テスト技術者資格制度

Advanced Level シラバス日本語版
テクニカルテストアナリスト

Version 2012.J02

International Software Testing Qualifications Board

改訂履歴

◆ ISTQB®

バージョン	日付	摘要
ISEB v1.1	2001 年 9 月 4 日	ISEB Practitione シラバス
ISTQB 1.2E	2003 年 9 月	EOQ-SG による ISTQB Advanced Level シラバス
V2007	2007 年 10 月 12 日	テスト技術者 Advanced Level シラバス、2007 版
D100626	2010 年 6 月 26 日	2009 年に承認された変更箇所の反映と資格種別ごとの各章の分離
D101227	2010 年 12 月 27 日	書式変更と文字校正の反映
ドラフト版 V1	2011 年 9 月 17 日	合意されたスコーピングドキュメントに基づく、新しい独立した TTA シラバスの初版 AL WG レビュー
ドラフト版 V2	2011 年 11 月 20 日	NB レビュー版
Alpha 2012	2012 年 3 月 9 日	10 月版に対して受領したすべての NB コメントの反映
Beta 2012	2012 年 4 月 7 日	GA のためのベータ版リリース
Beta 2012	2012 年 6 月 8 日	NB への文書編集済みバージョンのリリース
Beta 2012	2012 年 6 月 27 日	EWG および用語集コメントの反映
RC 2012	2012 年 8 月 15 日	リリース前バージョン – 最終 NB 編集箇所の反映
RC 2012	2012 年 9 月 2 日	BNLTB および Stuart Reid のコメントの反映 Paul Jorgensen のクロスチェック
GA 2012	2012 年 10 月 19 日	GA リリースのための最終編集

◆ JSTQB®

バージョン	日付	摘要
Version 2012.J01	2018 年 3 月 5 日	ISTQB Advanced Level Syllabus Technical Test Analyst Version 2012 の日本語翻訳版
Version 2012.J02	2018 年 3 月 15 日	・「3.2.4 コールグラフ」の節において、索引設定不備によるエラー表現に対する修正 ・「8. 索引」の章において、見出し不備を修正

目次

謝辞

このドキュメントは、International Software Testing Qualifications Board Advanced Level Sub Working Group - Technical Test Analyst(Technical Test Analyst 作業部会)のコアメンバである Graham Bath（Chair）、Paul Jorgensen、Jamie Mitchell が執筆した。

本コアチームは、レビューチームおよびすべての国の国際部会に所属するメンバによる提案と意見に感謝したい。

本 Advanced Level シラバス完成時の、Advanced Level Working Group（Advanced Level 作業部会）のメンバは以下のとおりである（アルファベット順）。

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (Vice Chair), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (Chair), Geoff Thompson, Erik van Veenendaal, Tsuyoshi Yumoto.

次のメンバが、本資料のレビュー、意見表明および投票に参加した。

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng, Shaomin Zhu.

このドキュメントは、2012 年 10 月 19 日に開催された ISTQB®の総会で正式に発行された。

0. 本シラバスの紹介

0.1 本書の目的

本シラバスは、国際ソフトウェアテスト資格 Advanced Level テクニカルテストアナリスト向けのベースとなる。ISTQB®は、本シラバスを次の趣旨で提供する。

1. 各国の委員会に対し、各国語への翻訳および教育機関の認定の目的で提供する。各国の委員会は、本シラバスを各言語の必要性に合わせて調整し、出版事情に合わせてリファレンスを修正することができる。
2. 試験委員会に対し、各シラバスの学習の目的に合わせ、各国語で試験問題を作成する目的で提供する。
3. 教育機関に対し、コースウェアを作成し、適切な教育方法を確定できるようにする目的で提供する。
4. 受験志願者に対し、試験準備（研修コースの一部、または独立した形）の目的で提供する。
5. 国際的なソフトウェアおよびシステムエンジニアリングのコミュニティに対し、ソフトウェアやシステムをテストする技能の向上を目的とする他、書籍や記事を執筆する際の参考として提供する。

ISTQB®では、事前に書面による申請があった場合に限り、第三者がこのシラバスを先に定めた以外の目的での使用を許諾することがある。

0.2 概要

Advanced Level は、次の 3 つの独立したシラバスで構成している。

- テストマネージャ
- テストアナリスト
- テクニカルテストアナリスト

『Advanced Level シラバス概要』[ISTQB_AL_OVIEW]には、次の情報を記載している。

- 各シラバスのビジネス成果
- 各シラバスの概要
- 各シラバスの関係
- 知識レベルの説明（K レベル）
- 付録

0.3 試験のための学習の目的

学習の目的はビジネス成果を支援し、Advanced Level テクニカルテストアナリスト認定を取得するための試験問題作成を行うために使用する。基本的に本シラバスのすべての箇所は K1 レベルで試験対象となる。つまり、受験志願者は、用語や概念について認識し、記憶し、想起することになる。このため、関連する章の最初には、K2、K3、K4 レベルの学習の目的のみを記載する。

0.4 前提事項

テクニカルテストアナリストの学習の目的には、次の領域で基本的な経験があることを想定しているものがある。

- プログラミングの一般的な概念
- システムアーキテクチャの一般的な概念

1. リスクベースドテストにおけるテクニカルテストアナリストのタスク - 30 分

用語

プロダクトリスク、リスク分析、リスクアセスメント、リスク識別、リスクレベル、リスク軽減、リスクベースドテスト

「リスクベースドテストにおけるテクニカルテストアナリストのタスク」の学習の目的

1.3 リスクアセスメント

TTA-1.3.1 (K2)テクニカルテストアナリストが通常考慮すべき一般的なリスク要因を要約する。

共通的な学習の目的

次の学習の目的は、本章の複数の節で説明する内容に関連する。

TTA-1.x.1 (K2)テスト計画とテスト実行に対してリスクベースドアプローチを適用する際の、テクニカルテストアナリストの活動を要約する。

1.1 イントロダクション

テストマネージャは、リスクベースドテスト戦略の確立とマネジメントに全体的な責任を持つ。通常、リスクベースドアプローチを正しく適用するために、テストマネージャはテクニカルテストアナリストの関与を要求する。

テクニカルテストアナリストには特殊な技術的専門知識があるため、次のリスクベースドテストのタスクに積極的に関与する。

- リスク識別
- リスクアセスメント
- リスク軽減

これらのタスクは、プロダクトリスクが新たに見つかった場合、およびリスクの優先度を変更する場合に対処し、定期的にリスクのステータスを評価し共有するために、プロジェクトを通じて反復的に実行される。

テクニカルテストアナリストは、プロジェクトのテストマネージャが確立したリスクベースドテストのフレームワークの中で作業する。また、セキュリティ、システムの信頼性、性能に関するリスクといった、プロジェクト固有の技術的なリスクについての知識を提供する。

1.2 リスク識別

リスク識別プロセスでは、可能な限り幅広いステークホルダを召集することにより、重要なリスクを数多く検出することが可能になる。テクニカルテストアナリストは技術的な専門スキルを保有するので、プロダクトリスクが存在しそうな領域を判定するために、専門家へのインタビューの実施、同僚とのブレインストーミング、および現在と過去の経験の分析に非常に適している。また特に、テクニカルテストアナリストは、技術的なリスクがある領域を判定するために、同僚の技術者(例:開発者、アーキテクト、運用技術者)と密接に連携して作業する。

識別される可能性のあるリスクの例を次に示す。

- 性能リスク(例:高負荷状態では応答時間を達成できない)
- セキュリティリスク(例:セキュリティ攻撃による機密データの漏えい)
- 信頼性リスク(例:アプリケーションがサービスレベルアグリーメントで規定された可用性を満たせない)

特定のソフトウェア品質特性に関するリスク領域については、本シラバスの関連する章で説明する。

1.3 リスクアセスメント

リスク識別ができるだけ多くの関連するリスクを識別することであるのに対し、リスクアセスメントは、各リスクを分類し、各リスクの発生確率とその影響度合いを判定するために、識別されたリスクを調査することである。

リスクレベルを決定する場合、通常はリスクアイテムごとに、リスク顕在化の発生確率および顕在化した際の影響度合いを評価する。発生確率は、潜在的な問題がテスト対象のシステムに存在する可能性と考えることができる。

テストアナリストが、発生し得る問題に潜在するビジネスへの影響度合いの把握に貢献するのに対し、テクニカルテストアナリストは、各リスクアイテムに潜在する技術的なリスクの検出と把握に貢献する。

通常、考慮すべき一般的な要因には、次のものがある。

- 技術の複雑度
- コード構造の複雑度

- 技術的要件に関するステークホルダ間の対立
- 開発組織が地理的に分散していることによるコミュニケーションの問題
- ツールと技術
- 時間、リソース、およびマネジメント層からのプレッシャー
- 早期からの品質保証活動の欠如
- 技術的要件の頻繁な変更
- 技術的品質特性に関連する欠陥の大量検出
- インターフェースと統合の技術的な問題

入手可能なリスク情報に基づき、テクニカルテストアナリストは、テストマネージャが確立したガイドラインに従い、リスクレベルを確定する。たとえば、テストマネージャは、1～10 の値(1 が最もリスクが高い)に分類してリスクを決定することがある。

1.4 リスク軽減

プロジェクトの期間中、テクニカルテストアナリストは、識別したリスクに対するテストの方法を検討する際に関与する。これには一般的に次のことを含む。

- テスト戦略およびテスト計画に従い、リスクを軽減するために、最も重要なテストを実行し、適切な軽減活動とコンティンジェンシー活動を実施する。
- プロジェクト活動を行う中で明らかになった追加情報に基づいてリスクを評価し、以前識別し分析したリスクの発生確率、または影響度合いを軽減するためにその情報を用いる。

2. 構造ベースドテスト - 225 分

用語

不可分条件、条件テスト、制御フローテスト、判定条件テスト、複合条件テスト、パステスト、短絡評価、ステートメントテスト、構造ベースの技法

「構造ベースドテスト」の学習の目的

2.2 条件テスト

TTA-2.2.1 （K2）条件カバレッジを達成する方法、および条件カバレッジがデシジョンカバレッジよりも厳格なテストではない場合がある理由を理解する。

2.3 判定条件テスト

TTA-2.3.1 （K3）定義されたカバレッジの度合いを達成するために、判定条件テストのテスト設計技法を適用して、テストケースを記述する。

2.4 改良条件判定カバレッジ（MC／DC）テスト

TTA-2.4.1 （K3）定義されたカバレッジの度合いを達成するために、改良条件判定カバレッジ（MC／DC）テストのテスト設計技法を適用して、テストケースを記述する。

2.5 複合条件テスト

TTA-2.5.1 （K3）定義されたカバレッジの度合いを達成するために、複合条件テストのテスト設計技法を適用して、テストケースを記述する。

2.6 パステスト

TTA-2.6.1 （K3）パステストのテスト設計技法を適用して、テストケースを記述する。

2.7 API テスト

TTA-2.7.1 （K2）API テストの適用と、このテストにより検出される欠陥の種類を理解する。

2.8 構造ベースの技法の選択

TTA-2.8.1 （K4）所定のプロジェクト状況に応じて、適切な構造ベースの技法を選択する。

2.1 イントロダクション

この章では主に、構造ベースのテスト設計技法について説明する。この技法は、ホワイトボックス技法またはコードベースのテスト技法とも呼ぶ。これらの技法では、テスト設計のベースとして、コード、データ、アーキテクチャ、およびシステムフローを使用する。それぞれの技法により、体系的にテストケースを導き出し、構造の特定の側面に焦点を絞ることができる。これらの技法は、計測すべきカバレッジ基準を提供する。このカバレッジ基準は、各プロジェクトや組織で定義された目的に紐づく必要がある。100%のカバレッジの達成は、すべてのテストを十分に行ったということを意味するのではなく、テスト設計している構造に対して、使用している技法ではこれ以上有用なテストを提示できないことを意味する。

本シラバスで取り上げる構造ベースのテスト設計技法は、条件カバレッジを除いて、Foundation Level シラバス[ISTQB_FL_SYL]で説明したステートメントカバレッジ技法およびデシジョンカバレッジ技法よりも厳格になる。

本シラバスでは、次の技法を取り上げる。

- 条件テスト
- 判定条件テスト
- 改良条件判定カバレッジ(MC／DC)テスト
- 複合条件テスト
- パステスト
- API テスト

上記に示す最初の 4 つの技法は、判定述語に基づき、同じ種類の欠陥をまんべんなく検出する。判定述語は、いかに複雑であっても、真または偽のいずれかに判定され、判定されたコードが実行されて、判定されないコードは実行されない。欠陥が検出されるのは判定述語が複雑で期待通りの評価がされないために、意図したパスを通らない場合である。

一般的に、最初の 4 つの技法は、上から下へ順に厳格さが増す。下の技法ほど、目標のカバレッジを達成し、より検出しづらい欠陥を検出するために、より多くのテストを定義する必要がある。

[Bath08]、[Beizer90]、[Beizer95]、[Copeland03]、および[Koomen06]を参照されたい。

2.2 条件テスト

デシジョン(ブランチ)テストは、全体の判定をまとめて考え、個別のテストケースで真と偽の結果を評価するが、一方、条件テストでは、個々の判定がどのように行われたかを考慮する。各判定述語は、1つ以上の単純な「不可分」条件でなり、各条件は個別のブール値となる。これらを論理的に組み合わせて、判定の最終結果を決定する。この度合いのカバレッジを達成するために、各不可分条件は真偽の両方を取るようにテストケースを作成する。

適用
条件テストは、下記に記した制限／注意事項のため、その考え方に意義があるだけである。しかし、より高い度合いのカバレッジを達成するためには、条件テストを理解することが必要である。

制限／注意事項
判定において2つ以上の不可分条件が存在する場合、テスト設計時にテストデータを安易に選択すると、条件カバレッジは達成するが、デシジョンカバレッジを達成できない場合がある。
たとえば、判定述語「A かつ B」の場合を想定する。

	A	B	A and B
テスト 1	偽	真	偽
テスト 2	真	偽	偽

上記表の 2 つのテストを実行すれば 100%の条件カバレッジを達成できる。これらの 2 つのテストは 100%の条件カバレッジを達成するが、両方のケースとも述語は偽となるので、デシジョンカバレッジを達成することはできない。

判定が 1 つの不可分条件の場合、条件テストはデシジョンテストと同一である。

2.3 判定条件テスト

判定条件テストは、条件カバレッジ(上記参照)を達成する必要があるが、デシジョンカバレッジ(Foundation Level シラバス[ISTQB_FL_SYL]を参照)も満たす必要がある。不可分条件のテストデータ値を注意深く選択すれば、条件カバレッジの達成に必要なテストケースでこの度合いのカバレッジを達成できることがある。

