SEC BOOKS

組込みソフトウェア開発における品質向上の勧め［テスト編～事例集～］

独立行政法人**情報処理推進機構**　技術本部 ソフトウェア・エンジニアリング・センター　編

はじめに

近年における携帯情報端末、デジタル家電、自動車などの普及と技術革新はとどまるところがありませんが、こうした情報機器のすべてにマイクロコンピュータが搭載され、組込みソフトウェアが動作しています。

組込みソフトウェアは当該製品に寄せられるニーズの拡大と高度化に伴い、品質や信頼性・安全性などが以前にも増して重視されるようになってきましたが、同時に近年のグローバル競争の激化はソフトウェア開発現場に対して、否応なしにコスト削減や開発期間短縮を迫っています。こうした市場が要請する難しい課題に直面する中で、品質確保に向けた十分な検証活動が追いつかず、市場においてソフトウェア不具合によるシステム障害などの問題が引き起こされていると考えられます。一方で、ソフトウェアはどんなにテストしてもバグがゼロにはなり得ないという現実があり、どこまでテストすれば十分であるかの判断に苦慮しているという実態もあります。

こうした背景に基づき独立行政法人 情報処理推進機構(IPA)ソフトウェア・エンジニアリング・センター(SEC)では、品質向上のための『実践的』なV&V(Verification and Validation)改善に向け、ソフトウェア開発現場が抱える課題を抽出し、先進企業におけるそれら課題に対する考え方や対策事例を収集し、ソフトウェアテストの「指針」と「目安」に関する有識者の知見を収集するべく、2011年度に「テスト部会」を発足しました。

本書は、先進企業の組込みソフトウェア開発現場で豊富な経験を持ち、V&V改善を実践・推進している有識者の方々に部会委員として参加いただき、組込みソフトウェアテストに関する考え方や品質向上・効率向上にかかわる知見を整理し、『組込みソフトウェア開発における品質向上の勧め[テスト編～事例集～]』(Recommendations for Improvement of Quality of Embedded Software Development [w/Practical Testing Examples])(略称：SECのテスト事例集)としてまとめたものです。

本書が組込みソフトウェアの設計・テストを担当される方々をはじめ、品質管理に携わる多くの皆様方のV&V実践活動にとって有用な事例情報としてご活用いただければ幸いです。

最後になりますが、本書の編纂に際し、委員会での議論や事例情報の提供、編集にご協力いただいたテスト部会の皆様に改めて御礼申し上げます。

2012年秋

独立行政法人 情報処理推進機構(IPA)
技術本部 ソフトウェア・エンジニアリング・センター(SEC)
組込み系プロジェクト
三原　幸博
石井　正悟
石田　　茂

上梓によせて

本書に取り上げた事例は、先進的事例よりも、識者の方々にとっては「当たり前」と考えられている事例のほうが多いと思います。しかし、「当たり前」と考えていても、その妥当性を論理的に証明するのは難しい、あるいは自分自身でも確信を持つに至らない、というのが実情ではないでしょうか。また、作成者一同も、自社のテストに関する現場の実態は公開できないけれど、ぜひとも他社と比較して自社の取り組みの妥当性を評価したいと常々考えていました。

本書は、論理的に正しさを解説するまでには至らないため、組込みソフトウェア開発リファレンスという位置付けのESTR（Embedded System development Testing Reference）とはせず、その前段の事例集としました。体系的・網羅的・論理的に解説していない本書をあえて出版したのは、「IPAの部会に集まる先進企業の識者達が、そうあるべきと考え、実際そのように実践して成功している」ことを示すことで、読者の皆様が自社の実態と比較し、自社の取り組みの妥当性に自信を持ち、そして他者への説得の際に有益な情報となるであろうと考えた次第です。

本書では、ソフトウェア品質メトリクスの基準値に関しても具体的な数値を示している事例が幾つかあります。それら基準値についても論理的に正当性を示してはいませんが、事例を提供してくださった企業様の現場において実際に使って有効であることが実証された数値です。読者の皆様が自組織の基準値を設定される際の参考に、また、既に使われている基準値があれば、自組織の基準値と本書に示した基準値を比較して、自組織の数値の妥当性について議論するきっかけにしていただけることを、作成者一同、切に願っています。

2012年秋
組込み系プロジェクト
テスト部会

目次

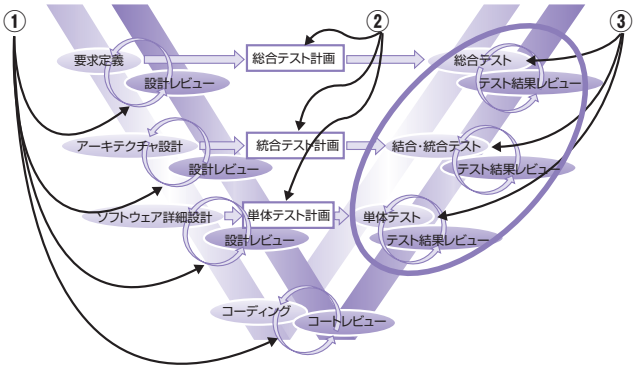
本書の使い方

●本書の対象領域

W字モデル開発プロセスにおけるV&Vの対象領域としては、次の領域があります。

①設計レビュー/モデル検証/コードレビュー、
②テスト計画、
③テスト実施/結果レビュー

SECがソフトウェア開発現場の方々に対して行ったアンケート/インタビューでいただいたコメントから、実際の組込みソフトウェア現場ではとくに②と③に大きな、そして差し迫った課題があることがうかがわれたことから、本書では②と③の所謂狭義のテスト領域を主な対象としました。



●本書の構成と利用方法

本書では、上記②、③の領域で組込みソフトウェアテストが抱える課題を大きく以下の３つの視点で分類し、それぞれの視点での個別課題とその対策事例を整理しています。

Part1. テストの役割と限界
Part2. テストへの要求と対応状況
Part3. テストの基本的テクニック

本書では先進企業における事例を紹介し、どのような場面でどのように対処したかを解説しています。それらは必ずしもあらゆる製品領域の、あらゆる制約下の組込みソフトウェア開発においても同等に有用性が当てはまるとは限りません。しかし、いずれの事例も実際の開発現場における成功事例ですので、皆様が組込みソフトウェア開発のテストに関して、同様の場面に直面し、対策を検討される際のご参考となることを想定しています。

●個別事例の読み方

本書に掲載している事例は、1件ずつ以下の体裁で整理しています。

見出し

ESxRシリーズの“作法”、“作法詳細”にならい、各事例での対策における基本的な考え方あるいは重要事項を端的に記述しています。

どの企業においても外部には提供し難い、企業の貴重な財産である事例をご提供いただき、なるべく多くの事例をご紹介するため、分類した3つの視点との関係が多少直感的でない部分もありますことをご了承ください。

解説

開発現場で実践されている事例の内容を図表なども活用して解説しています。

留意点

事例の内容に関して留意すべき点や、関連するトピックスなどを記述しています。

事例

事例に関する実際の具体例をポイントを示しながら記述しています。

Column

事例解説に関連した事項を記述しています。

(注)本書の事例は、各社で使用している情報システム用語で記述しています。

Part 1

テストの役割と限界

ソフトウェアの品質作り込みは、設計・製造フェーズまでで行われ、テストは設計・製造フェーズまでで作り込まれたバグを市場に出さないようにする、いわば品質を守る最後の砦です。品質を守るというテストの役割の認識を誤り、テストに過大な期待をすると、テストフェーズに至るまでの上流工程での品質作り込み意識が低下するという弊害もあり得ます。

また、ソフトウェアテストに完全ということはあり得ません。どんなに優れた手法・技法を用い、どんなにコストをかけても潜在バグがゼロになることはあり得ません。もう一方で、テストにかけられるコストには限りがあります。つまり、時間的・経済的な理由からテストにかけられるコストの許容範囲内でテストを完了させ、かつ製品への品質要求を満たさなければなりません。そのため、相反する品質要求とコストのバランスをどのようにとるかの判断が非常に重要で、しかも悩ましいところです。

本章では、テスト実務者をはじめ経営者、品質管理責任者にテストの役割と限界を理解していただきテストの戦略、品質基準を考える際に参考にしていただくことを念頭に、先人達が単体/結合/総合テストの各フェーズをどのような役割と位置付けてきたか、品質要求とコストのバランスをどのようにとってきたか、について事例を紹介します。品質要求とコストのバランスは品質メトリクスの基準値(目標値と終了条件)を調整することになるので、どのように基準値を設定しているかの考え方を中心に、幾つかの具体的な基準値についても紹介します。

1.1 テストの役割

1.1.1 テストの目的と位置付けの明確化

関連：1.2.2
3.1.1

単体、結合、総合テストフェーズの各フェーズ及び各フェーズ内での各種テストの目的と位置付けを明確に定義し共有する

解説

単体、結合、総合いずれのテストフェーズにおいても機能テストが実施されるが、フェーズごとに確認すべきことは異なるものである。

①単体における機能テスト

すべての API に関して、パラメータの正常系と異常系に関する動作確認を行う。

②結合における機能テスト

モジュール間で定義したインターフェースで使われるパラメータのすべての正常系と異常系に関する動作確認を行う。完全な網羅性確保が非効率的・非現実的であれば、同値分割、境界値分析などによってパラメータがとり得る値を論理的に網羅するようにテスト設計する。

③総合における機能テスト

成果物全体の動作確認を行うため、機能仕様書からテストケースを抽出する所謂ブラックボックステストで正常系と異常系に関する動作確認を行う。

〈機能テスト〉

単体テスト	モジュールごとの動作確認
結合テスト	モジュール間I/F、連携動作の確認
総合テスト	成果物全体の動作確認

単体テストと結合テストは往々にしてその定義があいまいになりがちであり、同じ社内であっても部門により、また、対象とする職域・階層によっても異なることもある。比較的定義が明確と言われるハードウェアにおいても、その担当分野によっては具体的に指し示すテスト対象にズレがあることも珍しくない。

統一した定義が必ずしも全組織・全部門にとって最適ではないかもしれないが、組織部門として統一された定義が共有されていることが大切である。まず、統一した定義を共有した上で調整可能な運用ルールとすることが望ましい。

近年では、モジュール単位のテストを指して単体テストと呼ぶ傾向があり、モジュールの粒度にも様々な定義が存在する。このような状況において例えば、アーキテクチャ設計段階で作成し全開発者が共有するドキュメントとしてこうしたモジュールを説明する機能仕様書が存在するならば、その仕様書をベースとしてモジュールの粒度を共有することが出来る。その上で、モジュール単位のテストを単体テストと定義すれば、プロジェクト内だけでなく組織としても「モジュールの粒度は当該仕様書をベースとする」、「モジュール単位のテストを単体と定義する」といった開発規約を定めることにより共有を図ることが出来る。

留意点

「単体テストは行わず、デバッグに含める」という定義も、明確化の一例であって間違いとは言えない。考え方が組織として統一され、関係者間で認識が共有されていることが重要である。定義を統一することによって共通のモノサシで測ることが出来、結果を組織横断的に比較評価出来るようになる。

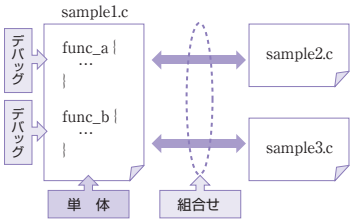
事例

ある開発組織でのテスト定義例を以下に示す。

■単体テストと結合テストの定義例

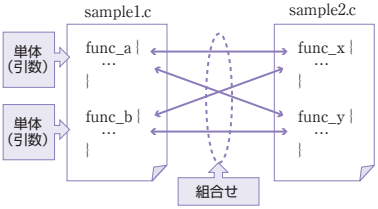
・モジュールのテストを単体テストと称する場合

関数単位の動作はデバッグであって、単体テストではモジュールの動作確認を行い、結合テストではモジュール間インターフェースの動作確認を行う。



・関数のテストを単体テストと称する場合

単体テストでは関数への引数パターンをテストケースとして、関数単位の動作確認を行う。結合テストでは関数を組み合わせたモジュールの動作確認を行う。



1.1.2 単体テスト容易性の向上

関連：1.3.3

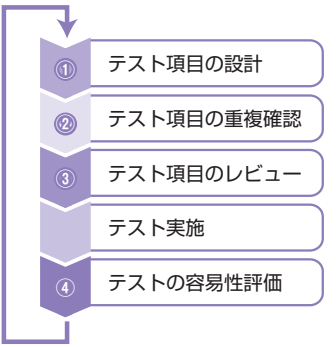
単体テストでは、テスト実施の容易性向上による効率改善のための評価を行い、次のテストフェーズや次の開発に活かす。

解説

単体テストフェーズでは、システム構成要素のすべてがそろっていないのが組込みシステムの特徴であって、すべてがそろっていないためテスト実施に手間がかかるものである。次の開発や次のテストフェーズのためにもテスト実施効率の評価を行うとよい。

単体テストとは、仕様やソースコードを分析して作成する関数またはモジュール単位レベルのテストである。その設計・実施手順の一例を以下に示す。

この例ではテスト実施後に容易性評価を行い、テスト項目消化の阻害要因を明らかにして、次回以降の単体テスト仕様書レビュー用チェックリストに反映するという手順が示されている。



①テスト項目の設計

実施するテスト設計の特性に応じた技法を用いてテスト項目を作成する。

また、実施に際してデバイスドライバ / スタブが必要な場合は、そのデバイスドライバ / スタブの開発も同時に行う。

②テスト項目の重複確認

テスト設計者本人により重複するテスト項目を確認し、重複しているテスト項目を統合することでテストの十分性・網羅性を保ちながらテスト項目数の削減を行い、テストの効率化を図る。

③テスト項目のレビュー

レビュアを交えて、単体テスト仕様書レビュー用チェックリストなどのレビュー基準に従ってレビューを行い、テスト項目の無駄や重複 / 抜け / 観点漏れ / 確認方法の妥当性などをチェックする。

④テストの容易性評価

設計された単体テスト実施後、テスト時間当たりのテスト項目消化率などを

測定し、テストの容易性を判断し、容易性を向上させることでの効率化改善を図る。結果は、結合テスト以降のテストフェーズや、次の開発に活かす。

留意点

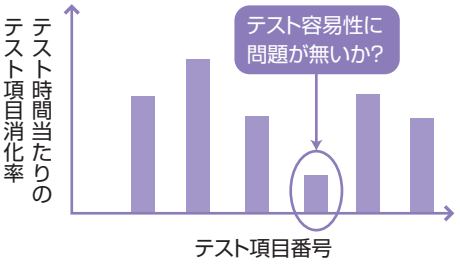
◆チェックリスト管理の弊害の 1 つにリストの肥大化がある。これはチェック項目の単純追加が原因であり、項目数が多くなってくると機械的作業となってチェック行為の形骸化を引き起こし、効率化の阻害要因となる。
◆あえてチェックリストに載せる必要が無くなったチェック項目は除外するなどの定期メンテナンスにより、チェックリストの肥大化を防ぐことが必要である。
◆チェックリストの記述の抽象度にも注意が必要である。低すぎればほとんどが非該当になってしまい、逆に高すぎればすべてが該当してチェック行為が形骸化する。

事例

品質指標としてテスト項目消化率を採用した例を以下に示す。

■テスト実施に要した時間当たりのテスト項目消化数を計測して、テスト実施容易性に問題が無いかをチェックする。

■テスト項目消化率の状況を把握する際には、テスト項目単位に集計・分析している。このグラフからテスト項目消化率が低いテスト項目について、実施容易性阻害要因が人的要因であるか、環境要因であるか、あるいは技術要因であるかを分析し、それぞれに応じた改善を行っている。



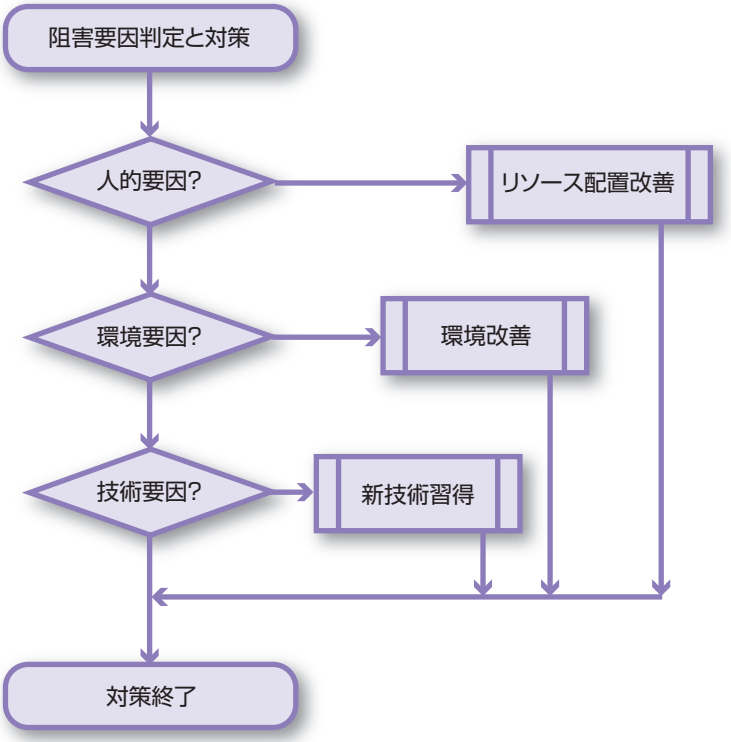
■テスト項目消化率が低い場合、原因は大きく人的要因、環境要因、技術要因に分類できると考えられる。各要因の具体例と対応策例を以下に示した。

●人的要因：ドメイン知識不足、スキル不足などが考えられ、対応策としては、必要な知識、スキルの再教育や、要員の交替などを行う。

●環境要因：テストツールの機能不足、テスト用試作機台数の不足、開発用サーバのパワー不足などが考えられ、対応策としては、テスト環境の充実、効率を向上させる支援・管理ツールの導入などを行う。

