ISTQB® 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

v4.0.J02

International Software Testing Qualifications Board

著作権について

改訂の経緯

◆ ISTQB®

バージョン	日付	備考
v4.0	2021/06/30	v4.0 版の GA リリース
v4.0	2021/04/28	ベータレビューのフィードバックを元にドラフトを更新
2021 v4.0 ベータ版	2021/03/01	アルファレビューのフィードバックを元にドラフトを更新
2021 v4.0 Alpha	2020/12/07	アルファレビュー用ドラフトを更新 • 全体的なテキストの改善 • K3 TTA-2.6.1（2.6 パステスト）に関連するサブセクションを削除し、LO を削除 • K2 TTA-3.2.4 (3.2.4 コールグラフ)に関連するサブセクションを削除し、LO を削除 • TTA-3.2.2 (3.2.2 データフロー解析)に関連するサブセクションを書き換え、K3 とした • TTA-4.4.1 及び TTA-4.4.2（4.4. 信頼性テスト）に関連するセクションを更新 • TTA-4.5.1 及び TTA-4.5.2（4.5 性能テスト）に関連するセクションを更新 • 運用プロファイルに関する 4.9 項を追加 • TTA-2.8.（2.7 ホワイトボックステスト技法の選択）に関連するセクションを更新 • TTA-3.2.1 に循環的複雑度を含むように更新（試験問題への影響はなし）。 • TTA-2.4.1（MC/DC）を他のホワイトボックス LO と整合するように更新（試験問題への影響はない）。
2019 v1.0	2019/10/18	2019 年版の GA リリース
2012	2012/10/19	2012 年版の GA リリース

◆ JSTQB®

バージョン	日付	備考
Version2024v4.0.J02	2026/06/27	軽微な日本語訳の修正
Version2024v4.0.J01	2024/07/14	v4.0 版の日本語版を公開

目次

謝辞

本ドキュメントの 2019 年版は、International Software Testing Qualifications Board Advanced Level Working Group のコアチームである、Graham Bath（vice-chair）、Rex Black、Judy McKay、Kenji Onoshi、Mike Smith（chair）、Erik van Veenendaal が執筆した。

本シラバス 2019 年版のレビュー、コメント、投票に参加したのは以下の方々である。

Dani Almog
Armin Born
Wim Decoutere
Peter Foldhazi
Jan Giesen
Tamás Horváth
Attila Kovács
Marton Matyas
Petr Neugebauer
Meile Posthuma
Adam Roman
Benjamin Timmermans
Andrew Archer
Sudeep Chatterjee
Klaudia Dusser-Zieger
David Frei
Attila Gyuri
N. Khimanand
Claire Lohr
Judy McKay
Ingvar Nordström
Stuart Reid
Jan Sabak
Stephanie van Dijck
Rex Black
Tibor Csöndes
Melinda Eckrich-Brájer
Karol Frühauf
Matthias Hamburg
Jan te Kock
Rik Marselis
Dénes Medzihradszky
Pálma Polyák
Lloyd Roden
Péter Sótér
Paul Weymouth

本ドキュメントは、International Software Testing Qualifications Board Advanced Level Working Group のコアチームである、Armin Born、Adam Roman、Stuart Reid が執筆した。

本ドキュメントの更新である v4.0 は、International Software Testing Qualifications Board Advanced Level Working Group のコアチームである、Armin Born、Adam Roman、Christian Graf、Stuart Reid が執筆した。

本シラバスの更新版である v4.0 のレビュー、コメント、投票に参加したのは以下の方々である。

Adél Vécsey-Juhász
Ágota Horváth
Benjamin Timmermans
Erwin Engelsma
Gary Mogyorodi
Geng Chen
Gergely Ágnecz
Jane Nash
Lloyd Roden
Matthias Hamburg
Meile Posthuma
Nishan Portoyan
Joan Killeen
Ole Chr. Hansen
Pálma Polyák
Paul Weymouth
Péter Földházi Jr.
Rik Marselis
Sebastian Małyska
Tal Pe'er
Wang Lijuan
Zuo Zhenlei

コアチームは、レビューチームとナショナルボードの方々の提案と意見に感謝したい。

このドキュメントは、2021 年 6 月 30 日の ISTQB®の総会で正式に発行された。

日本語訳については、Japan Software Qualifications testing Board メンバー及び以下の日本語翻訳ワーキンググループメンバーにより行われた。
日本語翻訳ワーキンググループメンバー：

湯本　剛　　（株式会社 ytte Lab／フリー株式会社／JSTQB 技術委員）
苅田　蓮　　（フリー株式会社）
高橋　完爾　（バルテス株式会社）
角田　俊　　（株式会社マネーフォワード）
藤原　考功　（株式会社ユーザベース）

0. 本シラバスの紹介

0.1 本シラバスの目的

本シラバスは、テクニカルテストアナリスト向けの国際ソフトウェアテスト資格 Advanced Level 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、各国語への翻訳及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及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 ソフトウェアテスト向けテスト技術者資格制度 Advanced Level

Advanced Level Core 資格認定は、次の役割に関連する 3 つの独立したシラバスで構成し ている。

- テストマネージャー
- テストアナリスト
- テクニカルテストアナリスト

ISTQB® テクニカルテストアナリスト上級レベルの概要は、別文書[CTAL_TTA_OVIEW]で、以下の内容を含んでいる。

- ビジネスアウトカム
- ビジネスアウトカムと学習の目的の間のトレーサビリティを示すマトリクス
- 概要

0.3 試験のための学習の目的と知識レベル

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。

本シラバスでは各章の先頭で、K2 レベル、K3 レベル、及び K4 レベルの「学習の目的」を以下の分類にて示している。

- K2：理解
- K3：適用
- K4：分析

0.4 経験への期待

テクニカルテストアナリストの学習目的の中には、以下の分野での基本的な経験があることを前提としているものがある。

- 一般的なプログラミングの概念
- システムアーキテクチャの一般的な概念

0.5 Advanced Level テクニカルテストアナリスト試験

Advanced Level テクニカルテストアナリストの認定資格試験は本シラバスに基づく。試験問題に対する解答は、本シラバスの複数の節にまたがる情報に基づくことがある。本シラバスのすべての節は、「イントロダクション」と付録 A を除いて試験対象である。標準、文献、及び他の ISTQB®シラバスを情報源としているが、それらに関して本シラバス自体の中で要約されている以上の内容は試験対象ではない。

試験の形式は多肢選択式である。問題の数は 45 である。総得点の 65%以上を獲得した場合に合格となる。

試験は、認定トレーニングコースの一部として、または（例えば試験センターや公的試験で）独立して実施しても良い。認定トレーニングコースの受講完了は試験のための前提条件ではない。

0.6 認定資格試験の受験資格

Advanced Level テクニカルテストアナリストの認定資格試験を受験するには、Certified Tester Foundation Level 認定資格を取得している必要がある。

0.7 コース認定

ISTQB®のメンバー委員会にて、教育コースの教材が本シラバスに従っている教育機関を認定する。教育機関はメンバー委員会または認定を行う機関から認定ガイドラインを入手しなければならない。教育コースがシラバスに従っていると認定されると、教育コースの一部として ISTQB®の試験を実施することができる。

0.8 シラバスの詳細レベル

本シラバスの詳細レベルは国際的に一貫した教育と試験を可能にする。このゴールを達成するために本シラバスは以下の内容で構成される。

- 全般的な教育内容の目的（Advanced Level テストアナリストの意図について説明）
- アイテムのリスト（想起できる必要のあるアイテム）
- 各知識領域の学習の目的（達成しなければならない知識レベルの学習の成果について説明）
- 教えるべき主要なコンセプトの説明（承認された文献や標準を情報源として含む）

本シラバスの内容は全知識領域の説明ではない。詳細レベルは、Advanced Level のトレーニングコースでカバーされることを示している。本シラバスは、ソフトウェア開発ライフサイクルを含むすべてのソフトウェアプロジェクトに適用できる資料に重点を置いて説明している。本シラバスでは、特定のソフトウェア開発モデルに関連する個別の学習の目的は取り上げていない。ただし、これらのコンセプトをアジャイルソフトウェア開発、他の種類のイテレーティブ/インクリメンタル開発モデル、及びシーケンシャル開発モデルで、これらの概念をどのように適用するかについて説明している。

0.9 本シラバスの構成

6 つの章で構成され、すべて試験対象である。各章のトップレベルの見出しには、その章の最低時間が指定されており、章より下のレベルの時間幅は指定されていない。認定トレーニングコースでは、シラバスでは最低 20 時間の講義が要求され、6 つの章で以下のように配分する。

- 第 1 章：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）
- 第 2 章：ホワイトボックステスト技法（300 分）
- 第 3 章：静的解析と動的解析（180 分）
- 第 4 章：テクニカルテストのための品質特性 （345 分）
- 第 5 章：レビュー（165 分）
- 第 6 章：テストツールと自動化（180 分）

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

キーワード
プロダクトリスク、プロジェクトリスク、リスクアセスメント、リスク識別、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

1.2 リスクベースドテストのタスク

TTA-1.2.1 (K2) テクニカルテストアナリストが通常考慮する必要のある一般的なリスク要因を要約する。

TTA-1.2.2 (K2) テスト活動におけるリスクベースドアプローチの中で、テクニカルテストアナリストの活動を要約する。

1.1 はじめに

テストマネージャーは、リスクベースのテスト戦略を確立し、マネジメントする全体的な責任を有する。テストマネージャーは通常、リスクベースドアプローチが正しく実装されるように、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は、プロジェクトのテストマネージャーによって確立されたリスクベースドテストのフレームワークの中で仕事をする。セキュリティ、システムの信頼性、性能に関するリスクなど、プロジェクトに内在する技術的なプロダクトリスクに関する知識を提供する。また、性能、信頼性、セキュリティテストのためのテスト環境の取得やセットアップなど、テスト環境に関連するプロジェクトリスクの識別と対応に貢献する必要がある。

1.2 リスクベースドテストのタスク

テクニカルテストアナリストは、以下のような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、新たに発生するリスクや優先度の変化に対応し、リスクの状況を定期的に評価・伝達するために、プロジェクト期間中に繰り返し実施する。

1.2.1 リスク識別

リスク識別プロセスでは、可能な限り広範なステークホルダーに参加を呼びかけることで、重要なリスクを可能な限り多く検出できる。テクニカルテストアナリストは、優れた技術スキルを持っているため、専門家へのインタビューを実施したり、同僚とブレーンストーミングを行ったりすること、そして、彼らの経験を分析して、プロダクトリスクが存在する可能性が高い領域を特定するのに特に適している。特に、テクニカルテストアナリストは、開発者、アーキテクト、運用エンジニア、プロダクトオーナー、ローカルサポートオフィス、技術エキスパート、サービスデスク技術者などの他の関係者と密接に協力し、プロダクトやプロジェクトに影響を与える技術的リスクの領域を識別する。他のステークホルダーを巻き込むことで、すべての見解が考慮されるようになる。通常、他のステークホルダーを巻き込むのは、テストマネージャーが手助けする。

テクニカルテストアナリストが識別する可能性のあるリスクは、通常、本シラバス第 4 章に記載されているプロダクト品質特性 [ISO 25010]に基づいている。

1.2.2 リスクアセスメント

リスク識別が適切なリスクをできるだけ多く識別することであるのに対し、リスクアセスメントは、識別したリスクを調査し、各リスクを分類し、リスクの可能性と影響を決定することである。

プロダクトリスクの可能性は、通常、テスト対象のシステムで故障が発生する確率と解釈される。テストアナリストが、問題が発生した場合の潜在的なビジネスへの影響を理解することに貢献するのに対し、テクニカルテストアナリストは、各技術的なプロダクトリスクの確率を理解することに貢献する。

プロジェクトリスクが顕在化し、問題として表出すると、プロジェクト全体の成功に影響を与える可能性がある。典型的には、以下のような一般的なプロジェクトリスク要因を考慮する必要がある。

- 技術的な要件に関するステークホルダー間の対立
- 開発組織の地理的分散に起因するコミュニケーションの問題
- ツール、及び技術（関連スキルを含む）
- 時間、リソース、マネジメントのプレッシャー
- 早期からの品質保証活動の欠如
- 技術的な要件の高い変更率

プロダクトリスクが問題になると、欠陥の数が多くなる場合がある。一般的に、以下のような一般的なプロダクトリスク要因を考慮する必要がある。

- 技術の複雑性
- コードの複雑度
- ソースコードの変更量（追加、削除、修正）
- 技術的な品質特性に関連する欠陥が大量に検出されること（欠陥の履歴）
- インターフェースと統合の技術的な問題

利用可能なリスク情報を考慮して、テクニカルテストアナリストは、テストマネージャーによって確立されたガイドラインに従って、リスクの可能性の初期値を提示する。初期値は、すべてのステークホルダーの見解を考慮したときに、テストマネージャーが修正することがある。リスクの影響は、通常テストアナリストが決定する。

1.2.3 リスク軽減

プロジェクト期間中、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対応方法に影響を与える。これには、一般的に次のようなことが含まれる。

- 高リスク領域に存在するリスクに対応するテストケースを設計し、残存リスクの評価を支援する。
- 設計したテストケースを実行することと、テスト計画に記載した適切な軽減策や不測の事態への対応策を実行することにより、リスクを減らす
- プロジェクトの進展に伴い収集した追加情報に基づいてリスクを評価し、その情報を使ってリスクの可能性を減らすための軽減策を実装する

テクニカルテストアナリストは、多くの場合、セキュリティや性能などの分野のスペシャリストと協力して、リスク軽減策やテスト戦略の要素を定義する。追加の情報は、ISTQB® Specialist シラバス（セキュリティテストシラバス［CT_SEC_SYL］や性能テストシラバス［CT_PT_SYL］など）から得ることができる。

2. ホワイトボックステスト技法 - 300 分

キーワード

API テスト, 不可分条件、制御フロー、デシジョンテスト、改良条件判定テスト、複合条件テスト、安全度水準、ステートメントテスト、ホワイトボックステスト技法

ホワイトボックステスト技法の学習の目的

2.2 ステートメントテスト

TTA-2.2.1 (K3) 定義したカバレッジレベルを達成するためにステートメントテストを適用し、テスト対象を考慮してテストケースを設計する。

2.3 デシジョンテスト

TTA-2.3.1 (K3) 定義したカバレッジレベルを達成するために、デシジョンテスト技法を適用し、テスト対象を考慮してテストケースを設計する。

2.4 改良条件判定テスト

TTA-2.4.1 (K3) 完全な改良条件判定カバレッジ（MC/DC）を達成するために、改良条件判定テスト技法を適用し、テスト対象を考慮してテストケースを設計する。

2.5 複合条件テスト

TTA-2.5.1 (K3) 定義したカバレッジレベルを達成するために、複合条件テスト技法を適用し、テスト対象を考慮してテストケースを設計する。

2.6 ベースパステスト（*本シラバスのバージョン 4.0 から削除された。*）

TTA-2.6.1 は、本シラバスのバージョン 4.0 から削除された。

2.7 API テスト

TTA-2.7.1 (K2) API テストの適用範囲と発見する欠陥の種類を理解する。

2.8 ホワイトボックステスト技法の選択

TTA-2.8.1 (K4) プロジェクトの状況に応じて適切なホワイトボックステスト技法を選択する。

2.1 はじめに

この章では、ホワイトボックステスト技法について説明する。これらの技法は、コードや、ビジネスプロセスのフローチャートのような、制御フローを持つその他の構造へ適用する。

それぞれのテスト技法は、テストケースを体系的に導き出し、構造の特定の側面に焦点を当てることを可能にする。テスト技法から生成したテストケースは、目的として設定したカバレッジ基準を満たし、それに対して測定をする。完全なカバレッジ（つまり 100%）を達成することは、すべてのテストが完了したことを意味するのではなく、使用しているテスト技法が、対象の構造に対して有用なテストをこれ以上示唆しなくなったことを意味する。

テストの入力は、テストケースがコードの特定の部分（ステートメントや判定結果など）を実行するように生成される。コードの特定部分を実行するテストの入力を決定することは、難しい場合がある。特に、実行対象のコードが長い制御フローのサブパスの末尾にあり、複数の判定を持つ場合に難しいことがある。期待する結果は、要件仕様書や設計仕様書、あるいは別のテストベースなど、この構造の外部の情報源に基づいて特定される。

