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) |

# テストとは

テストとは、テスト対象のプログラムを実行して操作し、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。本書ではテストの設計技法については述べませんが、十分なカバレッジをもつ統合テストのテストケースを準備し、実施する方法を説明します。

# テストの分類

テストは、さまざまな観点から分類できます。本節では、そのいくつかを紹介し、本章の範囲を示します。

## 機能テストと非機能テスト

ソフトウェアの品質には、機能的な側面と非機能的な側面があります。例えば、「このボタンを押したらこう振舞うべき」というのは機能的な側面です。しかし、たとえ機能的には問題なく動作しても、処理速度がとても遅くて使いものにならないことがあります。処理速度のような、機能的でない側面を非機能的な側面といいます。非機能的な側面はまた、パフォーマンス、セキュリティ、リライアビリティなどの側面に分類されます。非機能的な側面をテストする場合でも、criteriaをきちんと設定すれば問題なくテストできます。例えば、「この操作は2秒以内に終わるべきこと」と決めれば、このテストにpassするかどうかを明白に判断できます。本章では機能テストと非機能テストの区別は気にしません。

## マニュアルテストとオートメーション

マニュアルテストとは手作業によるテストのことです。テストオートメーション、もしくは単にオートメーションといえば自動化されたテストのことです。本来オートメーションという言葉には自動化という意味しかありませんが、ソフトウェア開発の文脈では、オートメーションといえば自動化されたテストを指します。本章では、この両方を扱います。

## ブラックボックステストとホワイトボックステスト

ブラックボックス(黒い箱)とは、外側がどう振舞うかはわかるが、内部がどのように動作しているかはわからない装置のことを指しています。つまり、ブラックボックステストとは、そのようなソフトウェアをテストすることです。これに対し、ホワイトボックステストとは、内部の挙動までわかっているソフトウェアを、それを踏まえてテストすることです。例えば、プログラムコードの実行経路を漏れなくテストするなどです。この網羅性を測る基準としては、命令網羅(statement　coverage), 分岐網羅(branch coverage), 条件網羅(decision coverage）などが使われます。本章で扱うのはブラックボックステストです。

## 単体テストと統合テスト

単体テスト(unit test)は、ひとつのクラス(ソフトウェアを構成する部品)を対象としたテストです。つまり、ひとつのクラスをテストの単位(ユニット)としているわけです。これに対して、統合ビルドをテストすることを統合テスト(integration test)といいます。これを指して結合テストということもあります[[1]](#footnote-1)。さらに、実際に運用される環境とほぼ同じ状態で実施する統合テストをシステムテスト(system test)、ユーザーに納品する直前の統合テストを受け入れテスト(acceptance test)といいます。ユーザーが製品を受け入れるためのテストということです。

一般に、単体テストの自動化は比較的容易なので、必ず自動化します。これに対し、統合テストの自動化はそれほど簡単ではないので、自動化できるものは自動化し、そうでないものは手作業で実施します。単体テストと統合テストについては、後の節で改めて取り上げます。

## 動的テストと静的テスト

動的テストとは、テスト対象となるビルドやモジュールを実際に動作させて、それを検証するテストです。本節でここまでに紹介したテストは、すべて動的テストに分類されます。これに対し、静的テストとはソースコードをレビューしたり、静的解析ツールを使って品質を評価したりするテストです。プログラム中の関数名や変数名が命名規則に従っているか、コメントが記述されているか、正しくインデントされているか、といった分析のほか、例外をキャッチして握りつぶす(何もしない)コードがないか、実行されることがないコードを含んでいないか、といった問題を検出できます。古典的な静的解析ツールにはlintがあります。コードをコミットする前にこのようなツールを動かして、報告された問題をすべて除去してからコミットするのは良い習慣です。

また、種々のドキュメントのレビューも静的テストに含まれます。静的テストは動的テストよりも早い段階で実施できるので、上手に使えばその効果は大きいものです。以降の節で扱うのは、すべて動的テストです。

