6. テストケースの実装

|  |
| --- |
| Experience is a hard teacher because she gives the test first, the lesson afterwards.  経験とは厳しい教師である。それはまず試練を、次にレッスンを与えるからだ。  Vernon Sanders Law (アメリカのプロ野球選手) |

# なぜテストは難しいか

**テストの目的はこれだけではない**

**テストの主目的はこれではない**

テストとは、エラーを見つけようとしながらプログラムを実行し、品質に関する情報を収集するプロセスです。テストは、ソフトウェア開発にとても重要なタスクですが、品質の良いテストを設計するのはとても難しいものです。テストに関する名著に、グレンフォード・マイヤーズ氏が1979年に著した「ソフトウェア・テストの技法」という本があります。この冒頭にすぐれた問題がのっていますので、ここに引用しましょう[[1]](#footnote-1)。

**Ｇｏｏｄ**

|  |
| --- |
| あるプログラムは、カードから3つの整数を読み、それらを辺の長さとする三角形の形を診断して、不等辺三角形・二等辺三角形・正三角形のいずれかを画面に表示する。このプログラムをテストするテストケースを列挙せよ。 |

このテストケースの例は、(3, 3, 3) や (3, 4, 5) などのような3つの整数の組み合わせです。実際にいくつか書き出してみましょう。終わったら、表6 - 1にある項目を見ながら、あなたが書き出したテストケーススイート[[2]](#footnote-2)に点数をつけてください。各項目を満たすごとに+1点です。全部で14項目ありますから、全部で14満点です。

さて、どうでしたか？ この本によれば、経験豊かなプログラマの平均点はわずか7.8点だそうです。

**？？**

3つの整数の組み合わせは無限にありますから、そのすべての組み合わせをテストすることは物理的に不可能です。そこで、この無限にある組み合わせをさまざまな側面で有限個に分類し、これで無限にある組み合わせを代替しなければなりません。

これだけ単純なプログラムでも、これだけ多くのテストケースが必要になるのですから、規模の大きなソフトウェアを漏れなくテストするには膨大な数のテストケースが必要になることがわかります。これが、ソフトウェア・テストの本質的な難しさです。

表6 - 1 あなたのテストスイートの採点項目

|  |  |  |
| --- | --- | --- |
| # | 分類 | 例 |
| 1 | 不等辺三角形になるケースがあるか。 | (2, 3, 4) など |
| 2 | 正三角形になるケースがあるか。 | (1, 1, 1) など |
| 3 | 二等辺三角形になるケースがあるか。 | (3, 3, 4) など |
| 4 | #3について、すべての組み合わせがあるか。 | (3, 3, 4), (3, 4, 3), (4, 3, 3) など |
| 5 | 1つがゼロのケースがあるか。 | (0, 1, 1) など |
| 6 | 1つが負値のケースがあるか。 | (-1, 1, 1) など |
| 7 | 2つの和がほかの1つと等しい不正な三角形があるか。 | (1, 2, 3) など |
| 8 | #7を、すべての辺に対してテストするケースがあるか。 | (1, 2, 3), (1, 3, 2), (3, 1, 2) など |
| 9 | 2つの和がほかの1つより小さい不正な三角形があるか。 | (1, 2, 4) など |
| 10 | #9を、すべての辺に対してテストするケースがあるか。 | (1, 2, 4), (1, 4, 2), (4, 1, 2) など |
| 11 | すべてが0のケースがあるか。 | (0, 0, 0) |
| 12 | 整数でない値をもつケースがあるか。 | (0.5, 1, 1) など |
| 13 | 数の個数が3個でないケースがあるか。 | (1, 2) など |
| 14 | 各テストケースに、期待する結果を付記したか。 | (1, 1, 1) は正三角形、  (3, 3, 4) は二等辺三角形、など |

# ソフトウェアのテストはサンプリング

前節で示したように、実際に存在するテストケースの数は無限にあります。このすべてを完全に実施することは不可能ですから、境界値分析や同値分割などのテクニックや、また表6 - 1のようなさまざまな側面で、無限にあるテストケースを分類し、その中から代表的なケースを選ぶ必要があります。

**並びは逆にした方がいい**

冒頭にあげたプログラムをもう一度検討してみましょう。例えば、カードから正しく数字を読み取っていることを確認するために、開発者がカードの数字が (9, 9, 9) だったら画面に「メーテル」と表示するデバッグコードを書いたとします。ところが、そのデバッグ用コードを除去するのを忘れてしまい、コードレビューを通り抜けてそのままリポジトリにコミットされてしまいました。もちろん、(9, 9, 9) と入力されたら「正三角形」と表示するのが正しい仕様です。このバグは、表6 - 1の基準で満点をとるテストスイートでも、発見できない可能性があります。あるいは、(2011, 2, 16) と入力されたら「ハッピーバースデー俺」と表示するというデバッグコードが誤って埋め込まれてしまったとします。このバグをテストで発見することは、恐らくできないでしょう。