このシラバスでは、次の技法を深く見ていく。
- ステートメントテスト
- デシジョンテスト
- 改良条件判定テスト
- 複合条件テスト
- API テスト

Foundation シラバス[CTFL_SYL]では、ステートメントテストとデシジョンテストが紹介されている。ステートメントテストは、コード内の実行可能なステートメントを実行することに焦点を当てる一方、デシジョンテストは、判定結果に着目する。

上記の改良条件判定、複合条件といった技法は、複数の条件を含む判定文に基づくものであり、同様の種類の欠陥を発見するものである。判定文がどんなに複雑でも、TRUE か FALSE のどちらかに評価され、それによってコード内のパスが決定される。判定文が期待通りに評価されず、意図したパスが通らない場合に欠陥が検出される。

[ISO 29119]を参照することで、これらの技法の仕様と実例をより詳しく知ることができる。

2.2 ステートメントテスト

ステートメントテストは、コード内の実行可能なステートメントに着目する。カバレッジは、テストによって実行されたステートメントの数をテスト対象の実行可能なステートメントの総数で割った値で測定し、通常はパーセントで表示する。

適用
ステートメントカバレッジを完全に達成することは、テストされるすべてのコードに対して最低限考慮されるべきだが、実際には必ずしも可能ではない。

制限/注意事項

ステートメントカバレッジを完全に達成することは、テストされるすべてのコードに対して最低限考慮されるべきだが、利用可能な時間、そして/または労力の制約により、実際には常に可能とは限らない。ステートメントカバレッジが高い割合であっても、コードのロジックにある特定の欠陥を検出しない場合がある。多くの場合、到達不可能なコードのために、ステートメントカバレッジを 100%にすることは不可能である。到達不能コードは一般に良いプログラミング手法とは考えられていないが、例えば switch ステートメントがデフォルトのケースを持たなければならないのに、すべての可能なケースが明示的に処理されている場合に発生する可能性がある。

2.3 デシジョンテスト

デシジョンテストは、コード内の判定結果に着目する。そのために、テストケースは、判定ポイントからの制御フローを追う（例えば、IF ステートメントでは、真の結果に対して 1 つ、偽の結果に対して 1 つの制御フローがあり、CASE ステートメントでは、いくつかの可能な結果があり、LOOP ステートメントでは、ループ条件の真の結果に対して 1 つ、偽の結果に対して 1 つの制御フローがある）。

カバレッジは、テストが用いる判定結果の数をテスト対象の判定結果の総数で割ったものとして測定し、通常はパーセンテージで表現する。1 つのテストケースで複数の判定結果を評価することがあることに注意する。
後述の改良条件判定テスト技法や複合条件テスト技法と比較して、デシジョンテストでは、判定の内部構造の複雑度に関係なく、判定全体を考慮し、真と偽の結果のみを評価する。

ブランチテストは、デシジョンテストと同じ意味で使われることが多いが、それは、すべてのブランチをカバーすることと、すべての判定結果をカバーすることが、同じテストで実現できるからである。ブランチテストは、コード内のブランチをテストする。ブランチは、通常、制御フローグラフのエッジとみなす。判定がないプログラムの場合、上記の判定カバレッジの定義では、いくらテストを実行してもカバレッジは 0/0 となり、未定義となるが、入り口から出口点までの分岐（入り口と出口点が 1 つずつあると仮定）は、ブランチカバレッジが 100%達成されることになる。この 2 つの測定の違いに対応するため、ISO29119-4 では、デシジョンカバレッジ 100%を達成するために、デシジョンがないコードに対して少なくとも 1 つのテストを実行することを要件としており、ほぼすべてのプログラムにおいてデシジョンカバレッジ 100%とブランチカバレッジ 100%が同等となるようにしている。安全関連システムのテストに使用されるものを含め、カバレッジ尺度を提供する多くのテストツールは、同様のアプローチを採用している。

適用
このレベルのカバレッジは、テストされるコードが重要、あるいはクリティカルである場合に考慮されるべきである（安全関連システムについては 2.8.2 節の表を参照のこと）。この技法は、コードだけでなく、ビジネスプロセスモデルのような判定ポイントを含むモデルにも使用することができる。

制限/注意事項
デシジョンテストは、複数の条件を持つ判定がどのように行われるかの詳細を考慮しないため、判定結果の組み合わせによる欠陥を検出できない場合がある。

2.4 改良条件判定テスト（MC/DC）

デシジョンテストが判定全体を考慮し、真と偽の結果を評価するのに比べ、改良条件判定テスト（MC/DC）は判定が複数の条件を含む場合にどのように構成されているかを考慮する（判定が 1 つの不可分条件のみからなる場合は、単にデシジョンテストとなる）。

各判定文は、1 つ以上の不可分条件から構成される。それぞれの条件はブール値として評価される。これらは論理的に結合され、判定の結果を決定する。この技法は、各不可分条件が独立して正しく全体の判定結果に影響することをチェックする。

この技法は、複数の条件を含む判定がある場合に、ステートメントカバレッジやデシジョンカバレッジよりも強いレベルのカバレッジを提供する。N 個の一意で互いに独立した不可分条件を仮定すると、MC/DC は、通常、判定を N+1 回実行することで達成される。改良条件判定テストでは、単一の不可分条件の結果の変化が独立して判定結果に影響を与えることを示すテストのペアが必要となる。1 つのテストケースで複数の条件の組み合わせを実行することがあるため、MC/DC を達成するために N+1 回の別々のテストケースを実行する必要は必ずしもないことに注意する。

適用
航空宇宙産業や自動車産業など、セーフティクリティカルなシステムで使用されている技術である。故障が大惨事を引き起こす可能性のあるソフトウェアをテストするときに使われる。改良条件判定テストは、デシジョンテストと複合条件テスト（テストする組み合わせが多いため）の中間的な位置づけとして妥当なものであると言える。デシジョンテストよりも厳密だが、判定に不可分条件がいくつかある場合、複合条件テストよりもはるかに少ないテスト条件で実施することができる。

制限/注意事項
複数の条件を持つ判定において、同じ変数が複数回出現する場合、MC/DC の達成は複雑になることがある。このような場合、条件は「結合」している可能性がある。判定によっては、1 つの条件の値を変化させて、それだけで判定結果を変化させることができない場合がある。この問題に対処するための 1 つのアプローチは、改良条件判定テストを使用して、結合されていない不可分条件のみをテストするように指定することである。もう 1 つの方法は、結合が発生する各判定を解析することである。

一部のコンパイラやインタプリタは、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。例えば、判定「A and 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わることはないため、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。通常、テスト時に短絡評価を行わないようにコンパイラを設定することが可能だが、テストコードとデリバリーされるコードが同一でなければならないセーフティクリティカルなアプリケーションでは、これが許されない場合がある。

2.5 複合条件テスト

まれに、判定を含む不可分条件の可能な組み合わせをすべてテストすることが要求される場合がある。このレベルのテストは、複合条件テストと呼ばれる。N 個のユニークで相互に独立した不可分条件を仮定すると、2^N 回実行することにより、判定に対する完全な複合条件カバレッジを達成することができる。1 つのテストケースで複数の条件組み合わせを実行できるため、100%の複合条件カバレッジを達成するために、2^N のテストケースを実行する必要は必ずしもないことに注意する。

カバレッジは、テスト対象の全判定において、実行された不可分条件の組み合わせの数として測定され、通常はパーセンテージで表す。

適用
この技法は、リスクの高いソフトウェアや、長時間クラッシュすることなく確実に動作することが求められる組込みソフトウェアなどのテストに用いられる。

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き出すことができるため、このカバレッジレベルは容易に決定することができる。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、判定に複数の不可分条件がある状況では、改良条件判定テストがより現実的になる。

コンパイラが短絡評価を行う場合、不可分条件に対して行われる論理演算の順序やグループ化によって、実行可能な条件の組み合わせの数が少なくなることが多い。

2.6 ベースパステスト

この章は、このシラバスのバージョン v4.0 から削除された。

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、プログラムが他のソフトウェアシステムを呼び出すための定義されたインターフェースで、リモートリソースへのアクセスなどのサービスを提供する。典型的なサービスとしては、Web サービス、エンタープライズ・サービス・バス、データベース、メインフレーム、Web UI などがある。

API テストは、テスト技法というよりも、テストのやり方の一種である。ある点では、API テストはグラフィカルユーザーインターフェース（GUI）のテストとよく似ている。焦点は、入力値と返されたデータの評価にある。

API を扱う際には、しばしばネガティブテストが重要になる。API を使用して自分のコードの外部のサービスにアクセスするプログラマーは、API インターフェースを意図しない方法で使用しようとする可能性がある。つまり、誤った運用を避けるためには、強固なエラー処理が不可欠なのである。API は他の API と組み合わせて使われることが多く、また、1 つのインターフェースに複数のパラメーターが含まれ、それらの値がさまざまに組み合わされる可能性があるため、多くのインターフェースの組み合わせテストが必要となる場合がある。

API は疎結合であることが多く、その結果、トランザクションの損失やタイミングの不具合が発生する可能性が非常に高くなる。そのため、リカバリー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 インターフェースを提供する組織は、すべてのサービスが非常に高い可用性を持つことを保証しなければならない。これには、API パブリッシャーによる厳格な信頼性テストと、インフラのサポートがしばしば必要となる。

適用
API テストは、システムオブシステムズのテストとしてより重要性を増している。システムオブシステムズは、個々のシステムが分散化したり、一部の作業を他のプロセッサーにオフロードする方法としてリモート処理を使用したりするためである。その例としては、以下のようなものがある。

- システムコール
- サービス指向アーキテクチャ（SOA）
- リモートプロシジャーコール（RPC）
- Web サービス

ソフトウェアのコンテナ化とは、ソフトウェアプログラムを複数のコンテナに分割し、上記のような仕組みを用いて相互に通信させることである。API テストは、これらのインターフェースもターゲットにする必要がある。

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が専用ツールを使用する必要がある。API には通常、直接的なグラフィカルインターフェースがないため、初期環境のセットアップ、データの変換、API の呼び出し、結果の判定にツールが必要になる場合がある。

カバレッジ
API テストはテストのやり方の一種の説明であり、特定のカバレッジレベルを示すものではない。最低限、API テストは現実的な入力値と、例外処理をチェックするために予期しない入力の両方で API を呼び出すことを含むべきである。より徹底した API テストは、呼び出し可能なエンティティが少なくとも一度は確認できること、または可能なすべての関数を少なくとも一度は呼び出すことができることを確認する。

REST(Representational State Transfer)は、アーキテクチャスタイルの 1 つである。RESTful Web サービスは、リクエストしたシステムが、統一されたステートレス操作のセットを使用して Web リソースにアクセスすることを可能にする。ソフトウェアの統合のためのデファクトスタンダードである RESTful Web API には、いくつかのカバレッジ基準が存在する[Web-7]。それらは、入力カバレッジ基準と出力カバレッジ基準の 2 つのグループに分けることができる。その中で、入力基準は、すべての可能な API 操作の実行、すべての可能な API パラメーターの使用、及び API 操作のシーケンスのカバレッジを要求することがある。その中で、出力基準は、すべての正しいステータスコードと誤ったステータスコードの生成、及びすべてのプロパティ（またはすべてのプロパティタイプ）を示す、リソースを含むレスポンスの生成を要求することがある。

欠陥の種類
API のテストによって発見される欠陥のタイプは、非常に多岐にわたる。インターフェースの問題は一般的であり、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損失、トランザクションの重複、例外処理の問題も同様に一般的である。

2.8 ホワイトボックステスト技法の選択

選択されたホワイトボックステスト技法は、通常、テスト技法を適用することによって達成する要求カバレッジレベルを明示する。例えば、100%のステートメントカバレッジを達成する要件は、通常、ステートメントテストを使用する。しかし、通常、ブラックボックステスト技法を、最初に適用し、その後、カバレッジを測定し、要求したホワイトボックスのカバレッジレベルを達成しなかった場合にのみ、ホワイトボックステスト技法を使用する。状況によっては、ホワイトボックステストは、あまり形式的には使用せず、カバレッジを向上する必要がある場所の指標を提供するために使用する（例えば、ホワイトボックスのカバレッジレベルが特に低い場所に追加のテストを作成する）。そのような非形式的なカバレッジ測定には、通常、ステートメントテストで十分である。

ホワイトボックスカバレッジの要求レベルを指定する場合は、100%で指定するのが良い。その理由は、カバレッジレベルを低く指定した場合、テストを実施しない部分は、テストが最も困難な部分であり、通常、最も複雑でエラーが発生しやすい部分であることを意味するからである。そのため、例えば 80%のカバレッジを要求し達成することは、検出可能な欠陥の大部分を含むコードをテストしないまま放置することを意味する場合がある。このため、ホワイトボックスのカバレッジ基準を標準として指定する場合、ほとんどの場合 100%で指定する。高いカバレッジレベルの定義は、カバレッジ

レベルが達成できないことがある。しかし、ISO 29119-4 で規定されているものでは、実行不可能なカバレッジアイテムを計算から除外できるため、カバレッジ 100%を達成可能な目標にすることができる。

テスト対象に対して要求されるホワイトボックスカバレッジを指定する場合、カバレッジ基準を 1 つだけ指定することが必要となる。（例えば、100%のステートメントカバレッジと 100%の MC/DC の両方を要求する必要はない）。終了基準を 100%とした場合、カバレッジ基準が他のカバレッジ基準を包含することを示すため、階層を包含するいくつかの終了基準と関連付けることが可能である。すべてのコンポーネントとそれらの仕様において、1 つ目のカバレッジ基準を満たすための一連のテストケースが 2 つ目の基準も満たす場合、あるカバレッジ基準は他のカバレッジ基準を包含すると言える。例えば、ブランチカバレッジはステートメントカバレッジを包含する。なぜなら、（100%になる）ブランチカバレッジを達成すれば、100%のステートメントカバレッジを保証することになるからである。このシラバスで扱うホワイトボックステスト技法では、ブランチカバレッジとデシジョンカバレッジはステートメントカバレッジを包含し、MC/DC はデシジョンカバレッジとブランチカバレッジを包含し、複合条件カバレッジは MC/DC を包含していると言える（ブランチとデシジョンカバレッジが 100%であるときに同じものであると考えれば、互いに含まれることになる）。

システムに対して達成すべきホワイトボックスのカバレッジレベルを決める際は、システムのさまざまな部分に対してそれぞれ異なったレベルを定義するのがごく普通であることに留意する。これは、システムの異なる部分がリスクに与える影響が異なるためである。例えば、アビオニクスシステムにおいて、機内エンターテイメントに関連するサブシステムは、フライトコントロールに関連するサブシステムよりも低いリスクレベルが割り当てられることがある。インターフェースのテストをすることは、あらゆる種類のシステムに共通であり、通常、安全関連システムのすべての安全度水準において要求される（安全度水準については、2.8.2 節参照）。API テストに要求されるカバレッジのレベルは、通常、関連するリスクに基づいて増加する（例えば、リスクのレベルが高いパブリックなインターフェース関連には、より厳格な API テストが必要になることがある）。

どのホワイトボックステスト技法を使用するかの選択は、一般的にテスト対象の性質とその認識されるリスクに基づく。もしテスト対象が安全関連（すなわち、故障が人や環境に害を及ぼす可能性がある）と考えられる場合、規制としての標準が適用され、要求されるホワイトボックスカバレッジレベルを定義する（2.8.2 節を参照）。テスト対象が安全性に関係しない場合、どのホワイトボックスカバレッジレベルを達成するかの選択は、より主観的になるが、2.8.1 節で説明しているように、認識されたリスクに大きく基づくべきである。

2.8.1 安全性に関連しないシステム

安全性に関係しないシステムに対してホワイトボックステスト技法を選択する場合、以下の要素（優先度は不問）を一般的に考慮する。

- 契約 - 契約上、特定のレベルのカバレッジを達成することが要件となっている場合、このカバレッジレベルを達成できない場合、契約違反となる可能性がある
- 顧客 - 例えばテスト計画の一部として、顧客が特定のカバレッジレベルを要求した場合、このカバレッジレベルを達成しないと、顧客との間で問題が発生する可能性がある
- 規制の標準 - 一部の産業分野（例えば、金融）では、要求されるホワイトボックスのカバレッジ基準を定義する規制の標準をミッションクリティカルシステムへ適用する。安全関連システムの規制の基準のカバレッジについては、2.8.2 節を参照のこと
- テスト戦略 - 組織のテスト戦略にホワイトボックスコードカバレッジの要件が規定されている場合、組織の戦略に沿わなければ、上層部から指摘されるリスクがある