●技術要因：再現のための準備（ビッグデータを準備するなど）、遷移の中間状態でしか発生しないなどの他、新技術・未経験技術などの原因も

考えられる。この場合の対応策としては、教育による技術レベルの向上が必要である。



1.1.3 模擬環境と実機環境の相互補完

関連:2.1.7
3.2.1

**模擬環境でテストしても、実機環境でのテストも不可欠である。
模擬環境と実機環境で実施するテスト項目の
切り分けを明確にして、漏れと重複を排除する。**

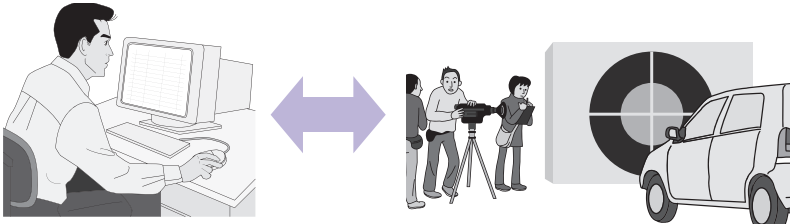
解説

模擬環境は実機環境を完全に置き換えるものではない。模擬環境と実機環境では、できること/できないこと、実行が容易なこと/実行が困難なことが異なり、相互に補完する関係にある。また、そのように模擬環境を構築すべきである。

シミュレータなど模擬環境によるテストは、効率化にとって有効な手段であるが、事前に想定した動作・現象を再現させて検証を行うものであるため、ハードウェア設定・制御やリソース/タイミングの相違など想定外の問題は検証が困難である。こうした模擬環境でテストできない項目については、実機環境でのテスト実施が不可欠である。

一方、実機環境でも、テストに必要なすべての環境や条件を作り出せるものではないので、範囲と理由を明確にして実機環境でしか確認できない項目に絞る。

どちらの環境でもテスト実施が可能ならば、どちらで実行するのが効率的かを考えてテスト設計する。



シミュレーション　　実機テスト

留意点

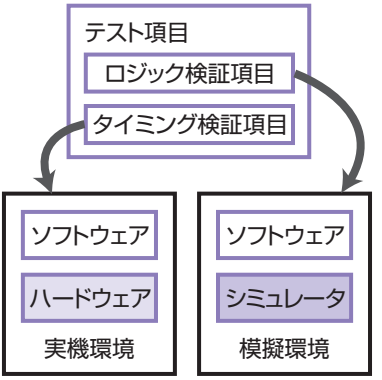
◆模擬環境でテストする場合は、可能であれば、どのテスト項目が実機と同等であるかの根拠を明確にしておくことがアカウンタビリティ確保のために有用である。
◆例えば、システムシミュレータの中には、100%実機と同じソースコードを使い、非常に高い精度（命令レベルで 98% など）で実機と同じタイミングを再現しているものもある。その場合には「98% 実機と同等とみなせる」とするなど。

事例

ある製品開発における事例を以下に示す。

■ハードウェア開発と並行してソフトウェア開発を実施するために、ハードウェア仕様書に記載された入出力インターフェースを忠実にモデル化した模擬環境を構築し、そこで製品に搭載するデバイスドライバや上位層のアプリケーションソフトウェアなどのテストを実施した。

■実機のハードウェア仕様のインターフェースを忠実に再現しているので論理的な動作は実機環境と同じと考えられるため、ロジック検証のためのテスト項目はすべて模擬環境で確認することとした。ただし、性能測定を含め、タイミング依存となるテスト項目は、模擬環境上でロジックの問題をすべて解決した後、実機環境で実際に動作させ最終確認を実施した。



この事例では、実機環境と模擬環境で同じハードウェア仕様書通りに動作するものを使用しているため、両環境で論理的な動作の違いは無いことを根拠として、ロジック検証は模擬環境、タイミング検証を実機環境という切り分けを行った。

Column

シミュレータとエミュレータの用語説明を付録に記したが、定義に拘ると言葉遊びになってしまう。観点が異なるゆえの意見の相違であろう。一般的には、シミュレータのほうが広義で使われるようである。

シミュレーション環境を活用する上で重要なのは、以下のことを明確にすることである。

・何の目的でそのシミュレータを用いるのか
・そのためには、どこまでの動作が実機と同じである必要があるのか。つまり、どこから先をモデル化してどの程度抽象化するのか
これらを明確にしてシミュレーション環境を構築し、テスト項目を抽出すること。この判断/明確化を誤ると、
・構築においては、無駄なシミュレーション環境構築作業をした挙句、遅くて使い物にならないシミュレータを構築してしまう。ハードウェア検証用シミュレータをそのままソフトウェア検証用に用いて失敗するのもこのケースである。
・テスト項目抽出においては、シミュレータ上で意味の無いテストを実施して、実機テストとの重複実行をしてしまう。

1.1.4 ソフトウェアタイプに応じたテスト項目チェックリスト

関連:1.2.2

**テスト設計においては、
対象ソフトウェアのタイプに応じたチェックリストを作成し、
テストの漏れを防止する。**

解説

テストするべき対象製品機能は、GUI 画面やコマンドなど様々なソフトウェアタイプとして実装されるため、おのおののタイプや利用者の操作場面を想定したテスト観点に基づいてテスト項目を考えることが大切である。

またソフトウェアをインストール・アンインストールする作業が利用者側で行われる可能性があるような製品の場合、その動作確認テストも行う必要がある。更に、当該製品上の機能を使用するに際してライセンス取得が必要になる場合がある。このような場合のテストにあたっても、注意すべき事項を明確にするべきである。

留意点

◆ツールを活用した自動化では、概して制御系はハードルが高い。ソフトウェアのうちハードウェアに依存する部分や外部 API は、開発の都度に詳細仕様が異なる設計となりがちであるなどが主な理由である。
◆ GUI 系は制御系に比べれば自動化しやすいが、GUI であるがゆえに変更に対する柔軟性もあり、仕様変更対応の負荷が高くなるという傾向がある。

事例

ある端末装置開発を行っている組織で運用している、テスト設計観点の例を示す。

■GUI

ユーザが最も利用するインターフェースであるため、網羅度を上げてテストする必要がある。

・すべての画面のすべての入力項目で境界値テスト(境界値分析技法)を実施すること。
(例)1-100 の範囲の項目の場合、0 と101 は無効になり、 1 と100 は正常に処理されること
・無効な入力値についてはGUIでガードがかかっていること。
(例)1-100 の範囲の項目の場合、英字、2バイト文字、0以下、101以上は入力できない、またはOKボタン押下時にエラーが表示されること。
・プロジェクトで規定しているGUI設計ガイドに沿っていること。
・他の画面と統一がとれていること(表示、操作方法)。
・ユーザが操作しやすいこと(マニュアルがなくても感覚的に操作できるとよい)。
・ウイザードによる画面遷移がある場合、すべての画面で「戻る」や「キャンセル」のテストをすること。
・ショートカットキーもすべてテストすること。
・処理が長いときは、途中でキャンセルが出来ること。
・ファイル名の指定で、存在しないファイル名を指定した場合、適切なメッセージを表示すること。
・すべてのメッセージが適切であること。

■コマンド

GUIと同様、ユーザがよく利用するインターフェースであるため、十分テストする必要がある。

・すべてのコマンドのすべてのオプションで境界値テスト(境界値分析技法)を実施すること。
(例)1-100 の範囲のオプションの場合、0 と101 は無効になり、 1 と100 は正常に処理されること。
・コマンドの戻り値を確認すること。
・使用例通りに入力し、期待する動作をすること。
・コマンドのオプションの組み合わせをテストすること。

・オプションに無効な値を入力しても、適切な処理/対応をすること。
・処理が長いときは、途中でキャンセルが出来ること。
・ファイル名の指定で、存在しないファイル名を指定した場合、適切なメッセージを表示すること。
・すべてのメッセージ内容が適切であること。

■ API・マクロ

API やマクロを利用するプログラムが無い場合は、それらを利用するテストプログラムを作成してテストを実施する。

・すべてのパラメータのテストを実施すること。
・戻り値を確認すること。
・API/ マクロ利用にあたって、事前条件がある場合、その条件を満たしてテストを実施すること。
・API/ マクロ利用にあたって、事前条件がある場合、その条件を満たさずにテストを実施し正しくエラーが返されること。
・使用例通りに使用し、期待する動作をすること。

1.1.5 結合テスト項目抽出のための３つの基本方針

関連：1.1.1
1.3.3
3.1.1

結合テスト設計は基本方針を明確にして、テスト計画/テスト項目を抽出する。

解説

ここでは、以下の３つを基本方針とした場合について解説する。

◆追加・変更した単機能は100%網羅する（基本は全機能100%網羅であるが非現実的である）

◆単機能の組み合わせはホワイトボックス的に重点テスト範囲を網羅する（すべての組み合わせ実施は非現実的である）

◆ユーザの観点でホワイトボックステストを補完する

- ✔ 単機能はすべてテストする
- ✔ 単機能の組み合わせの観点
- ✔ ユーザの観点

結合テストでは以下のようにテスト設計方針を明確にし、方針に基づいたテスト設計を行う。

①単機能はすべてテストする

単機能とは「利用者に見えるレベルの入出力を持つ最少単位の機能」であり、確認可能な入出力結果を得られる単位で１項目とする。新機能に関してはすべての単機能を抽出して 100% 網羅する。また、既存機能に関しても、新規にテスト項目を作成する必要が無いか十分確認する。

②単機能の組み合わせの観点を持つ

単機能のテスト項目の抽出後は、テスト仕様書に記載されている関連の強い機能の組み合わせについてテスト項目を作成する。更に、テスト仕様書に明確に記載されていない機能組み合わせのテスト項目についても、その必要性を検討するべきである。

③ユーザ観点を考慮する

テスト項目作成にあたっては仕様通りに動作することのテストだけではなく、ユーザ観点でのテストも必要である。

・作り手の意図を超えてユーザが運用する可能性のあるものは無いか
・間違った運用をしても問題にならないか
・ユーザが利用しやすい操作・機能・表示となっているか

などの観点でテストを実施する。

事例

結合テスト項目を設計する場合の方針について、ある企業での具体例を以下に示す。

①単機能はすべてテストする

・単機能であっても確認手段が無い場合は、複数の単機能を合わせて確認が可能になるようにしたものを１項目とする。
例えば、単機能の確認のために、デバイスドライバやスタブの開発をする手間を省くために、デバイスドライバ、スタブの代替として他の単機能モジュールを組み合わせて動作確認をする場合がある。

・仕様書上でテスト項目を再確認し、新規機能に関してはすべて網羅されていることを確認する。また、既存機能に関しても関連性を十分考慮し、新規にテスト項目を作成する必要が無いか十分に確認する。
例えば、あるワークフローの処理に新規機能が追加された場合、機能の観点からは閉じていてもその新規機能の確認だけではなく、ワークフローの中での他の機能との処理のタイミングや見落としによるトラブルを防ぐために、幾つかの既存のユースケースや新規機能と関連した既存部分のワークフロー処理の確認も行う。

②単機能と他要因との組み合わせ方を考慮する

機能組み合わせテスト項目は、組み合わせる因子を設計書などから抽出し(機能、環境、障害、タイミング)、因子を図表などにまとめ、必要な組み合わせを抽出する。

例えば、組み合わせによる評価を網羅性を保ちながら効率よく実施するために直行表などの手法を使った評価を実施したり、状態遷移図を作成したあと、すべての遷移を確認するために、状態遷移図からテストケースの作成を行う。

③ユーザ観点を考慮する

GUI 画面で数値のみ入力項目に数値以外を入力、設定処理の途中でのキャンセルなどの確認を実施する。

例えば、数値のみの入力項目に文字を入力してエラー表示し再入力出来ることの確認や、範囲外の数値を入力してエラー処理をきちんと行い誤動作しないことなどを確認する。また、処理途中でのキャンセル処理が正常に動作し、処理を無効にし、再入力可能であることなどを確認する。

1.1.6 差分開発での影響範囲解析への構造解析ツール活用

関連：1.2.1
2.1.6
3.1.1

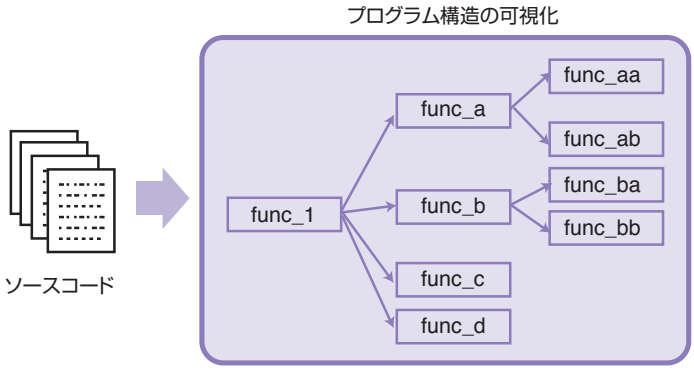
差分開発・派生開発の上流工程でテスト計画を立案する際には、ソースコード構造解析などのツールを活用して影響範囲を検討する。

解説

ソースコード構造解析ツールは、一般的には改造前の影響度分析やリファクタリング目的で用いたり詳細設計書を代替したりするために用いられるものであり、所謂テストツールとは異なる。しかし、ソースコード構造解析ツールによるプログラム構造の可視化と影響度分析は、設計ドキュメントが不十分あるいは最新ではない流用・改造ソースコードのテストケース抽出に有用である。

差分開発において、全体を一律にテスト実施しようとするとテスト工数の浪費に繋がりかねないが、かといって一様にテスト簡略化してはテスト漏れになる可能性が高い。そこで、重点的にテスト実施すべき範囲を決定する判断材料として、ソースコードツール実行結果を活用できる場合がある。

また、従来から流用し続け、動作的にも安定している、所謂塩漬けソースコードは基本的にテスト不要とするのが一般的であろう。しかし、改造による影響が無いということの裏付けを得ることは必要であり、ソースコード構造解析ツールでの自動分析結果が動的検証結果を補足・補完するための有効なエビデンスとなる。例えば下図において func_c が改造対象の場合、func_aa ～ func_bb には改造の影響が及ばないことが分かる。



留意点

流用部分のドキュメントが存在しなかったり、あるいは存在してもタイムリーに更新されていなかったり、ということはあり得る。それを理由に確認出来ない、ドキュメント作成からやり直すべきであるということは正論かも知れないが、投入マンパワーや日程的には難しい場合もある。このような場合に、ツール活用を試してみる価値がある。

事例

過去の塩漬けソースコードを解析する必要性が生じたが、そのためにソースコードへのコメント挿入などの効率の悪い作業は行いたくなかったため、ツールを組み合わせて構造解析した事例を以下に示す。

■ある製品開発にて流用すべき実際のソースコードを確認したところ、コメントがなかったり、コメントは存在するものの実行するソース行の実態に合致していない部分などが多数含まれることが判明した。コメント記述の有効性が疑わしい部分では、コメントの有効性確認にも解析結果が有効に活用できるはずという予測に基づき、以下のような２種類のツールを組み合わせて実施した。

・ツールA：このツールの解析結果はその表示機能が優れており、要素間の依存関係が直感的で分かりやすい。

・ツールB：実行ソース行を読んで関数/変数/インクルードレベルでプログラム内部構造の参照関係を解析するもので、解析結果とソースコードの連動表示が可能である。

■このツールBの解析結果をツールAへの入力とすることによって、塩漬けのソースコードにコメント行追加のような手を加えることなく、要素間依存関係を分かりやすい表形式で表示することが出来た。

Column

アリアン５型ロケット爆発事故の原因は、旧型ロケットでは通過しなかった既存ソフトウェアの潜在バグだった。従来から流用し続け、動作的にも安定していたことから未通過パスの存在が見逃され、検証が漏れたことによる。

1.2 テストの限界

1.2.1 リグレッションテスト資産の蓄積と活用

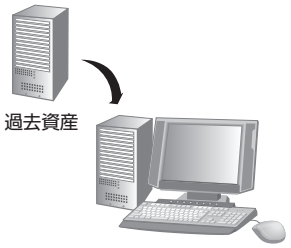
関連:3.1.3
1.4.1

リグレッションテストのテストケースは、テスト資産を蓄積し、狙いを明確にして見直しを繰り返す。リグレッションテストを効率よく実施するため、テスト自動化を推進する。

解説

リグレッションテストとは、障害修正確認と同時に、その修正が他の部分に影響しないかどうか、つまりデグレード防止のために実施するテストであるため、一度実施したテストケースをすべて再実行することが理想である。しかし、バージョンアップする都度テストケースは追加されるため、バージョンアップを重ねるうちにテストケース数が膨大になり、次のバージョン開発ですべて再実行すると、実施できる限界を超えてしまう。よって、開発中のテスト資産蓄積とともに、開発完了したら毎回テストケースの見直しが必要になる。

また、極力自動化を進めて効率化するにはトレードオフも生じる。テスト自動化に要するコストも考慮し、設計段階からリグレッションテスト項目を意識した取り組みが必要である。



留意点

◆テスト自動化を徹底することで効率化出来るが、UIはバージョンアップで大きく変更されがちなため、UIテスト自動化のテスト資産は流用しにくい。UI操作を外付けで定義する自動化ツールを選択するなど、UI変更に追随可能な仕組みを検討すべきである。
◆リグレッションテストの効率化の観点から、長時間を要するテストに対しては自動化に加え、加速度テストの実施も検討すべきである。
◆テストシナリオの統廃合による効率化も検討すべきである。

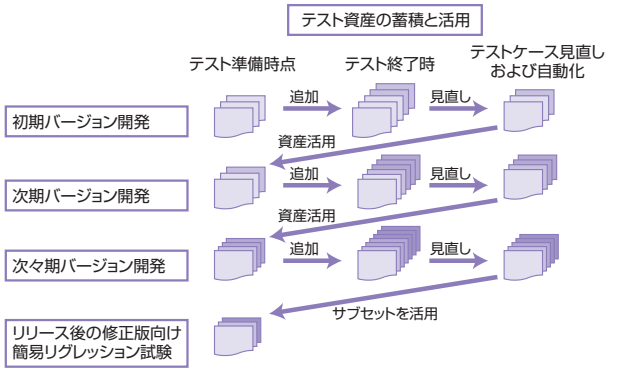
事例

テストの効率化に配慮しながらその十分性を確保するには、テスト資産の蓄積と見直しを実施すべきである。テスト資産の管理に関する取り組み例を以下に示す。

■同一プロジェクト内で何回かテストする場合や、シリーズ開発する場合にはテスト資産が蓄積されていく。重要なリリースポイントでのリグレッションテストでは、蓄積したテストケースをすべて実施するのが理想である。

しかし、単純に追加していてはテスト作業工数が爆発するため、この組織では開発終了したら資産化するテストケースの見直しを行い、また効率化のため手動で実施したテストケースは自動化している。

