東京ITスクール
TOKYO IT SCHOOL

テスト技法

目次

1. はじめに

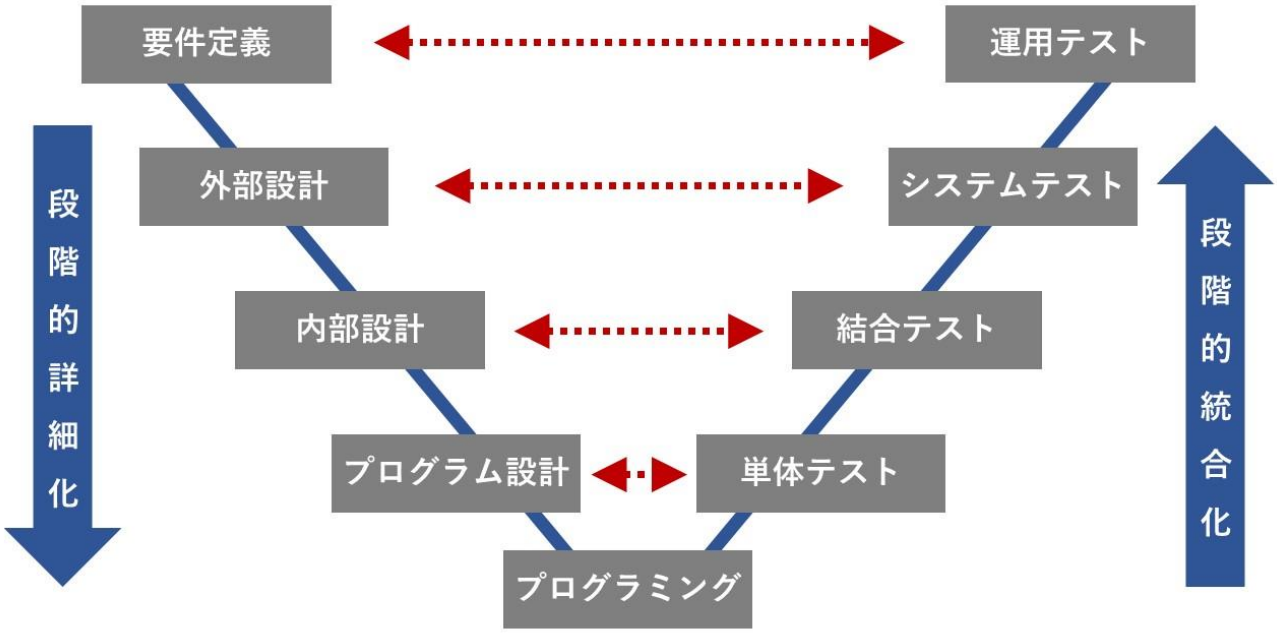
システムの開発はプログラミングまで終わったら完了というわけではありません。作成したプログラムが設計書の内容に沿って動いているかを確認する必要があります。このプログラムを動作確認することを「テスト」と呼びます。

本資料では、代表的なテストの方法を紹介します。テスト作業はエンジニアであれば一度は行う業務です。本資料の内容を参考にしてテストの方法を学びましょう。

2. テストとレビュー

1　設計フェーズとテストフェーズ

ウォーターフォールモデルの設計フェーズ、製造フェーズは段階的詳細化のトップダウンアプローチですが、テストフェーズは段階的統合化のボトムアップアプローチとなります。設定フェーズで作られたテスト計画書をもとにテストが行われ、その結果が設計フェーズにフィードバックされます。例えば、外部設計でシステムテスト計画書を作り、それに基づいてシステムテストを行います。



両者の対応関係は上図のようにＶ字になるため、ウォーターフォールモデルをＶ字型モデルと呼ぶこともあります。

2　テスト手法

システムの品質を保つ上で重要となるのが、テストフェーズです。テストはバグの検出、品質の評価、仕様適合性の検証などを目的として実施されます。テストは開発段階ごとに単体テスト、結合テスト、システムテスト、運用テストの順で進めていきます。以下に詳細を示します。

テスト	テストの対象	テストの種類
単体テスト	モジュールごとにテスト	・ブラックボックステスト ・ホワイトボックステスト
結合テスト	複数のモジュールを組み合わせてテスト	・トップダウンテスト ・ボトムアップテスト ・ビッグバンテスト
システムテスト	完成したシステムをテスト	・機能テスト ・性能テスト ・例外処理テスト ・負荷テスト ・操作性テスト
運用テスト	実際に運用してテスト	・レグレッションテスト

なお各テストとも事前にテストケースを設定し、それに沿ったテストデータを準備しておく必要があります。

3　単体テスト

単体テストは、プログラムを機能単位に分割したモジュール単体で行うテストで、モジュール内論理構造を検証します。モジュールの単位は開発現場ごとに変わります。主なテスト手法として後述する 2 種類のテスト方法があります。