- コーディングスタイル - 判定の中に複数条件がないようなコードの書き方であれば、MC/DC や複数条件カバレッジなどのホワイトボックスカバレッジレベルを要件とするのは無駄である
- 過去の欠陥情報 - 特定のカバレッジレベルを達成する有効性に関する過去のデータが、このテスト対象に使用することが適切であると示唆する場合、利用可能なデータを無視することは危険である。そのようなデータは、プロジェクト、組織、業界内で利用可能であることに注意する
- スキルと経験 - テストを実施できるテスト担当者が、特定のホワイトボックス技法について十分な経験と熟練を積んでいない場合、その技法を選択したとしても、その技法を誤解する可能性があり、不必要なリスクをもたらす可能性がある
- ツール - ホワイトボックスのカバレッジは、カバレッジツールを使用することによってのみ、実際に測定することができる。もし、あるカバレッジの尺度をサポートするそのようなツールがない場合、その尺度を選択して達成することは、高いリスクレベルをもたらすことになる

テクニカルテストアナリストは、安全性に関連しないシステムに対してホワイトボックステストを選択する場合、適切なホワイトボックスカバレッジを推奨することに、安全性に関連するシステムよりも高い自由度を持つ。このような選択は、通常、認識されたリスクと、ホワイトボックステストによってこれらのリスクを処理するために必要なコスト、リソース、時間との間の妥協点である。状況によっては、他のソフトウェアテストアプローチや他の方法（例えば、異なる開発アプローチ）で実装する可能性がある他の処置がより適切な場合がある。

2.8.2 安全性に関連するシステム

テストされるソフトウェアが安全性に関連するシステムの一部である場合、達成すべき必要なカバレッジレベルを定義している規制の標準を通常使用しなければならない。このような標準は、通常、システムに対してハザード分析を行うことを要求し、その結果得られたリスクを用いて、システムのさまざまな部分に安全度水準を割り当てる。要求されるカバレッジのレベルは、安全度水準のそれぞれについて定義されている。

IEC 61508 (functional safety of programmable, electronic, safety-related systems [IEC 61508]) は、このような目的で使用される、包括的な国際標準である。理論的には、あらゆる安全関連システムに適用できるが、業界によっては特定の標準を制定している（例：ISO26262 [ISO 26262]は自動車システムに適用される）、また、いくつかの業界では独自の標準を作成している（例えば、DO-178C [DO-178C]は航空機用ソフトウェアに適用される）。ISO 26262 に関する追加情報は、ISTQB® Automotive Software Tester Syllabus [CT_AuT_SYL]に記載されている。

IEC 61508 では、4 つの安全度水準 (SIL) を定義しており、各レベルは、安全機能によってもたらされるリスク軽減の相対レベルとして定義している。リスク軽減の相対レベルは、認識されるハザードの頻度と重要度によって決められる。テスト対象が安全に関連する機能を果たす場合、故障のリスクが高いほど、そのテスト対象は高い信頼性を持つべきことを意味する。次の表は、SIL に関連する信頼性レベルを示している。なお、連続運用の場合の SIL 4 の信頼性レベルは、平均故障間隔（MTBF）が 10,000 年以上に相当するため、非常に高い信頼性である。

IEC 61508 SIL	連続運用（危険な故障が発生する確率/時）	オンデマンド（オンデマンドでの故障確率）
1	10^{-5} 未満 10^{-6} 以上	10^{-1} 未満 10^{-2} 以上

2	10^{-6} 未満 10^{-7} 以上	10^{-2} 未満 10^{-3} 以上
3	10^{-7} 未満 10^{-8} 以上	10^{-3} 未満 10^{-4} 以上
4	10^{-8} 未満 10^{-9} 以上	10^{-4} 未満 10^{-5} 以上

各 SIL に関連するホワイトボックスカバレッジレベルの推奨値を以下の表に示す。ある項目が「強く推奨する」と示されている場合、実際にはそのカバレッジレベルの達成は通常必須であると考えられる。これに対して、「推奨」とだけ表示されている項目については、多くの現場の技術者がこれをオプションとみなし、適切な根拠を示すことでその達成を回避している。従って、SIL 3 に割り当てられたテスト対象は、通常、100%のブランチカバレッジを達成するようにテストされる（包含の順番が示すように、100%のステートメントカバレッジも自動的に達成される）。

IEC 61508 SIL	100%ステートメントカバレッジ	100%ブランチカバレッジ	100%MC/DC
1	推奨	推奨	推奨
2	強く推奨	推奨	推奨
3	強く推奨	強く推奨	推奨
4	強く推奨	強く推奨	強く推奨

上記した IEC 61508 の SIL とカバレッジレベルに関する要件は、ISO 26262 とは異なり、また、DO-178C でもまた異なるので注意する。

3. 静的解析と動的解析 - 180 分

キーワード

不正、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、静的解析、ワイルドポインター

静的解析と動的解析の学習の目的

3.2 静的解析

TTA-3.2.1 (K3) 制御フロー解析を使用して、コードに制御フローの不正があるかどうかを検出し、サイクロマティックの複雑度を測定する。
TTA-3.2.2 (K3) データフロー解析を使用して、コードにデータフローの不正があるかどうかを検出する。
TTA-3.2.3 (K3) 静的解析を適用してコードの保守性を向上させる方法を提案する。

注：TTA-3.2.4 は本シラバスのバージョン v4.0 から削除した。

3.3 動的解析

TTA-3.3.1 (K3) 指定された目標を達成するために動的解析を適用する。

3.1 はじめに

静的解析（3.2 節参照）は、ソフトウェアを実行せずに行うテストの一形態である。ソフトウェアの品質は、その形式、構造、内容、または文書に基づいて、ツールまたは人によって評価する。このようにソフトウェアを静的に見ることで、テストケースの実行に必要なデータや事前条件を作成することなく、詳細な分析が可能になる。

静的解析とは別に、静的テスト技法にはさまざまな形式のレビューも含まれる。テクニカルテストアナリストに関連するものは、5 章でカバーしている。

動的解析（3.3 節参照）は、コードの実際の実行を要求し、コードが実行されているときにより簡単に検出できる欠陥（例えば、メモリリーク）を見つけるために使用する。動的解析は、静的解析と同様に、ツールに依存することもあれば人に依存することもある。実行中のシステムを監視して、メモリ使用量の急激な増加などの指標を確認する。

3.2 静的解析

静的解析の目的は、コードやシステムアーキテクチャの実際の、または潜在的な欠陥を検出することと、それらの保守性を向上させることである。

3.2.1 制御フロー解析

制御フロー解析は、通常はツールを使用し、制御フローグラフを用いて、プログラム中の手順を解析する静的な技法である。この技法を使用してシステムで検出できる不正は、適切な設計でないループ（例えば、複数の開始点を持つ、あるいは終了しないなど）、特定の言語における対象が曖昧な関数呼び出し、誤った処理順序、到達できないコード、呼び出されていない関数など多数存在する。

制御フロー解析は、サイクロマティック複雑度を求めるために用いることができる。サイクロマティック複雑度とは、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。

一般に，コンポーネントの複雑度を表す指標として，サイクロマティック複雑度を用いる。Thomas McCabe の理論[McCabe76]では，システムが複雑になればなるほど，保守が困難になり，欠陥が多く含まれるようになる。多くの研究で，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が指摘されている。複雑度が高いと測定されたコンポーネントは、複数のコンポーネントに分割するなど、リファクタリングの可能性を再検討する必要がある。

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術をカバーしている。制御フローパスに沿った各変数のライフサイクル（すなわち、宣言、定義、使用、破棄される場所）を調査することで、これらのアクションが順番通りに使用されていない場合に、潜在的な不正を特定することができる[Beizer90]。

一般的な技法として、変数の使い方を 3 つの不可分アクションの 1 つに分類している。

- 変数が定義、宣言、または初期化されたとき（例：x:=3）。
- 変数が使用または読み込まれたとき（例：if x > temp）
- 変数が削除、破棄、スコープ外に出たとき（例：text_file_1.close、ループ制御変数（i）ループから抜けるときなど）

このような一連の動作は、潜在的な不正の指標となる。

- 定義に続いて別の定義がある、または使用されずに削除する。
- 定義に対して削除がない（例えば、動的に割り当てられた変数のメモリリークの可能性）。
- 定義の前に使用、または削除がある
- 削除の後に使用、もしくは削除がある

プログラミング言語によっては、これらの不正の一部はコンパイラによって識別されるかもしれないが、データフローの不正を識別するためには、別の静的解析ツールが必要になる場合がある。例えば、使用することなく再定義することはほとんどのプログラミング言語で認められており、意図的にプログラムされている可能性がある。しかし、このような場合、データフロー解析ツールによって、不正の可能性があるのでチェックすべきというフラグを立てることがある。

変数のアクションシーケンスを決定するために制御フローパスを使用すると、実際には起こりえない潜在的な不正をレポートすることがある。例えば、一部のパスは実行時に変数に割り当てられた値に基づいてのみ決定するため、静的解析ツールは制御フローパスが実行可能であるかどうかを常に特定することはできない。また、解析対象がレコードや配列のような動的に変数を割り当てるデータ構造の一部である場合、ツールが特定することが困難なデータフロー解析の問題の分類が存在する。プログラム内の同時実行制御スレッド間で変数が共有されている場合も、データに対するアクションのシーケンスを予測することが困難になるため、静的解析ツールでの潜在的なデータフロー不正の特定が困難である。

データフロー解析が静的なテストであるのに対して、データフローテストは動的なテストで、プログラムコードの「定義使用ペア」を動作するためにテストケースが生成される。この定義使用ペアは、プログラム中の変数の定義と以降での使用との間の制御フローパスであるため、データフローテストは、データフロー解析と同じ概念を用いている。

3.2.3 静的解析による保守性の向上

静的解析は、コードやアーキテクチャ、Web サイトの保守性を向上させるために、さまざまな方法で適用できる。

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高くなる。

静的解析は、コーディング標準やガイドラインへの準拠を検証するために使用する。非準拠のコードが特定された場合、保守性を向上するためにコードを更新することができる。これらの標準やガイドラインは、命名規則、コメント、インデント、モジュール化など、必要なコーディングや設計のプラクティスを記述している。静的解析ツールは、一般に欠陥を検出するのではなく、警告を発することに注意してほしい。コードが構文的に正しくても、（例えば、複雑度に関する）警告を出すことがある。

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コンポーネント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、コンポーネント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（コンポーネントが実行時に互

いに依存する度合い）が低く、凝集度（コンポーネントが自己完結し、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す。
- コードやアーキテクチャの中で、構造的に複雑度が高い部分をハイライトする。

また、Web サイトのメンテナンスは、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。
- より困難なテストタスク
- メンテナンス負荷の増加

静的解析ツールは、保守性の評価に加えて、Web サイトの実装に使用されるコードに適用することで、コードインジェクション、Cookie 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 コードインジェクションなどのセキュリティの脆弱性にさらされる可能性をチェックすることが可能である。詳細は、4.3 節及びセキュリティテストシラバス[CT_SEC_SYL]参照のこと。

3.3 動的解析

3.3.1 概要

動的解析は、コードを実行したときにしか症状が見えないような故障を検出するために使用する。例えば、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見つける）、動的解析を使用するとメモリリークはすぐに判明する。

すぐに再現できない（間欠的な）故障は、テスト作業、及び、ソフトウェアをリリース、または本番での利用の能力に重大な影響を与える可能性がある。このような故障は、メモリリークやリソースリーク、ポインターの誤った使用、及びその他の破壊(システムのスタックなど)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も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、適切な場合には、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開始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。
- メモリリークの検出（3.3.2 節参照）やワイルドポインターの検出（3.3.3 節参照）により、故障の発生を未然に防ぐ。
- 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。
- コードプロファイラを使用して、実行時のシステム動作に関する情報を提供し、情報に基づいた変更を行うことで、システムの性能を向上させる。

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的を定める。
- 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。

動的解析ツールは、テクニカルテストアナリストの技術スキルが最低限の場合でも使用できる。これは、使用するツールは通常、必要な技術スキルや分析スキルを持つ人が分析できるような包括的なログを作成するためである。

3.3.2 メモリリークの検出

メモリリークは、メモリ（RAM）の領域がプログラムに割り当てられ、その後、不要になったときに解放されない場合に発生する。このメモリ領域は、再利用ができない。このような現象が頻繁に発生し、メモリ不足の状態になると、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。歴史的には、メモリ操作はプログラマーの責任であった。動的に確保されたメモリ領域は、メモリリークを防ぐために、確保したプログラムによって解放しなければならなかった。最近のプログラミング環境の多くは、自動または半自動の「ガベージコレクション」を備えており、割り当てられたメモリは使用後にプログラマーが直接介入することなく解放される。割り当てられたメモリが自動ガベージコレクションによって解放されるべきケースでは、メモリリークの切り分けが非常に困難になることがある。

メモリリークは、通常、ある程度の時間が経過してから、つまり、かなりの量のメモリがリークして利用できなくなったときに問題が発生する。ソフトウェアを新規にインストールしたとき、またはシステムを再起動したときは、メモリが再割り当てされるため、メモリリークに気づかない。例えば、テストをすると頻繁にメモリが割り当てられるため、メモリリークの検出を妨げることがある。このような理由から、メモリリークの負の影響は、プログラムが本番稼動したときに初めて気づくことがある。

システムの応答時間が徐々に悪化し、最終的にシステム故障につながる可能性がメモリリークの主な症状である。このような故障は、システムを再始動（リブート）することで解決できる場合もあるが、システムによっては必ずしも現実的ではなく、可能でない場合もある。

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ーを使用して、時間ととも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。

3.3.3 ワイルドポインターの検出

プログラム内のワイルドポインターとは、その時点で正しくないため使ってはいけないポインターのことである。例えば、ワイルドポインターが指し示すべきオブジェクトや関数を「失って」いたり、意図したメモリ領域を指していなかったりする（例えば、配列の割り当て境界を越えた領域を指している）。プログラムでワイルドポインターが使用されると、以下のようなさまざまな結果が発生する可能性がある。

- これは、現在プログラムによって使用していないメモリや、概念的に「解放」しているメモリ、そして/または妥当な値を含んでいるメモリにワイルドポインターがアクセスした場合である可能性がある。
- プログラムがクラッシュすることがある。この場合、ワイルドポインターが原因で、プログラム（オペレーティングシステムなど）の実行に重要なメモリの一部が間違って使用された可能性がある。
- プログラムが必要とするオブジェクトにアクセスできないため、プログラムが正しく機能しない。このような状況でも、エラーメッセージは表示されるが、プログラムは機能し続けることがある。
- ポインターによってメモリ上のデータが破壊され、その後に間違った値が使用される可能性がある（これはセキュリティ上の脅威にもなりえる）。

プログラムのメモリ使用方法の変更（例えば、ソフトウェアの変更に伴う新しいビルド）は、上記の 4 つの結果のいずれかを引き起こす可能性があることに注意してほしい。これは、ワイルドポインターが使用されているにもかかわらず、当初はプログラムが期待通りに動作し、ソフトウェアの変更後に予期せずクラッシュした場合（おそらく本番環境でも）に特に重大である。ツールは、プログラムの実行に与える影響に関係なく、プログラムによって使用されるワイルドポインターを特定するのに役立つ。一部のオペレーティングシステムでは、実行時にメモリアクセス違反をチェックする機能が組み込まれている。例えば、あるアプリケーションが、そのアプリケーションで許可されているメモリ領域の外側にあるメモリ位置にアクセスしようとすると、オペレーティングシステムは例外を返すことがある。

3.3.4 性能効率性の分析

動的解析は、故障の検出や関連する欠陥の特定に役立つだけではない。プログラム性能の動的解析により、ツールは、性能効率性のボトルネックを特定し、開発者がシステムの性能を調整するために使用可能な幅広い性能メトリクスを生成する。例えば、あるコンポーネントを実行中に何回呼び出すかという情報を提供できる。頻繁に呼び出されるコンポーネントは、性能強化の候補になりえる。プログラムは、実行時間の大部分(80%)を少数のコンポーネント(20%)に費やすというパレートの法則がしばしば当てはまる [Andrist20] 。

プログラム性能の動的解析は、システムテストを実施する際に行われることが多いが、テストハーネスを使用したテストの初期段階において、単一のサブシステムをテストする際にも行われることがある。詳細は性能テストシラバス[CT_PT_SYL]に記載されている。

