テストの対象となるソースコードの実装を始める前に、そのテストの内容を検討して記述しよう。また、ソースコードの実装を行った後には、自動化するテストコードの実装を行おう、ということ。「テストの目的」は、例えば「ゴールド会員としてログインし、正しいウェルカムメッセージが表示されるかどうかを検証する」というものである。

†† 訳注：テスト担当者がテストの設計と実行を同時に行い、該当する領域を学習しながらテストを進める手法のこと。「12.9 参考文献」に載せた *"Exploratory Testing Explained (v.1.3 4/16/03)"* に詳しい説明がある。

るために利用するからです。探索的テストは、テスト担当者やビジネスアナリストが不具合を発見するために、システムを実際に動かし探索する構造化された手法です。探索的テストで機能のある領域に非常に多くの不具合が発見された場合は、この領域の既存の自動化されたテストケースを見直します。

探索的テストについて考慮すべき点は、次のとおりです。

- システム統合環境で実行する
- 探索的テストの作業を高いレベルで記録する（Wiki上でもよいだろう）
- 設定にかかる時間を削減するために、設定を自動化する
- 探索的テストの一部として、破壊テスト†を含める

12.3.4 統合テスト

統合テストは、アプリケーションの本番導入の際に必要となる、個々の部品の統合に関するテストです。ウォーターフォールのプロジェクトでは、開発中のアプリケーションが必要とする、プロジェクトの対象外のアプリケーションとの統合だけではなく、アプリケーション内部にある個々のモジュール間の統合も含みます。アジャイルなプロジェクトでは、アプリケーション内部にある個々のモジュール間の統合は継続的ビルドでまかないます。このため、プロジェクトの開発対象外の外部インターフェイスが統合テストの焦点になります。

統合テストについて考慮すべき点は、次のとおりです。

- 統合テストの実行時に、現在のイテレーションでは開発されていない機能について考慮する
- 統合ポイントが機能テストを通して呼び出される場合でも、コードのデバッグに役立つようにその統合ポイント自体を対象とする統合テストを作成する††
- 統合テストを自動化し、システム統合用の継続的ビルドに含める

---

† 訳注：誤った利用法や想定外の入力が行われた際にも、正しい動作を行うことを検証するためのテストのこと。「破壊テスト」または「破壊検査」（destructive testing）は通常ハードウェアに対して実施されるが、近年はソフトウェアでもその考え方が取り入れられるようになってきている。

†† 訳注：例えば、機能テストで画面からサブミット処理のテストを行い、正しいデータの取得を確認することで、ビジネスロジックとデータベースとの統合ポイントも正しく動作していると見なすことはできる。しかし、コードのデバッグがしやすくなるように、このビジネスロジックとデータベースとの統合ポイントのみを対象とする統合テストを行うべきだということ。

テストの失敗はビルドの失敗です。

12.3.5 データの妥当性確認

データの妥当性確認は、既存データの移行が必要な場合に実行しなければならないものです。データの妥当性確認によって、既存のデータが新しいスキーマに正しく移行されたことや、新しいデータの追加、冗長なデータの削除が保証されます。ウォーターフォールのプロジェクトもアジャイルなプロジェクトもほぼ同様の方法でこの種のテストに取り組みますが、1つだけ例外があります。それは、アジャイルなプロジェクトではできるかぎりテストを自動化するという点です。

データの妥当性確認について考慮すべき点は、次のとおりです。

- システム統合環境、ステージング環境、本番環境で実行する
- できるかぎりテストを自動化する
- テストをシステム統合用の継続的ビルドに含める

テストの失敗はビルドの失敗です。

12.3.6 ユーザ受け入れテスト

ユーザ受け入れテスト（UAT：User Acceptance Testing）は完全なビジネスプロセスを対象とし、アプリケーションがビジネスの仕組みに合っていることやビジネスニーズを満たしていることを保証するものです。また、ユーザ受け入れテストでは、顧客、消費者、管理者や他のユーザにとってのアプリケーションのユーザビリティも確認します。ウォーターフォールのプロジェクトでは、テストのライフサイクルの早い段階で発見すべきバグを、この段階で発見することになるのが一般的です。また、この種のテストは多くの場合、開発チームからリリースされたアプリケーションの品質を検証するために、ビジネス側で実施されます。アジャイルなプロジェクトでは、ユーザ受け入れテスト開始時点でのコードの品質が高いため、このフェーズではアプリケーションがビジネスニーズを満たしていると保証することに集中できます。ビジネス側はアジャイルなプロジェクトの早期のテストフェーズにかかわっているため、リリースされるものを強く信頼できるのです。

ユーザ受け入れテストについて考慮すべき点は、次のとおりです。

- 先にテストを手動で実行してみて、システムが期待どおりに動作することを検証してから自動化する
- 自動化されたテストをシステム統合用の継続的ビルドに含める

- アプリケーションのエンドユーザに、手動でのテストをひととおり行ってもらう。ただし、プロジェクトのテスト担当者がそのテストのまとめ役を担当する
- ステージング環境で受け入れの承認のためのユーザ受け入れテストを行う
- 1つのビジネスプロセスや主要なUIコンポーネントが完成するたびにユーザ受け入れテストを実行する

テストの失敗はビルドの失敗です。

12.3.7 性能テスト

性能テストの対象領域は数多くありますが、一般的には次の3つに分類できます。

キャパシティテスト

中核となる機能のコンポーネントについて、それぞれ単独でのキャパシティを検査する。例えば、同時に何人のユーザが並行して検索を実行できるか、1秒間に何回の検索を実行できるか、などである。キャパシティテストは、システムの究極の限界を測定するために利用され、さらに、スケーラビリティの検討やキャパシティプランニングにも役立つ

ロードテスト

負荷がかかったときのシステムの振る舞いを対象とする。負荷には、予想されるシステムのトラフィックの構成を反映すべきである

ストレステスト

ストレスのある状況でシステムがどのように振る舞うかを対象とする。ソークテストは、一般的なストレステストのテクニックである。これは、メモリリークやリソースリークといった長期に及ぶ問題をあぶり出すために、システムを長い期間、負荷の高い状態に置くことである。ストレステストの対象範囲には、フェイルオーバーや復旧も含まれる。例えば、システムを負荷の高い状態においてクラスタ構成の一方のサーバに障害を発生させ、そのサーバが正しく停止して復旧するかを検査する

ウォーターフォールのプロジェクトでは、プロジェクトの最後、つまりアプリケーションが「完全に」開発されて単体テストと機能テストがすべて実行されるまでは、性能テストを実行しません。アジャイルなプロジェクトでは、性能テストをできるだけ早い段階で実行します。

性能テストについて考慮すべき点は、次のとおりです。

- 性能に関するメトリクスを機能テストに組み込む。例えば、あるテストが最初の実行時にどれほどの時間かかるかを測定し、その後の各テスト実行時に、時間の差をパーセンテージで比較する（増加は×、減少は○）
- システム統合用の継続的ビルドに一部の性能テストを含める
- ビジネスプロセスや重要な機能、およびインターフェイスが完成するたびに性能テストを実行する
- ステージング環境で性能テストを実行した後でのみ、性能テストを承認できる

テストの失敗はビルドの失敗です。

12.3.8 非機能テスト

非機能テストは多くの異なる分野を対象にしており、通常は性能テストもここに含まれます。しかし、性能テストはエンタープライズソリューションにおける重要な要素であり、さまざまな人材やスキルセットを必要とするため、異なるテストフェーズとして分類しました。一般的に、非機能テストが対象とする分野には、運用（モニタリング、ロギング、監査や履歴を含む）、信頼性（フェイルオーバー、単一コンポーネントの障害、全体障害、インターフェイスの障害を含む）、セキュリティが含まれます。ウォーターフォールのプロジェクトもアジャイルなプロジェクトもこのテストフェーズに取り組み、そのアプローチの違いはほとんどありません。

非機能テストについて考慮すべき点は、次のとおりです。

- 通常、非機能要求は明確に定義されず、定義されたとしても（例えば99.9%の稼働時間というように）簡単に測定できるものではない
- できるかぎり非機能要求に対するテストを自動化し、システム統合用のテスト環境に含める
- テストケースを定義する際に、本番環境をモニタリングしサポートする要員をかかわらせる
- 非機能テストのうちモニタリングは、アプリケーションが本番稼動を始めてからも続く

12.3.9 回帰テスト

ウォーターフォールのプロジェクトにとって、回帰テストは時間と費用の面で最もコストがかかるテストフェーズの1つです。もしプロジェクトのライフサイクルの終盤、例えばユーザ受け入れテストのフェーズで不具合が見つかったら、その修

正を行ったアプリケーションのビルドでは、すべての単体テスト、機能テスト、ユーザ受け入れテストを再実行する必要があります。

ほとんどのウォーターフォールのプロジェクトには自動化されたテストがないため、回帰テストのコストが高くつきます。アジャイルなプロジェクトは、継続的ビルドと自動化されたテストによって回帰テストを取り込んでいるため、すべてのビルドで回帰テストが実行されます。

回帰テストについて考慮すべき点は、次のとおりです。

- 手作業でのテストは各イテレーションの最後に実行し、早めにフィードバックする（テスト一式が巨大な場合は、3～4回のイテレーションごとに実行されるようにテストのローテーションを組む）

12.3.10 本番検証

本番検証では、本番環境のアプリケーションを検査します。このテストでは、ユーザがシステムを利用する前に、すべてが適切にインストールされてシステムが運用できる状態にあることを確認します。一方、本番システムより前の段階では完全に実行できないテストが存在することもあり、このような場合はこのテストを本番検証でできるかぎり早く実行します。本番検証に対するウォーターフォールのアプローチとアジャイルなアプローチに違いはありません。

本番検証について考慮すべき点は、次のとおりです。

- エンドユーザに本番検証のテストを実行してもらう
- 本番システム上で、稼動開始前に初期のテストフェーズで作った自動化された回帰テストをできるだけ多く実行する

ウォーターフォールのプロジェクトとアジャイルなプロジェクトのテストフェーズは類似していますが、各テストフェーズの焦点や実施時期に違いがあります。アジャイルなプロジェクトは、自動化されたテストを多く作成し、また継続的インテグレーションを利用することで、プロジェクトにおける回帰テストがもたらす影響を和らげます。ウォーターフォールのプロジェクトでは、アプリケーションの品質が低い状態のまま、前半のフェーズで作ったテストを後半のフェーズで実行する（つまり、ユーザ受け入れテストで機能テストを実行する）のが一般的です。アジャイルなプロジェクトはテストの無駄を削減し、障害を早めに検知し、リリースされるアプリケーションへの信頼を高めます。

12.4 環境

テスト環境は、アプリケーションが正常に動作することを保証するために、開発プロセスのさまざまな段階で利用されます。後のフェーズで利用される環境ほど、予定している本番環境に類似します。代表的なテスト環境としてまず開発統合環境†があり、これは開発者がコードを統合して一部のテストを実行できる環境です。システム統合環境は開発統合環境と似ていますが、より多くのサードパーティ製のアプリケーションや、より大規模なデータ一式と統合されます。ステージング環境はできるだけ本番に近い環境であり、本番前の最終ステージです。

アジャイルなプロジェクトとウォーターフォールのプロジェクトでは、必要とされる環境に大きな違いはありません。違うのは、アジャイルなプロジェクトはプロジェクトの開始時からすべての環境を必要とし、プロジェクトのライフサイクル全体を通して環境を利用するという点です。アジャイルなプロジェクトにとっては、環境がいつでも利用でき、正しく動作していることも欠かせません。もしなんらかの理由で環境に問題が発生すれば、環境を立て直すことが最優先の作業になります。アジャイルとウォーターフォールにおけるもう1つの違いは、環境の計画や調達への影響力であり、これは特に環境がプロジェクト外部のチームによって管理されている場合に当てはまります。

12.4.1 開発統合環境

開発統合環境は、開発者がコードを統合し、開発中のアプリケーションを作成するために利用されます。ウォーターフォールのプロジェクトは、この環境を重要なものとは考えません。開発統合環境は頻繁に壊れるものであり、開発者が他の開発者とコードを統合しなければならないとき、つまり一般的にはプロジェクトの終盤のみに必要なものと考えています。アジャイルなプロジェクトでは、開発統合環境は開発作業に不可欠であり、あらゆるコーディングの開始以前に利用できなければなりません。継続的にコードを統合しテストスイートを実行するためにこの環境を利用します。なんらかの理由で環境が壊れた場合は、環境を立て直すことが最も優先度の高い項目となります。