次の例は、上記と同じ判定述語「A かつ B」をテストしている。異なるテスト値を選択することにより、同じ数のテストで判定条件カバレッジを達成できる。

	A	B	A and B
テスト 1	真	真	真
テスト 2	偽	偽	偽

従って、この技法により、効率的になる場合がある。

適用
この度合いのカバレッジは、テストするシステムが重要ではあるが(セーフティ)クリティカルとまではいえない場合に、考慮する必要がある。

制限／注意事項
デシジョンテストよりも多くのテストケースが必要になる場合があるので、時間が重要な場合は、この技法は適さないことがある。

2.4 改良条件判定カバレッジ(MC／DC)テスト

この技法は、より強力な度合いの制御フローカバレッジを提供する。N 個の一意な不可分条件を想定した場合、MC／DC は通常、N+1 個のテストケースで実現できる。MC／DC は判定条件カバレッジを達成するが、次の要件も満たす必要がある。

1. 不可分条件 X が真になると判定結果が変わる 1 つ以上のテスト
2. 不可分条件 X が偽になると判定結果が変わる 1 つ以上のテスト
3. 各不可分条件に、上記の要件 1 と要件 2 を満たすテストが存在する。

	A	B	C	(A or B) and C
テスト 1	真	偽	真	真
テスト 2	偽	真	真	真
テスト 3	偽	偽	真	偽
テスト 4	真	偽	偽	偽

上記の例は、デシジョンカバレッジ(判定述語の結果は真と偽の両方が存在する)、および条件カバレッジ(A、B、C のいずれも真と偽の両方で評価する)を達成している。

テスト 1 では、A が真であるため、結果は真になる確認をしている。A を偽に変えた場合、結果が偽に変わる(テスト 3 で、A 以外の値を変えないことで確認している)。
テスト 2 では、B が真であるため、結果は真になる確認をしている。B を偽に変えた場合、結果が偽に変わる(テスト 3 で、B 以外の値を変えないことで確認している)。
テスト 1 では、C が真であるため、結果は真になる確認をしている。C を偽に変えた場合、結果が偽に変わる(テスト 4 で、C 以外の値を変えないことで確認している)。

適用
この技法は、航空宇宙ソフトウェア業界、およびその他多くのセーフティクリティカルシステムで広く使用されている。どのような故障でも大事故を引き起こすかもしれないセーフティクリティカルなソフトウェアに対処する場合に、この技法を使用すべきである。

制限／注意事項
1つの式の中に同一の項が複数存在すると、MC／DC カバレッジの達成が困難になる場合がある。この場合、この項は「結合」しているという。コード内の判定ステートメントによっては、結合している項のみの値を変えても判定結果を変えることができない場合がある。この問題に対処する 1 つのアプローチは、結合していない不可分条件のみが MC／DC レベルとなるようにテストを決めることである。別のアプローチは、ケースバイケースで結合が発生する各判定を解析することである。

一部のプログラミング言語やインタプリタは、コード内の複雑な判定ステートメントを評価する際に、短絡評価の振る舞いを示すように設計されている。つまり、式の一部を評価すれば式の最終結果を判定できる場合は、コードの実行時に式全体を評価しない場合がある。たとえば、判定「A かつ B」を評価する場合は、A が偽であれば、B は評価する必要はない。B の値では最終値が変更することがないため、B を評価しないことで実行時間を節約できる。必要な一部のテストを実行できないために、短絡評価は MC／DC カバレッジの達成に影響を与えることがある。

2.5 複合条件テスト

判定に含まれる値のすべての可能な組み合わせをテストすることが必要な場合が、まれにある。この網羅的な度合いのテストを、複合条件カバレッジと呼ぶ。必要なテスト数は、判定ステートメントの不可分条件の数に依存し、2^n である。ここで。n は結合していない不可分条件の数である。これまでと同じ例を使用すると、複合条件カバレッジを達成するには、次のテストが必要になる。

	A	B	C	(A or B) and C

テスト 1	真	真	真	真
テスト 2	真	真	偽	偽
テスト 3	真	偽	真	真
テスト 4	真	偽	偽	偽
テスト 5	偽	真	真	真
テスト 6	偽	真	偽	偽
テスト 7	偽	偽	真	偽
テスト 8	偽	偽	偽	偽

言語が短絡評価を使用すると、実際のテストケースの数は、不可分条件で実行される論理演算の順序とグループ化によって、減少することが多い。

適用

従来、この技法は、長期間クラッシュすることなく高い信頼性で稼働することが期待される組み込みソフトウェアのテストに使用されていた(30 年間の稼働が期待される電話交換機など)。この種類のテストは、ほとんどの(セーフティ)クリティカルなアプリケーションで MC／DC テストに置き換えることができる。

制限／注意事項

テストケースの数は、すべての不可分条件を含む真理値表から直接導けるので、このレベルのカバレッジは容易に確定できる。ただし、複合条件テストでは非常に多くのテストケースを必要とするので、ほとんどの状況で、テストケースの数を少なくできる MC／DC カバレッジを適用する。

2.6 パステスト

パステストは、コードを通過するパスを識別し、次にそれらをカバーするテストケースを作成する。概念的には、システム中のあらゆる一意なパスをテストするために役立つ。しかし、通常のシステムでは、ループ構造の性質上、テストケースの数が膨大になる可能性がある。

しかし、不定ループの問題は別にして、パステストの一部を実行するのが現実的である。この技法を適用する場合、[Beizer90]はモジュールの入り口から出口まで通る多くのパスを実行するテストを作成することを推奨している。また、複雑になる可能性があるタスクを単純化するために、次の手順を使用して体系的に単純化することを推奨する。

1. 最も単純で機能を実現する、開始から終了までのパスを選択する。
2. 前のパスとわずかに違うように1つのパスを追加する。後続の各テストで異なるようにパス中の1つのブランチだけを変えてみる。できる限り、長いパスよりも短いパスを採用する。機能を実現しないパスではなく実現するパスを採用する。
3. カバレッジに必要な場合のみ、機能を実現しないパスを選択する。Beizer はこのルールの中で、そのようなパスは本質的でないことがあるので疑う必要がある、と述べている。
4. 最も実行される可能性が高いパスを選択する際は、直感を使う。

この戦略を使用すると、一部のパスセグメントが複数回実行される可能性があることに注意する。この戦略のキーポイントは、コード中の考えられるあらゆるブランチを少なくとも 1 回、できれば複数回テストすることである。

適用
上記で定義した部分的なパステストは、しばしばセーフティクリティカルなソフトウェアで実行される。このテストは、判定を行う方法だけでなく、ソフトウェア全体を通じたパスを考察するので、本章で説明している他の方法と共に使用することが望ましい。

制限／注意事項
このテストは、パスを判定するために制御フローグラフを使用できるが、実際には、複雑なモジュールのパスを計算するためにツールが必要になる。

カバレッジ
すべてのパス(ループは除く)をカバーする十分なテストケースを作成すれば、ステートメントカバレッジとブランチカバレッジの両方を達成できる。パステストは、比較的少ないテスト数の増加により、ブランチカバレッジよりも厳格なテストになる[NIST 96]。

2.7 API テスト

アプリケーションプログラミングインターフェース(API)は、異なるプロセス、プログラム、およびシステムの間の通信を可能にするコードである。多くの場合、API は、あるプロセスが何らかの機能を別のプロセスに提供する、クライアント／サーバの関係で使用する。

ある点では、API テストは、グラフィカルユーザインターフェース(GUI)のテストにかなり似ている。入力値と返されるデータの評価に焦点を当てている。

API を対象とする際は、多くの場合、否定テストが不可欠である。API を使用して自分のコード外のサービスにアクセスするプログラマは、意図しない方法で API を利用しようとする場合がある。つまり、誤った操作を避けるために、堅牢なエラー処理が不可欠である。多くの場合、API は他の API と一緒に使用され、1 つのインターフェースが複数のパラメータを含み、それらの値がさまざまな方法で組み合わされることがあるので、多くの異なるインターフェースの組み合わせテストが必要になる場合がある。

API はしばしば疎結合しているので、トランザクションの失敗やタイミングによる故障が発生する可能性が高い。このため、復旧とリトライのメカニズムを徹底的にテストする必要がある。API を提供する組織は、すべてのサービスが非常に高い可用性を持っていることを確認しなければならない。そのためには多くの場合、インフラストラクチャのサポートに加えて、API 提供者による厳格な信頼性テストが必要になる。

適用
現在では、多くのシステムが分散化、すなわちある作業を他のプロセッサに処理させる方法としてリモート処理を使うほど、API テストがより重要になっている。その例としては、オペレーティングシステムコール、サービス指向アーキテクチャ(SOA)、リモートプロシージャコール(RPC)、Web サービス、および事実上その他すべての分散アプリケーションがある。API テストは、特にシステムオブシステムズのテストに適している。

制限／注意事項
API を直接テストするには、通常テクニカルテストアナリストは、専用のツールを使用する必要がある。一般的に、API と関連する直接のグラフィカルインターフェースは存在しないので、初期環境の設定、データのマーシャリング、API の呼び出し、および結果の判定を行うために、ツールが必要になる。

カバレッジ
API テストは、テストの種類を示すもので、特定度合いのカバレッジを示すものではない。また、最低限の API テストでも、すべての有効値と妥当な無効値を使って、API に対するすべての呼び出しを実行する必要がある。

欠陥の種類
API テストにより検出可能な欠陥の種類は、非常に多様である。検出される欠陥は一般的に、インターフェースの問題の他、データ処理の問題、タイミングの問題、トランザクションの失敗、およびトランザクションの重複である。

2.8 構造ベースの技法の選択

テスト対象のシステムの背景により、達成すべき構造ベースドテストのカバレッジ度合いが決まる。システムがクリティカルであるほど、必要なカバレッジの度合いが高くなる。一般的に、必要なカバレッジの度合いが高いほど、その度合いを達成するために、より多くの時間とリソースが必要になる。

必要なカバレッジの度合いが、ソフトウェアシステムに適用可能な標準により決定する場合もある。たとえば、ソフトウェアが航空機搭載用であれば、標準 DO-178B(ヨーロッパでは ED-12B)に従う必要がある。この標準は、次の 5 つの故障条件を含む。

A. Catastrophic:故障が飛行機の安全な飛行または着陸に必要な重要な機能を損なう原因になり得る。
B. Hazardous:故障が安全性または性能に重大な悪影響を与えることがある。
C. Major: 故障は重大であるが、A または B よりは深刻でない。
D. Minor:故障は注目すべきであるが、C よりも影響が小さい。
E. No effect: 故障は安全性に影響しない。

ソフトウェアシステムをレベル A に分類した場合、MC／DC カバレッジでテストする必要がある。レベル B の場合は、MC／DC もオプションであるが、デシジョンカバレッジのレベルでテストしなければならない。レベル C では、最低でもステートメントカバレッジが必要になる。

同様に、IEC-61508 はプログラマブル電子安全関連システムの機能安全に関する国際標準である。自動車、鉄道、製造工程、原子力発電所、機械装置など、多くの異なる領域で、この標準を採用している。致命度は段階的な安全性完全性レベル(SIL:Safty Integrity Level)を使って定義し(重要性は 1 が最も低く、4 が最も高い)、カバレッジは次のように推奨している。

1. ステートメントカバレッジとブランチカバレッジを推奨
2. ステートメントカバレッジを強く推奨し、ブランチカバレッジを推奨
3. ステートメントカバレッジとブランチカバレッジを強く推奨
4. MC／DC を強く推奨

現代のシステムでは、すべての処理を 1 つのシステムで実行することはまれである。API テストは、処理の一部をリモートで実行する際に、常に実施する必要がある。システムの重要性により、API テストで投入すべき作業量を決定しなければならない。

テクニカルテストアナリストは常に、テスト対象のソフトウェアシステムの背景を考えて、テストで使用する方法を決定する必要がある。

3. 分析技法 - 255 分

用語

制御フロー解析、サイクロマティック複雑度、データフロー解析、定義使用ペア、動的解析、メモリリーク、ペアワイズ統合テスト、近隣統合テスト、静的解析、ワイルドポインタ

「分析技法」の学習の目的

3.2 静的解析

TTA-3.2.1 (K3)制御フロー解析を使用し、コードに存在する制御フローの不正を検出する。
TTA-3.2.2 (K3)データフロー解析を使用し、コードに存在するデータフローの不正を検出する。
TTA-3.2.3 (K3)静的解析を適用し、コードの保守性を改善する方法を提案する。
TTA-3.2.4 (K2)統合テスト戦略を確立するために、コールグラフの用途を説明する。

3.3 動的解析

TTA-3.3.1 (K3)動的解析を使用して、達成する目標を定める。

3.1 イントロダクション

解析には、静的解析と動的解析の 2 種類がある。

静的解析（3.2 節）には、ソフトウェアを実行せずに行える分析的テストが含まれる。ソフトウェアを実行しないので、実行した時に正しく処理するかどうかの判定は、ツールまたは人間が行う。このソフトウェアの静的な視点により、実行するシナリオの基になるデータと事前条件を作成することなく、詳細な解析が可能になる。

テクニカルテストアナリストに関連するレビューのさまざまな形式については、第 5 章で説明する。

動的解析（3.3 節）は、コードを実際に実行する必要があり、コードを実行した方がより容易に検出できるコード中の欠陥（メモリリークなど）をみつけるために使用する。動的解析は、静的解析と同様にツールを必要とする場合もあれば、急速な使用メモリの増加のような指標を観察するなど、稼働中のシステムを監視する人を必要とする場合もある。

3.2 静的解析

静的解析の目的は、コードまたはシステムアーキテクチャ中の顕在する欠陥あるいは潜在的な欠陥を検出することと、保守性を改善することである。一般的に、静的解析はツールの支援が必要である。

3.2.1 制御フロー解析

制御フロー解析は、プログラム全体の制御フローを制御フローグラフまたはツールを使用して解析する静的技法である。この技法を使用してシステムで検出できる不正には、誤って設計したループ（開始点が複数存在するなど）、特定の言語(Scheme など)における関数呼び出しの曖昧な対象、誤った処理順序など、多数存在する。

制御フロー解析の最も一般的な用途は、サイクロマティック複雑度の決定である。サイクロマティック複雑度の値は、強連結グラフにおいて、独立したパスの数を表す正の整数である。このグラフではループおよび反復は、一度通過したら無視される。それぞれの独立したパスは、開始から終了まで、モジュールを通して一意なパスであることを表している。それぞれの一意なパスはテストする必要がある。

一般的に、サイクロマティック複雑度の値は、モジュール全体の複雑度を理解するために使用する。Thomas McCabe の理論[McCabe 76] によれば、システムが複雑なほど保守するのが難しく、より多くの欠陥を含む。長年にわたる多くの研究は、複雑度と含まれる欠陥の数との間に、このような相関関係があることを示している。NIST（米国国立標準技術研究所）は、複雑度値が 10 以下であることを推奨している。これより高い複雑度が測定されたすべてのモジュールは、複数のモジュールに分割することが必要になる場合がある。

3.2.2 データフロー解析

データフロー解析は、システムにおける変数の使用に関する情報を収集する、さまざまな技法をカバーしている。変数のライフサイクル（つまり、変数の宣言、定義、読み込み、評価、および破棄）を監視するのは、これらの操作の中で不正が発生する可能性があるからである。

定義使用表記法(define-use notation)と呼ばれる一般的な技法の 1 つでは、各変数のライフサイクルを次の 3 つの異なる不可分なアクションに分割する。