4. テクニカルテストのための品質特性 - 345 分

キーワード

責任追跡性、適応性、解析性、真正性、可用性、キャパシティ（容量満足性）、共存性、互換性、機密性、障害許容性、設置性、インテグリティ、保守性、成熟性、修正性、モジュール性、否認防止性。運用プロファイル、性能効率性、移植性、品質特性、回復性、信頼性、信頼度成長モデル、置換性、資源効率性、再利用性、セキュリティ、試験性、時間効率性

テクニカルテストのための品質特性に関する学習の目的

4.2 一般的な計画の懸念事項

TTA-4.2.1 (K4) 特定のシナリオについて、非機能要件を分析し、テスト計画のそれぞれのセクションを記述する。
TTA-4.2.2 (K3) 特定のプロダクトリスクを想定し、最も適切な特定の非機能テストタイプを定義する。
TTA-4.2.3 (K2) アプリケーションのソフトウェア開発 ライフサイクルにおいて、非機能テストを通常適用するべき段階を理解し、説明する。
TTA-4.2.4 (K3) あるシナリオについて、異なる非機能テストタイプを用いて発見されると予想される欠陥のタイプを定義する。

4.3 セキュリティテスト

TTA-4.3.1 (K2) セキュリティテストをテストアプローチに含める理由を説明する。
TTA-4.3.2 (K2) セキュリティテストの計画及び仕様において考慮すべき主要な点について説明する。

4.4 信頼性テスト

TTA-4.4.1 (K2) テストアプローチに信頼性テストを含む理由を説明する。
TTA-4.4.2 (K2) 信頼性テストの計画及び仕様において考慮すべき主要な点について説明する。

4.5 性能テスト

TTA-4.5.1 (K2) テストアプローチに性能テストを含める理由を説明する。
TTA-4.5.2 (K2) 性能テストを計画し、仕様化する際に考慮すべき主要な点について説明する。

4.6 保守性テスト

TTA-4.6.1 (K2) テストアプローチに保守性テストを含める理由を説明する。

4.7 移植性テスト

TTA-4.7. (K2) テストアプローチに移植性テストを含む理由を説明する。

4.8 互換性テスト

TTA-4.8.1 (K2) テストアプローチに共存性テストを含む理由を説明する。

4.1 はじめに

一般に、テクニカルテストアナリストは、プロダクトが「何を」行うかという機能的側面よりも、プロダクトが「どのように」機能するかを中心にテストを行う。これらのテストは、どのテストレベルでも行うことができる。例えば、リアルタイム及び組み込みシステムのコンポーネントテストでは、性能効率性のベンチマークやリソース使用量のテストが重要である。運用受け入れテストやシステムテストでは、回復性などの信頼性をテストすることが適切である。このレベルのテストは、特定のシステム、すなわち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を目的としている。テスト対象の特定のシステムには、さまざまなサーバー、クライアント、データベース、ネットワーク、その他のリソースが含まれる場合がある。テストレベルに関係なく、テストはリスクの優先度と利用可能なリソースに従って実施されるべきである。

本章で説明する非機能品質特性のテストには、レビュー（2 章、3 章、5 章参照）を含む静的テストと動的テストの両方が適用できる。

ISO 25010 [ISO 25010]に規定されているプロダクト品質特性の説明は、特性及びその副特性のガイドとして使用する。テストアナリストとテクニカルテストアナリストのシラバスがどの特性/副特性をカバーしているかを示す印とともに、これらを以下の表に示す。

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
機能適合性	機能正確性、機能適切性、機能完全性	○	
信頼性	成熟性、障害許容性、回復性、可用性		○
使用性	適切度認識性、習得性、運用操作性、ユーザーインターフェース快美性、ユーザーエラー防止性、アクセシビリティ	○	
性能効率性	時間効率性、資源効率性、キャパシティ（容量満足性）		○
保守性	解析性、修正性、試験性、モジュール性、再利用性		○
移植性	適応性、設置性、置換性	○	○
セキュリティ	機密性、インテグリティ、否認防止性、責任追跡性、真正性		○
互換性	共存性		○
	相互運用性	○	

現在は廃止となっている ISO 9126-1 標準（本シラバスの 2012 年版に使用されているもの）と、その後の ISO 25010 標準で説明されている特性を比較した表を付録 A で提供している。

この節で論じたすべての品質特性及び副特性について、適切なテストアプローチを形成し、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。品質特性のテストは、ライフサイクルのタイミング、必要なツール、必要な標準、ソフトウェアと文書の可用性、及び技術的専門知識に特に注意を払う必要がある。各特性及びその固有のテストニーズに対処するためのアプローチを計画しなければ、テスト担当者は、十分な計画、準備及びテスト実行時間をスケジュールに組み込むことができないかもしれない。

このテストの一部、例えば性能テストは、広範囲な計画、専用の設備、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、かなりの時間を必要とする。品質特性及び副特性のテストは、その取り組みに十分なリソースを割り当てた上で、全体的なテストスケジュールに統合しなければならない。

テストマネージャーは、品質特性や副特性に関するメトリクス情報をまとめてレポートすることに関わる一方、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、（上の表に従って）各メトリクスから、それらの情報を集める。

テクニカルテストアナリストが本番前のテストで収集した品質特性の測定値は、ソフトウェアシステムのサプライヤーとステークホルダー（顧客、オペレーターなど）間のサービスレベル合意書（SLA）のベースとなる場合がある。場合によっては、テストはソフトウェアが本番稼動した後も継続的に実行されることがあり、多くの場合、別のチームまたは組織によって実行される。これは、本番環境ではテスト環境と異なる結果を示すことがある、性能テストや信頼性テストで通常見受けられる。

4.2 一般的な計画の懸念事項

非機能テストの計画に失敗すると、プロジェクトの成功はかなりのリスクにさらされる可能性がある。テクニカルテストアナリストは、テストマネージャーから、関連する品質特性（4.1 節の表参照）についての主要なリスクを特定し、提案されたテストに関連する計画上の問題に対処するように要求される場合がある。この情報は、マスターテスト計画の作成に使用することがある。

これらの作業を行う際には、以下の一般的な要因を考慮する。

- ステークホルダーの要件
- テスト環境の要件
- 必要なツールの入手とトレーニング
- 組織的な配慮
- データセキュリティへの配慮

4.2.1 ステークホルダー要件

非機能要件は、しばしば、十分に規定されていないか、あるいは、存在しない。計画段階では、テクニカルテストアナリストは、影響を受けるステークホルダーから技術的な品質特性に関する期待レベルを取得し、これらが示すリスクを評価できなければならない。

一般的なアプローチは、顧客が既存バージョンのシステムに満足している場合、達成された品質レベルが維持されている限り、新しいバージョンでも満足し続けると仮定することである。これにより、既存バージョンのシステムをベンチマークとして使用することができる。これは、ステークホルダーが要件を特定することが困難な、性能効率性などの非機能的な品質特性の一部に採用する場合に、特に有効なアプローチとなる。

非機能要件を把握する際には，複数の視点を持つことが望ましい。それらは、顧客、プロダクトオーナー、ユーザー、運用担当者、メンテナンス担当者などのステークホルダーから引き出す必要がある。主要なステークホルダーが除外されていると、いくつかの要件が見落とされる可能性がある。要件抽出の詳細については、テストマネージャー向けの Advanced Level シラバス[CTAL_TM_SYL]を参照のこと。

アジャイルソフトウェア開発では、非機能要件はユーザーストーリーとして記述されることもあれば、非機能制約としてユースケースで指定された機能に追加されることもある。

4.2.2 テスト環境要件

多くの技術的なテスト（セキュリティテスト、信頼性テスト、性能効率性テストなど）では、現実的な尺度を提供するため、本番に近いテスト環境が必要となる。テスト対象システムの規模や複雑度によっては、テストの計画や資金調達に大きな影響を与える可能性がある。このような環境はコストが高くつく可能性があるため、以下のような選択肢を検討することが考えられる。

- 本番環境を利用する
- スケールダウンしたバージョンのシステムを使用し、得られたテスト結果が本番システムを十分に代表するものであるように注意する
- リソースを直接取得する代わりに、クラウドベースのリソースを利用する
- 仮想化環境を利用する

特定の時間帯（例えば、使用率の低い時間帯）にしか実行できないテストがある可能性は十分にあるため、テスト実行時間は慎重に計画する必要がある。

4.2.3 必要なツールの入手とトレーニング

ツールはテスト環境の一部である。市販のツールやシミュレーターは、性能効率性テストや特定のセキュリティテストに特に適している。テクニカルテストアナリストは、ツールの取得、学習、実装に関わるコストと期間を見積もる必要がある。特殊なツールを使用する場合は、新しいツールの学習曲線や、外部のツール専門家を雇用するコストを考慮して計画する必要がある。

複雑なシミュレーターの開発は、それ自体が開発プロジェクトとなる場合があるため、そのように計画する必要がある。特に、開発したツールのテストと文書化は、スケジュールとリソース計画の中で説明しなければならない。シミュレーターのアップグレードと再テストは、シミュレートするプロダクトの変更に伴い、十分な予算と時間を計画しなければならない。セーフティクリティカルなアプリケーションで使用されるシミュレーターの計画では、独立した機関によるシミュレーターの受け入れテストと認証の可能性を考慮に入れなければならない。

4.2.4 組織的な検討事項

テクニカルテストでは、完全なシステム（サーバー、データベース、ネットワークなど）において、いくつかのコンポーネントの動作を測定することがある。これらのコンポーネントが複数の場所や組織に分散している場合、テストを計画し調整するために必要な労力は重大なものとなる可能性がある。例えば、あるソフトウェアコンポーネントは、特定の時間帯にしかシステムテストに利用できないかもしれないし、組織は限られた日数しかテストのサポートを提供しないかもしれない。他の組織からシステムコンポーネントやスタッフ（つまり専門的な技術を「借用」）がテストをするために「必要なときにすぐに呼び出せること」が可能であることの確認を怠ると、予定されているテストに深刻な混乱をきたす可能性がある。

4.2.5 データセキュリティとデータ保護

システムに対して実装される特定のセキュリティ対策は、すべてのテスト活動が可能であることを保証するために、テスト計画段階で考慮するべきである。例えば、データ暗号化の使用は、テストデータの作成とテスト結果の検証が困難にする場合がある。

データ保護ポリシーや法律により、本番データ（個人データ、クレジットカードデータなど）に基づく必要なテストデータの作成ができない場合がある。テストデータを匿名化することは、テスト実装の一部として計画しなければならない無視できない作業である。

4.3 セキュリティテスト

4.3.1 セキュリティテストを検討する理由

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ことで、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調査すべき潜在的な脅威のリストである。

- アプリケーションやデータの無許可でのコピー
- 無許可なアクセスコントロール（例：ユーザーが権限を持っていないタスクを実行できること）。ユーザーの権限、アクセス権、特権がこのテストの焦点となる。この情報は、システムの仕様書に記載されていることが望ましい
- 意図した機能を実行する際に、意図しない副作用を示すソフトウェア。例えば、メディアプレーヤーは正しくオーディオを再生するが、暗号化されていない一時記憶装置にファイルを書き出すため、ソフトウェア違法コピーに悪用される可能性のある副作用を示す
- 後続のユーザーによって実行される可能性がある Web ページに挿入されたコード（クロスサイトスクリプティングまたは XSS）。このコードは悪意のあるものである可能性がある
- ユーザー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長さを超える文字列を入力することによって引き起こされる可能性がある、バッファオーバーフロー (バッファオーバーラン)。バッファオーバーフローの脆弱性は、悪意のあるコード命令を実行する機会を意味する
- ユーザーがアプリケーションとやり取りするのを妨害するサービス拒否(例えば、「迷惑な」リクエストで Web サーバーに過度の負荷をかけること)
- 第三者による通信（例：クレジットカード決済）の傍受、模倣、改ざん、及びその後の中継により、ユーザーが第三者の存在に気づかないようにすること（「中間者」攻撃）
- 機密データを保護するために使用される暗号化コードを解読すること
- コードに悪意を持って挿入され、特定の条件下(例:特定の日)でのみ起動する、ロジックボム(イースターエッグと呼ばれることもある)。ロジックボムが作動すると、ファイルの削除やディスクのフォーマットなど、悪意のある行為を行う可能性がある

4.3.2 セキュリティテスト計画

一般に、セキュリティテストを計画する場合、次のような点が特に関係する。

- セキュリティの問題は、システムのアーキテクチャ、設計、及び実装の段階で取り込まれる可能性があるので、 セキュリティテストは、コンポーネントテスト、統合テスト、及び、システムテストで予定することがある。 セキュリティの脅威の性質が変化するため、セキュリティテストは、システムが本番に入った後にも定期的に予定することがある。使用するソフトウェアとハードウェアの要素に対する多くの更新によって本番の段階が特徴づけられる、モノのインターネット(IoT)のような動的なオープンアーキテクチャに特に当てはまる。
- テクニカルテストアナリストが提案するテストアプローチには、アーキテクチャ、設計、コードのレビューや、セキュリティツールを使用したコードの静的解析を含むことがある。これらは、動的テストでは見逃されやすいセキュリティ上の問題点を発見するのに有効である。
- テクニカルテストアナリストは、ステークホルダー（セキュリティテストの専門家を含む）との慎重な計画と調整を必要とする、特定のセキュリティ「攻撃」（以下を参照）の設計と

実施を求められることがある。その他のセキュリティテストは、開発者またはテストアナリストと協力して実施することもある （例えば、ユーザー権限、アクセス権、特権のテストなど）。
- セキュリティテスト計画の重要な側面は、承認を得ることである。テクニカルテストアナリストにとって、これは、計画したセキュリティテストを実施するために、テストマネージャーから明確な許可を得ていることを確実にすることを意味する。計画されていない追加のテストが実施されると、実際の攻撃と思われ、そのテストを実施する人は、法的措置のリスクにさらされる可能性がある。意図と承認を示す文書が何もない場合、「セキュリティテストを行っていた」という言い訳を、説得力を持って説明するのは難しいかもしれない。
- 組織が情報セキュリティ担当者の役割を持つ場合、すべてのセキュリティテスト計画は、その担当者と調整する必要がある。
- システムのセキュリティを改善する可能性がある場合、その性能効率性や信頼性に影響を与える可能性　に注意する必要がある。セキュリティの改善を行った後、性能効率性または信頼性テスト（4.5 節及び 4.4 節参照）を実施する必要性を検討することが望まれる。

セキュリティテスト計画を実施する際には、産業用オートメーション及び制御システムに適用される[IEC 62443-3-2]のように、個別の標準が適用される場合がある。

セキュリティテストシラバス［CT_SEC_SYL］は、主要なセキュリティテスト計画の要素に関するさらなる詳細を含む。

4.3.3　セキュリティテスト仕様

特定のセキュリティテストは、セキュリティリスクの発生原因に従って、グループ化することができる［Whittaker04］。これらには、次のようなものがある。
- ユーザーインターフェース関連 - 無許可なアクセス、悪意のある入力など
- ファイルシステム関連 - ファイルやリポジトリに保存され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 - パスワードなどの機密情報を暗号化なしでメモリに保存すること。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洩する可能性がある。
- 外部ソフトウェア関連 - システムが利用する外部コンポーネント間で発生する可能性のある相互作用。これは、ネットワークレベル（例：間違ったパケットやメッセージの受け渡し）またはソフトウェアコンポーネントレベル（例：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）である可能性がある。

ISO 25010 セキュリティの副特性 [ISO 25010] もまた、セキュリティテストを規定する際の基礎となり得る。これらは、セキュリティの次の側面に焦点を当てている。
- 機密性
- インテグリティ
- 否認防止性
- 責任追跡性
- 真正性

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）を利用することができる。
- 従業員の氏名、物理的な住所、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、テストを仕様化する上で有用と思われる情報を収集する。
- 広く利用可能なツールを使用して、脆弱性スキャンを実施する。このようなツールは、システムを侵害するために直接使用するのではなく、セキュリティポリシーに違反する、または違反する可能性のある脆弱性を特定するために使用する。また、米国国立標準技術研究所

（NIST）［Web-1］や Open Web Application Security Project™（OWASP）［Web-4］が提供するような情報やチェックリストを使って、特定の脆弱性を識別することも可能である。
- 収集した情報を用いて、「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侵害することを意図したテストにおける行動の計画）を作成する。最も深刻なセキュリティ欠陥を検出するために、攻撃計画にはさまざまなインターフェース（例：ユーザーインターフェース、ファイルシステム）を経由する複数の入力を指定する必要がある。[Whittaker04]で示すさまざまな 「攻撃」は、セキュリティテストのために特別に開発された技法の貴重な情報源となる。

