第 31 章　テストの種類

この章の内容

この章では、テストに関わる以下の 5 つのテーマについて議論する。

1. 検証と妥当性確認
2. テスト・ケースの設定の方法
3. テストの種類
4. 移行
5. テスト技術者のための資格

「検証」と「妥当性確認」

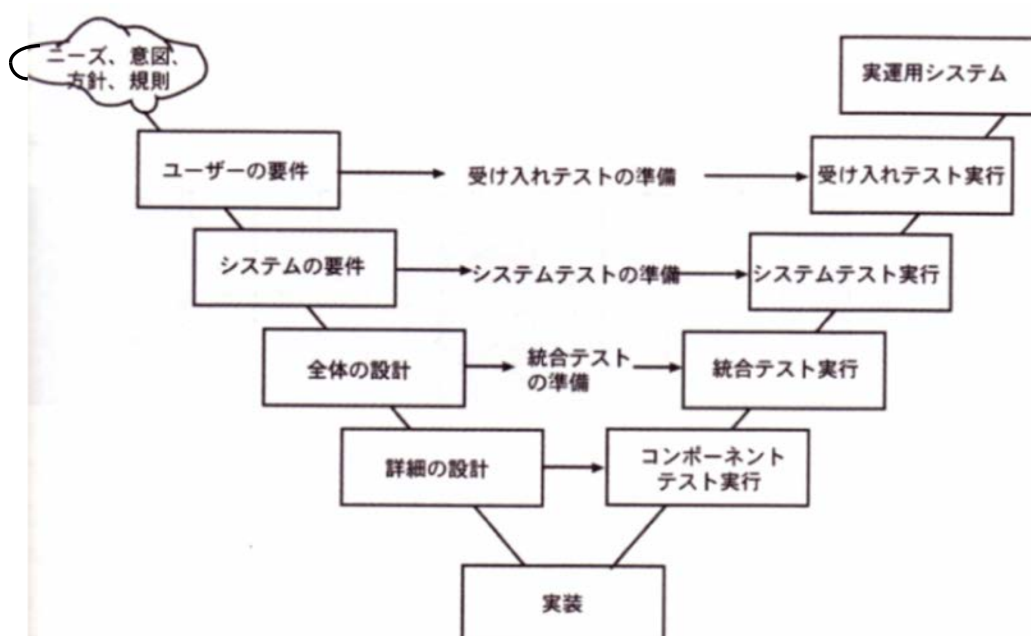
この章での最初のテーマは、検証と妥当性確認についてである。

レビューも含めて、広い意味でのテストには 2 つの種類がある。「検証 (Verification)」と「妥当性確認（Validation)」である。英語ではこの両者の頭文字がいずれも V で始まっているので、この両者を包含して呼ぶときに「V&V」と呼ぶことがある。

カーネギー・メロン大学のソフトウェア工学研究所が策定した「開発のための CMMI（Capability Maturity Model Integration、能力成熟度モデル統合）第 3 版」では、検証と妥当性確認をそれぞれ次のように定義している[CMM10a]。

検証(verification)：作業成果物が明記された要件を適切に反映しているかどうか確認すること。つまり、検証は、「正しく構築した」ことを確実なものにすることである。

妥当性確認(validation)：提供された（または提供されるであろう）成果物がその意図された用途を充足しているかどうか確認すること。つまり、妥当性確認は、「正しいものを構築した」ことを確実なものにすることである。



図表 31-1　V 字型開発の図

つまり「検証」は、最終的には当初提示された要件定義書通りにソフトウェアが構築されていることを確認する作業である。ここで、最終的にソフトウェアとの突き合わせの対象になるものは「要件定義書」である。これ以外にもプログラム設計書とモジュール、詳細設計書とプログラムなど、いくつかのレベルで確認をすることができる。これはウォータフォール型開発の流れをソフトウェア開発段階で折り返す「V 字型開発」の図で、的確に表現される。その V 字型開発の図を、図表 31-1 に示す。

一方の「妥当性確認」は、前述の通りソフトウェアがその意図した用途を充足しているかを確認する作業である。それでは、ソフトウェアを「何と」突き合わせて妥当性確認を行えばよいのだろうか。検証を易しいというつもりはないが、妥当性確認のこの問題は難しい問題である。

要件定義書で提示された要求が過大要求や過小要求、あるいは不適切な要求であって、そのソフトウェアを手に入れようとしている企業にとってふさわしいものでない場合、これは妥当性確認のチェックの対象である。この確認は要件定義書のレビューの段階で実施するのが良い。それ以外にも、テスト段階で開発者以外の人がそのソフトウェアを実際に使用してみて、妥当なものであるかどうかを判断することで妥当性確認を行うことができる。これは普通、運用テストの段階で実施されることになる。

いずれにしろ妥当性確認も、レビューとテストを通して行うことになる。これは、検証の場合と変わらない。

「テスト・ケースの設定方法」

この章での 2 つ目のテーマは、「テスト・ケースの設定の方法」についてである。

テスト・ケースの設定の方法には、以下の 3 つがある[GRA07]。

① 仕様ベースのテスト・ケース設定
② 構造ベースのテスト・ケース設定
③ 経験ベースのテスト・ケース設定

これらの方法でのテスト・ケースの設定はいずれかだけを行えばよいというものではなく、次に述べるテストの種類に応じてこれらのテスト･ケースの設定法を適宜使い分けるのがよい。