- d: 変数を宣言、定義、または初期化する。
- u:計算または判定述語で、変数を使用すなわち読み込む。
- k: 変数を削除、破棄する、または変数がスコープ外になる。

これら 3 つの不可分なアクションをペア(定義使用ペア)として組み合わせ、データフローを説明する。たとえば、「du パス」は、データ変数を定義した後で使用するコードの一部を示す。

データ不正が起こり得るのは、変数に対する正しいアクションを誤ったタイミングで実行するか、または変数内のデータに対して誤ったアクションを実行するためである。このような不正には、次のものがある。

- 無効な値を変数に割り当てる。
- 変数に値を割り当てずに、その変数を使用する。
- 制御述語中の誤った値により、誤ったパスを実行する。
- 変数を破棄した後で、その変数を使用する。
- スコープ外の変数を参照する。
- 使用しない変数を宣言および破棄する。
- 使用する前に変数を再定義する。
- 動的に割り当てた変数を解放しない(メモリリークが発生する可能性がある)。
- 変数を変更した結果、予想外の副作用が発生する(グローバル変数のあらゆる使用を考慮せずにその変数を変更した場合の波及効果など)。

使用する開発言語が、データフロー解析で使用するルールをガイドする場合がある。不正ではない変数を用いて特定のアクションを実行するプログラムが、特定の状況下でシステムがプログラマの期待とは異なる振る舞いを引き起こす場合がある。たとえば、特定のパスを実行すると、実際には使用しない変数を二度定義する、などである。データフロー解析では、多くの場合、この使用方法を「疑わしい」と分類する。これは、変数割り当て機能の適正な使用である場合もあるが、後に、このコードの保守性に問題が発生する可能性もある。

データフローテストは、「制御フローグラフを使用して、データに起こりうる不合理なことを探る」[Beizer90]ので、制御フローテストとは異なる欠陥を検出する。これらの欠陥の多くは、動的テストの実行では検出が困難な断続的な故障を引き起こすので、テクニカルテストアナリストはテストを計画する際にこの技法を採用する必要がある。

ただし、データフロー解析は静的技法なので、システムの実行時にデータに起きるいくつかの問題を見逃す場合がある。たとえば、本来実行時に動的に作成されるまで存在しない配列へのポインタを、静的データ変数が、すでに定義してしまっている場合がある。また、マルチプロセッサの使用とプリエンプティブマルチタスクが、データフロー解析または制御フロー解析では検出できない競合状態を作り出す場合がある。

3.2.3 保守性を改善するための静的解析の使用

静的解析は、コード、アーキテクチャ、および Web サイトの保守性を改善する多くの方法で適用できる。

コメントがなく構造化されていない、分かりにくいコードは、保守が難しい場合が多い。コード内の欠陥を探して解析するために、開発者はより多くの労力を必要とし、欠陥を修正するあるいは新機能を追加するためにコードを修正すると、さらに欠陥を埋め込む可能性がある。

ツール利用した静的解析を行って、コーディングの標準とガイドラインに対する標準適合性を確認し、コードの保守性を改善できる。これらの標準とガイドラインは、命名規則、コメント、インデント、コードのモジュール化など、必要なコーディングプラクティスを記述している。一般的に、静的解析ツールは、コードが構文的に正しい場合でも、エラーではなく警告が通知される。

一般的にモジュール化設計はコードの保守を容易にする。静的解析ツールは、次の方法でコードのモジュール化を支援する。

- 重複しているコードを検索する。コードのこれらの部分は、リファクタリングしモジュール化するための候補になる場合がある(ただし、リアルタイムシステムでは、モジュール呼び出しで生じる実行時オーバーヘッドが問題になることがある)。
- コードのモジュール化の有用な指標となるメトリクスを生成する。これらは、結合度と凝集度の測定値を含んでいる。保守しやすいシステムは、結合度(モジュールが実行時に互いに依存する度合い)が低く、凝集度(モジュールが自己完結し、単一のタスクに集中している度合い)が高い傾向がある。
- オブジェクト指向コードで、派生オブジェクトにおける親クラスへの可視性の過不足がどこにあるかを示してくれる。
- コードあるいはアーキテクチャにおいて、構造の複雑度が高い領域を強調表示する。この複雑度が高いことは保守性が低いことと欠陥を含む可能性が高いことを示している。保守性と欠陥予防を考えてコードのモジュール化を確実に実施するために、ガイドラインでサイクロマティック複雑度(3.2.1 節を参照)の受容レベルを規定している場合がある。大きなサイクロマティック複雑度を持つコードは、モジュール化の候補になり得る。

また、Web サイトの保守は、静的解析ツールを使用してサポートすることもできる。この目的は、サイトのツリー構造のバランスが良いかどうか、またはバランスの悪さが存在するかどうかをチェックすることである。バランスの悪さは以下のようなことを引き起こす原因となる。

- テストがより困難になる。
- 保守の作業負荷が増加する。
- ユーザのナビゲーションが困難になる。

3.2.4 コールグラフ

コールグラフは、通信の複雑度を静的に表現したものである。これは有向グラフで、ノードはプログラムユニットを示し、エッジはユニット間の通信を示している。

コールグラフは、異なる関数またはメソッドが互いを呼び出すユニットテスト、個別のモジュールが互いを呼び出す統合テストとシステムテスト、または個別のシステムが互いを呼び出すシステム統合テストで使用する場合がある。

このグラフは、次の目的で使用できる。

- 特定のモジュールまたはシステムを呼び出すテストを設計する。
- ソフトウェア内でモジュールまたはシステムの呼び出し箇所の数を確認する。
- コードおよびシステムアーキテクチャの構造を評価する。
- 統合の順序を提案する(ペアワイズ統合と近隣統合)。これについては、以降で詳細に説明する。

Foundation Level シラバス[ISTQB_FL_SYL]では、統合テストの 2 つの異なるカテゴリであるインクリメンタル(トップダウン、ボトムアップなど)と非インクリメンタル(ビッグバン)について説明している。インクリメンタル方式が好ましいと説明され、その理由はコードを徐々に増やすために、関連するコード量が限定されるので欠陥の分離が容易になるからである。

本 Advanced Level シラバスでは、コールグラフを使用するさらに 3 つの非インクリメンタル方式を紹介する。これらは、テストを完了するために追加のビルドを必要とし、テストをサポートするために書く(プロダクトには実装されない)コードを必要とする可能性があるインクリメンタル方式よりも望ましい場合がある。これらの 3 つの方法には、次のものがある。

- ペアワイズ統合テスト(ブラックボックステスト技法の「ペアワイズテスト」と混同しないこと)は、統合テストのコールグラフで示されるような連動しているコンポーネントペアを対象にする。この方法で減らせるビルドの数は少ないが、必要なテストハーネスコードの量を減らせる。

- 近隣統合テストは、統合テストのベースである所定のノードに接続しているすべてのノードをテストする。コールグラフ内の特定ノードの全先行ノードと後続ノードは、テストのベースになる。
- McCabe の設計述語アプローチは、モジュールのコールグラフに適用する際に、サイクロマティック複雑度の理論を使用する。このため、モジュールが互いを呼び出す次のようなさまざまな方法を示すコールグラフを作成する必要がある。
    - 無条件呼び出し: あるモジュールから別のモジュールの呼び出しが常に発生する。
    - 条件付き呼び出し: あるモジュールから別のモジュールの呼び出しが場合によって発生する。
    - 相互排他的条件付き呼び出し: あるモジュールがいくつかの異なるモジュール中の 1 つのみを呼び出す。
    - 反復呼び出し: あるモジュールが別のモジュールを1回以上呼び出す。
    - 条件付き反復呼び出し: あるモジュールが別のモジュールを 0 回以上呼び出す。

  コールグラフの作成後、統合の複雑度を計算し、そのグラフをカバーするテストを作成する。

コールグラフとペアワイズ統合テストの使用に関する詳細は、[Jorgensen07]を参照されたい。

3.3 動的解析

3.3.1 概要

動的解析は、すぐには兆候が現れないことのある故障を検出するために使用する。たとえば、メモリリークの可能性は静的解析で検出できる場合もあるが(メモリを割り当てているが解放していないコードをみつける)、動的解析を使用するとメモリリークは容易に判明する。

すぐに再現できない故障は、テスト作業、およびソフトウェアのリリースまたは本番使用に大きな影響を与える可能性がある。このような故障は、メモリリーク、ポインタの誤った使用、およびそのほかの破壊(システムスタックなど)により発生する場合がある[Kaner02]。システム性能の緩やかな低下や、システムクラッシュを含むこれら故障の性質上、テスト戦略ではこのような欠陥に関するリスクを考慮し、適切な動的解析(通常ツールを使用して)を実行して、リスクを軽減する必要がある。これらの故障を検出し修正するには多大なコストがかかる場合が多いので、プロジェクトの早期に動的解析を実行することを推奨する。

動的解析は、次のことを実現するために適用する場合がある。

- ワイルドポインタ、およびシステムメモリの損失を検出することにより、故障の発生を防止する。
- 容易に再現できないシステムの故障を解析する。
- ネットワークの振る舞いを評価する。
- 実行時のシステムの振る舞いに関する情報を提供することにより、システム性能を改善する。

動的解析はどのテストレベルでも実行可能であり、次のことを実行する技術的スキルとシステムのスキルが必要である。

- 動的解析のテスト目標を定める。
- 解析を開始して終了する適切な時間を決定する。
- 結果を分析する。

システムテストでは、テクニカルテストアナリストは最小限の技術的スキルを持っていれば、動的解析ツールを使用できる。通常、使用するツールは包括的なログを生成し、必要な技術的スキルを持つテクニカルテストアナリストはそれを解析できる。

3.3.2 メモリリークの検出

メモリリークは、プログラムが使用できるメモリ(RAM)領域を割り当て、その後不要になっても開放しない場合に発生する。このメモリ領域は割り当てられたままで再使用できない。この現象が頻繁に発生するか、またはメモリが少ない状況では、プログラムが使用可能なメモリを使い果たす可能性がある。以前は、メモリ操作はプログラマの責任であり、メモリリークを避けるために、動的に割り当てたいかなるメモリ領域も割り当てたプログラムが正しいスコープ内で開放しなければならなかった。現代の多くのプログラミング環境は、自動または半自動の「ガーベジコレクション」を持つので、割り当てたメモリはプログラマが直接操作しなくても開放される。既存の割り当てられたメモリを自動ガーベジコレクションが解放している場合、メモリリークの特定は非常に困難である。

メモリリークは、時間をかけて進行し、必ずしもすぐには明らかにならない問題を引き起こす。明らかにならない理由は、ソフトウェアをインストールした直後や、システムを再起動した後にテストを行うことが多いためである。これらの理由により、メモリリークの悪影響は、プログラムが本番環境で稼働して初めて気づく場合が多い。

メモリリークの兆候は、システムの応答時間が徐々に悪くなり、最終的にシステムの故障となる場合がある。このような故障は、システムの再起動(リブート)で解決できるが、この方法は必ずしも現実的ではなく、実行できない場合もある。

多くの動的解析ツールは、メモリリークを起こすコード領域を識別するのでコード修正が可能になる。また、単純なメモリモニタを使用して、時間と共に使用可能なメモリが減少しているかどうか、全体の様子を見ることができるが、この減少の正確な原因を特定するには、引き続き解析を行う必要がある。

また、その他にも考慮すべきリーク発生源がある。たとえば、ファイルハンドル、セマフォ、リソースの接続プールである。

3.3.3 ワイルドポインタの検出

プログラム内の「ワイルド」ポインタは、使用してはならないポインタである。たとえば、ワイルドポインタは、ポイントすべきオブジェクトまたは関数を「失って」いる場合、または意図したメモリ領域をポイントしていない場合がある(配列に割り当てられた領域の境界を越えてポイントしている、など)。プログラムがワイルドポインタを使用すると、次のようなさまざまな結果が発生することがある。

- プログラムが期待通りに動く。これは、ワイルドポインタが現在プログラムにより使用されていない、そして概念的には「解放」され、ないし有効な値を持つメモリにアクセスしているケースである。
- プログラムがクラッシュする。これは、ワイルドポインタが、プログラム(オペレーティングシステムなど)の実行に不可欠なメモリの一部が不正に使用される原因となっていた場合である。
- プログラムが必要なオブジェクトにアクセスできないために、正しく機能しない。このような状況で、エラーメッセージが発行されてもプログラムが動き続けることがある。
- ポインタによりメモリ内のデータが破壊され、その後不正な値が使用される。

プログラムのメモリの使い方をどう変更しても(ソフトウェア変更後の新規ビルドなど)、上記 4 つの結果のいずれかを引き起こす場合があることに注意する。このことは、ワイルドポインタを使用しても、最初はプログラムが期待通りに動き、ソフトウェアの変更後に予想外のクラッシュが発生する場合(本番環境でも発生する可能性がある)があるので、特に重要である。このような故障は多くの場合、潜在的な欠陥(つまり、ワイルドポインタ)の兆候であることに注意する必要がある([Kaner02]の「鉄則 74」を参照)。ワイルドポインタがプログラムで使用された際、プログラムの実行に対する影響に関係なく、ツールはワイルドポインタを識別する助けになる。一部のオペレーティングシステムは、実行時のメモリアクセス違反をチェックする機能を組み込んでいる。たとえば、オペレーティングシステムは、アプリケーションが許可されたメモリ領域にないメモリ位置にアクセスすると、例外を検知する。

3.3.4 性能の解析

動的解析は、故障の検出に有用なだけではない。ツールでプログラム性能の動的解析を行うと、性能のボトルネックを識別し、広範な性能メトリクスを生成する。開発者はそれを使用して、システムの性能を調整することができる。たとえば、実行中にモジュールが呼び出される回数に関する情報を提供することができる。頻繁に呼び出されるモジュールは、性能向上の候補になる可能性が高い。

ソフトウェアの動的振る舞いに関する情報と、静的解析（3.2.4 節を参照）でのコールグラフから得られる情報を合わせて、テスト担当者は詳細で広範なテストの候補となるモジュールを識別することができる（頻繁に呼び出され、多くのインターフェースを持つモジュールなど）。

プログラム性能の動的解析は多くの場合、システムテストで実行するが、テストハーネスを使用してテストの初期段階で、1つのサブシステムをテストする時に実行する場合もある。

4. テクニカルテストのための品質特性 - 405 分

用語

環境適応性、解析性、変更性、共存性、効率性、設置性、保守性テスト、成熟性、運用受け入れテスト、運用プロファイル、性能テスト、移植性テスト、回復性テスト、信頼度成長モデル、信頼性テスト、置換性、資源効率性テスト、頑健性(堅牢性)、セキュリティテスト、安定性、試験性