(1) ブラックボックステスト

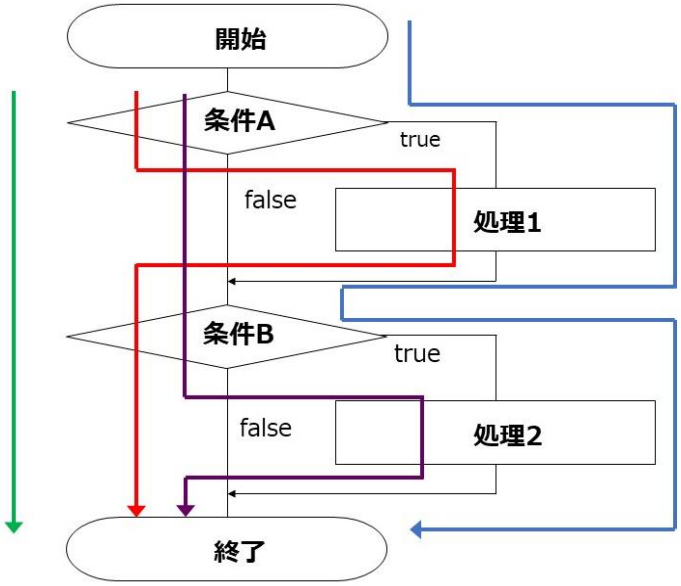
モジュールの内部構造を考慮するのではなく、仕様書どおりに機能するかどうかをテストします。プログラマが設計者の意図した機能を実現できているかどうかのテストであり、主にプログラマ以外の第三者が実施します。テストデータの作成方法に限界値分析と同値分割があります。

名称	説明	正常データ範囲が 15～60 時のテストデータ
限界値分析	有効値と無効値の境界となる値をテストデータとする方法	14，15（最小値の有効値と無効値） 60、61（最大値の有効値と無効値）
同値分割	有効値と無効値のグループに分け、それぞれのグループの代表値をテストデータとする方法	10（最小未満の無効値から代表的な数値） 30（有効値から代表的な数値） 70（最大を超える無効値から代表的な数値）

例えば、年齢(整数値)の正常データ範囲が 15≦年齢≦60 である場合を考えます。限界値分析では有効同値クラスの最小値と最大値、及びそれらを一つ超えた値をテストデータとします。この例では 14、15、60、61 となります。対する同値分割では有効同値、無効同値クラスから、それぞれ代表となる値をテストデータとします。この例では 10、30、70 などが該当します。

(2) ホワイトボックステスト

ホワイトボックステストは、モジュールの内部構造に着目して行うテストです。プログラムの内部構造や論理が記述された内部仕様書に基づくテストであり、主にプログラマ自身が行います。ホワイトボックステストでは、コードカバレッジ(Code Coverage)というプログラムのロジックをどの程度網羅したかを表す基準を設定し、その目標を達成できるようなテストデータを作成します。コードカバレッジでは以下のような基準で網羅性を確かめます。様々な条件を網羅するほど品質は上がりますが、システム完成までの時間を要することにもなります。



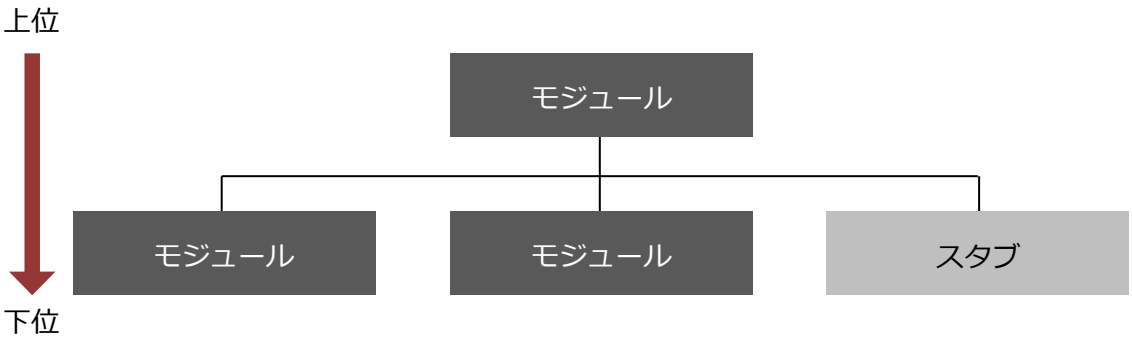
名称	説明
命令網羅	プログラム中の全ての命令を、少なくとも 1 回は実行させる。 例) 開始→条件 A→処理 1→条件 B→処理 2→終了というルートの確認を実施 画像内の青線部分
分岐網羅 (判定条件網羅)	条件分岐において、真偽のパターンを少なくとも 1 回は実行させる。 例) 命令網羅のルートに加え、開始→条件 A→条件 B→終了というルートの確認を実施 画面内の青線と緑線部分
条件網羅	複数の条件分岐が組み合わさっている場合、それぞれにつき真偽のパターンを少なくとも 1 回は実行させる。 例) 分岐網羅のルートに加え、開始→条件 A→処理 1→条件 B→終了、条件 A→条件 B→処理 2→終了という両ルートの確認を実施 画面内のすべての線
複合条件網羅	条件分岐が複合条件の場合、真偽の全ての組み合わせパターンを少なくとも 1 回は実行させる。