開発統合環境について考慮すべき点は、次のとおりです。

† 訳注：開発者間でコードを統合するための開発環境を表しており、統合開発環境（IDE）とは異なる。

- コードの統合、ビルド、テストにかかる時間を15分以内に抑える
- 各開発者が利用中の開発統合環境を同じものにそろえる（ハードウェアが異なるのはよいが、ソフトウェアは同じにすべきである）
- 利用するデータはできるだけ本番データに近いものにする。本番データが大きすぎる場合は、その一部をコピーしたものを利用できる。各リリースサイクルの開始時に、本番データを基にデータをリフレッシュすべきである
- 環境の管理はプロジェクトチームの責任とすべきである
- この環境へは1時間ごとにデプロイするのがよいだろう
- この環境へのデプロイを自動化する

12.4.2 システム統合環境

システム統合環境は、開発中のアプリケーションと、それが連携する他のアプリケーションとを統合するために利用されます。ウォーターフォールのプロジェクトでは、この環境は（存在する場合でも）プロジェクトの終盤でのみ利用され、複数のプロジェクト間で共同利用されることが多いでしょう。アジャイルなプロジェクトでは、コーディングを始める日からこの環境を利用できる状態でなければなりません。アプリケーションは頻繁にこの環境にデプロイされ、機能テスト、統合テスト、ユーザビリティテスト、および探索的テストが実行されます。アプリケーションのデモ（見本の展示）はこれ以降の環境で行われます。

システム統合環境について考慮すべき点は、次のとおりです。

- 統合ポイントでは実際の外部のアプリケーションに置き換えるべきである。ただし、この外部のアプリケーションは、本番バージョンではなくテスト環境のものを利用すべきである
- アーキテクチャを本番環境と同様にする
- この環境では本番環境のデータのコピーを使い、各リリースサイクルの開始時にデータをリフレッシュすべきである
- 完全なテストスイートを実行するシステム統合用の継続的ビルドをこの環境で行う
- 環境の管理はプロジェクトチームの責任とすべきである
- この環境へは1日ごとにデプロイするのがよいだろう
- この環境へのアプリケーションのデプロイを自動化する

12.4.3 ステージング環境

ステージング環境は、アプリケーションが本番環境にデプロイされ動作することを検証するために存在します。この目的のために、ステージング環境は、ネットワーク設定、ルータ、スイッチ、コンピュータの処理能力を含め、本番環境と同様の設定をしたものになります。ウォーターフォールのプロジェクトでは、一般的にはこの環境を「予約する」必要があり、必要なデプロイ回数やいつデプロイを行うのかについて計画します。アジャイルなプロジェクトは開発統合環境やシステム統合環境を重視していますが、ステージング環境についてはそこまでは依存していません。しかしなお、プロジェクトのライフサイクルを通して、多数のデプロイの実行を含めこの環境に頻繁にアクセスできる必要があります。

ステージング環境について考慮すべき点は、次のとおりです。

- 利用するデータは本番データのコピーとし、アプリケーションの各デプロイの前にデータをリフレッシュすべきである
- ユーザ受け入れテスト、性能テスト、非機能テスト（安定性、信頼性など）に関する承認のために利用する
- この環境へは2週間などのイテレーション単位でビルドをデプロイするのがよいだろう
- この環境は本番環境の管理者が管理すべきである。これにより、この管理者たちはアプリケーションを早い段階から見て、必要な知識を得ることができる
- この環境へのアプリケーションのデプロイを自動化する

12.4.4 本番環境

本番環境は、アプリケーションが本番稼動する場所です。本番検証テストはこの環境で実行され、また、テストの有効性を測定するためのメトリクスの収集も行われます。これはウォーターフォールのプロジェクトでもアジャイルなプロジェクトでも同様です。アジャイルなプロジェクトでは、本番への頻繁なリリースができるように、リリースのプロセスをできるだけ自動化することになるでしょう。

本番環境について考慮すべき点は、次のとおりです。

- 稼動開始前（もしくは稼動開始直後）に、自動化された回帰テストのテストスイートを本番環境で実行する
- テストの有効性を測定するためのメトリクスを利用できるようにする。例えば、最初の3か月から6か月にユーザが報告する問題の重要度や発生数などである

あらゆるプロジェクトのスケジュール上、環境が必要なときに利用できるかどうかは非常に重要です。ウォーターフォールのプロジェクトは、かっちりした計画のとおりに進めようとするため、環境の利用予定を決めることは簡単です。アジャイルなプロジェクトはもっと流動的です。ステージング環境に進めると保証できるほどの十分な機能がビルドされるのかもしれないし、ビジネスが本番への早期の移行を決定するのかもしれません。アジャイルなプロジェクトを支援するためには、ステージング環境やその後の本番環境への迅速なデプロイを可能にするリリースプロセスを整えて、システム統合環境を利用できるようにすべきです。

12.5 課題管理

課題には不具合（バグ）と変更依頼が含まれます。ウォーターフォールのプロジェクトは不具合や変更依頼に対して非常に厳格な管理を行いますが、アジャイルなプロジェクトは変化を抱擁しているため、変更管理についてはそこまで厳格ではありません。変更が必要であれば、新しいストーリー（複数の場合もある）が作成されて、バックログに追加されます。最も優先度の高いストーリーが次のイテレーションに割り当てられます。

不具合管理はアジャイルなプロジェクトでも適用できます。開発中のストーリーで不具合が発見されたときは、インフォーマルな（肩越しの）コミュニケーションによって開発者たちにこれらの不具合が伝えられて、その不具合は即座に修正されます。現在のイテレーションのストーリーに関係しない不具合や、開発中のストーリー内での重要でない不具合が発見されれば、その不具合は不具合追跡ツールに登録されます。不具合はその後、ストーリーと同様に扱われます。つまり、ストーリーカードが作成され、顧客によって優先度を付けられ、ストーリーのバックログに追加されます。チームは、顧客が不具合を理解して優先度を付けるための十分な情報を提供し、一方で顧客にとって優先度が高くない不具合に多くの時間をかけすぎないように、不具合に対するトラブルシューティングのバランスを保つ必要があります。

不具合の内容（説明、対象コンポーネント、重要度など）はアジャイルなプロジェクトでもウォーターフォールのプロジェクトでも、1つの例外を除いては同じです。この例外とは、アジャイルなプロジェクトで記録するべきビジネス価値のことであり、可能であれば金額で記述します。これは、不具合が解決されたときに実現されるビジネス価値のことです。不具合にビジネス価値を関連づけることで、その不具合が新機能よりも価値が高く、そのため優先度が高いのかどうかを、顧客が

判断しやすくなります。

12.6 ツール

すべてのプロジェクトがなんらかのツールをある程度は利用します。ウォーターフォールのプロジェクトでは、効率の向上だけではなく、プロセスの徹底のためにもツールを利用しますが、この2つは対立することもあります。アジャイルなプロジェクトは効率の向上のためにツールを利用し、プロセスの徹底のためには利用しません。アジャイルなプロジェクトでは、あらゆるチームメンバがすべてのテストを自分の環境で実行できなければなりません。このため、すべてのチームメンバが、テストケースを自動化するツールを利用できる必要があります。この理由で、アジャイルなプロジェクトでは通常、オープンソース製品が利用されます。これは、これらのツールを利用するのに必要なスキルが多岐にわたることを意味します。商用ツールと比較すると、オープンソースのツールのドキュメントやサポートは十分ではありません。このため、利用者はコーディングに熟練している必要があります。開発者と有能ではないプログラマとをペアにすることは、後者のスキルを高める優れた手法です。アジャイルなプロジェクトでも商用ツールを使うことはできますが、商用ツールの大部分はアジャイルなプロセスを考慮して開発されていません。このため、商用ツールはアジャイルなプロセスに簡単には適合しません。特に継続的インテグレーションに関する商用ツールを動作させるためには、コードを大量に追加しなければならない可能性があります。

テストのために、プロジェクトは次のようなタスク用のツールを検討すべきです。

- 継続的インテグレーションツール（例：CruiseControl、Tinderbox）
- 単体テスト（例：JUnit、NUnit）
- コードカバレッジ（例：Clover、PureCoverage）
- 機能テスト（例：HttpUnit、Selenium、Quick Test Professional）
- ユーザ受け入れテスト（例：Fitness、Quick Test Professional）
- 性能テスト（例：JMeter、LoadRunner）
- 課題追跡（例：Bugzilla、JIRA）
- テスト管理（例：Quality Center）

12.7 レポートとメトリクス

ソフトウェアの品質やテストの量を測定するために、メトリクスが収集されます。ウォーターフォールのプロジェクトのテストメトリクスには、次の点を満たす必要のあるものがあります。テストの実行前にすべてのテストケースを作成していること、およびアプリケーションの完成後にそのテストを実行することです。そして、実行するテストケースごとの不具合数や、1日のテストケース実行数などのメトリクスは、アプリケーションがリリースできる状態にあるかどうかを判断するために収集し、利用することが可能です。アジャイルなプロジェクトでは、機能が完成した際にテストケースを作成し実行します。このため、ウォーターフォールのテストで利用されるメトリクスをそのまま適用することはできません。

メトリクスを収集する理由に話を戻しましょう。アプリケーションの品質やテストの量を測定するためには、次の点について調査するとよいでしょう。

- コードカバレッジを使用してテストの量を測定する。これは特に単体テストで有効である
- 探索的テストで発見された不具合数は、単体テストと機能テストの有効性を示す
- ユーザ受け入れテストで不具合が見つかる場合は、前段階までのテストがユーザ受け入れテストを実行するには不十分であり、ソフトウェアのバグではなくビジネスプロセスの検証を焦点とすべきであったことを示している。特に、ユーザ受け入れテストでソフトウェアのバグよりも機能的な課題が多く見つかる場合は、ストーリーや変更要求への理解に不足があることを示している
- ストーリーの完成後に発見される不具合数は、ソフトウェアの品質に対する適切な基準である。これは、統合テスト、非機能テスト、性能テスト、およびユーザ受け入れテストで発見される不具合（機能変更ではなくソフトウェアのバグ）の数である
- 不具合の再発生率。もし不具合が頻繁に再発生していれば、ソフトウェアの品質が低いことを示している

12.8 テストのロール

テストのロールは人と1対1に対応するわけではありません。1人がすべてのテストのロールを担うかもしれませんし、別々の人々がそれぞれのロールを担うかも

しれません。ここで紹介するロールは、プロジェクトのテストを確実に成功させるために必要なものです。優れたテスト担当者はこれらすべてのロールの一面を持っています。ロールは、アジャイルなプロジェクトでもウォーターフォールのプロジェクトでも同じです。違うのは誰がロールを担うかです。アジャイルなプロジェクトでは、チームメンバ全員がなんらかのテストのロールを担います。図 12-2 は、アジャイルなプロジェクトの各チームメンバが担うロールの例を示しています。これは従うべきものではなく、各チームによって事情は異なります。しかし、これを優れたロールの組み合わせの例として見ることができます。

12.8.1 テスト分析

テスト分析では、要求、アーキテクチャ、コードやその他の成果物を基に、何をテストすべきか、どこを集中してテストすべきかを決定します。

ウォーターフォールのプロジェクトでは、通常は 1 人または複数の上級担当者がこのロールを担います。彼らは、関連するドキュメント（要求、設計、アーキテク

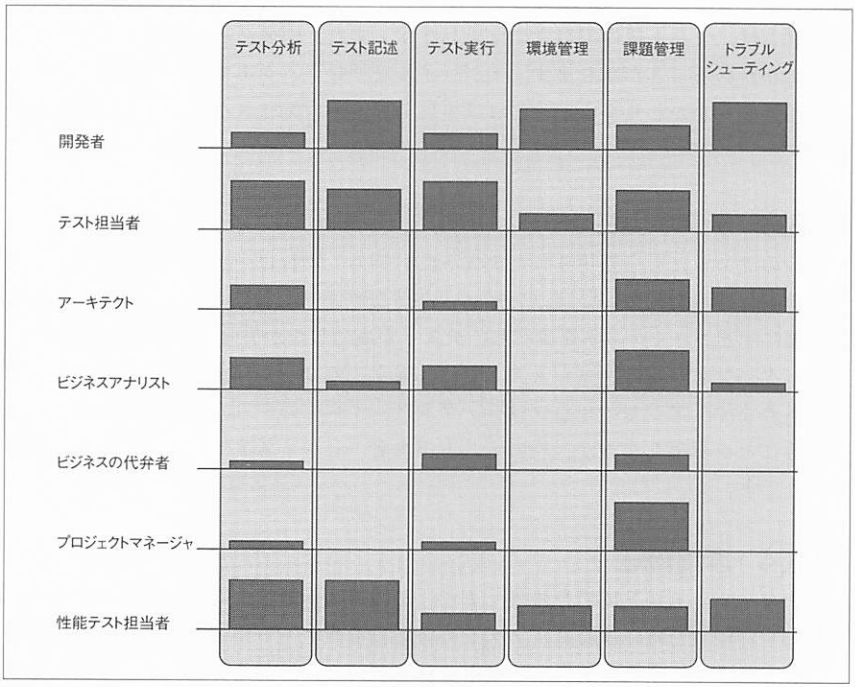

図 12-2 さまざまなチームメンバに対するテストのロール

チャ）を分析して、テスト計画を作成し、高いレベルのテストケース説明書を作成します。その後詳細なテストを記述する初級担当者にすべてを引き渡します。アジャイルなプロジェクトではすべてのチームメンバがこのロールを担うことを奨励します。開発者が単体テストを作成するためにコードと設計の分析に力を注ぎ、機能テストの作成時にビジネスアナリストやテスト担当者を手助けすることもあります。また、非機能テストや性能テストの分析にもかかわります。ビジネスアナリストとテスト担当者は密に連携して機能テストの作成やユーザ受け入れテストの作成、探索的テストの実行を行います。顧客やエンドユーザはユーザ受け入れテストに積極的にかかわります。