このようなバグを意図して埋め込むことも可能です。たとえば、パスワード入力画面を作った開発者が、パスワードに「secret」と入力すれば、どんなIDでもログインできるように作っておくこともできます。しかし、すべてのケース (この例では、入力可能なパスワードのすべて) をテストすることは不可能です。このように、テストとはサンプリングでしかありません。テストでソフトウェアの品質を完全に保証することはできないのです。

**そもそもそのつもりのテストケースではない**

また、テストだけでソフトウェアの品質を作ることもできません。美しいソフトウェアのアーキテクチャやバランスの良い開発プロセス、そしてプロジェクトメンバー全員の品質に対する態度が、ソフトウェアの品質を作ります。

# テストケースの管理

実際にテストを設計するときには、漏れなく包括的なケースを作成できるように、さまざまな側面に注目しなければなりません。例えば、機能的な側面 (ファンクショナルテスト) はもちろん、負荷 (ストレステスト)、性能 (パフォーマンステスト)、安全性 (セキュリティテスト) などを考慮する必要があります。まずは、必要な側面を見落とさないことが、包括的なテストケースを設計するポイントです。また、エンドレスなテスト作業を、どの範囲についてどこまでやるかを検討する必要があります。これらについてテスト設計仕様書を記述し、具体的にテストケースを書き出し、テストオートメーション (自動化されたテスト) を実装する、といった手順でテストを開発することになります。

しかし、テストの職人技にこれ以上立ち入ることは、本書の範囲を超えます。本書の関心事は、そのような膨大なテストケースをどうやって管理したらいいのか、ということです。さらに、管理すべきなのはテストケースだけではありません。各テストケースは、各ビルドに対して何度もテストしますから、その結果も管理しなければなりません。このほか、同じケースを条件を違えて複数回テストしたり、テストケースとオートメーションコード (テストスクリプト) との対応づけを管理したり、テストケースそのものの変更履歴を管理したりすることなどが必要なので、テストケースの管理に必要なコストはとても大きくなります。

**テストケースの定義と矛盾**

**・・・仕様書とテストケースとの関係**

**なんでこれだけで取り出しているの？**

# テストケース管理システム

**これでできるのは一部**

**？ 整理しないと分からない**

前節で述べたニーズに応えるのが、テストケース管理システムです。商用の製品では、Microsoft Test Managerや、Rational Test Suiteなどがあります。これらの企業は、テストケース管理ツール以外にもSCMやBTSなどを含む、ソフトウェア開発をトータルに支援するスイート製品をリリースしています。このような製品に含まれる各ツールは連携することも容易になっており、大変便利です。このほか、無償で利用できるテストケース管理ツールにはTestLinkやMaven2などがあります。最近は、これらのオープンなツールにも、ほかのSCMやBTSと連携して動作できるものが増えてきています。

テストケース管理システムは、多くのマニュアルケースを管理し、そのテストの実行結果をビルド番号ごとに集計してレポートします。また、オートメーションフレームワークと連携し、オートメーションを自動実行できるものもあります。さらに、テストケース管理システムには、5章で紹介したBTSと同じく、各ビルドのリリースノート (ビルドノート) を登録し、各ビルドの状態 (テスト可能とかテスト不可など) を追跡できるものがあります。そのようなテストケース管理ツールは、テスト結果と、テスト対象としたビルドの番号の間に簡単にリンクが張れるようになっており、便利です。

図6 - 1に、テストケースと、テスト対象のビルドのビルドノートが、テスト結果で関連づけられるイメージを示します。ビルドの状態については、★図2-9を参照してください。



図6 - 1 テストケースと、テスト対象のビルドを  
テスト結果で関連づけるイメージ[[3]](#footnote-3)

# テストケースとは

ここまで、何の断りもなくテストケースという語を使ってきました。テストケースとは、実行可能なかたちで仕様を詳細に記述したドキュメントのことです。ケースとは、場合とか事例といった意味ですから、テストケースとはテスト対象に対して無数にあるテスト手順を詳細に場合分けしたものであると捉えることができます。テストケースには、テスト手順と期待する結果が含まれます。テスト対象に対してテストケースを実行すると、パス (Pass; テストに合格する) かフェイル (Fail; テストに落ちる、失敗する) のどちらかの結果が得られます。

**これはテストスクリプト**

**いずれも違う**

**違う**

テストケースは、人間の手で実行するマニュアルテストケースと、コンピュータが直接実行できる自動化されたテスト (テストオートメーション) の2つに分類できます。次を見てください。これは、とても簡単なマニュアルテストケースドキュメントの例です。

1表6 - 2 マニュアルテストケースの書式 (例)

|  |
| --- |
| **TC#: 0025**  **Title: 会議室予約**  **Subsystem: \Project\Main**  **Setup:**  2010年12月12日の3時から5時まで、X会議室が予約されていない状態にしておく。  **Steps:**   1. 会議室予約システムにログインする。 2. X会議室を、2010月12日の3時から5時まで予約する。   **Expected:**  X会議室が予約されたことを表示する。  **Teardown:**  X会議室の予約をキャンセルしておく。 |