4　結合テスト

結合テストは、複数のモジュールを結合させてテストする手法です。モジュール間論理構造を検証します。複数のモジュールを組み合わせて実行した際に、想定される処理の流れ、および結果となるかを確認します。結合テストの代表的な手法を 3 つ紹介します。

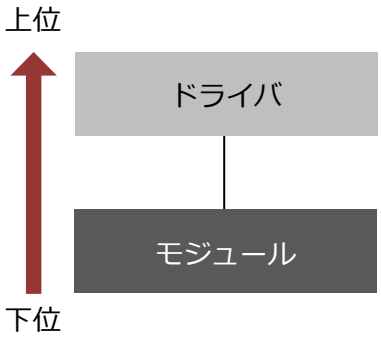
(1) トップダウンテスト

上位のモジュールから下位のモジュールへと順次結合して検証します。下位のモジュールが完成していない場合、スタブという、テスト対象のモジュールからの呼出命令の条件に合わせて値を返すダミーモジュールが必要です。



(2) ボトムアップテスト

下位のモジュールから上位のモジュールへと順次結合して検証します。上位のモジュールが完成していない場合、ドライバという、テスト対象モジュールに引数を渡して呼び出すダミーモジュールが必要です。



(3) ビッグバンテスト

全てのモジュールを結合して、一気に検証するテスト手法です。トップダウンテストやボトムアップテストに比べて手間が少ないという利点がありますが、大規模システムにおいてはバグの特定が困難となるため、主に小規模システムで用いられます。

5　システムテスト

システム全体の動作を、業務上例外的な処理とされるデータや、実際に業務で使うデータ、実際に使うデータを使って検証するテストです。外部設計の要件を満たしているかの確認にもなります。以下のようなものがあります。

名称	説明
機能テスト	システム要件に定められている機能が、全て含まれているかどうかを検証する。
性能テスト	要求される処理能力や応答時間を満たしているかどうかを検証する。
例外処理テスト	間違ったデータを入力したときに、エラーとして認識されるかどうかを検証する。
負荷テスト	量的な負荷をかけ、システムが業務に耐えられるかどうかを検証する。
操作性テスト	操作性の良さや表示されるエラーの分かりやすさを検証する。

6　運用テスト

利用者部門が主体となり、実際の運用と同じ条件でテストし、不都合がないかどうかを検証します。改修の場合では、新たに修正した内容が、今まで正確に動作していた他の機能に悪影響を与えていないかどうかを検証するレグレッションテストなどを先に実施することもあります。

7　レビュー

レビューは、設計の工程ごとに設計品質の評価を行い、各工程が終了したかどうかを判断するために実施する検討会です。デザインレビューとも言います。レビュー資料はレビュー用に用意するのではなく、設計作業時に作られたものを使用します。代表的なレビュー手法に、ウォークスルーとインスペクションがあります。

(1) ウォークスルー

設計上の誤りを早期に発見することを目的として、各設計の終了時点で作成者が、成果物を順に説明し、複数の関係者が設計書を検証、指摘するものです。

(2) インスペクション

あらかじめレビューの参加者が役割を決め、いくつかの選ばれた局面に注意を払い、迅速にレビュー対象を検証します。開発者ではない、第三者が責任のもとに行います。

3. 【補足】テストを行う際のポイント

1　結合試験仕様書の読み方

① シナリオ

結合試験の試験項目を考える場合は、まずどのような機能要件について確認したいのかを考え(これを観点と言います)、次にその機能要件を確認する手順を考えます。この過程で考え出された一連の手順の纏まりをシナリオと呼びます。例えば、何かのデータに関する編集機能を確認したいのであれば、ログイン→該当データの取り出し→データの編集→ログアウトまでに必要な操作と操作の結果をテスト項目化し、シナリオとして纏めます。

今回の例の「社員管理システム」は操作出来る範囲が限られていますので、「正常」「異常」の二つのシナリオに簡単に纏めていますが、商用のシステムにおいては、さらに多数のシナリオが作られます。