12.8.2 テスト記述

このロールは、テストを詳細に記述するロールです。テストには手動のものも、自動化されたものもあります。ウォーターフォールのプロジェクトでは、通常は初級担当者がこの記述を担当します。この担当者は、テスト計画とテストケース説明書を利用して、詳細なステップごとの手動テストの指示書を作成します。自動化スクリプトは、より上級の担当者が作成します。また、開発者も単体テストケースの記述にかかわることがあります。アジャイルなプロジェクトでは、開発者がテストの記述に多くかかわりますが、これは主にテストの自動化スクリプトを作成するためです。

12.8.3 テスト実行

手動のテストにも自動化されたテストにもテスト実行ロールが存在します。しかし、自動化されたテストではコンピュータがこのロールを担います。テスト実行者は詳細に記述されたテストを実行し、テストが成功したか失敗したかを判断します。ウォーターフォールのプロジェクトでは、通常このロールはテスト担当者によって実行されます。アジャイルなプロジェクトは、すべてのチームメンバがこのロールを担うことを奨励しますが、特にテスト担当者、ビジネスアナリスト、および顧客がそうです。

12.8.4 環境管理

このロールはテスト環境を管理します。このテスト環境には、アプリケーションが稼動する環境だけではなく、テストの自動化をサポートするために必要なインフラも含まれます。また、テストで利用されるデータや外部インターフェイスにも気を配ります。このロールはウォーターフォールのプロジェクトとアジャイルなプロ

ジェクトで類似しています。

12.8.5 課題管理

課題は発生するものであり、発生すれば継続的に対応する必要があります。このロールは、課題を選別して、重要度や優先度、対象コンポーネントなどの情報を含めて適切に登録します。課題のライフサイクルや関連ツールを管理することも、このロールの一部です。このロールは、ウォーターフォールのプロジェクトとアジャイルなプロジェクトで非常に類似しています。

12.8.6 トラブルシューティング

このロールは、課題が生じたときにトラブルシューティングを行い、その課題がソフトウェアの不具合かどうかを判断します。ソフトウェアの不具合に対しては、原因を調査し、解決策や回避策を考えます。このロールはウォーターフォールのプロジェクトとアジャイルなプロジェクトで類似しています。

アジャイルなチームは、これらのロールが確実に実行されていることを重視し、誰が行っているか、誰が責任を持つかについてはあまり重視しません。テスト担当者と他のチームメンバに境界はほとんどなく、チームが確実に約束を果たすことに力を注ぎます。開発中のソフトウェアが高い品質であり、各メンバがこの目的を果たすためにできるあらゆる方法で貢献しているようなチームであることが重要です。アジャイルなチームでは、テスト担当者はすべてのチームメンバからのサポートを求めることができ、逆に自分は他のメンバに対して、テストのスキル向上のために支援します。このアプローチにより、チーム全員が全力を挙げて高い品質のアプリケーションを提供できるようになります。

12.9 参考文献

- Kent Beck: "*Test-Driven Development: By Example*", Addison-Wesley Professional, 2002.
  (邦訳は『テスト駆動開発入門』長瀬 嘉秀 監訳、株式会社テクノロジックアート訳、ピアソン・エデュケーション)
- James Bach: "*Exploratory Testing Explained*" (http://www.satisfice.com/articles/et-article.pdf), v.1.3 4/16/03, ©2002-2003.

13章
実践的な性能テスト

James Bull ◎ QAコンサルタント

性能の悪いソフトウェアを使うことで生活が楽になることはありません。むしろ人々はいらだち、組織の活動は妨げられます。そのソフトウェアがどの程度の機能を提供していたとしても、知覚品質†に強い悪影響を及ぼします。あなたの会社が開発したソフトウェアでひどい経験をした顧客は、次のバージョンで改善されるのを待ったりしません。他社のソフトウェアを買うことで意思表示をするでしょう。

速くて信頼性があってスケーラビリティが高いシステムと、そうでないシステムのどちらを選ぶかは明らかです。性能テストという言葉は一般的に、性能だけでなくスケーラビリティと信頼性も含むものとして使われます。なぜなら、これらは同じツールで同時にテストされることが多いからです。このエッセイでは、これらの特性（筆者はしばしばまとめて性能と呼びます）を備えた最終製品を確実に作る方法について説明します。

13.1 性能テストとは何か

前述のソフトウェアの特性は単によいものであるだけでなく、お金を払う価値があるという点に、今のところ異論はないでしょう。ここでの問題は、性能テストを特性全体に適合させるには何が必要かということです。適切な性能を持ったソフトウェアを開発するという目標に、性能テストをどのように活用すればよいのでしょうか。

性能テストでは、性能に関する要求を満たした製品を確実にリリースできるよう、必要な作業をすべて行わなければいけません。重要な要素は4つあり、すなわち要

† 訳注：自社の商品やサービスと競合他社を比べたときに、顧客が優位性を感じた相対的品質。

求、製品の性能データ、コミュニケーション、プロセスです。

これらの要素のどれかが欠ければ、性能テストの効果は著しく下がります。テストのみを行ったのでは、どのくらいの速度で動作すると予想されているかがわからないので、実際あまりよいとは言えません。そのため、ある程度の要求収集も行う必要がありますし、また、テスト結果を伝えなければ、問題があることに誰も気づかず、対策をとることができません。要求を確立して、製品をテストして、結果を伝えたとしても、まだ終わりではありません。計画に性能の改善を行う余地を残さなかった場合や、報告結果に基づいて改善作業を計画するようにプロセスを定義しなかった場合、実際に出荷されるソフトウェアに与える効果はきわめて小さくなります。この最後のケースでは、かなりのお金を使いながら、性能に関してどのくらい失敗または成功したかをとてもよく知るだけで終わることになります。

必要なのは単によく知ることではなく、得た情報を開発中のソフトウェアに反映させて、要求された性能を満たすか、最悪でも満たした状態に近づけることです。4つの要素について、それぞれ詳細に説明しましょう。

13.2 要求収集

要求収集は過小評価され、見落とされることが多くあります。ここでは、測定項目、必要な性能を知る方法、実際に役立って逆効果にならないような数値を見つけ出す方法について説明してみましょう。

13.2.1 何を測定するのか

性能に関する重要な測定項目は、最大スループットと一定のスループットにおけるレスポンスタイムです。多数の異なるスループット量でのレスポンスタイムを測定して、システム負荷がどのようにレスポンスタイムに影響するかを調べるとよいでしょう。レスポンスタイムに対する厳密な目標がある場合、その目標値を維持しながら達成できるスループットは最大値より著しく低くなる可能性があります。レスポンスタイムが受け入れ基準を満たした状態でのスループットと、要求のスループットで実行した状態でのレスポンスタイムを測定する必要があります。

スケーラビリティについては、データセット容量と接続ユーザ数またはシステムが稼働中のハードウェアの変化に応じて、初期の性能測定値がいかに変わるかということが重要な測定項目です。

信頼性については、異常な高負荷がかかったときにシステムが正確に機能し続けるか、またシステムが長時間にわたり稼動しているときに正確に機能し続けるかと

いうことが重要な測定項目です。

13.2.2 どのように数値を設定するか

システムで達成する必要のあるスループットを知るには、何人のユーザがシステムを利用して、その利用パターンがどのようなものになる予定かを知る必要があります。つまり、ユーザは特定の機能をどのような頻度で実行し、どのくらいの速さで完了させる必要があるのでしょうか。

これらの情報は対象業務にかかわる誰かから得られるはずです。その人には、これらの情報が定期的に必要になることを認識してもらい、最小限の手間で情報を集めることのできるプロセスを確立すべきです。

必要な情報が確実に得られる状態にした上で、対象業務を支援するために必要な数値を適時に得られるプロセスを確立する必要があります。これらの数値を定期的に計算しなければ、もはや的外れになった目標に向かって作業をする羽目になりかねません。

現時点で要求されているスループットを決定すれば、レスポンスタイムについて考えることができます。ユーザインターフェイス（UI）を考える際、UIが1秒の何分の1かで反応する必要があるから、UIが実行する機能もこの時間内に完了しなければいけないという思考の落とし穴に陥りやすいものです。UIは、ユーザにリクエストが処理中であることを知らせる表示をすぐに行うべきです。実行中の機能に依存していないアプリケーションの残りの機能は、利用可能のままであるべきです。

レスポンスタイムの目標は主にUIのためのものでなければならず、また低くなければいけません。どの特定の機能もその時間内に完了するはずだという予想を含んだ目標であってはいけません。

不明な点がある場合のために、要求収集の簡単な実例を示します。

13.2.3 要求収集を通常のソフトウェア開発プロセスにどのように適合させるのか

理想を言えば、その週の性能テストに関する要求を決定するミーティングには、プロジェクトマネージャ、性能の利害関係者、上級開発者、性能テスト担当者が参加すべきです。開発者は、明らかに満たせないと思われる要求や非現実的な要求が検討されていないか指摘するために参加する必要があります。性能の利害関係者は、対象業務についての情報と知識を提供して、要求決定を助けるために参加します。プロジェクトマネージャは何が決定されているかを把握して指示を与える必要がありますし、性能テスト担当者は当然、自分がどのような要求に対してテストするの

かを把握するために出席する必要があります。

次に、これらについて話し合う相手を決める必要があります。対象業務における性能要求決定の責任者と連絡をとれるようにしておくことがとても重要です。このことにより、顧客も開発者も、性能要求決定の責任者が望んでいることを確実に知ることができます。議論に上った要求を必ず達成しなければいけないものと考えるべきではありません。他のすべての要求と同様に、何が達成可能かについて顧客と対話する出発点にしなければいけません。

要求が決まれば、これらの要求が満たされていることを確認する方法について合意して、他の作業と同様に、これらのテストを準備する作業を見積って計画することができます。

13.2.4 開発者も性能テストによって要求を持たないのか

開発者の要求はさまざまですが、なぜ要求を持つかというと、あるコードに修正が必要になった場合、そのとき起こっていたことについて追加情報が必要になるためです。要求はコードプロファイラ出力からスレッドダンプや単なる追加ログに至るまで、広範囲にわたります。アプリケーションサーバと比較したデータベースの負荷状況やピーク負荷時でのネットワークの混み具合を知りたい場合もあります。

テストの前から準備してこれらの質問すべてに答えようとすると、かなりの作業量になるため割に合わないでしょう。それでも、あるコードに修正が必要になった場合、開発者が効果的に問題を解決するためにどのような情報が必要かを分析し、その情報について顧客と話し合った上で、情報提供を今後やるべき作業のリストに加えることができます。リストに加える際、情報提供の作業を今後の全テストで行うことが簡単かどうか、あるいは今回の特別なインシデントに対応する1回かぎりの作業となるのか考えることができます。

開発者の要求が要求ミーティングで持ち出された場合は、全員がその要求について知ることになります。したがって、後で作業の優先順位を付けたり計画を立てたりする際に、全員が要求を考慮に入れることができます。最終的に、行われる性能テストによって顧客と開発者の両者の要求が満たされます。そうなれば、顧客は開発中のソフトウェアを信頼します。また、開発者は発生する問題の原因を突き詰めて解消できます。