5章で説明したバグ報告票の項目と、とても良く似ていることに気がつくでしょう。これは、バグ報告票がテストケースとしての機能をもっているからです。違うのは、バグ報告票の項目数よりもテストケースの項目数が少ないことと、テストケースの項目にはテストの準備 (Setup) とテストの終了処理 (Teardown) が加わっていることだけです。この2つの項目は、山のようにあるテストケースを効率よく、正しく実行することができるようにとの配慮によるものです。テストの内容によっては、これらの項目を必ずしもテストケースに記載する必要はありませんし、逆に必要ならバグ報告票にもこれらの項目を記入すべきです。バグ報告票が、完全にテストケースの機能を包含していることが分かるでしょうか。ここにあげたテストケースの項目の説明については、5章のバグ報告票の同名の項目を参照してください。

**これは示した例に対してのみ言えること**

テストケースのテスト手順とバグ報告票の再現手順の違いがあるとすれば、テストケースの手順は包括的・網羅的な手順を準備することでバグを発見することに興味がありますが、バグ報告票の手順は特有の手順で当該のバグを再現することに関心があることです。テストケースで準備したテスト手順でバグを発見できるかどうかはわかりませんが、バグの再現手順ではバグを再現できることが求められます。より網羅的であることが求められるテストケースのテスト手順は、その記述方法に工夫を凝らせることがあります。これについては後述します。



**？？**

**おかしい**

図5 - 1 バグ報告票の3つの側面

# コラム ソフトウェア開発中の催し⑥ ポストモータム

|  |  |
| --- | --- |
| **コラム 開発中の催し⑥ ポストモータム**  ポストモータム (Postmortem) とは、医療の文脈では検死とか死体解剖という意味です。ソフトウェア開発においては、プロジェクトが終了した段階で行う反省会のことです。また、ソフトウェア開発以外のプロジェクトでも、事後検討会という意味で広く使われる語です。忌憚のない意見を述べることができるように、ユーザーを同席せずに行うのが普通です。ポストモータムでは、そのプロジェクトでうまくいったこと、うまくいかなかったことをまとめて、次のプロジェクトへのインプットとします。功を奏したことは習慣にして、同じ過ちは繰り返さないようにするのです。  **ユーザが入るやつも必要**  ポストモータムを上手に実施するのはなかなか難しいものです。愚痴をこぼし合って傷を舐め合うだけになるのはかわいい方で、下手をすると個人攻撃や罵り合いになることもまれにあります (筆者は、そんなポストモータムに何度か居合わせてしまったことがあります)。ここではポストモータムを上手に行うためのプラクティスをいくつか紹介しましょう。  **計測データの利用等、他にもポイントある**   * 概要を告知すること   きちんとポストモータムの開催を準備し、討論したい議題は前もって参加者にメールしておきましょう。チームメンバーが、これに対して言いたいことがあると感じたら、この告知によりポストモータムへの参加を促せたことにもなります。  **おかしい**   * 全てのプロジェクトメンバーが参加すること   立場によって、プロジェクトに対する人の感じ方はさまざまです。ある人には見えていなかった問題点が、ほかの人には見えていたこともあるでしょう。誰かに認知された問題は、漏れなく収集できるように配慮しましょう。   * 議事録を取り、チームの内外と共有すること   ポストモータムは、この結果を次に活かすために実施するものです。この成果を文書にしないと忘れてしまいますし、チーム外の人と共有できません。   * 批判しないこと   誰が何をしたか、は重要ではありません。何がなされたか、に注目しましょう。また、悪かったことだけでなく、良かったことについても話しましょう。   * 次のプロジェクトに活かせる内容にすること   **おかしい**  次のプロジェクトを改善するための具体的な提案をしましょう。過去のプロジェクトを批評するだけで終わるのではなく、将来のプロジェクトの当事者としてポストモータムに参加しましょう。  ところで、もしこのプロジェクトが1年間継続するものだったらどうでしょうか。プロジェクトが終わり、ポストモータムを実施する頃には、1年前のことなど忘れてしまっているかもしれません。そこで、プロジェクトが終わる前から、毎週もしくは毎月といった頻度でポストモータムを行うことで、現在のプロジェクトを継続して改善しようという取り組みがなされることがあります。この、ポストモータムよりもすぐれたプラクティスを、レトロスペクティブ (ふりかえり) といいます。プロジェクトがまだ終わっていないのに、「検死」をするのは縁起が悪いですからね。Retrospective (過去に向けられた、回顧的な) とは、prospective (予期された、将来の) の反語です。回顧とは、過去をみるという意味です。また、retroには懐古 (昔を懐かしむ) という意味があります。いわゆるレトロ趣味のレトロですね。   |  | | --- | | To: 開発プロジェクトチーム  From: 開発マネージャ  Date: 2011/02/22  Subject: ポストモータムしましょう  みなさんのおかげで、このプロジェクトを無事  終えることができました。お疲れさまでした。  最後に、いつものポストモータムやりましょう。  特に、今回のプロジェクトではネットワークの接続性の検証が  遅れた原因を追究できればと思います。  このほかにも、頭痛の種や良心の呵責があった方は  お知らせください。  日時: 2/23 13:00 - 14:00  場所: 第2会議室  よろしくお願いします。 |   このほか、実行中のソフトウェアがいきなりクラッシュしてお亡くなりになったとき、この原因調査の作業を指してポストモータムということもあります。文字通り、このソフトウェアの検死解剖をして死因を特定するという意味です。 |