「テクニカルテストのための品質特性」の学習の目的

4.2 一般的な計画上の問題

TTA-4.2.1 (K4)特定のプロジェクトとテストするシステムに対して、非機能要件を分析し、テスト計画の各節を記述する。

4.3 セキュリティテスト

TTA-4.3.1 (K3)セキュリティテストのアプローチを定義し、高位レベルテストケースを設計する。

4.4 信頼性テスト

TTA-4.4.1 (K3)信頼性の品質特性とそれに対応する ISO 9126 の副特性のアプローチを定義し、高位レベルテストケースを設計する。

4.5 性能テスト

TTA-4.5.1 (K3)性能テストのアプローチを定義し、高位レベル運用プロファイルを設計する。

共通する学習の目的

次の学習の目的は、本章の複数の節で説明する内容に関連する。

TTA-4.x.1 (K2)テスト戦略ないしテストアプローチに保守性テスト、移植性テスト、および資源効率性テストを含める理由を理解し、説明する。

TTA-4.x.2 (K3)特定のプロダクトリスクがある場合、最も適切な特定の非機能のテストタイプ(複数の場合あり)を定義する。

TTA-4.x.3 (K2)アプリケーションのライフサイクルで、非機能テストを適用する必要がある段階を理解し、説明する。

TTA-4.x.4 (K3)特定のシナリオに対して、非機能のテストタイプの活用によって検出を期待する欠陥の種類を定義する。

4.1 イントロダクション

一般的に、テクニカルテストアナリストがテストで注目するのは「どのように」製品が動くかであり製品が「何」をするかという機能面ではない。これらのテストは、どのテストレベルでも実行可能である。たとえば、リアルタイムシステムと組み込みシステムのコンポーネントテストでは、性能ベンチマークの実施とリソース使用量のテストが重要である。システムテストと運用受け入れテスト(OAT)では、回復性などの信頼性面のテストが適している。このレベルのテストは、特定のシステム、つまりハードウェアとソフトウェアの組み合わせをテストすることが目的である。テストする具体的なシステムとしては、さまざまなサーバ、クライアント、データベース、ネットワーク、およびその他のリソースがある。テストレベルに関係なく、テストはリスクの優先度と使用可能なリソースに基づいて実行する必要がある。

プロダクトの特性を説明するためのガイドとして、ISO 9126 で示されているプロダクト品質特性の記述を使用する。ISO 25000 シリーズ(ISO 9126 の後継)などの他の標準を使用する場合もある。ISO 9126 品質特性は、副特性を持つ複数の特性に分かれている。次の表では、これらの特性／副特性およびテストアナリストシラバスとテクニカルテストアナリストシラバスがカバーする特性／副特性を示す。

特性	副特性	テストアナリスト	テクニカルテストアナリスト
機能性	正確性、合目的性、相互運用性、標準適合性	○	
	セキュリティ		○
信頼性	成熟性(頑健性)、障害許容性、回復性、標準適合性		○
使用性	理解性、習得性、運用性、魅力性、標準適合性	○	
効率性	性能(時間効率性)、資源効率性、標準適合性		○
保守性	解析性、変更性、安定性、試験性、標準適合性		○
移植性	環境適応性、設置性、共存性、置換性、標準適合性		○

この作業の割り当ては組織によって異なるかもしれないが ISTQB シラバスではこの割り当てに従っている。

各品質特性には標準適合性という副特性がある。安全を重視する環境、または規制された環境では、各品質特性が特定の標準と規制に準拠することが必要な場合がある。これらの標準は業界によって大きく異なるため、ここでは詳細に説明しない。テクニカルテストアナリストが、標準適合性要件の影響を受ける環境で作業する場合は、これらの要件を理解し、テストとテスト文書の両方が標準適合性要件を満たすことが重要である。

この節で説明するすべての品質特性と副特性に関して、適切なテスト戦略を立て文書化できるように、典型的なリスクを認識しなければならない。品質特性のテストでは、ライフサイクルのタイミング、必要なツール、ソフトウェアとドキュメントの入手可能状況、および技術的専門知識に特に注意が必要になる。各特性、およびその固有なテスト要求に対処する戦略を計画しないと、テスト担当者は、テストに十分な計画、準備そしてテスト実行時間をスケジュールに組み込めないことがある[Bath08]。性能テストなど一部の品質特性のテストでは、広範な計画、専用の装置、特定のツール、専門的なテストスキル、そしてほとんどの場合かなりの時間が必要になる。品質特性と副特性のテストでは、十分なリソースを作業に割り当て、テストスケジュール全体に統合する必要がある。これらの各側面は固有のニーズを持ち、側面ごとに特有の課題を持ち、この課題はソフトウェアライフサイクル中のさまざまな時点で発生することがある。このことについては、以降の節で説明する。

テストマネージャは、品質特性と副特性に関する要約した測定情報の編集とレポート作成に関わるが、テストアナリストまたはテクニカルテストアナリストは(上記の表に従って)、各測定情報を収集する。

テクニカルテストアナリストが本番前のテストで集めた品質特性の測定値は、ソフトウェアシステムの供給者とステークホルダ(顧客やオペレータなど)間のサービスレベルアグリーメント(SLA)の基になることがある。ある場合には、ソフトウェアが本番に入ってからも、しばしば別のチームや組織がテストを継続することがある。このようなことは通常、本番環境とテスト環境で異なる結果を示すことのある効率性テストと信頼性テストで起きる。

4.2 一般的な計画上の問題

非機能テストの計画を怠ると、アプリケーションの成功はかなり危うくなる。テストマネージャはテクニカルテストアナリストに、関連する品質特性(4.1 節の表を参照)の主要なリスクを識別し、提案するテストに関わる計画上の問題に対処するように求める場合がある。これらは、マスターテスト計画の作成時に使用される場合がある。また、これらのタスクを実行する際は、次の一般的な要因を考慮する。

- ステークホルダからの要件
- 必要なツールの調達とトレーニング
- テスト環境の要件
- 組織に関する考慮
- データセキュリティに関する考慮

4.2.1 ステークホルダからの要件

非機能要件は多くの場合、明記されず、存在しない場合もある。計画段階では、テクニカルテストアナリストは、影響のあるステークホルダからの技術的な品質特性に関する期待度合いを把握し、それらが示すリスクを評価する必要がある。

一般的なアプローチは、顧客がシステムの現行バージョンに満足しているなら、達成する品質レベルを維持する限り、新しいバージョンにも顧客が引き続き満足すると想定することである。このためシステムの現行バージョンをベンチマークとして使用できる。このアプローチは、ステークホルダが要件を明記するのが難しい、性能などの一部の非機能的な品質特性に採用すると特に有用である。

非機能要件を獲得する際は、多様な観点を獲得するのが得策である。これらの観点は、顧客、ユーザ、運用スタッフ、および保守スタッフなどのステークホルダから引き出す必要がある。これができない場合、いくつかの要件を見落とす可能性がある。

4.2.2 必要なツールの調達とトレーニング

市販のツールまたはシミュレータは、特に性能テストおよび特定のセキュリティテストに適している。テクニカルテストアナリストは、ツールの入手、学習、および実装に必要なコストと時間を見積る必要がある。また、専用のツールを使用する際は、計画時に、新規ツールの習得時間、および外部のツール専門家を雇うコストを考慮する必要がある。

複雑なシミュレータの開発は、それ自体で開発プロジェクトとなるため、計画が必要な場合がある。特に、開発するツールのテストと文書化を、スケジュールとリソース計画に含めなければならない。シミュレートする製品の変更に従って行うシミュレータのアップグレードと再テストに必要な予算と時間を十分計画に含めるべきである。セーフティクリティカルなアプリケーションで使用するシミュレータの計画では、独立した組織によるシミュレータの受け入れテストと、必要とされる認定の取得を考慮しなければならない。

4.2.3 テスト環境の要件

多くのテクニカルテスト(セキュリティテストや性能テストなど)では、現実的な測定値を得るために、本番に近いテスト環境が必要になる。このことは、テスト対象のシステムの規模と複雑度に応じて、テストの計画と予算に重大な影響を持つ可能性がある。このような環境のコストが高くなる場合、次のような代替案を検討する。

- 本番環境を使用する。
- システムの規模縮小版を使用する。この場合、得られるテスト結果が、本番システムをテストした場合のテスト結果を代替できるということを、十分に説明できるようにする。

このようなテストの実行タイミングは慎重に計画する必要があり、特定の時間(使用率が低い時間など)にしか実行できないことがよくある。

4.2.4 組織に関する考慮事項

テクニカルテストでは、全システム中のいくつかのコンポーネント(サーバ、データベース、ネットワークなど)の振る舞いを測定する場合がある。これらのコンポーネントが多くの異なる場所と組織に分散している場合は、テストの計画と調整にかなりの労力が必要になることがある。たとえば、特定のソフトウェアコンポーネントが 1 日の特定の時間、または 1 年の特定の時期にしかシステムテストに使用できない場合や、組織が限られた日数しかテストをサポートできない場合がある。システムコンポーネントと他の組織のスタッフ(つまり「一時的に得た」専門知識)がテストで「必要な時」に活用できることを確認しないと、予定したテストを中断せざるを得ないことがある。

4.2.5 データセキュリティの考慮

すべてのテスト活動を確実に実行するために、システムに実装する具体的なセキュリティ対策をテスト計画の段階で考慮する必要がある。たとえば、データの暗号化を行うと、テストデータの作成および結果の確認が困難になる場合がある。

データ保護のポリシーと法律により、本番データに基づく必要なテストデータを生成できない場合がある。テストデータの秘匿化は容易なタスクではないので、テスト実装の一部として計画する必要がある。

4.3 セキュリティテスト

4.3.1 イントロダクション

セキュリティテストは、次の 2 つの重要な点で、他の機能テストとは異なる。

1. 標準的な技法で選ばれたテスト用の入力データは、重要なセキュリティの問題を見逃す場合がある。
2. セキュリティの欠陥を示す症状は、他の機能テストで見つかる欠陥の症状とは大きく異なる。

セキュリティテストは、システムのセキュリティポリシーを侵害しようとする行為によって脅威に対するシステムの脆弱性を評価する。以下は、セキュリティテストで想定すべき潜在的な脅威の一覧である。

- アプリケーションまたはデータの不正なコピー。
- 不正なアクセス制御(権限を持たないユーザがタスクを実行できるなど)。ユーザの権限、アクセス権、および権限レベルが、このテストで焦点を当てる事柄である。この情報は、システム仕様書に書く必要がある。
- 目的の機能を実行した際に、意図しない副作用を示すソフトウェア。たとえば、メディアプレイヤーがオーディオを正しく再生しても、暗号化されない一時ストレージにファイルを書き出せば、それは悪意あるユーザが悪用できる副作用になる。

- 以降に利用するユーザにより実行される可能性があるコードの Web ページへの挿入(クロスサイトスクリプティング(XSS))。このコードに悪意のある場合がある。
- バッファオーバーフロー(バッファオーバーラン)。ユーザインターフェースの入力フィールドに、コードが正しく処理できるよりも長い文字列を入力することにより、発生する場合がある。バッファオーバーフローの脆弱性は、悪意のあるコードを実行する可能性があることを意味する。
- サービス拒否。ユーザがアプリケーションを操作できなくなる(「妨害」要求で Web サーバを過負荷にするなど)。
- 中間者攻撃。ユーザに存在を気づかれることなく、第三者が通信(クレジットカード決済情報など)の傍受、なりすまし、改ざん、およびその後の他者への引き渡しを行う。
- 機密データを保護するために使用する暗号化コードの解読。
- 論理爆弾(イースターエッグとも呼ぶ)。悪意を持ってコードに挿入され、特定の条件下でのみ(特定の日など)作動する。論理爆弾が作動すると、ファイルの削除やディスクのフォーマットなど、悪意ある行為が実行される場合がある。

4.3.2 セキュリティテストの計画

一般的に、セキュリティテストを計画する時は次の点に特に留意する。

- セキュリティの問題は、システムのアーキテクチャ、設計、および実装の段階で入り込む可能性があるので、セキュリティテストはユニットテスト、統合テスト、およびシステムテストのレベルでスケジュールされる場合がある。また、セキュリティの脅威は変化するものなので、システムが本番稼働した後でも、セキュリティテストを定期的にスケジュールする場合もある。
- テクニカルテストアナリストが提案するテスト戦略は、コードレビューとセキュリティツールを使用した静的解析を含むことがある。これらは、動的テストで見逃しやすい、アーキテクチャ、設計ドキュメント、およびコードに関するセキュリティの問題を検出するのに有効である。
- テクニカルテストアナリストは、注意深い計画とステークホルダとの調整を必要とする、セキュリティ「攻撃」(下記の「攻撃計画」を参照)の設計と実行を求められる場合がある。開発者またはテストアナリストと協力して、その他のセキュリティテスト(ユーザの権限、アクセス権、権限レベルのテストなど)を実行する場合がある。セキュリティテストの計画では、このような組織的問題を注意深く考慮しなければならない。
- セキュリティテスト計画の不可欠な側面に、承認の取得がある。これはテクニカルテストアナリストが計画したセキュリティテストを実行するために、テストマネージャから明示的な許可を得ることを意味する。計画にない追加のテストを実行すれば、実際の攻撃と見なされる可能性があり、このようなテストを実行した者は、法的措置を受けるリスクがある。目的と承認を示す文書無しに、「セキュリティテストを実施した」という弁解では説得性のある説明は難しいことがある。
- システムのセキュリティ改善を行うと、性能に影響する場合があることに注意する必要がある。セキュリティの改善後、性能テストを実施する必要性を検討することを推奨する(以降の 4.5 節を参照)。

4.3.3 セキュリティテストの仕様

特定のセキュリティテストは次のようなセキュリティリスクの発生源に従ってグループ化できる[Whittaker04]。

- ユーザインターフェース関連 -不正なアクセスと悪意のある入力。
- ファイルシステム関連 - ファイルまたはリポジトリに格納した機密データへのアクセス。
- オペレーティングシステム関連 - 悪意のある入力によりシステムがクラッシュした際に、漏えいする可能性がある機密情報のストレージ(メモリ内の暗号化していないパスワードなど)。
- 外部ソフトウェア関連 - システムが使用する外部コンポーネント間で発生する場合がある相互作用。これらは、ネットワークレベル(不正なパケットやメッセージの受信など)、あるいはソフトウェアコンポーネントレベルの場合もある(ソフトウェアが依存するソフトウェアコンポーネントの故障など)。

セキュリティテストを開発するために、次のアプローチ[Whittaker04]を使用する場合がある。

- テストを仕様化するのに有用な情報を収集する(従業員の名前、住所、内部ネットワークに関する詳細情報、IP アドレス、使用するソフトウェアまたはハードウェアの識別情報、オペレーティングシステムのバージョンなど)。
- 入手しやすいツールを使用して、脆弱性のスキャンを実行する。このようなツールは、直接システムに侵入するためでなく、セキュリティポリシーに違反するか、結果的に違反する可能性がある脆弱性を発見するために使用できる。また、NIST(米国国立標準技術研究所)[Web-2]などが提供するチェックリストを使用しても、特定の脆弱性を発見することができる。
- 収集した情報を使用して、「攻撃計画」(つまり、特定のシステムのセキュリティポリシーを破ることを意図したテスト計画)を作成する。攻撃計画では、最も深刻なセキュリティ上の欠陥を検出するために、さまざまなインターフェース(ユーザインターフェース、ファイルシステムなど)を経由するいくつかの入力を指定する必要がある。[Whittaker04] にあるさまざまな「攻撃」は、セキュリティテスト専用に開発した技法の有用な情報源である。

