テスト技術者資格制度
Advanced Level シラバス

テスト自動化エンジニア

Version 2016.J01

International Software Testing Qualifications Board

Advanced Level Test Automation Working Group: Bryan Bakker, Graham Bath, Armin Born, Mark Fewster, Jani Haukinen, Judy McKay, Andrew Pollner, Raluca Popescu, Ina Schieferdecker; 2016.

改訂履歴

ISTQB®

バージョン	日付	摘要
第 1 版	2015 年 8 月 13 日	第 1 版
第 2 版	2015 年 11 月 5 日	LO の対応付けと場所の変更
第 3 版	2015 年 12 月 17 日	LO の見直し
ベータドラフト版	2016 年 1 月 11 日	ドラフトの改訂
ベータ版	2016 年 3 月 18 日	ベータ版の発行
シラバス 2016	2016 年 10 月 21 日	正式版の発行

JSTQB®

バージョン	日付	摘要
Version 2016.J01	2020 年 9 月 30 日	Version 2016の日本語翻訳版

目次

謝辞

このドキュメントは、International Software Testing Qualifications Board Advanced Level 作業部会のコアチームメンバーにより執筆された。

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のメンバーによる提案と意見に感謝したい。

このモジュールの Advanced Level シラバスの完成時において、Advanced Level 作業部会 - テスト自動化のメンバーは以下のとおりである。Bryan Bakker、Graham Bath (Advanced Level Working Group Chair)、Armin Beer、Inga Birthe、Armin Born、Alessandro Collino、Massimo Di Carlo、Mark Fewster、Mieke Gevers、Jani Haukinen、Skule Johansen、Eli Margolin、Judy McKay (Advanced Level Working Group Vice Chair)、Kateryna Nesmyelova、Mahantesh (Monty) Pattan、Andrew Pollner (Advanced Level Test Automation Chair)、Raluca Popescu、Ioana Prundaru、Riccardo Rosci、Ina Schieferdecker、Gil Shekel、Chris Van Bael

このシラバスのコアチーム著作者は以下のとおりである。Andrew Pollner (Chair)、Bryan Bakker、Armin Born、Mark Fewster、Jani Haukinen、Raluca Popescu、Ina Schieferdecker

次のメンバーが、本シラバスのレビュー、意見表明および投票に参加した（アルファベット順）。Armin Beer、Tibor Csöndes、Massimo Di Carlo、Chen Geng、Cheryl George、Kari Kakkonen、Jen Leger、Singh Manku、Ana Paiva、Raluca Popescu、Meile Posthuma、Darshan Preet、Ioana Prundaru、Stephanie Ulrich、Erik van Veenendaal、Rahul Verma

本ドキュメントは、2016 年 10 月 21 日に開催された ISTQB の総会で正式に発行された。

日本語訳については、Japan Software Testing Qualifications Boardメンバーおよ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:
伊藤 由貴（ベリサーブ）
江添 智之（バルテス）
清水 歩（日本ナレッジ）
須原 秀敏（ベリサーブ）
髙橋 寿一（ロジギアジャパン）
宮北 裕子（バルテス）

0. イントロダクション

0.1 本書の目的

本シラバスは、テスト自動化 - エンジニアリング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、本シラバスを以下の趣旨で提供する。

- メンバー委員会に対し、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。メンバー委員会は、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追加することができる。
- 認定委員会に対し、本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。
- 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。
- 受験志願者に対し、試験準備の目的で提供する（研修コースの一部として、または独立した形態で実施）。
- 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ほか、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りISTQBから承認された場合に限り、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 本書の範囲

0.2.1 範囲内

本書は、テスト自動化ソリューションの設計、開発、保守におけるテスト自動化エンジニア（TAE）のタスクについて説明する。動的な機能テストにおける自動化の概念、手法、ツール、プロセスと、それらのテストとテストマネジメント、構成管理、欠陥マネジメント、ソフトウェア開発プロセス、品質保証の関係に重点を置く。

通常、説明する手法はさまざまなソフトウェアライフサイクルアプローチ（アジャイル、シーケンシャル、インクリメンタル、イテレーティブなど）、ソフトウェアシステムの種類（組込み、分散、モバイルなど）、テストタイプ（機能テストや非機能テスト）に適用できる。

0.2.2 範囲外

以下の側面は、このテスト自動化 – エンジニアリングシラバスの範囲外である。

- テストマネジメント、テスト仕様書の自動作成、テストの自動生成
- テスト自動化マネージャー（TAM）がテスト自動化ソリューションの開発と進化について計画、管理、調整を行うタスク
- 非機能テスト（パフォーマンスなど）の自動化の詳細
- 静的解析（脆弱性解析など）の自動化と静的テストツール
- ソフトウェアエンジニアリング手法やプログラミングの教育（テスト自動化ソリューションを実現するためにどの標準を使うべきか、どのようなスキルが必要かなど）

- ソフトウェア技術の教育（テスト自動化ソリューションを実装する際にどのスクリプト技術を使うか、など）
- ソフトウェアテストの製品およびサービスの選定（テスト自動化ソリューションにどの製品やサービスを使うか等々）

0.3 テスト技術者資格制度 Advanced Level テスト自動化エンジニア

0.3.1 期待事項

Advanced Level 資格認定は、Foundation Level で取得した知識とスキルをベースに、1 つ以上の特定領域の専門性を高めることを希望する者を対象とする。Advanced Level Specialist で提供するモジュールは、広範なテストに関するトピックをカバーする。

テスト自動化エンジニアとは、テストについて汎用的で幅広い知識を持ち、テスト自動化という特定の領域について深く理解している者を言う。深い理解とは、組織および／またはプロジェクトが機能テストのために自動化ソリューションを設計、開発、保守する際に、方向付けとして使用できるテスト自動化理論とプラクティスについて、十分な知識を持っていることと定義される。

Advanced Level Modules Overview[ISTQB-AL-Modules]ドキュメントでは、このモジュールのビジネス成果について説明する。

0.3.2 受験および更新の要件

Advanced Level の一般的な受験基準は、ISTQB Web サイト[ISTQB-Web]の Advanced Level 節で説明されている。

Advanced Level テスト自動化エンジニア認定試験を受験するには、この一般的な受験基準に加えて、ISTQB Foundation Level 認定[ISTQB-CTFL]を取得している必要がある。

0.3.3 知識レベル

本シラバスの学習の目的は、認識しやすいように各章の先頭に示されている。各章に割り当てられた学習の目的に従って、シラバスのそれぞれのトピックを試験する。

学習の目的に割り当てられている知識レベル（「K レベル」）は、ISTQB Web サイト[ISTQB-Web]で説明されている。

0.3.4 試験

Advanced Level 認定試験は、本シラバスに加え、Foundation Level シラバス[ISTQB-FL]に基づくものとする。試験問題に対する解答は両シラバスの複数の節を基にしている場合がある。

試験の形式は、ISTQB の Web サイト[ISTQB-Web]の Advanced Level 節で説明されている。ISTQB Web サイトには、試験の受験に役立つ情報も含まれている。

0.3.5 認定審査

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。

ISTQB Web サイト[ISTQB-Web]の Advanced Level 節では、認定審査される教育機関に適用される固有の規則が説明されている。

0.4 標準パートと情報提供パート

シラバスの標準パートは試験対象である。具体的には以下が含まれる。

- 学習の目的
- キーワード

シラバスのこれ以外の部分は情報提供で、学習の目的について詳述している。

0.5 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ようになっている。

- 各知識領域の学習の目的(達成すべき知識レベルの学習の成果とマインドセットについての説明。この部分が標準にあたる)
- 教えるべき情報の一覧(教えるべき主要な概念の説明、一般に認知されている文献や標準などの情報源、必要に応じてその他の情報源への参照を含む。この部分が情報提供にあたる)

本シラバスの内容はテスト自動化エンジニアリングの全知識領域の説明ではない。詳細レベルは、Advanced Level の認定トレーニングコースでカバーされることを示している。

0.6 本シラバスの構成

本シラバスには 8 つの章がある。各章の一番上の見出しは、章の学習時間を指定している。次に例を示す。

3. 汎用テスト自動化アーキテクチャ　270 分

これは、第 3 章の教材の説明に 270 分を想定していることを示す。
各章の始めには、具体的な学習の目的を示す。

0.7 用語、定義、頭字語

ソフトウェアの文献で使われている多くの用語は、厳密な区別なく使用されている場合がある。本 Advanced Level シラバスにおける定義は、ISTQB が公開しているソフトウェアテスト標準用語集[ISTQB-Glossary]に記載されている。