以下で、それぞれについて概説する。

「仕様ベースのテスト・ケースの設定法」

仕様ベースのテスト・ケースの設定法とは、要件定義書などを基にテスト・ケースを設定する方法である。別のいい方をすれば、プログラムの中を一切参照せずにテスト・ケースを設定する。このやり方でテスト・ケースを設定して行うテストを、ブラック・ボックス・テストとも呼ぶ。

仕様ベースでのテスト・ケースの設定の方法には、さらに次の 4 つの方法がある[GRA07]。

① 同値分割法
② 限界値分析
③ ディシジョン・テーブルによる方法
④ 状態遷移図による方法

「同値分割法」とは、以下のような方法をいう。

例えば、社員として高校卒業生と大学卒業生を採用することにし、60 歳の誕生日を過ぎた月

の月末に該当する社員が定年退職するというルールを持っている会社があるとする。この場合社員の年齢は、18 歳未満、18 歳以上 60 歳以下、及び 60 歳を超えた範囲の、3 つの区分に分割される。ただし、最初の 18 歳未満と最後の 60 歳を超えたところは、社員としては対象外の範囲である。

この場合には、テスト・ケースは次の 3 件を用意すればよい。

① 18 歳未満
② 18 歳以上 60 歳以下
③ 60 歳を超えた範囲

そしてこのうちの②だけが処理の対象になり、①と③はエラーにするなど処理の対象外の取り扱いをしていることが確認できればよい。

実際に設計を行い、プログラムを書いた経験のある人なら、上の例で 18 歳と 60 歳のところに欠陥が入り込みやすいことを実感としてしっているだろう。設計段階での以上／超える、および以下／未満の取り扱いのミス、プログラミングの段階での条件判定の等号の取り扱いのミスなどで、この欠陥が生じる。この欠陥をテストする方法を、「限界値分析」という。上で挙げた例ではこの 18 歳と 60 歳が限界値になり、テスト・ケースとしては次の 4 種類を用意することになる。

① 17 歳
② 18 歳
③ 60 歳
④ 61 歳

ここでは②と③がエラーにならず、①と④がエラーになればよい。

この 4 件で前述の同値分割も含めて、全部のケースを網羅したことになる。しかし私ならこの 4 件に加えて、35 歳あたりのデータをもう 1 件付け加えて、全体として 5 件でこの両方の方法を満足するテスト・ケースにするだろう。

処理の種類が何種類かあって、どの処理を行うかの条件が複雑に組み合わされている場合は、ディシジョン・テーブルを使ってこの条件の組み合わせと処理の内容を整理して、テスト・ケースを設定するのがよい。

また状態の遷移が複雑で、そこが重要なテストの対象である場合には状態遷移図を書いて、それを見ながらテスト・ケースを設定するのがよい。

仕様ベースのテスト・ケースの設定は、検証のためのテストを対象にしたものである。

「構造ベースのテスト・ケースの設定法」

「構造ベースのテスト・ケースの設定法」とは、プログラムを読んで、その結果によってテスト・ケースを設定することをいう。この方法でテスト・ケースを設定した場合のテストを、「ホワイト・ボックス・テスト」、または「グラス・ボックス・テスト」ともいう。

余談になるが、当初はグラス・ボックス・テストいう言葉はなかった。ブラックボックス（黒い箱）は、箱の中を見ることができない。それに対してホワイトボックス（白い箱）は、箱の中が見えることになっていた。しかし現実は、ホワイトボックスもブラックボックスと同様、箱の中は見えない。どうしても箱の中が見えるものが必要になって、グラスボックス（ガラスの箱）という言葉が使われるようになったと推測する。もっと理屈をいえば、ガラスはガラスでも透明ガラスでなければならない、というような議論がその中に出てくるのかもしれない。

プログラムの中を見てテスト・ケースを設定する場合、そのプログラムの「全てのステート

メントを最低 1 度は通過させる」ということが大命題になる。その通過のさせ方として、単にステートメントだけに注目する場合と、全ての判断（IF 文だけではなく、LOOP の継続／停止の判定なども含めて）で最低 1 度ずつは True と False に抜けるケースを設定するレベル、IF 文などの条件判定の部分が AND や OR、NOT などを組み合わせて複雑な場合に、それぞれの判定の要素毎に True と False に抜けるケースを設定するレベルまで、いろいろのレベルがある[GRA07]。

単純にステートメントに注目して全パスの通過を計るより、個々の判定の要素には関わらないにしても全ての判定文で True と False に抜けるケースを設定することで、結果として全部のステートメントを通過させることが望ましい。