② テスト項目

「No」は、テストの通し番号です。テスト項目は一つ一つにナンバリングが施され、一意のナンバーで管理されています。

「分類」は、機能分類を表します。

「テスト手順」は、テストの担当者が実際にシステムに対して行う操作を記したものです。開発者がそのまま結合試験を担当するとは限りませんので、誰が読んでも間違えない書き方になっている必要があります。あいまいな表現はテストの漏れの元です。

「期待する結果」は、テスト手順を実行した際に起こるシステムの反応を表します。この通りの結果が出ていれば、この項目に関するテストは合格ということになります。

「実施日」は、そのテストを実施した日付を記入する欄です。

「確認者」は、そのテストを行った人の名前を記入する欄です。

「合否」は、テスト結果が OK だったのか NG だったのかを記入する欄です。

2　エビデンス

結合試験の実施結果は OK、NG という言葉だけでなく、画面のキャプチャも試験を実施した証拠として残しておく必要があります。これをエビデンスと言います。例として「社員管理システム_結合試験エビデンス」を参照して下さい。

3　障害管理票によるステータス管理

試験結果が NG であった場合は、開発担当者にその情報を渡し、修正を行ってもらった上で再度試験をやり直す必要があります。障害発生→担当者による解析→修正→再試験→OK までの一連のステータスを管理する際に使用されるのが障害管理票です。

実際の使用方法は「社員管理システム_結合試験障害管理票」を参照して下さい。障害発生直後の報告内容の粒度が、その後の解析、修正までの作業のスピードに影響しますので、発生した状況、使用したデータなどは必ず記述しておく必要があります。

4. 品質

1　QCD とは

システム開発において、最初に計画されるのはスケジュール(納期)や費用(コスト)であり、品質に関しては後回しにされてしまうケースが少なからず見受けられます。しかし、この製品の 3 要素である QCD(品質、コスト、納期)はどれも重要かつ合理的な指標であり、バランスを保つことが開発プロジェクト成功への鍵となります。以下に 3 要素の関係を示します。

名称	バランスが保たれた場合	バランスが保たれない場合
品質 Quality	良い	悪い
コスト Cost	費用がかからない	費用がかかる
納期 Delivery	予定通り	遅延
	品質が良いため修正作業などに追われることなく、期日までにシステムをリリース。コストが膨れ上がることもない、など。	品質が悪いためバグが多発し、追加テストを実施するためエンジニアを増員。結果コストが膨れ上がり、納期にも間に合わない、など。

2　品質管理とは

品質管理とは、提供するシステムやサービスが顧客の要求する QCD となるよう技術、統計を駆使し管理していく概念です。品質を維持するための目標、指標を定め、レビューなどを通じ各工程での不具合の修正や工程の見直し、改善を図ります。よく用いられる指標としてはテスト密度(単位当たりのテストケースの数)やバグ密度(単位当たりのバグの数)などがあります。以下にその例を示します。今回の例では、過去プロジェクトより、少し複雑性が増したものを作成する見込みのため、バグ密度を 10%増加させています。実際に、どれほど増減するかは、現場の方針などで異なります。

	システム規模	総バグ数	バグ密度	総バグ数目標値
過去プロジェクトデータ	100kstep	200 件	2.0 件 / kstep	
現行プロジェクトデータ	150kstep		2.2 件 / kstep	330 件

過去の統計と難易度に基づき、やや多めに設定

これは、過去の類似プロジェクトのデータを目安にバグ数を推測する例です。なお kstep とは、意味のあるコードの行数 step の 1000 行分を意味する単位です。

しかし、毎回類似プロジェクトのデータが存在するとは限りません。その場合は IPA(独立行政法人情報処理推進機構)の掲げる指標を 1 つの目安とするのも手でしょう。以下は Java における新規開発、システム改修でのテスト密度、バグ密度の指標です。

	フェーズ	新規開発			システム改修		
		最小値	基準値	最大値	最小値	基準値	最大値
テスト密度 (件 / kstep)	単体テスト	36.1	54.2	81.3	60.2	90.3	135.5
	結合テスト	18.1	27.1	40.7	40.1	60.2	90.3
	システムテスト	5.5	8.2	12.3	10.8	16.2	24.3
バグ密度 (件 / kstep)	単体テスト	1.5	3.0	6.0	1.3	2.6	5.2
	結合テスト	0.8	1.5	3.0	0.7	1.3	2.6
	システムテスト	0.2	0.3	0.6	0.1	0.2	0.4

また、テストを統計的に検討する際の基準として信頼度成長曲線などが用いられます。一般的にテスト開始段階では多くのバグが潜んでいますが、テストを消化するにつれてバグが修正され減少し、品質の高いプログラムとなっていきます。テスト開始後から累積バグ件数をグラフに表すと通常、以下のような曲線、信頼度成長曲線が現れます。傾きが緩やかになるにつれて、バグの発見数が減少し、潜在的なバグ数が少ない、つまり信頼度が高いことを意味します。