また、セキュリティの問題は、レビュー(第 5 章を参照)、および静的解析ツール(3.2 節を参照)を使用して発見することもできる。静的解析ツールは、セキュリティ脅威専用の広範なルールセットを持ち、このルールに対してコードをチェックする。たとえば、データを割り当てる前にバッファサイズをチェックしないことで発生するバッファオーバーフローの問題は、このツールで検出できる。

静的解析ツールを Web コードに使用すると、コードインジェクション、クッキーのセキュリティ、クロスサイトスクリプティング、リソース改ざん、SQL インジェクションなどのセキュリティの脆弱性を発見することができる。

4.4 信頼性テスト

プロダクトの品質特性の ISO 9126 による分類では、信頼性に次の副特性が定義されている

- 成熟性
- 障害許容性
- 回復性

4.4.1 ソフトウェア成熟性の測定

信頼性テストの目的は、ソフトウェア成熟性の統計的測定値を定常的に監視し、これをサービスレベルアグリーメント(SLA)として表されることのある期待する信頼性目標と比較することである。これらの測定値は、平均故障間隔(MTBF)、平均修復時間(MTTR)、またはその他の故障強度測定値(1 週間で発生した特定の重要度の故障数など)の形式を取る場合がある。これらは、終了基準(本番リリース用など)として使用される場合がある。

4.4.2 障害許容性のテスト

想定外の入力値の処理に関するソフトウェアの障害許容性を評価する機能テスト(否定テストとも呼ぶ)に加えて、テスト対象のアプリケーションの外部で発生する障害に対するシステムの許容性を評価するテストが必要である。このような障害は一般的に、オペレーティングシステムが通知する(ディスクフル、プロセスまたはサービスが使用できない、ファイルが見つからない、メモリが使用できない、など)。システムレベルでの障害許容性のテストを特定のツールがサポートする場合がある。

「頑健性(堅牢性)」および「エラー耐性」という用語は、一般的に障害許容性を説明する際にも使用されることに注意する(詳細は[ISTQB_GLOSSARY]を参照)。

4.4.3 回復性テスト

信頼性テストではさらにソフトウェアシステムがハードウェアまたはソフトウェアの故障から事前に決定した方法で回復し、正常な運用を再開できることを評価する。回復性テストは、フェイルオーバーテスト、およびバックアップ／リストアテストを含む。

フェイルオーバーテストを実行するのは、ソフトウェアの故障が大きな被害をもたらすので、故障が発生してもシステムの運用を確保するために、特定のソフトウェア／ハードウェア対策を実装する場合である。たとえば、フェイルオーバーテストを適用できるのは、財政的損失のリスクが非常に大きい場合、または重大な安全性の課題が存在する場合である。故障が大惨事によって発生する場合は、この種の回復性テストを「災害復旧」テストと呼ぶ場合もある。

一般的なハードウェア故障の予防対策としては、複数のプロセッサ間での負荷分散、および故障が発生した場合、即座に別の機器に処理を引き継ぐサーバ、プロセッサ、またはディスクのクラスタリング（冗長システム）がある。一般的なソフトウェア対策は、いわゆる冗長な異種システムにおける、ソフトウェアシステムの複数の独立したインスタンスの実装がありうる（航空管制システムなど）。冗長システムは一般的に、ソフトウェア対策およびハードウェア対策の組み合わせであり、独立したインスタンスの数（2、3、4）に応じて、それぞれ二重化、三重化、四重化システムと呼ぶ。ソフトウェアの異種性は、異なるソフトウェアで同じサービスを提供するという目的で、2つ（またはそれ以上）の独立した無関係の開発チームに、同じソフトウェア要件を与えて実現する。これにより、冗長な異種システムが同じ欠陥を持つ入力から同じ結果となる確率を下げる。システムの回復性を向上するこれらの対策は、信頼性にも直接影響する場合があるので、信頼性テストを実行する時に考慮する必要がある。

フェイルオーバーテストは、故障モードをシミュレートするか、または制御された環境で実際に故障を発生させることにより、システムを明示的にテストするように設計されている。フェイルオーバーメカニズムのテストは、データの損失や破損がなく、合意したサービスレベル（機能の可用性や応答時間など）が維持されていることについて故障をたどることで確認する。フェイルオーバーテストの詳細については、[Web-1]を参照されたい。

バックアップ／リストアテストは、故障の影響を最小化するための手続き的対策の確立に焦点を当てる。このようなテストでは、さまざまな形式のバックアップを取り、データの損失や破損が発生した場合に、そのデータをリストアするための手順（通常はマニュアルに記述する）を評価する。テストケースは、各手順のクリティカルパスをカバーするように設計する。テクニカルレビューで、これらのシナリオを「試運転」し、実際の手順と比較してマニュアルを確認する場合がある。運用受け入れテスト（OAT）では、本番環境または本番に近い環境でシナリオを実行して、実際に使えることを確認する。

バックアップ／リストアテストの測定値には、次のようなものがある。

- 異なる種類のバックアップ（完全バックアップ、差分バックアップなど）の実行にかかる時間
- データのリストアにかかる時間
- 保証されているデータバックアップのレベル（過去 24 時間以内のすべてのデータを回復、過去 1 時間以内の特定のトランザクションデータを回復など）

4.4.4 信頼性テストの計画

一般的に、信頼性テストを計画する際は次の点が特に重要である。

- ソフトウェアの本番稼働中に信頼性を引き続き監視する。テスト計画の目的に適した信頼性要件を収集する際は、ソフトウェアの運用を担当する組織とスタッフに相談する必要がある。
- テクニカルテストアナリストは、時間経過とともに期待する信頼性レベルを示す信頼度成長モデルを選択する場合がある。信頼度成長モデルは、期待する信頼性レベルと達成した信頼性レベルを比較することにより、有用な情報をテストマネージャに提供できる。

- 信頼性テストは、本番に近い環境で行う必要がある。使用する環境は、定常的に信頼性の傾向を監視できるように、できるだけ安定している必要がある。
- 信頼性テストでは多くの場合、システム全体を使用する必要があるので、システムテストの一部として実行するのが最も一般的である。しかし個々のコンポーネントは統合したコンポーネントセットと同様に信頼性テストを実行することもできる。また、アーキテクチャ、設計、およびコードの詳細なレビューにより、実装するシステムで発生する信頼性の問題のリスクをある程度除去することもできる。
- 統計的に意味のあるテスト結果を得るために、信頼性テストは通常、長期間実行する必要がある。このため他に計画したテストの中で実行するのは難しいことがある。

4.4.5 信頼性テストの仕様

信頼性テストは、事前に決めたテストセットを繰り返す形を取る場合がある。これらのテストは、開発済みのテストからランダムに選択したテストケースあるいは、ランダムないし擬似ランダムな方法を使って静的モデルから生成したテストケースの場合がある。また、テストは時々「運用プロファイル」とも呼ぶ使用パターンに基づく場合もある(4.5.4 節を参照)。

特定の信頼性テストでは、発生する可能性があるメモリリークを検出できるように、メモリ集約型の操作を繰り返し実行することを規定する場合がある。

4.5 性能テスト

4.5.1 イントロダクション

ISO9126 によるプロダクト品質特性の分類では、性能(時間効率性)は効率性の副特性である。性能テストは、コンポーネントまたはシステムが、ユーザやシステムの入力に対して指定した時間内に指定した条件で応答する能力に焦点を当てている。

性能の測定はテストの目的によって変わる。個々のソフトウェアコンポーネントでは、性能を CPU サイクルに基づいて測定する場合があるが、クライアントベースシステムでは、特定のユーザ要求に応答する時間に基づいて性能を測定する場合がある。複数のコンポーネント(クライアント、サーバ、データベースなど)からなるアーキテクチャのシステムでは、性能の「ボトルネック」を明らかにするために個々のコンポーネント間のトランザクションに対して性能を測定する。

4.5.2 性能テストの種類

4.5.2.1 ロードテスト

ロードテストは、多数の同時ユーザまたは並行プロセスが生成するトランザクション要求から生じる、現実的な負荷の想定に対するシステムの処理能力に焦点を当てている。典型的な用途のさまざまなシナリオ(運用プロファイル)に基づいて、ユーザへの平均応答時間を、測定し分析することができる([Splaine01]を参照)。

4.5.2.2 ストレステスト

ストレステストは、想定および指定した負荷の限界、および限界を超えた時、あるいはアクセス可能なコンピュータ能力や使用可能な帯域幅などのリソースの可用性が下がった時に、システムまたはコンポーネントが最大負荷を処理する能力に焦点を当てている。ストレスの度合いが上昇しても故障せずに性能は徐々に想定通りに下がるべきである。特に、処理中に起こりうる機能面の欠陥やデータ不整合を検出するために、ストレスをかけながらシステムの機能完全性をテストする必要がある。

ストレステストの考えられる目的の1つは、「鎖の最も弱い部分」を特定するために、システムが実際に故障する限界を発見することである。ストレステストにより、能力(メモリ、CPU、データベースストレージなど)をシステムにタイムリーに追加できる。

4.5.2.3 拡張性テスト

拡張性テストは、現在の要件を超えた将来の効率性要件を、システムが満たす能力に焦点を当てている。このテストの目的は、現在定めている性能を遵守しながら、故障せずに拡張できるシステムの能力(ユーザの増加、格納するデータ量の増加など)を見極めることである。拡張性の限界を把握することで、しきい値を設定し本番での監視中に差し迫った問題に対して警告を発することができる。さらに、本番環境を適切なハードウェア台数によって調整することもある。

4.5.3 性能テストの計画

4.2 節で説明した一般的な計画上の課題に加えて、次の要因が性能テストの計画に影響する可能性がある。

- 使用するテスト環境とテストするソフトウェアによっては(4.2.3 節を参照)、有効な性能テストの実行のためにはシステム全体が実装済みであることが必要な場合がある。この場合通常、性能テストの実行をシステムテスト中に計画する。コンポーネントレベルで効果的に実行できるその他の性能テストは、ユニットテストの期間にスケジュールすることもある。
- 一般的に製品に近いものが未だ使用できない時でも、性能テストをできるだけ早期に行う方が良い。このような早期のテストで性能問題(ボトルネックなど)を検出し、ソフトウェア開発の最終段階や本番で時間のかかる修正を避けることにより、プロジェクトリスクを低減できる。
- 特にデータベース操作、コンポーネントとのやりとり、およびエラー処理に焦点を当てたコードレビューは、性能問題(特に「待機してリトライ」するロジックと非効率なクエリに関する問題)を特定できるので、ソフトウェアライフサイクルの早期にスケジュールする必要がある。
- 性能テストの実行に必要なハードウェア、ソフトウェア、およびネットワーク帯域幅を計画し予算化する必要がある。ニーズは主に生成する負荷によるが、それはシミュレートする仮想ユーザ数とそれらが生成するであろうネットワークトラフィック量に基づく場合がある。このことを考慮しないと代表値といえない性能測定値を取得してしまう場合がある。たとえば、かなりの訪問者があるインターネットサイトの拡張性要件の確認では、数十万の仮想ユーザのシミュレーションが必要になる場合がある。
- 性能テストに必要な負荷の生成がハードウェアとツールの調達コストに大きな影響を与える場合がある。このことは、十分な予算を確保するために、性能テストの計画で考慮する必要がある。
- 性能テスト用の負荷を生成するコストは、必要なテストインフラをレンタルすることにより最小化できる場合がある。この例としては、性能ツールの「トップアップ(必要な分だけ事前にチャージする) 」ライセンスをレンタルする、またはハードウェアのニーズを満たすサードパーティプロバイダのサービス(クラウドサービスなど)を使用する、などがある。このアプローチを取る場合、性能テストを実行できる時間が制限される場合があるので、慎重に計画する必要がある。
- 計画段階で注意が要るのは、使用する性能ツールが、テスト対象システムが使う通信プロトコルとの必要な互換性を持つことを確認することである。
- 性能関連の欠陥は多くの場合、テスト対象システムに重大な影響を及ぼす。性能要件が必須なシステムでは、多くの場合、システムテストを待たずに重要なコンポーネントに対して(ドライバとスタブを介して)性能テストを実行することが有用である。

4.5.4 性能テストの仕様

ロードテストやストレステストなど、さまざまな種類の性能テストの仕様は、運用プロファイルの定義に基づいている。これらは、アプリケーションと相互作用する際、ユーザの振る舞いのさまざまな形態を示している。1つのアプリケーションに対して、複数の運用プロファイルが存在する場合がある。

運用プロファイルあたりのユーザ数は、モニタリングツールを使用するか(実際ないし同等なアプリケーションを使える場合)、または使用状況を想定して取得できる。このような想定は、アルゴリズムに基づいて行うか、またはビジネス組織が提供する場合がある。特に拡張性テストに使用する運用プロファイルを指定するために重要である。

運用プロファイルは、性能テストで使用するテストケースの数と種類の基になる。これらのテストは多くの場合、テスト対象のプロファイルに記した「仮想」ユーザ、またはシミュレートするユーザ数を作成するツールが制御する。

4.6 資源効率性テスト

ISO9126 によるプロダクト品質特性の分類では、資源効率性は効率性の副特性である。資源効率性に関するテストは、システム資源(メモリ、ディスク容量、ネットワーク帯域幅、コネクションなど)の使用状況を、事前に定義したベンチマークに対して評価する。これらの使用状況は、使用量が不自然に増加していないかを判断するために、通常の負荷状況と、トランザクション量やデータ量が多いなどのストレス状況の両方で比較する。

たとえばリアルタイム組み込みシステムでは、メモリ使用量(「メモリフットプリント」とも呼ぶ)が性能テストの重要な役割を果たす。メモリフットプリントが許容値を超えた場合は、特定時間内にそのタスクを実行するのに十分なメモリが、システムに存在しない可能性がある。このためにシステムは遅くなり、あるいはシステムクラッシュすることさえある。

また、資源効率性の調査(3.3.4 節を参照)、および性能ボトルネックの検出を行うために動的解析を適用する場合もある。

4.7 保守性テスト

ソフトウェアは多くの場合、そのライフサイクルの中で開発期間よりも保守期間の方が大幅に長い。保守テストは、稼働中のシステムへの変更、または環境の変更が稼働中のシステムに与える影響をテストするために行う。保守をできる限り効率的に行うために、保守性テストでは、コードを解析、変更、およびテストできる容易性を測定する。

関係するステークホルダ(ソフトウェアの所有者や運用者など)の一般的な保守性の目的は、次のとおりである。

- ソフトウェアを所有または運用するコストを最小化する。
- ソフトウェアの保守に必要なダウンタイムを最小化する。

次の要因が1つ以上当てはまる場合は、テスト戦略およびテストアプローチに保守性テストを含む必要がある。

- ソフトウェアの本番稼働後にソフトウェアを変更する可能性がある(欠陥の修正や計画した更新の導入など)。
- 影響力のあるステークホルダがソフトウェアライフサイクルを通して保守性の目的(上記参照)を達成することの利点が、保守性テストを実行し、必要な変更を行うコストを上回ると考えている。
- ソフトウェアの保守性が悪い(ユーザまたは顧客により報告された欠陥への対応時間が長いなど)場合のリスクが、保守性テストの正当な理由になる。