# 同値分割と境界値分析

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。この冒頭によい問題がのっていますので、ここに引用しましょう[[2]](#footnote-2)。

|  |
| --- |
| あるプログラムは、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できません。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(の同値クラス)に分類し、その代表値で無限にある組み合わせを代替しなければなりません。単純に、たくさんのテストケースを用意してもだめなことに注意してください。必要なのは、テストケースの数ではなくカバレッジ(網羅性)です。カバレッジを確保できるようにテストの戦略を立て、テストを設計することが求められます。いかに同値クラスを発見して、無数に考えられるテストケースの中から適切なサンプリングをして、実施すべきテストケースを選びだすか、ということです。

表6 - 1 あなたのテストケースの採点項目

|  |  |  |
| --- | --- | --- |
| # | 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、すべての辺が長辺になるケースがあるか。 | (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 |

このようなテクニックを同値分割といいます。また、同値クラスの中のはじっこの値を代表値として選ぶことを境界値分析といいます。この技法はとても強力で基本的なテクニックです。そのため、これを最初に紹介した書籍「ソフトウェア・テストの技法」はソフトウェアテストの古典となっています。同値分割と境界値分析のテクニックは、ブラックボックステストに分類されます。

# 包括的なテストケースの準備

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。そのために、下記の文書を順に記述します。

* テスト戦略書  
  どのような方針で、テスト対象となるソフトウェアのどの側面の品質を評価するか。どのようにカバレッジを確保するのか。
* テスト計画書  
  テストに必要なリソース (人数、期間、PCなどのハードウェアの台数など) とスケジュール。テスト対象とするソフトウェアのビルド番号。
* テスト仕様書  
  評価したいソフトウェアの品質 (機能やパフォーマンス) の具体的な説明。テストの対象とする範囲としない範囲。テストを実施する環境。テスト戦略に基づき、テストをどのように構成するか。
* テスト設計書 (テストケース)  
  テスト仕様を、(表6 - 1のレベルにまで) 具体的にしたもの。また、自動テスト (オートメーション) の場合には、オートメーションコード (プログラム) の設計を含む。
* テストスクリプト  
  テストケースを、実際にテスト可能なレベルにまで詳細にした手順書。オートメーションの場合には、テスト設計を実装したテストコード (プログラム) 。

テストの戦略や設計方法については、本書の範囲外なのでこれ以上は説明しません。本書の関心事は、膨大な数になるテストケースとテストスクリプトをどのように管理するかです。そこで、これらについてもう少し詳細に見ていきましょう。

# テストケースとテストスクリプト

テストケースとは、ある環境でこういう操作(入力)をするとどういう結果(出力)になるべきかを記述したものです。それを、詳細な手順に書き下したものがテストスクリプトです。例えば、★6-2節で紹介したテストケースのひとつをテストスクリプトに書き下すと、次のようになります。

|  |
| --- |
| **準備:**  MyersTriangle.exeをリリース場所のファイルサーバからテスト用PCにコピーする  **手順:**   1. コマンドプロンプトを起動する 2. MyersTriangle.exe 2 3 4と入力する   **期待する結果:**  画面に「不等辺三角形です」と表示される |

このようなテストスクリプトを、設計したテストケースに基づいて漏れなく準備します。そして、ある機能が利用可能になったビルドがテストチームにリリースされたら、それをテストするテストケースを実施して結果を記録します。

ただし、テストケースとテストスクリプトを厳密に区別せず、テストスクリプトを指してもテストケースということがあります。現在広く使われているテストケース管理ツールは、その多くが実際にはテストスクリプトの管理をするもののようです。逆に、経験豊かなテスターが作業するときは、すべてのテストケースを詳細なテストスクリプトに書き下す必要がないこともあります。そこで本書でも、これ以降はテストケースとテストスクリプトを区別せず、どちらもテストケースと呼ぶことにします。

# 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。 |

# テストケースに記載すべき項目

テストケースに記載すべき項目について、一般的なものを紹介しましょう。

## テストケースの番号 (TestCase#)

このテストケースを一意に識別する番号です。

## 件名 (Title)

このテストケースの内容を簡潔に説明する項目です。全てのテストケースの中で、どの部分に相当するテストケースなのかをわかりやすく示せるように、件名の名前付けルールを準備すると良いでしょう。例えば、後述する項目(環境やカテゴリなど)のいくつかを示す符丁を件名に含めると、後でテストケースの検索や分類がしやすくなります。