なお、攻撃の計画は侵入テストのために作成されることもある。

3.2 節（静的解析）とセキュリティテストシラバス[CT_SEC_SYL]には、セキュリティテストに関するさらなる詳細を含む。

4.4 信頼性テスト

4.4.1 はじめに

ISO 25010 のプロダクト品質特性の分類は、成熟性、可用性、障害許容性、回復性を信頼性の副特性として定義している。信頼性のテストは、システムやソフトウェアが指定された期間、指定された条件下で指定された機能を実行する能力に関するものである。

4.4.2 成熟性のためのテスト

成熟性とは、システム（またはソフトウェア）が通常の運用条件下で信頼性要件を満たす度合いのことで、通常、運用プロファイル（4.9 節参照）を用いて規定する。成熟性測定は、使用される場合、しばしばシステムのリリース基準の 1 つに設定される。

従来、成熟性は、セーフティクリティカルな機能（例えば、航空機の飛行制御システム）に関連するような、成熟性目標が規制の標準の一部として定義された高信頼性システムに対して規定され測定されてきた。このような高信頼性システムの成熟性要件は、最大 10^9 時間の平均故障間隔 (MTBF) となるであろう (ただし、これを測定することは現実的に不可能である)。

高信頼性システムの成熟性をテストする通常のアプローチは信頼度成長モデルとして知られており，通常、他の品質特性に関するテストが終了し，検出された故障に関連する欠陥が修正された後のシステムテストの最後に行う。これは統計的アプローチで、通常、できるだけ運用環境に近いテスト環境で行う。指定した MTBF を測定するために、運用プロファイルに基づいてテスト入力を生成し、システムを実行して、故障を記録する（その後、修正する）。故障の発生頻度を減らすことで，信頼度成長モデルを用いて MTBF を予測することができる。

成熟性が信頼性の低いシステム（例えば、安全性に関連しないシステム）の目標として用いられる場合、運用上の利用を想定して定められた期間中に観測された故障の数（例えば、影響の大きい故障は週に 2 回以下）を用いることができ、システムのサービスレベル合意（SLA）の一部として記録することができる。

4.4.3 可用性のためのテスト

可用性は通常、システム（またはソフトウェア）が通常の運用条件でユーザーや他のシステムが利用可能な時間の合計で規定する。システムは成熟性は低くても、可用性は高い場合がある。例えば、電話網が何度か通話接続に失敗しても（従って成熟性は低い）、システムが迅速に回復して次の接続を

試みることができれば、ほとんどのユーザーは満足することができる。しかし、1 回の故障で電話網が数時間停止した場合は、許容できないレベルの可用性となる。可用性は、Web サイトや SaaS（Software as a Service）アプリケーションなどの運用システムで、SLA の一部として規定され、測定されることが多い。システムの可用性は、99.999%（「ファイブナイン」）と表現されることもあり、この場合、年間 5 分以上利用できないことがないはずである。また、システムの可用性を利用できない時間として規定することもある（例：月間 60 分以上システムをダウンしてはならない）。

運用前の可用性の測定（例えば、リリース判定の一部として）は、成熟性の測定に使われるのと同じテストを用いて行われることが多い。テストは、長期間にわたって予想される使い方の運用プロファイルに基づき、できるだけ運用環境に近いテスト環境で行う。可用性は MTTF/(MTTF + MTTR)で測定できる。MTTF は平均故障時間、MTTR は平均修復時間（MTTR）であり、保守性テストの一部として測定されることが多い。システムが高信頼性で回復性(4.4.5 節参照)を取り入れている場合、故障からの回復に時間がかかる場合、この方程式では MTTR の代わりに平均回復時間で表すことができる。

4.4.4 障害許容性のためのテスト

極めて高い信頼性が要求されるシステム (あるいはソフトウェア) では，故障が発生しても理想的には目立ったダウンタイムなしに運用を継続できるようなフォールトトレラント設計が採用されることが多い。システムの障害許容性を測る主な尺度は、システムが故障を許容できる能力である。そのため、障害許容性のテストでは、故障が発生したときにシステムが運用を継続できるかどうかを判断するために、故障をシミュレートすることを含む。テストすべき潜在的な故障条件を特定することは、障害許容性のテストの重要な部分である。

フォールトトレラント設計では、通常、1 つまたは複数のサブシステムを複製することを含み、故障の際に冗長性の段階を提供する。ソフトウェアの場合、共通モード故障を避けるために、このような重複したシステムを独立して開発する必要がある。このアプローチは、N バージョンプログラミングとして知られている。航空機の飛行制御システムは、3〜4 段階の冗長性を持つ場合があり、最も重要な機能は複数の種類で実装されることがある。ハードウェアの信頼性が懸念される場合、組み込みシステムは複数の異なるプロセッサーで実行されるかもしれない。一方、重要な Web サイトは、プライマリーサーバーが失敗した場合に引き継ぐために常に利用可能な、同じ機能を実行するミラー（フェイルオーバー）サーバーで実行されることがある。どのような障害許容性のアプローチを実装するにしても、それをテストするには、通常、故障の検出とその後の故障に対する応答の両方をテストする必要がある。

フォールトインジェクションテストは、システムの環境における欠陥（例えば、電源の故障、形式が正しくない入力メッセージ、プロセスやサービスが利用できない、ファイルが見つからない、メモリが利用できない）、及び、システム自体の欠陥（例えば、宇宙線によるビットの反転、設計不良、コーディング不良）が存在する場合のシステムの堅牢性をテストするものである。フォールトインジェクションテストは、ネガティブテストの一種である。システムに意図的に欠陥を注入し、システムが期待通り（安全関連のシステムであれば、安全）の反応をすることを確認する。ときには、テストする欠陥シナリオは、決して発生してはならないものであり（例えば、ソフトウェアタスクが「死ぬ」ことや、無限ループに陥ることはあってはならない）、従来のシステムテストではシミュレーションできない。しかし、フォールトインジェクションテストでは、故障を確実に検出し処理することができるようにするために、欠陥シナリオを作り、故障後のシステムの挙動を測定する。

4.4.5 回復性のためのテスト

回復性とは、システム（またはソフトウェア）が故障から回復する能力の尺度であり、回復に要する時間（運用状態が低下する場合もある）または失われたデータの量のいずれかを意味するものである。回復性テストのアプローチには、フェイルオーバーテストとバックアップ・リストアテストがある。

どちらも通常、ドライランに基づくテスト手順を含み、時々、理想的には抜き打ちで、運用環境でのハンズオンでのテストが行われる。

バックアップ・リストアテストは、システムデータへの故障の影響を最小限にするための手順をテストすることに重点を置いている。テストでは、データのバックアップと復元の両方の手順を評価する。データのバックアップのテストは比較的簡単だが、バックアップされたデータからシステムを復元するテストはより複雑な場合があり、運用システムの中断を最小限に抑えることを確実にするために慎重な計画が必要になることがよくある。測定には、さまざまなタイプのバックアップ（フルバックアップ、増分バックアップなど）の実施にかかる時間、データの復元にかかる時間（目標復元時間）、許容できるデータ損失のレベル（目標復元ポイント）が含まれる。

フェイルオーバーテストは、システムアーキテクチャがプライマリーシステムとプライマリーシステムに障害が発生した場合に引き継ぐフェイルオーバーシステムの両方で構成されている場合に実施する。システムが壊滅的な故障（洪水、テロ攻撃、深刻なランサムウェア攻撃など）から回復できなければならない場合、フェイルオーバーテストはしばしば災害復旧テストと呼ばれ、フェイルオーバーシステムはしばしば地理的に別の場所にある場合がある。運用中のシステムで完全な災害復旧テストを行うには、リスクと混乱（しばしば、復旧のマネジメントに取られがちなシニアマネージャーの自由時間のために）を考慮し、極めて慎重な計画が必要である。もし、完全な災害復旧テストが失敗したら、すぐにプライマリーシステムに戻すことになる（実際には破壊されていないからだ！）。フェイルオーバーテストは、フェイルオーバーシステムへの移行のテストを含み、フェイルオーバーシステムが引き継いだら、必要なサービスレベルを提供できるかテストする。

4.4.6 信頼性テストの計画

一般に、信頼性テストを計画する場合、特に次のような点が特に関係する。

- タイミング - 信頼性テストは通常、テストされるべき完成したシステムと他のテストタイプの完了を必要とし、実施には長い時間がかかる場合がある
- コスト - 高信頼性システムは、要求される高い MTBF を予測可能にするための、故障のない状態でシステムをテストする長い期間によって、テストが高額になることで知られている
- 期間 - 信頼度成長モデルを使った成熟性テストは、検出した故障に基づいて行い、高い信頼性レベルのためには、統計的に有意な結果を得るまでに長い時間を要する
- テスト環境 - テスト環境はできるだけ運用と同じにする必要があり、また、運用環境を使用することもある。しかし、運用環境を使用する場合、ユーザーに迷惑をかける可能性があり、例えば災害復旧テストが運用システムに悪影響を与えた場合、高リスクとなる可能性がある
- 範囲 - 異なるサブシステムやコンポーネントは、異なるタイプやレベルの信頼性をテストされる場合がある
- 終了基準 - 信頼性要件は、安全関連アプリケーションに対しては、規制の標準によって設定されるべきである
- 故障 - 信頼性測定は故障のカウントに大きく依存するため、何をもって故障とするかについて、前もって合意しておく必要がある
- 開発者 - 信頼度成長モデルを使った成熟性テストでは、特定された欠陥はできるだけ早く修正するように開発者と合意に至る必要がある
- 運用信頼性の測定は、リリース前の信頼性測定と比較すると、測定しなければならないのは故障のみであり比較的シンプルだが、運用担当者との連携が必要な場合がある
- 早期テスト - 高い信頼性を実現するためには、（信頼性を測定するのとは対照的に）、可能な限り早くテストすることや、初期のベースライン文章の厳密なレビュー、コードの静的解析などが求められる

4.4.7 信頼性テストの仕様

成熟性と可用性をテストする場合、テストは主に通常の運用条件下でシステムをテストすることに基づいている。このようなテストでは、システムがどのように使用されることを期待するかを定義した運用プロファイルが要求される。運用プロファイルの詳細については、4.9 項を参照のこと。

障害許容性や回復性をテストするためには、システムがどのように反応するかを判断するための、環境やシステム自体に故障を再現するテストの生成が必要になることが多い。フォールトインジェクションテストは、このためにしばしば使用する。考えられる欠陥とそれに対応する故障を特定するために、さまざまな技術やチェックリストが利用可能である（例えば、フォールトツリー解析、故障モード影響解析など）。

4.5 性能テスト

4.5.1 はじめに

ISO 25010 では、プロダクト品質特性の分類として、時間効率性、資源効率性、キャパシティ（容量満足性）を性能効率性の副特性として定義している。(性能効率性の品質特性に関する)性能テストは、使用される資源の量に対する、指定された条件下でのシステムやソフトウェアの性能の測定に関係している。典型的な資源には、経過時間、CPU 時間、メモリ、帯域幅がある。

4.5.2 時間効率性のためのテスト

時間効率性のためのテストは、指定された運用条件下におけるシステム（またはソフトウェア）の以下の側面を測定する。

- 要求を受けてから最初の応答までの経過時間（すなわち、要求された活動を完了するまでの時間ではなく、応答を開始するまでの時間）で、応答時間とも呼ばれる
- アクティビティを開始してからアクティビティを完了するまでのターンアラウンドタイムで、処理時間とも呼ばれる
- 単位時間あたりに完了したアクティビティの数（例：1 秒あたりのデータベース操作数）、スループット率とも呼ばれる

多くのシステムにおいて、異なるシステム機能に対する最大応答時間が要件として規定されている。このような場合、応答時間は経過時間にターンアラウンドタイムを加えたものとなる。システムがある活動を完了するために多くのステップ（パイプラインなど）を実施しなければならない場合、1 つまたは複数のステップがボトルネックの原因になっているかどうかを判断するために、各ステップにかかる時間を測定し、その結果を分析することが有用な場合がある。

4.5.3 資源効率性のためのテスト

資源効率性のためのテストは、指定された運用条件下におけるシステム（またはソフトウェア）の以下の側面を測定する。

- CPU の使用率、通常は使用可能な CPU 時間に対する割合
- メモリ使用率、通常は使用可能なメモリに対する割合
- I/O デバイスの使用率、通常は使用可能な I/O デバイス時間に対する割合
- 帯域幅の使用率、通常は使用可能な帯域幅に対する割合

4.5.4 キャパシティ（容量満足性）のためのテスト

キャパシティ（容量満足性）のテストは、指定された運用条件下におけるシステム（またはソフトウェア）の以下の側面の最大限度を測定する。

- 単位時間あたりに処理されるトランザクション（例：1 分間に最大 687 語の翻訳）
- 同時にシステムにアクセスするユーザー数（例：最大 1223 ユーザー）
- 単位時間あたりに追加される、システムにアクセスできる新規ユーザー数（例：1 秒間に最大 400 ユーザーが追加される）

4.5.5 性能テストの共通点

時間効率性、資源効率性、またはキャパシティ（容量満足性）をテストする場合、複数の測定値を取得し、平均値をレポート値として使用するのが普通である。これは、測定された時間値が、システムが実行している場合がある他のバックグラウンドタスクによって変動する可能性があるためである。状況によっては、測定値はより慎重に扱われ（例えば、分散または他の統計的尺度を使用）、または、外れ値が調査され、もし適切であればされば破棄されるかもしれない。

動的解析（3.3.4 節参照）は、ボトルネックを引き起こしているコンポーネントの特定、資源効率性テストのための使用リソース測定、キャパシティ（容量満足性）テストのための最大限度の測定に使用することができる。

4.5.6 性能テストのタイプ

性能テストは、他の多くのテスト形式とは異なり、2 つの異なる目的が存在する可能性がある。1 つは、テスト対象のソフトウェアが指定された受け入れ基準を満たすかどうかを判断することである。例えば、システムが要求された Web ページを最大値として規定された 4 秒以内に表示できるかどうかを判断する。もう 1 つの目的は、システム開発者に情報を提供し、システムの効率性の向上を助けることである。例えば、予想外に多くのユーザーがシステムに同時にアクセスした場合のボトルネックを検出し、システムアーキテクチャのどの部分に悪影響があるかを特定することである。

4.5.2 節、4.5.3 節及び 4.5.4 節で説明した性能テストはすべて、テスト対象のソフトウェアが指定された受け入れ基準を満たしているかどうかを判断するために使用することが可能である。また、システムを変更した際に、後で比較するための基準値を測定するためにも使用する。以下の性能テストタイプは、異なる運用条件下でシステムがどのように応答するかという情報を開発者に提供するために、より頻繁に使用する。

4.5.6.1 負荷テスト

負荷テストは、システムがさまざまな負荷を処理する能力に焦点を当てている。これらの負荷は、通常、システムに同時にアクセスするユーザー数または同時に実行されるプロセス数に関して定義され、運用プロファイルとして定義することができる（運用プロファイルの詳細については 4.9 節を参照のこと）。これらの負荷の処理は、通常、システムの時間効率性と資源効率性の観点で測定する（例：ユーザー数を 2 倍にした場合の応答時間への影響を判断する）。負荷テストを行う場合、低負荷から始めて徐々に負荷を増加させながら、システムの時間効率性や資源効率性を測定するのが一般的な慣行である。負荷テストから得られる情報のうち、開発者にとって有益であると思われる典型的な情報は、システムが特定の負荷を処理したときの応答時間やシステムリソースの使用状況における予期せぬ変化を含むであろう。

4.5.6.2 ストレステスト

ストレステストには 2 つの種類があり、1 つは負荷テストに似たもので、もう 1 つはロバストネステストの一種である。

1 つ目の負荷テストは通常、最初の負荷を想定される最大に設定し、その後、システムが失敗する（例えば、応答時間が不当に長くなる、またはシステムがクラッシュする）まで増加させて実施する。ときには、システムを強制的に故障させる代わりに、高い負荷を用いてシステムにストレスを与え、その後負荷を通常のレベルまで下げて、システムの性能レベルがストレスのかかる前のレベルに回復しているかを確認する。