同時に、リリース後の修正版リリースでは、フルテストケースから必要な項目を抜き出したサブセットによる簡易リグレッションテストを実施するなど、工夫している。



■リグレッションテストは繰り返しテストなので、効率向上のために自動化推進が望ましいが、自動化のための工数と全体のコスト・効果とのバランスがとれない場合もある。従って、自動化の検討の際には、

・テスト実施工数の大幅短縮

・テスト操作性向上により、テスト実施ミス・実施漏れを防止可能

といったテストケースから自動化する。

テストするたびに修正が必要なテストケースは無理に自動化せず、効果・効率を判断しながらその都度実施する。

■テスト設計時に、自動化を前提とし、自動化しやすい環境を構築し、ツールを選択する。

1.2.2 重複、無駄、漏れを回避する総合テスト設計

関連：1.1.1
1.1.4

総合テストの各試験の種類と留意点を明確にして、テストケースの重複や無駄、漏れを回避する。

解説

総合テストには利用者観点に基づく機能確認の他にも運用保守上の確認など様々な種類があり、おのおののテストでは何を確認するのかを明確にしてテストケースの重複や無駄、漏れを防がなければならない。とくに総合テストはテスト対象がシステム全体であるため、単体・結合テストとは異なるポイントが存在することを理解する。

事例

■総合テストでどのような種類のテストを行うかは、システム特性やプロジェクト特性をもとに検討すべきである。ここでは、あるシステムでの総合テストで選択したテストの種類と、それぞれのテストで何を確認すべきか、注意すべきかを以下に例示した。

テストの種類	確認内容	留意事項
機能確認	・マニュアルや操作手引書に従って機能を確認する ・シナリオテスト（とくにGUI系）を行う ・ランダム操作テスト（携帯など）を行う	機能確認は単体テスト、結合テストである程度網羅されていることを前提とし、総合テストのマニュアルテストでは必ずしも網羅的な確認でなくてもよい ★重複をなくす
運用・保守	・H/Wの故障などを想定して、部品の交換やソフトウェアのアップデートなどを実施する ・各種H/W構成の組み合わせを確認する ・ソフトウェアライセンスの追加・削除なども確認する	H/Wエラーについて、通常の操作で再現が困難なテストケースであれば、総合テストでも人為的に工夫して再現させ、工数削減につとめる ★漏れをなくす

過去事例	・主にメインフレーム系で過去のユーザ障害事例データの再現データなど大量にテストデータを保持している。それらをすべて実施し、過去の問題が再現していないことを確認する	仕様変更によって再現し得ない障害もある。実施する意味の無いものはなぜ実施しないかを明確化する ★無駄をなくす
性能	・主にスループット系ではピーク性能が出る／負荷がかかる測定用のデータを用意しておき、それを流してきちんと性能が出ているかを測定する ・ランニングと平行して実施することで、滞留などにより性能を満たせないことが無いことを併せて確認する	RAID やネットワーク装置などは性能がとくに重要であり、障害修正や機能追加などで性能劣化が無いことが重要 ★漏れをなくす
ランニング	・長時間ランニング試験 過去事例のユーザ障害事例データを延々と繰り返すようなのも含む ・多数オペレータによる負荷テストを並行して実施する ・温度、湿度などの環境試験も実施する	なるべく、試験の最後のほうで実施する。障害が発生した場合、その影響範囲の分析は必須である。 ★無駄をなくす

1.2.3 戦略的テストスケジューリング

関連：1.4.1

テスト効率向上のため異常系テスト項目も含め、過去のトラブルケースやブロッキングバグなど、リスクの高い項目から優先的に実施する。

解説

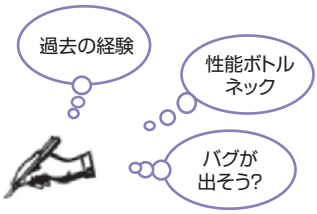
テスト優先順位の判断基準があいまいなまま、機能観点で大中小項目と機能分割してテストケースが書かれることが多い。別の観点でテストケースを書くことによって、少ないテストで大きな効果を出す方法がある。

組込み系システムでは、テスト全体に占める異常系テストの割合が高く、項目数も多い。また、異常系では発生条件をそろえるための準備に手間がかかるケースも多い。そのため、効率よく行うには、実施するべきテスト項目の優先付けが必要である。

少ないテスト件数で大きな効果を得る手法として、組み合わせテストでの組み合わせパターン数を間引く HAYST 法（直交表）、Pairwise（All-Pairs）法などがある。これらは製品特性に基づく主要な組み合わせを絞り込む方法で、その優先度付けを行う際の観点には以下のようなものがある。

①プラットフォーム及びこれに準じる共通的な機能のように、当該部分で生じるバグが全体のテスト進捗に影響を及ぼすと推測される部分を優先的にテストする。
②過去にバグが多発したモジュールから先にテストする。
③ブロッキングバグとなる可能性が高い（リスクが大きい）ものから先にテストする。
④基本機能は一通りテストする。
⑤性能のボトルネックになりやすい部位をテストする。

	a1	a2	a3	a4	a5	a6	a7	a8
b1	✔		✔					
b2	✔					✔		
b3		✔	✔					
b4				✔				
b5				✔		✔		
b6				✔				✔



また異常系のテスト項目抽出にあたっては、HAZOP[注]などの手法を活用するケースもある。

留意点

◆差分開発か、スクラッチ開発かによってもテスト実施優先度は異なる。差分開発では、主要機能は既に動作実績があるので、変更個所を優先して実施することが出来る。一方、スクラッチ開発ではすべてを新規に実装するので、キーになる機能、全体にかかわる部分などから着手することになる。
◆過去のプロジェクトのバグ分析からバグがありそうなポイントを推測することを推奨する、というのはよくあるが、実践するのは必ずしも容易ではない。現実的には、そういうポイントを勘と経験から見つけ出す達人に依存することが多い。論理的ではなくても、戦略的に優れた方法である。この場合、達人からの技術の継承が課題である。

事例

【事例1】以下に示すように、バグが発生する可能性が高いと思われるないしは、過去にバグが多く発生した機能部分については、テスト項目を重点的に追加する必要がある。

●リソース確保/開放のバグ発生の場合の実施例
- ・リソース確保/解放が確実に実施されることをテストする。
- ・リソース確保失敗のパターンをテストする。
- ・リソース確保成功と失敗が混在している場合、リソース確保成功の場合のみ開放されることをテストする。

●マルチタスクや割り込みバグ発生の場合の実施例
- ・データがプロセス/スレッドで共有されているかをチェックする。共有されている場合、そのデータに関してのアクセスパターンを分析しテストする。
- ・すべてのプロセス/スレッドについて、生成と消滅の組み合わせをテストする。
- ・出来るだけたくさんのプロセス/スレッドを立ち上げてテストを行う(プロセス/スレッドが突然終了する、もしくは終了させることが出来なくなる場合がある)。

【事例2】あるプロジェクトで実施された例。優先度を上げてテストすべき異常系テスト項目の対象となった過去のトラブルや、影響度の大きいバグが発生した際の具体的な対応内容を以下に示す。

●ブートプログラムの起動を行うことが出来ず、その上で動作するすべてのアプリケーションの確認が出来なくなる現象が発生した。

●アプリケーション動作に必要な情報を休止状態のようなステータスモードでいったんセーブしておき、起動時にそのデータを読み出すことでブート時間を短縮するといった起動高速化を実施している。この高速化処理では、目標時間内に完了出来るよう処理の短縮化を図る必要があり、このため、アプリケーション側のセーブデータ量をいかに少なく出来るかをトライ&エラーの試行錯誤で検討していたところ、アプリケーション側のデータ構造などにも影響が出てしまい、機能確認が出来ない状況になってしまった。

Column

(注) HAZOP とは

HAZard and OPerability study(潜在危険と運転性の解析手法)の略。

設計・運用の意図からの逸脱・ズレ(リスク)、それによって引き起こされる事故(影響)を解析する手法。

1.2.4 コードカバレッジの目標と終了条件

関連：1.3.4
1.3.7

カバレッジ基準は、カバレッジ以外の指標や手段との併用など複合的視点に基づいて設定し、本来実行されるべきコードが実行されているかという観点でチェックする。

解説

大規模化し、複雑化した組込みソフトウェアでC0 100%は現実的でない場合が多い。一方、小規模でも割り込みを多用するOSやデバイスドライバでは、C0 100%は十分条件に成り得ない。いずれの場合もC0 100%は誤った目標ではないが、終了条件とすることには疑問がある。

例えば、C0カバレッジ（命令網羅）は原則としてテストツール計測による100%網羅を目標とするが、終了条件としては、実施コストとのトレードオフも勘案の上、目視によるコードレビューも含めて100%としても構わないとするのが現実的である。

また制御系機能に対してC1カバレッジ（分岐網羅）や状態遷移網羅テストを実施する場合においても、100%実施を必須とするかどうかについて同様の考え方があったとしても間違いではない。形式的な網羅度の達成を目指すよりも、本来実行されるべきコードが正しく実行されているかという観点でのチェックを行うことがより重要である。結果として、網羅率100%となることが理想である。

事例

ある組込み製品開発におけるカバレッジ基本方針の具体例を以下に示す。

【A社】

・網羅率が100%に満たない場合は、カバレッジ未達部分に対してコードインスペクションを実施する。

・C0：100%実施を目標とし、更に複雑度が高い場合はC1：100%実施を目標とする。

【B社】

・C0カバレッジ網羅：新規／修正した実装単位にすべてのソースコードを対象に確認する。

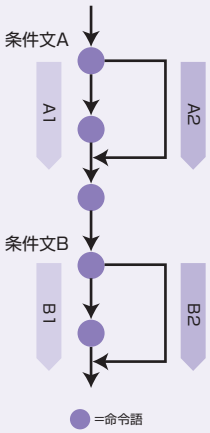
・C1 カバレッジ網羅:新規 / 修正した実装単位にすべての分岐を対象に確認する。
・状態遷移網羅テスト：状態遷移モデルを特定した上で、すべての状態遷移を確認する。
・API 網羅テスト：API 仕様を特定した上で、すべての API の全入力パラメータの正常値、両端値、異常値を確認する。
・C0 カバレッジ、API 網羅テストは基本的に必ず実施する。
・制御系機能に関するものは、C1 カバレッジ、状態遷移網羅テストを実施する。

Column

●一般的に品質指標として用いられる C0、C1、C2 の定義と上記との関連

一般的には、以下のように定義されているが、ツールベンダや各企業・組織部門によっては定義が異なる。

・C0：命令網羅率(ステートメントカバレッジ)
すべての命令を 少なくとも 1 回は実行。
・C1：分岐網羅率（ブランチカバレッジ）
すべてのブランチを少なくとも 1 回は実行。右図では例えば A1-B1 と A2-B2 のパスを実行することで達成される。
・C2：条件網羅率(コンディションカバレッジ)
すべての条件を少なくとも 1 回は実行。右図では A1-B1、A1-B2、A2-B1、A2-B2 のパスを実行することで達成される。



1.3 基準値・目標値・終了条件

1.3.1 ソースコード行数の数え方

関連：1.3.2
1.3.4

テストに関するソフトウェア品質基準値の算出に行数を用いる際は、経験に基づき、実態を反映した行数見積もりとする

解説

テスト工数やテストで使用する品質基準値を算出する際、ソースコード行数をベースとすることは一般的に実施されている。また、修正モジュール全体への影響度合いや、再利用モジュールのテスト工数見積もりにおいては、対象製品やプロジェクトの特性によって係数を変えるなどの工夫も行われている。ただし、その都度適切な係数を決定するのも容易ではないという現実もあり、この重み付け係数を固定値として設定するケースも多い。

社内で係数を統一することにより、過去開発データとの比較が可能となることのメリットは大きい。

留意点

差分開発の場合、仮に新規開発の場合と開発対象行数が同じであったとしても、バグ密度の目標値を変えたり、行数の代わりに機能数やI/F数、モジュール数を分母にしたり、実態を反映するための補正を要する。また、機能数を用いる場合の補正では、経験の有無によって結果が大きく左右されやすい点に注意が必要である。

事例

差分開発、流用開発における例を以下に示す。

【A 社の場合】

■既存資産に基づく差分開発、流用開発では、単純に追加・変更行数を足し合わせただけの行数ベースの見積方法では実態に合わないことが多い。例えば、行数が少なくても影響範囲が大きい場合などがこれに該当する。そこでソースコード行数算出式として以下のように重み付けした換算を行っている。

Σ（新規モジュール行数）＋ 0.5 Σ（修正モジュール行数）＋ 0.2 Σ（再利用モジュール行数）

■第 2 項は、修正行数だけではなく、修正が当該モジュール全体に影響を及ぼし得ることを意味する。

■第 3 項は修正なしで再利用するモジュールであっても、結合テストや総合テストの対象となるので、そのテスト工数分を反映しておく必要があるということを意味する。

【B 社の場合】

■ソフトウェア規模を以下の式に基づいて換算している。

開発規模 = 新規規模 + 改造規模 + 流用規模

新規規模：新規規模 = 新規作成ファイルステップ数を合計

改造規模：各ファイルの改造規模を以下で算出し合計

追加・削除ステップ数が改造ファイルステップ数の半分以上なら改造規模 = 改造ファイルステップ数とする

そうでない場合は、追加・削除ステップ数× 2 を改造規模とする

流用規模：各流用ファイル（手を付けないファイル）の規模× 0.1 を合計

■改造規模は単純に追加・削除したステップ数以上に他への影響などを考慮する必要があるため、規模が増える方向で換算する。元のプログラムの半分以上に変更が行われるなら、新規作成と変わらないという経験則から、半分以上か否かを基準にしている。

■流用部についてもリグレッションテストなどでかかるテスト工数分を加算する。

Column

下記は、いずれの計測方法が正しいというものではなく、統一されていることが重要である。

・差分開発、流用開発、派生開発をする際、コンパイルオプションのSwitch でコンパイル対象から除外する部分の行数を含めるか否か。
除外する場合、行数の計測にコンパイラのプリプロセッサなどのツールが必要になる。

・マクロ部分を、マクロ展開する前の行数とするか、マクロへの入力を反映させて展開したあとの行数とするか否か。

ソースコード行算出のモデル式としては、COCOMOⅡソフトウェア再利用モデル式というものもある。本式の係数導出には開発者の手間がかかるので、適用するならば開発者の負担増加(コスト増加)に見合う効果が期待できるか検討し、開発者に理解できるよう説明すべきである。

1.3.2 関係者との合意に基づく目標値設定

関連：1.2.4
1.3.1
3.1.4

テスト基準値は過去の開発実績や当該製品要件、プロジェクト特性などに基づいて設定し、設定値とその設定理由はテスト実施に先立って関係者と合意する。

解説

テスト密度やバグ密度などの基準値はプロジェクト開発計画作成時にプロジェクトリーダが判断して設定することになるのだが、実際にテストが完了しその結果を関係者でレビューする時点でその目標値の是非が取りざたされることは多い。従って、テストに先立ちその妥当性・十分性を品質保証部門他の関係者と合意しておくことが、円滑なプロジェクト運営において重要である。

目標値をどのように設定したらよいかについては、ESQRに記載されている指標や算出の考え方が参考になる。しかし、ESQRに記された参考値を常にそのまま適用するのではなく、実績に基づいて目標値算出の計算式に手を加えるなど自組織に合った改善を行うべきである。例えば、あるシステムでESQRに示された指標を参考にして設定したとき、テスト密度は60項目となったが、ソースコードにOSS（オープン・ソース・ソフトウェア）の占める割合が高いため、結合テストのテスト密度を20項目、総合テストのテスト密度を40項目とするなど。

留意点

ソフトウェア規模などのプロジェクト特性が目標値に大きな影響を与える結果、部門基準値と乖離することはある。この場合、その差に対する妥当性判断はレビュアに依存することになるので、こうしたケースをも含め、組織としての判断基準がルール化されその処置が事前合意されていることが望ましい。

事例

ある組込み機器開発においてESQRを参考に設定された指標について、以下に例を示す。

この部門ではプロジェクトの方針・戦略、部門基準、プロジェクト特性、製品特性などから単体・結合・総合テストそれぞれのテストの目的・方針・観点を調整要因に鑑み、関係者による事前レビューの上で、目標値を設定している。

■目標値

・テストレビュー作業充当率＝テストレビュー工数/テスト準備・確認工数
・テスト作業充当率＝テスト工数/開発全工数
・テスト作業実施率＝テスト工数/KLOC
・テスト仕様書ボリューム数＝テスト仕様書頁数/KLOC
・テスト密度＝テスト項目数/KLOC
・不具合発見率＝検出不具合数/KLOC
・不具合収束率＝各テスト工程までの検出不具合数/目標検出不具合数

■調整要因

・メンバの当該システムの経験年数やスキル
・開発途中での仕様変更の多寡、設計者とプログラマが同一か異なるか
・OSS(オープン・ソース・ソフトウェア)使用の有無、程度
・新規の技術、ハードウェア、ソフトウェアの採用有無、程度
・既存資産の流用程度、改造程度
・オフショア開発時の委託範囲や委託先のスキルレベル、実績

1.3.3 コストとのバランスを考慮したテスト項目抽出

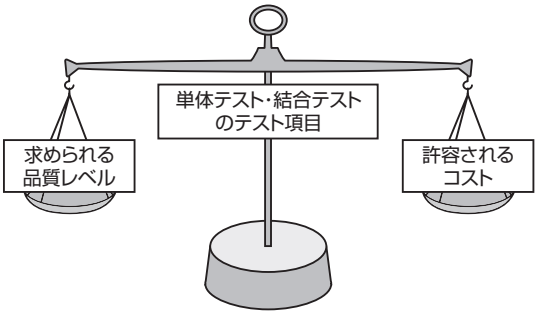
関連：1.1.1 1.1.2 1.2.3 1.4.1

単体・結合テストのテスト項目は、求められる品質レベルと許容されるコストとのバランスを考慮して抽出する。

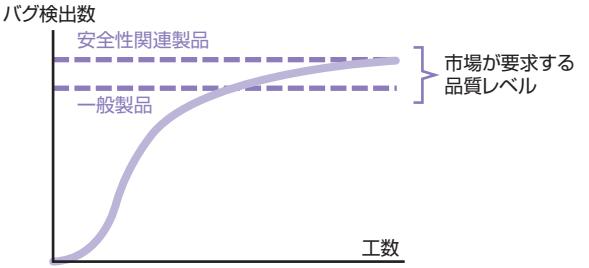
解説

単体テスト及び結合テストでは、テスト対象とする機能、状態遷移モデル、API 仕様を特定し、そのすべてを網羅的に確認することが基本である。例えば、状態遷移モデルであれば取り得るすべての状態遷移、API 仕様ならすべての API に対する全入力パラメータの正常値、両端値、異常値に対する確認を実施することが望ましい。