保守性テストの適切な技法には、3.2 節と5.2 節で説明した静的解析とレビューがある。保守性テストは設計ドキュメントが入手可能になり次第開始し、コード実装中も継続する必要がある。保守性は、個々のコンポーネントのコードとドキュメントに組み込まれているので、システムの完成と稼働を待たずに、ライフサイクルの早期に評価できる。

動的な保守性テストは、特定のアプリケーションを保守する(ソフトウェアアップグレードの実行など)ために作成した、文書化された手順に焦点を当てている。選択した保守シナリオをテストケースとして使用し、要求されたサービスレベルが文書化された手順で達成できることを確認する。この種類のテストは、基になるインフラが複雑で、サポート手順に複数の部門／組織が関わる場合に、特に適している。また、この種類のテストは、運用受け入れテスト(OAT)の一部として使う場合がある[Web-1]。

4.7.1 解析性、変更性、安定性、試験性

システムの保守性は、システム内で発見された問題の診断(解析性)、コード変更の実装(変更性)、および変更したシステムのテスト(試験性)に必要な労力によって測定できる。安定性は特に、変更に対するシステムの反応に関連する。安定性の低いシステムは、変更を行うとその先で多くの問題を起こす(「波及効果」とも呼ぶ)[ISO9126] [Web-1]。

保守作業の実行に必要な労力は、使用するソフトウェア設計の方法論(たとえば、オブジェクト指向)、コーディング標準など、さまざまな要因に依存する。

この文脈での「安定性」を、4.4.2 節で説明した「頑健性(堅牢性)」および「障害許容性」という用語と混同しないように注意すること。

4.8 移植性テスト

移植性テストは一般的に、ソフトウェアを目的の環境に、新規にあるいは既存環境から移植できる容易さに関連する。移植性テストは設置性、共存性／互換性、環境適応性、および置換性のテストを含む。移植性テストは、個々のコンポーネント(たとえば、1つのデータベース管理システムを別のものに替えるような特定のコンポーネントの置換性)から始め、入手できるコードが増えるにつれてテスト範囲を広げることができる。設置性をテストできるのはプロダクトの全コンポーネントが動くようになってからになる。移植性は、設計してプロダクトに組み込む必要があるので、アーキテクチャと設計のフェーズの早期に検討する必要がある。アーキテクチャと設計のレビューは、潜在的な移植性の要件と問題(特定のオペレーティングシステムへの依存など)を発見するのに特に有効である。

4.8.1 設置性テスト

設置性テストは、対象とする環境にソフトウェアをインストールするために使用するソフトウェアと、文書化された手順に対して行う。たとえば、オペレーティングシステムをプロセッサにインストールするために開発したソフトウェアや、プロダクトをクライアント PC にインストールするために使用するインストール「ウィザード」などがある。

一般的な設置性テストの目的は、次のとおりである。

- インストールマニュアルの指示(インストールスクリプトの実行を含む)に従って、またはインストールウィザードを使用して、ソフトウェアを正常にインストールできることを確認する。この確認は、異なるハードウェア／ソフトウェア構成、およびさまざまなインストールの程度(新規、更新など)に対するインストールオプションの実行を含んでいる。
- インストール中に発生した故障(特定の DLL をロードできないなど)が、システムを未定義の状態(部分的にインストールされたソフトウェアや誤ったシステム構成など)のままにせず、インストールソフトウェアにより正しく処理されるかどうかをテストする。
- 部分インストール／アンインストールが完了するかどうかをテストする。
- インストールウィザードが、無効なハードウェアプラットフォーム構成、または無効なオペレーティングシステム構成を正しく識別できるかどうかをテストする。
- インストールプロセスが特定の時間内、または特定のステップ数以内で完了するかどうかを測定する。

- ソフトウェアを問題なくダウングレード、アンインストールできるかどうかを確認する。

インストールに関する機能性テストは通常、インストールが引き起こす欠陥（誤った構成や使用できない機能など）を検出するためにインストールテストの後に実施する。インストールに関する使用性テストは通常、設置性テストと並行して行う（インストール時に、分かりやすい指示やフィードバック／エラーメッセージがユーザに提示されているかを確認するなど）。

4.8.2 共存性／互換性テスト

互いに無関係なコンピュータシステムが、互いの振る舞いに影響を与える(リソース競合など)ことなく、同じ環境（同じハードウェアなど）で稼働できる場合、共存性/互換性があるという。新規のあるいはアップグレードしたソフトウェアを、アプリケーションがすでにインストールされている環境に展開する場合、共存性/互換性テストを実行する必要がある。

共存性/互換性の問題は、あるアプリケーションをそのアプリケーションのみがインストールされている環境でテストし（この場合、共存性/互換性の問題は検出されない）、その後他のアプリケーションも動いている別の環境（本番環境など）に展開した場合に発生することがある。

一般的な共存性/互換性テストの目的は、次のとおりである。

- 複数のアプリケーションを同一環境にロードした時に、機能に悪影響がないかを評価する（サーバで複数のアプリケーションを実行した場合のリソース使用の競合など）。
- オペレーティングシステムの修正やアップグレードの展開に起因する、あらゆるアプリケーションへの影響を評価する。

共存性/互換性の問題は、目的の本番環境を計画する際に分析すべきであるが、実際のテストは通常、システムテストおよびユーザ受け入れテストを正常に完了した後で実行する。

4.8.3 環境適応性テスト

環境適応性テストでは、全ターゲット環境（ハードウェア、ソフトウェア、ミドルウェア、オペレーティングシステムなど）で所定のアプリケーションが正しく機能するかどうかを確認する。従って、環境適応性のあるシステムとは、環境ないしシステム自体の部分的変更に応じて、その振る舞いを適応できるオープンなシステムである。環境適応性テストを仕様化するには、ターゲット環境の組み合わせを識別し、設定し、テストチームが使用できるようにする必要がある。その後、環境内に存在するさまざまなコンポーネントを動かす機能テストのテストケースを選択し、これらの環境をテストする。

環境適応性は、事前に定義した手順を実行して、さまざまな特定の環境に移植できるソフトウェアの能力にも関連する。テストではこの手順を評価することもある。

環境適応性テストは、設置性テストと一緒に実行することもでき、一般的にはその後に機能テストを行って、ソフトウェアを異なる環境に適応させることで発生する欠陥を検出する。

4.8.4 置換性テスト

置換性テストは、システム内のソフトウェアコンポーネントを他のコンポーネントに置換できる能力に焦点を当てる。このテストは、特定のシステムコンポーネントとして市販ソフトウェア（COTS）を使用するシステムでは特に重要となる場合がある。

置換性テストは、完成システムに統合するのに複数の代替コンポーネントを使用できる場合、機能統合テストと並行して実行する場合がある。置換性は、アーキテクチャおよび設計段階で、テクニカルレビューまたはインス

ペクションにより評価する場合があり、その際は、潜在的に置換できるコンポーネントのインターフェースを明確に定義することが重要である。

5. レビュー - 165 分

用語

アンチパターン

「レビュー」の学習の目的

5.1 イントロダクション

TTA 5.1.1 (K2)テクニカルテストアナリストにとって、レビューの準備が重要である理由を説明する。

5.2 レビューでのチェックリストの使用

TTA 5.2.1 (K4)シラバスが提供するチェックリストに従って、アーキテクチャ設計を分析し、問題を識別する。

TTA 5.2.2 (K4)シラバスが提供するチェックリストに従って、コードまたは擬似コードの部分を分析し、問題を識別する。

5.1 イントロダクション

テクニカルテストアナリストは、レビュープロセスに積極的に参加し、独自の見解を提供する必要がある。テクニカルテストアナリストは公式なレビュートレーニングを受け、テクニカルレビュープロセスにおける自分の役割についてよく理解しなければならない。すべてのレビュー参加者は、テクニカルレビューを十分に行い、成果を生むように専念する必要がある。多くのレビューチェックリストを含む、テクニカルレビューの詳細な説明については、[Wiegers02]を参照されたい。テクニカルテストアナリストは通常、開発者が見逃すことのある運用(動作)上の観点を持って、テクニカルレビューとインスペクションに参加する。さらに、テクニカルテストアナリストは、レビューチェックリストと欠陥重要度情報の定義、適用、および保守において、重要な役割を果たす。

実行するレビューの種類に関係なく、テクニカルテストアナリストは適切な準備時間を確保できる必要がある。この準備時間は、成果物をレビューする時間、一貫性を確認するために相互参照しているドキュメントをチェックする時間、および成果物に何が欠けているかを判定する時間を含んでいる。適切な準備時間がなければ、レビューが本当のレビューではなく、単なる編集作業となる可能性がある。優れたレビューは、記述された内容の理解、欠けている内容の判定、および成果物に記述された内容が、すでに開発済み、もしくは開発中の他の成果物と整合性があることの確認を含んでいる。たとえば、テクニカルテストアナリストは統合レベルテスト計画のレビューで、統合するアイテムについて統合の準備ができているか、文書化する必要がある依存関係が存在するか、および統合箇所のテストに使用できるデータが存在するかを検討する必要がある。レビューは、レビューする成果物のみを対象とするのではなく、そのアイテムと、システム内の他のアイテムとの相互作用も考慮する必要がある。

レビューされる成果物の作成者が、批判されていると感じることは珍しいことではない。テクニカルテストアナリストは、できるだけ最高の成果物を作成するために作成者と協力しているという観点で、レビューコメントを提示する必要がある。このアプローチを使用することで、コメントの表現が建設的になり、コメントの対象を作成者ではなく、成果物にすることができる。たとえば、記述が曖昧な場合は、「この要件は曖昧なので、誰も理解できない」と述べるよりも、「この要件が正しく実装されていることを確認するために、何をテストするべきか私には理解できません。私が理解できるように教えてくれませんか？」と述べる方が良い。

テクニカルテストアナリストはレビューにおいて、成果物で提供された情報が、テスト作業をサポートするのに十分であることを確認する。その情報が存在しない、または明確でない場合、それは欠陥である可能性があり、作成者が修正する必要がある。批判的なアプローチではなく前向きなアプローチを続けることで、コメントが受け入れられやすくなり、会議がより生産的になる。

5.2 レビューでのチェックリストの使用

レビューでチェックリストを使用することにより、参加者はレビュー時に確認する具体的なポイントを認識できる。また、チェックリストは、属人的なレビューを避けるのにも役立つ。たとえば、「このチェックリストはすべてのレビューで使用しているものと同じで、あなたの成果物のみを対象としているのではない」と示すことができる。チェックリストは、汎用化してすべてのレビューで使用することも、または特定の品質特性や領域に焦点を絞ることもできる。たとえば、汎用的なチェックリストは、用語「必須」と「推奨」の適切な使用を確認し、適切な書式および同じような適合項目を確認することができる。セキュリティの問題や性能の問題に焦点を当て、対象を定めたチェックリストもある。

最も有用なチェックリストは、個々の組織が次の項目を反映しながら徐々に発展させたチェックリストである。

- プロダクトの特性
- ローカルの開発環境
  - スタッフ
  - ツール

- 優先度
- これまでの成果と欠陥の記録
- 特定の問題(性能、セキュリティなど)

チェックリストは、組織あるいは特定のプロジェクトに合わせて、カスタマイズする必要がある。本章で説明するチェックリストは、例として挙げているものである。

一部の組織は、ソフトウェアのチェックリストの一般的な考えを拡張して、共通の誤り、稚拙な技法、およびその他の効果のないプラクティスを意味する「アンチパターン」を含めるように拡張している。この用語は実際の状況で共通問題に対して効果的であることが証明された再利用可能なソリューションである「デザインパターン」という普及した概念から来ている[Gamma94]。アンチパターンはよくある失敗であり、しばしば避けるべき近道として作成する。

要件がテストできない、すなわちテクニカルテストアナリストがテスト方法を決定できるように要件が定義されていないなら、それは欠陥である、と覚えておくことは重要である。たとえば、「ソフトウェアは速くなければならない」と記述された要件は、テストできない。この場合、テクニカルテストアナリストは、ソフトウェアが速いかどうかをどのように判定できるだろうか？一方、要件が、「ソフトウェアは特定の負荷状態で、3 秒以内の応答時間を実現する必要がある」と記述された場合は、「特定の負荷状態」(同時ユーザ数やユーザが実行する作業など)を定義すれば、この要件の試験性は大幅に向上する。普通のアプリケーションであれば、この1つの要件から個別のテストケースを容易に多数作り出すことができるので、これは包括的な要件でもある。また、要件が変更されたらすべてのテストケースをレビューして、必要に応じて更新しなければならないので、この要件からテストケースまでのトレーサビリティも重要である。

5.2.1 アーキテクチャレビュー

ソフトウェアアーキテクチャは、そのコンポーネントと各コンポーネント間および環境との関係を具現化するシステムの基本的構成、およびその設計と進化を統制する原則からなる([ANSI/IEEE Std 1471-2000]、[Bass03])。

たとえば、アーキテクチャレビューのチェックリストは、[Web-3]から引用した、次の項目が適切に実装されていることの確認を含む。

- コネクションプール- 共有のコネクションプールを作成することでデータベース接続の確立に伴う実行時間のオーバーヘッドを削減する。
- 負荷分散 - リソースセットの間で、負荷を均等に分散する。
- 分散コンピューティング
- キャッシング - アクセス時間を減らすために、データのローカルコピーを使う。
- 遅延インスタンス化
- トランザクションの並行処理
- オンライントランザクション処理(OLTP)とオンライン分析処理(OLAP)間のプロセスの分離
- データの複製

詳細については(認定試験には関係しない)、[Web-4]を参照されたい。それには 24 の出典から 117 のチェックリストを調査した論文に言及している。チェックリスト項目のさまざまな分類を説明し、適切なチェックリスト項目と避けるべき項目を例示している。

5.2.2 コードレビュー

コードレビューのチェックリストは必然的に非常に詳細であり、アーキテクチャレビューのチェックリストと同様に言語、プロジェクト、および会社固有である時に、非常に役に立つ。チェックリストにコードのアンチパターンを含むと有用であり、経験の浅いソフトウェア開発者にとっては特に有用である。

コードレビューで使用するチェックリストには、次の 6 項目を含めることができる([Web-5]に基づく)。

1.構造
- コードは、設計を完全に正しく実装しているか？
- コードは、関連するコーディング標準に準拠しているか？
- コードは、適切に構造化され、スタイルと形式に一貫性があるか？
- 不要なプロシージャ、または到達できないコードは存在しないか？
- コード内にスタブやテストルーチンが残っていないか？
- 外部の再利用可能なコンポーネントまたはライブラリ関数を呼ぶことで置換できるコードは存在しないか？
- 1つのプロシージャにまとめられる反復的なコードブロックは存在しないか？
- ストレージの使用は効率的か？
- 「マジックナンバー」定数または文字列定数ではなく、シンボルを使用しているか？
- 過度に複雑で再構築または複数のモジュールへの分割が必要なモジュールは存在しないか？