構造ベースのテスト・ケースの設定では、ステートメント数でも判定条件でも、全体の中のどれだけをカバーしたかを示すカバレッジが問題になる。当然のことながらカバレッジは、100%が必要である。ちなみに仕様ベースのテスト・ケース設定では、カバーできる範囲は 60%から 75%程度まで、アドホックなテストでは 30%見当といわれている[GRA07]。より完全なテストを実施するためにカバレッジを上げようとするなら、構造ベースでのテスト・ケースの設定が欠かせない。

構造ベースのテスト・ケースの設定も、検証のためのテストを対象にしたものである。

「経験に基づくテスト・ケースの設定法」

「経験に基づくテスト・ケースの設定法」とは、テスト担当者のこれまでの経験で欠陥が隠れていると思われるところに重点を置いてテスト・ケースを設定することをいう。

この方法は、仕様ベースの同値分割／限界値分析や構造ベースの方法のような論理的なものではない。つまりこの方法による欠陥の発見の具合は、当然のことながらテスト担当者によって異なってくる。それでも一般にこの方法でのテスト・ケースの設定は、検証の場合でもたいへん有効である。テスト担当者が設計やプログラミングを担当したソフトウェア技術者のバックグランドや性格をよくしっている場合には、欠陥発見の割合を一層高くすることができる。

妥当性確認のためのテスト・ケースの設定は、もっぱらこの方法で行うことになる。

「テストの種類」

この章での 3 つ目のテーマは、「テストの種類」についてである。

通常の開発では、ソフトウェアのテストは次の 4 つの段階を踏んで実施されることが多い。

① 単体テスト
② 結合テスト
③ システム・テスト、及び受け入れテスト
④ 運用テスト

これ以外に、例えば「重要インフラ投資ステム」のような高信頼性ソフトウェアを開発する場合には、この 4 種類のものに加えて追加のテストが行われる場合がある。日本情報システム・ユーザー協会（JUAS）では、独自のソフトウェア開発の手順として提案している U 字型開発方式の中で、単体テスト段階から機能確認のテストを実施することを推奨している。JUAS ではこれを、「単体機能テスト」と呼んでいる。これは高信頼性ソフトウェア開発での、追加のテストの 1 つになりうるものである。

これらに加えて、システム・テスト以降のテストで欠陥を発見した場合に、その欠陥が的確に取り除かれたことを確認する「確認テスト（再テスト）」、やはりシステム・テスト以降のテ

ストや保守作業で既存のソフトウェアを修正した場合、修正対象ではない部分の機能が損なわれていないことを確認する「回帰テスト」など、ソフトウェアのテストには多くの種類がある。

なお個々のテストの名称は、この原稿ではここであげたものを用いる。しかし現実は、それぞれの組織で違う呼び方で呼ばれていることがある。したがってここでは概念を把握することにして、名前にはあまりこだわらない方が良い。

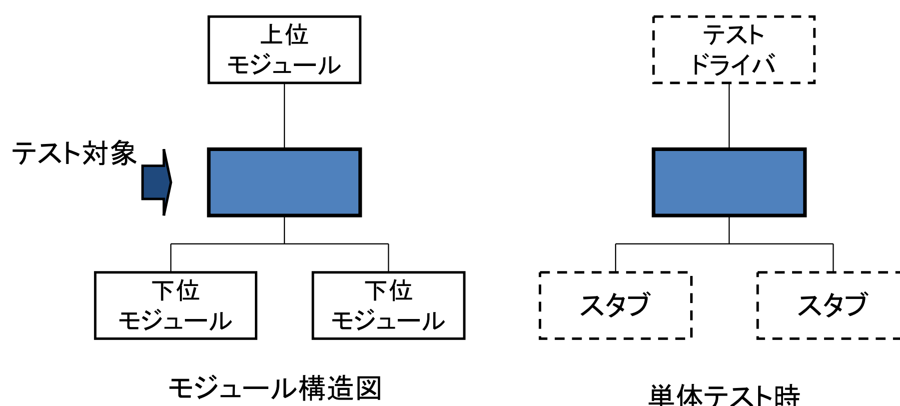
以下で、これらのソフトウェアのテストについて述べる。

「単体テスト」

「単体テスト」は普通プログラミングが行われた直後に、そのプログラム（モジュールとか、メソッドとかと呼ばれることがある）を作成したプログラマによってなされる。単体テストの目的は、モジュール仕様書に明記された機能をこのモジュールが果たすことの確認である。

このテスト実施の方法を、図表 31-2 に示す。

ここで単体テスト時は、テスト対象のモジュール、またはメソッド（これ以降は、「モジュール、またはメソッド」という言葉を、「モジュール」という言葉で総称する）を 1 つだけを取り上げてテストすることが重要である。そのためには、本番稼働時には図表 31-2 の左側のような形で動くモジュールを、右側の形にしてテストを行うことになる。つまり上位のモジュールの代わりにテスト・ドライバを用意し、下位のモジュールの代わりにスタブと呼ぶテスト用のモジュールを必要な数だけ用意する。テスト・ドライバは必要なテスト・ケースを設定して対象のモジュールに必要な回数だけコントロールを渡し、どのテスト・ケースでその対象モジュールがどういう答えを返したかの記録を取る。スタブは逆にどういう条件で対象のモジュールから呼ばれて、どういう答えを返したかを同じ記録上に記載する。そしてこの記録を後で分析することで対象モジュールの動きを確認することによって、単体テストの結果を検証する。