■しかし、実際には日程やコスト・要員の制限などの理由により、すべての状態遷移や API 仕様を網羅的にテスト出来ない場合がある。従って、求められる品質レベルに応じたテストをするという考え方が必要になる。



■例えば、あらかじめ製品に要求される品質レベル、市場で問題が生じた場合の重要度や被害の大きさなどをプロファイリングし、その結果からテスト項目ごとにシビリティ（重大性）やプライオリティ（緊急性）を定めておき、それに従って抜き取りで実施する。あるいは、バグ検出数の基準値を、一般製品/安全性関連製品などに応じた体系にするなどの対応が考えられる。



■重要度の決定方法としては例えば、ESQR のシステムプロファイリングやプロジェクトプロファイリングの結果から、各テスト項目にシビリティやプライオリティを設定することなどが考えられる。

事例

ある通信機器開発メーカでは、従来から、実装している通信プロトコルごとに利用者の使い方なども想定したテストを入念に実施している。しかし、このオープン化の時代にあっては、通信プロトコルと想定される使い方の組合せ数が爆発的に増加し、かつ製品価格低下により開発コスト削減を要求されている。その結果、すべてをテストすることが難しくなってきた。品質とコストのバランスをいかに取るかに知恵を絞っていかざるを得ない状況にあり、市場の要求レベルに見合ったテストとするべく現実的な対応を進めている。

Column

グローバル化のこの時代、新興国の追い上げにあい日本企業はこれまでの高機能・高品質戦略一辺倒では事業的に厳しくなっている。高価格でも付加価値があれば売れるかも知れないが、どんな製品でも単に品質が良いからという理由で売れる時代ではない。

海外から「日本はテストをやりすぎる」という指摘を受けることがあり、国内でも過剰品質が指摘されるようになってきた。

かといって、一見過剰なまでの高品質が重要な付加価値/競争力となる製品もあるであろうし、「安かろう悪かろう」もいただけない。市場が要求する品質に応じたテストという考え方が大切であろう。

1.3.4 総合テスト開始判断基準

関連：1.2.4
1.3.5
1.3.6

**総合テスト開始の判断基準を定めて、
柔軟に運用する。**

解説

総合テストはそれ以前の単体テストや結合テストと異なり、利用者の視点に基づく要求仕様を満たしているかという観点から実施するものであり、その実施が可能な状態であるかを客観的なエビデンスに基づいて判断することが求められる。

総合テスト開始可否の判断基準を組織として定めるべきである。

この判断基準項目には例えば以下のようなものがあるが、製品規模や開発の実状も加味した現実的な運用とすべきである。

①静的解析によるソースコードのチェック

・静的解析チェックが実施されたあとに単体テストが実施されたか。
・単体テストは移行判断基準を満たして実施されたか。
（例：サイクロマティック複雑度が30以上の場合はコードレビューを実施する）

②単体テスト

・原則としてすべてのモジュールの単体テストが完了済みであること。
・ただし、製品特性(ソフトウェア規模、ソフトウェア構成、開発人員構成、要求される信頼レベルなど)により単体テストの実施の要否・程度はリーダが判断する。
・単体テストの実施の要否に関しては、計画段階で実施判断基準書を作成し、不要の場合には基準に従った理由が明確にされていること。
・基準値と実績値の乖離の理由が明記されていること。合理性はリーダが判断する。

留意点

総合テスト開始条件として「単体テストが完了していること」というケースもあるが、そもそも単体テストが実施されない場合もあるし、小規模開発(1、2人で1、2カ月で実施)では自己レビューでレビュー実施とみなすこともある。これらを良しとするか否かは判断基準を作成し、それに照らし合わせて決定すべきである。

事例

総合テスト開始判断指標の一部の具体例として、静的解析ツールのメトリクスを用いた例について以下に示す。

それぞれの指標に対する許容範囲がプロジェクト開始時点に設定され、総合テスト移行時の判断基準となる。なお、この許容範囲値は、プロジェクト終了時に見直すなどのメンテナンスも行っている。

項目	許容範囲	備考
経路複雑度	10 以下	
本質的な経路複雑度	1 以下	構造化の尺度(OneEntry/OneExit)
複雑さ	10 以下	論理演算子数による複雑さ(マイヤーズ・インターバルの算出元)
ネストの深さ	15 以下	15 を超えるときは、意味の単位で関数を分割するなどの対応を行う。
保守コード行数	200 以下	コメントを含めた関数本体の行数。行数超過時は、意味単位での関数分割、インライン関数などによる処理の分割を実施する。

1.3.5 段階付き合否判断基準

関連：1.3.4
1.3.6

テストフェーズごとの合否判定は単なる合否だけでなく、複数段階の基準で判断してもよい。

解説

テスト項目設定率、エラー検出率などの品質指標は、テストフェーズごとにその合否判定の基準値を設定し運用するべきだが、この際その基準値に数段階のレベルを設けて、指標測定結果がどのレベルに収まるかにより、「問題なく合格」「条件付き合格」「テスト項目の見直し」などといった処置をレベルごとに定義してもよい。

基準値に絶対的なものはなく、実績を積んで次第に精度が高められていくものである。まずは自社内に実績データがあればそれを参考に、無ければESQRの算出方法を参考にするとよい。

留意点

◆設定数が少なくて検出数が多いが許容ゾーン内にある場合と、設定数も検出数も多いが許容ゾーンを超えている場合どちらが本当は良いのかなどの判断に際しては、内容をよく吟味せず機械的に行ってしまいがちである。

◆テスト密度は高いが検出数が少ないのは、品質が良いのか、テスト設計での検出能力が低かったのか、など再度レビューしてみるべきである。テストに偏りがあり、最も問題が潜在していそうな個所が漏れていた場合、そこを再テストするなどの対応が必要になる。

事例

ある組込み製品開発で用いられているバグ密度などの判定基準とその運用事例を以下に示す。判断基準に幅が設けられ、その中で段階的な合否判断を行っている。

■単体テスト、結合テスト、総合テストとフェーズごとに下々限、下限、上限、上々限の指標を設定し、以下のようなチェック基準で判定し基準値と処置をルール化している。

① XX ＜下々限：テスト項目の設定が粗すぎる⇒テスト項目見直し

②下々限≦ XX ≦下限：条件付き合格⇒第三者の受け入れ試験結果を反映する

③下限≦ XX ≦上限：合格
④上限≦ XX ≦上々限：条件付き合格⇒設定率が高い理由を明示する
⑤上々限< XX：実施可能か問題がある⇒テスト項目見直し

■数値がある程度把握できた時点で基準値と比較評価を行い、その差異などの原因を考えさせ意識させるような取り組みを行っている。
■ステップ数を入力すると、各指標との比較を自動的に示してくれるツール(チェックシート)なども運用している。
■ほとんどの場合、プロジェクト終了時に活動の振返りとしてデータ収集と指標値による分析を実施している。
■プロジェクトを進める上での各フェーズでの指標値の収集と判定基準によるチェックは義務付けているが、弾力的に運用している。

1.3.6 テストフェーズ移行判断基準の統一

関連：1.3.4
1.3.5

テスト項目やテスト工数、テストフェーズ移行などの判断基準は組織として統一しておく。

解説

所与の開発期間内で適切な製品品質を確保していく上では、過去の失敗例なども反映し、テスト基準を組織標準として統一しておくべきである。こうした標準を構成する要素には、例えば以下のようなものがある。

①テスト工数　：標準テスト工数は過去の実績と経験値に基づく
②テスト項目　：過去の不具合傾向分析からの経験に基づく
③検証の種類　：担当所掌部門とその内容ごとに定義
④評価判断基準：移行のための判断プロセスと基準の定義

留意点

◆ヒューマンマシンインターフェースなどでは、“項目数”のように決めやすい指標もあると同時に、“テスト準備に要する手間”などのように定量化しにくいものもあるため、目標値としてどのように扱うかを組織で定めておくことがテスト結果を判断する際に役立つ。

◆開発者がテスト実施すると開発工数や負荷状況がテスト内容に影響するといった、恣意性混入の可能性がある。そうはいっても開発者と同等レベルの第三者テスト担当を確保するのは難しいという現実課題があり、結局、開発者がテスト設計して、テスト実施するケースも多い。そこで、客観的な判断基準も必要になる。

事例

ある組込み製品開発企業での組織標準を以下に抜粋した。

■テスト工数

・全機能検証は2カ月、修正した場合の追加検証は1カ月、品質保証部門による検証はシステムの基本部分に関するテストを厳しく見積もり0.5カ月とする。
・また標準シナリオパターンのテストを実施する場合は、1パターン当たりの標準工数として1人当たり月1500項目と定め、実際の製品の複雑さなどで調整する。

■テスト項目

・過去の不具合傾向分析からの分析により、評価項目と実施順番を決定する。
・どこからテストすれば重度なバグを早く検出できるかとの推定・期待により、決定する。

■検証種類の定義

・機能検証(開発担当者)、最終評価(品証部門)など
・リリーステスト：最終評価までは変更実施個所に対して行い、それ以降はリリースごとに確認する。

■評価判断基準

①最終評価開始条件
- 判断は「開発担当者」「プロダクト責任者」「評価責任者」の3名。
- 新規の場合、全機能実装済みである。
- テスト計画書、テスト仕様書の作成が完了している。
- テスト仕様書がレビューされている。
- 変更モジュール、変更量が把握されている。
- 評価移行審議の記録として、開始チェックシートやテスト仕様レビュー記録がある。

②最終評価終了条件
- 品質保証部門が判断する。
- 不具合発生件数が想定値を超えていない。
- 不具合管理システムで重大性が高いものの残件数が無い。
- 仕様変更への対応が完了している。
- 重度の不具合は必ずすべて修正する。
- 再現しない不具合は、コードチェックなどを行い対策を実施する。
- テスト報告書の作成が完了している。

1.3.7 達成不可能なコードカバレッジ基準とその終了可否判断

関連：1.2.4

**コードカバレッジ基準に満たない場合は
その理由を明確にして合理的に判断する。**

解説

単体テストの結果、設定されているコードカバレッジ基準値に満たなかった場合、その結果をどのように解釈するのかが問題になることがある。また、基準値達成に必要なリソース投入が難しいというような現実的な状況に直面することもあり得る。

また、差分開発・流用開発においては、目標値設定時点で実行不可能なパスが存在することまで把握できないこともある。

このような場合、網羅しきれなかった部分とその理由などを明確にし、必要に応じて別の指標と組み合わせて評価するといった、合理的な判断基準でテスト終了とするのが一般的である。

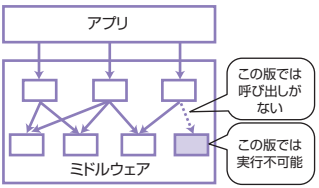
留意点

絶対に通過しないはずのパスでも、実際にテストしたら実行してしまうこともある。例えば、下記事例のように上位関数でパラメータチェック済みであっても、上位関数でのチェック方法に誤りがあったり、上位関数の改造時に無駄なチェックを削除したときなど。

事例

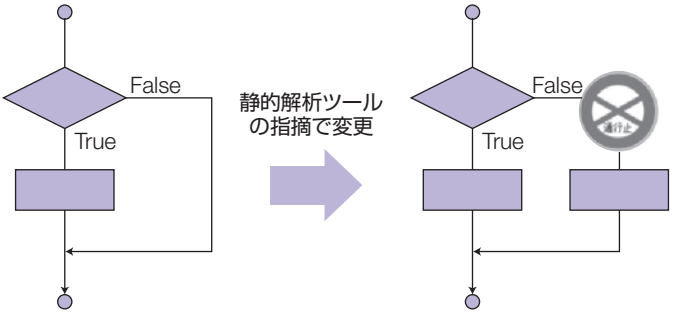
ある組込み製品を開発している組織で、網羅率100%を目標としたが、100%を達成できずに、無駄な時間・工数を費やした事例を以下に示す。

■汎用性を考慮したミドルウェアの中の内部関数に、そのバージョンのシステムでは使われない関数が含まれていた。構造解析ツールで調べても、使用される関数からの呼び出し関係があり、実際には当該内部関数が使用されないこと



が自明ではなかったため、通過し得ないことが判明するのに時間がかかった。

■入力パラメータのチェックを必須とし、更に if then else の対応付けを行うようコーディング規約に定められていたため機械的に else 文を挿入したが、上位関数でチェック済みであったため、実行不可能なパスとなり、カバレッジ 100% になり得ないケースが生じた。



■コードを作成した本人は絶対通らないパスであることを認識していたと思われるが、コーディング規約で定められていることに対してその都度コメントを挿入していなかったため、差分開発でのテスト担当者には分からず、テストケース作成に無駄な時間を費やした。

1.4 テスト管理

1.4.1 テスト効率向上のための戦略的テストシナリオ作成

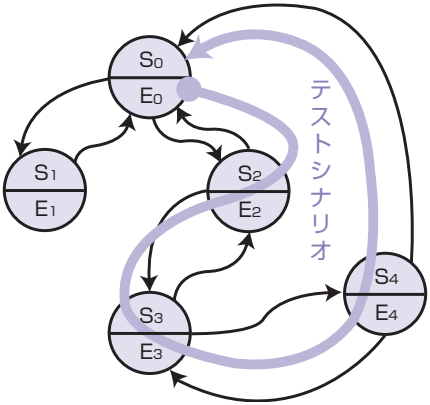
関連：1.2.1 1.2.3

組込みシステムでは異常系のテストケースが多く再現に手間がかかるケースも多いため、テスト項目同志の繋がりを考慮したテストシナリオを考えるなど効率よくテストするための工夫をする。

解説

組込みシステムの特徴として、ハードウェア異常検出やリトライ、リカバリなどの異常系処理が処理全体の大半を占めている。テストのためには、それら多くの異常状態を再現させる必要があるが、必ずしも再現は容易ではない。そこで、再現手順を容易にするための工夫と同時に再現手順の回数を削減させる工夫も、テスト効率化に有用である。

テスト実施に必要な状態とするための前処理に手間がかかったり、その状態にすること自体が困難なテストケースでも、テストシナリオによっては一連の作業の中で容易に作り出せることがある。テスト環境やテスト対象の状態がテストシナリオに合わせて変化していくので、シナリオに沿ってテストを行うことにより自ずと目的の状態になるというケースである。



また、あるテストの実行結果が次のテストのインプットになるなど、単独ではなく繋がりで考えることにより、少ないテストで効果が出せるケースもある。

テスト1によりinput2が自動的に生成されるのでinput2作成の手間が省ける

留意点

特に組込みシステムの異常系に関しては、状態作成が困難/手間がかかる場合が多い。そのため、必ずしも正常系と異常系とを分けてテスト実施するのではなく、正常系と異常系を組み合わせたシナリオとし、効率化することも検討すべきである。

事例

以下に実際の開発組織で行われている取組み例を示した。これらの多くは、テスト可能状態にするために長時間を要したり手順が複雑であるような限界値テスト・異常テストケースである。とくに総合テスト段階で、その都度そうした手間をかけていることが現実的に困難であるという事情がある。

■メモリ容量に制限があるハンディ端末では、上限値に近い数 MB のデータを蓄積させ、その状態で更にデータを入れた場合の異常テストを行うが、この状況を生成することが容易ではないため、一連のテストシナリオの中に組み込んで、上限に近い状態で正常系も含めたテストを行う手順とする。

■フラッシュメモリのバンク切替えが必要になるような境界条件を作り出す限界テストを総合テストで行う際には、その状況を生み出す処理を他機能テストの連鎖で作り出せるようなテストシナリオを作成する。

1.4.2 テスト管理における問題の見える化

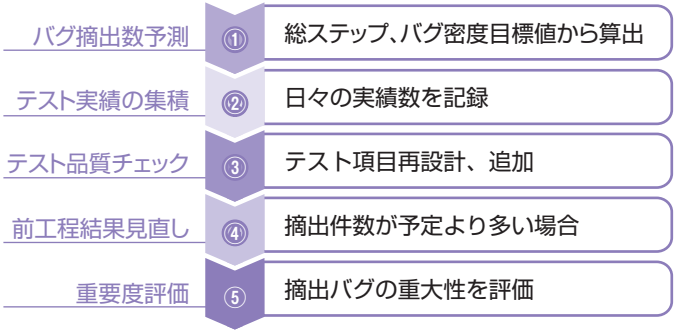
関連:2.1.2

テストの進捗状況や品質指標の分析・予測にグラフ、図表を活用して問題の見える化を図る

解説

組み合わせテスト以降のテスト管理では、テスト消化曲線(残存テストケース数)とバグ発生曲線(累積バグ数)を活用する手法がある。グラフ表示によりテストの進捗状況を一目で理解することが出来るとともに、品質予測やプロジェクトの課題分析など問題の見える化を図り、先手管理に用いることが出来る。以下にその活用手順を示す。

①今回の総ステップ数とバグ密度の目標値(過去の類似案件の平均値などを算出したもの)から、バグ摘出総数を予測する。
②日々のテスト実施数(テスト項目の総数とテスト工数)を記録する。
③予実の乖離、とくにテスト項目の消化速度に比べ、バグ摘出件数がかなり少ない場合などは、テスト品質(テスト項目の質)をチェックし、テスト項目の再設計や追加を行う。
④逆にバグ摘出件数が予定より多い場合は、前工程での品質確保が十分行われていない可能性があり、その対応を検討する。
⑤また、バグ件数だけでなく重大性も評価し必要に応じて対応を検討する。



留意点

◆小規模・単納期プロジェクトでは、予実の乖離がより大きくなる傾向もあり、また、小規模といえども一定の管理コストはかかるため、プロジェクトにある程度の規模が無いと、かけた手間に見合う効果を得にくい。
◆類似バグを多数検出すればそれで品質が向上したと思いがちであり、またその検出数カウントも恣意的となる可能性もある。類似バグは同じものとして 1 件としてカウントするか、また机上検討で発見したものは別々にカウントするかなどの尺度は統一しておくことが望ましい。
◆予定との乖離が大きい場合だけでなく、計画通りに推移している場合でも、検出されているバグの内容と併せて評価しないと実状を見誤りかねないため、個別の懸案リストなどで個々の不適合内容とその解決状況を定期的にモニタリングしている。