13.2.5 性能要求の利害関係者を見つけられない場合はどうなるか

性能要求の利害関係者を見つけられない場合、いくつかのリスクが生じます。まず、対象業務に適していないソフトウェアが完成し、プロジェクトは失敗するで

しょう。2番目に、顧客は自分たちに相談がなかったと思うので、実際にどうであるかにかかわらず、製品が対象業務に適していないという意見を持つでしょう。3番目のリスクとして次のようなことが考えられます。開発チームは、作業が顧客から発生したものではないため必要ないと考え、そのような作業を強要されることでチーム内に緊張が生じるかもしれません。このことは、性能に対するあなたの懸念が当たっているかどうかにかかわらず起こる可能性があって、必要な作業が終わらなかったり、逆に不要な作業に時間を浪費したりすることにつながります。

13.2.6 顧客があまり技術に詳しくなく、不可能と考えられることを要求する場合はどうなるか

顧客が製品に達成不可能な性能または必要以上に高額になる性能を要求するリスクがあります。このようなリスクを防ぐために、的を射た質問をして、対象業務で実際に必要な性能に焦点を当てた会話をする必要があります。

スループットに関する質問には次のようなものがあります。毎営業日にいくつのトランザクションが処理されますか。これらのトランザクションはどのように分布していますか。それらは均一に発生しますか、または波がありますか。毎週金曜日の午後に急激な増加がありますか、それともいつピークが来るか特別なパターンはないですか。

レスポンスタイムに関する質問には次のようなものがあります。UIのレスポンスタイムはこのシステムを使って行われる仕事の量にどのように影響しますか。実際の処理とUIを分離することはできますか。例えば、ユーザが入力したデータの処理に長い時間がかかるというシナリオを考えてみましょう。ユーザはデータ処理が終わるまで次のデータを入力できない状態を避けたいと望んでいます。つまり、データ処理がすぐに完了することを望むというよりも、UIと実際の処理が分離されて、システムがバックグラウンドで入力データを処理している一方で、ユーザがUIを使ってデータ入力を続けられるようになることを期待しているのです。

このように、実際に対象業務に貢献する性能に、顧客と開発者の考えが集中しているようにすれば、どうしても必要な性能と単にあれば快適に感じる性能とを区別することができます。本当に必要な性能が、プロジェクトの現在の制約では達成不可能であることや非常に高額になることが判明する場合もあります。もし要求分析が行われていなければ作業が続行され、顧客が続行するかどうかの選択をできなかった可能性がありますが、要求分析をすれば、そうなる前に決めることができます。

顧客が達成不可能な要求をすることの問題は、システムが対象業務での必要性を

十分に満たす性能を持っていたとしても、顧客が失望することです。顧客との話し合いには二重の効果があります。話し合いにより対象業務で実際に必要となる要求が明らかになり、それと同時に、顧客は要求を自覚することができます。このことにより、顧客は要求を満たすシステムができていれば満足します。顧客の期待が不必要な性能に向けられることがはるかに少なくなり、もし顧客が失望したとしたら、正当な理由がある可能性が高いのです。

13.2.7 なぜビジネスアナリストにもこれらの要求収集に参加させないのか

2つの理由から、ビジネスアナリストが参加する必要はありません。まず、機能要求の収集はすでに完了しているからです。次に、たとえビジネスアナリストが参加したとしても、開発者が参加する必要性はなくなりません。開発者は性能の問題を調査するために必要な要求を明確に説明し、同時に、検討されている要求によって問題が起こったり別の方法が必要になったりする可能性を伝えるために参加します。性能テスト担当者は、前述のような質問をして会話を進めることができるはずです。また、各機能のテストの難易度についても発言することもできます。それならば、ビジネスアナリストは他のことに時間を使ったほうがよいでしょう。

13.2.8 まとめ

対象業務の目標達成を支援するために、製品にどの程度の性能が必要であるかをよく知ることが要求収集の目的です。顧客にも参加してもらうのは、自身の業務について最も知識を持っているためです。顧客の参加により、的確な要求を収集することができます。また、どんな性能を要求しているのかを顧客がはっきりと自覚できるという利点もあるので、システムに対する顧客の期待をコントロールできます。

13.3 テストの実行

実行するテストの種類とその時期について簡単に説明します。

13.3.1 どのようなテストを繰り返し行うのか

頻繁に実行されるユーザアクション† はすべてテストする必要があります。これ

† 訳注：ユーザが画面を通してシステムにアクセスすること。性能テスト支援ツールに関連して使われる用語。

らのテストでは、スループット、エラー率、レスポンスタイムのメトリクスを記録しなければいけません。次に、これらのテストを再利用して、さらに複雑なテストを準備する必要があります。つまり、これらのテストを現実に近い組み合わせですべて同時に実行するテストが必要となります。こうして、製品の性能に関する十分な情報が得られます。

これが終わったら、前述の現実に沿ったテストを使い、異なる接続ユーザ数と異なるデータ容量でテストを実行して、スケーラビリティを調べます。可能であれば、異なるマシン数でシステムを稼動して、ハードウェアが追加されるにつれて性能がどのように変化するかを明らかにする必要もあります。これにより、スケーラビリティを把握するために必要な情報が得られます。

最後に、障害点を見つけ出すため、予想以上の負荷をシステムにかけるテストが必要になります。また、現実に近い接続ユーザ数の推移を再現したテストを基にして、実際の時間の流れより進行速度を速めてテストを行う必要があります。このテストを長期間にわたって行うことによって、システムの信頼性がわかるはずです。

13.3.2 いつテストすべきか

答えは明らかに「可能なかぎり頻繁に」です。当然、これには問題があります。本質的に、性能テストスイートの実行には長時間かかる傾向があります。特に信頼性テストは結果が出るようになるまで長時間行う必要があるので、日中は、すべてのビルドに対して総合的な性能テストスイートを実行する時間が十分とれません。とはいえ、開発者には迅速にフィードバックする必要がありますし、製品を総合的にテストする必要もあります。

1つの解決策として、単一の専用機を用意し、最新のビルドで一部の性能テストを継続的に実行する手法があります。このとき、テスト結果が前のビルドと著しく異なる場合はテストが失敗するように設定しなければいけません。これで得られるテスト結果は現実の性能を示すものではありませんが、早期警戒システムとなり、開発者の作業によって製品の動作に顕著な変化が生じたかどうかを知らせてくれるでしょう。

次に、すべてのテストスイートをできるかぎり頻繁に、完全な性能環境で実行していなければいけません。1日に数回行える場合もありますが、環境へのアクセスに制限があるなら、夜間に実行することがよい解決策になります。

信頼性テストには明らかに長時間かかり、さらに他のあらゆるテストと環境を共有しなければいけないことがよくあります。したがって、平日の間に信頼性テストを行うことは不可能であり、信頼性テスト専用の環境を別に作らないとすれば、毎

週末にテストすることが最大限できることです。

13.3.3 どのような環境でテストすべきか

できるかぎり、本番相当のシステムを構築するよう試みるべきです。本番システムがあまりに大規模でそれが不可能な場合は、本番環境の一部に似せた環境を構築して、本番の各サーバに最低限必要な台数を明らかにする必要があります。

専用の性能テスト環境を作ることができない場合、少しやりにくいものになります。機能テストチームと環境を共有しなければいけない場合は、夜間に性能テストスイートを実行するという方法があります。この場合、性能テストと機能テストのデータベースを分けて、テストプログラムを切り替えるためのスクリプトを作るのが得策です。このようにすれば、相反する2通りのシステム利用によって、それぞれの作業を妨げられることがありません。これは当然、デスクトップでテスト対象のアプリケーションが実行でき、性能テストを開発できることが前提となります。

夜間のテストで注意しなければいけない問題点は、ネットワークの状態が日中と違うことです。人々が働いていないため、ネットワークは日中に比べて混雑していませんが、バックアップなどの夜間のバッチプロセスがネットワークトラフィックに著しい影響を与えている可能性があります。性能テスト中にネットワークアクティビティが一気に生じることによって、おそらくテスト結果に影響が出るでしょう。性能テストとトラフィックを生じさせるジョブの両方が同時に開始されるようになっている場合、テスト結果に一貫性のある影響が出る可能性があり、その影響は違う時間帯にテストしないかぎり見つけられません。平日中に1回、テストスイートを実行するようにすれば、時間帯の違いによってテスト結果に顕著な差が生じるかどうかわかるでしょう。顕著な差がある場合、平日の平均的なネットワークトラフィックをシミュレートするか、または2つの時間帯で生じた差に一貫性があるかどうかを調べて、その結果を踏まえてテスト結果を考察することができます。

筆者の経験から言えば、1つのテスト環境で著しい競合がある場合、誰がいつそのシステムを使用して、どのデータベースと競合しているのかについて、よく整理する必要があります。著しい競合がある場合、テストがテスト環境で失敗するにもかかわらず、ローカル環境で成功することもあります。この場合、環境を性能データベース用にして、問題が解決するまでは他の品質保証（QA）でシステムを使用しない必要があります。これらの問題、そして環境を共有することによってテストの実行頻度が著しく制限される事実を考えれば、たとえ最も頻繁にテストを行う環境が本番と異なることになるとしても、可能なかぎり、専用のテスト環境を作るよう試みる価値はあります。

本番と異なる環境か、限られた時間帯しか利用できない本番相当の環境かを選ばなければいけなくなった場合は、両方利用することです。専用の環境では、邪魔されず定期的にテストを準備、実行することができます。これを継続的インテグレーションシステムに組み込んでいくこともできるかもしれません。本番相当の環境で得たテスト結果は、本番と異なるシステムでより定期的にテストして得たテスト結果と比較することができます。これにより、得られたテスト結果がシステムの最終的な性能にどのように関連しそうかがわかります。

13.3.4 性能の低いテスト機でのテスト結果をどのように本番環境に関連づけるか

テスト環境が本番環境と同じ構成でないという問題はよく起きます。テストシステムと本番がまったく異なる場合、違うハードウェアでシステムがどのように動作するか、ほとんど判断できないことを覚えておいてください。本番より小規模な環境を利用せざるをえない場合は、どのようにテストすればよいのでしょうか。ここでは、システムの代表的な構成を例にして、筆者の主張を詳しく説明します。

大規模のWebアプリケーションを例として考えてください。システムの基本的なアーキテクチャとして、数台のアプリケーションサーバ、Webサーバ、データベースサーバがあると考えられます。本番システムが数台のデータベースサーバ（非常に高性能）、データベースサーバ1台につき2台のアプリケーションサーバ（高性能）、アプリケーションサーバ1台につき4台のWebサーバ（低性能）で構成される場合は、以下のように環境を構築する方法があります。処理能力が本番の半分のデータベースサーバを1台買います。本番で使用するうちの1台と同じスペックのアプリケーションサーバを1台買って、Webサーバを1台買います。

アプリケーションサーバとデータベースサーバのマシンスペックを比べた場合の比は本番システムと同じですが、速度は約半分であるはずです。また、Webサーバのキャパシティも不十分です。

アプリケーションの性能は、アプリケーションサーバを直接実行することで調査できます。これによって、各アプリケーションサーバの達成できる数値がわかります。次に、1台のWebサーバ経由でテストします。このシナリオでは、テストによりWebサーバがボトルネックになっていることを明らかにするべきです。これにより、他のサーバから影響を受けていない状態でのWebサーバ1台の数値を得ることができます。この数値を使って、本番システム向けに議論している各種サーバの構成比が的確であるかを判断し、ある程度の自信を持って本番システムの性能を見積れるはずです。

マシンの台数が増えても、すべてのWebリクエストを1種類のアプリケーションサーバとデータベースサーバの構成で処理するのであれば、その直接的な結果としてスループットのみが増加すると覚えておかなければいけません。システムにかかる負荷レベルが変わらないと仮定すれば、レスポンスタイムは各マシンのCPUパワーまたはメモリを増強した場合にだけ改善します。CPUが高速になればより多くのリクエストをより迅速に処理でき、メモリが増えればより多くのデータをキャッシュできます。

これは当然、マシンが変わらないこと、CPUの製造元がすべて同じであること、すべて同じOSの同じバージョンであること、同じデータベースサーバ、Webサーバ、アプリケーションサーバの構成で稼動していることが前提となります。

マシンスペックや使用されているソフトウェアなどが本番から離れたものになればなるほど、システムの性能見積りが信頼できなくなると心に留めておくべきです。前述の例では、本番環境一式のコストの何分の1かでテスト環境を構築することができ、またシステムに関して見積った数値がかけ離れたものではないという多少の自信を持てるのではないでしょうか。Oracleの代わりにMySQL、またはJBossの代わりにWebSphereを使い、さらにマシンスペックが大幅に異なる場合、性能の変化を測定できることに変わりありませんが、見積りの値は疑わしくなるでしょう。