図表 31-2　単体テストの方法

そのような機能のあるなしにかかわらずテスト担当者（プログラマ）は、前記目的達成のための確認と併せて、100%のテスト・カバレッジを実現しなければならない。「単体テストで 100%のカバレッジを実現しないプログラマは犯罪者である」という言葉を、私は以前にどこかで聞いたことがある。したがって、カバレッジ測定のための機能をドライバが持っていることが必要である。

個人的な話で恐縮だが、私は昔新人のプログラマを 4 人預かって、モジュールの設計の方法

からプログラムの書き方、テストの方法までを、あるプロジェクトの中で半年かけて指導／訓練する経験を持ったことがある。新人が本番で稼働するプログラムを作成し、私がそれらのプログラムをテストするためのドライバとスタブを作って、単体テストでの全パス通過を徹底した。その結果このプログラムは、本番で稼働している間に一度も障害を起こさなかったという記録を作った。私のソフトウェア技術者としての経験の中で、これが唯一ではないかもしれないが、少なくとも最初の経験だった。その新人の中の何人かはその後、すばらしいソフトウェア技術者に育った。

「結合テスト」

単体テストの後に普通に行われるテストは、「結合テスト」である。結合テストでは単体テスト済みのモジュールを、インタフェースを確認しながら順次結合していって最終的に 1 つのプログラムを完成し、そのプログラムがプログラム仕様書で要求されている機能などを実現しているのを確認することを目的としている。

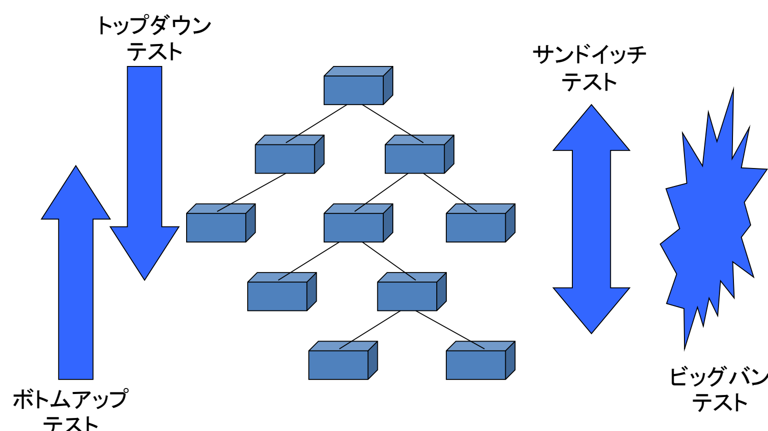
結合テストでのインタフェースの確認のためのテスト・ケースの設定は、構造ベースの方法で行われる。つまりある段階までの結合テストでは、結合を行おうとする対象の 2 つのプログラムを調べて、両方のインタフェースを明確に認識して結合を行う。

結合のさせ方には図表 31-3 で示すように、以下の 4 つの方法が提案されている。

①　ビッグバン・テスト
②　トップダウン・テスト
③　ボトムアップ・テスト
④　サンドイッチ・テスト

前記 4 つのテストの方法のうち、ビッグバン・テストは全てのモジュールを全部同時に結合してしまおうという方法である。しかしこれは、良い方法とはいえない。トラブルが起きるとその欠陥を解明するのがたいへんであり、しかもまず間違いなくトラブルが起きる。この方法は、避けるべきである。

トップダウン・テストは上位のモジュールから順次結合してゆく方法であり、ボトムアップ・テストは下位のモジュールから順次結合してゆく。サンドイッチ・テストは真ん中のモジュールから始めて、上と下の両方向に結合をのばしてゆく。いずれの場合も、「一度に結合を試みるのは 1 つのモジュール」というのが原則である。



図表 31-3　結合テストの方法

それでは、この 3 つの方法のうちどれが優れているのだろうか。

普通の場合、上位のモジュールはデータの種類や状況を判断して処理の流れをコントロールする機能を持っている。そして下位のモジュールは、個々の計算などの具体的な機能を実行する。したがってもし処理の流れが複雑な場合には、上位のモジュールを先にテストするトップダウン・・テストが良い。逆に計算などの実際の処理の内容が複雑な場合には、下位のモジュールを先にテストするボトムアップ・テストが良い。そのプログラムの性格を見て、この 2 つを使い分けるのが適切な進め方である。その意味からいえば、サンドイッチ・テストを採用する理由がないことになる。

トップダウン・テストの場合、テスト・ドライバは一度作ればそれに手を入れながら最後まで使い続けることができるが、スタブはその都度作り続けなければならない。逆にボトムアップ・テストはスタブを作る必要はないけれど、テストのたび毎にテスト・ドライバを作り直さなければならない。