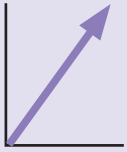
事例

ある組込み製品を開発している組織で使用されている品質状況モニタリング及び分析の例を以下に示す。

■下表は不具合密度とテスト項目密度を用いて品質状況を把握する際の分析例で、高、低の判断は品質管理計画で立案された上限下限値をもとに行う。

		テスト項目密度	
		低	高
不具合密度	高	・品質に問題がある可能性高。 ・テスト項目以外の部分で不具合が多発している可能性有。 →テスト項目数、またはテスト項目の品質を見直す。	・ソフトウェア品質に問題はあるが、テストで不具合を出し一定の品質が保てる見込み有。 ・後続のテストでも不具合密度が高い場合は設計の見直し必要。 ・特定部分だけの不具合を検出している可能性も有。 →テストの偏り有無を分析。
	低	・テスト項目不足の可能性高。 ・不具合が残存している可能性高。 →再度、追加テストの実施を含め、現状を確認する。	・テストは十分に実施されているが、不具合が検出されていない。 ・成果物の品質が良かったか、あるいはテスト項目の品質が悪いことが想定される。

■不具合検出進捗とテスト実施、テスト成功進捗を用いて、品質状況を把握する際の分析例を下記に示した。ここで「テスト実施」とは実行したテスト項目の数を、「テスト成功」は不具合なく成功したテスト項目の数を意味している。

不具合検出進捗	テスト実施、成功進捗	分析結果
不具合増加傾斜が高い 	テスト実施は停滞 	停滞時点でテストしている機能に関するモジュールの品質が悪い。
	テスト実施は順調 テスト成功率は低い 	全体的に品質が悪い。 テスト実施レベルでないものでテストを開始している。
	テスト実施は順調 テスト成功率は高い 	不具合修正のときに、新たなバグが埋め込まれている可能性がある。 計画したテスト以外から不具合が発見されている可能性がある。
不具合増加傾斜が低い 	テスト実施は順調 テスト成功率は高い 	進捗からは品質が高いことが分かる。また、テストが不十分なため不具合が検出できていないことも考えられるため、テスト設計を確認する必要がある。
	テスト実施は低調 	テスト実施者の人数、テスト機器、またはテストスキル不足のために進捗が悪い。

：テスト実施件数　：テスト成功件数

Part 2

テストへの要求と対応状況

組込みソフトウェアは規模の拡大、複雑化が進み続けており、それに伴い更なるテスト強化・充実が要求され、ときにはテスト実施が非現実的なほどにテストボリュームが肥大化しています。

他方、製品サイクルの短期化、製品価格低下により、テストにかけられるコストの削減が要求されています。

この相反する要求に対して、すべてのテストを手作業で行うことが不可能な状況に陥ることはもはや必定です。そこで、作業を効率化する手段としてツール活用や自動化が必須になっていますが、いつ・どこで・誰が・どのツールを・どのように活用するかは非常に多くの選択肢があり、適切にツール活用しなければ期待通りの効果を発揮できないどころか、無駄（コスト増加）やテスト漏れ（品質低下）の原因になることも少なくありません。

本章では、こうした品質と効率の難しい要求に直面しているテスト設計者及びマネージャ、品質評価部門関係者を念頭に、どのような状況においてどのように適切にツール活用しているか、その勘所を中心にツール活用事例を紹介します。

また企業における事例の他、Android 開発環境など最新の開発プラットフォームの話題、公共機関におけるツール活用状況も含め、組込みソフトウェアのテスト環境を取り囲む状況についての事例をまとめました。

2.1 ツール活用

2.1.1 テスト実行支援ツールの使い分け

関連：1.1.3/2.1.2/2.1.3/2.1.4/2.1.5/2.1.6

テスト実行支援ツールは、ツールの用途・特徴を理解して適切に使い分ける。

解説

テスト実行を支援するツールには自製/市販/OSS（オープン・ソース・ソフトウェア）があり、簡易なツールから大がかりなツールまで、多様である。用途別では、以下のようなものがある。

①コードレベルの自動化支援ツール（市販/OSS、静的解析/構造分析ツール）
②操作系の自動化支援ツール（GUIの入力自動化ツール/ボタン操作ロボット）
③シェルスクリプト（多くの局面を支援する非常に強力なツール）
④print文と目視による結果確認またはtext差分比較などの結果確認ツール
⑤デバッガ（自動実行や再現が困難なケースのテスト用途で使用する）
⑥シミュレータ（簡易なスタブや高度なモデリング技術によりインターフェースを模擬するツール）
⑦エミュレータ（システム構成要素の一部の動作を代替手段により再現するツール）

いずれのツールにもツール開発者が想定した用途があり、万能ツールは無い。またツールの出来映え、導入コスト、メンテナンスコストも様々なので、これらのツールを有効に利用するには、用途や特徴を理解した上で使い分ける必要がある。

留意点

◆ print 文は多くの局面で動作を確認する上で非常に有効であるが、print 文を実行する際に動作のタイミングやメモリ、CPU のリソース使用状況が実運用時と異なるため、その影響で潜在バグが残る可能性もあるので、print 文を実行させない最終動作確認は必須である。
◆デバッガを用いた場合、リソース利用状況に影響を与えないようにすることは可能だが、print 文挿入の場合と同様、実運用時とはタイミングが異なる。

留意点

◆ツールに過度な期待をしないこと。例えば、高度なシミュレーション環境を構築すれば実機環境が不要になるという考えは誤りである。

・実機環境でなければ出来ないテストケース

・シミュレーション環境だから容易に実施可能なテストケース

があり、２つの環境は相互に補完するものなので、シミュレーション環境を構築しても実機環境が完全に不要になることはない。

事例

ある企業でのツール活用状況の例を以下に示す。

■静的解析ツール

・ツールの得手不得手、プロジェクト特性によって複数ツールの一方または両方を使用。

・コードレビュー前にA社のツールでコーディング規約への適合性を確認する。

・テスト前にB社のツールで引数レベルでのインターフェースを検証する。

■単体テスト

・C言語用、Java用それぞれの推奨ツールがある。

■WebGUI系

・リグレッションにブラウザベースのOSSマクロレコーダー（Webブラウザのアドオン）を使用。

・画面変更が多いソフトには、テストデータや期待値を外付けに出来るツールを推奨。

■Windows GUI系

・テストに必要な前提条件作り込みを含めて、自動化可能なテストスクリプト言語を受け入れテストに使用。

■負荷テスト

・スタブや入力データ作成、モデリングなどに多くのコストをかけられないプロジェクトは、H/W依存部負荷テストにデバッガを使用。

・コストが見合うプロジェクトでは、OSSのシミュレータ（SILS）やPC+FPGA（HILS）で擬似装置を作成する。

■異常系

・デバッガで値を書き換えて異常状態を模擬的に発生させて、目視確認する。

Column

例えば、不具合原因特定に至らないバグが残った場合のように、場合によっては、不具合原因推定のため、テスト実施後に静的解析ツールや構造解析ツールなどのツールにかけるという使い方もある。

2.1.2 テスト管理ツールの統一

関連：1.4.2/2.1.1/2.1.3/2.1.4/2.1.5/2.1.6

組織で、テストスケジュール、テストケース、発生した障害などを管理するツールの統一を図る場合は、部門ごとに要求の違いがあることを考慮する。

解説

テスト管理ツールの統一によって、以下の効果を期待できる。

・複数部門/プロジェクトの品質比較による、問題の早期発見、横断的な対策実施
・部門ごとのツールのメンテナンス負担軽減

しかし、部門ごとの要求や事情を考慮せずにトップダウンでテスト管理ツールの統一を強引に押し付けた場合、

・開発部門においては使いづらく開発効率改善にならない
・ツール管理部門においても部門ごとの要求への対応が間に合わない

といった状況になり、ツール統一に失敗することが少なくない。

テスト管理ツールの統一を図る場合は、まず以下の点を明確にする。

◆利用する範囲

→部門、プロジェクト、対象商品、開発規模、社内/社外、国内/海外など

◆統一するツールの種類

→スケジュール管理、テストケース管理、障害管理など

◆ツールの導入や保守

→体制、時期、導入支援、セキュリティ対策、ディザスターリカバリ対策など

そして、

・統一ツールを導入する対象部門の現状把握
・対象部門が現在使用しているツールと統一ツールの比較

といった事前調査を行って、部門ごとの要求の違いを把握すべきである。

もし事前調査が十分出来ないとしても、部門ごとに要求の違いがあることを想定して、柔軟な対応が可能な運用ルールとすべきである。

留意点

◆表計算ソフトのマクロ機能などを駆使してテスト管理の改善に取り組んでいる部門が多いので、既存ツールと連携しやすい仕組みや運用ルールを用意すべきである。
◆表計算ソフトは、データ量の制約から大規模開発では使えなくなることがある。大規模化が予測されるならば、専用管理ツールの導入を検討すべきである。
◆国内協力会社や海外との分散開発においてテスト管理ツールを統一するには、セキュリティ対策が重要である。とくに、障害管理のデータは取り扱いに注意が必要である。

事例

某社のテスト管理ツール使用状況。

スケジュール管理	表計算ソフト、社内システム、市販スケジュール管理ソフト
テストケース管理	大多数が表計算ソフト。 市販／ OSS ツールも検討中だが、運用しているのはごく少数
障害・バグ管理	多くの部門が社内システム。その他は表計算ソフト

■障害管理の社内システムは使いづらいという意見もあるが、海外を含む分散開発が多くなり、障害情報を共有しなければ開発を進められないという背景から統一されている。
■スケジュール管理、テストケース管理も社内システムでの統一に取り組んでいるが、使い勝手の問題への対応、部門ごとの要求の違いへの対応が間に合わず、進んでいない。
■テストケース管理ソフトの表計算ソフトには多くのデータ蓄積があり、ツール変更は容易ではない。差分開発が多く、差分開発では過去から使ってきたテストケースに追加する形で運用しているからである。

表計算ソフトは導入しやすく、自由度も高いため、表計算ソフトを使っている部門が多い。しかし、部門ごとの文化の違いから、管理方法はまちまちであり、表計算ソフトのシートを横展開できないという問題がある。また、開発規模が大規模化しており、表計算ソフトの限界になりつつある。まだ限界でなくても、ソフト起動・操作がとても重くなっており、専用管理ツール導入の必要性は感じている。

Column

既存ツールからのデータ移行の仕組み/既存ツールとのデータ連携の仕組みを用意することで、ツール統一を促進できる。しかし、多くのツール間でのデータ移行/連携の仕組みを用意するのも容易ではない。その場合、表計算ソフトとのデータ移行(インポート/エクスポート)の仕組みを用意するだけでもかなり役に立つ。

2.1.3 ツールのポータル化

関連：2.1.1/2.1.2/2.1.4/2.1.5/2.1.6

ツールを活用するには、導入後のメンテナンスコスト負担の問題を解消する必要がある。とくに大きな組織でツールを統一する場合には、ツールをポータル化してサービス提供することも選択肢に入れる。

解説

ツールを適用する対象は、特定開発チーム、同一ドメイン、全社、グループ会社全体など様々である。

ツールを適用するには、

①ツール利用環境構築

②利用者への運用教育

③ライセンス費用

④保守サービス費用

などのイニシャル/ランニング・コスト、ツール運用・管理担当者、利用者向け教育が必要になるが、全社やグループ会社全体など広い範囲で一斉にツールを適用する際には、全利用者にそれらの負担を要求できない場合がある。そのような場合には、推奨ツールをポータル化して、利用者側のツールメンテナンス負担を軽減する方法がある。

ポータル化によって、導入・運用常態化の促進も容易になる。

留意点

◆ツール利用のタイミングが遅れることがないよう、開発担当者がいつでもツールを使えるようにサービス提供すべきである。

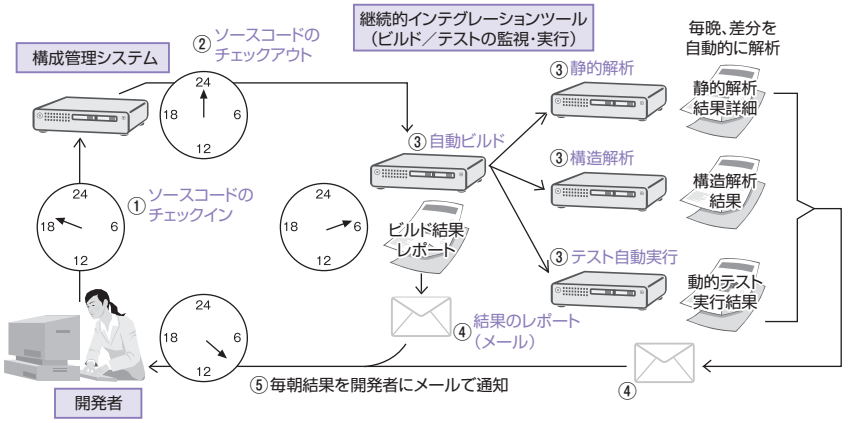
◆静的解析ツールの利用タイミングが遅れたため、ツールの指摘に対する利用者(＝開発者)の対応が表面的なものになってしまい、ツール適用効果が得られなかったという事例がある。

事例

ツール利用のタイミングについて、以下に例を示す。

ソースコード静的解析ツールとソースコードバージョン管理ツールとを連携し、夜間バッチを利用して、チェックインされたソースコードを自動的に静的解析ツールにかけて、朝には担当者に結果がメールで配信される仕組みを構築した。

①開発者は、帰宅前にソースコードをリポジトリにチェックインする
②夜になると継続的インテグレーションツールがソースコードの差分を読み込む
③深夜に、ソースコードの差分に関して、静的解析、構造解析、ビルド、更に可能であればテスト実行までを自動的に実行する
④静的解析、構造解析、ビルド実行、テスト実行の結果を朝までに開発者宛てにメール自動送信する
⑤開発者は、出社したらメールを開き、結果を確認する
★このサービスを受けるには、構成管理が出来ていることが前提となっている。



2.1.4 ツールチェーン

関連:2.1.1/2.1.2/2.1.5/2.1.7

**ツールチェーンは環境構築の負荷が大きいので、
推奨ツール群を示すだけではなく、
適用部門での負荷を軽減する仕組みも検討する。**

解説

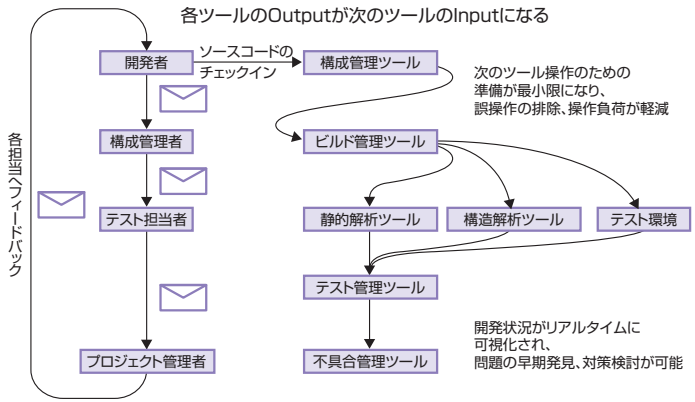
ツールチェーンを活用すると次のようなメリットがある。

・ツール連携により、ツール間をまたがった操作を自動化
→誤操作の排除、操作負荷軽減に繋がる

・各ツールに蓄積された開発情報を用い、定量的プロジェクト管理支援ツールで可視化
→同一視点で可視化して比較可能になり、問題の早期発見、対策検討に役立つ

しかし、ツールチェーンは単独ツール適用に比べて効果が大きい反面、導入が困難である。ツールチェーン環境構築には、各ツールを単独で適用するとき以上の手間・コストとノウハウが必要になるからである。そこで、ツールチェーンを推奨する際には、ツールチェーン環境構築手順の具体的な説明資料、あるいは環境構築自動化の仕組みも併せて提供することを検討すべきである。

ツール連携機能には、ツール付属のプラグインを使うと各ツールのバージョンアップへの追随が容易である。必要なプラグインがなければ、推進部門がプラグインを作成し、利用部門へ設定手順書を提供するか、自動設定機能を提供するとよい。



留意点

ツールチェーン全体のセキュリティ確保、シングルサインオンを実装することが望ましいが、実装すると社内活用に限定され、社外の委託先には提供できない場合もある。

事例

ある企業では製品の高機能化・多機能化に伴い、多くの人員によって開発が行われるようになり、大量の成果物を適切に管理し、情報共有することが難しくなってきた。この問題を解決するため、ソフトウェア開発の実装～テスト工程向けに下記の開発環境(ツールチェーン)を構築し、作業の自動化、及び情報の可視化を実現可能とした。また、それらのツール普及拡大のための仕組みを作成した。

【ツールチェーン作成のポイント】

・自製ツールとOSSを用い、利用者のライセンス費負担は不要
・ツールの連携により、ツール間をまたがった操作を自動化
・各ツールに蓄積された開発情報は定量的プロジェクト管理支援ツールで可視化
・仮想サーバ上に環境構築し、運用開始までのリードタイムを短縮

【推奨ツールチェーン選定の手順】

・実装工程以降のユースケースシナリオ分析を行い、ツールチェーンに必要なツールと連携機能を導出
・必要なツールを機能/費用/社内導入実績の観点でベンチマークして選定
・連携機能はツール付属のプラグインを利用。プラグインが入手できないものは内作
・セキュリティ確保(ウイルスチェックツール、OpenLDAPによる統合認証)

【ツールチェーン構成ツール】

・構成管理ツール：Subversion
・継続的インテグレーションツール：Jenkins
・ソースコード解析管理ツール：自製ツール
・不具合管理ツール：Redmine もしくは自製ツール
・テスト管理ツール：自製ツール
・定量的管理支援ツール：自製ツール

・上記の連携に必要なプラグイン

【ツールチェーンを推奨する上での工夫】

・このツールチェーン環境を仮想サーバ上に環境構築して、仮想サーバイメージを配布することで、ツールチェーン環境の新規構築に要する時間を大幅に短縮
・シングルサインオンを導入して、セキュリティを確保しつつ操作性を向上
・利用者向け教育を実施
・ベストプラクティスに基づく活用ガイド作成