13.3.5 性能テストに適したデータベース容量はどのくらいか

性能テストを行うときには、次の点に注意することが大切です。テーブルからレコードを取り出す時間に、データベース容量がとても大きな影響を与えることがあります。テーブルに正しくインデックスが設定されていない場合、少しのデータしかなければ問題をはっきりとは確認できないかもしれませんが、本番データが数千レコードを超えるときには性能が著しく低下します。

本番データベースのコピーを入手してコードをテストできるよう、性能の利害関係者に相談すべきです。本番データベースを使うとき、データ保護の問題を意識すること、また利用するデータベース内の個人情報が適切に削除または変更されていることが重要です。

保存データの容量がどのように変化する見込みかについても、性能の利害関係者と話し合うべきです。ほぼ一定なのでしょうか、増える可能性があるのでしょうか。増える見込みの場合、徐々に増えるのでしょうか、急速に増えるのでしょうか。これらを把握することによって、現在よりかなりデータ量の多いデータベースでテストすべきかどうか決定できます。

よりデータ量の多いデータベースを入手する最適な方法は、安定性テストを利用

して新規のデータベースを作成することです。安定性テストの一環として、新しいユーザおよびトランザクションを作成しているはずです。システムが週末の間ずっと問題なく稼動すれば、今後使用できる相当大容量のデータセットができあがっているはずです。

13.3.6 サードパーティインターフェイスをどのように扱うか

システムに多くのサードパーティインターフェイスが存在する場合、これらのシステムを直接実行することはよい考えでありません。これには2つの理由があります。まず、サードパーティが喜んで性能テストに参加することはおそらくありません。次に、サードパーティがテスト環境を提供してくれたとしても、自分たちの管理下にないサードパーティに頼ることでテストの信頼性は低下します。

最適な方法は、このシステムがレスポンスを返す平均の速さをテストして調べて、その時間を待って一定のレスポンスを返すだけのモックまたはスタブを作成することです。レスポンスを直ちに返すように作ることは簡単かもしれません。しかし、その場合、アプリケーションサーバは潜在的にデータベースコネクションまたはネットワークコネクションを実際より速く取得する可能性があるので†、最終的なテスト結果に違いが生じることが考えられます。そのため、シナリオの現実味が薄れるでしょう。

13.3.7 何種類のテストケースが必要か

これはとても重要な問題です。多すぎても少なすぎても、誤った量のデータを使用すれば、テスト結果はひどく現実から離れたものになるからです。使用するテストケースが少なすぎる場合、関連する情報がすべてキャッシュされて、システムが実際より速く動作するテスト結果が出ます。使用するテストケースの種類が多すぎる場合、キャッシュがあふれることになって、システムは通常稼動で実際に動作するより遅くなるでしょう。

正しい数のテストケースを使用するためには、予想されるシステムの使用量について性能の利害関係者と話し合って、可能であれば、既存システムの使用量に関するログを入手する必要があります。例えば、アプリケーションの中で顧客情報を取得する場合、通常稼動中に取得される顧客数と同程度の顧客数の情報を取得するべきなのは明らかです。特定日にシステム内のレコードを5%取得するのであれば、

† 訳注：サードパーティのプログラムがコネクションを取得しない分、テスト対象のプログラムで使用できるコネクションに余裕ができる。

この顧客数をカバーするテストケースが必要です。

13.3.8 レスポンスタイムとスループットに数種類の測定方法をとるのはなぜか

一般に、アイドル状態から負荷を増やし始めてすぐにシステムのレスポンスタイムが低下するということはあまりありません。負荷が増え続けると、ある時点で単位時間あたりのトランザクション総数が増加し始めますが（つまりスループットが増加します）、代償としてレスポンスタイムも増加し始めます。サーバのキャパシティが最大に達すると、レスポンスタイムは顕著に増加し始めますが、スループットは初め一定を保ちます。最終的に、要求されている仕事量にマシンが耐え切れなくなるため、スループットは急速に低下します。この時点でレスポンスタイムが急激に増加して、システム全体が停止します。

入手したい情報がいくつかあります。まず知りたいのは、システムのスループットが最大になる時点です。その他に入手したい情報として、レスポンスタイムが目標値を満たし続けられる負荷レベル、達成可能な最小レスポンスタイム、初めに測定した最大スループットの80%および90%の状態におけるレスポンスタイムが挙げられます。

これらの情報を得ることができれば、アプリケーションサーバの各マシンとのコネクション数を制限して、システムの性能特性を要求収集で合意した範囲内に維持できます。レスポンスタイムは、負荷が最大に近づくと劇的に変化しやすくなりますが、80%または90%では著しく変化が少ないことがわかります。一定レベルの性能を保証しなければいけないのであれば、このことを念頭に置いておく価値はあります。

13.3.9 システムのすべての機能をテストする必要があるのか

システムの機能1つ1つをテストできることは稀です。それでも、使用頻度の高い機能を必ずテストすることが重要です。そのためには、システムの主な使用方法を特定し、各シナリオに合わせて異なるテストを準備する必要があります。

例えば、オンラインショッピングサイトの主な利用形態は、閲覧と購入であると考えられます。商品を購入しに来て、広範囲に閲覧した人全員が購入するとはかぎりませんし、全員が長時間にわたって閲覧するわけではありません。必要な作業は、閲覧と購入のためのスクリプトを1つずつ作ることです。これらのスクリプトを現実に近づけるために必要な情報は、1つのブラウザで閲覧される平均商品数、平均的な注文での商品数、典型的な1日を通してこのサイトで全ユーザが閲覧する全商

品の全体に占める割合です。

13.3.10 まとめ

性能テストのプロセスでは多くの疑問が生じえます。測定対象、頻度、スクリプトの量やデータ量などです。性能の利害関係者との定期的な話し合いは、このような疑問を明らかにして必要な情報を集められる場になると覚えておくとよいでしょう。また、いろいろなことの進行状況が性能テストの有効性に大きな影響を与えると思ったときは、プロジェクトマネージャ、性能の利害関係者と話す時間を作るべきです。

13.4 コミュニケーション

テスト結果を伝えることは重要です。ここでの問題は、具体的に何を伝えているかということです。テスト結果を伝えるというのは、単に生の数値データを報告するだけではありません。この程度の報告にとどめてしまえば、チームの全員は他にやるべきことがある中で、テスト結果の解析に時間を費やさなければいけなくなります。テスト結果の基礎解析と解釈を行って、要約を提供すれば、全員が楽になります。

したがって、合意された性能要求と現在の性能の観点で、テスト結果を解釈する必要があります。まず、性能が目標まであとどのくらいか、速いのか遅いのかについて報告する必要があります。次に、製品の性能が著しく変化していないか報告する必要があります。これは、性能目標を下回っているかどうかにかかわらず、広く知らせなければいけないでしょう。大きな機能が追加されて、製品の性能への影響が避けられず、対処法もほとんど考えられないという場合もありえます。しかし、データベースのインデックス不足など、単純で簡単に修正できる場合もあります。

13.4.1 知る必要があるのは誰か

3つの立場の人々にテスト結果を伝える必要があります。すなわち、開発者、プロジェクトマネージャ、顧客です。開発者とプロジェクトマネージャは、問題発生時にチームが迅速かつ適切に対応できるように、テストが行われたらすぐにテスト結果を知る必要があります。顧客には必ずしも、毎日些細な問題を知らせて面倒をかける必要はありません。些細な報告を続ければ、知らせるべき重要な問題があっても聞いてくれなくなるでしょう。とはいえ、報告をおろそかにしてはいけませんので、テスト結果を話し合う週次の定例ミーティングを開催するのが得策でしょう。

立場によって異なる情報に興味を持っていることも考慮に値します。顧客またはプロジェクトマネージャがハイレベルな観点での情報を求めているのに対して、開発者は生データ（一定期間内のレスポンス件数など）に興味があると考えられます。適切な人に適切な情報を提供することができれば、製品の状況を伝える作業ははるかに簡単になるでしょう。

13.4.2 つまり、単にレポートを作成すればよいということか

まったく違います。レポートを作成して回覧するだけでは、大半の人は読んでくれないので、伝えたい情報は失われてしまいます。作成したレポートはすべて、メッセージを伝える道具にすぎないので、それで済ませられるわけではありません。

人々に見てもらえるWebサイトに最新の性能テスト結果を掲載しておくと便利です。誰かのデスクまで結果を伝えに行った際、Webページを表示して、気づいたことを知らせることができます。残念ながら、テストレポートは大半の人にとって読みたいと心惹かれるものではありません。メッセージを確実に伝える唯一の方法は、デスクに行くか電話をかけて、レポートについて話し合うことです。

13.4.3 まとめ

目標は、要求が周知されて、テスト結果が出るたびに、受け入れ基準を満たしているか顧客に尋ねる必要のない状況にすることです。週に1回、顧客に会ってプロジェクトの現状を話すときには、テスト結果の例外や異常を指摘、説明できなければいけません。ソフトウェアのある部分の性能が特に悪いが、重大な問題ではないと判断された場合、顧客にはそこが遅くなっている理由、プロジェクトマネージャが最優先の問題と考えなかったこととその理由を説明できなければいけません。顧客がこの決定に同意しない場合は、プロジェクトマネージャと顧客がじっくりと話し合わなければいけません。

13.5 プロセス

性能テストはプロジェクトの最後になるまで行われないことがよくありますが、性能テストの効果に大きな影響が出ます。性能テストで最も重要なことは定期的に行うことです。プロジェクトの最後の数週間までテストを後回しにしてしまうと、短期間で多くの作業をこなさなければいけなくなります。スクリプトを作成することや製品から数値を得ることにほとんどの時間を費やすことになるでしょう。その結果、システムの速さについて大まかにはわかるが、それで十分かどうかはほとん

どわからず、必要な変更を行う時間がないという状況に陥ります。

性能テストはコーディングの初めから行うべきです。まだテストできるものは何もないかもしれませんが、それでもできることは多くあります。使用予定の技術について開発者と話し、適したツールを評価して、製品のテストに必要な機能を持つツールを見つけることができます。また、性能の利害関係者を特定して、彼らとともに要求収集プロセスを開始できます。

13.5.1 すべてをどのようにつなげるのか

ここから先は、1週間のサイクルで作業することができるようになります。週の初めに、性能の利害関係者とミーティングを行います。このミーティングでは、現在実装されている機能に関する要求について話し合います。またこの場では、予定しているテストについて話して、なぜそのテストによって、要求が満たされていることがわかるのか説明できます。顧客はこの時点で、追加のテストを依頼することもできます。週の残りには、最近完成した機能をテストしたり、自動テストを保守したり、テスト結果を確認したりできます。週の終わりに、性能の利害関係者と再び会います。この2回めのミーティングの目的は2つあります。まず、その週に準備したテストを見せることです。この新しいテストによって、製品が以前話し合った要求を満たしていることを実際に確認できるかどうか、顧客と議論することができます。ミーティングの2つめの目的は、既存の性能テストの最新のテスト結果をチェックすることです。

13.5.2 遅れを防ぐためにはどうすればよいか

1週間のサイクルで作業していけば、遅れているかどうかはすぐに明らかになるはずです。遅れを取り戻すには、性能テストに割り当てるリソースを追加するか、計画している作業の量を減らすかのどちらかの方法をとります。どちらを選ぶかは主に、プロジェクトでの性能要求の重要性によって決めるべきです。

遅れを防ぐため、顧客と合意したテストを基にして、必要な作業のリストを作る方法があります。そのリストを顧客と一緒に検討して、優先順位を付けることができます。そうすれば、1週間でできる最大限の作業をして、次に進めます。この方法でテストカバレッジに大きくむらができた場合は、性能テストにむらができないよう、もっと力を入れる必要があるのかもしれません。しかし、この方法をとっていれば、最も難易度が高く最も優先順位が低いテストをやめることで、テストカバレッジを十分に満たしたテストスイートを遅れずに準備できるということがわかるでしょう。

13.5.3 問題が発見されたとき、対策漏れがないようにするにはどうすればよいか

プロジェクトの初めに、性能の改善にどう取り組むか、プロジェクトマネージャと話し合うことが重要です。プロジェクトマネージャが性能改善への取り組み方に合意して、収集された要求を妥当と考えていることを確認する必要があります。また、プロジェクトマネージャが性能問題をバグとして報告されることを嫌がらず、問題が生じたら取り組んでくれるようにしておくことで、プロジェクトの終わりになって、取り組むべき既知の問題が多く残っているという状況に陥らないようにする必要があります。結局のところ、必要なときに性能を改善しないのであれば、現在の性能が受け入れ基準を満たしているか確認することにほとんど意味はありません。

13.6 まとめ

性能テストプロセスの主なメリットは、必要な性能と実際の性能が確実にわかること、そしてシステム全体に対して複数のテストを必ず行うことです。このメリットのおかげで、どんな問題が発生しても対処できる可能性が大幅に高まります。開発作業と並行して各機能をテストできれば、必要な変更を行う時間が生まれます。要求を収集していれば、変更が必要かどうかわかります。要求が顧客から発生したもので、顧客のビジネスプロセスやビジネスの実績値に基づいていれば、チーム全体が要求を信頼します。つまり、チームメンバは性能の改善がすべて価値のある作業であることを知っているので、嫌がらず改善を行います。

参考文献

Eric Evans: "*Domain-Driven Design: Tackling Complexity in the Heart of Software*", Addison-Wesley Professional, 2003.

Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts: "*Refactoring: Improving the Design of Existing Code*", Addison Wesley Longman, 1999.
(邦訳は『リファクタリング — プログラミングの体質改善テクニック』児玉公信＋友野 晶夫＋平澤 章＋梅澤 真史 翻訳、ピアソン・エデュケーション)

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.
(邦訳は『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 改訂版』本位田 真一＋吉田 和樹 監訳、ソフトバンククリエイティブ)

訳者あとがき

最初に翻訳のお話をいただいたのは、2008年3月がもう少しで終わる頃でした。翌年度はオブジェクトの広場が10周年を迎えることもあり、私たちの間に「何かやりたいね。」という雰囲気がありました。そこに翻訳のお話をいただき、しかも、あのThoughtWorks社のエッセイ集だということでにわかに活気づいたのを、つい昨日のことのように覚えています。まだ、原著の"*The ThoughtWorks Anthology*"が発売前であったにもかかわらず、一も二もなく引き受けたのは当然の成り行きだったと言えるでしょう。

ThoughtWorks社はとても注目されている企業で、いわばソフトウェア技術イノベーションの震源地です。プログラミング言語ギークがCTOをしているところなど、さすがはThoughtWorks社と言えるでしょう。本書にはさまざまなトピックがありますが、ThoughtWorkerたちは他にもいろいろなところで活躍しています。いくつか紹介しておきましょう。DI（Dependency Injection）では、Martin Fowler氏が名付け親というだけでなく、ThoughtWorks社が最初のDIコンテナであるPicoContainerを開発しました。BDD（Behavior-Driven Development）にも取り組んでいて、Java用のフレームワークであるJBehaveは、Dan North氏が開発しています。他にも、EIP（Enterprise Integration Patterns）やDBリファクタリングといった、現在注目を集める技術トピックの多くがThoughtWorkerから発信されています。また、Rubyにも積極的で、JRuby、XRubyのコミッタにもThoughtWorkerがいますし、Ruby on Railsを本気でSIに採用している企業としても有名です。

さて、本書はThoughtWorkerたちが寄稿したエッセイ集となっていて、それぞれに思うところを書いています。それだけでも十分に興味深い書籍なのですが、さらに、本当に現場で役立ちそうな、現場のエンジニアがワクワクできる内容となっています。おそらく、ThoughtWorks社が実際にワールドワイドでSIをしており、現

場で技術を使い倒しているからこその味や深みがあるからでしょう。余談ですが、本書にあるThoughtWorkerの肩書きには見慣れないものがあります。どうやら、ThoughtWorks社では各人が好きな肩書きを持てるようですね。

本書は問題提起から始まりますが、ソフトウェア開発の「ラストマイル」問題はとても衝撃的です。多少乱暴かもしれませんが、いわゆる「開発」、V字モデルの谷に近いところにおける大部分の問題は、数々のアジャイルプラクティスによりすでに解決済みであると言っているからです。事実、本書の後半のエッセイはV字モデルのより上のほう、または開発を支えるマネジメントのほうへ興味が移っています。ところで、日本に目を向ければ、開発の現場ではまだまだ「開発」の部分での問題が大きな関心事なのではないでしょうか。近い将来、日本の開発現場でも「ラストマイル」が一番の問題となるときが来るかと思います。本書の問題提起を真摯に受け止め、しっかりと準備しておきたいものです。

続けて、Martin Fowler氏のDSLに関するエッセイがあります。半ば冗談のような語り口から始まりますが、その内容はとても深いものになっています。本書では、エッセイということで十分に説明しきれていない箇所がありますし、また、執筆時点では明確に結論付けていない問題もあります。現在（翻訳時点）、Martin Fowler氏が新たにDSLの書籍を執筆しています（興味のある方は、http://martinfowler.com/dslwip/ を参照してください）。おそらく、このエッセイが新しい書籍への架け橋となるのでしょう。ここで示したアイデアをさらに深め、どう結論付けるのか、とても楽しみな一冊です。

さらに、言語やプログラミングに関するエッセイが続きますが、特に、多言語プログラミングのアイデアは、その呼び名こそなかったものの敏感なエンジニアの間ではすでに予感されていたものです。そのテーマをこのように正面から取り上げているのは大変に刺激的です。その後は、「ラストマイル」問題に対する答えとなるエッセイが続きます。ドメインアノテーションに関するエッセイはとても貴重だと言えるでしょう。現時点では、アノテーションに関する議論は要素技術的なものがほとんどだからです。ドメインモデルの1要素としてのアノテーションというアイデアはとても斬新なのですが、それが本来の使い方なのかもしれません。

なお、一部のエッセイはブログで公開されているアイデアをまとめたものになっています。議論の過程を知ることができますので、オンラインのリソースもぜひ探してみてください。

訳者についても軽く触れておきます。"*The ThoughtWorks Anthology*" が扱うトピックはさまざまですが、今回の翻訳メンバもまさに十人十色です。各メンバの興味は多岐にわたり、言語、ツール、開発環境、プロセス、モデリング、アジャイ

ルなどさまざまです。また、本業においても、業務系 Web システムを担当するメンバも、組み込みシステムを担当するメンバもいます。アーキテクトも、PM も、コンサルタントもいます。本書は、いわば“*The ThoughtWorks Anthology*”の翻訳アンソロジーといったところでしょうか。この点も楽しんでいただければうれしく思います。

最後に、本書の翻訳出版を支えてくださった皆様に感謝します。いつも暖かく見守り、励まし続けてくれた翻訳メンバの家族に感謝します。本当にありがとうございました。大変に忙しい中、質問メールに丁寧に対応してくださった原著者の皆様、そして ThoughtWorks 社の窓口となって質問を取り次いでくださった Darren Smith 氏に感謝します。また、技術や英語をいろいろと教えてくださった Tilmann Bruckhaus 氏、小井土亨氏、福井厚氏、そして同僚の森原剛氏に感謝します。また、翻訳の進捗が芳しくない中でも辛抱強く励ましてくださった株式会社オライリー・ジャパンの宮川直樹氏にお礼を申し上げます。

2008 年 10 月 14 日

オブジェクトの広場編集部

索引

記号

1クリックデプロイ ☞「継続的インテグレーション」を参照
　設計の改善 …… 67-68
　「ラストマイル」問題 …… 1-11

A

Alistair Cockburn …… 92
Antリファクタリング …… 151-186
　Antの呼び出し …… 179
　Antリファクタリングカタログ …… 153-184
　applyによるexecの置き換え …… 181-182
　build.xml内へのラッパースクリプトの取り込み …… 177-178
　build.xmlはリファクタリング可能か …… 153
　CI Publisherの利用 …… 182
　descriptionによるコメントの置き換え …… 166-167
　filesetによる多数のライブラリ定義の置き換え …… 171
　filtersfileの導入 …… 162-163
　IDを用いた要素の再利用 …… 173-174
　locationによるvalue属性の置き換え …… 176-177
　macrodefの抽出 …… 155-157
　tasknameの属性の追加 …… 178-180
　依存によるcallの置き換え …… 160-161
　実行時プロパティの移動 …… 171-173
　出力ディレクトリの親ディレクトリへの移動 …… 180-181
　宣言の導入 …… 159-160
　ターゲットの抽出 …… 157-159
　ターゲットのラッパービルドファイルへの移動 …… 165-166
　デプロイ用コードのimport先への分離 …… 167-168
　内部ターゲットの強制 …… 180
　プロパティによるリテラルの置き換え …… 161-162
　プロパティのターゲット外部への移動 …… 174-176
　プロパティファイルの導入 …… 164-165
　明確なターゲット名の導入 …… 182-183
　要素のantlibへの移動 …… 168-171
Apache Commonsプロジェクト …… 58

C

C …… 53
C++ …… 53

D

Dave Farley(11章の著者) …… 187-198
Dave Thomas …… 41
David Orchard …… 112
DSL
　～における変数 …… 19
　オブジェクトの利用 …… 20-28
　クロージャの利用 …… 28-29
　グローバル関数の利用 …… 17-20
　コンテキストの評価 …… 29-32
　動的受信 …… 39-41
　内部～ …… 13
　秘密基地の例 …… 13-16
　リテラルコレクション …… 32-38

E

else句 …… 71-72
Erik Doernenburg(9章の著者) …… 127-150
Erlang …… 46, 53
Evansの値オブジェクト …… 18
Expression Builder …… 23-26

F

filtersfileの導入 …… 162-164
Fortran …… 43, 53
Fred George …… 82, 86, 88

G

Getter、Setter、プロパティ …… 79
Greenspun形式 …… 37
Groovy …… 57-58
　～の方法でファイルを読む …… 57-58

H

Haskell ····· 45-46, 53
☞「Jaskell」を参照

I

Ian Robinson(8章の著者) ····· 103
IDを用いた要素の再利用 ····· 173-174
if-else構文 ····· 71-72, 159
if要素 ····· 160

J

James Bull(13章の著者) ····· 221-236
Jaskell ····· 60-63
Java ····· 45, 53
☞「Ant」「多言語プログラミング」を参照
isBlank() ····· 58-60
～のアノテーション ····· 131
～の起源 ····· 55
テスト ····· 63-65
ファイルを読む ····· 57-58
Jeff Bay(5章の著者) ····· 67-80
Jim Weirich ····· 32, 40
JMock ····· 63
JRuby ····· 58-60, 65
Julian Simpson(10章の著者) ····· 151-186

K

Kristan Vingrys(12章の著者) ····· 199

L

Lisp ····· 44, 53
～とRuby ····· 13

M

macrodefの抽出 ····· 155-157
Martin Fowler(2章の著者) ····· 13-42
Mary Poppendieck ····· 83
method_missingメソッド ····· 39-41
Michael Robinson(1章の著者) ····· 1
Mike Aguilar(まえがきの著者) ····· ix
Mike Royle ····· 134
Mingle ····· 65-66

N

Neal Ford(4章の著者) ····· 55-66
.NET ····· 129-131

O

OO言語 ····· 45

P

Prolog ····· 46, 53

R

Rebecca J. Parsons(3章の著者) ····· 43-54
Roy Singham(1章の著者) ····· 1
Ruby ····· 53
言語としての～ ····· 45
識別子 ····· 18
シンボル ····· 19
ダックタイピング ····· 49
匿名クラス(構造体) ····· 40
評価コンテキスト ····· 31
リテラルハッシュ ····· 34

S

Scala ····· 53
Schematron ····· 113-115
Scheme ····· 53
SQL ····· 46
Stelios Pantazopoulos(7章の著者) ····· 91

T

Tiffany Lentz(6章の著者) ····· 81-89
Tom Poppendieck ····· 83

W

WSLA(Web Service Level Agreement) ····· 122-123

あ行

アジャイルかウォーターフォールか ····· 199
回帰テスト ····· 209-210
課題管理 ····· 214-215
機能テスト ····· 204-205
性能テスト ····· 208-209
探索的テスト ····· 205-206
単体テスト ····· 204
ツール ····· 215
データの妥当性確認 ····· 207
テスト環境 ····· 211-214
テストの種類 ····· 203
テストのライフサイクル ····· 200-203, 201の図12-1
テストのロール ····· 216-219, 217の図12-2
統合テスト ····· 206-207
非機能テスト ····· 209
本番検証 ····· 210
ユーザ受け入れテスト ····· 207-208
レポートとメトリクス ····· 216
アセンブラ ····· 53
アノテーション
Javaの～と.NETの属性 ····· 129-131
ドメイン～ ····· 131-132
ドメイン～の実例 ····· 132-134
ドメイン駆動設計と～ ····· 127-128