# テストケースを実装する手順

品質の良いソフトウェアを開発するためには、いきなりプログラムの実装を始めてはいけません。最初にきちんと計画と設計を行い、機能仕様書を記述する必要があります。これはテストを実装するときも同様です。良いテストケースを書くためには、この話題を避けて通るわけにはいきません。そこで、ソフトウェア開発において記述しなければならない文書を順に紹介しましょう。図6 - 2を見てください。

**順番が違う**



**あまり良い体系ではない**

**= この体系じゃ駄目**

**テストに関していえば、少なくともテスト設計書とテスト仕様書はわけるべき**

図6 - 2 実行できないドキュメントを、実行可能な  
ドキュメントに変換するプロセス

これは、あるソフトウェアに機能を追加するとき、その仕様を段階的に詳細化しながら実行可能なものに変換していく流れを示しています[[4]](#footnote-4)。実行可能なビルドを手にいれるには、ソースコード (プログラム) だけを記述すればいいように見えるかもしれませんが、そうは問屋が卸しません。インプットが何もなければソースコードを記述できませんし、そのアウトプット (ソースコード) の正しさを検証する仕組みも必要です。この前者の役割を担うのが仕様書であり、後者の役割を担うのがテストケースです。テストケース (マニュアルテストケースとオートメーションコード) も、ソースコードと同じく仕様書をインプットとして書くこと、またソースコードと同じく実行可能なドキュメントであることに注目してください。

みなさんは、電話帳を開いたことがあるはずです。あの分厚い電話帳に含まれる膨大な電話番号データは、なぜ入力ミスがほとんどないのでしょうか。これは、2人の入力オペレータが同じ番号を入力する作業を別々に行っているのだそうです。その後で2人の入力結果を突き合わせれば、数字が違うところを入力ミスとして検出できます。2人のオペレータが、同じ位置の数字を同じように間違える可能性は非常に低いからです。みなさんがインターネットのサービスを使うとき、アカウント作成時にはパスワードを2回入力することが求められますが、それと同じですね。

ビルドとテストケースの関係も、これと非常によく似ています。ビルドとテストケースは、お互いを突き合わせて、その妥当性を検証し合うというわけです。もし開発初期の段階に実施したテストがすべて成功したら、危険信号です。テストに成功したということは、バグを発見することに失敗したということだからです。そんな時期にビルドの品質が良いということは考えられませんから、テストの品質が悪かったとか、テストケースの数が不十分だったことを疑うべきです。

**本質的に違う**

表6 - 3 テスト結果の評価

|  |  |  |
| --- | --- | --- |
|  | 最初に疑うべきは | 次に疑うべきは |
| テストにパスした | テストの品質が悪い？ | ビルドの品質が良い？ |
| テストにフェイルした | ビルドの品質が悪い？ | テストの品質も悪い？ |

次節から、図6 - 2に示した文書を簡単に説明しましょう。

# 1ページ仕様書の記述

ある機能をソフトウェアに追加するときは、最初にプロジェクトリードもしくは開発者がを書きます[[5]](#footnote-5)。クイック・スペックということもあります。ソフトウェアは、いきなり実装しようとしてもうまくいかないのと同様に、いきなり設計しようとしてもうまくいきません。まず、その機能がユーザーにどのような価値を提供するのか、あるいはしないのか、機能のを明確にする必要があります。そのために、新機能の仕様の概要を可能な限り短くまとめた仕様書を書きます。機能の規模にもよりますが、およそ数ページにまとめることができるはずです。このように、最初の仕様書には概要だけを記述すれば、その機能を少しずつ詳細化できます。この仕様書の項目としては、機能概要、この機能を追加する正当性 (追加するメリットやしない場合の影響)、設計とテストに対する要件、機能のゴールとノン・ゴールなどが適当です。この段階では、設計はしないことに注意してください。つまり、この機能をコンピュータで実現する手段は、ここでは考えません。ただし、次の段階へのインプットとして、設計に対する要件は書いておくと良いでしょう。例えば、こういう拡張性は考慮する必要はあるがああいう拡張性は考慮しないとか、これだけのパフォーマンスやスケーラビリティを確保する必要がある、などといったことです。この文書のドラフト版を書き終えたら、ほかの上級エンジニアにレビューを依頼します。必要なだけレビューサイクルを回してこの文書をコンプリートし、レビューが通ったら次の段階にすすみます。

**以下、少なくとも要求と仕様は区別しなきゃ**

ソフトウェアのドメインによっては、これをユーザーに持っていって提案することもあるでしょう。そのときは、ユースケース図を添付するなどして、1ページ仕様書をよりわかりやすく訴求力のあるものにします。また、見積もりについても言及の必要があるかもしれません。