本 Advanced Level シラバスの各章の始めに示されているキーワードは、[ISTQB-Glossary]で定義される。

本書では以下の頭字語を使用する。
CLI　コマンドラインインターフェース(Command Line Interface)
EMTE　同等の手動テスト工数(Equivalent Manual Test Effort)

gTAA	汎用テスト自動化アーキテクチャ（Generic Test Automation Architecture。テスト自動化ソリューションの全体概要を提供する）
GUI	グラフィカルユーザーインターフェース（Graphical User Interface）
SUT	テスト対象システム（System Under Test）、テスト対象も参照のこと
TAA	テスト自動化アーキテクチャ（Test Automation Architecture。TASのアーキテクチャを定義するためにgTAAを具体化したもの）
TAE	テスト自動化エンジニア（Test Automation Engineer。成果物であるTASの実装と、その保守や技術改善を含む、TAAの設計についての責任者）
TAF	テスト自動化フレームワーク（Test Automation Framework。テスト自動化に必要な環境で、テストハーネスやテストライブラリなどの成果物を含む）
TAM	テスト自動化マネージャー（Test Automation Manager。TASの開発と進化の計画および監督についての責任者）
TAS	テスト自動化ソリューション（Test Automation Solution。TAAを具体化/実装したもので、テストハーネスやテストライブラリなどの成果物を含む）
UI	ユーザーインターフェース（User Interface）

1. テスト自動化の概要と目的 - 30 分

キーワード

APIテスト、CLIテスト、GUIテスト、テスト対象システム、テスト自動化アーキテクチャ、テスト自動化フレームワーク、テスト自動化戦略、テスト自動化、テストスクリプト、テストウェア

「テスト自動化の概要と目的」の学習の目的

1.1 テスト自動化の目的

ALTA-E-1.1.1　（K2）テスト自動化の目的、利点、欠点、制限を説明する

1.2 テスト自動化の成功要因

ALTA-E-1.2.1　（K2）テスト自動化プロジェクトの技術的成功要因を特定する

1.1 テスト自動化の目的

ソフトウェアテストにおいて、テスト自動化（自動テスト実行を含む）とは以下のタスクの1つ以上を指す。
- 専用ソフトウェアツールを使用してテストの事前条件を制御および設定する
- テストを実行する
- 実行結果と期待結果を比較する

テストに使用するソフトウェアとテスト環境内対象システム（SUT）自身を分離し、依存関係を最低限にとどめるのは良い習慣である。テストに用いるソフトウェアをSUTに導入しなければならない、組込みシステムなどの例外もある。

テスト自動化は、異なるバージョンのSUTや環境で多くのテストケースを一貫性をもって繰り返し実行することに役立つものと期待されている。しかし、テスト自動化は人手を介さずにテストスイートを実行するメカニズムだけにとどまらない。テスト自動化とは、以下を含むテストウェアの設計プロセスを包含する。
- ソフトウェア
- 文書
- テストケース
- テスト環境
- テストデータ

テストウェアは、以下を含むテスト活動を必要とする。
- 自動テストケースの実装
- 自動テスト実行の監視および制御
- 自動テスト結果の解釈、報告、記録

テスト自動化には、SUTとの相互作用において、以下のような異なるアプローチが存在する。
- SUTのクラス、モジュール、ライブラリの公開インターフェースを使用したテスト（APIテスト）
- SUTのユーザーインターフェースを使用したテスト（GUIテスト、CLIテストなど）
- サービスやプロトコルを使用したテスト

テスト自動化の目的には、以下が含まれる。
- テストの効率性の向上
- 機能カバレッジの拡大
- 総テストコストの削減
- 手動テスト担当者が行えないテストの実施
- テスト実行期間の短縮
- テスト実行頻度の向上およびテストサイクルに要する時間の短縮

テスト自動化の利点には、以下が含まれる。
- より多くのテストをビルドごとに実行可能にする
- 手動では行えないテストを作成できるようにする（リアルタイム、リモート、並列テスト）
- 手動より複雑なテストの実行
- テスト実行の高速化
- オペレーターのミスによるテスト結果への影響を低減
- より効果的・効率的にテストリソースを使用する
- ソフトウェア品質に関するフィードバックの迅速化
- システムの信頼性向上（再現性、一貫性など）

- テストの一貫性の向上

テスト自動化の欠点には、以下が含まれる。

- 追加コストが必要
- TASのセットアップのための初期投資
- 追加技術が必要
- チームに開発と自動化のスキルが必要
- 継続的なTASの保守が必要
- テストの目的から逸脱する可能性がある（例えば、テストの実行を犠牲にしてでも、テストケースの自動化に集中する）
- テストが複雑になる
- 自動化によって新たなエラーが引き起こされる可能性がある

テスト自動化の制限には、以下が含まれる。

- すべての手動テストを自動化できない
- 自動化によってチェックできるのは、ツールが解釈できる結果のみ
- 自動化によってチェックできるのは、あらかじめ定義された期待結果によって検証可能な実行結果だけである
- 探索的テストを自動化に置き換えることはできない

1.2 テスト自動化の成功要因

以下の成功要因は現在実施中の自動化プロジェクトに適用でき、長期的なプロジェクトの成功への影響に主眼を置いている。そのためここでは、パイロット段階のテスト自動化プロジェクトの成功に影響を与える要因は考慮していない。

テスト自動化の主な成功要因には、以下が含まれる。

テスト自動化アーキテクチャ（TAA）

テスト自動化アーキテクチャ（TAA）は、対象ソフトウェア製品のアーキテクチャと非常に密接に関連している。アーキテクチャがサポートする機能要件と非機能要件を明確にする必要がある。通常これが最も重要な要件となる。

多くの場合、TAAは保守性、性能、習得性を考慮して設計される。（これらの詳細や、その他の非機能特性については、ISO/IEC 25000:2014 を参照のこと。）SUTのアーキテクチャを理解しているソフトウェア技術者を関与させることも有用である。

SUTの試験性

SUTは自動テストのテストのしやすさを考慮して設計する必要がある。GUIテストの場合、SUTではできる限り多くの GUI操作やデータをGUI画面構成から切り離さなければならない。APIテストの場合、テストを行えるようにするために、公開されたクラス、モジュール、コマンドラインインターフェース数を増やさなければならない場合がある。

SUTのテスト可能なパーツを最初のターゲットにする必要がある。通常、テスト自動化の重要な成功要因は、自動テストスクリプトの実装しやすさにある。この目的に留意し、またこの考え方を実証するため、テスト自動化エンジニア（TAE）は、簡単に自動テストできるSUTのモジュールまたはコンポーネントを特定し、そこから着手する必要がある。

テスト自動化戦略

現実的で一貫性のあるテスト自動化戦略では、SUTの保守性と一貫性に対応する。

SUTの既存部分と新規部分の両方に対して、同じ自動化戦略を適用できない場合がある。自動化戦略を作成する際は、コードのさまざまな部分にそれを適用することのコスト、利点、リスクを考慮する。。

ユーザーインターフェースのテストと APIテストの両方を自動化する場合はその結果の一貫性について考慮する必要がある。

テスト自動化フレームワーク（TAF）

使いやすく、ドキュメントが充実しており保守性の高いテスト自動化フレームワーク（TAF）を維持することにより、自動テストに対して一貫性のあるアプローチをとることができる。