2.文書化
- コードは、保守しやすいコメント形式を使用して、明確かつ適切に文書化されているか？
- すべてのコメントは、コードと一貫性があるか？
- 文書は、適用可能な標準に準拠しているか？

3.変数
- すべての変数は、意味のある一貫した明確な名前を使用して、適切に定義されているか？
- 冗長あるいは未使用な変数は存在しないか？

4.算術演算
- コードは、浮動小数点数の等号による比較を避けているか？
- コードは、体系的に丸め誤差を防いでいるか？
- コードは、桁数が大きく異なる数字の加算および減算を避けているか？
- 除数がゼロまたは無効値のテストをしているか？

5.ループとブランチ
- すべてのループ、ブランチ、および論理構造は、完全で、正しくて、また適切にネストしているか？
- IF-ELSEIF チェーンの最初でテストするのは最も一般的なパターンとなっているか？
- 全ケースは ELSE ないし DEFAULT 句を含んだ IF-ELSEIF ないし CASE ブロックでカバーされているか？
- あらゆる CASE ブロックには DEFAULT 句があるか？
- ループの終了条件は明白で必ず成立するか？
- インデックスすなわち添え字は、ループの直前で適切に初期化されているか？
- ループ内のステートメントで、ループ外に出せるものはないか？
- ループ内のコードはループの終了時に、インデックス変数の操作または使用を避けているか？

6. 防御的プログラミング

- 配列、レコード、またはファイルの境界に対して、インデックス、ポインタ、および添え字をテストしているか？
- インポートするデータおよび入力引数に対して、入力の妥当性を網羅的にテストしているか？
- すべての出力変数を割り当てているか？
- 各ステートメントで、正しいデータ要素を操作しているか？
- すべてのメモリ割り当てを解放しているか？
- 外部デバイスのアクセスで、タイムアウトまたはエラートラップを使用しているか？
- ファイルにアクセスする前に、存在していることをチェックしているか？
- プログラムの終了時に、すべてのファイルとデバイスを正しい状態にしているか？

さまざまな段階のコードレビューで使用するチェックリストのより多くの例については、[Web-6]を参照されたい。

6. テストツールおよび自動化 - 195 分

用語

データ駆動テスト、デバッグツール、フォールトシーディングツール、ハイパーリンクテストツール、キーワード駆動テスト、性能テストツール、キャプチャ/プレイバックツール、静的アナライザ、テスト実行ツール、テストマネジメントツール

「テストツールおよび自動化」の学習の目的

6.1 ツール間の統合と情報交換

TTA-6.1.1 (K2)複数のツールを一緒に使用する際に考慮すべき技術的側面を説明する。

6.2 テスト自動化プロジェクトの定義

TTA-6.2.1 (K2)テスト自動化プロジェクトを立ち上げる際に、テクニカルテストアナリストが実行する活動をまとめる。
TTA-6.2.2 (K2)データ駆動とキーワード駆動による自動化の違いをまとめる。
TTA-6.2.3 (K2)自動化プロジェクトが計画した投資効果を達成できない原因となる、一般的な技術的問題をまとめる。
TTA-6.2.4 (K3)特定のビジネスプロセスに基づいたキーワード表を作成する。

6.3 特定のテストツール

TTA-6.3.1 (K2)フォールトシーディングツールとフォールトインジェクションツールの目的をまとめる。
TTA-6.3.2 (K2)性能テストツールとモニタリングツールの主な特性と実装上の問題をまとめる。
TTA-6.3.3 (K2)Web ベースのテストで使用するツールの一般的な目的を説明する。
TTA-6.3.4 (K2)どのようにツールがモデルベースドテストの概念をサポートするかを説明する。
TTA-6.3.5 (K2)コンポーネントテストとビルドプロセスをサポートするために使用するツールの目的を概説する。

6.1 ツール間の統合と情報交換

テストマネージャは、ツールの選択と統合に責任を持っているが、静的解析、テスト実行自動化、構成管理など、さまざまなテスト領域から得られるデータを正確に追跡するツールや、ツールセットの統合のレビューはテクニカルテストアナリストに依頼することがある。さらに、テクニカルテストアナリストは、十分なプログラミングスキルを用いて、「購入しただけ」では実現できないツール間の統合をするためにコードを書くこともある。

理想的なツールセットでは、ツール間の情報の重複は除去すべきである。たとえば、テスト管理データベースと構成管理システムの両方にテスト実行スクリプトを格納すると、より労力がかかり、間違いも起こしやすくなる。構成管理機能を搭載しているか、または組織にすでにある構成管理ツールと統合できるテストマネジメントシステムを持つのが望ましい。欠陥追跡ツールとテストマネジメントツールを上手く統合すれば、テスト担当者は、テストケース実行中にテストマネジメントツールから直接欠陥レポートを起票できる。上手く統合された静的解析ツールでは、検出したインシデントと警告を欠陥マネジメントシステムに直接報告できる必要がある(ただし、多数の警告が生成されることがあるので、この機能は設定可能である必要がある)。

同じベンダーからテストツール一式を購入しても、それらのツールが適切に連携して機能するとは限らない。ツールを統合するアプローチを検討する際は、データを重視する方が良い。データは、障害回復を含む保証された正確性で、人手を介さずにタイムリーに交換できる必要がある。一貫性のあるユーザエクスペリエンスも有益だが、ツール統合の最も重要な点は、データの取得、格納、保護、および表示でなければならない。

組織は、人手の介入が必要なせいで情報を失ったり、データが同期しなかったりするリスクと、情報交換を自動化するコストを比較し評価する必要がある。統合が高価または困難になる場合があるので、統合はツール戦略全体で最も重視すべきである。

一部の統合開発環境(IDE)では、その環境で動くツール間の統合を簡単にできることがある。このことは、同じフレームワークを共有するツールのルックアンドフィールを統一するのに役立つ。ただし、ユーザインターフェースが似ていても、コンポーネント間のスムーズな情報交換が保証されるわけではない。統合を完成するには、コーディングが必要なことがある。

6.2 テスト自動化プロジェクトの範囲

費用対効果を出すために、テストツール、特にテスト自動化ツールを慎重に設計する必要がある。堅牢なアーキテクチャ無しにテスト自動化戦略を実装すると、一般的に保守コストが高く、目的を十分に達成できず、目標の投資効果を達成できないツールセットになる。

テスト自動化プロジェクトは、ソフトウェア開発プロジェクトとして考える必要がある。この場合、アーキテクチャドキュメント、詳細な設計ドキュメント、設計とコードのレビュー、コンポーネントテストとコンポーネント統合テスト、そして最後にシステムテストが必要になる。不安定または不正確なテスト自動化コードを使用すると、テストが意味なく遅延したり、複雑化したりする可能性がある。テクニカルテストアナリストはテスト自動化に関して、次のような複数の活動を実行する。

- テスト実行の責任者を決定する。
- 組織、日程、チームのスキル、および保守の要件に合うツールを選択する(この場合、ツールを購入せずに作成することを決定する場合もある)。
- 自動化ツールと、テストマネジメントツールや欠陥マネジメントツールなどの他のツールとの間のインターフェース要件を定義する。
- 自動化のアプローチ(キーワード駆動またはデータ駆動)を選択する(以降の 6.2.1 節を参照)。
- テストマネージャと協力して、トレーニングを含む実装コストを見積る。
- 自動化プロジェクトの日程を立て、保守用の時間を割り当てる。

- 自動化用のデータを使用し提供するために、テストアナリストとビジネスアナリストをトレーニングする。
- 自動化したテストを実行する方法を決定する。
- 自動化したテストの結果と手動によるテストの結果を一つにまとめる方法を決定する。

これらの活動とそれに伴う決定は、自動化ソリューションの拡張性と保守性に影響する。オプションの調査、使用可能なツールと技術の調査、および組織の今後の計画の把握には、十分に時間をかける必要がある。上記のテクニカルテストアナリストがテスト自動化に関して行う活動の中には、特に意思決定プロセスで、他の活動と比較してより検討が必要になるものがある。これについては、次の節で詳細に説明する。

6.2.1 自動化アプローチの選択

テスト自動化は、GUI を使用するテストに限定されない。テスト対象ソフトウェアのコマンドラインインターフェース(CLI)および他のインターフェースを通じて、API レベルでテストの自動化を支援するツールが存在する。テクニカルテストアナリストが最初にすべきことの 1 つに、テストを自動化するためにアクセスする最も効果的なインターフェースの決定がある。

GUI によるテストの困難な点の 1 つは、ソフトウェアの進化に伴って GUI も変化しがちなことである。テスト自動化コードを設計する方法によっては、このことが保守における大きな負担になる可能性がある。たとえば、テスト自動化ツールの記録再生機能を使用すると、GUI を変更した場合、自動化されたテストケース(通常はテストスクリプトと呼ぶ)が期待通りに動作しない場合がある。記録されたスクリプトは、テスト担当者がグラフィカルオブジェクトを手動で操作した際の相互作用をキャプチャしているからである。スクリプトを記録する際に操作したオブジェクトに対して変更を行うと、変更を反映するために、スクリプトの更新も必要なことがある。

自動化スクリプトを開発するための第一歩として、キャプチャ／プレイバックツールを使うことは便利である。テスト担当者はテスト実行の一連の操作を記録し、その後、保守性を高めるために記録したスクリプトを変更する(記録したスクリプトの一部を、再利用可能な関数に置き換えるなど)。

テストするソフトウェアによっては、実行するテスト手順が事実上同じであるにも関わらず、使用するデータが異なる場合がある(複数の不正値を入力して、それぞれから戻されるエラーをチェックすることで、入力フィールドのエラー処理をテストするなど)。テストするこれらの値のそれぞれに自動テストスクリプトを開発し保守するのは非効率である。この問題に対する一般的な技術的解決策は、データをスクリプトから、スプレッドシートやデータベースなどの外部に移すことである。テストスクリプトのそれぞれの実行で、専用のデータにアクセスする関数を書くことで、1 つのスクリプトで入力値と期待結果(テキストフィールドに表示する値やエラーメッセージなど)となる一連のテストデータを処理できるようになる。このアプローチをデータ駆動と呼ぶ。このアプローチを使うと、与えたデータを処理するテストスクリプトに加えて、スクリプトまたはスクリプトセットの実行を支援するのに必要なハーネスとインフラストラクチャも開発できる。スプレッドシートまたはデータベースに保存される実際のデータは、ソフトウェアが利用される業務に精通したテストアナリストが作成する。この作業の分割により、テストスクリプトの開発担当者(テクニカルテストアナリストなど)は高度な自動化スクリプトの実装に集中し、一方テストアナリストは実際のテストに対する責任を持つ。ほとんどの場合、自動化を実装しテストした後は、テストアナリストがテストスクリプトの実行に責任を持つ。

キーワード駆動またはアクションワード駆動と呼ぶ別のアプローチでは、さらに一歩進めて、与えたデータに対して実行するアクションをテストスクリプトから外部に移す [Buwalda01]。アクションを外部に移すために、ドメインの専門家(テストアナリストなど)が、意味の分かる高位レベルのメタ言語を作成する。この言語の各ステートメントは、ドメインのテストを必要とするビジネスプロセスの全体または一部を記述する。たとえば、ビジネスプロセスのキーワードには、「Login」(ログイン)、「CreateUser」(ユーザの作成)、「DeleteUser」(ユーザの削除)などがある。これらのキーワードは、アプリケーションで実行する高位レベルのアクションを記述している。

また、ビジネスプロセスのキーワードでは適さない GUI 機能のテストに、ソフトウェアインターフェース自体との相互作用を示す「ClickButton」(ボタンのクリック)、「SelectFromList」(リストからの選択)、「TraverseTree」(ツリー上の移動)などの低位レベルのアクションを定義し使用する場合がある。

使用するキーワードとデータを定義した後は、テスト自動化担当者(テクニカルテストアナリストなど)が、ビジネスプロセスのキーワードと低位レベルのアクションをテスト自動化コードに変換する。キーワードとアクションは、使用するデータと共にスプレッドシートに格納するか、またはキーワード駆動テストの自動化をサポートする専用ツールを使用して入力する。テスト自動化フレームワークでは、1 つ以上の実行可能な関数またはスクリプトのセットとして、キーワードを実装する。ツールは、キーワードを使用して記述したテストケースを読み込み、それらを実行する適切な関数またはスクリプトを呼び出す。実行可能ファイルは、特定のキーワードに容易にマッピングできるように、高度なモジュール化方式で実装する。これらのモジュール化したスクリプトを実装するには、プログラミングスキルが必要になる。

このようにビジネスロジックの知識をテスト自動化スクリプトの実装に必要な実際のプログラミングから分離すると、テストリソースを最も効果的に利用できるようになる。テスト自動化担当の役割を持ったテクニカルテストアナリストは、多くのビジネス領域にわたるドメインの専門家になる必要はないので、プログラミングスキルを効果的に適用できる。

変化しやすいデータからコードを分離すれば、自動化はその変化の影響を受けることがなくなり、コードの全体的な保守性を改善し、自動化の投資効果を向上できる。

どんなテスト自動化設計においても、ソフトウェアの故障を予想し対処することが重要である。自動化担当者は、故障が発生した場合に、ソフトウェアに対して行うべきことを決める必要がある。たとえば、故障を記録してテストを継続すべきか？ テストを終了すべきか？ 特定のアクション(ダイアログボックスのボタンをクリックするなど)ないしテストスクリプトに待機時間を追加することなどで故障に対処できるか？
ソフトウェアの故障に対処しないと、故障が発生した時に実行していたテストで問題を引き起こすだけでなく、後続のテストで誤った結果が生じることがある。

また、テストの開始時と終了時のシステム状態を考慮することも重要である。テスト実行の完了後、システムを事前に定義した状態に確実に戻すことが必要なことがある。この場合システムを既知の状態に戻すのに人手は介在しないので、自動化された一連のテストを繰り返し実行できるようになる。そのためには、テスト自動化で、たとえば作成したデータを削除したり、データベースレコードの状態を変更したりしなければならないことがある。自動化フレームワークでは、テストの終了時に適切に終了することを保証する(つまり、テストの終了後にログアウトする)必要がある。

6.2.2 自動化のためのビジネスプロセスのモデル化

テスト自動化のキーワード駆動アプローチを実装するために、テストするビジネスプロセスを高位レベルのキーワード言語でモデル化する必要がある。その言語は、ユーザ(通常はプロジェクトで作業するテストアナリスト)にとって直感的であることが重要である。

キーワードは一般的に、ビジネスとシステムの相互作用を高位レベルで表現するのに使う。たとえば、「Cancel_Order」(注文のキャンセル)では、注文の存在チェック、キャンセルを要求するユーザのアクセス権の確認、キャンセルする注文の表示、およびキャンセルの確認要求が必要になる場合がある。テストアナリストは、テストケースを記述するために、一連のキーワード(「Login」、「Select_Order」、「Cancel_Order」など)と関連するテストデータを使用する。次のような簡単なキーワード駆動入力表を使うと、ユーザアカウントを追加、リセット、および削除するソフトウェアの機能をテストできる。