【問題点】

・社内ユーザ認証システムとの連携によりセキュリティ確保やシングルサインオンを導入したため、社内活用限定となり、社外の委託先とは開発環境を共有できない。
・仮想サーバのイメージを配布する方式のため、利用者自身がメンテナンス出来ず、ツール提供側がメンテナンスする必要がある。例えば、使用する各 OSS ツールのバージョンアップでも提供側がメンテナンスしなければならない。

【効果】

仮想サーバを構築したイメージを配布する方式とした結果、運用開始までのリードタイムを短縮出来た。

Column

最近の仮想化ソフトは実行性能が良いため、仮想サーバだからということで、より高いスペックが要求されることはなくなってきた。

2.1.5 標準ツール選定とガイドライン制定

関連:2.1.1/2.1.2/2.1.3/2.1.4/2.1.6

**標準ツール／推奨ツールとするには、
移行しやすいツールを選定し、
ツール適用に関するガイドラインを制定する。**

解説

生産性向上効果や投資対効果の最大化、ノウハウ共有などを目的に全社的にツール活用を推進するには、ガイドラインを制定し、標準ツール/推奨ツールとして整理することが望ましい。ガイドラインでは、ツール教育受講についても示す。

部門(ドメイン)固有ツールのうち、標準ツールと同種のツールであれば、標準ツールへ移行させる。それを容易にするためにも、ツール選定にあたり、ツール利用の現状調査を行い、最大公約数となるツールを選択し、スタッフ部門で評価した上で全社標準ツールとしてガイドラインを制定するという方法がある。

留意点

欧米のツールベンダは統廃合が激しく、サポート方針が変更されることも珍しくないため、ベンダ評価も実施すること。

事例

ある製品の開発における事例を以下に示す。この開発では標準ツールとするにあたり、ツール導入におけるイニシャル・コストの問題を回避するため、OSS の中から標準ツールを選択した。

- コードチェック　：　FindBugs、CheckStyle、CCFinderX
- テスト実行　：　xUnit(CppUniit、JUnitなど)、Selenium、Sikuli
- カバレッジ測定　：　Cobertura/Jcoverage
- メモリ関連検証　：　valgrind
- 性能測定・性能検証：　EclipseTPTP
- 負荷テスト　：　JMeter

2.1.6 特徴の異なる静的解析ツールの使い分け

関連：1.1.6/2.1.1/2.1.2/2.1.3/2.1.4/2.1.5

静的解析ツールを用いる場合、ツールの特徴を把握すること。そして、開発のどのフェーズで用いるか、どういう問題の検出を期待するかによって、いつ・誰が・どのツールを用いるかを決定する。

解説

静的解析ツールを適用する場合、ツールが検出可能な欠陥種別など、特徴の異なる複数の静的解析ツールを適用することも検討する。

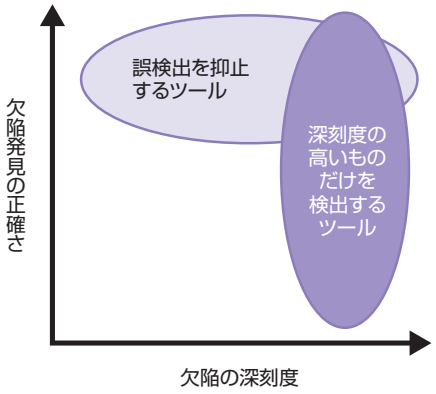
例えば、以下のような3つのツールを適用する。

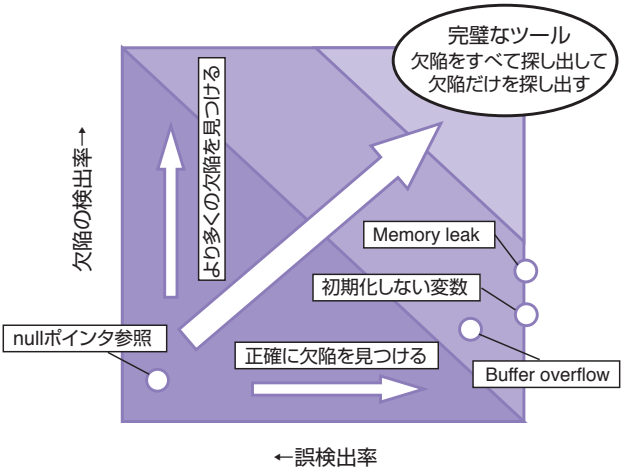
〔ツール1〕比較的短時間で大規模なコードベースにおける一般的なコーディング欠陥や疑わしいコーディングパターンを発見するのを得意とするもの。

〔ツール2〕同様のツールに比べて終了までの時間が長くかかるが、より深いパス解析を必要とするあいまいな欠陥や疑わしいコーディングパターンを発見できるもの。

〔ツール3〕プログラムにおける主に3つの最も一般的なタイプのソフトウェアの欠陥（初期化されていない変数、nilポインタの逆参照、範囲外配列インデックスの使用）を早く簡単に発見できるもの。

それぞれのツールの特徴を考慮し、適用する時期を検討するとよい（コーディング終了後、レビュー前、チェックイン前、単体テスト実行前、外部チームからの受け入れ時、など）。





静的解析ツールの特徴を把握する

留意点

一般的な静的解析ツールには、ツールが検出すべき問題点を定義するデータベースがあり、検出数を調整する機能がある。しかし、それでも静的解析ツールでの誤検出は避けられないので、誤検出抑止のためのフィルター/ラッパーを自部門向けに作成して、静的解析ツールと組み合わせて使うとよい。

Column

複数の静的解析ツールを使い分けるというのは、ツールの導入費用や実行時間などに厳しい制約が無い場合に可能なことであって、自分達でツールを準備、あるいは実行できない場合は、第三者によるCDI(Code Inspection)サービスの利用も検討するとよい。

2.1.7 Android開発での模擬環境と実機環境の使い分け

関連：1.1.3 3.2.1

シミュレーション環境と実機環境でのテストケースを切り分ける。

解説

Androidには豊富な開発環境が整備されている。テスト環境としては、OSS(オープン・ソース・ソフトウェア)で高速なバイナリシミュレータQemu(キューエミュ)を用いた仮想HW(goldfish)が用意されており、PC上で製品としてのシステムシミュレーションが可能になっている。

goldfishは、基本HWのモデルをそろえている。その他の周辺HWについては、利用者がモデルを作成する。そのためのツールとして、Android仮想デバイス(Android Virtual Device : AVD)の設定をサポートしている。

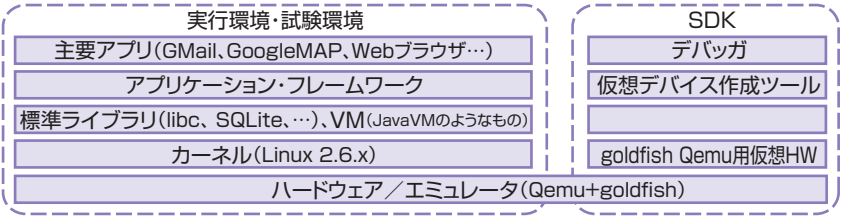
ただし、goldfishによるシミュレーション環境は、実機用のソフトウェアをそのまま使用出来るので機能面はかなり忠実に(低レベルの抽象度で)模擬するが、タイミングについては実機のタイミングを模擬していない。

つまり、機能テストについては、製品を構成する部品がそろった環境で実機用のカーネルから、デバイスドライバ、ミドルウェア、アプリまでを統合してテスト出来るので、ほとんどのテストケースを実行することが出来る。とは言っても、製品の外部については適切なモデルを作成して組み込む必要があり、それはたやすいことでないことに注意すべきである。

一方、タイミングに関するテストは、実機での実行結果と異なるため、タイミングに依存するテストケースは実機環境で行わなければならない。実機環境で実施するためのテストプログラムの検証にシミュレーション環境を使用することによって、実機環境使用時間を短縮出来る。

シミュレーション環境で出来るテストケース、実機で実施すべきテストケース、実機使用前にシミュレーション環境で準備出来る作業を事前に切り分けて、以下のことを心がけるべきである。

①無意味なテストを実施しない(不適切なテスト環境でのテスト実施排除)。
②同じテストケースの重複テスト実行を避ける(無駄を排除)。
③実機環境での作業時間を最小にする(テスト効率化)。



Android実行環境と統合開発環境

留意点

製品の外部については適切な抽象度のモデル(外界モデル)を作成して組み込む必要があり、それはたやすいことではない。テスト対象が何であるかを明確にした上で、作成するモデルの抽象度を決定する。いたずらに抽象度を下げて現実の動作を、仕組みまで含めて忠実にモデリングすることが優れたモデリング技術ではないことを理解すべきである。

事例

Qemuを用いたシステムシミュレーションは、タイミングを模擬出来ないという短所があるが、Qemuにアドオンする形で実機と同じタイミングを再現する方法について、以下に示す。

■論文「QEMUによるHW/SW 協調シミュレータの構築」(設計・検証技術、一般セッション、アーキテクチャ、情報処理学会創立50周年記念)の方法によると、ソフトウェア内部のタイミング(命令レベルの時間精度)を保証し、かつ周辺ハードウェアモデルに設計通りのタイミングを再現させることが出来るため、タイミングに依存するテストケースについても、かなりの部分を実機レスでテストすることが出来る。この方法は実機と同等の実行性能であることも大きな特徴である。

■単純にタイミングを再現するだけであれば、それほど高度な技術を必要としないが、実行性能が実機の何百倍も遅かったら、ソフトウェア検証用として用をなさない。ハードウェア検証用シミュレータには、この手のものが多いので、シミュレータを選択するときには、ハードウェア検証用シミュレータとソフトウェア検証用シミュレータの違いを意識する必要がある。前述の論文では、高速なソフトウェア検証用シミュレータとしてQemuを用いた。

Column

CPU動作をモデル化することによって、“コンピュータを使ってコンピュータをシミュレートする”ものを、エミュレータあるいは命令セットシミュレータ(ISS：Instruction Set Simulator)と呼ぶ。Qemuはこれに属する。

一般的にISSでは高速化のため、1命令ごとにシミュレートするのではなく、命令ブロック(ベーシックブロックとも呼ぶ)単位でシミュレートする。

2.2 公的機関における基準

2.2.1 公的機関における基準

関連：1.3.3
2.1.1
2.1.6

高度な安全性を要求される製品に対する、公的機関による安全性検証の例

解説

開発対象システムのソフトウェア不具合が甚大な人的・経済的損失を引き起こすような問題を特定し、その問題の有無をテストする場合には、様々な観点でのテストが必要になる。

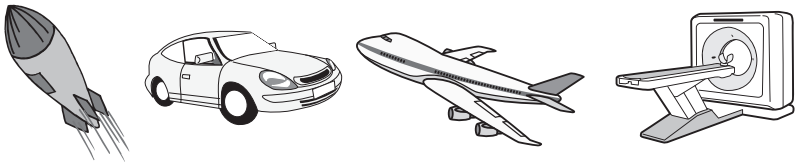
例えば、以下のような観点でのテストをそれぞれ実施することが考えられる。

①コーディングの欠陥(実装)
②アルゴリズムの欠陥(設計・ロジック)
③タスクの干渉(競合状態、データ破損)
④故障保護の不足

ただし、テスト対象とする問題を事前に特定することが重要であり、問題を特定せずに、これらすべてのテストを実施しても効果が期待出来ず、コストやスケジュールを無駄に消費する可能性が高い。

また、どこまでコストをかけて行うかは、製品に要求される安全性・信頼性の程度によって判断する必要がある。

公的機関がどのように、どこまで検証して問題の有無を判断したかは、参考になる。



ソフトウェアの不具合が甚大な人的・経済的損失を引き起こしかねない組込みシステム

留意点

様々な観点から多くの検証を行うには、ツールを活用して効率化を図る必要がある。その際、ツールには得手・不得手があるため、複数のツールを組み合わせることが有効である。

事例

検証を行っている公的機関として、例えばNASAでは、実際に以下のような分析・検証を行い、ソフトウェア上の欠陥が見つからなかったことを報告している。

1.コーディングの欠陥(実装)

静的ソースコード解析ツールを使用したソフトウェア実装解析にそれぞれ特徴の異なる3種類のツールを使用。

2.アルゴリズムの欠陥(設計・ロジック)

モデルベース設計ツールを使用したソフトウェアのアルゴリズム設計分析を実施。

3.タスクの干渉(競合状態、データ破損)

ツールを用いた設計分析を行っている。SPINなどのモデル検査ツールを使用したソフトウェアロジックモデルのチェック。

4.故障保護の不足

ロジックモデルのチェックとして、問題発生条件の組み合わせテストを行った。FTAなどの分析手法を用いて故障保護の不足が無いかの検証を実施。

Part 3

テストの基本的テクニック

Part1 でテストには限界があり、完全には出来ないことを示し、それ故にこそ限られたコストの中で効率良くテスト実施するための工夫について言及しています。また Part2 ではテストボリュームが非現実的なほど増加していることを示し、その対策としてツール活用を提示していますが、効率的かつ漏れなくテストを行うには、これに加えて適切な手法・技法の採用が欠かせません。

実際の組込みソフトウェア開発現場で新たな手法・技法を採用する際、専門の解説書に記載された通りに実施することが難しかったり、期待する効果が得られないことも少なくありません。

テスト手法・技法の詳細解説はそれぞれの専門書に委ねることとして、本章では、テスト設計者及びマネージャ、テストを実施、評価する品質評価部門担当者及びテストエンジニアが実際の現場で基本的なテスト手法・技法を採用する際、その手法・技法に何を期待してどのように採用すべきかの考え方、採用するときの留意事項を中心に事例を紹介します。

3.1 テスト技術・技法の分類

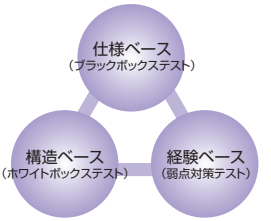
3.1.1 単体テストの観点に対応したテスト技法

関連：1.1.2 1.3.2 1.3.3

単体テスト設計では、仕様/構造/経験の各観点に立ち、おのおのに適した技法を活用して設計する。

解説

テストには複数の観点とそれに対応した技法があり、これらは相互補完関係にあることを理解してテスト設計を進めることを推奨する。単体テスト設計に際しては、例えば以下のように、複数の観点に基づいた設計を心がけることで、テスト漏れを防ぐよう配慮すべきである。



①仕様ベースのブラックボックステスト技法
②構造ベースのホワイトボックステスト技法
③上記を補完する経験ベースの弱点対策テスト

機能実装漏れは構造ベースの観点では摘出出来ないため、仕様ベースのテストが必須である。またこうした体系的な技法では、製品特性・プロジェクト特性などによって発生する弱点に焦点をあてていないため、経験ベースでの補完を要する。例えば次の例のように。

ホワイトボックステストにおける網羅率には、条件網羅/判定条件網羅/複合条件網羅があるが、ソフトウェア全体に対しては条件網羅を行うことを基本として、過去の経験に照らし脆弱と思われるモジュールに対しては、最も組み合わせ条件が多い複合条件網羅を適用する。

ソフトウェア全体に対して一律に複合条件網羅を適用すると、テストコストの増大を招いてしまうが、弱点にフォーカスすることでコストに配慮しつつ品質向上を図ることも可能となる。

留意点

網羅率は十分性の指標を定量化出来るが、経験ベースのテストでは数値として見える化することが難しいケースも考えられる。その場合、機械的に定量化しても

留意点 単なる数字操作に成りかねない。従って、弱点フォローの実施度合いなどは経験者の判断に基づいて評価するなどの対応が現実的である。

事例

仕様／構造／経験の３つの観点に基づく単体テスト設計の検討手順の例を、以下に示す。

■仕様ベースのテスト

詳細設計仕様書や関数/メソッド定義を入力としてプログラムが仕様通りに動作するか、実装漏れが無いかを確認することを目的とし、ブラックボックステスト技法を用いてテスト項目を設計する。

■構造ベースのテスト

ソースコードを入力としてプログラムの論理構造の正しさを確認することを目的とし、ホワイトボックステスト技法(主に制御フローテスト技法)を用いてテスト項目を設計する。

■経験ベースのテスト

上記2つの体系的な技法によるテストの補完的な位置付けとして、経験上バグを作り込みやすい部分のテスト項目を設計する。

Column

独立行政法人 宇宙航空研究開発機構（JAXA）がRTOSを高信頼化するためのテクニックをまとめた「リアルタイムOS高信頼化ハンドブック」という技術資料がある。高信頼化は、信頼に足る機能を有するだけではなく、適切な技法を用いて検証されることによって実現できるという考えの下、主に動的検証に着目して、テストを円滑かつ確実に進めるための目的設定、分析、設計、実施と報告などについてサンプルを交えつつ説明している。

高信頼性は組込みシステム共通の命題であり、多くの組込みシステム開発における、テストのガイドライン策定、テスト仕様書構築、テスト設計の参考になると思われる。JAXAの資料に書かれている方針を参考に、自分達のスタイルに合ったテストのやり方や、なぜその手法、技法を使うのかを考える際に有用であろう。

テスト技法については､その技法の考え方から技術的説明までを示しており、テスト教育の教材としても有効な書と言えるであろう。ただし、詳細説明に関してはμITRON開発を前提としているため、一般的な制御系カーネル開発に詳細までをそのまま活用できるものではない点には注意すべきである。

3.1.2 同値分割、境界値分析手法の活用

関連:

**同値分割・境界値分析においては、
値の決定方法や考え方を明確にする。**

解説

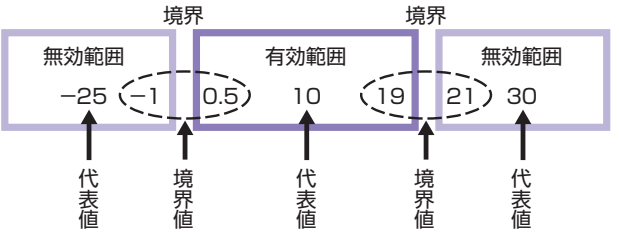
仕様ベースのテストでよく使われる手法として、境界値分析や同値分割がある。

例えば、関数の入力に用いるパラメータの範囲を同じ結果をもたらす集合(同値クラス)に分割し、各集合の代表値を用いたり、その同値クラスの境界値付近(境界値、境界値の前後)の値を用いたりしてテストを行う。