結合テストはそのテストの準備から始めて、最後の機能などの確認まで全部のテストを推進する専任のチームを作って、そのチームが担当するのが望ましい。

「システム・テスト」と「受け入れテスト」

図表 31-1 で示したように「システム・テスト」の目的は、要件定義書で提示した事項、つまり機能要求と非機能要求が、開発したソフトウェアで実現されていることを確認することである。

これは、結合テスト済みの、その情報システムを構成する全てのプログラムを使用して行われる。システム・テストのテスト・ケースの設定は、仕様ベースと経験ベースの両方の方法で行われる。

機能の確認のためのテストは、普通は機能確認テストと呼ばれ、本番環境か、あるいは本番環境と基本的に同じテスト環境で実施される。別のいい方をすれば、このテストのために特別の環境を作ったり、特別のツールやプログラムを用意したりすることは、普通はしない。

しかし非機能要求の確認のためには特別の環境を用意したり、そのためのツールを準備したりしなければならないことがある。例えばあるオンラインシステムで、1 秒間に 100 件のトランザクションを処理することが要求されているとすれば、1 秒間に 100 件以上のトランザクションを発生させることができるツールを用意して、最初はそのツールが作り出した 1 秒間に 100 件のトランザクションを、少なくとも数分間テスト対象のプログラムに処理させて、順調に処理が進んでいることの確認を行わなければならない。さらに同じツールを使って、例えば 1 秒間に 110 件のトランザクションを与え続けるとどういう事態になるのかの確認も必要である。このためには、このような特別のテストのための環境とツールが必要になる。

システム・テストでは、機能要求に加えて非機能要求も確認する必要があるために、以下のような種類のテストを実施しなければならない。

- 機能確認テスト
- 負荷・圧力テスト
- 容量テスト
- 構成テスト
- 互換性・適用性テスト
- セキュリティテスト

- 性能テスト
- 導入性テスト
- 信頼性・可用性テスト
- 回復テスト
- サービス性テスト
- ヒューマン・ファクター・テスト
- ……

システム・テストの終了基準を決めることは、難しい。個々のモジュールの単体テストでは、モジュールの機能の確認と併せてカバレッジの 100%達成がその単体テストの終了基準になる。結合テストの場合も、あるプログラムについてそのプログラムを構成する全てのモジュールの結合が終り、要求された機能などの具備が確認できればそのプログラムの結合テストは終了する。

それではシステム・テストは、何を持って終了基準にすればよいのだろうか。もしテスト計画をしっかりと作り、それに応じて充分なテスト・ケースを準備できたとすれば、その全てのテスト・ケースのテストを終了し、テスト期間中に発見された欠陥が全て解決されることが、テスト終了のための 1 つの条件になりうる。しかし必ずしも、これだけでは充分ではない。もう 1 つ、システム・テストの期間中にそのソフトウェアが持っている残存欠陥数を推定して、その数が当初決めた目標を下回ったことが確認できることを、別の終了のための条件にすることができる[1]。この 2 つの条件を共に満足することを、システム・テストの終了基準としたい。

いずれにしろシステム・テストは開発者の立場で行う最後のテストであるから、テスト漏れなどがないようにしっかりと計画を立て、充分な準備をして実施することが必要である。

ソフトウェアの開発を社外に委託した場合、開発を受託した企業はシステム・テストを終了したものを発注者に納入する。発注者は開発者とは別にシステム・テストベースのテストを行って、そのソフトウェアをそのまま受け入れるかどうかを判断することになる。このためのテストを、「受入れテスト」と呼んでいる。

「運用テスト」

システム・テスト、または受入れテストが終われば、次は「運用テスト」と呼ばれるテストを実施する。このテストは開発者の手を離れて、このソフトウェアが稼働を開始した後このソフトウェアを使用して実際に業務を遂行するユーザと、この情報システムの運用を担当する運用者の両方が、それぞれの立場からこの情報システムを使用して、それぞれの作業を満足に遂行できるかどうかの確認を行うためのテストである。

運用テストのテスト・ケースはユーザと運用者が設定するので、どのような設定方法をとるのかということについて開発者が関知するところではない。しかし、基本的には経験ベースのものになるものと考えられる。

このテストの終了基準は、ユーザと運用者がともに「OK」を出すことである。ユーザや運用者がどういう基準で「OK」を出すのかということについて、開発者は基本的に関与しなくて良い。