使いやすく保守性の高い TAFを作るためには、以下のことを行わなければならない。

- レポート機能を実装する: テストレポートでは、SUTの品質についての情報（成功/失敗/エラー/未実行/異常終了、統計情報など）を提供する必要がある。レポートは、関連するテスト担当者、テストマネージャー、開発担当者、プロジェクトマネージャーなどのステークホルダーが品質の概要を把握できるよう情報を提供する必要がある。
- 簡単なトラブルシューティングを可能にする: テストの実行と記録に加え、TAFは失敗したテストのトラブルシューティングを行う簡単な方法を提供する必要がある。テストは、以下の理由で失敗する可能性がある。
  - SUTで見つかった故障
  - TASで見つかった故障
  - テスト自身またはテスト環境の問題
- テスト環境に適切に対応する: テストツールはテスト環境の一貫性に依存する。自動テストには専用のテスト環境が必要である。テスト環境やテストデータをまったく制御できない場合、テスト実行の要件を満たさず、誤った実行結果の生成につながる可能性がある。
- 自動テストケースを明文化する: テスト自動化の目的を明らかにする必要がある。たとえば、アプリケーションのどの部分をどの程度テストするか、どの属性をテストするかなど（機能および非機能）である。この点は明確に説明し、文書化しなければならない。
- 自動テストをトレースする: テスト自動化エンジニアがテストケースの個々のステップをトレースできるように、TAFはトレースをサポートする必要がある。
- 簡単な保守を実現する: 保守がテスト自動化作業の大部分とならないよう、自動テストケースを簡単に保守できるのが理想的である。さらに、保守作業はSUTに対して行う変更の規模に比例する必要がある。このためには、ケースを簡単に分析、変更、拡張できなければならない。加えて、自動化テストウェアを頻繁に再利用し、変更が必要になる項目数を最小限に抑える必要がある。
- 自動テストを最新の状態に保つ: 新しい要件や変更された要件によってテストやテストスイート全体が失敗した場合は、失敗したテストを無効にするのではなく、修正する。

- 導入の計画を立てる: テストスクリプトを簡単に導入、変更、再導入できるようにする。
- 必要に応じてテストを削除する: 自動テストスクリプトが不要になった場合に、簡単に削除できるようにする。
- SUTの監視と復旧を行う: 実際の現場では、1 つまたは一連のテストケースを継続的に実行するためには、SUTを継続的に監視しなければならない。SUTで致命的なエラー(クラッシュなど)が発生した場合、TAFはエラーから回復し、問題のあるテストケースをスキップして、次のケースからテストを再開できなければならない。

テスト自動化コードの保守は煩雑になる可能性があり、テスト用のコードの量がSUTのコードと同程度の量になることも珍しくない。そのため、テストコードの保守性は最重要である。これは、使用されるテストツール、使用される検証の種類、保守しなければならないテストウェア成果物(テスト入力データ、テストオラクル、テストレポートなど)に違いがあるためである。

これらの保守に関する考慮事項と行うべき重要な事項があることを踏まえた上で、次のような事は行うべきではない。

- インターフェースの影響を受けやすいスクリプト(グラフィカルインターフェースや APIの重要でない部分の変更による影響を受けるスクリプト)を作成しない。
- データの変更による影響を受けやすい、または特定のデータ値に大きく依存するテスト自動化(他のテスト出力に依存するテスト入力など)を行わない。
- コンテキスト(オペレーティングシステムの日付や時刻、オペレーティングシステムの多言語化のパラメーター、別のアプリケーションのコンテンツなど)の影響を受けやすい自動化環境を作成しない。この場合、環境を制御できるように、必要に応じてテスト用のスタブを使うのが望ましい。

上に記した成功要因の数が多いほど、テスト自動化プロジェクトが成功する可能性も高くなる。要因をすべて満たしていなくても構わない。実際すべての要因が満たされることはまれである。重要なことは、テスト自動化プロジェクトを開始する前に、該当する要因と該当しない要因を考慮し、選択したアプローチのリスクとプロジェクトの状況を念頭に置いて、プロジェクトが成功する可能性を分析することである。TAAが整ったら、それに該当しない項目やさらに作業が必要な項目を調査することが重要である。

2. テスト自動化の準備 - 165 分

キーワード

試験性、ドライバ、干渉のレベル、スタブ、テスト実行ツール、テストフック、テスト自動化マネージャー

「テスト自動化の準備」の学習の目的

2.1 テスト自動化に影響するSUT要因

ALTA-E-2.1.1 （K4）テスト対象システムを分析して適切な自動化ソリューションを判断する

2.2 ツールの評価と選定

ALTA-E-2.2.1 （K4）テスト自動化対象プロジェクトで用いるテスト自動化ツールを検討し、技術的所見や推奨事項を報告する

2.3 試験性と自動化を考慮した設計

ALTA-E-2.3.1 （K2）SUTに適用できる「試験性を考慮した設計」手法および「テスト自動化を考慮した設計」手法を理解する

2.1 テスト自動化に影響するSUT要因

SUTの状況や環境を評価する場合、テスト自動化に影響する要因を特定して適切な対応をとらなければならない。これには、以下の内容が含まれる。

- SUTインターフェース
  自動化したテストケースは、SUTでのアクションを呼び出す。このためには、SUTを制御できるインターフェースをSUTが提供しなければならない。これは UIコントロールで実現できるが、より低レベルなソフトウェアインターフェースでも実現できる。さらに、一部のテストケースでは通信レベルでのインターフェースを通したコミュニケーションが可能である(TCP/IP、USB、独自のメッセージングプロトコルなど)。

  SUTを分割すると、異なるテストレベルそれぞれに対してテスト自動化が可能になる。SUTが適切にサポートする場合に限り、特定のレベル(コンポーネントレベル、システムレベルなど)でテストを自動化することも可能である。例えば、コンポーネントレベルでテストに利用できるユーザーインターフェースが存在しないことがある。その場合、テストを自動化するには別のソフトウェアインターフェース(テストフックとも呼ばれる)が利用できる必要がある。このソフトウェアインターフェースは独自のカスタマイズが必要になることもある。

- サードパーティ製ソフトウェア
  多くの場合、SUTは開発している組織が書いたものだけではなく、サードパーティが提供するソフトウェアを含んでいる。状況によっては、このサードパーティ製ソフトウェアのテストが必要になる。テスト自動化を行うのが適切な場合は、APIの使用など、別の自動化ソリューションが必要になる可能性がある。

- 干渉のレベル
  テスト自動化のアプローチが異なれば(異なるツールを使用すれば)、干渉のレベルも異なる。自動テストのためにSUTに対して行わなければならない変更の数が多いほど、干渉のレベルも高くなる。既存のUI要素を使う場合は、干渉のレベルは低くなるが、専用のソフトウェアインターフェースを使う場合、高い干渉のレベルが必要になる。SUTのハードウェア要素(キーボード、スイッチ、タッチスクリーン、通信インターフェースなど)を使う場合は、干渉のレベルがさらに高くなる。

  干渉のレベルが高いことによる問題は、誤警告のリスクである。テストの干渉のレベルが高い場合、TASが失敗を示したとしても、ソフトウェアシステムが実際の稼働環境で使われている場合に同様の事象が発生する可能性は低い。通常は、高い干渉のレベルで行うテストほど、テスト自動化のアプローチとして簡単な解決策となる。

- 異なるSUTアーキテクチャ
  異なるSUTアーキテクチャには、異なるテスト自動化ソリューションが必要になる場合がある。例えばCOMテクノロジーを使う C++ で書かれたSUTは、Pythonで書かれたSUTとは異なるアプローチが必要である。このような異なるアーキテクチャを同じテスト自動化戦略で扱うことが可能な場合もあるが、それには両方をサポートしたハイブリッド戦略が必要になる。

- SUTのサイズと複雑度
  現在のSUTのサイズと複雑度を考慮して、今後の開発計画を立てる。小さく単純なSUTには、複雑で非常に柔軟度の高いテスト自動化アプローチは適さず、単純なアプローチの方が適している可能性がある。逆に、大規模で複雑なSUTで小さく単純なアプローチを用いるのは賢明でない場合がある。ただし、複

雑なSUTでも小さく単純なものから始めるのが適切な場合もある。しかし、これは一時的なアプローチとすべきである（詳細については、第 3 章を参照）。

ここに記載するいくつかの要因（サイズと複雑度、利用できるソフトウェアインターフェースなど）は、SUTが利用できるようになってから明らかになる。ただし、ほとんどの場合、テスト自動化の開発はSUTが利用可能になる前に開始すべきである。その場合、いくつかの事項を見積る必要がある。または、TAEが必要なソフトウェアインターフェースを指定することもできる（詳細については、2.3節 を参照）。

SUTがまだ存在しない場合でも、テスト自動化計画を開始することができる。次に例を示す。

- 要件（機能または非機能）がわかっている場合、その要件とそれをテストする手段から自動化の候補を選定できる。自動化の計画はその候補から始まる。これには、自動化要件の特定やテスト自動化戦略の決定も含まれる。
- アーキテクチャと設計が作成されている間に、テストをサポートするソフトウェアインターフェースの設計に着手できる。