## 作成者 (Created by)

このテストケースを記述した人の名前です。作成者が、このテストケースの保守を担当することも多いでしょう。ただし、テストケースを実施する人がテストケースの作成者と違うことはよくあります。もし、実施者がこのテストケースを実施する上で不明なことがあっても、テストケースにその作成者の名前が書かれていれば、その人に問い合わせることができます。

## 担当者 (Assigned to)

このテストケースを実施する担当者の名前を記入します。

## 実施に必要な工数 (★Estimation)

分もしくは時間といった単位で実施に必要な工数の見積もりを記入します。このような項目があれば、テストケースのセットを実施するときに、おおまかに全体の工数見積もりを出せます。また、担当者ごとに見積もり工数を出せば、各担当者の負荷を平準化できます。

## 環境 (Configuration; Environment)

どのような環境に対して有効なテストケースであるかを記述します。例えば、Windowsアプリケーションであればテスト対象とするWindowsのバージョン(Windows VistaとかWindows 7など)を、Webアプリケーションであればテスト対象とするブラウザ(Microsoft Internet ExplorerとかMozilla Firefoxなど)とそのバージョンを列挙します。

## テストの準備 (Setup)

テストの実施を準備するための手順を記入します。多くの場合、テスト対象のビルドをインストールするとか、データベースにテストデータを挿入するなどの記述になるでしょう。

## 手順 (Steps)

テストの実施手順です。いわば、このテストケースの本体となる重要な部分です。

## 期待する実行結果 (Expected Results)

手順を実施した結果、どうなるべきかを記述します。場合によっては、手順ごとに実行結果が記述されることもあります。そのときは、この操作をしたらどうなるべき、次にこの操作をしたらこうなるべき、などのように、手順の中に期待する実行結果が一緒に記述されることになります。

## テストを後始末する手順 (Teardown)

続けて実施する次のテストに影響がないように、このテストの後始末をする手順を必要に応じて記入します。Teardownとは、解体するとか、跡形もなく消し去る、という意味です。

## カテゴリ (Category; Feature path)

このテストケースの分類です。テストの設計で重要なのはテストケースの数ではなくカバレッジですが、十分なカバレッジを確保するために、テストケースの数がとても大きくなることは非常によくあります。そこで、テストを実施する単位として、テストケースをさまざまな側面から分類しておく必要が生じます。このビルドがテストチームにリリースされたらこのカテゴリのテストケースを実施とか、次のビルドがリリースされたらそっちのカテゴリのを実施、という具合です。具体的に、どのような側面でテストケースを分類するかはテスト対象のソフトウェアのドメインによりますが、少なくとも要件や機能ごとに分類しておくことが必要です。要件や機能の一覧をツリー構造で管理できるテストケース管理ツールもあります。このほか、複数の機能から重要なシナリオのものをいくつか取り出して、短時間で実施できるスモークテスト用のテストケースのセット(カテゴリ)を作成しておくのも有用です。

## レビュー済みかどうか

テストケースを記述するプロセスとして、レビューという段階を設けることがあります。その場合は、テストケースの項目として、レビュー済みかどうかを示すものを準備しておくと便利です。レビューアの名前を書く項目を用意してもいいでしょう。

## 有効なテストケースか

レビューが済んで、利用可能となったテストケースであっても、仕様変更やバグの混入などにより、一時的もしくは恒久的にそのテストケースが無効となることがあります。このようなテストケースは削除せず、無効であるマークをつけて管理します。そうすれば、後でテストケースを修正するか、バグの修正をまって有効に戻すことができます。

## 一度でもpassしたか

リグレッション(バグ)の発生を検出するためには、テストを繰り返し実施(リグレッションテスト)することが有効です。しかし、リグレッションが発生していない限りは、一度でも実施してpassしたことがあるテストケースは、再度実施してもpassするでしょう。

しかし、まだ一度もpassしていないテストケースが対象とする機能は、リグレッション(退化)する以前に、まだ実装(進化)が済んでいません。実装の完了を確認するためには、少なくとも一度はテストにpassしなければなりません。そこで、開発(機能の実装)の進捗や、テストケース実施の優先順位を判断する材料として、一度でもpassしたテストケースと、そうでないテストケースを分類しておくことがあります。