システム・テストまでのテストは、検証に重きが置かれていた。しかし運用テストでは、妥

[1] テスト期間中に行うソフトウェア全体を対象にした残存欠陥数の推定方法は、第 32 章で述べる。

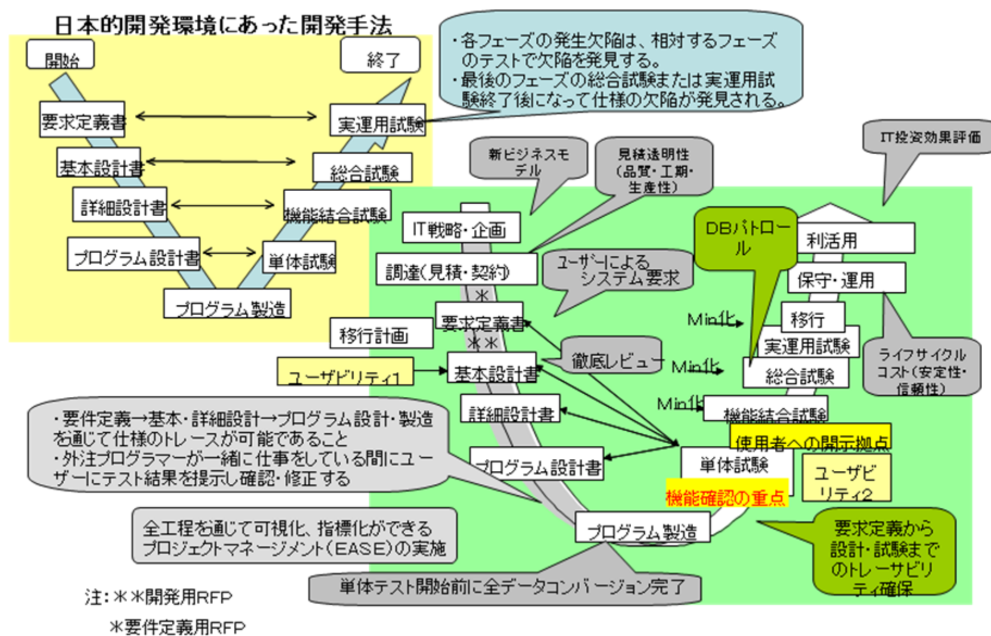
当性確認に重きが置かれることある。

このテストが終了すれば、後は移行（切り替え）作業を行い、その後このソフトウェアの本番稼働が始まる。広い意味の開発組織では、ここで狭義の開発作業を終了して、この後保守[2]の作業が開始されることになる。

「単体機能テスト」

これまで述べてきたテストの方法では、ソフトウェアの機能の全体的な確認はシステム・テスト、または受け入れテストの段階でなされることになる。このソフトウェアの開発がウォータフォール型で行われたものとすると、この確認の時期は開発開始からたいへん遅くなってしまい、次の 2 つの観点で好ましいことではない。

- バリー・ベーム氏が述べているように、機能の確認は開発のできるだけ早い段階で行うことが望ましい[BOE81]。
- ソフトウェアの開発を外部に委託した場合、システム・テストの段階では実際にプログラム開発を担当したプログラマはすでに開発の現場から去っており、そこでプログラム修正が発生すると、実際に開発を行ったプログラマではなく、その時開発作業に従事している別のソフトウェア技術者が行うことになる。このためプログラム修正の効率は、必ずしも良くない。



図表 31-4　 U 字型開発手順（[JUAS05]より）

この 2 つの問題を解決するために、ユーザの立場での機能確認をもっと早い時期に行うことを日本情報システム・ユーザー協会（JUAS）は提案している。具体的には、単体テストを終了

---

[2] 保守については、第 33 章で述べる。

したモジュールを開発者から受け取って、ユーザ企業の技術者ができる範囲での機能確認のテストを順次実施しようというものである。このテストを JUAS では、「単体機能テスト」と呼んでいる。

このテストの段階で実施できるものは、前述の通り機能の全てではない。しかし計算を含む基本的な処理の内容や、ディスプレイや帳票への出力の形式などはこの段階で確認することが可能である。

この段階に引き続き、結合テストの段階でもできる範囲のテストを継続すると良い。単に機能の確認にとどまらず、非機能要求で確認できるものがあればそれも対象にすると良い。このような観点で機能などの確認を早期から行うことで、スケジュール上にも効果がある。JUAS はこのテストを、U 字型開発手順の中で提案している [3][JUA05]。U 字型開発手順の図を、図表 31-4 として再録する。

「確認テスト」（再テスト）

システム・テストや保守の段階で欠陥を発見すると、その欠陥を発見したテスト担当者はしかるべき手順に則ってその欠陥の報告を行う。その報告が最終的に担当するソフトウェア技術者の手元に届いて、そこで原因の究明と欠陥の除去、開発者として欠陥が除去できたことの確認が行われる。

その確認が終わるとその報告は当初とは逆のルートを通って、テスト担当者の手元に戻る。そこでテスト担当者は欠陥を発見した時のものと同じテストを再度行って、本当に欠陥が除去されていることを確認する。この 2 回目のテスト担当者が行う欠陥除去の確認のためのテストを「確認テスト」、または「再テスト」と呼んでいる。

「回帰テスト」

確認テストがシステム・テスト期間の中などで発見された欠陥の除去を確認するテストであるのに対して、「回帰テスト」は確認テストと同じタイミングと状況の下で、この欠陥除去のためのプログラムの修正で発見された欠陥以外の機能などが損なわれていないことを確認するためのテストである。

実際システム・テストや保守の段階で欠陥を発見し、それを除去するためにプログラムの修正を行う際に、既存の、今まで問題なく稼働していた機能を損なってしまうことがある。これを、「デグレード」という。回帰テストの機能を持っていないと、このような場合の、損なわれた機能という新たな欠陥を発見し、それを除去するための対応はたいへんに難しい。