**この検証じゃ弱い**

**こんなのりで特定するようなことではない**

**いずれも意味不明**

**こんなところでやる話ではない**

**概要と仕様は違う**

**微妙、恐らく×**

このようなドキュメント書きのタスクも、5章で説明したBTSに登録して管理することがありますが、BTSに仕様書を直接書き込むことはしません。BTSの中にソースコードを直接書くことはしないのと同じです[[6]](#footnote-6)。

# 機能設計仕様書の記述

1ページ仕様書のレビューが通れば、次にその実装を担当する開発者が機能設計仕様書を書きます。ソフトウェアのドメインによっても記述すべき項目が変わってきますし、少々本書の範囲を超えますので、ここでは機能仕様書に適切と思われる項目の一例を示すにとどめます。機能の概要と詳細、この機能に対する要件や制約、設計のゴールとノン・ゴール、設計の詳細とアーキテクチャ、他の機能との依存関係、実装 (コードコンプリート) に必要な工数の見積もり、考えられるリスクなどがこの項目として適切です。

**上と矛盾している**

**非常に良くない**

ただし、詳細すぎる設計に立ち入らないように注意してください。これには、設計仕様書を記述する対象の機能について、適切な粒度を選択することが大切です。細かい機能のそれぞれについて設計仕様書を起こすのではなく、関連した機能のグループについて1つの設計仕様書を起こすようにします。また、XPでは、紙上での設計をに避けており、リファクタリングを継続して行うことで紙上の設計を代替します。リファクタリングは、機能追加やバグの修正と同時には行えないことに注意してください。

機能設計仕様書がレビューにパスすれば、開発者はソースコードの記述を、QAはテスト設計仕様書の記述を開始できます。

# テスト設計仕様書の記述

テスト設計仕様書は、包括的で網羅的なテストケースを書くためのインプットになる文書です。これには、機能の概要、テストをする範囲としない範囲、テストに必要なリソース、テストの戦略、オートメーションコードの設計、テストケースの実装と実施に必要な工数の見積もりなどの項目が適当です。テストの戦略とは、どのようにテストケースを分類し、網羅性を確保するかということです。テスト設計仕様書のレビューには、機能設計仕様書を書いた開発者にも参加してもらうと良いでしょう。

**違う**

**テスト設計書、テスト戦略、テスト計画　・・・　の内容が混在している**

**冗長、良くない習慣**

# テストケースの記述

さて、ようやくテストケースの記述を開始できるところまでたどりつきました。テスト設計仕様書に基づいて、包括的かつ網羅的なテストケースを準備します。

マニュアルテストケースについては、すでに6-4節で説明していますが、テスト手順 (Steps) と期待するテスト結果 (Expected) の項目について補足しておきます。これらの項目は、マトリックスで記述できることがあります。例えば、冒頭の三角形を診断するプログラムをテストするテストケースは、表6 - 4のように記述できます。これだけで、表6 - 1の採点基準で9点をゲットできました。さらに各値の範囲を広くとれば、13点までいけます (表6 - 1の項目#13 数の個数が3個でないケースだけは、このマトリックスでカバーできません)。つまり、表6 - 4のようにケースを列挙できれば、表6 - 1のようにケースの分類を分析するのも容易になります。

**意味不明**

**∵ 現実的には網羅できないんでしょ**

表6 - 4 マトリックスを使った  
包括的なテストケースの例

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X=3:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Y=1 | Y=2 | Y=3 | Y=4 | Y=5 | | Z=1 | - | - | 二等辺 | - | - | | Z=2 | - | 二等辺 | 二等辺 | 不等辺 | - | | Z=3 | 二等辺 | 二等辺 | 正三角形 | 二等辺 | 二等辺 | | Z=4 | - | 不等辺 | 二等辺 | 二等辺 | 不等辺 | | Z=5 | - | - | 二等辺 | 不等辺 | 二等辺 |   X=4:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Y=1 | Y=2 | Y=3 | Y=4 | Y=5 | | Z=1 | - | - | - | 二等辺 | - | | Z=2 | - | - | 不等辺 | 二等辺 | 不等辺 | | Z=3 | - | 不等辺 | 二等辺 | 二等辺 | 不等辺 | | Z=4 | 二等辺 | 二等辺 | 二等辺 | 正三角形 | 二等辺 | | Z=5 | - | 不等辺 | 不等辺 | 二等辺 | 二等辺 |   X=5:  … |

テストケースは、このように漏れがないように準備することがとても重要です。このようなマトリックスは、包括的なテストデータを記述するのに大変適しています。これはMicrosoft Excelのようなスプレッドシートで記述し、テストケース管理ツールに添付するなどして管理します。

**×**

このほかにも、テスト対象のソフトウェアによって、さまざまなテストケース (テスト手順とテストデータ) の書式が使われます。みなさんも、包括的なテストをするためのテストケースの書式を工夫してみてください。