2 つ目は、想定されるリソースへのアクセスを減らす（例えば、利用可能なメモリや帯域幅を減らす）ことで、システムを意図的に低下させて性能テストを実行する。ストレステストの結果は、システムのどの部分が最も重要で（つまり弱い部分）、アップグレードが必要であることの示唆を開発者に提供できる。

4.5.6.3 拡張性テスト

拡張性のあるシステムは、さまざまな負荷に適応することができる。例えば、拡張性のある Web サイトは、需要が増加するとより多くのバックエンドサーバーを使用するように拡張し、需要が減少するとより少なく使用するように縮小することができる。拡張性テストは、負荷テストに似ているが、負荷の変化（例えば、現在のハードウェアが処理できるより多いユーザー数）に直面したときに、システムがスケールアップ及びスケールダウンする能力をテストする。

性能テストシラバス[CT_PT_SYL]では、他の性能テストのタイプについても含めている。

4.5.7 性能テストの計画

一般に、性能効率性のテストを計画する際には、以下の点が特に重要である。

- タイミング - 性能テストは多くの場合、システム全体を実装し、代表的なテスト環境で実行する必要がある。つまり、通常はシステムテストの一部として実行する
- レビュー - コードレビュー、特にデータベースとのインタラクション、コンポーネントとのインタラクション、エラーハンドリングに焦点を当てたレビューは、性能効率性の問題（特に「待機と再試行」ロジックと非効率的なクエリに関して）を特定することができ、コードが利用可能になり次第（すなわち、動的テスト の前に）実行するようスケジュールする必要がある
- 早期テスト - 一部の性能テスト（例：重要なコンポーネントの CPU 使用率の特定）は、コンポーネントテストの一部としてスケジュールされる可能性がある。性能テストによりボトルネックであると特定されたコンポーネントは、更新された後、コンポーネントテストの一部として、独立して再テストされることもある
- アーキテクチャの変更 - 性能テストで悪い結果が出た場合、ときにシステムアーキテクチャの変更につながることがある。性能テストの結果からシステムの大きな変更が示唆される場合、そのような問題に対処するために利用できる時間を最大化するために、性能テストはできるだけ早く開始する必要がある
- コスト - ツールやテスト環境は高価になる可能性があるため、一時的なクラウドベースのテスト環境を借りたり、「追加購入」のツールライセンスを使用したりすることがある。このような場合、テスト計画では通常、コストを最小化するためにテストの実行時間を最適化する必要がある

- テスト環境 - テスト環境は、可能な限り運用環境を代表するものである必要があり、そうでなければ、テスト環境から期待される運用環境へ性能テストの結果をスケーリングするという課題が増える
- 終了基準 - 性能効率性要件は顧客から得ることがときとして困難な場合があるため、以前のまたは類似のシステムからのベースラインから導き出されることが多い。組込み安全関連システムの場合、CPU やメモリの最大使用量など、一部の要件は規制の標準で指定されている場合がある
- ツール - 負荷生成ツールは、性能テストをサポートするためにしばしば必要となる。例えば、人気のある Web サイトの拡張性を検証する場合、何十万人もの仮想ユーザーのシミュレーションが必要になることがある。また、リソースの制限をシミュレートするツールは、ストレステストに特に有効である。テストをサポートするために取得したツールは、テスト対象のシステムで使用する通信プロトコルと互換性があることを確認するよう注意する必要がある

性能テストシラバス[CT_PT_SYL]では、性能テストの計画に関する詳細を含んでいる。

4.5.8 性能テストの仕様

性能テストは主に、指定された運用条件下でシステムをテストすることに基づいている。このようなテストには、システムがどのように使用されると期待するかを定義した運用プロファイルが必要である。運用プロファイルの詳細については、4.9 節を参照のこと。

性能テストの場合、システムの想定される運用利用からの変化をシミュレートするために、運用プロファイルの一部を修正し、システムの負荷を変更することがしばしば必要である。例えば、キャパシティ（容量満足性）テストの場合、通常、キャパシティテストを行う変数の領域で運用プロファイルを上書きする必要がある（例えば、システムが応答しなくなるまでシステムにアクセスするユーザー数を増やして、ユーザーアクセスのキャパシティを決定する）。同様に、負荷テストを行う際には、トランザクションの量を徐々に増加させる場合がある。

性能テストシラバス[CTSL_PT_SYL]では、性能効率性のテスト設計の詳細を含んでいる。

4.6 保守性テスト

ソフトウェアは、しばしばその寿命の大部分を、開発よりも保守に費やす。メンテナンスの作業が可能な限り効率的であることを保証するために、保守性テストは、コードの解析、変更、テスト、モジュール化、再利用が容易に行えるかどうかを測定するために実施される。保守性テストは、運用中のソフトウェアに加えられた変更をテストするために実施されるメンテナンステストと混同してはならない。

影響を受けるステークホルダー（ソフトウェアオーナーやオペレーターなど）の代表的な保守性の目的には、以下のようなものがある。

- ソフトウェアの所有や運用のコストを最小化すること
- ソフトウェアメンテナンスに必要なダウンタイムを最小化すること

保守性テストは、以下の要素の1つ以上に該当する場合、テストアプローチに含めるべきである。

- ソフトウェアが本番稼動後に変更される可能性がある（欠陥の修正や計画的なアップデートの導入など）

- 影響を受けるステークホルダーが、ソフトウェア開発ライフサイクルを通じて保守性の目標を達成することの利益について、保守性テストの実施と必要な変更の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場合のリスク（例えば、ユーザー及び/または顧客からレポートされた欠陥に対する長い応答時間）は、保守性テストの実施を正当化する

4.6.1 静的または動的な保守性テスト

静的な保守性テストに適切な技法としては、3.2節と5.2節で説明した静的解析とレビューを含んでいる。保守性テストは、設計書ができ次第すぐに開始し、コード実装の期間中も継続する必要がある。保守性はコードと各コードコンポーネントの文書に組み込まれているので、完成して動作するシステムを待つことなく、ソフトウェア開発ライフサイクルの早い段階で保守性を評価することができる。

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ために開発された文書化された手順（例：ソフトウェアアップグレードの実施）に焦点を当てている。メンテナンスシナリオをテストケースとして使用し、文書化された手順で要求されるサービスレベルが達成可能であることを確認する。この形式のテストは、基盤となるインフラストラクチャが複雑で、サポート手順が複数の部門/組織を巻き込む可能性がある場合に特に関連する。この形式のテストは、運用受け入れテストの一部として実施されることがある。

4.6.2 保守性の副特性

システムの保守性は、次のような観点で測定できる。

- 解析性
- 修正性
- 試験性

これらの特性に影響を与える要因としては，優れたプログラミング手法の適用（コメント付け，変数の命名，インデントなど）、技術文書の可用性（システム設計仕様，インターフェース仕様など）などを含んでいる。

他の保守性に関連する品質副特性［ISO 25010］は以下の通り。

- モジュール性
- 再利用性

モジュール性は、静的解析によってテストできる（3.2.3 節参照）。再利用性のテストは、アーキテクチャレビューの形で行うことができる（5 章参照）。

4.7 移植性テスト

4.7.1 はじめに

一般に、移植性テストは、ソフトウェアコンポーネントまたはシステムが、（当初または既存の環境から）意図した環境に移行できる程度、新しい環境に適応できる程度、または別のエンティティを置換できる程度に関するものである。

ISO 25010 [ISO 25010]には、移植性の副特性として次のようなものを含む。

- 設置性
- 適応性

- 置換性

移植性テストは、個々のコンポーネントから始め（例えば、あるデータベースマネジメントシステムから別のものへの変更のような、特定のコンポーネントの置換性）、より多くのコードが利用可能になるにつれて範囲を拡大することができる。設置性は、プロダクトの全コンポーネントが機能的に動作するようになるまでテストできない場合がある。

移植性はプロダクトに設計して組み込まなければならないため、設計とアーキテクチャの早い段階で考慮する必要がある。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や問題（特定のオペレーティング・システムへの依存など）を特定する上で、特に生産的なものとなりうる。

4.7.2 設置性テスト

設置性テストは、ソフトウェアと、そのソフトウェアを対象の環境にインストールするために使用する手順書に対して実施する。これには、例えば、オペレーティングシステムをインストールするために開発されたソフトウェアや、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」などが含まれることがある。

典型的な設置性テストの目的には、以下のようなものがある。

- 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、ソフトウェアをインストールできることを妥当性確認すること。これは、さまざまなハードウェア/ソフトウェア構成、さまざまなインストールの程度（例：初期または更新）に対するインストールオプションの行使を含む
- インストール中に発生した故障（例えば、特定の DLL のロードに失敗した場合）が、システムを未定義の状態（例えば、部分的にインストールされたソフトウェアや間違ったシステム構成）にすることなく、インストールソフトウェアによって正しく処理されるかどうかをテストすること
- 部分的なインストール／アンインストールが完了できるかどうかをテストすること
- 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またはオペレーティングシステム構成を特定できるかどうかをテストすること
- 指定された時間（分）内、または指定されたステップ数内でインストールプロセスを完了できるかどうかを測定すること
- ソフトウェアのダウングレードやアンインストールが可能かどうかを妥当性確認すること

機能適合性テストは通常、設置テストの後に実施し、インストールによって生じた欠陥（例：誤った構成、使用できない機能）を検出する。使用性テストは通常、設置性テストと並行して実施される（例：ユーザーへインストール中に分かりやすい説明とフィードバック／エラーメッセージが提供されていることを妥当性確認するため）。

4.7.3 適応性テスト

適応性テストは、あるアプリケーションが、意図されたすべての対象の環境（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）において、正しく機能するかどうかをチェックする。適応性のためのテストを仕様化するためには、意図した対象の環境を特定し、その環境を構成し、テストチームが利用できるようにすることが必要となる。そして、これらの環境は、その環境内に存在するさまざまなコンポーネントを実行する機能テストケースから選択されたものを使用してテストする。

適応性は、あらかじめ定義された手順を実行することによって、さまざまな指定された環境にソフトウェアを移植する能力に関連する場合がある。テストではこの手順を評価することができる場合がある。

4.7.4 置換性テスト

置換性テストは、システム内の既存のソフトウェアコンポーネントを置き換えることができるソフトウェアコンポーネントの能力に焦点を当てる。これは、特定のシステムコンポーネントに市販ソフトウェア（COTS）を使用するシステムや、IoT アプリケーションに特に関連する可能性がある。

置換性テストは、完全なシステムを構成するために、複数の代替可能なコンポーネントの選択肢がある場合、機能面での統合テストと並行して実施することができる場合がある。置換性は、アーキテクチャや設計レベルでのテクニカルレビューまたはインスペクションによって評価することもできる。そこでは、代替品として使用することができる可能性があるコンポーネントのインターフェースの明確な定義に重点が置かれる。

4.8 互換性テスト

4.8.1 はじめに

互換性テストは、以下の側面を考慮する[ISO 25010]。

- 共存性
- 相互運用性

4.8.2 共存性テスト

互いに関係のないコンピューターシステムが、同じ環境（例えば、同じハードウェア上）で、互いの動作に影響（例えば、リソースの競合）を与えることなく実行できるとき、共存していると言える。共存性テストは、新規またはアップグレードされたソフトウェアを、すでにインストールしているアプリケーションを含む環境に展開する場合に実施する必要がある。

共存性の問題は、アプリケーションが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（非互換性の問題が検出可能でない環境）でテストされ、その後、他のアプリケーションも実行する別の環境（本番環境など）にデプロイされた場合に発生する可能性がある。

共存性テストの代表的な目的は以下の通り。

- 同一環境下でアプリケーションをロードした場合に考えられる機能適合性への悪影響の評価（例：サーバーで複数のアプリケーションを実行した場合のリソースの使用の競合など）
- オペレーティングシステムの修正とアップグレードのデプロイによる、アプリケーションへの影響の評価

共存性の問題は、対象となる本番環境を計画する際に解析しておく必要があるが、実際のテストは通常、システムテストが完了した後に実施する。

4.9 運用プロファイル

運用プロファイルは、信頼性テストや性能テストなどを含む、いくつかの非機能テストタイプでテスト仕様書の一部として使用される。これらは、テストする要件に「指定された条件下で」という制約が含まれる場合、これらの条件を定義するために使用することができるため、特に有用である。

運用プロファイルは、システムの利用パターンを定義するもので、通常、システムの利用者とシステムで実施される操作という観点から定義する。ユーザーは、通常、どれくらいの人数が（どの時間に）システムを使用すると予想されるか、そして、また場合によってはそのタイプ（例えば、一次ユーザー、二次ユーザー）の観点から特定する。システムによって実施することが予想されるさまざまな操作は、通常、その頻度（及び発生確率）とともに特定する。この情報は、モニタリングツールを使用する（実際のまたは類似のアプリケーションがすでに利用可能な場合）か、アルゴリズムまたは事業組織が提供する推定値に基づいて使用状況を予測することによって得ることができる。

運用プロファイルに基づいたテスト入力を生成するためツールを使用することがでる。多くの場合、テスト入力を疑似ランダムで生成するアプローチを使用する。このようなツールは、運用プロファイルに一致する量（例えば、信頼性と可用性のテスト向けの）またはそれを超える量（例えば、ストレスまたはキャパシティテスト向けの）の「仮想」またはシミュレーションユーザーを作成するために使用することができる。これらのツールの詳細については、6.2.3 節を参照。

5. レビュー- 165 分

キーワード

レビュー、テクニカルレビュー

レビューの学習の目的

5.1 レビューにおけるテクニカルテストアナリストのタスク

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

5.2 レビューにおけるチェックリストの活用

TTA 5.2.1 (K4) シラバスで提供されるチェックリストに従って、アーキテクチャ設計を解析し、問題点を特定する。

TTA 5.2.2 (K4) シラバスで提供されるチェックリストに従って、コードの一部または疑似コードを解析し、問題点を特定する。

5.1 レビューにおけるテクニカルテストアナリストのタスク

テクニカルテストアナリストは、テクニカルレビュープロセスに積極的な参加者であるべきであり、独自の意見を提供する。すべてのレビュー参加者は、それぞれの役割をよりよく理解するために形式的なレビュートレーニングを受けるべきであり、また、適切に運用されたテクニカルレビューの利点を理解し、それに基づいて行動しなければならない。これには、レビューコメントを記述し、議論する際に、作成者と建設的な業務関係を維持することを含む。多数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、テクニカルレビューやインスペクションに参加し、開発者が見逃してしまうような運用（行動）の視点をもたらす。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の定義、適用、メンテナンス、及び欠陥の重要度情報の提供において重要な役割を担っている。

実施するレビューのタイプに関係なく、テクニカルテストアナリストには、準備のための十分な時間が与えられなければならない。これには、作業成果物をレビューする時間、一貫性を検証するために相互参照された文書をチェックする時間、作業成果物に何が欠けている可能性があるのかを判断する時間などが含まれる。十分な準備時間がないと、レビューが真のレビューではなく、編集作業になってしまう可能性がある。優れたレビューには、書かれていることの理解、何が欠けているかの判断、技術的側面に関する正確さの検証、そして記述されたプロダクトがすでに開発されている、または開発中の他のプロダクトと整合していることの検証が含まれる。例えば、統合レベルのテスト計画書をレビューする場合、テクニカルテストアナリストは、統合されるアイテムについても考慮しなければならない。それらは統合の準備ができているか？文書化されなければならない依存関係はあるか？統合ポイントをテストするために利用可能なデータはあるか？レビューは、レビューされる作業成果物に限定されるものではない。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しなければならない。

5.2 レビューにおけるチェックリストの活用

チェックリストはレビュー中に使用され、参加者に特定のポイントを確認することを思い出させる。チェックリストは、「これは、我々がすべてのレビューで使用する同じチェックリストであり、あなたの作業成果物だけを対象にしているわけではない」など、レビューを客観的に行うためにも有効である。チェックリストは、汎用的なものとしてすべてのレビューに使用できる他、特定の品質特性や分野に焦点を当てることができる。例えば、汎用的なチェックリストは、用語「shall」と「should」の適切な使い分けを検証できる場合があり、適切な書式と類似の適合性項目を検証できる場合がある。チェックリストはセキュリティの問題や性能効率性に焦点を当てることもある。

最も有用なチェックリストは、以下の内容を反映するため、各組織で徐々に作成する。

- プロダクトの性質
- ローカル開発環境
  - スタッフ
  - ツール
  - 優先度
- 過去の成功例と欠陥の履歴
- 特定の問題（性能効率性、セキュリティなど）