ちなみに米国での調査によると、デグレードは 1%から 20%程度、平均では 7%程度の発生率であるという[JON96c]。システム・テストは、積み上げの作業である。テストの計画を立て、手順を決め、その手順に従って着実に一歩一歩機能の確認を進める。その過程で、以前に「OK」と確認された機能を再度テストの対象に取り上げて確認するという仕組みは存在しない。そうする余裕も、普通はない。このため後のテストである欠陥を発見し、その修正で以前に「OK」と確認した機能を損なっても、それを認識して取り除くことが現実は非常に難しい。回帰テストは、この作業を行うためのきっかけを与える。これは、たいへん貴重である。

回帰テストはこのような状況でたいへん有効なテストの方法で、ソフトウェア開発組織はすべからくこの機能を持つべきである。しかし現実は、この充分な機能を用意することはたいへ

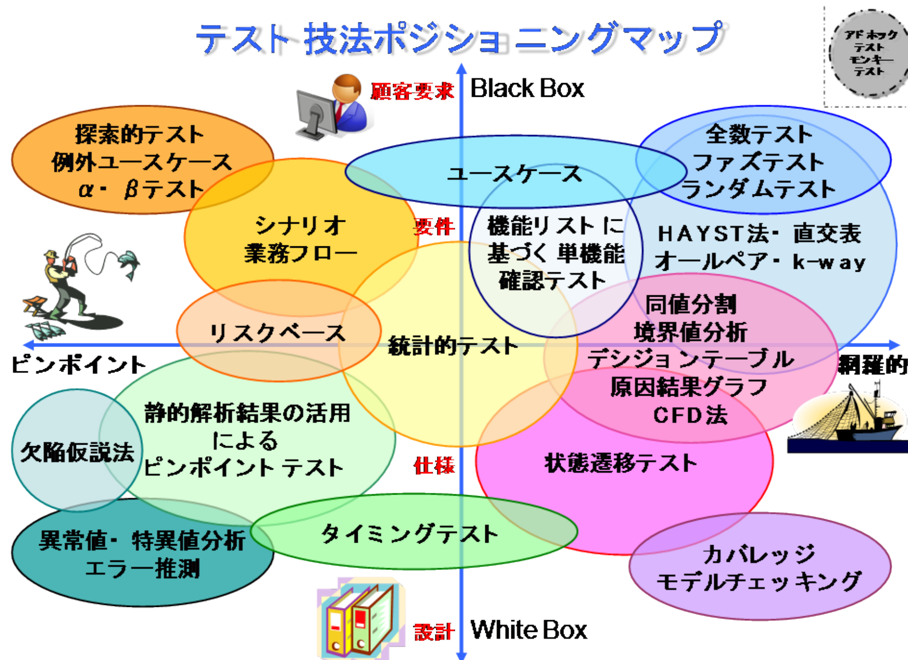
[3] U 字型開発については、すでに第 13 章で述べた。

んに難しい。

その他のテスト

平成 22 年秋に、情報処理推進機構（IPA）は「高信頼化ソフトウェアのための開発手法ガイドブック」と題する資料をインターネットで公開した。この資料は、いかにしてソフトウェアの信頼性を上げるかについて、様々な角度から検討したものである[IPA10a]。

その中に、テストについての素晴らしい図が掲示されている。それを図表 31-5 として掲示する。ここでは図を掲載するだけでその詳細には触れないが、テストには多くの種類があることが分かる。基本的には目的に合わせて、これらのテストを使い分けなければならない。



図表 31-5　テスト技法のポジショニングマップ（[IPA10a]より）

「移行」

運用テストが終わると、本番開始に備えて「移行」作業が実施される [4]。移行には、ライブラリの移行とデータの移行がある。

ライブラリの移行とは、テスト段階にはプログラムにテスト用の機能などが入っている場合などではそのテスト用の機能などを取り除いて、本番用のプログラムを具備する作業である。

またデータの移行とは、本番の実稼働に備えてマスター・ファイルなどを準備する作業である。最近は、全く新しい作業をシステム化するというケースが少なくなった。つまりこれまで何らかのシステムが稼働していて、そのシステムを刷新するケースが増えている。このような場合データの移行で、古い情報システム用のデータを新しいシステム用に作り直す必要がある。そのために、移行用のプログラムが必要になることが多い。

移行は数時間で終わるようなケースから、数日かかるようなケースまで、規模に大きな違い

[4] 移行作業は、テストではない。

がある。数日かかるようなケースでは、移行の機会は年に 2 回ぐらいしかないことがある。移行に伴って、コンピュータの稼働場所（コンピュータ・センター）を移すような場合もある。また全てのユーザを対象に一括で移行するケースと、順次段階的に移行するケースもある。

いずれにしろ移行は一回限りの重要な作業であるので、最初にしっかりとした計画を立て、レビューし、関連部門とよく打ち合わせをして、しっかりと準備して取りかかることが肝要である。

さらに、場合によれば移行に失敗して、元のシステムに戻さなければならないこともある。どのような場合に先に進め、どのような場合に後戻りするのかの判断基準も、準備作業の一環として取り決めておかなければならないこともある。