2.2 ツールの評価と選定

ツールの選定および評価プロセスの責任者は、テスト自動化マネージャー（TAM）である。ただし、TAEも TAM への情報提供や評価および選定作業の実施に関与する。ツールの評価および選定プロセスの概念については、Foundation Level で取り入れられている。このプロセスの詳細は、Advanced Level – テストマネージャーシラバス［ISTQB-AL-TM］に記載されている。

TAEはツールの評価および選定プロセスの全体に関与するが、特に以下の作業を中心に行う。

- 組織の成熟度評価およびテストツールのサポート体制の評価
- 期待するテストツールのサポートに関する適切な評価
- 候補ツールの情報収集および選定
- 目的やプロジェクトの制約に対するツール情報の分析
- 具体的なビジネスケースに基づく費用対効果の見積り
- 適切なツールの推薦
- ツールとSUTコンポーネントとの互換性の判断

機能テスト自動化ツールは、自動化プロジェクトで生じる想定や状況のすべてを満たせるとは限らない。そのような問題の一部（当然ながら、すべてを網羅したものではない）を以下に示す。

問題	例	考えられる解決策
テスト自動化ツールのインターフェースが、既に導入されている他のツールと連携しない	• テストマネジメントツールがアップデートされ、接続インターフェースが変更された • プリセールスサポートの情報が誤っており、レポートツールに転送できない情報がある	• アップデートの前に注意してリリースノートを確認し、大規模な移行の場合は本番環境に移行する前にテストを行う • 実際のSUTを使ったツールのデモをオンサイトで行わせる • ベンダーやユーザーコミュニティフォーラムによるサポートを検討する
一部のSUTの依存性が変更され、テストツールがサポートしないものが含まれるようになった	• 開発部門が最新バージョンの Java にアップデートした	• 開発環境/テスト環境とテスト自動化ツールでアップグレードを同期する
GUIのオブジェクトがキャプチャできない	• オブジェクトは表示されているが、テスト自動化ツールから操作できない	• 開発の際には、よく知られている技術やオブジェクトのみを使用するよう心がける • テスト自動化ツールを購入する前にパイロットプロジェクトを行う • 開発担当者にオブジェクトの標準を定義させる
ツールが非常に複雑に見える	• ツールには大量のフィーチャーセットがあるが、その一部しか使う予定がない	• ツールバーから不要な機能を削除してフィーチャーセットを制限する方法を探す • ニーズを満たすライセンスを選択する • 必要なフィーチャーに特化した別のツールを探す
他のシステムと競合する	• 他のソフトウェアをインストールすると、テスト自動化ツールが動作しなくなる（またはその逆）	• インストール前にリリースノートや技術要件を読む • サプライヤーから他のツールに影響がないことを確認する • ユーザーコミュニティフォーラムに質問する
SUTへの影響	• テスト自動化ツールの使用中または使用後に、SUTの反応が変わる（応答時間が長くなるなど）	• SUTを変更する（ライブラリのインストールなど）必要がないツールを使用する
コードへのアクセス	• テスト自動化ツールがソースコードの一部を変更する	• ソースコードを変更する（ライブラリのインストールなど）必要がないツールを使用する

問題	例	考えられる解決策
リソースの制限（主に組込み環境）	• テスト環境の空きリソースが限られているか、リソースが枯渇している（メモリなど）	• リリースノートを読み、ツール開発元と話し合い、環境が問題につながらないことを確認する • ユーザーコミュニティフォーラムに質問する
アップデート	• アップデートですべてのデータが移行されない、または既存の自動テストスクリプトやデータ、構成が破損する • アップグレードに別の（よりよい）環境が必要	• テスト環境でアップグレードのテストを行い、開発元から移行が問題なく行えることを確認する • アップデートの前提条件を読み、アップデートする価値があるかを判断する • ユーザーコミュニティフォーラムによるサポートを求める
セキュリティ	• テスト自動化ツールを使うにあたり、テスト自動化エンジニアに提供されていない情報や権限を必要とする	• テスト自動化エンジニアにアクセス権を与える必要がある
異なる環境やプラットフォームでの互換性がない	• テスト自動化がすべての環境やプラットフォームで動作するとは限らない	• ツールの独立性を最大限にするように自動テストを実装し、複数のツールを使うことによるコストを最小限に抑える

2.3 試験性と自動化を考慮した設計

SUTの試験性（SUTの制御や観測を可能にするなど、テストをサポートするソフトウェアインターフェースの可用性）は、SUTの他のフィーチャーの設計や実装と並行して設計、実装すべきである。試験性はシステムの非機能要件の 1 つにすぎないため、試験性の設計、実装はソフトウェアアーキテクトが行うことができるが、多くの場合、TAE自身が行うか、TAEが関与して行う。

試験性を考慮した設計は、いくつかの要素によって構成される。

- 観測性: SUTは、システムの内部を把握できるインターフェースを提供する必要がある。その後、テストケースはそのインターフェースを使い、意図した振る舞いと実際の振る舞いが等しいかどうかなどをチェックする。
- 制御（性）: SUTは、SUTを操作するインターフェースを提供する必要がある。これには、UI要素、関数呼び出し、通信要素（TCP/IP、USBプロトコルなど）、電気信号（物理スイッチ）などがある。
- 明確に定義されたアーキテクチャ: 試験性を考慮した設計で 3 つ目の重要な点は、すべてのテストレベルにおいて制御と可視性を実現し、明確でわかりやすいインターフェースを提供するアーキテクチャである。

TAEは、自動テストを含め、効果的（適切な領域をテストして重大なバグを見つける）かつ効率的（大量の作業を発生させない）にSUTをテストできる方法を検討する。特定のソフトウェアインターフェースが必要な場合は、常に

TAEがそれを明示し、開発担当者が実装しなければならない。これは試験性を定義するために重要である。また、必要に応じて開発作業の予定を立てて予算を確保できるように、プロジェクトの早い段階で追加のソフトウェアインターフェースを定義する際にも重要になる。

テストをサポートするソフトウェアインターフェースの例には、以下のようなものがある。

- 最新のスプレッドシートによる強力なスクリプト機能。
- スタブまたはモックを適用してまだ入手していないソフトウェアやハードウェア、高価すぎて購入できないソフトウェアやハードウェア（電子金融取引、ソフトウェアサービス、専用サーバー、電子基板、機械部品など）をシミュレートすることにより、実際のインターフェースが存在しないソフトウェアのテストが可能になる。
- ソフトウェアインターフェース（スタブおよびドライバ）を使ってエラー条件をテストすることが可能である。内部ハードディスクドライブ（HDD）を搭載したデバイスを例に挙げる。この HDDを制御するソフトウェア（ドライバと呼ばれる）には、HDDの故障や摩耗に対するテストを行える必要がある。これを行うためにHDDが故障するのを待つのは、効率のよい（あるいは信頼性が高い）方法とは言えない。HDDの欠陥や遅延をシミュレートするソフトウェアインターフェースを実装すれば、ドライバソフトウェアが正常に動作すること（エラーメッセージ、リトライなど）を検証できる。
- 利用できる UIがまだ存在しない場合は、別のソフトウェアインターフェースを使ってSUTをテストすることも可能である（この方法は、UIを使うよりも優れたアプローチだと見なされることが多い）。多くの場合、組込みソフトウェアは、デバイスの温度を監視し、温度が一定レベルを超えたときに冷却機能を起動する必要がある。これは、温度を指定するソフトウェアインターフェースを使えば、ハードウェアがなくてもテストできる。
- SUTの状態の振る舞いを評価する場合、状態遷移テストが使われる。SUTが正しい状態にあるかどうかをチェックするには、この目的を実現するためにカスタマイズされたソフトウェアインターフェースを経由して問い合わせる（ただし、これにはリスクもある。2.1節の「干渉のレベル」を参照）。

自動化の設計では、以下を考慮すべきである。

- 早い段階で既存のテストツールとの互換性を確立すべきである。
- テストツールの互換性の問題は重大である。これは、重要な機能のテストの自動化可否に影響する可能性がある（例えば、グリッドコントロールと互換性がない場合、そのコントロールを使うすべてのテストができなくなる）ためである。
- プログラムコードの開発や APIの呼び出しが必要な解決策もある。

試験性を考慮した設計は、優れたテスト自動化アプローチにとって最重要であり、手動テスト実行の場合にもメリットになる場合がある。

3. 汎用テスト自動化アーキテクチャ - 270 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、汎用テスト自動化アーキテクチャ、キーワード駆動テスト、線形スクリプティング、モデルベースドテスト、プロセス起動スクリプティング、構造化スクリプティング、テスト適合レイヤー、テスト自動化アーキテクチャ、テスト自動化フレームワーク、テスト自動化ソリューション、テスト定義レイヤー、テスト実行レイヤー、テスト生成レイヤー