# コラム ソフトウェア開発のメタファ⑥ 車の運転

|  |
| --- |
| **コラム ソフトウェア開発のメタファ⑥ 車の運転**  ケント・ベック氏は、著書「XPエクストリーム・プログラミング入門」で、初めて車を運転したときのことを語っています。車をまっすぐに走らせようとしたが、ちょっとぼんやりしたときに車が車道からそれ、砂に乗り上げてしまったといいます。単に車をまっすぐ走らせようとするのではなく、フィードバックに基づいてハンドルを微調整することが、安全なドライブに必要だったのです。車を運転するときには、ハンドルの操作以外にも、ガスの残量や走行距離に応じて適切なアクションをとらなければなりません。このように、要件やテクノロジー、チームなどにおけるさまざまな変化に追随し続けることが、ケント・ベック氏らの提唱するExtreme Programmingというソフトウェア開発手法のパラダイムであることが、この著書に述べられています。ソフトウェア開発を駆動 (drive) するためには、正しいフィードバックを得て、適切にハンドルを切らねばならないのですね。  このほかにも、ソフトウェア開発と車の運転との類似性を示唆する文章が多くありますが、その中でも、ジェラルド・ワインバーグ氏のテストに関する著書「パーフェクトソフトウェア」に優れた示唆があります。みなさんのプロセスにテストフェーズはありますか？もしイエスとお考えなら、車の運転をするときにフロントガラスから外を見るフェーズはいつなのか、考えてみましょう。運転中は、ずっとフロントガラスを見ていますよね。ソフトウェア開発におけるテストとは、運転中にフロントガラスを見るのと同じく、フェーズではありません。テストは開発と並行して継続的に行うべきで、多くの人が「テストフェーズ」だと考えているのは「修正フェーズ」あるいは「安定化フェーズ」と呼ぶべきものです。  ところで、ビルドサーバの多くが「ダッシュボード」という機能をもっています。これは、ビルドにかかった時間や自動化されたテストの実施状況などを、ビルドごとにグラフでWebブラウザなどに表示する機能です。これはまさしく、ビルドの健康を追跡し、プロジェクトの状態を管理するためのダッシュボード (計器類) です。  **テストフェーズをそう使っているのが問題**  **分かりづらい** |

# オートメーションコードの記述

テストの自動化について考えてみましょう。テストを自動化するのは、リグレッションテストの工数を削減するための優れた方法です。テストを自動化すると、複数のビルドに対して同じテストを繰り返して実行できるようになります。つまり、テストを自動化する価値は、すべてのビルドに対して、いままで実施したテストを容易にリグレッションできるようになることにあります。開発期間が短いなどの理由で、何度も繰り返しては実施しないテストは、自動化する価値がありません。

といっても、どんな場合でも、記述されたテストケースを1回しか実施しないことはありません。仮にある機能をスムーズに実装でき、それについてバグを作り込むことがなかったとしても、その機能をテストするテストケースは、開発者がQAリリース候補ビルドで1回、QAリリース後にQAが1回、最終ビルド候補でリグレッションテストとしてもう1回と、これだけでもテストする機会が3回あります。

**おかしい**

しかし、3回しかテストを実施しないで済むなら、テストを自動化する価値はないかもしれません。テストを自動化することにも工数が掛かりますから、そのトレードオフを考慮したうえで、テストを自動化すべきか判断します。

また、手順にハードウェアの操作を含むものなど、そもそも自動化が不可能なテストケースもあります。必ずしも、すべてのテストを自動化する必要はないと心得た上で、少しずつ自動化されたテストを増やしていきましょう。自動化されたテストのメリットがあなたの腑に落ちれば、もっとテストの自動化に投資でき、ノウハウの蓄積もできるようになります。

テスト自動化ツールを導入したからといって、テストに必要な工数を削減できるとは限りません。すべてのビルドでリグレッションテストをするのですから、テストに掛かる工数は逆に増加する可能性もあります。テストの工数を削減するためにテストの自動化ツールを導入すると、失望するかもしれません。テスト自動化ツールとは、自動化とはいっても無人でテストを行えるようするためのものではないのです。しかし、テストツール導入によりテストの工数が増加したのなら、いままでが必要なテストをちゃんと実施していなかっただけ、ということかもしれません。統合テストの自動化は困難ですが、取り組んでみる価値は大いにあります。

# テストハーネス

**ドライバとスタッブについても説明した方がいい**

テストハーネスとは、テスト対象のビルドとテストオートメーションを接続するためのソフトウェアで、テスティングフレームワークとも呼ばれます。いわば、ビルドを組み上げるための足場の役割を果たすものです。建物を建築中のときは足場を取り付けますが、完成したら足場は取り外しますね。テストハーネスも最終的なソフトウェア製品には含まれませんが、その開発には必須のものです。Harnessには、馬や牛などの引き具という意味があります。図6 - 3には、盲導犬のハーネスのイメージを使って、ビルドとオートメーションの関係を示しました。オートメーションは、人の手に代わって、ハーネスを介してビルドを操作し、その動作が意図した結果になるかを確認します。



図6 - 3 ただ今、ビルドのテスト中…

テストを自動化するときにまず検討すべきことは、どのテストハーネスを使うか (使えるか) ということです。適切なハーネスがなければ、ハーネスを新規に開発する必要があるかもしれません。また、選択したテストハーネスを装着できるように、ビルドに少し手を加えなければならないこともあります。例えば、Webアプリケーションを開発するなら、HTTP経由以外の方法でもアプリケーションを操作して処理結果を取得する方法を用意しておけば、テストハーネスの開発が非常に楽になります。ビルドへのテストハーネスの接続のしやすさが、ビルドのtestability (テスタビリティ; テストのしやすさ) を決める大きな要因のひとつです。ビルドのテスタビリティは、その品質を大きく左右します。