代表値決定方法には、入力パラメータの取り得る範囲が閉じている場合と閉じていない場合があるなど様々なケースを想定出来るので、あらかじめそのルールを定めておくとよい。

複数の条件式により同値クラスが分割されているような場合には、あらかじめ条件を複合させて1つの条件式とみなすのではなく、それぞれの条件式に対して単一の条件の場合と同様に、同値クラスを設定する。

また、境界値分析では1つの境界点に対して、境界値の前後を合わせた3つを採用する、あるいは境界値とその外側の2つを採用するなど、考え方は幾つも存在するため、事前にその考え方を明確にすべきである。



留意点

◆うるう年における 2 月 29 日などは特殊であり、それゆえ注意を要する例の 1 つと言えよう。また、小数点の丸め誤差は演算順序により結果が変わってくる。こうした影響も境界値分析において考慮すべきである。
◆隠れた境界にも留意すること。扱うデータが連続か離散か、データ型や CPU が 16/32/64 ビットのいずれか、などによっても隠れた境界は変わってくる。

事例

ある組織における同値分割・境界値分析の基準値をどのように定めているかについて、以下に例を示した。

■例えば、条件式 10＜x＜200のように、入力パラメータの取り得る範囲が閉じている場合は、以下のような計算式を用いて代表値を求める。
同値クラス代表基準値 =(最大境界値－最小境界値)/2 = 95
有効同値クラス代表値 =(最大境界値＋最小境界値)/2 = 105
負方向無効同値クラス代表値 = 最小境界値－同値クラス代表基準値 = －85
正方向無効同値クラス代表値 = 最大境界値－同値クラス代表基準値 = 295

■算出された結果が、設定不可能な値(パラメータの上限を超えているなど、上記例なら入力パラメータがunsignedの8ビット変数の場合)である場合は、取り得る最大値、または最小値と境界値の中間値を代表値とする。
負方向無効同値クラス代表値=(変数の取り得る最小値+最小境界値)/2 = 5
正方向無効同値クラス代表値=(変数の取り得る最大値+最大境界値)/2 = 227

■条件式 x≦75のように、入力パラメータの取り得る範囲が負の方向に閉じていない場合は、その変数が取り得る最小値を最小境界値と仮定し、範囲が閉じた場合と同様に考える。ただし、負方向無効同値クラス代表値は存在しない。

■同様に、正方向に閉じていない場合(例：x≧75)は、その変数の取り得る最大値を最大境界値と仮定し、範囲が閉じた場合と同様に考える。

3.1.3 加速度テストの目的と適用上の注意事項

関連：1.1.3
1.2.1

加速度テストの実施にあたっては、
目的に対する適用妥当性や適用上の限界も考慮する。

解説

ここではソフトウェアの加速度テストについて、その目的や適用上の注意事項などについて解説する。

①目的

加速度テストは以下のような目的で実施される。

・テスト時間を短縮し、また短時間でテスト網羅度を向上させる。
・過負荷状態のテスト環境を再現する。
・実機で所定期間内にテストするには非現実的なほどに長時間を要する。
・リグレッションテストでのテスト網羅度を上げてデグレードを回避する。

②分類

加速する対象として、テスト対象そのものを加速するケースとテスト対象の周辺要素を加速するケースがある。

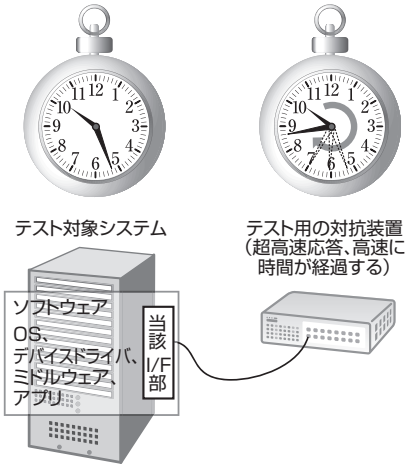
前者の事例として、通信機器での対抗装置からの応答を早めること、すなわち、対抗装置が実際の処理応答をはるかに超えるような高速処理状態を再現するケースが該当する。また後者の事例としては、携帯電話のGUI操作のテストを実施する場合に、通話やメールの受信を超高頻度に発生させるようなケースがある。

③注意事項・適用限界

加速度テストの適用にあたっては、次のような点に注意が必要である。

・物理現象そのものは加速出来ない
例えば、冷蔵庫の温度設定を変えることなく、製氷機で氷が出来る時間を短縮することは出来ない。温度計の計測値下降を加速することは出来ても、実際に温度が下がらなければ水は凍らない。
・内部(テスト対象)と外部(対抗)の時間経過が異なる
対抗装置からの応答を速めることによって、テスト対象モジュール(インターフェース部)の高負荷状態を再現することは出来る。しかし、テスト対象システムの時間が早く経過するわけではなく、対抗装置の時間だけが早く経過

している。テスト対象システムでは対抗とのインターフェース部が高負荷になるのみで、他の部分は高速処理もしていない。よって、テスト対象モジュールを含むテスト対象システムの動作が長時間経過することによって発生するメモリーリークを短時間で再現することは出来ない。



留意点

◆外部環境モデルまでを含めて時間精度保証が可能な SILS システムシミュレータであれば、システム全体の時間経過を早めることが出来、通常動作が長時間経過した場合に発生するメモリーリークについてもかなりの精度で再現することが出来る。

◆ただし、現状は以下のような事情から、実用レベルの製品開発で適用されているケースは多くない。

- システムシミュレータは高価である
- 外界(外部環境)モデルまで含めたシステムのモデリングには、高度なモデリング技術が必要である
- 時間精度を保証(≒タイミングを再現)しつつ高速実行可能なシステムシミュレータは少ない
- 外界モデルまで含めて時間精度を保証可能なシミュレータは少ない

事例

加速度テストの適用例及び今後の可能性について以下に示す。

■加速度テストを実施している例としては、以下のようなものがある。

- デジカメなどのバッテリー残量の減少速度を加速する。
- 炊飯器の温度調整をタイマー調整し時間短縮して実施する。

■今後、加速度テストの適用が有用と思われるケースとしては、プリンタのインク量の模擬がある。インク量が多→少となるスレッシュホールドを通過するケースを通常のやり方で模擬しようとすると長時間を要するため、加速度テスト適用が有効であろう。

3.1.4 合意に基づいた非機能テスト項目抽出

関連：1.2.3
1.3.1
1.3.2

非機能テストの項目の妥当性・十分性は、関係者の合意に基づいて設定する。

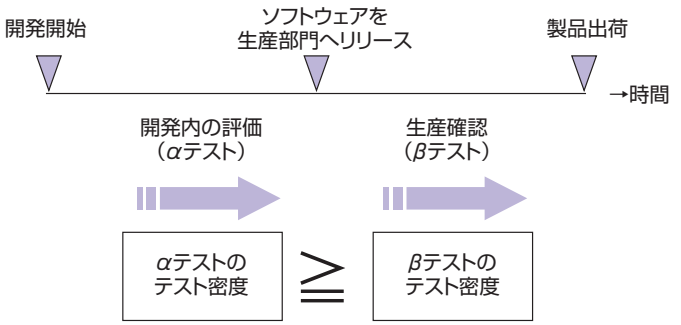
解説

非機能要件は環境に依存するところが多い上、ユーザからの明確な要求があるわけではないため、根拠や優先順位、十分性の判断は概して難しいものとなる。

非機能テスト項目の抽出は、テストする観点を製品ごとにあらかじめ設定し、その中から非機能要件の記載に従って、必要に応じて抽出する。この際、テスト項目の妥当性・十分性を関係者の合意に基づいて確認することがポイントである。

ある組織では、非機能テストを大中小の項目でカテゴライズしており、まず「非機能グレードの大項目：性能・拡張性」「中項目：性能目標値」の中の小項目からテスト項目を選択していく、という優先順位を付けている。

また、テスト密度については、αテストのテスト密度よりβテストのテスト密度が小さくなるよう基準値を設定する、とルール化している。



Column

テスト密度の算出に用いる測定量とその単位

●ソースコード全行数(TLOC)

①計測単位：KLOC

②計測方法：

ソフトウェアの一部または全体に対し、そのソースコードの物理行数を計測し、総和を算出する。計算にあたっては以下を条件とする。

・コメント行はすべて計測する。

・空行(何も記述していない行)もすべて計測する。

・1つの処理が複数行にまたがって記述されている場合も、純粋にその行数分を計測する。

●テスト項目数(NOTI)

①計測単位：項目

②計測方法：

・テストを実施した項目数を計測する。

●テスト密度(DOTI)

①計測単位：項目

②計測方法：

テスト項目数/ソースコード全行数

DOTI=NOTI/TLOC

3.1.5 複数因子間組み合わせの勘所

関連：1.2.3
1.3.3

**2因子組み合わせテストの実施方法を工夫して、
許容コスト内でテスト網羅率やテスト客観性を向上させる。**

解説

複数因子間の組み合わせは往々にして組み合わせの数が膨大になり、コスト面からすべてはテスト実施できず、一方で品質面からは高い網羅率が要求される。よって、そのバランスをとることが重要である。複数因子間組み合わせの手法を適用する際にバランスをとるための勘所としては、以下のようなものがある。

- 品質確保するための手法として機能同士の組み合わせを2因子間組み合わせテストで行う場合、テストケース抽出を経験と勘のみに頼るのではなく論理的に行うことは重要である。とくにその網羅率を原則100%として実現しようとする際には、その実施効率に貢献できるだけでなく、顧客など第三者への合理的な説明を行う上でも効果がある。
- 2因子組み合わせ法を工夫することにより、ほぼ9割程度のバグを組み合わせテストで検出可能というデータもあるが、因子の選び方に品質が左右されることもある。従って、いかに漏れ無く因子を抽出出来るかがカギとなり、そのためにはお客様視点で6W2Hの観点から機能や利用シーンを分析するのも有効である。
- 因子間に論理関係や順序関係がある場合は、状態遷移テストなど他の手法との併用が有効である。
- HAYST法においては因子の抽出漏れは致命的となるため、注意深く分析することが前提となる。因子が増えても直交表サイズがそれほど大きくなるとは限らないため、幅広い因子抽出を心がけることに注力したほうがよい。
- 一方、水準が増えると直交表の維持が難しくなる傾向があるので、その場合には絞り込みや集約を行う。時系列に変わっていく因子、論理関係・順序関係を持つ因子同士というのは直交表ではテスト出来ないため、状態遷移図、ディシジョンテーブルの併用が効果的であろう。

留意点

複数因子間の組み合わせテスト網羅率はプロジェクト特性、要求品質レベルなどの製品特性に基づいてマネージャやリーダなどが判断して設定することになる。この場合でも60%以上実施することが望ましいなど、何らかの指標を設けて運用すべきであろう。

事例

■手法の選択事例

①適切な因子・水準を漏れなく洗い出す。

因子の洗い出しにあたっては、テスト対象となる機能・条件・変数などの項目を抽出する。また、他の因子に影響を与える依存関係などを確認する。

②水準の洗い出しは、因子ごとにバリエーション項目を洗い出す。

水準の設定には同値分割法や境界値分析などを用いる。

③そのまますべての組み合わせを実施すると膨大なテストケースとなる場合、直交表やAll-Pairs法を用いてテスト項目の絞り込みを行い、論理的にカバレッジ100%とし、品質確保することを目指す。

No.	s0	s1	s2	s3	s4	s0'	s1'	s2'	s3'	s4'	s0"
1	e1	e8	–	–	–	e1	e8	–	–	–	end
2	e1	e8	–	–	–	e2	–	e3	e5	–	end
3	e1	e8	–	–	–	e1	e9	–	–	end	–
4	e1	e8	–	–	–	e2	–	e4	–	end	–
5	e1	e9	–	–	e7	–	e9	–	–	end	–
6	e1	e9	–	–	e6	–	–	–	e5	–	end
7	e1	e9	–	–	e7	–	e8	–	–	–	end
8	e2	–	e3	e5	–	e2	–	e3	e5	–	end
9	e2	–	e3	e5	–	e2	–	e4	–	end	–
10	e2	–	e3	e5	–	e1	e8	–	–	–	end
11	e2	–	e3	e5	–	e1	e9	–	–	end	–
12	e2	–	e4	–	e7	–	e9	–	–	end	–
13	e2	–	e4	–	e7	–	e8	–	–	–	end
14	e2	–	e4	–	e6	–	–	–	e5	–	end

■製品への適用例

①情報端末製品の接続確認テスト試行

・現行のテスト仕様書(マトリクス)について、2因子間の網羅率を測定

・現行のテスト仕様書をベースに、水準の見直しや禁則を設定してテストケースを作成

⇒　因子：9個(2水準：6個、3水準：3個)
　　【テストケース数】約130件→106件　2因子間網羅率：72%→100%

②認証用装置システム

・「ユーザ登録～認証」テストへの適用例

⇒　因子：17個(2水準：7個、3水準：6個、4水準：2個、5水準：2個)
　　【テストケース数】約1,168件→64件　2因子間網羅率：100%

③画像装置

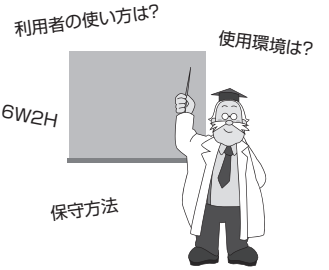
・論理回路に基づく組み合わせテストケースへの適用

⇒　因子：12個(2水準：6個、3水準：2個、4水準：3個、5水準：1個)
　　40件のテストケースで2因子間網羅率：100%

■ 6W2H の視点でのテスト項目抽出

炊飯器開発でのテスト項目抽出に対して、6W2Hの視点を活用して発想を広げていった例を以下に示す。

・When　：朝・昼・夕食仕度時/就寝前(タイマーセット)/...
・Where：台所で/学校や職場で/屋外で/...
・Who　：主婦が/男性(既婚者)が/未婚者が/子供が/...
・What　：白米を/玄米を/パンを/ケーキを/焼き魚を/...
・Why　：米を炊くため/パン・ケーキを作るため/魚を焼くため/...
・Whom：自分に/家族に/友人・知人に/被災者に/...
・How　：共通プラットフォームは？/各機能の類似性は？(設計上考慮したことなど)...
・How much：(価格・量)各機能に対するユーザの期待結果(各機能がどのように動作するか、操作性は、など)...



3.1.6 モデル図を用いたテスト項目抽出手法

関連：1.4.1

モデル図を活用したテスト設計では各図の特徴を理解し、特徴を活かした項目抽出手法を採用する。

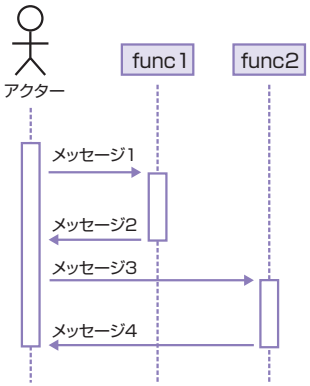
解説

近年、組込みシステムにおいてもモデルベース開発の取組みも実施されるようになっている。ここではある組込み端末装置開発にUMLを適用した場合に、その特徴を活用してテスト設計を行う際のポイントを紹介する。

①シーケンス図からのテスト設計

・シーケンス図からのテスト項目設計では、シーケンス図内のメッセージをすべて網羅する観点でテスト項目を作成する。シーケンス図内に複合フラグメントがあり、分岐が発生する場合は分岐網羅するようにテスト項目を作成する。分岐網羅することで、すべての単機能を網羅することが出来る。

・シーケンス図の中で出力が確認出来る箇所はすべてテスト時の確認項目とする。シーケンス図内のメッセージを順にたどりながら、入力に対する出力が確認出来る箇所を見出す。

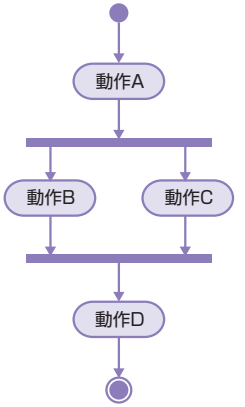


①シーケンス図からのテスト設計

・また、シーケンス図内のライフラインやメッセージ名を見ただけでは、テスト項目作成に必要な入力、出力が分からない場合がある。その際は、機能仕様書内のシーケンス図以外の記述（クラス図など）を参照したり、設計者に確認して情報を補う必要がある。

② アクティビティ図からのテスト設計

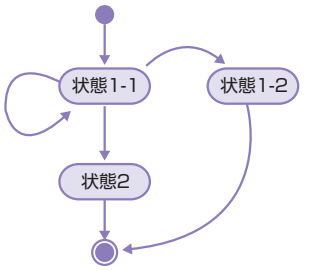
・アクティビティ図中のパスを分岐網羅する、すなわちすべてのパスを少なくとも1回は通るようにシナリオを作成し、そのシナリオを実行するためのテスト項目を作成する。
右図では、例えば「動作A」→「動作B」→「動作D」と「動作A」→「動作C」→「動作D」となるような2通りのシナリオを考えることになる。
・分岐網羅することで、アクティビティ図中の単機能をすべて網羅することが出来る。



②アクティビティ図からのテスト設計

③ ステートマシン図からのテスト設計

・状態遷移のテストでは、ステートマシン図や状態遷移表の中から単機能を抽出し、テスト項目を作成する。
・まず状態遷移の仕様を確認し、次に単機能を網羅するシナリオを作成し、最後にシナリオを実行するためのテスト項目を作成する。



③ステートマシン図からのテスト設計

テスト項目作成は次の順に行う。
(1)状態遷移の仕様を確認
(2)単機能を網羅するシナリオを作成
(3)シナリオを実行するためのテスト項目を作成
(4)状態遷移が発生しない部分のテスト項目の作成

留意点

シーケンス図はある局面のライフライン間のメッセージのやりとりの一例を示したものとして記述する場合が多い。その場合、シーケンス図をもとにしたテストもその“ある局面”のテストであり、仕様の全体に対してどの範囲を網羅しているかには留意する必要がある。

3.2 環境

3.2.1 性能評価環境がそろわない状況での性能評価

関連：1.1.3
2.1.7

**実機による性能評価環境がそろわない場合、
部分的な測定と理論的な裏付けから
実効性能を算出して評価する**

解説

厳しい性能要求や性能目標達成に高度な技術的課題がある場合には、上流工程での性能評価が必要となる。しかし、プロトタイプでフィージビリティを検証する余裕が無い場合もある。