## オートメーションコードとの対応

もし、自動化されたテストケース(オートメーション)があれば、そのオートメーションコードがリポジトリのどのフォルダに格納されているかや、オートメーションコードのビルド方法や実行方法を記入する項目をテストケースに設けることがあります。優れたテストケース管理ツールには、オートメーションの実行をツールから直接指示できたり、実行結果を自動的に収集して記録するものもあります。オートメーションについては後述します。

## その他、添付データやコメント、変更履歴など

このテストケースについて、必要となるテストデータファイルや、マトリックス(表)などが添付されることがあります。また、このテストケースの変更履歴やコメントを記入する項目を設けることもあります

ここに紹介したすべての項目を用意すべき、というわけではないことに注意しましょう。みなさんの組織やソフトウェアに合ったものを選択してください。また、ここに示した(ひとつの)テストケースドキュメントの中に、複数の手順(テストスクリプト)と、それに対応する期待する実行結果をまとめて記入することもあります。その場合は、すべての手順で期待通りの結果となればテスト結果はpassとし、ひとつでもそうならなかったものがあればテスト結果はfailとして管理します。

# テストケースの実行結果の管理

テストケースに関連づけて記録すべき情報の中でも、テストの実行結果(passしたかfailしたか)はとても重要です。テストケースは、最終的にはこの実行結果を得るためのものです。しかし、前節ではテストの実行結果を記入する項目を紹介しませんでした。テストケースは、複数のビルドに対して実施するものだからです。実際、よくできたテストケース管理ツールでは、テストの実行結果はテストケースとは分けて記録します。



図6 - 1 テストケースと、テスト対象のビルドの  
関係を示すクラス図

つまり、テストケースとテスト対象のビルドは多対多の関係をもち、テスト結果はこの関係を説明します。このため、テストケースに複数の実行結果を記入する項目を用意するよりも、テストケースとその実行結果は分けて管理した方が、いろいろと都合が良いのです。これを分けずにテストケースに直接テスト結果を書いてしまうと、繰り返しテストを実行することで、テスト結果を書くところが足りなくなってしまうかもしれません。しかし、テストケースとテスト結果を分けて管理すれば、テスト対象のビルドごとにテスト結果を集計して表示したりすることが簡単にできます。

# テスト結果に記載すべき項目

テスト結果に記載すべき項目を紹介しましょう。

## テスト結果の番号 (TestResult#)

このテスト結果を一意に識別する番号です。

## 実施者 (Assigned to)

実際にこのテストケースを実施した人の名前を記録します。このテスト結果に責任をもつ人です。

## 環境 (Configuration; Environment)

このテストを実施した環境を記入します。テストケースに記述されている環境の中のひとつ(もしくは複数)になるでしょう。

## テスト対象としたビルドの番号 (Build#)

テスト対象としたビルドの番号を記入する項目です。いつテストしたかよりも、どのビルドをテストしたかを記録することの方が大切です。

## テストの開始日時と終了日時 (Start Time/End Time)

テストを実施した日時を記入します。テストにかかった時間はツールに計算させましょう。前述の項目ほど重要な項目ではありませんが、記入しておけば、後で興味深い統計情報や、より正確な工数見積もりを得られるようになります。

## テストの状態 (Status)

未着手(not started)、実行中(in progress)、完了(completed)、調査中(investigate)、解決済み(resolved)などの選択肢を準備します。もしテスト結果が期待したものでなかったら、これを調査して、製品のバグなのかテストケースのバグなのかを突き止めます。どちらのバグであるにせよ、バグ報告票を起票します。バグを起票したら、テストケースの状態はresolvedとする、などの運用が考えられます。これはテストケースの状態を管理する方法の一例です。みなさんのプロジェクトではどのような運用が好ましいか、検討してください。テストの状態をリアルタイムに集計できるテストケース管理ツールを使えば、この項目によりテストの詳細な進捗をチーム内で簡単に共有できます。

## テスト結果 (Result)

最も重要な項目のひとつです。成功(pass)か失敗(fail)かを記入します。このほか、実施せず(not run)とか適用不可(not applicable)といった選択肢を準備することもあります。

## 添付資料 (Attachments)

テストを実行したときのログファイルや画面写真などを添付します。また、これらのファイルはテスト結果に添付せずにファイルサーバに置き、テスト結果にはその場所を記入する項目を設けることもあります。