# 単体テストと統合テスト

**定義、きわめて怪しい**

開発中のソフトウェアの一部だけをテストすることを、単体テスト (unit test) といいます。一般に、単体テストの実装と実施は、そのコードを書いた開発者が担当します。これに対し、ビルドマシンで統合ビルドしたものをテストすることを、統合テスト (integration test) といいます。統合テストの実装と実施は、QAが担当します。それぞれ、マニュアルケースとオートメーションの両方を使ってテストできますが、単体テストは自動化するのが比較的簡単なので、ふつう単体テストといえば自動化されたテストを指します。

**違う**

現在は、単体テストにはxUnitというオープンなテストハーネスが使いやすく、人気があります。ケント・ベック氏がSmalltalkというプログラミング言語で開発したSUnitが最初で、これをJava用に移植したJUnit, C++用のCppUnit, Microsoft .NET用のNUnitなどがあります。例えばJava言語で開発したソフトウェアをテストするには、JavaとJUnitを使ってオートメーションコード (テスト手順と期待する結果) を実装します。これをPCで実行すると、「テストに合格」(Pass) もしくは「テストに不合格」(Fail) が自動でわかるというわけです。このように、単体テストはテスト対象のビルドと同じプログラミング言語で開発します。その方が、テストハーネスの接続がずっと簡単だからです。単体テストのオートメーションのソースファイルは、テスト対象のユニット (サブシステムやコンポーネント、クラスなど) のソースファイルのディレクトリ位置に対して、どこに配置するのか決めておき、ビルドのたびに各ユニットの単体テストが自動実行されるようにします。先に単体テストを書いてコミットしておけば、各ユニットの実装の進捗を定期的なビルドで測れるようになります[[7]](#footnote-7)。

統合テストには、事実上標準といえるオープンなテストハーネスがありません。これは、ソフトウェアの種類によって接続できるハーネスが大きく異なるからです。テストハーネスによっては、独自のスクリプト言語でオートメーションコードを書く必要があります。また、人の手で操作した手順を自動的に記録し、そのログをテストスクリプトにできるようなものもあります。このような統合テストのためのテストハーネスは非常に数が多く、ここではとても紹介しきれません。Webアプリケーションのテストハーネスで人気があるものだけでも、Selenium, Watir, WatiN, HttpUnit, HtmlUnit, Canoo WebTestなどがあります。また、.NET環境にはMicrosoft Windows UI Automation APIというUIテスティングフレームワークがあり、高度なオートメーションが可能です。自動化した統合テストのオートメーションコードは、単体テストのように、そのテスト対象のソースと関連づけて分類することは難しいので、製品のソースコードとは別のディレクトリツリーに、人間にわかりやすい機能ごとのフォルダを作って、この中に管理します[[8]](#footnote-8)。また、必要なときに必要なテストケースを取り出してテスト結果を記録できるように、テストケース管理ツールにも、各機能をテストするオートメーションがどこにあるか、それぞれどのように実行するかなどを書いて管理します。

**ここはもうちょっと丁寧に説明した方が良い**

**例えばCOTS等の場合を除き、これらはUnitとはいわない**

**, → 、**

一般に、統合テストのオートメーションはQAが開発します。開発者と密に連携し、必要であればテストハーネスが接続しやすくなるように開発者にソースコードの修正を依頼したり、仕様を確認したりします。このため、統合テストの自動化を担当するQAには、開発者と同程度のプログラミングスキルや、当該のテストハーネスに関する専門性が求められます。オートメーションコードは、ソースコードと同じくリポジトリで管理しますから、SCMの操作やレビュープロセスについても、QAはよく理解しておかねばなりません。

**それと他のところにもあまりでてきていないようですが、**

**トレーサビリティ管理も**

**？？、当然構成管理すべきでしょう**

# テストケースの追加とバグ報告票

テストケースは、開発を通して、インクリメンタルに追加していく必要があります。ただし、テストケースの数は、必ずしもその品質が良いことを意味しません。例えば、冒頭に紹介した問題で、正三角形のケースばかり(1, 1, 1) から (10000, 10000, 10000) までをテストしても、バグを発見できる見込みはほとんどありません。バグを発見できる可能性の高いテストケースや、何度も繰り返しリグレッションテストをする価値のあるテストケースを追加していくことは、それほど易しいことではありません。

**問題なのはテストケース数ではなく、カバレッジ**

実は、テストケースを記述するために役立つ非常に重要なインプットが、ここまでに説明した手順のほかにもあります。それはバグ報告票の再現手順です。図6 - 4を見てください。バグ報告票がテストケースとしての側面を含むことは、すでに説明しました。まだテストケースという形でドキュメント化されていない再現手順 (テスト手順) によるバグを発見したら、バグ報告票を起票します。これは取りも直さず、次のビルドのためのテストケースを追加する作業にほかなりません。