「汎用テスト自動化アーキテクチャ」の学習の目的

3.1 gTAAの概要

ALTA-E-3.1.1 （K2）汎用 TAA（gTAA）の構造を説明する

3.2 TAAの設計

ALTA-E-3.2.1 （K4）プロジェクトに適切なTAAを設計する
ALTA-E-3.2.2 （K2）TAA内の各レイヤーが果たす役割を説明する
ALTA-E-3.2.3 （K2）TAAの設計に当たっての考慮事項を理解する
ALTA-E-3.2.4 （K4）TASの実装、使用、保守要件の要素を分析する

3.3 TASの開発

ALTA-E-3.3.1 （K3）gTAAのコンポーネントを適用して専用のTAAを構築する
ALTA-E-3.3.2 （K2）コンポーネントの再利用性を確認する際に考慮すべき要素を説明する

3.1 gTAAの概要

テスト自動化エンジニア（TAE）の役割は、テスト自動化ソリューション（TAS）の設計、開発、実装、保守である。各ソリューションの開発にあたっては、同じようなタスクを行い、同じような質問に答え、同じような問題に優先順位をつけて対処する必要がある。テスト自動化で繰り返し発生するこのような概念、手順、アプローチは、gTAAと略される汎用テスト自動化アーキテクチャの基本となる。

gTAAは、gTAAのレイヤー、コンポーネント、インターフェースを表す。これらは特定のTAS向けのTAAとして具体化される。これにより、構造化されたモジュール形式のアプローチを利用して、以下の方法でテスト自動化ソリューションを構築できるようになる。

- 内部および外部で開発したコンポーネントによってTASを具体化できるように、TASのコンセプト、レイヤー、サービス、インターフェースを定義する
- シンプルなコンポーネントを利用して効果的かつ効率的なテスト自動化を行えるようにする
- 各種ソフトウェアの製品ラインや製品ファミリー、およびソフトウェア技術やツールが異なるものを含め、別のTASの開発やTASの改良の際 にテスト自動化コンポーネントを再利用する
- TASの保守や改良を容易にする
- TASユーザー用の基本的なフィーチャーを定義する

TASは、テスト環境（およびその成果物）とテストスイート（テストデータを含む一連のテストケース）の両方で構成する。テスト自動化フレームワーク（TAF）は、TASを実現するために使用でき、テスト環境の具体化のサポートおよびツール、テストハーネス、サポートライブラリを提供する。

TASのTAAは、TASを簡単に開発、改良、保守できるよう以下の原理に従うことが推奨される。

- 単一責任: あらゆるTASコンポーネントは、単一の責任を持たなければならない。また、その責任はコンポーネント内に完全にカプセル化されなければならない。つまり、TASのすべてのコンポーネントは、キーワードやデータの生成、テストシナリオの作成、テストケースの実行、結果記録、実行レポートの生成など、厳密に 1 つのことを担当すべきである。
- 拡張性（B. Myerによるオープン/クローズドの原則なども参照）: あらゆるTASコンポーネントは、拡張に対してオープンでなければならないが、変更に対してクローズドでなければならない。この原則は、下位互換性機能に影響を与えずにコンポーネントの振る舞いを変更または強化できる必要があることを意味している。
- 置換（B. Liskovによる置換原則なども参照）: あらゆるTASコンポーネントは、TASの全般的な振る舞いに影響を与えずに置換可能でなければならない。コンポーネントは 1 つまたは複数のコンポーネントで置き換えることができるが、外から見える振る舞いは同じでなければならない。
- コンポーネントの分離（R.C. Martin によるインターフェース分離の原則なども参照）: 汎用的で多目的なコンポーネントよりも、具体的なコンポーネントの方が望ましい。これにより、不要な依存性がなくなるので、置換や保守が容易になる。
- 依存性の逆転: TASのコンポーネントは、低レベルの詳細な部分に対してではなく、抽象的な部分に依存しなければならない。つまり、コンポーネントは特定の自動テストシナリオに依存すべきではない。

通常、gTAAをベースとしたTASは、一連のツールとそのプラグイン、およびコンポーネントによって実装される。gTAAは特定のベンダーに依存しないことが重要である。gTAAは、TASを実現するいかなる具体的な手法、技術、ツールも規定しない。gTAAは、構造化、オブジェクト指向、サービス指向、モデル駆動などの任意のソフトウェアエンジニアリングアプローチを使って実装できる。また、任意のソフトウェア技術やツールを使うこともできる。実際、TASは既製のツールを使って実装されるのが一般的であるが、通常はSUTに固有な機能や対応を追加する必要がある。

TAS関連の他のガイドラインや参照モデルには、指定された SDLC（Software Development LifeCycle）のソフトウェアエンジニアリング標準、プログラミング技術、フォーマット標準などがある。汎用的なソフトウェアエンジニアリングについての学習は本シラバスの範囲ではないが、TAEはソフトウェアエンジニアリングにおけるスキル、経験、専門性を有していることが求められる。

さらに、TAEは業界のコーディング標準やドキュメント標準、ベストプラクティスを認識し、それらをTASの開発に活用する必要がある。このような標準やプラクティスにより、TASの保守性、信頼性、セキュリティを向上させることができる。通常、このような標準はドメインに特化したものである。よく知られた標準には次のようなものがある。

- C または C++ 用の MISRA
- C++ 用の JSF コーディング標準
- MathWorks Matlab/Simulink® 用の AUTOSAR ルール

3.1.1 gTAAの概要

gTAAは、以下による水平レイヤー構造となっている。

- テスト生成
- テスト定義
- テスト実行
- テスト適合

gTAA（図 1: 汎用テスト自動化アーキテクチャを参照）には、以下が含まれる。

- テスト生成レイヤーは、テストケースの手動または自動での設計をサポートし、テストケースを設計する手段を提供する。
- テスト定義レイヤーは、テストスイートおよびテストケースの定義と実装をサポートし、SUTやテストシステムの技術およびツールからテスト定義を分離する。ハイレベルおよびローレベルのテストを定義する手段が含まれており、これらのテストはテストデータ、テストケース、テスト手順、テストライブラリコンポーネント、およびそれらの組み合わせによって処理される。
- テスト実行レイヤーは、テストケースとテスト結果記録作業の実行をサポートする。選択されたテストを自動的に実行するテスト実行ツールと、記録およびレポート用のコンポーネントを提供する。
- テスト適合レイヤーは、SUTのさまざまなコンポーネントやインターフェースを自動テストに適合させるために必要なコードを提供する。API、プロトコル、サービスなどを通じてSUTに接続するための各種アダプターを提供する。
- さらに、テスト自動化に関連するプロジェクトマネジメント、構成管理、テストマネジメント用のインターフェースも搭載する。例えば、テストマネジメントとテスト適合レイヤー間のインターフェースを利用すると、選択されたテスト構成に応じて適切なアダプターを選択し、構成することができる。

通常、gTAAレイヤーとコンポーネント間のインターフェースは固有であるため、ここでは詳述しない。

TASにはこれらのレイヤーが存在することもあれば存在しないこともあるという点を理解することは重要である。次に例を示す。

- テスト実行を自動化する場合、テスト実行レイヤーとテスト適合レイヤーを利用する必要がある。この 2 つを分離する必要はなく、ユニットテストフレームワークなどで同時に実現することも可能である。
- テスト定義を自動化する場合、テスト定義レイヤーが必要になる。
- テスト生成を自動化する場合、テスト生成レイヤーが必要になる。

ほとんどの場合、TASの実装は下から上へ向かって行われる。ただし、手動テスト用のテストケースを自動で生成する場合など、他のアプローチも役立つ可能性がある。一般的には、できるだけ早くTASを使用してTASの付加価値を証明できるように、TASを段階的（スプリントなど）に実装することが推奨される。また、テスト自動化プロジェクトの一環としてコンセプトの証明（PoC）を行うことも推奨される。

すべての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認識、編成、管理する必要があり、専任のプロジェクトマネジメントも必要になる。TAF開発のプロジェクトマネジメント（企業全体または製品ファミリーや製品ラインのテスト自動化サポート）とTASのプロジェクトマネジメント（具体的なプロダクトのテスト自動化）は分離できる。