## コメント (Comment)

テストが失敗したときの状況や、気になる点などがあればコメント欄に記入します。また、テスト結果を実施せずとか適用不可としたときには、その理由を記入します。

## バグ報告票の番号 (Bug#)

テスト結果が失敗だったときは、バグ報告票を起票します。このバグ報告票の番号をテスト結果に記入しておけば、このテストケースを再度実施して失敗したときに、重複したバグ報告票を起票せずに済みます。すでに起票済みのバグ報告票があれば、情報を追記しておきましょう。

テストを実施するたびに、これだけの情報を記録するとしたら、テストケースにテスト結果を直接記入するのはやはり都合が悪いことがわかるでしょう。しかし、ここに紹介したすべての項目を使うべき、というわけではありません。あなたのプロジェクトに適切な項目を設計してください。テストケース管理システムのようなツールの支援がないときは、テスト対象のビルド番号とテスト結果だけをテストケースに直接記入する方が管理しやすいでしょう。

# テストケース管理システム

テストケースとテスト結果を、Microsoft Excelのようなスプレッドシートで記述して管理する組織もまだ多いようです。しかし、スプレッドシートでは、複数のテスターによるテストの進捗をリアルタイムに把握したり、テスト結果を集計してビルドごとに表示したりするのは容易ではありません。テストケースとテスト結果を分離して管理するのも困難です。現実問題として、比較的少ないケース数でカバレッジを確保できたとしても、そのケース数は非常に大きくなってしまうことはよくあります。これらを上手に複数のテスターに割り当て、進捗を管理し、結果を集計するのはとても手間がかかります。

このような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[[3]](#footnote-3)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などがあります。最近は、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。テストケース管理システムには、テスト手順をモジュール化できるものもあります。例えば、テスト手順の最初にログインの操作手順を詳細に書いてしまうと、もしログインの方法が変わったとき、多くのテストケースを書き直さなければなりません。しかし、ログインの手順が共通の操作としてモジュール化されて記述されていれば、その部分だけを修正すれば済みます。このほか、バグが再現したとき、その再現手順を画面写真に撮ったり、動画ファイルに録画したり、バグ報告票を起票する作業を支援したりするものもあります。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) を追跡できるものもあります。

図6 - 2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 - 2 テストケースと、テスト対象のビルドの  
関係を示すインスタンス図

# 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。  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。   * 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。   * 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   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、毎月もしくは毎週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。レトロ趣味という言葉もありますね。   |  | | --- | | 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。 |

# 仮想化技術によるテスト環境の構築

昨今は電子書籍リーダーが広がりを見せていますが、日本語のコンテンツはまだまだ少ない状況です。そこで、本を自分でスキャナで取り込んで、コンテンツを自炊する人が増えているそうです。電子化した本をバーチャルな本棚にしまえば、もはや元の(紙の)本は物理的に所有する必要はなく、廃棄すれば奥さんにも喜ばれます。

PCの仮想化技術も、これとよく似ています。つまり、複数のPCを電子化して論理的なPC(仮想PC)とし、これらをひとつの物理PCに入れて同時に稼動させるのがPCの仮想化技術です。仮想化したPCをバーチャルなラックにしまうというわけです。

テレビゲームが好きな人は、電子書籍よりもエミュレータを想像してもらう方と良いでしょう。PCの仮想化は、Wiiの中でファミコンのゲームを動かすのと似ています。ただし、Wiiの中で同時に動かせるファミコンは1台だけですが、PCの仮想化技術ではひとつの物理PCの中で複数の仮想PCを同時に動かすことができます。

仮想マシンといえばJavaを連想する人も多いでしょう。Java仮想マシンの上ではJavaバイトコードが動きますが、仮想PCの上では通常のOSが動作し、さらにその上で通常のアプリケーションソフトウェアが動きます。

稼働中の仮想PCは、リモートデスクトップ接続などを使って接続すれば、普通のPCとまったく同様に利用できます。現在はハードウェアによる仮想化技術への支援もあり、仮想PCは非常に高速に動作します。CPUコア数やメモリを潤沢に積んだ強力なサーバ用PCを使えば、コンパクトにテスト・ラボやビルド・ラボを構築できます。現在Windows環境でよく使われる仮想化技術には、VMwareやMicrosoft Hyper-Vなどがあります[[4]](#footnote-4)。