チェックリストは、組織に合わせて、また場合によって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。本章で紹介するチェックリストはその例である。

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、システムの基本的な概念または特性からなり、その要素、関係、及び設計と進化の原則が組み込まれている [ISO 42010]。

Web サイトの時間効率性のアーキテクチャレビューに用いるチェックリスト [1] には、例えば、[Web-2]から引用した以下の項目の適切な実装の検証を含めることができる。

- 「コネクションプーリング - コネクションの共有プールを確立することにより、データベース接続の確立に関連する実行時間のオーバーヘッドを削減する
- ロードバランシング - 一連のリソース間で負荷を均等に分散させる
- 分散処理
- キャッシング - アクセス時間短縮のためにデータのローカルコピーを使用する
- 遅延インスタンス化(Lazy instantiation)
- トランザクションの並行性
- 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）間のプロセス分離
- データの複製」

5.2.2 コードレビュー

コードレビューのためのチェックリストは、必然的にローレベルのものになり、言語に特化したときに最も有効である。特に経験の浅いソフトウェア開発者には、コードレベルのアンチパターンを含めると有用である。

コードレビューに使用されるチェックリスト[1]には、以下のような項目を含めることができる。

1.構造

- コードは設計を完全かつ正確に実装しているか？
- コードが適切なコーディング標準に準拠しているか？
- コードはきちんと構造化されているか、スタイルに一貫性があるか、書式が一貫しているか。
- 呼び出されていない、または不要なプロシジャーや、到達できないコードはないか？
- コードの中にスタブやテストルーチンが残っていないか？
- どのコードも、外部の再利用可能なコンポーネントやライブラリ関数の呼び出しに置き換えることができるか？
- 繰り返されるコードの中に、1 つのプロシジャーに凝縮できるブロックはないか？
- ストレージの使用は効率的か？
- 「マジックナンバー」定数や文字列定数ではなく、シンボリックが使用されているか？
- モジュールが非常に複雑であるため、再構築または複数のモジュールに分割したほうが良いものはあるか？

2.ドキュメンテーション

- コードが保守しやすいコメントスタイルを使い、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整合しているか？
- 文書は、適用可能な標準に適合しているか？

3.変数

- すべての変数が、意味のある一貫した明確な名前で適切に定義されているか？
- 冗長または未使用の変数はないか？

[1]試験問題では、質問に回答するためのチェックリストのサブセットを提供する。

4.算術演算

- コードは浮動小数点数が等しいかどうかの比較を避けているか？
- コードは丸め誤差をシステム的に防いでいるか？
- コードは大きさが大きく異なる数値の加算や減算を避けているか？
- 除数は、0 やノイズでテストされたか？

5.ループとブランチ

- ループ、ブランチ、ロジックの構成はすべて完全で、正しく、適切にネストされているか？
- IF-ELSEIF チェーンで、最も一般的なケースを最初にテストしているか？
- ELSE または DEFAULT 句を含む IF-ELSEIF または CASE ブロックで、すべてのケースがカバーされているか？
- すべての CASE ステートメントにデフォルトがあるのか？
- ループの終了条件は明白であり、必ず達成できるのか？
- ループの直前で、インデックスや添え字が適切に初期化されているか？
- ループで囲まれたステートメントは、ループの外側に置くことができるか？
- ループ内のコードは、インデックス変数の操作やループを抜ける際の使用を避けているか？

6.防御的プログラミング

- インデックス、ポインター、添え字は、配列、レコード、ファイルの境界に対してテストしているか？
- インポートしたデータや入力引数の妥当性・完全性に対してテストしているか？
- すべての出力変数が割り当てられているか？
- 各ステートメントで正しいデータエレメントが操作されているか？
- すべてのメモリ割り当てが解放されているか？
- 外部機器アクセスについてタイムアウトやエラートラップが使用されているか？
- ファイルへのアクセスを試みる前に、ファイルの存在をチェックしているか？
- プログラム終了時に、すべてのファイルやデバイスが正しい状態で残されているか？

6. テストツールと自動化 - 180 分

キーワード

キャプチャ/プレイバック、データ駆動テスト、エミュレーター、フォールトインジェクション、フォールトシーディング、キーワード駆動テスト、モデルベースドテスト(MBT)、シミュレーター、テスト実行

テストツールと自動化の学習の目的

6.1 テスト自動化プロジェクトを定義する

TTA-6.1.1 (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行う活動を要約する。
TTA-6.1.2 (K2) データ駆動型自動化とキーワード駆動型自動化の違いについて要約する。
TTA-6.1.3 (K2) 自動化プロジェクトが計画した投資収益率を達成できない原因となる一般的な技術的問題を要約する。
TTA-6.1.4 (K3) 与えられたビジネスプロセスに基づいてキーワードを構築する。

6.2 特定用途のテストツール

TTA-6.2.1 (K2) フォールトシーディングとフォールトインジェクションのためのツールの目的を要約する。
TTA-6.2.2 (K2) 性能テストツールの主な特徴や実装上の問題点を要約する。
TTA-6.2.3 (K2) Web ベースのテストに使用されるツールの一般的な目的を説明する。
TTA-6.2.4 (K2) モデルベースドテストの実践をサポートするツールについて説明する。
TTA-6.2.5 (K2) コンポーネントテストやビルドプロセスをサポートするためのツールの目的を概説する。
TTA-6.2.6 (K2) モバイルアプリケーションのテストを支援するために使用するツールの目的を概説する。

6.1 テスト自動化プロジェクトを定義する

費用対効果を高めるために、テストツール（特にテスト実行をサポートするツール）は、慎重にアーキテクチャ構築と設計を行う必要がある。しっかりとしたアーキテクチャを持たずにテスト実行自動化戦略を実装すると、通常、保守が高価になり、目的に対して不十分で、目標とする投資収益率を達成できないツールセットとなる。

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考えるべきである。これには、アーキテクチャの文書化、詳細設計の文書化、設計とコードのレビュー、コンポーネントとコンポーネント統合テスト、及び最終的なシステムテストの必要性が含まれる。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、テストが不必要に遅れたり、複雑になったりすることがある。

テスト実行の自動化に関して、テクニカルテストアナリストが実践できるタスクは複数ある。その中には、以下のようなものがある。

- 誰がテスト実行の責任を持つかを決定する（テストマネージャーと連携する場合もある）
- 組織、スケジュール、チームのスキル、メンテナンス要件に適したツールを選択する（これは、ツールを取得するのではなく、使用するツールの作成を決定することを意味する。）
- 自動化ツールと他のツールとの間のインターフェース要件を定義する。他のツールとは、テストマネジメント、欠陥マネジメント、継続的インテグレーションに使用するツールなどがある
- テスト実行ツールとテスト対象ソフトウェア間のインターフェースを作成するために必要となる可能性があるアダプターの開発
- 自動化アプローチ、すなわちキーワード駆動かデータ駆動の選択（6.1.1 節参照）
- テストマネージャーと協力して、トレーニングを含む実装のコストを見積もる。アジャイルソフトウェア開発では、この側面は通常、チーム全体とのプロジェクト/スプリントプランニング会議で議論され、合意されると考えられる
- 自動化プロジェクトのスケジューリングとメンテナンスのための時間確保
- 自動化のためのデータを使用や提供するために、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
- 自動テストをいつ、どのように実行するかを決定する
- 自動化されたテストの結果と手動テストの結果をどのように組み合わせるかを決定する

テスト自動化を強く重視するプロジェクトでは、テスト自動化エンジニアがこれらの活動の多くを担当することがある（詳しくは、テスト自動化エンジニアのシラバス[CT_TAE_SYL]を参照のこと）。特定の組織的なタスクは、プロジェクトのニーズと選択に応じて、テストマネージャーが担当することがある。アジャイルソフトウェア開発では、これらのタスクの役割への割り当ては、通常、より柔軟であり、形式的ではない。

これらの活動やその結果の決定は、自動化ソリューションの拡張性と保守性に影響する。選択肢の調査、利用可能なツールや技術の調査、組織の将来計画の把握に十分な時間を費やす必要がある。

6.1.1 自動化アプローチの選択

このセクションでは、テスト自動化のアプローチに影響を与える以下の要因について考察する。

- GUI、API、CLI による自動化
- データ駆動アプローチの適用
- キーワード駆動アプローチの適用
- ソフトウェア故障への対応
- システムの状態の考慮

テスト自動化エンジニアのシラバス[CT_TAE_SYL]には、自動化アプローチの選択に関するさらなる詳細が含まれている。

6.1.1.1 GUI、API、CLI を使った自動化

テストの自動化は、GUI によるテストに限定されるものではない。API レベル、コマンドラインインターフェース（CLI）や、その他テスト対象ソフトウェアのインターフェースポイントを使ったテストの自動化を支援するためのツールも存在する。テクニカルテストアナリストが最初に決定しなければならないことの 1 つは、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を決定することである。一般的なテスト実行ツールでは、これらのインターフェースへのアダプターを開発することが要求される。計画にあたっては、アダプター開発のための労力を考慮しなければならない。

GUI によるテストの難しさの 1 つは、ソフトウェアの進化に伴って GUI が変化しやすいことである。テスト自動化コードの設計方法によっては、メンテナンスに大きな負担がかかることにつながる可能性がある。例えば、テスト自動化ツールのキャプチャ/プレイバック機能を使用すると、GUI が変わった場合に自動テストケース（多くの場合、テストスクリプトと呼ばれる）が望んだ通りに実行されなくなることがある。これは、テスト担当者がソフトウェアを手動で実行する際に、記録されたスクリプトがグラフィカルオブジェクトとのインタラクションをキャプチャするためである。アクセスされるオブジェクトが変更された場合、記録されたスクリプトもその変更を反映するために更新が必要になることがある。

キャプチャ/プレイバックツールは、自動化スクリプトを開発するための便利な出発点として使用されることがある。テスト担当者はテストセッションを記録し、記録されたスクリプトは（例えば、記録され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の）保守性を向上させるために修正する。

6.1.1.2 データ駆動アプローチの適用

テストするソフトウェアによっては、実行するテストステップはほぼ同じでも、各テストで使用するデータが異なる場合がある（例えば、複数の無効な値を入力し、それぞれに返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）。このようなテスト対象の値ごとに自動テストスクリプトを開発・保守するのは非効率的である。この問題に対する一般的な技術的解決策は、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ストアにデータを移動させることである。テストスクリプトの各実行に対して特定のデータにアクセスするための関数が書かれ、これにより、入力値と期待される結果値（例えば、テキストフィールドに示される値やエラーメッセージ）を供給するテストデータのセットを通して、単一のスクリプトが動作することが可能になる。この方法は、データ駆動型と呼ばれる。

このアプローチを使用する場合、供給され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のセットの実行をサポートするハーネスとインフラストラクチャが必要になる。スプレッドシートやデータベースに保持される実際のデータは、ソフトウェアのビジネス機能に精通したテストアナリストが作成する。アジャイルソフトウェア開発では、ビジネスの代表者（例えばプロダクトオーナー）も、データ定義に関与することがある、特に受け入れテストにおいてそうである。この分業により、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）は自動化スクリプトの実装に集中でき、テストアナリストは実際のテストのオーナーシップを維持することができる。ほとんどの場合、テストアナリストは、自動化が実装され、テストされると、テストスクリプトの実行を担当する。

6.1.1.3　キーワード駆動アプローチの適用

もう 1 つのアプローチは、キーワード駆動、またはアクションワード駆動と呼ばれるものである。これは、さらに一歩進んで、テストデータに対して実行される動作をテストスクリプトから分離する[Buwalda01]。このさらなる分離を実現するために、直接実行可能な言語ではなく、記述可能な高水準言語を作成する。キーワードは、高レベルのアクションと低レベルのアクションの両方を定義することができる。例えば、ビジネスプロセスのキーワードには、「Login」、「CreateUser」、「DeleteUser」などがある。このようなキーワードは、アプリケーション領域で実施される高レベルのアクションを表している。低レベルのアクションは、ソフトウェアインターフェース自体との相互作用を表す。「ClickButton」,「SelectFromList」,「TraverseTree 」のようなキーワードは、GUI 機能をテストするために使用されることがあるが、ビジネスプロセスのキーワードにうまく当てはまらない。キーワードはパラメーターを含むことができる。例えば、キーワード「Login」は、user_name と password の 2 つのパラメーターを持つことができる。

使用するキーワード及びデータを定義すると、テスト自動化担当者（テクニカルテストアナリストまたはテスト自動化エンジニアなど）は、ビジネスプロセスのキーワード、及び低レベルのアクションをテスト自動化コードへ変換する。キーワードとアクションは、使用するデータとともに、スプレッドシートに格納することもあり、またはキーワード駆動テストの自動化をサポートする特定のツールを使用して入力することもある。テスト自動化フレームワークでは、1 つまたは複数の実行可能な機能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、キーワードで記述されたテストケースを読み取り、それらを実装する適切なテストの関数またはスクリプトを呼び出す。実行ファイルは、特定のキーワードに容易に対応できるように、モジュール性を高くして実装する。これらのモジュール性スクリプトを実装するためには、プログラミングのスキルが必要となる。

このように、ビジネスロジックの知識と、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を分離することで、テストリソースを最も効果的に使用することができる。テクニカルテストアナリストは、テスト自動化担当者として、ビジネスの多くの領域にわたるドメインの専門家になることなく、プログラミングスキルを効果的に適用することができる。

コードを変更可能なデータから分離することで、自動化を変更から隔離し、コード全体の保守性を向上させ、自動化の投資収益率を向上させることができる。

6.1.1.4　ソフトウェア故障への対処

テスト自動化の設計では、ソフトウェアの故障を予測して対処することが重要である。故障が発生した場合、テスト自動化担当者はテスト実行しているソフトウェアが何をすべきかを判断しなければならない。故障を記録し、テストを継続するべきか？テストを終了するべきか？故障は特定の動作（ダイアログボックスのボタンのクリックなど）で対処できるのか？あるいはテストに遅延を追加することで対処できるのか？ソフトウェアの故障に対処しないと、その後のテスト結果に影響を与える可能性があり、それだけでなく、故障が発生したときに実行されていたテストに問題を引き起こす。

6.1.1.5　システムの状態の考慮

各テストケースの開始時と終了時のシステムの状態を考慮することも重要である。テストの実行が完了した後、システムをあらかじめ定義された状態に戻すことが必要な場合もある。これにより、システムを既知の状態にリセットするために手動で介入することなく、自動化されたテストスイートを繰り返し実行することができるようになる。これを行うために、テスト自動化は、例えば、作成したデータを削除したり、データベース内のレコードのステータス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は、テストの終了時に適切な終了が達成されたことを確認する必要がある（テスト完了後のログアウトなど）。

6.1.2 自動化のためのビジネスプロセスのモデリング

テスト自動化のためのキーワード駆動アプローチを実装するには、テストするビジネスプロセスをハイレベルなキーワード言語でモデル化する必要がある。この言語は、プロジェクトに携わるテストアナリストや、アジャイルソフトウェア開発の場合はビジネス担当者（プロダクトオーナーなど）が、直感的に理解できることが重要である。

キーワードは一般に、システムとのハイレベルな業務上のやりとりを表現するために使用する。例えば、「Cancel_Order」は、注文の存在を確認し、キャンセルリクエストをする人のアクセス権を確認し、キャンセルする注文を表示し、キャンセル確認リクエストを要求することができる。テストアナリストは、キーワードのシーケンス（「Login」、「Select_Order」、「Cancel_Order」など）と、関連するテストデータを使用して、テストケースを記述する。

検討すべき懸念事項は以下の通り。

- キーワードを細かくすればするほど、カバーできるシナリオは具体的になるが、ハイレベル言語の保守が複雑化する可能性がある
- テストアナリストがローレベルのアクション（「ClickButton」、「SelectFromList」など）を指定できるようにすると、キーワードを使ったテストケースはさまざまな状況に対処できるようになる。しかし、これらのアクションは GUI と直接結びついているため、仕様変更があった場合にテストケースのメンテナンスが必要になる可能性がある
- 集約されたキーワードを使用すると、開発は簡単になるが、メンテナンスは複雑になる。例えば、一括でレコードを作成する 6 つのキーワードがあるとする。しかし、6 つのキーワードをすべて呼び出す上位のキーワードを作成することは、全体として最も効率的な方法とは言えないかもしれない
- キーワード言語をいくら分析しても、新しいキーワードやキーワードの変更が必要になることはよくある。キーワードには 2 つの領域がある（つまり、キーワードの背後にあるビジネスロジックと、キーワードを実行するための自動化機能）。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、テスト自動化のメンテナンスコストを大幅に削減することができる。初期設定にコストがかかるかもしれないが、プロジェクトが長く続けば、全体として安くなる可能性が高い。