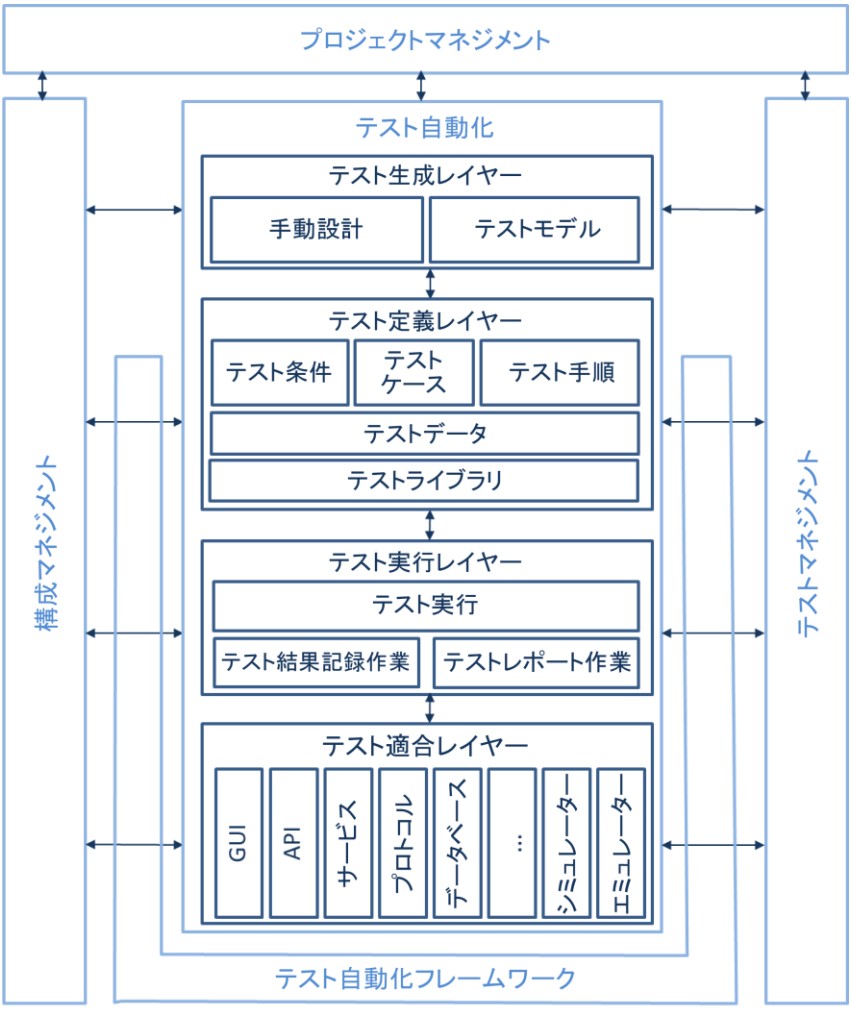

図 1: 汎用テスト自動化アーキテクチャ

3.1.2 テスト生成レイヤー

テスト生成レイヤーは、以下の支援ツールで構成される。

- テストケースの手動設計
- テストデータの作成、キャプチャ、導出
- SUTやその環境を定義するモデルからのテストケースの自動生成（自動モデルベースドテスト）

このレイヤーの一般的なコンポーネントは次のとおりである。
- テストスイート構造を編集またはナビゲーションする
- テストケースとテスト目的またはSUT要件を関連付ける
- テスト設計を文書化する

自動テスト生成には、次のような機能が含まれる。
- SUT、SUTの環境、テストシステムをモデル化する機能
- テスト方針の定義およびテスト生成アルゴリズムの構成やパラメーター化を行う機能
- 生成されたテストからモデル（要素）にさかのぼる機能

3.1.3 テスト定義レイヤー

テスト定義レイヤーは、以下の支援ツールで構成される。
- ハイレベルかつ/またはローレベルのテストケース定義
- ローレベルテストケースのテストデータ定義
- 単一のテストケースや一連のテストケースのテスト手順の定義
- テストケースを実行するためのテストスクリプトの定義
- 必要に応じてテストライブラリへのアクセスの提供（キーワード駆動アプローチの場合など）

このレイヤーの一般的なコンポーネントは次のとおりである。
- テストデータを分類/制限、パラメーター化、具体化する
- テストシーケンスや一連のテストの振る舞い（制御文や式を含む）を明示し、それらをパラメーター化およびグループ化する
- テストデータ、テストケース、テスト手順を文書化する

3.1.4 テスト実行レイヤー

テスト実行レイヤーは、以下の支援ツールで構成される。
- テストケースの自動実行
- テストケースの実行結果の記録
- テスト結果の報告

テスト実行レイヤーには、以下の機能を提供するコンポーネントが含まれる。
- テスト実行のためにSUTの前処理および後処理をする
- テストスイート（テストデータを含む一連のテストケース）の前処理および後処理をする
- テスト準備の構成および設定を行う
- テストデータとテストケースを処理して実行可能スクリプトに変換する
- テストシステムやSUTを測定し、テスト実行やフォールトインジェクション（の一部）を記録する
- テスト実行中のSUTの結果を分析し、後続のテストの進行を調整する
- 自動テストケース実行結果についてSUTのテスト結果の妥当性を確認する（期待結果と実行結果を比較する）
- 自動テスト実行が時間内に終わるよう制御する

3.1.5 テスト適合レイヤー

テスト適合レイヤーは、以下の支援ツールで構成される。

- テストハーネスの制御
- SUTの操作
- SUTの監視
- SUT環境のシミュレートまたはエミュレート

テスト適合レイヤーは、以下の機能を提供する。

- 特定の技術によらないテスト定義と、SUTやテストデバイスに固有の技術要件との間を仲介する
- SUTと連携するために、さまざまな技術固有のアダプターを適用する
- 複数のテストデバイスやテストインターフェースにテスト実行を分散する、またはローカルでテストを実行する

3.1.6 TASの構成管理

通常、TASはさまざまなイテレーションやバージョンで開発され、SUTのイテレーションやバージョンとの互換性が必要となる。TASの構成管理には、以下が必要になる。

- テストモデル
- テストデータ、テストケース、ライブラリを含むテストの定義や仕様
- テストスクリプト
- テスト実行エンジンと補助ツール、補助コンポーネント
- SUT用のテストアダプター
- SUT環境のシミュレーターやエミュレーター
- テスト結果やテストレポート

これらの項目がテストウェアを構成する。これらは、SUTのバージョンと一致した正しいバージョンでなければならない。古いバージョンのSUTで現場の問題を再現しなければならない場合など、状況によっては以前のTASのバージョンに戻さなければならない場合がある。これは適切な構成管理を行えば可能になる。

3.1.7 TASのプロジェクトマネジメント

テスト自動化プロジェクトはすべてソフトウェアプロジェクトであるため、他のソフトウェアプロジェクトと同じプロジェクトマネジメントが必要になる。TASの開発においては、TAEは確立された SDLC手法のすべてのフェーズのタスクを行う必要がある。さらにTAEは、ステータス情報（メトリック）を容易に抽出できるか、またはTASのプロジェクトマネジメントに自動的に報告されるようにTASの開発環境を設計しなければならない、ということを理解する必要がある。

3.1.8 TASのテストマネジメントサポート

TASは、SUTのテストマネジメントをサポートしなければならない。テスト結果記録やテスト結果を含むテストレポートは、抽出が容易であるか、SUTのテストマネジメント（人またはシステム）に自動的に提供される必要がある。

3.2 TAAの設計

3.2.1 TAAの設計の概要

TAAの設計には多くの重要な作業が必要になる。これらの作業は、テスト自動化プロジェクトや組織のニーズに応じて順位付けすることができる。作業の詳細については、以降の節で詳述する。TAAの複雑度により、必要な作業は増減する場合がある。

適切なTAAを定義するために必要な要件を把握する
テスト自動化アプローチの要件として、以下を考慮する必要がある。

- テストマネジメント、テスト設計、テスト生成、テスト実行などのテストプロセスのうち、どの作業やフェーズを自動化すべきか。なお、テスト自動化はテスト設計とテスト実装の間にテスト生成を挿入することによって基本的なテストプロセスを洗練する点に留意すること。
- コンポーネントレベル、統合レベル、システムレベルなどのうち、どのテストレベルをサポートすべきか。
- 機能テスト、適合テスト、相互運用性テストなどのうち、どのタイプのテストをサポートすべきか。
- テスト実行者、テストアナリスト、テストアーキテクト、テストマネージャーなどのうち、テストに関するどの役割をサポートすべきか。
- 実装されるTASの寿命や存続期間などを定義するために、どのソフトウェアプロダクト、ソフトウェア製品ライン、ソフトウェア製品ファミリーをサポートすべきか。
- SUTに使われる技術との互換性を考慮してTASを定義するために、どの技術をサポートすべきか。