ひと昔前は、デュアルブートとかマルチブートという技術がよく使われました。これはひとつの物理PCに複数のOSをインストールし、その物理PCの電源を入れたときにどのOSで起動するかを選択できるようにします。しかし、マルチブートでは、ひとつのPCに複数のOSをインストールしても、同時に稼働できるOSはひとつだけです。これに対し、PCの仮想化技術では、物理PCのリソース(CPUやメモリなど)が許す限り、その中でいくつでも仮想PCを同時に稼働させることができます。また、稼働が不要な仮想PCの電源は切っておくこともできます。これにより物理PCのリソースを空け、この中でさらにほかの仮想PCをセットアップして稼働させることもできます[[5]](#footnote-5)。

さらに、仮想PCにはスナップショットを撮る機能があります。OSと開発中のソフトウェアを仮想PCの中にセットアップした直後や、あるバグが再現できる環境を仮想PCの中に構築できたら、その仮想マシンのスナップショットを撮っておきます。すると、後でそのスナップショットを適用する操作により、何度でも仮想PCをその状態に戻せます。仮想PCのスナップショットは、さまざまなOSやブラウザのためのテスト環境をきれいな状態に戻したり、バグを再現できる環境を保存したりする上で大変強力な機能です。PCの仮想化は適用範囲の広いテクノロジーですが、特にソフトウェアテストをするときには非常に便利なものです。

仮想PCを管理してテスト・ラボを構築し、マニュアルテスト環境を構築したり、オートメーションを各仮想PCに流し込んだりする機能をもつテスト支援ツールにはMicrosoft Visual Studio Lab Management などがあります。仮想PCのセットアップは、床に這いつくばって配線しなくてもいいので、手作業で行っても物理PCのセットアップよりずっと楽です。しかし、仮想PCを制御できるテストケース管理システムを使えば、仮想PCの管理はさらに楽ちんです。ソフトウェアのテストという文脈の中で、仮想PCを管理できるからです。例えば、マニュアルテストのための環境の準備を一括して行うことができます。

# テストの自動化

テストをするのは、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。

しかし、テストを自動化することにも工数が掛かりますから、繰り返し実施する価値のあるテストかどうかを考慮したうえで、その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できるわけではないし、その必要もないと心得た上で、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るようになり、ノウハウも蓄積できます。

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。

# 統合ビルドをオートメーションする

統合ビルドをオートメートするときは、主に2つの方法があります。

## ツール駆動によるオートメーション

多くの場合、これはUIオートメーションのことです。UIとはUser Interfaceの略で、ユーザーとコンピュータをつなぐ界面を指します。広義にはキーボードやマウスなども含まれますが、テストオートメーションの文脈においては、UIは画面のことと考えて差し支えありません(キーボードやマウスをロボットに操作させてオートメーションを実現することは、まだ)。つまり、これはユーザーによる操作を直接的に自動化する、というアプローチです。

WindowsなどのGUI環境[[6]](#footnote-6)では、ボタンなどのUIコントロールにマウスのクリックイベントを送ることで、ユーザーによる操作をシミュレートできます。そこで、UIオートメーションを実現するツールには、ユーザーが操作したことで発生したイベントを記録して自動的にテストスクリプトを生成し、それを後で再生する機能をサポートするものがあります。その結果、UIが期待通りに変化すればpassとします。

ツール駆動によるオートメーションの利点は、テスト対象のソフトウェアを変更する必要がなく、テスターには高度なプログラミングのスキルが求められないことです。ただし、テスト対象のソフトウェアのUI画面の変更に弱いという欠点があります。テスト対象のUIが変更されると、テストスクリプトの中で押したいボタンが押せなくなったりするため、テストスクリプトを保守する工数はそれほど小さくありません。

## コード駆動によるオートメーション

テスト対象のソフトウェアに、テスト実行用のAPI(Application Programming Interface; プログラムとプログラムをつなぐための界面)を装備させて、このAPIを介してテストを自動化する、というアプローチです。コード駆動によるオートメーションをするには、テスト対象のソフトウェアをtestableにするための修正が必要です。どのような修正が必要となるか、次節で見てみましょう。