テスト自動化エンジニアのシラバス[CT_TAE_SYL]には、自動化のためのビジネスプロセスのモデリングに関する詳細が含まれている。

6.2 特定用途のテストツール

この節では、Foundation Level シラバス[CTFL_SYL]で説明されているものを超えて、テクニカルテストアナリストが使用する可能性のあるツールの概要情報を記載する。

なお、ツールに関する詳細な情報は、以下の ISTQB® のシラバスで提供されている。

- モバイルアプリケーションテスト[CT_MAT_SYL]
- 性能テスト[CT_PT_SYL]
- モデルベースドテスト[CT_MBT_SYL]
- テスト自動化エンジニア[CT_TAE_SYL]

6.2.1 フォールトシーディングツール

フォールトシーディングツールは、テスト対象のコードに変更を加えて（事前に定義されたアルゴリズムを用いることがある）、指定されたテストケースによって達成されるカバレッジをチェックする。体系的に適用することで、テストの品質（すなわち、埋め込まれた欠陥の検出能力）を評価し、必要に応じて改善することができる。

フォールトシーディングツールは、一般的にテクニカルテストアナリストが使用するが、新規に開発されたコードをテストする際に開発者が使用することもある。

6.2.2 フォールトインジェクションツール

フォールトインジェクションツールは、ソフトウェアが欠陥に対処できることを確認するために、意図的に誤った入力をソフトウェアに供給する。注入された入力は否定的な条件を引き起こし、エラー処理を実行するはずである（そしてテストされる）。このように、コードの正常な実行フローを乱すことで、コードのカバレッジも向上する。

フォールトインジェクションツールは、一般的にテクニカルテストアナリストが使用するが、新規に開発されたコードをテストする際に開発者が使用することもある。

6.2.3 性能テストツール

性能テストツールには、次のような主に機能がある。

- 負荷の生成
- 与えられた負荷に対するシステムの応答の測定、監視、可視化、分析を提供し、システムやネットワークコンポーネントのリソースの挙動に関する洞察を与える

負荷生成は、事前に定義された運用プロファイル（4.9 節参照）をスクリプトとして実装することで行う。このスクリプトは、最初は単一ユーザー用としてキャプチャする（場合によってはキャプチャ/プレイバックツールを使用する）ことがあり、その後、性能テストツールを使用して指定した運用プロファイル用に実装される。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のセット）ごとのデータのばらつきを考慮しなければならない。

性能ツールは、特定の入力データ量の生成を含むタスクを達成するために、指定された運用プロファイルに従って大量の複数のユーザー（「仮想」ユーザー）をシミュレートすることで負荷を生成する。個々のテスト実行自動化スクリプトと比較して、多くの性能テストスクリプトは、グラフィカルユーザーインターフェースを介したユーザーインタラクションをシミュレートせずに、通信プロトコルレベルでシステムとのユーザーインタラクションを再現する。これにより、通常、テスト中に必要な個別の「セッション」の数を減らすことができる。負荷生成ツールの中には、システムに負荷がかかっているときの応答時間をより詳細に測定するためにユーザーインターフェースを使用してアプリケーションを駆動できるものもある。

性能テストツールは、テスト実行中または実行後の分析を可能にするために、さまざまな測定値を取得する。取得される典型的なメトリクスと提供されるレポートには以下のものを含む。

- テスト中にシミュレートしたユーザー数
- シミュレートしたユーザーが発生させたトランザクションの数、種類、及びトランザクションの到着率
- ユーザーによる特定のトランザクションリクエストに対する応答時間
- 応答時間に対する負荷のレポートとグラフ
- リソースの使用状況に関するレポート（例：最小値、最大値を含む経時的な使用状況）

性能テストツールの実装で考慮すべき重要な要素には、以下のようなものを含む。

- 負荷を生成するために必要なハードウェアとネットワーク帯域幅

- テスト対象システムが使用する通信プロトコルとツールの互換性
- 異なる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・レポート設備

性能テストツールは、その開発に要する労力から、自社開発するよりも入手するのが一般的である。しかし、技術的な制約により利用可能なプロダクトが使えない場合や、商用ツールが提供する負荷プロファイルや設備と比較して提供される負荷プロファイルや設備が単純である場合には、特定の性能ツールを開発するのが適切な場合がある。性能テストツールの詳細については、性能テストシラバス[CT_PT_SYL]にて提供している。

6.2.4 Web サイトのテスト用ツール

Web テストには、オープンソースや商用のさまざまな専用ツールが用意されている。以下のリストは、一般的な Web ベースのテストツールのいくつかの目的を示している。

- ハイパーリンクテストツールは Web サイトをスキャンし、ハイパーリンクが壊れていないこと、または欠落していないことを確認するために使用する
- HTML・XML チェッカーは、Web サイトで作成されるページの HTML や XML の標準適合性をチェックするツールである
- 性能テストツールは、多数のユーザーが接続したときにサーバーがどのように反応するかをテストする
- 異なるブラウザーで動作する軽量の自動実行ツール
- サーバーコードをスキャンし、以前 Web サイトからアクセスされた孤立した（リンクされていない）ファイルをチェックするツール
- HTML 固有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
- 標準違反をチェックするツール。例えば、米国の第 508 条アクセシビリティ基準や、欧州の M/376
- さまざまなセキュリティ上の問題を発見するツール

以下は、オープンソースの Web テストツールの良い情報源である。

- World Wide Web Consortium (W3C) [Web-3] 。この団体は、インターネットの標準を制定し、その標準に対するエラーをチェックするためのさまざまなツールを提供する
- Web Hypertext Application Technology Working Group (WHATWG)[Web-5]。この組織は、HTML の標準を制定する。HTML の妥当性確認を行うツールを持っている。[Web-6]

Web スパイダーエンジンを含む一部のツールでは、ページのサイズやダウンロードに必要な時間、及びページの有無（HTTP エラー404 など）に関する情報の提供もできる。これは、開発者、Web マスター、テスト担当者にとって有益な情報を提供するものである。

テストアナリストや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。

6.2.5 モデルベースドテストを支援するツール

モデルベースドテスト（MBT）は、有限ステートマシンのようなモデルを使用して、ソフトウェア制御システムの意図された実行時の振る舞いを記述する技法である。市販の MBT ツール（[Utting07]を参照）は、ユーザーがモデルを「実行」するためのエンジンを提供することが多い。興味深い実行スレッドは保存され、テストケースとして使用することができる。ペトリネットや状態図のような他の実行可能モデルも MBT をサポートしている。

MBT モデル（及びツール）は、個別の実行スレッドの大規模なセットを生成するために使用することができる。また、MBT ツールは、モデル内で生成される非常に多くの考えられるパスを削減するのに役立つ。これらのツールを使用したテストは、テスト対象のソフトウェアに対して異なる視点を提供することができる。その結果、機能テストでは見落としたかもしれない欠陥が発見される可能性がある。

モデルベースドテストツールの詳細については、モデルベースドテストシラバス[CT_MBT_SYL]にて提供している。

6.2.6 コンポーネントテストとビルドツール

コンポーネントテストやビルド自動化ツールは開発者向けのツールだが、特にアジャイル開発のコンテキストでは、多くの場合、テクニカルテストアナリストが使用・保守する。

コンポーネントテストツールは、コンポーネントのプログラミングに使用される言語に特化したものであることが多い。例えば、プログラミング言語として Java を使用した場合、コンポーネントテストを自動化するために Junit を使用することがある。他の多くの言語には、独自の特別なテストツールがあり、これらは総称して xUnit フレームワークと呼ばれている。このようなフレームワークでは、作成した各クラスに対してテスト対象を生成するため、コンポーネントテストを自動化する際にプログラマーが行うべき作業を簡素化する。

ビルド自動化ツールの中には、コンポーネントが変更されると自動的に新しいビルドを開始するものがある。ビルドが完了した後、他のツールは自動的にコンポーネントテストを実行する。このレベルのビルドプロセスの自動化は、通常、継続的インテグレーション環境で見られる。

このツールを正しくセットアップすると、テスト用にリリースされるビルドの品質に良い影響を与えることができる。プログラマーが行った変更によってビルドにリグレッション欠陥が発生した場合、通常は自動化されたテストのいくつかが失敗し、ビルドがテスト環境にリリースされる前に失敗の原因を直ちに調査するきっかけとなる。

6.2.7 モバイルアプリケーションテストを支援するツール

エミュレーターやシミュレーターは、モバイルアプリケーションのテストを支援するツールとして頻繁に使用されるツールである。

6.2.7.1 シミュレーター

モバイルシミュレーターは、モバイルプラットフォームのランタイム環境をモデル化したものである。シミュレーター上でテストされたアプリケーションは、専用バージョンにコンパイルされ、シミュレーター上では動作するが、実際のデバイス上では動作しない。シミュレーターは、テストにおいて実機の代替として使用するが、一般的には、初期機能テストや負荷テストにおいて多数の仮想ユーザーをシミュレーションする場合に限定される。シミュレーターは（エミュレーターと比較して）比較的シンプルで、エミュレーターよりも高速にテストを実行することができる。しかし、シミュレーター上でテストされるアプリケーションは、配布されるアプリケーションとは異なる。

6.2.7.2 エミュレーター

モバイルエミュレーターは、ハードウェアをモデル化し、物理的なハードウェアと同じランタイム環境を利用する。エミュレーター上でデプロイ・テストするためにコンパイルされたアプリケーションは、実機でも使用することができる。

しかし、エミュレーターは、模倣しようとするモバイル機器とは異なる動作をする可能性があるため、完全に機器を置き換えることはできない。また、（マルチ）タッチや加速度センサーなど、一部の機能がサポートされていない場合もある。これは、エミュレーターの実行に使用されるプラットフォームの制限に起因する。

6.2.7.3 共通点

シミュレーターやエミュレーターは、実機と置き換えることでテスト環境のコストを削減するためによく利用する。シミュレーターやエミュレーターは、通常、開発環境と統合して、アプリケーションの迅速なデプロイ、テスト、モニタリングを可能にするため、開発の初期段階において有用である。エミュレーターまたはシミュレーターを使用するには、それを起動し、そこに必要なアプリケーションをインストールし、実際のデバイスにあるかのようにアプリケーションをテストする必要がある。モバイル OS の開発環境には、通常、それぞれ独自のエミュレーターやシミュレーターがバンドルされている。また、サードパーティー製のエミュレーターやシミュレーターも利用可能である。

通常、エミュレーターやシミュレーターでは、さまざまな取り扱い方をパラメーターとして設定することができる。これらの設定には、異なる速度でのネットワーク・エミュレーション、信号強度やパケット損失、向きの変更、割り込みの発生、GPS 位置データなどが含まれる場合がある。これらの設定の中には、グローバルな GPS 位置や信号強度など、実機で再現することが困難であったり、コストがかかったりするものもあるため、非常に有用なものとなる。

モバイルアプリケーションテストのシラバス[CT_MAT_SYL]には、さらに詳細な情報を提供している。

7. 参考文献

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。

[DO-178C] DO-178C - Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2011
第 2 章

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

[ISO 25010] ISO/IEC 25010: 2011, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models
第 1 章、第 4 章、及び付録 A。
JSTQB 訳注）日本では JIS X 25010 として発行されている。

[ISO 29119] ISO/IEC/IEEE 29119-4:2015, Software and systems engineering - Software testing - Part 4: Test techniques
第 2 章

[ISO 42010] ISO/IEC/IEEE 42010:2011, Systems and software engineering - Architecture description
第 5 章.

[IEC 61508] IEC 61508-5:2010, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Part 5: Examples of methods for the determination of safety integrity levels
第 2 章
JSTQB 訳注）日本では JIS C 0508-5 として発行されている。

[ISO 26262] ISO 26262-1:2018, Road vehicles - Functional safety, Parts 1 to 12.
第 2 章

[IEC 62443-3-2] IEC 62443-3-2:2020, Security for industrial automation and control systems - Part 3-2: Security risk assessment for system design
第 4 章.

7.2 ISTQB® ドキュメント

[CTAL_TTA_OVIEW] ISTQB® Technical Test Analyst Advanced Level Overview v4.0

[CT_SEC_SYL] Security Testing Syllabus, Version 2016

[CT_TAE_SYL] Test Automation Engineer Syllabus, Version 2017

JSTQB 訳注）日本では「テスト技術者資格制度 Advanced Level Specialist シラバス テスト自動化エンジニア 日本語版 Version 2016」として発行されている。

[CTFL_SYL] Foundation Level Syllabus, Version 2018

JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level シラバス日本語版概要 Version

2018」として発行されている。

[CT_PT_SYL] Performance Testing Syllabus, Version 2018

JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level Specialist シラバス 性能テスト担当者 日本語版 Version 2018」として発行されている。

[CT_MBT_SYL] Model-Based Testing Syllabus, Version 2015

[CTAL_TM_SYL] Test Manager 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 テストマネージャ Version 2012」として発行されている。

[CT_MAT_SYL] Mobile Application Testing Syllabus, 2019

JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level Specialist シラバス モバイルアプリケーションテスト担当者 日本語版 Version 2019」として発行されている。

[ISTQB_GLOSSARY] Glossary of Terms used in Software Testing, Version 3.5, 2020

[CT_AuT_SYL] Automotive Software Tester, Version 2018 版

JSTQB 訳注）日本では「テスト技術者資格制度 Foundation Level Specialist シラバス 自動車ソフトウェアテスト担当者 日本語版 Version 2018」として発行されている。

7.3 書籍 と記事

[Andrist20] Björn Andrist and Viktor Sehr, C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition, Packt Publishing, 2020

[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。

[Buwalda01] Hans Buwalda, “Integrated Test Design and Automation”, Addison-Wesley Longman, 2001, ISBN 0-201-73725-6

[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「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。

[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320

[Utting07] 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] http://www.nist.gov (NIST National Institute of Standards and Technology)

[Web-2] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-3] http://www.W3C.org
[Web-4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[Web-5] https://whatwg.org
[Web-6] https://validator.w3.org/
[Web-7] https://dl.acm.org/doi/abs/10.1145/3340433.3342822

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

8. 付録 A:品質特性の概要

以下の表は、現在は置換された ISO 9126-1 標準（2012 年版テクニカルテストアナリストシラバスで使用）と、より新しい ISO 25010 標準（シラバスの最新版で使用）で説明されている品質特性を比較したものである。
機能適合性と使用性は、テストアナリストのシラバスの一部としてカバーされていることに注意すること。

ISO/IEC 25010	ISO/IEC 9126-1	備考
機能適合性	**機能性**	新しい名称は、より正確で、"機能性"の他の意味との混同を避けることができる。
機能完全性		表明されたニーズのカバレッジ
機能正確性	正確性	正確さよりも一般性を重視
機能適切性	合目的性	暗黙のニーズのカバレッジ
	相互運用性	互換性に移動した。
	セキュリティ	今は特性の 1 つ
性能効率性	**効率性**	ISO/IEC 25062 の効率性の定義と矛盾しないように名称を変更した
時間効率性	時間効率性	
資源効率性	資源効率性	
キャパシティ（容量満足性）		新しい副特性
互換性		新特性
共存性	共存性	移植性から移動した。
相互運用性		機能性（テストアナリスト）から移動。
使用性		暗黙の品質問題を明確化した
適切度認識性	理解性	新名称はより正確
習得性	習得性	
運用操作性	運用操作性	
ユーザーエラー防止性		新しい副特性
ユーザーインターフェース快美性	注目性	新名称はより正確
アクセシビリティ		新しい副特性
信頼性	**信頼性**	
成熟性	成熟性	
可用性		新しい副特性
障害許容性	障害許容性	
回復性	回復性	
セキュリティ	**セキュリティ**	過去に副特性がいない
機密性		過去に副特性がいない
インテグリティ		過去に副特性がいない
否認防止性		過去に副特性がいない

責任追跡性		過去に副特性がいない
真正性		過去に副特性がいない
保守性	**保守性**	
モジュール性		新しい副特性
再利用性		新しい副特性
解析性	解析性	
修正性	安定性	変更性と安定性を組み合わせたより正確な名称
試験性	試験性	
移植性	**移植性**	
適応性	環境適応性	
設置性	設置性	
	共存性	互換性に移動
置換性	置換性	