異なる設計やアーキテクチャを用いたアプローチと比較、対比を行う
TAAの選択したレイヤーを設計する際、TAEはさまざまなアプローチの利点と欠点を分析する必要がある。これには以下の内容が含まれるが、それに限定されるものではない。

テスト生成レイヤーの考慮事項:

- テスト生成を手動にするか自動にするかの選択
- 要件ベースド、データベースド、シナリオベースド、振る舞いベースドなどのテスト生成の選択
- テスト生成戦略の選択(データベースのアプローチのためのクラシフィケーションツリーなどのモデルカバレッジ、シナリオベースのアプローチのためのユースケース/例外ケースカバレッジ、振る舞いベースのアプローチのための遷移/状態/パスカバレッジなど)
- テストの選定戦略の選択。実際には、すべての組み合わせのテストを生成するのは、テストケースの爆発的増加につながる可能性があるため、不可能である。そのため、現実的なカバレッジ基準、重み付け、リスクアセスメントなどを参考にしてテスト生成およびそれに続くテストの選定を行うべきである。

テスト定義レイヤーの考慮事項:

- データ駆動、キーワード駆動、パターンベースド、モデル駆動のどのテスト定義を利用するかの選択
- テスト定義の表記法の選択(スプレッドシート、ドメインに特化したテスト用の言語、Testing and Test Control Notation(TTCN-3)、UML Testing Profile(UTP)などを使用した表、状態ベースの表記法、確率的表記法、データフロー表記法、ビジネスプロセス表記法、シナリオベースの表記法など)
- 高品質なテストを定義するためのスタイルガイドやガイドラインの選択
- テストケースリポジトリ(スプレッドシート、データベース、ファイルなど)の選択

テスト実行レイヤーの考慮事項:

- テスト実行ツールの選択
- テスト手順を実現するために(仮想マシンの使用による)インタプリタ型のアプローチとコンパイル型のアプローチのどちらを採用するかの選択。通常、このアプローチは選択されたテスト実行ツールに依存する。

- テスト手順を実装する技術（C などの命令型。Haskell や Erlang などの関数型。C++、C#、Java などのオブジェクト指向型。Python や Ruby などのスクリプト。またはツール固有の技術）の選択。通常、この技術は選択したテスト実行ツールに依存する。
- テスト実行を容易にするヘルパーライブラリ（テストデバイスのライブラリ、エンコーディング/デコーディングライブラリなど）の選択。

テスト適合レイヤーの考慮事項:

- SUTとのテストインターフェースの選択
- テストインターフェースを操作および観測するツールの選択
- テスト実行の際にSUTを監視するツールの選択
- テスト実行をトレースするツール（テスト実行のタイミングを含む）の選択

抽象化によるメリットが得られる領域を特定する

TAAを抽象化することにより、異なるテスト環境や対象技術で同じテストスイートを使えるようになり、技術への依存度を軽減でき、テスト成果物の移植性が向上する。さらに、ベンダーへの非依存性が保証され、TASのロックイン効果を回避できる。抽象化により、保守性や新しいSUTに使われる技術およびその変化への環境適応性も向上する。そのうえ、抽象化は技術者以外がTAA（およびそれをTASとして具体化したもの）を使いやすくすることにもつながる。テストスイートを文書化（視覚的手段を含む）して高レベルで説明でき、それによって可読性や理解性が向上するためである。

TAEは、ソフトウェア開発、品質保証、テストにおいて、TASのどの領域でどのレベルの抽象化を使用するかをステークホルダーと議論する必要がある。例えば、テスト適合レイヤーやテスト実行レイヤーのインターフェースのうち、どれを外部化し、どれを形式的に定義し、TAAの存続期間にわたってどの安定性を確保する必要があるかなどが挙げられる。さらに、抽象化したテスト定義が使われるのか、それともTAAはテスト実行レイヤーでテストスクリプトのみを使うのかについても議論する必要がある。同様に、テストモデルとモデルベースドテストアプローチを使用してテスト生成を抽象化するかどうかについても、共通認識とする必要がある。TAEは、機能性、保守性、拡張性のすべてにおいて、TAAの高度な実装と単純な実装との間にトレードオフがあることを認識しておく必要がある。TAAでどの抽象化を使用するかを決定するにあたっては、これらのトレードオフを考慮する必要がある。

TAAで多くの抽象化を行うほど、進化の余地や新しいアプローチや技術への移行に関する柔軟性が向上する。その代償として、初期投資が増加する（テスト自動化のアーキテクチャやツールの複雑度が増す、スキルセット要件が高くなる、学習コストが大きくなるなど）。それによって最初の損益分岐点に到達するまでの時間が長くなるが、長期的に見ればそれに見合うだけの成果をあげられる可能性がある。さらに、TASのパフォーマンスが下がる場合もある。

ROI（投資収益率）について詳細に検討することは TAM の責任範囲であるが、TAEはさまざまなテスト自動化のアーキテクチャやアプローチについて、タイミングやコスト、工数、利益に関する技術的評価や比較を行い、ROI分析に必要な情報を提供する必要がある。

SUTに使われる技術やTASとの関係について理解する

あらゆる自動テスト実行で中核となるのは、SUTのテストインターフェースにアクセスすることである。このアクセスは、以下のレベルで実現される。

- ソフトウェアレベル: SUTとテストソフトウェアが相互にリンクされる場合など
- APIレベル:（リモートの）アプリケーションプログラミングインターフェースが提供する関数/操作/メソッドをTASが呼び出す場合など
- プロトコルレベル: HTTP、TCPなどでTASがSUTと通信する場合など
- サービスレベル: Web サービスや RESTful サービスなどでTASがSUTのサービスと通信する場合など

TASとSUTが API、プロトコル、サービスで分離されている場合、TAEはTASとSUTの間の連携に使用するTAAの相互作用のパラダイムも決定する必要がある。このパラダイムには、以下のものが含まれる。

- 交換されるイベントやイベントバスを通した相互作用を実現するイベント駆動パラダイム
- サービス要求者がサービス提供者のサービスを呼び出すという相互作用を実現するクライアント/サーバーパラダイム
- ピア間での任意の方向のサービス呼び出しによる相互作用を実現するピアツーピアパラダイム

多くの場合、パラダイムの選択はSUTアーキテクチャに依存し、SUTアーキテクチャに影響する可能性がある。SUTとTAAの間の相互作用は、2 つのシステム間の将来を見据えたアーキテクチャを選択するために、注意深く分析して設計する必要がある。

SUT環境を理解する

SUTは、スタンドアロンソフトウェア、他のソフトウェアとの関係でのみ動作するソフトウェア（システムオブシステムズなど）、ハードウェア（組込みシステムなど）、環境コンポーネント（サイバーフィジカルシステムなど）である可能性がある。TASは、自動テストのセットアップの一環として、SUT環境のシミュレートまたはエミュレートを行う。

テスト環境の例とサンプル使用の例を以下に示す。

- SUTとTASの両方の機能を備えたコンピューター。ソフトウェアアプリケーションのテストに有用。
- SUTやTASのそれぞれとネットワークで接続された個々のコンピューター。サーバーソフトウェアのテストに有用。
- SUTの技術的インターフェースを操作および観測する追加のテストデバイス。セットトップボックスなどのソフトウェアのテストに有用。
- SUTの操作環境をエミュレートするネットワーク接続テストデバイス。ネットワークルーターのソフトウェアのテストに有用。
- SUTの物理環境をシミュレートするシミュレーター。組込み制御ユニットのソフトウェアのテストに有用。

指定されたテストウェアアーキテクチャ実装の時間と複雑度

TASプロジェクトにかかる作業の見積りは TAM の責任範囲であるが、TAEはTAA設計の時間と複雑度を適切に見積ることを通して、TAM を支援する必要がある。見積りの方法や例には、次のようなものがある。