また、性能テストは概して開発プロセスの後半で行われることが多いが、その実施時期になれば評価環境のシステム構成要素すべてが揃うとは限らない。

そのような場合、性能評価環境がそろわなくても出来る限り上流工程で実効性能を評価することを考えなければならない。一般的にはソフトウェア的に評価したり、またその結果に基づいて机上計算するなどの取り組みが推奨される。

例えば、性能評価ツールを活用し、ロジックの一部の実行時間測定を行い、その結果を積み重ねてシステム実効性能を理論的に算出するなど、ハードウェア環境が無くとも、出来る範囲で性能評価が実施出来るようにすることが望ましい。

留意点

◆こうした性能評価が常に適用出来るとは限らない。例えば、画像認識での「暗所でどこまで撮影できるか」のように、性能の良し悪しが俗人的判断とならざるを得ないような評価を上流工程での理論値の積み重ねで行うことは難しく、限界がある。

◆また、上流工程で実機を使わずに性能評価を実施出来たとしても、最終的には実機で性能評価すべきである。

事例

上流工程で行われている性能評価やそれらを念頭においた設計の具体例について、以下に示す。

■実機デバイスがまだ入手出来ていない上流工程においてデバイスドライバの性能評価を行う際、当該ロジックのパイプライン実行クロック数を算出し、目標性能をクロック数に換算した値と比較して評価する。この方法では、性能に大きな影響を与えるロジック部分のみの性能評価を行い、その他の部分は既存のデバイスドライバの性能測定値を用いて全体性能を机上で評価する。
■実時間シミュレータ(実機が単位時間に処理するクロック数分の命令を同じ仮想単位時間で模擬実行するシミュレータ)を用いて、ネットワーク装置の処理を仮想時間で計測し、性能評価する。

3.3 教育

3.3.1 テストに関する体系的教育カリキュラム例

関連:

ソフトウェアの開発やテストにかかわるエンジニアのスキル向上と平準化に向けた、体系的な取り組みを行う。

解説

総合テストやソフトウェア総合テストは第三者テストの観点から開発から独立したテストメンバにより行われるなど、開発の各工程で設計や品質保証など様々なエンジニアがテスト行為に携わる現状に鑑みると、ソフトウェア技術者全般を対象としたテストの手法、技法の知識習得教育を組織として体系的に実施することには大きな意義がある。

こうした教育の全体像を考えるにあたっては、テストエンジニアがどういうスキルを持つべきかの方針を明確にし、体系的にカリキュラムを考える必要がある。

また、現場ですぐに実践出来るよう、座学だけでなく演習も含めた実践的スキルの習得が出来るようなカリキュラムの工夫も必要であろう。

講座名	内容
ソフトウェアシステム概論	ソフトウェアシステムを取り巻く状況概論を解説。
テストの基礎	テストとは何か、なぜテストを行うのか、V字モデルとテストフェーズなどを解説。
テスト技法	同値分割法、境界値分析、デシジョンテーブル、状態遷移テストなどの技法を解説。
ソフトウェアメトリクス	ソフトウェアの規模、開発工数、開発期間、ソフトウェア欠陥率、ソフトウェア欠陥除去率、サイクロマティック複雑度、コードカバレッジなど品質指標を解説。

留意点

◆コアアーキテクチャは一度設計すると長年変更しないことが多く、設計者が定年退職してしまい、新人など次世代がそのノウハウや技術を継承出来ないというケースも現実に発生している。
◆そこで、近年では新人や中堅向けにコアアーキテクチャの教育を社内で実施する取り組みを行う企業も出てきた。現業の仕事はすべてキャンセルし２週間程度の集中研修を受講させることにより、技術の底上げに効果が出始めている。

事例

ある組織で運用中の個別カリキュラム例を以下に示す。

■単体テストのテスト項目設計のテクニックとテストツール活用のテクニック

不要な単体テスト項目を少なくするコーディング、ユニットテストツールの実装など、テストの効率化を意識した、実践的テクニックの知識を習得する。

■テスト技法

高品質のソフトウェアを製作するために、ソフトウェア開発工程全体で適切なテスト技法を用いた品質の作り込みを体得する。

■ソフトウェア品質保証

各種ソフトウェア品質保証技術、プロセス改善技術が編み出されてきた経緯を理解し、効果的にソフトウェア品質を向上させるためのポイントの本質を理解するなど、基本的な概念の理解から実践的技術活用までの知識を習得する。

■ソフトウェア品質管理と品質管理データの活用

実践的なソフトウェア品質管理とその運用方法を理解する必要がある。ソフトウェアの品質管理に必要な品質データをタイムリーに収集し、統計技法など各種技法を用いて実態を分析/評価し、PDCAでの運用実践するための知識を習得する。

■テスト計画の立て方、テスト戦略

効率的にテストを行い、品質を確保するためのテスト計画やテスト戦略の立案などテストチームリーダとして必要な能力を養成する。

■ SEPG/SQA 担当開発プロセススペシャリスト養成講座

ソフトウェア品質向上に向けた、開発プロセスの見える化と改善を図るために必要なスキルや管理方法を実践活動の中で体得する。品質管理計画、マスターテストプラン立案などの実習を含む。

上記の他､「テストエンジニアスキルのばらつき解決講座(仮称)」なども今後検討している。

組込みソフトウェア技術者向け教育メニューの具体例

C言語/C++言語/Java言語
MDA技術講座(Rhapsodyを用いたオブジェクト指向開発)
Perl入門
RTOS基礎/応用
UMLによるCプログラミング/C++プログラミング/Javaプログラミング
UMLモデリング
UML分析モデリング基礎(分析モデル作成のための基礎トレーニング)
UML分析モデリング実践(オブジェクト指向分析/設計講座)
アセンブラ基礎/応用
オブジェクト指向とUML入門
コーディング品質向上
ソフトウェアテスト技法基礎
ソフトウェアドキュメンテーション講座
ソフトウェア開発の構成管理と変更管理入門
デジタル回路入門
マイコンハードウェア入門
レビュー技法
構造化モデリング
設計モデリング・実践(C言語)
設計図活用
組込みC言語基礎1(ポート制御とビット演算)
組込みC言語基礎2(割込み制御)
組込みC言語基礎3(タイマ制御)
組込みC言語応用(チャタリング除去)
組込みLinuxアプリケーションプログラミング基礎
組込みLinuxデバイスドライバプログラミング基礎
組込みソフトウェアの代表的な詳細設計法
組込みソフトウェアの方式設計と実装法

3.3.2 テクニックと共にV&Vの考え方の理解が必要

関連:

テスト教育においては、テクニックだけではなく、利用者の使用方法や使用環境などの視点に立つことの重要性を理解させる。

解説

テスト設計においては、必要十分なテスト項目を洗い出すと共に、テスト環境の構築やテスト最適化といった関連事項も設計作業に含まれる。いずれの作業においても当該製品利用者の視点に基づくことが重要であり、その重要性はテスト教育においても強調されるべきである。

ソフトウェアV&Vにおいて、検証(Verification)は設計仕様書ベースに実施する一方、妥当性確認(Validation)では設計仕様の起点となっている利用者のニーズとそれに基づく業務要件・設計方針、製品仕様、利用者の運用スタイル、保守方法などに照らした妥当性確認を行う。従って、最終利用者の使用方法(製品の使われ方)、使用環境などに対する理解は不可欠であり、テスト教育でもテクニックだけでなく、考え方も理解させる必要がある。

例えば、テスト項目抽出法の1つに6W2Hで抽出する方法があり、これを活用すると利用者の使用方法や使用環境の視点に立って考えやすい、など。

Column

・検証(Verification)とは、仕様・設計・計画などの要求事項を満たしていることの確認であり、妥当性確認(Validation)とは、機能や性能が本来意図された用途や目的にかなっているか、実用上の有効性があるかの確認である。Barry W. Boehm氏が以下のように分かりやすくまとめている。

○Verification：Are we building the product right?
製品は仕様通りに作られているか？

○Validation：Are we building the right product?
製品はユーザが必要とすることを行っているか？

・IEEEを中心に米国でまとめているSWEBOK(Software Engineering Body Of Knowledge)では、V&Vはソフトウェアエンジニアリングプロセスの中間ステップの妥当性確認(Validation)も行うとしている。つまり、V&Vには動的なテストだけでなく、レビューやインスペクションなどの静的解析も含む。

付録　用語解説

■模擬環境

一般的に開発現場で用いられる模擬環境には、以下の種類がある。

●シミュレーション(Simulation)

一般用語：ふりをする(外見を同じようにする)。模擬実験。

コンピュータ業界用語：インターフェースを再現する。

●エミュレーション(Emulation)

一般用語：見習う(中身を同じようにする)。模倣。

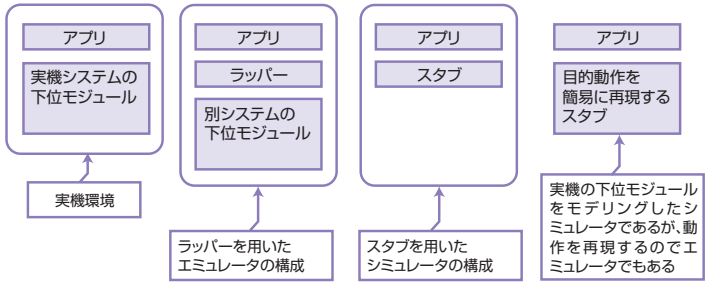
コンピュータ業界用語：中身の動作までを再現する。

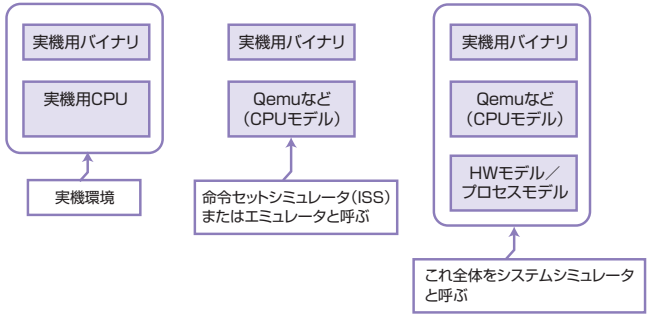
●ラッパー(Wrapper)

ラッパーは、異なるシステムで元のシステムの動作を代用するためにインターフェース調整をするモジュールである。つまり中身の動作を再現するので、エミュレータと言える。一方で、ラッパーはインターフェースを再現するのでシミュレータとも言える。

●スタブ(Stub)

スタブは、インターフェースを再現するのでシミュレータである。しかし、動作を再現するスタブもあり、その場合はエミュレータと言える。





●シミュレータの種類

シミュレーション形態は、大別するとHILSとSILSに分けることができる。

・HILS(Hardware In the Loop Simulation)
　模擬動作させる部分にハードウェアを組み込んだシミュレーション形態/シミュレータ。

・SILS(Software In the Loop Simulation)
　ソフトウェアのみで行うシミュレーション形態/シミュレータ。
　その他、シミュレータ関連ツールベンダーによると以下のような分類もあるが、定義が一意ではない。

・MILS : Model In the Loop Simulation

・PILS : Processor In the Loop Simulation

・(S)PILS : Software-based Processor In the Loop Simulation

■ソースコード行数の数え方

●一般的によく使われるソースコード行数の数え方

・全ソースコード行数(単位：ステップ、行、Lines)
　意味：コメント行、空白行も含めた行数
　用途：コメント量の妥当性を確認する際、(Lines － LOC)÷Lines でコメント行の比率を計る

・ソースコード行数(単位：LOC Lines Of Code)
　意味：コメント行、空白行を除いた行数
　用途：ソフトウェア規模を計る

●COCOMOⅡ ソフトウェア再利用モデル式

本式は、使用するには複雑で係数導出に手間がかかるが、様々な要素を考慮している点は参考になるであろう。

AAF = 0.4･DM＋0.3･CM＋0.3･IM

ESLOC＝ASLOC･[AA＋AAF･(1＋0.02･SU･UNFM)]/100(AAF≦50 のとき)

ESLOC＝ASLOC･(AA＋AAF＋SU･UNFM)/100 (AAF＞50 のとき)

ESLOC(Equivalent Source Lines Of Code)
　新規開発相当ソースコード行数

ASLOC(Adapted Source Lines Of Code)
　再利用される元のソフトウェアのソースコード行数

DM(Percent Design Modified)
　設計変更の割合(%)

CM（Percent Code Modified）
　ソースコード変更の割合（%）
IM（Percent of Integration Required for Modified Software）
　再利用される元のソフトウェアでコーディング及びテストに要した工数に対しての、再利用にあたってのコーディング及びテストに要する工数の割合（%）
AAF（Adaptation Adjustment Factor）
　ESLOC を計算するための中間変数
AA（Assessment and Assimilation）
　再利用ソフトウェアの評価とドキュメント修正の必要度
SU（Software Understanding increment）
　再利用ソフトウェアの設計、ソースコード、ドキュメントの明瞭さ
UNFM（Programmer Unfamiliarity）
　プログラマの熟練度

■カバレッジ

●主要なコードカバレッジ

・命令網羅（Statement Coverage：SC）
　コード内のすべての命令が少なくとも1回は実行されるようにテストを設計する。欠陥検出力は弱い。
・判定条件網羅、分岐網羅（Decision Coverage：DC、Branch Coverage：BC）
　コード内の判定条件の結果として、真になる場合と偽になる場合がそれぞれ少なくとも1回は出現するようにテストを設計する 。
・条件網羅（Condition Coverage：CC）
　コード内の条件判定における個々の条件について、すべての真偽が少なくとも1回は出現するようにテストを設計する 。
・判定条件/条件網羅（Decision/Condition Coverage：DC/CC）
　条件網羅と判定条件網羅を合わせたコードカバレッジ。個々の条件の真偽と、判定結果の真偽がそれぞれ少なくとも1回は出現するようにテストを設計する。
・複合条件網羅（Multiple Condition Coverage：MCC）
　コード内の判定文におけるすべての条件で、あり得るすべての結果の組み合せが少なくとも1回は出現するようにテストを設計する 。
・経路組み合わせ網羅（path coverage）

コード内のすべてのパス(制御パス)が少なくとも1回は実行されるようにテストを設計する。

●判定条件網羅、条件網羅、複合条件網羅

判定条件網羅(＝分岐網羅)、条件網羅、複合条件網羅の違い。

ある判定文の条件が「a=0 or b=0」であったとする。
「条件1 or 条件2」という複合条件である。

○判定条件網羅：判定条件の真、偽を実行
「a=0、b=1」(判定：真)
「a=1、b=1」(判定：偽)
※b=0つまり条件2：真となる組み合わせを実行していないがよしとする。
○条件網羅：各条件の可能な結果を1回はとる
「a=0、b=1」(条件1：真、条件2：偽)
「a=1、b=0」(条件1：偽、条件2：真)
※判定：偽となる組み合わせを実行していないがよしとする。
○複合条件網羅：各条件の可能な結果のすべての組み合わせを実行
「a=0、b=0」(条件1：真、条件2：真)
「a=0、b=1」(条件1：真、条件2：偽)
「a=1、b=0」(条件1：偽、条件2：真)
「a=1、b=1」(条件1：偽、条件2：偽)

・複合条件網羅が最も網羅率が高く、網羅率の高さは次の順番になる。
複合条件網羅　＞　条件網羅　＞　判定条件網羅
・判定条件網羅では、判定文の条件が複合条件であっても、ANDやORで結ばれた個々の条件には着目せずに、結果として判定が真の場合と偽の場合について実行すればよい。
・一方、条件網羅では、判定文が複合条件の場合には、その個々の条件の真偽に着目する。

参考文献

■ [ESCR] IPA/SEC：【改訂版】組込みソフトウェア開発向け コーディング作法ガイド［C 言語版］、翔泳社、2007

■ [ESCR C++] IPA/SEC：組込みソフトウェア開発向け コーディング作法ガイド［C++ 言語版］、オーム社、2010

■ [ESMG] IPA/SEC：組込みソフトウェア向け プロジェクト開発計画立案トレーニングガイド、2011

■ [ESMR] IPA/SEC：組込みソフトウェア向け プロジェクトマネジメントガイド［計画書編］、翔泳社、2006

■ [ESPR] IPA/SEC：【改訂版】組込みソフトウェア向け 開発プロセスガイド、翔泳社、2007

■ [ESQR] IPA/SEC：【改訂版】組込みソフトウェア開発向け 品質作り込みガイド、2012

■ [SEC journal ESxR 特集号] 平山、松田、他：SEC journal ESxR 特集号、2009

■ [SEC journal 25] 三原、松田、浜田、十山、他：組込みソフトウェアの高品質化への取り組み、SEC journal No.25、Vol.7、No.2、pp.63-69、2011

■ [SEC journal 29] 三原、石田、石井、他：SEC journal No.29、Vol.8、No.2、pp.63-68、2012

■ [SEC journal 30] 石井、石田：テスト部会活動紹介、SEC journal No.30、Vol.8、No.3、pp.131-134、2012

■「リアルタイム OS 高信頼化ハンドブック」技術資料番号 PEB-10056：独立行政法人宇宙航空研究開発機構、2011 年 2 月

執筆者（敬称略）

池野　　宏　　横河電機株式会社
石井　正悟　　IPA/SEC（東芝ソリューション株式会社）
石田　　茂　　IPA/SEC（株式会社東芝）
鈴木　伸一　　キヤノン株式会社
中村　光宏　　富士電機株式会社
羽田　　裕　　日本電気通信システム株式会社
馬場　匡史　　株式会社富士通コンピュータテクノロジーズ
浜口　幸雄　　株式会社日立情報制御ソリューションズ
三原　幸博　　IPA/SEC 組込み系プロジェクト プロジェクトリーダー
（アルパイン株式会社）
（50 音順）

監　修

組込み系ソフトウェアプロジェクト

SEC BOOKS
組込みソフトウェア開発における品質向上の勧め〔テスト編〜事例集〜〕

2012年11月12日　1版1刷発行

監 修 者　独立行政法人情報処理推進機構（IPA）
技術本部 ソフトウェア・エンジニアリング・センター（SEC）
発 行 人　松本　隆明
発 行 所　独立行政法人情報処理推進機構（IPA）
〒113-6591
東京都文京区本駒込二丁目 28 番 8 号
文京グリーンコート センターオフィス
URL　http://sec.ipa.go.jp/

ISBN978-4-905318-15-6 Printed in Japan