移行作業が無事に終われば、本番稼働が始まる。

テスト技術者のための資格

この章の 5 つ目のテーマは、テスト技術者の資格についてである。

日本には、（財）日本科学技術連盟の関連の団体として 2005 年 4 月に JSTQB（Japan Software Testing Qualifications Board）が設立され、2006 年 1 月以降毎年 2 回 JSTQB 認定のテスト技術者の試験が、日本科学技術連盟が実施する形で行われている。

試験の種類は、日本では Foundation レベルだけで、2008 年 2 月までの 5 回の試験で 1,825 人の合格者を出した。

JSTQB は、国際的なテスト技術者の認定組織である ISTQB（International Software Testing Qualifications Board）の加盟機関と位置づけされている。その ISTQB に加盟している 32 カ国の団体がそれぞれこのような試験を実施しており、2008 年 6 月時点で日本を含めて Foundation レベルで 77,138 人、Advanced レベルで 5,706 人の合格者を出している。

日本で行われる Foundation レベルの試験の出題は、JSTQB が翻訳し、発行している ISTQB 作成のシラバス[IST11]に準拠して行われ、これは JSTQB のホームページからダウンロードすることがでる。日本では前述の通り試験の科目はまだ Foundation レベルだけだが、ISTQB のホームページからは英語の Advanced レベルのカリキュラムをダウンロードすることができる。

なおこの Foundation レベルのカリキュラムに準拠した試験対策の書籍が、日本語に翻訳されて出版されている[GRA07]。

ちなみに、日本でのソフトウェア技術者の分類と定義で最も権威がある IT スキル標準では、この「テスト技術者」というソフトウェア技術者は定義されていない[5]。

キィワード

検証、妥当性確認、仕様ベースのテスト・ケースの設定法、ブラック・ボックス・テスト、同値分割法、限界値分析、構造ベースのテスト・ケース設定法、ホワイト・ボックス・テスト、グラス・ボックス・テスト、カバレッジ、経験に基づくテスト・ケースの設定法、単体テスト、テスト・ドライバ、スタブ、結合テスト、システム・テスト、受け入れテスト、運用テスト、単体機能テスト、確認テスト、回帰テスト、移行、JSTQB、ISTQB、デグレード

略語

JSTQB：Japan Software Testing Qualifications Board

[5] IT スキル標準については、第 47 章で述べる。

ISTQB：International Software Testing Qualifications Board

人名

バリー・ベーム（Barry w. Boehm）

参考文献とリンク先

[BOE81] Barry w. Boehm, “Software Engineering Economics,” Prentice-Hall, 1981.

[CMM10a] CMMI 成果物チーム、「開発のためのCMMI® 1.3 版 CMMI-DEV, V1.23 CMU/SEI-2010-TR-033 ESC-TR-2010-033 より良い成果物のためのプロセス改善」、カーネギー・メロン大学ソフトウェア工学研究所、2010年

この資料は、次の URL からダウンロードできる（確認日：2017 年（平成 29 年）1 月 25 日）。

http://cmmiinstitute.com/resource/japanese-language-translation-of-cmmi-for-development-v1-3/

[GRA07] ドロシー・グラハム他著、秋山浩一他訳、「ISTQB シラバス準拠 ソフトウェアテストの基礎」、センゲージ・ラーニング（株）、2008 年.

この本の原書は、以下のものである。

Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black, “Foundations of Software Testing : ISTQB Certification,” Cengage Learning, 2007.

[IPA10a] 高信頼ソフトウェア領域 高信頼化のための手法WG、「高信頼化ソフトウェアのための開発手法ガイドブック・予防と検証の事例を中心に・（Ver.1.0）」、情報処理推進機構、2010年9月.

この資料は、以下の URL からダウンロードできる（確認日：2017 年（平成 29 年）1 月 25 日）。

http://sec.ipa.go.jp/reports/20100915.html

[IST11] International Software Testing Qualification Board 著、Japan Software Testing Qualification Board 訳、「テスト技術者資格制度 Foundation Level シラバス 日本語版（Version 2011）」

この資料は、以下の URL からダウンロードできる（確認日：2017 年（平成 29 年）1 月 25 日）。

http://jstqb.jp/dl/JSTQB-Syllabus.Foundation_Version2011.J01.pdf

[JON96c] Capers Jones 著、鶴保征城他監訳、「ソフトウェア開発の定量化手法 第 2 版」、構造計画研究所、1998 年 4 月 20 日.

この本の原書は、以下のものである。

Capers Jones, “Applied Software Measurement Assuring Productivity and Quality Second Edition,” McGraw-Hill, 1996.

[JUAS05] 日本情報システム・ユーザー協会編、「システム・リファレンス・マニュアル（SRM）」、日本情報システム・ユーザー協会、2005 年.

（2008 年（平成 20 年）9 月 18 日　初版作成）
（2011 年（平成 23 年）1 月 9 日　一部追加）
（2016 年（平成 28 年）5 月 20 日　一部修正）

（2017 年（平成 29 年）1 月 25 日　一部追加）