**それはその通りですが、コンテキスト/局面が混乱している**

不思議なことですが、「プログラムのある部分にバグがまだ残っている確率は、すでにその部分で見つかったバグの数に比例する」という事実があります。バグは、プログラムのある部分に偏在する傾向があるのです。が、少し考えてみればその理由は明らかでしょう。見つかったバグを修正するために手を入れると、新しいバグを作り込んだり、リグレッションしたりする可能性も高くなってしまいます。また、そもそもその部分の設計が悪いということも考えられます。バグ報告票が、品質の高い (=バグを発見できる可能性が高い) テストケースであることがよくわかります。一度修正したバグでも、その再現手順はリグレッションテストのインプットとして大きな価値があります。



**×**

図6 - 4 テストケースを追加するサイクル

テストケースを追加するサイクルを図6 - 4に示します。実際には、バグ報告票に記述された再現手順をテストケース管理ツールに転記するのは面倒なので、転記せずにそのままBTSで管理することが多いでしょう。しかし、もしバグの再現手順を自動化できれば、そのタイミングでテストケース管理ツールに転記します。また、BTSとテストケース管理ツールには連携するものがあり、そのようなツールを選択すれば、転記をせずともバグ報告票とテストケース間に相互リンクをつけておくだけで済みます。



図6 - 5 テストケースの分類

表5 - 2 テストケースとバグ報告票の品質

|  |  |  |
| --- | --- | --- |
|  | テストケースの  テスト手順 | バグ報告票の  再現手順 |
| 良い手順 | バグを発見できる | バグを再現できる |
| 悪い手順 | バグを発見できない | バグを再現できない |

# まとめ

本章では、テストケースをインクリメンタルに追加する手順を説明しました。継続的にテストを追加して実施し続けることが、ビルドの健康を守るために必要です。ただし、テストでソフトウェアの品質を作りこむことはできないことに注意してください。毎日体重計に乗るだけでは、やせることはできないのと同じです。テストは、体重計のように、ソフトウェアの健康状態を測る指標を提供します。より正確な指標を提供するには、上手にテストケースを設計し、無数にあるケースを上手にサンプリングすることです。また、バグがないことを証明しようとしないでください。そのような姿勢は、バグを発見しにくくします。テストは、バグを発見するために行ってください。完全なテストをすることは不可能であることを心得た上で、いかに少ないテストケースで効率的にバグを発見すべきか、工夫を凝らしてください。

# 参考文献

**工夫だけでどうにかなるものではない**

**×**

## ソフトウェア・テストの技法

Glenford J. Myers (著), 長尾真監、松尾正信 (訳)

<http://www.amazon.co.jp/dp/4764900599>

## パーフェクトソフトウェア

Gerald M. Weinberg (著), 伊豆原弓 (訳)

<http://www.amazon.co.jp/dp/4822284298>

## Requirements analysis

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

## Software testing

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

## Test automation

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

## Test harness

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

## JUnit

<http://www.junit.org/>

## TestLink

<http://blog.testlink.org/>

## Framework for Integration Testing

<http://fit.c2.com/>

## Selenium

<http://seleniumhq.org/>

## Watir

<http://watir.com/>

## WatiN

<http://watin.sourceforge.net/>

## HttpUnit

<http://httpunit.sourceforge.net/>

## HtmlUnit

<http://htmlunit.sourceforge.net/>

## Canoo WebTest

<http://webtest.canoo.com/webtest/>

## Fit

<http://fit.c2.com/>

## Microsoft UI Automation

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

## XPエクストリーム・プログラミング入門

Kent Beck (著), 長瀬 嘉秀 (監訳)

<http://www.amazon.co.jp/dp/4894716852>

## いつまでもデブと思うなよ

岡田 斗司夫 (著)

<http://www.amazon.co.jp/dp/4106102277>

1. これは、マイヤーズの三角形問題として知られています。 [↑](#footnote-ref-1)
2. スイート (suite)とは、ひと揃いという意味です。ホテルのスイートルームのそれと同じ意味ですね。 [↑](#footnote-ref-2)
3. すべてのマニュアルケースを、すべてのビルドに対して実施することは非現実的です。どのケースをどのビルドで実施していくか、うまく計画を立てましょう。 [↑](#footnote-ref-3)
4. 新規にソフトウェアを開発するときには、最初にソフトウェア開発計画書を書くべきですが、この図では割愛しています。また、ソフトウェアのドメインによっては、各文書の名称や内容が少しずつ違いますが、文書を詳細化する基本的な流れは同じです。 [↑](#footnote-ref-4)
5. ワン・ページと言っても、実際に1ページで書けることはあまりありません。 [↑](#footnote-ref-5)
6. バグの根本原因を説明するために、コードの断片をバグ報告票に貼ることはあります。 [↑](#footnote-ref-6)
7. もっとも、単体テストの実装の進捗を測ることはできません。完全な単体テストの実装を先に終えることは難しく、単体テストのコードもインクリメンタルに追加する必要があります。 [↑](#footnote-ref-7)
8. もちろんSCMで管理します。 [↑](#footnote-ref-8)