ドメイン固有メタデータ …… 128-129
安心 …… 187
依存によるcallの置き換え …… 160-161
イテレーションマネージャ …… 81-89
イテレーション …… 87-88
顧客 …… 86-87
チーム …… 84-86
能力と責任 …… 82-86
プロジェクト環境 …… 88-89
プロジェクトバイタルサイン …… 94
プロジェクトマネージャ …… 84
役割 …… 81-82
インスタンス変数 …… 76-78
インターフェイス(プロバイダ契約) …… 116
インタプリタ型言語 …… 51
インデント …… 69-71
受け入れテスト …… 193-194
エンタープライズWebアプリケーション …… 199
回帰テスト …… 209-210
課題管理 …… 214-215
機能テスト …… 204-205
性能テスト …… 208-209
探索的テスト …… 205-206
単体テスト …… 204
ツール …… 215
データの妥当性確認 …… 207
テスト環境 …… 211-214
テストの種類 …… 203-204
テストのライフサイクル …… 200-203, 201の図12-1
テストのロール …… 216-219, 217の図12-2
統合テスト …… 206-207
非機能テスト …… 209
本番検証 …… 210
ユーザ受け入れテスト …… 207
レポートとメトリクス …… 216
エンティティ …… 76
オブジェクト …… 20-28
Expression Builder …… 23-26
クラスメソッドとメソッドチェイン …… 20-22
オブジェクト指向設計 …… 67-80
9つのルール …… 69
else句 …… 71-72
Getter、Setter、プロパティ …… 79
インスタンス変数 …… 76-78
エンティティ …… 76
コード1行に複数のドット …… 73-75
データのカプセル化 …… 79-80
名前の省略 …… 75-76
ファーストクラスコレクション …… 78
プリミティブ型と文字列型 …… 73
メソッドのインデント …… 69-71

か行

回帰テスト …… 209-210
開発実況 …… 99-101, 100の図7-4と図7-5
開発者の権利章典 …… 83
開発統合環境 …… 211-212
カウンタベースのテスト …… 8
課題(コンシューマ駆動契約) …… 123-124
課題管理 …… 214-215
型推論 …… 49-50
型特性 …… 49-50
稼働環境と設計 …… 9-10
カプセル化 …… 79
可変長引数メソッド …… 38
環境変数 …… 161
関数型言語 …… 44-46, 61
関数プログラミングとJaskell …… 60-63
キー …… 35, 36
機能テスト …… 204-205
クラス
インスタンス変数 …… 76-78
エンティティと～ …… 76
コレクションと～ …… 78
クラスメソッド …… 20-22
クロージャ …… 28-29
グローバル関数 …… 17-20
オブジェクトと～ …… 20-28
継続的インテグレーション …… 9-10, 187-198
受け入れテスト …… 193-194
エンドツーエンドのリリースシステム …… 188-189
～の自動化 …… 197-198, 198の図11-2
～のプラクティス …… 187-188
定義 …… 165-166
テストの段階 …… 196-197
デプロイ …… 194-196
ライフサイクル全体 …… 189, 190の図11-1
リリース候補 …… 191-193
ケーススタディ:リロイのトラック …… 134-149
データの分類 …… 136-138
転送 …… 135の図9-1
ドメインモデル …… 134-136
ナビゲーションのヒント …… 142-145
ユーザ …… 137の図9-2
言語 ☞「コンピュータ言語」「動的言語」を参照
構成(Configuration)オブジェクト …… 15
構造体 …… 40
コード
コーディング標準 …… 68
～品質特性 …… 67-68
コード=カタ …… 41
コード1行に複数のドット …… 73-75
互換性に影響する変更 …… 112-115
顧客とイテレーションマネージャ …… 86-87

顧客と要求収集 …… 225-226
コマンド・問い合わせの分離原則 …… 22
コミットテスト …… 189
コミュニケーション ☞「イテレーションマネージャ」を参照
　性能テストと～ …… 233-234
　「ラストマイル」問題と～ …… 5-6
コンシューマ駆動契約 …… 103
　課題 …… 123-124
　互換性に影響する変更 …… 112-115
　～の特性 …… 120-121
　コンシューマ契約 …… 118-120
　サービス指向アーキテクチャの概要と～ …… 103-104
　サービスの進化 …… 105-106
　サービスレベル合意と～ …… 122-123
　実装 …… 121-122
　スキーマのバージョニング …… 106-112
　プロバイダ契約 …… 115-118
　メリット …… 122
コンシューマ契約 …… 118-120
コンテキストの評価 …… 29-32
コンテキスト変数 …… 26
コンパイル型言語 …… 51
コンピュータ言語
　Erlang …… 46
　Fortran …… 44
　Haskell …… 45-46
　Java …… 45
　Lisp …… 44
　Ruby …… 45
　SQL …… 46
　型特性 …… 49-50
　～特性 …… 52-53
　～論争 …… 53-54
　実行時の振る舞い …… 50-51
　実装モデル …… 51-52
　種類の多種性 …… 47-52
　☞「動的言語」を参照
　多言語プログラミング …… 55-66
　パラダイム …… 48
　プラットフォームからの分離 …… 55-56

さ行

サードパーティインターフェイス …… 231
サービス指向アーキテクチャ …… 103
　互換性に影響する変更 …… 112-115
　～の概要 …… 103-104
　サービスの進化 …… 105-106
　スキーマのバージョニング …… 106-112
サービスレベル合意(コンシューマ駆動契約) …… 123
時間計測テスト …… 8
識別子 …… 19
システム統合環境 …… 212
実行時の振る舞い …… 50-51
実行時プロパティ …… 171-173
実装モデル …… 51-52
自動化
　継続的インテグレーションパイプラインと～ …… 197-198, 198の図11-2
　デプロイと～ …… 194-196
　非機能要求テストと～ …… 7-8
　「ラストマイル」問題と～ …… 6-7
集合ベース開発 …… 87
情報発信器 …… 92-93
スキーマのバージョニング …… 106-112
スコープバーンアップ …… 93-96, 93の図7-1
ステージング環境 …… 213
スループット …… 222-226
スレッド …… 60
正格評価型言語 …… 50
静的型付け言語 …… 49
静的言語 …… 44
性能テスト …… 208-209, 221-236
　コミュニケーション …… 233-234
　～に適したデータベース容量 …… 230-231
　～のプロセス …… 234-236
　定義 …… 221-222
　テスト機と本番環境 …… 229-230
　テストの実行 …… 226-233
　ミーティング …… 223-224
　要求収集 …… 222-226
設計 ☞「オブジェクト指向設計」「ドメインアノテーション」を参照
　～と稼働環境 …… 9-10
　～とコード品質特性 …… 67-68
宣言型言語 …… 46
宣言の導入 …… 159-160

た行

ターゲット
　～の抽出 …… 157-159
　内部～の強制 …… 180
　明確な～名の導入 …… 182-183
対話 ☞「コミュニケーション」を参照
多言語プログラミング …… 55-66
　Jaskell …… 60-63
　Javaのテスト …… 63-65
　～の概要 …… 56
　～の未来 …… 65-66
　パラメータが空白かどうかの判定 …… 58-60
　ファイルを読む …… 57-58
ダックタイピング …… 49
タプル …… 62

探索的テスト …… 209-210
単体テスト …… 204
チーム知覚 …… 101, 101の図7-6
チームの許容作業量 …… 85
遅延評価型言語 …… 46, 50
逐次型言語 …… 50
逐次的な文脈 …… 17
チューリング完全 …… 47
直近のバイタルサイン
☞「プロジェクトバイタルサイン」を参照
ツール(アジャイルかウォーターフォールか) …… 215
強い型付け …… 49
提供品質 …… 96-97, 96の図7-2
データのカプセル化 …… 79-80
データの妥当性確認 …… 207
テクニカルアーキテクチャ部会(W3C TAG) …… 107
テスト
Java …… 63-65
受け入れ～ …… 193
回帰～ …… 209-210
環境 …… 211-214
機能～ …… 204-205
継続的インテグレーションと～ …… 196-197
コミット～ …… 192
自動化と～ …… 7
性能～ …… 208-209, 226-233
探索的～ …… 205-206
データの妥当性確認 …… 207
～ケースの必要数 …… 231-232
～と本番環境 …… 229-230
～に適したデータベース容量 …… 230
～の並行実行 …… 9
～のライフサイクル(アジャイルかウォーターフォールか) …… 200-203, 201の図12-1
統合～ …… 206-207
バグカウント …… 96-97, 96の図7-2
非機能～ …… 209
非機能要求～ …… 7-8
プレイバック～ …… 7
本番検証 …… 210
ユーザ受け入れ～ …… 207-208
「ラストマイル」問題と～ …… 7-8
テスト記述 …… 218
テスト駆動開発(TDD) …… 203
テスト駆動設計(TDD) …… 9
テスト分析 …… 217
手続き型言語 …… 44
デプロイ …… 194-196
デメテルの法則 …… 73-75
統合開発環境(IDE) …… 52
統合テスト …… 206-207
動的型付け …… 49
動的型付け言語 …… 44-45
動的言語
～としてのLisp …… 44-45
～と未定義のメソッド …… 39-41
～とメソッド呼び出し …… 39
動的受信 …… 39-41
ドメインアノテーション …… 127-150
Javaのアノテーションと.NETの属性 …… 129-131
ケーススタディ:リロイのトラック …… 134-149
データの分類 …… 136-138
ドメインモデル …… 134-136
ナビゲーションのヒント …… 142-145
配送 …… 135の図9-1
ユーザ …… 137の図9-2
設計と～ …… 127-128
～の実例 …… 132-134
～の特徴 …… 131-132
ドメイン固有メタデータ …… 128-129
ドメイン固有言語 ☞「DSL」を参照

な行

名前と値のペア …… 34
名前の省略 …… 75-76

は行

バージョニング戦略(W3Cテクニカルアーキテクチャ部会の) …… 107
バージョン管理システム …… 191
バージョンのないソフトウェア …… 10-11
バイナリの管理 …… 189-191
バグカウント …… 96-97, 96の図7-2
パラダイム …… 48
パラメータとメソッドチェイン …… 26-28
非機能テスト …… 209
非機能要求
～テストの自動化 …… 7-8
ビジネス価値
～とアジャイルプロセス …… 10-11
「ラストマイル」問題と～ …… 1-4
ビジネスソフトウェア ☞「ラストマイル問題」「エンタープライズWebアプリケーション」を参照
ビルドパイプライン …… 189, 190の図11-1
ファーストクラスコレクション …… 78
フィンガーチャート …… 86
不適切なところにぶら下がったメソッド …… 59
プリミティブ型 …… 73
プレイバックテスト …… 7
プログラミング言語 ☞「コンピュータ言語」を参照
プロジェクトトラッキングツール …… 65-66
プロジェクトバイタルサイン …… 91
開発実況 …… 99-100, 100の図7-4と図7-5
情報発信器 …… 92-93

スコープバーンアップ …… 93-96, 93の図7-1
チーム知覚 …… 101-102, 101の図7-6
定義 …… 91-92
提供品質 …… 96-97, 96の図7-2
プロジェクトヘルス …… 92
予算バーンダウン …… 98-99, 98の図7-3
プロセス(性能テスト) …… 234-236
プロバイダ契約 …… 115-118
～の特性 …… 118
プロパティのターゲット外部への移動 …… 174-176
分解 …… 76-78
並列型言語 …… 50
変数
DSLにおける～ …… 19
インスタンス～ …… 76-78
環境～ …… 161
コンテキスト～ …… 17
～と識別子 …… 19
ボトルネック …… 94
ポリモフィズム …… 62
本番環境 …… 213-214
本番検証 …… 210

ま行

マップ …… 34
ミーティング(イテレーションマネージャ) …… 87-88
ミーティング(性能テスト) …… 223-224
ミックスイン …… 62
命令型言語 …… 44
メソッド
可変長引数～ …… 38
クラス～ …… 20-22
不適切なところにぶら下がった～ …… 59
～のインデント …… 69-71
メソッドチェイン …… 20-22
～とパラメータ …… 26-28
メタデータ
.NETと～ …… 129
ドメイン固有～ …… 128
文字列型 …… 73
モックオブジェクト …… 63