キーワード	ユーザ	パスワード	結果
Add_User	User1	Pass1	「ユーザを追加しました」メッセージ
Add_User	@Rec34	@Rec35	「ユーザを追加しました」メッセージ
Reset_Password	User1	Welcome	「パスワードのリセットを行いますか」メッセージ
Delete_User	User1		「無効なユーザ名／パスワードです」メッセージ
Add_User	User3	Pass3	「ユーザを追加しました」メッセージ
Delete_User	User2		「ユーザが存在しません」メッセージ

この表を使用する自動化スクリプトは、自動化スクリプトが使用する入力値を探す。たとえば、キーワード「Delete_User」の行に来るとユーザ名のみが必要になる。新しいユーザを追加するには、ユーザ名とパスワードの両方が必要になる。また、2 番目のキーワード「Add_User」が示すように、入力データを用意されたデータから取得する場合もある。この場合、データ自体ではなくデータの取得先を入力するので、テストの実行中に変化することのあるデータにアクセスするための柔軟性が向上する。これにより、データ駆動技法をキーワード方式と組み合わせることができる。

考慮すべき点としては、次のものがある。

- キーワードを細かくすれば、より詳細なシナリオをカバーできるが、高位レベルの言語を使用することで保守がより複雑になることがある。
- テストアナリストが低位レベルのアクション（「ClickButton」、「SelectFromList」など）を指定できれば、キーワードテストがより多くのさまざまな状況に対応できるようになる。しかし、これらのアクションは GUI に直接結びついているので、変更が発生すると、テストケースに対してより多くの保守が必要になることがある。
- 集約したキーワードを使うと開発は簡単になるが、保守が複雑になる場合がある。たとえば、6 つの異なるキーワードをまとめて記述する場合と、アクションを簡略化するために、実際には 6 つのキーワードを順に呼び出す 1 つのキーワードを作成する場合のどちらが良いだろうか？
- キーワードをどんなに分析しても、新しい異なるキーワードが必要になることが多い。キーワードには、2 つの異なる領域（キーワードの背後にあるビジネスロジックと、それを実行する自動化機能）が存在する。従って、両方の領域に対応するプロセスを作成する必要がある。

キーワードベースのテスト自動化は、テスト自動化の保守コストを大幅に削減できるが、テスト自動化の開発にはよりコストがかかり、難易度が高く、期待する投資効果を得るために正しく設計するにはより長い時間がかかる。

6.3 特定のテストツール

この節では、テクニカルテストアナリストが使用する可能性があるツールについて、Advanced Level テストアナリスト[ISTQB_ALTA_SYL]および Foundation Level シラバス[ISTQB_FL_SYL]よりも詳しい情報を提供する。

6.3.1 フォールトシーディング／フォールトインジェクションツール

フォールトシーディングツールは、主にコードに対して使用し、コード中の 1 つまたは限られたタイプの欠陥を体系的に作成する。これらのツールは、テストスイートの品質（つまり、欠陥を検出する能力）を評価するために、意図的に欠陥をテスト対象に混入させる。

フォールトインジェクションは、テスト対象を異常な状態にすることで、テスト対象に組み込まれた故障を処理するメカニズムをテストすることに焦点を当てている。フォールトインジェクションツールは、意図的に不正な入力をソフトウェアに与えて、ソフトウェアが故障に対処できることを確認する。

これら 2 種類のツールは、一般的にテクニカルテストアナリストが使用するが、新しく開発したコードをテストする際に開発者が使用する場合もある。

6.3.2 性能テストツール

性能テストツールには、2 つの主な機能がある。
- 負荷生成
- 所定の負荷に対するシステム応答の測定と分析

負荷は、事前に定義した運用プロファイル（4.5.4 節を参照）をスクリプトとして実装し実行することで生成される。スクリプトは、最初に単一ユーザとしてキャプチャし（通常はキャプチャ/プレイバックツールを使用）、次に性能テストツールを使って指定した運用プロファイル用に実装する。この実装では、トランザクション（またはトランザクションセット）単位のデータのバリエーションを考慮する必要がある。

性能テストツールは、多数のユーザ（「仮想」ユーザ）をシミュレートして負荷を生成し、指定した運用プロファイルに従って具体的な量の入力データを生成する。多くの性能テストスクリプトは、GUI を介してユーザの相互作用をシミュレートするテスト実行自動化スクリプトと異なり、通信プロトコルレベルでシステムとユーザとの相互作用を再現している。これにより通常、テスト時に必要な個別の「セッション」数が減る。負荷生成ツールの中には、負荷のかかった状態でのシステムの応答時間をより厳密に測定するために、ユーザインターフェースを使用してアプリケーションを実行することができるものもある。

性能テストツールは、テストの実行中または実行後に分析できるように広範な測定値を取得する。取得する一般的なメトリクスと提供するレポートには、次のものがある。
- テスト中にシミュレートするユーザ数
- シミュレートしたユーザが生成するトランザクションの数と種類、およびトランザクションの達成率
- ユーザが作成した特定のトランザクション要求の応答時間
- 応答時間に対する負荷のレポートとグラフ
- リソース使用状況のレポート（時間経過に伴う最小値と最大値を含む使用状況など）

性能テストツールの実装時に考慮する重要な要素には、次のものがある。
- 負荷の生成に必要なハードウェアとネットワーク帯域幅
- ツールと、テスト対象システムが使用する通信プロトコルとの互換性
- さまざまな運用プロファイルを容易に実装できるツールの柔軟性
- 必要な監視、分析、およびレポートの機能

性能テストツールは、開発するのに多くの工数が必要になるため、社内で開発せずに購入するのが一般的である。ただし、技術的な制約のためにプロダクトが使用できない場合、または使用するロードプロファイルや機能が比較的単純な場合は、専用の性能ツールを開発する方が適切なこともある。

6.3.3 Web ベースのテストのためのツール

Web テストでは、さまざまなオープンソースツール、および市販の専用ツールを使用できる。次のリストは、一般的な Web ベースをテストするツールの目的である。
- ハイパーリンクテストツール。Web サイトをスキャンして、リンクが切れているハイパーリンクまたはリンク先不明のハイパーリンクが存在しないことをチェックするために使用する。
- HTML チェッカーと XML チェッカー。Web サイトに作成したページが HTML 標準と XML 標準に準拠していることをチェックするツールである。
- 負荷シミュレータ。多数のユーザが接続した際にサーバがどのように応答するかをテストする。
- さまざまなブラウザと連携する軽量の自動実行ツール。

- サーバ全体をスキャンし、孤立した(リンクされていない)ファイルをチェックするツール。
- HTML 専用のスペルチェッカー。
- カスケーディングスタイルシート(CSS)チェックツール。
- 標準違反のチェックツール。米国の第 508 条アクセシビリティ基準やヨーロッパの M/376 など。
- さまざまなセキュリティの問題を検出するツール。

[Web-7]はオープンソースの Web テストツールの良い情報源である。この Web サイトを運営している組織がインターネットの標準を定めており、これらの標準に対するエラーをチェックするさまざまなツールを提供している。

Web スパイダーエンジンを含むいくつかのツールは、ページのサイズ、それらをダウンロードするのに必要な時間、およびページが存在するかどうか(HTTP エラー404 など)に関する情報も提供できる。これらは、開発者、Web マスター、およびテスト担当者にとって有用な情報である。

テストアナリストとテクニカルテストアナリストは、主にシステムテストの際にこれらのツールを使用する。

6.3.4 モデルベースドテストをサポートするツール

モデルベースドテスト(MBT)は、ソフトウェア制御システムの意図した実行時の動作を記述するために、有限状態機械などの形式的なモデルを使用する技法である。市販の MBT ツール([Utting 07]を参照)は、ユーザがモデルを「実行」できるエンジンを提供することが多い。興味深い実行スレッドを保存してテストケースとして使用できる。ペトリネットやステートチャートなど、他の実行可能なモデルも MBT をサポートする。MBT モデル(およびツール)を使って異なる実行スレッドのセットを大量に生成できる。

MBT ツールを使用すると、モデルで生成される膨大な数のパスを減らすことができる。

これらのツールを使用したテストでは、テスト対象のソフトウェアに対して異なる見方が与えられる。その結果、機能テストで見逃していたかもしれない欠陥を発見できる可能性がある。

6.3.5 コンポーネントテストツールとビルドツール

コンポーネントテストツールとビルド自動化ツールは開発者向けツールであるが、テクニカルテストアナリストは、これらのツールを特にアジャイル開発で使用し保守することが多い。

コンポーネントテストツールは、特定のプログラミング言語に対応するものが多い。たとえば、プログラミング言語として Java を使用した場合は、JUnit を使用してユニットテストを自動化することができる。多くの他の言語にも独自の専用テストツールがあり、これらを総称して xUnit フレームワークと呼ぶ。このようなフレームワークは、作成するクラス単位でテスト対象を生成するので、コンポーネントテストを自動化する際にプログラマが実行する必要のあるタスクが簡単になる。

デバッグツールにより、非常に低い水準でのコンポーネントテストが手動で簡単に行えるため、開発者やテクニカルテストアナリストは実行中に変数の値を変更したり、テスト中にコードを 1 行ずつ進めたりすることができる。また、テストチームが故障を報告した際に、開発者はデバッグツールを使用することで、コード内の問題を切り分け、識別することができる。

ビルド自動化ツールは多くの場合、コンポーネントが変更されるたびに新しいビルドを自動的に行う。ビルドが完了すると、他のツールが自動的にコンポーネントテストを実行する。ビルドプロセスにおけるこの段階の自動化は、通常、継続的インテグレーション環境で行われる。

このツールセットを適切に構築すると、テストに向けてリリースされるビルドの品質に非常に良い影響を与えることができる。プログラマが行った変更により、ビルドに回帰欠陥が混入した場合、ビルドをテスト環境にリリースする前に、通常はいくつかの自動テストが失敗するので、これをきっかけに故障の原因を即時に調査できる。

7. 参照文献

7.1 標準

次の標準について、それぞれの章で記載している。

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems
  第 5 章

- IEC-61508
  第 2 章

- [ISO25000]: ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)
  第 4 章

- [ISO9126]: ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality
  第 4 章
  JSTQB 訳注) 日本では JIS X 0129-1 として発行されている。

- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992.
  第 2 章

7.2 ISTQB ドキュメント

- [ISTQB_AL_OVIEW]: ISTQB Advanced Level Overview, Version 2012
  JSTQB 訳注) 日本では「テスト技術者資格制度 Advanced Level シラバス日本語版概要 Version 2012」として発行されている。
- [ISTQB_ALTA_SYL]: ISTQB Advanced Level Test Analyst Syllabus, Version 2012
  JSTQB 訳注) 日本では「テスト技術者資格制度 Advanced Level シラバス日本語版テストアナリスト Version 2012」として発行されている。
- [ISTQB_FL_SYL]: ISTQB Foundation Level Syllabus, Version 2011
  JSTQB 訳注) 日本では「テスト技術者資格制度 Foundation Level シラバス日本語版 Version 2011」として発行されている。
- [ISTQB_GLOSSARY]: ISTQB Glossary of Terms used in Software Testing, Version 2.2, 2012
  JSTQB 訳注)日本では「テスト技術者資格制度　ソフトウェアテスト標準用語集 日本語版 Version 2.2」として発行されている。

7.3 書籍

[Bass03]: Len Bass, Paul Clements, Rick Kazman "Software Architecture in Practice (2nd edition)", Addison-Wesley 2003] ISBN 0-321-15495-9

JSTQB 訳注）日本では「実践ソフトウェアアーキテクチャ」（日刊工業新聞社、2005 年）として発行されている。
[Bath08]: Graham Bath, Judy McKay, "The Software Test Engineer's Handbook", Rocky Nook, 2008, ISBN 978-1-933952-24-6
[Beizer90]: Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3
JSTQB 訳注）日本では「ソフトウェアテスト技法」（日経 BP 社、1994 年）として発行されている。
[Beizer95]: Boris Beizer, "Black-box Testing", John Wiley & Sons, 1995, ISBN 0-471-12094-4
JSTQB 訳注）日本では「実践的プログラムテスト入門 ソフトウェアのブラックボックステスト」（日経 BP, 1997 年）として発行されている。
[Buwalda01]: Hans Buwalda, "Integrated Test Design and Automation" Addison-Wesley Longman, 2001, ISBN 0-201-73725-6
[Copeland03]: Lee Copeland, "A Practitioner's Guide to Software Test Design", Artech House, 2003, ISBN 1-58053-791-X
JSTQB 訳注）日本では「はじめて学ぶソフトウェアのテスト技法」(日経 BP, 2005 年)として発行されている。
[Gamma94]: Design Patterns:Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-201-63361-2
JSTQB 訳注）日本では「オブジェクト指向における再利用のためのデザインパターン」（ソフトバンククリエイティブ、1999 年）として発行されている。
[Jorgensen07]: Paul C. Jorgensen, "Software Testing, a Craftsman's Approach third edition", CRC press, 2007, ISBN-13:978-0-8493-7475-3
[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; "Lessons Learned in Software Testing"; Wiley, 2002, ISBN:0-471-08112-4
JSTQB 訳注）日本では「ソフトウェアテスト 293 の鉄則」（日経 BP 社、2003 年）として発行されている。
[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, "TMap Next for result-driven testing"; UTN Publishers, 2006, ISBN:90-72194-79-9
[McCabe76]: Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.PP 308-320
[NIST96]: Arthur H. Watson and Thomas J. McCabe, "Structured Testing:A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, Prepared under NIST Contract 43NANB517266, September 1996.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, "The Web-Testing Handbook", STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07]: Mark Utting, Bruno Legeard, "Practical Model-Based Testing:A Tools Approach", Morgan-Kaufmann, 2007, ISBN:978-0-12-372501-1
[Whittaker04]: James Whittaker and Herbert Thompson, "How to Break Software Security", Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0
[Wiegers02]: Karl Wiegers, "Peer Reviews in Software:A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0
JSTQB 訳注）日本では「ピアレビュー」（日経 BP 社、2004 年）として発行されている。

7.4 その他の参照元

以下は、インターネットで参照できる情報を示している。これらの参照については、本 Advanced Level シラバス発行時にチェックしているが、すでに参照できなくなっていても、ISTQB はその責を負わない。

[Web-1]: www.testingstandards.co.uk
[Web-2]: http://www.nist.gov NIST National Institute of Standards and Technology

[Web-3]: http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4]: http://portal.acm.org/citation.cfm?id=308798
[Web-5]: http://www.processimpact.com/pr_goodies.shtml
[Web-6]: http://www.ifsq.org
[Web-7]: http://www.W3C.org

第 4 章: [Web-1]、[Web-2]
第 5 章: [Web-3]、[Web-4]、[Web-5]、[Web-6]
第 6 章: [Web-7]

8. 索引

ふ

へ

ほ

ま

め

り

れ

ろ

わ