# オートメーションに必要な部品

オートメーションを構成するコードは、ドライバ、スタブ、ハーネスなどの部位に分けられます。図6 - 3に、馬車のイメージを使って、ビルドとオートメーションコードの関係を示しました。以下で、それぞれを説明しましょう。



1. 図6 - 3 ただ今、ビルドのテスト中…

## テストドライバ

テスト対象のソフトウェアをする、オートメーションコードの本体となる部分です。ただし、ドライバというと、一般に統合テストではなく単体テストのコードを指します。単体テストの対象となるクラスは、まだ、これを呼び出して動かす部品と統合されていませんから、その部品を代替するテスト用コードという意味がドライバという語に強調されます。

これに対し、統合テストのオートメーションコードの本体は、オートメーションと呼ばれます(そのまんまです)。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。

## テストスタブ

スタブも、単体テストをするときに必要となるソフトウェア部品です。テスト対象のクラスは、それが呼び出して使いたい(つまり、このクラスが依存する)ソフトウェア部品(コンポーネントやモジュール)がまだありませんから、やはりそのままでは動きません。そのため、ハリボテ部品を作ってテスト対象のクラスから呼び出させるようにします。このハリボテ部品がスタブです。スタブ(stub)とは、木の切り株とか根っこという意味です。また、stub outといえば「もみ消す」という意味で、テスト対象のクラスが依存するほかのクラスをもみ消すという意味があります。スタブは、テスト対象から見て正しく動作するふりをすれば十分です。例えば、計算処理をせずに正しい計算結果をただ返すだけのスタブを作ってテストに使います。もちろん、依存先のソフトウェア部品がすでにあれば、それを使ってテストすればよく、スタブを準備する必要はありません。ドライバとスタブは対になって単体テストを支援します。

また、テスト対象のクラスが、依存先のクラスのメソッドを期待通りに呼び出すかを検査できるスタブを、モックオブジェクト(あるいは単にモック)といいます。モックオブジェクトは、テスト駆動開発の派生である振る舞い駆動開発(Behavior Driven Development)で

## テストハーネス

テストハーネスとは、テスト対象のビルドとオートメーションコード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。

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、そのソフトウェアの品質を大きく左右します。テスト対象のビルドを簡単に操作できるハーネスを準備したら、これを使って各テストスクリプトを手順どおりに実装します。

## モックオブジェクト

TBD

# 単体テストと統合テスト

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、テスターが担当します。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。

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。

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[[7]](#footnote-7)。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。

一般に、統合テストのオートメーションはテスターが開発します。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テスターには、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、テスターはよく理解しておかねばなりません。

図 テストケース管理ツールとテストPCの関係

# ビルドマシンでオートメーションを実行する

ビルドサーバには、オートメーションを実行する機能をもつものがあります。

# 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。

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。

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 - 5 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 - 6 テストケースの分類

# 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。

# 参考文献

## 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>

## lint

<http://en.wikipedia.org/wiki/Lint_(software)>

## 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. 結合テストの英訳語をjoin testとする文章も見つかりますが、誤りです。英語でjoin testといえば「テストに参加する」という意味か、好意的に捉えても「joinキーワードを使ったSQL文をテストする」という意味であろうかと思います。 [↑](#footnote-ref-1)
2. 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-2)
3. スイート　(suit) とは、ひと揃いという意味です。背広の上下ひと揃いをスーツ (suits) といいますね。また、ホテルでリビングやダイニングが揃った部屋をスイートルーム (suite room) といいます。このほか、トランプでは、あるマークの1から13までのひと揃いをスート (suite) といいます。ソフトウェアにおいては、オフィススイートといえば、ワープロやスプレッドシートをひと揃いまとめたパッケージ製品のことです。また、テストスイートという語もありますが、これはテストケースが注入されてテスト可能になったテスト環境のことです。 [↑](#footnote-ref-3)
4. Windows環境を仮想化するなら、筆者はHyper-Vを勧めます。 [↑](#footnote-ref-4)
5. 仮想PCにOSやアプリケーションをインストールするには、仮想PCごとにソフトウェアライセンスが必要です。 [↑](#footnote-ref-5)
6. Graphical User Interfaceの略です。 [↑](#footnote-ref-6)
7. もちろんSCMで管理します。 [↑](#footnote-ref-7)