や行

ユーザ受け入れテスト …… 207-208
要求収集 …… 222-226
予算バーンダウン …… 98-99, 98の図7-3

ら行

「ラストマイル」問題 …… 1-11
解決 …… 4-5
コミュニケーション …… 5-6
自動化 …… 6
設計と稼動環境 …… 9-10
バージョンのないソフトウェア …… 10-11
非機能要求テスト …… 7-8
ビジネス価値 …… 2-3
本質 …… 2-4
リーンソフトウェア開発—アジャイル開発を実践する22の方法(書籍) …… 83
リスト …… 34-38
リテラルコレクション(リストとマップ) …… 32-38
リテラルとプロパティ …… 161-162
リテラルハッシュ …… 34
リファクタリング …… 151-186
Antリファクタリングカタログ …… 153-184
applyによるexecの置き換え …… 181-182
build.xml内へのラッパースクリプトの取り込み …… 177-178
build.xmlはリファクタリング可能か …… 153
CI Publisherの利用 …… 182
descriptionによるコメントの置き換え …… 166-167
filesetによる多数のライブラリ定義の置き換え …… 171
filtersfileの導入 …… 162-163
IDを用いた要素の再利用 …… 173-174
locationによるvalue属性の置き換え …… 176-177
macrodefの抽出 …… 155-157
tasknameの属性の追加 …… 178-180
依存によるcallの置き換え …… 160-161
いつすべきか …… 152
環境変数 …… 161
実行時プロパティの移動 …… 171-173
出力ディレクトリの親ディレクトリへの移動 …… 180-181
宣言の導入 …… 159-160
ターゲットの抽出 …… 157-159
ターゲットのラッパービルドファイルへの移動 …… 165-166
定義 …… 151-152
デプロイ用コードのimport先への分離 …… 167-168
内部ターゲットの強制 …… 180
プロパティによるリテラルの置き換え …… 161-162
プロパティのターゲット外部への移動 …… 174-176
プロパティファイルの導入 …… 164-165
明確なターゲット名の導入 …… 182-183
要素のantlibへの移動 …… 168-171
リリース候補 …… 191-193
レガシーシステム
自動化 …… 7
バージョンのないソフトウェア …… 10
「ラストマイル」問題 …… 3
レポートとメトリクス …… 216
連想配列としてのタプル …… 62

著者紹介

ThoughtWorks Inc. (ソートワークス)

アジャイルコミュニティへの貢献という点で世界に知られるグローバルなコンサルティング企業。Martin Fowlerが所属する組織としても有名。従事するThoughtWorkerは、設計、アーキテクチャ、SOA、テスト、アジャイル方法論などの分野を代表するリーダー的な存在。本書のエッセイは、Roy Singham、Martin Fowler、Rebecca J. Parsons、Neal Ford、Jeff Bay、Michael Robinson、Tiffany Lentz、Stelios Pantazopoulos、Ian Robinson、Erik Doernenburg、Julian Simpson、Dave Farley、Kristan Vingrys、James Bullによって執筆された。

訳者紹介

株式会社オージス総研 オブジェクトの広場編集部

(かぶしきがいしゃオージスそうけん オブジェクトのひろばへんしゅうぶ)

株式会社オージス総研のコミュニティで、毎月オンラインマガジンを公開している。オブジェクト指向を中心にソフトウェアに関するトピックを幅広く扱っており、各メンバが興味のあること、業務を通じて学んだことなどを記事にしている。慢性的な記事不足の中、なんだかんだで10周年を迎えた。継続は力なり。ちなみに「編集部」にはそれらしい肩書き以上の意味はない。

http://www.ogis-ri.co.jp/otc/hiroba/index.html

井藤 晶子 (いとう あきこ)　12章の翻訳を担当

株式会社オージス総研アドバンストモデリングソリューション部所属。2003年に株式会社オージス総研入社。これまで、金融系基幹システムの要求定義支援やデータモデリング、オブジェクト指向データベース製品の技術支援、Java EEフレームワークの開発・保守の支援などに携わる。米国の製品開発元との英語によるコミュニケーションや英語ドキュメントの翻訳など、日々の業務で英語を利用。「作る」よりも「伝える」業務に多く従事してきており、難しいことをいかにわかりやすく正確に伝えるかにいつも頭を悩ませている。「静かに熱く」がモットー。著訳書に『オブジェクトデザイン』(共訳、翔泳社)がある。

大西 洋平 (おおにし ようへい)　2章の翻訳を担当

株式会社オージス総研組み込みソリューション部所属。2007年に入社以来、メーカーの組み込みソフトウェア開発の現場で、ソフトウェアプロダクトライン開発の導入支援に従事している。ドメイン開発を行う傍ら、現在は製品開発に携わる開発者向けの文書作成を行い、関心の先は「製品開発者をいかに楽にするか」に向かっている。開発者の開発生産性を向上させるために、どのような環境や手法を用意すべきか日夜模索している。モットーは「人間の力を信じる」。

大村 伸吾（おおむら しんご） 6、7章の翻訳を担当
株式会社オージス総研アドバンストモデリングソリューション部所属。大学院博士後期課程に在学しながら、株式会社オージス総研で働く、オブジェクト指向やモデリングが大好きな28歳。大学院での研究とはまったく畑の違うソフトウェア業界で、日夜業務モデリングのコンサルティングに奮闘中。「ソフトウェアは人が人のために作る」という言葉も好きで、プロジェクトファシリテーション、リーダーシップ、組織作り、コミュニケーション論などにも興味あり。技術的なところでは、最近、関数型プログラミングの魅力に取り付かれ、Haskell（純粋！）、Scala（これから使えそう！）を勉強中。そこから導かれるようにλ計算や圏論などの理論も勉強中。好きな言葉は「無知の知」。

岡本 和己（おかもと かずみ） 4章の翻訳を担当
株式会社オージス総研エンタープライズソリューション部に所属。1998年に株式会社オージス総研入社。入社以来、組み込みから金融系システムの開発、オブジェクト指向の導入まで幅広くシステム開発に従事している。プログラミング言語が好きで、好きな言語はRubyとC++。最近の興味は、関数型言語と機械学習。夢は、人に自慢できるシステムを構築すること。著訳書に『コードで学ぶアジャイル開発 — 実践XPエクストリームプログラミング』（共著、アスキー）がある。

奥垣内 喬（おくがうち たかし） 8章の翻訳を担当
株式会社オージス総研技術部に所属。2006年に入社。中学生の頃、周囲の電気弦楽器デビューの波に抗うように電子鍵盤楽器にはまり、MIDIの仕様を学ぶ過程でプログラミングに興味を持った人文科学系の趣味プログラマ。並列仮想化によるサーバ仮想化やネットワーク設計といったインフラ構築から、オープンソース製品を中心とした各種SOA関連製品のプリセールス、マーケティング、そしてUMLモデリングツールの技術開発など、数々の業務に広く浅く従事する。

菅野 洋史（かんの ようじ） 10章の翻訳を担当
株式会社オージス総研エンタープライズソリューション部に所属。1998年の入社以来、モデリングツールの製品コンサル、オブジェクト指向教育、システム開発、アーキテクチャコンサル等に従事している。オブジェクト指向とかモデリングとか抽象的なものは好き。ごちゃごちゃした雑多な知識は、、、もっと好き。好きな言葉は「人間賛歌は『勇気』の賛歌ッ！」。著訳書に『コードで学ぶアジャイル開発 — 実践XPエクストリームプログラミング』（共著、アスキー）がある。

佐藤 匡剛（さとう ただよし） まえがき、1章の翻訳を担当
株式会社オージス総研アドバンストモデリングソリューション部所属。大学院でソフトウェア工学を学んだ後、ソフトウェアエンジニアとしてWebアプリケーション開

発に従事する。2006年に株式会社オージス総研入社後は、アーキテクトとして業務アプリケーションのアーキテクチャやフレームワークを企画、設計したり、顧客へ技術コンサルティングを行ったりしている。著訳書に『ソースコードリーディングから学ぶJavaの設計と実装』（技術評論社）、『Java基本API + Jakarta逆引きリファレンス』（共著、技術評論社）がある。雑誌ではJAVA PRESS、日経ソフトウェア、ITpro、@ITなどに寄稿している。日本ベジタリアン協会会員。
http://ameblo.jp/ouobpo/

田中 恒（たなか ひさし） 賞賛の声、11章、参考文献、著者紹介の翻訳を担当
株式会社オージス総研組み込みソリューション部所属。入社以来、Webシステムの開発やオブジェクト指向開発の教育に携わり、現在は組み込みシステムの製品企画、研究開発に従事している。もともと概念を識別するという考え方に惹かれてオブジェクト指向を学んだためか、モデリングがとにかく大好き。アイデアとも呼べないもやもやしたものが、ある視点から抽象化するときれいな構造になって整理されるとき、なんともいえない感動を覚える。座右の銘は「塞翁が馬」「明けない夜はない」。

辻 博靖（つじ ひろやす） 3章の翻訳を担当
株式会社オージス総研アドバンストモデリングソリューション部所属。入社以来、分散オブジェクト技術であるCORBA製品の技術支援や、UML・オブジェクト指向技術のトレーニングに携わる。今までJava、C++などのオブジェクト指向言語以外にはあまり興味がなかったが、担当章を読んでから、最近はやりの関数型プログラミングの世界にも興味を持つ。現在は、社内のScala勉強会に参加しているが、脳内がパラダイムシフトせず苦戦中。著訳書に『オブジェクトデザイン』（共訳、翔泳社）がある。

三好 由希子（みよし ゆきこ） 13章の翻訳を担当
昨年まで、翻訳コーディネーターおよびチェッカーを本業とする傍ら、主に医薬分野の翻訳を手掛ける。現在はフリーランス翻訳者（1級）。今回は分野が違ったが、専門用語などの指導を受けながら翻訳に参加。業界独特の表現に慣れるまで時間がかかったが、今後に活かせる経験となった。

村上 未来（むらかみ みく） 5章の翻訳を担当
株式会社オージス総研トヨタソリューション部所属。大学では英語学を専攻していた完全なる文系人間だが、システム開発というモノづくりの形に興味を持ち、IT業界に飛び込んだ。入社後は、数年にわたって業務アプリケーションの開発に従事し、システムの企画構想からリリースまでを担当した（開発言語は主にJava）。現在は新たに組み込みシステムの分野に挑戦中。モットーは、「明日できることは今日やらない、今日しかできないことをやる」。

山内 亨和 (やまうち みちたか)　13章の翻訳を担当

株式会社オージス総研アドバンストモデリングソリューション部所属。学生の頃にOMT法を学び、モデリング、オブジェクト指向に魅力を覚える。株式会社オージス総研入社後、いくつかのオブジェクト指向開発に携わった後、プロセスコンサルティング業務に転進する。現在は、主にユーザ企業向けに企画・調達から運用・保守までのプロセスを幅広く導入し、定着するまでを支援している。最近の興味は、プロセスの対極に位置するであろう「暗黙知」。「暗黙知」を他人に伝え、習得させるメソッドを開発できたらと、日々妄想しているが、残念ながらそう簡単にはいかない。著訳書に『その場でつかえるしっかり学べるUML2.0』(共著、秀和システム)がある。

山野 裕司 (やまの ゆうじ)　9章の翻訳を担当

OGIS International, Inc. 勤務。1990年代から、Unix、オブジェクト指向技術を使ったアプリケーション開発に携わる。2002年に株式会社オージス総研に入社し、金融系の大規模Java EEアプリケーションのアーキテクチャ設計やフレームワーク開発を行う。2008年からシリコンバレーの片隅でSOA関連技術の調査、研究開発を行っている。現在の興味は、OS、ソフトウェアアーキテクチャ、ソフトウェアパターン、大規模分散システム。美味しいお酒とご飯、旅、ファンキーな音楽が好き。著訳書に、『コードで学ぶアジャイル開発 — 実践XPエクストリームプログラミング』(共著、アスキー)、『ソフトウェアパターン入門 — 基礎から応用へ』(共著、ソフト・リサーチ・センター)がある。

ThoughtWorks アンソロジー
── アジャイルとオブジェクト指向によるソフトウェアイノベーション

2008年12月24日　初版第1刷発行

著　　者　ThoughtWorks Inc.（ソートワークス）
訳　　者　株式会社オージス総研 オブジェクトの広場編集部
　　（かぶしきがいしゃオージスそうけん オブジェクトのひろばへんしゅうぶ）
発 行 人　ティム・オライリー
制　　作　有限会社はるにれ
印刷・製本　株式会社平河工業社
発 行 所　株式会社オライリー・ジャパン
　　〒160-0002 東京都新宿区坂町26番地27 インテリジェントプラザビル 1F
　　TEL　(03)3356-5227
　　FAX　(03)3356-5263
　　電子メール　japan@oreilly.co.jp
発 売 元　株式会社オーム社
　　〒101-8460 東京都千代田区神田錦町 3-1
　　TEL　(03)3233-0641（代表）
　　FAX　(03)3233-3440

Printed in Japan（ISBN978-4-87311-389-0）
落丁、乱丁の際はお取り替えいたします。