- ファンクションポイント、3 点見積り、ワイドバンドデルファイ、専門家による見積りなど、類推による見積り
- マネジメントソフトウェアやプロジェクトテンプレートなどで見られるワークブレークダウンストラクチャの使用による見積り
- Constructive Cost Model（COCOMO）などの係数見積り
- ファンクションポイント法、ストーリーポイント法、ユースケース分析などによるサイズベースの見積り
- プランニングポーカーなどのグループ見積り

指定されたテストウェアアーキテクチャ実装の使いやすさ

TASの機能性、SUTとの互換性、長期安定性や進化性、作業要件、ROI に関する検討事項に加え、TAEにはTASの使用性の問題に対処する明確な責任がある。これには以下の内容が含まれるが、それに限定されるものではない。

- テスト担当者指向の設計
- TASの使いやすさ
- ソフトウェア開発、品質保証、プロジェクトマネジメントにおける他の役割へのTASの支援
- TASの効果的な編成、ナビゲーション、検索
- TASの有用な文書化、マニュアル、ヘルプテキスト
- TASによるTASについての実用的なレポート

- TASのフィードバックや経験的洞察に対応するためのイテレーティブ設計

3.2.2 テストケース自動化へのアプローチ

テストケースは、SUTに対して実行される一連のアクションに変換が必要である。この一連のアクションは、テスト手順での文書化および/またはテストスクリプトでの実装が可能である。自動テストケースは、アクションの他にSUTを操作するためのテストデータの定義や、SUTが期待結果を実現したことを検証する手順を含む必要がある。一連のアクションを作成する際に使用できるアプローチは多くある。

1. TAEが自動テストスクリプトに直接テストケースを実装する。この選択肢は、抽象化が不足しており、保守の負荷が増加するため、最も推奨されない。
2. TAEがテスト手順を設計し、それを自動テストスクリプトに変換する。この選択肢は抽象化されているが、テストスクリプトを生成する自動化が不足している。
3. TAEがツールを使用してテスト手順を自動テストスクリプトに変換する。この選択肢は、抽象化と自動テストスクリプト生成の両方を組み合わせている。
4. TAEが自動テスト手順を生成するツールを使用する、モデルから直接テストスクリプトを変換する、またはその両方を行う。この選択肢は自動化の成熟度が最も高い。

各選択肢はプロジェクトの状況に大きく依存する点に注意すること。一般的に容易な選択肢ほど実装も簡単なので、それを適用するところからテスト自動化を始めることが効率的な場合もある。これにより短時間で付加価値を提供できるが、保守性の低いソリューションが生まれることになる。

テストケースを自動化するアプローチで確立されているものとして、以下が挙げられる。

- キャプチャ/プレイバックアプローチ。これは選択肢 1 で使用できる。
- 構造化スクリプティングアプローチ、データ駆動アプローチ、キーワード駆動アプローチ。これらは選択肢 2 または 3 で使用できる。
- モデルベースドテスト（プロセス駆動アプローチを含む）。これは選択肢 4 で使用できる。

以降では、これらのアプローチの主な概念および利点と欠点について説明する。

キャプチャ/プレイバックアプローチ

主な概念

キャプチャ/プレイバックアプローチでは、ツールを使用してSUTとのやりとりをキャプチャしつつ、テスト手順で定義された一連の手順を実行する。入力がキャプチャされ、後ほどチェックするために出力も記録される場合がある。操作を再生する際には、次のようなさまざまな手動および自動の出力チェックを行うことが考えられる。

- 手動: テスト担当者がSUTの出力を監視して不正がないかを検出する必要がある
- 完全: SUTはキャプチャの際に記録されたすべてのシステム出力を再現する必要がある
- 厳密: SUTはキャプチャの際に記録されたすべてのシステム出力を記録の詳細のレベルまで再現する必要がある
- チェックポイント: ある時点で選択されたシステム出力の指定された値のみがチェックされる

利点

キャプチャ/プレイバックアプローチは、GUIレベルおよび/または APIレベルでSUTに使用する。導入初期の段階では、セットアップや使用が容易である。

欠点

キャプチャしたSUT操作はキャプチャ時のSUTのバージョンに強く依存するので、キャプチャ/プレイバックスクリプトの保守や進化は難しい。例えば、GUIレベルの記録は、たとえ要素の位置が変更されただけだとしても、GUIレイアウトが変更されるとテストスクリプトに影響する可能性がある。そのため、キャプチャ/プレイバックアプローチは変更に弱いという点は変わらない。

テストケース（スクリプト）の実装を開始できるのは、SUTが利用できるようになってからである。

線形スクリプティング

主な概念

すべてのスクリプティング技術と同じように、線形スクリプティングもいくつかの手動テスト手順から始まる。ただし、これらは文書化されてない場合もある点に注意すること。どのテストをどのように実行するかは、1 人または複数人のテストアナリストの「暗黙知」である場合がある。

テストツールが一連の手順を記録する間、各テストは手動で実行され、場合によってはSUTの画面出力を視覚的に記録する。この結果、一般的に、1 つのテスト手順につき 1 つの（通常は大きな）スクリプトになる。記録されたスクリプトを編集して可読性を向上させたり（重要な場所に何が起きているかのコメントを追加するなど）、ツールのスクリプト言語を使用してさらにチェックを追加したりすることもできる。

そして、このスクリプトはツールで再生することができ、記録時にテスト担当者が行ったのと同じ手順を繰り返す。この方法は GUIテストの自動化に使用できるが、ソフトウェアの多くのリリースのために大量のテストを自動化する必要があるような場合、適切な技法ではない。これは、SUTの変更に伴う保守コストが高くなる（SUTで変更が行われるたびに、記録したスクリプトで多くの部分を変更しなければならない可能性がある）ことが多いためである。

利点

線形スクリプトの主な利点は、自動化を始めるにあたって必要になる準備作業がまったくあるいはほとんどない点である。ツールの使用方法を習得すれば、手動テストを記録して再生するだけで済む（ただし、記録の部分では、テストツールの追加操作を行い、実行結果と期待結果を比較してソフトウェアが正常動作することの検証が求められる場合がある）。プログラミングのスキルは必須ではないが、役立つことが多い。

欠点

線形スクリプトには多くの欠点がある。テスト手順を自動化するために必要な作業の量は、それを実行するために必要なサイズ（手順の数）に依存することが多い。すなわち、1000 個目のテスト手順を自動化する際にも、100 個目のテスト手順を自動化するのと同じ作業量が必要になる。つまり、新しい自動テストを構築するコストを削減できる余地は多くない。

さらに、入力値が異なる同じようなテストのスクリプトがあった場合、2 つのスクリプトに同じ一連の命令が含まれることになる。この場合、命令に含まれる情報（命令の引数またはパラメーターと呼ばれる）のみが異なる。複数のテスト（すなわち複数のスクリプト）があった場合、すべてに同じ一連の命令が含まれる。スクリプトに影響するソフトウェアの変更があった場合、それらすべてを保守する必要がある。

スクリプトは自然言語ではなくプログラミング言語で書かれているため、プログラマー以外が理解するのは難しい。テストツールの中には独自の言語（ツールに独特な言語）を使用するものもあり、言語の学習や習得には時間がかかる。

記録されたスクリプトは、コメントがもしあれば、その中には通常の文章のみを含む。とりわけ長いスクリプトでは、テストの各手順で何が起こっているのかを説明するコメントが付いていることが望ましい。これにより、保守が容易になる。テストに多くの手順が含まれる場合、スクリプトはすぐに肥大化する（スクリプトに多くの命令が含まれる）。

スクリプトはモジュール形式ではないので、保守が難しい。線形スクリプティングは一般的なソフトウェアの再利用性やモジュール型パラダイムに準じておらず、使用するツールと密結合である。

構造化スクリプティング

主な概念

構造化スクリプティング技法が線形スクリプティング技法と異なる主な点は、スクリプトライブラリを導入していることである。これには、複数のテストで共通して用いられる、一連の命令を実行する再利用可能スクリプトが含まれる。このようなスクリプトの例として、SUTインターフェースの操作などを行うものが挙げられる。

利点

このアプローチの利点には、保守の際に必要な変更が大幅に削減されることや、新しいテストを自動化する際のコストを削減できることが挙げられる（最初からスクリプトを作成するのではなく、既存のスクリプトを使用できるため）。

構造化スクリプティングの利点の大部分は、スクリプトを再利用できることに起因する。線形スクリプティングアプローチで必要とされるほどの量のスクリプトを作成せずに、多くのテストを自動化することができる。これは、スクリプト作成と保守のコストに直接的に影響する。作成するスクリプトの一部は最初のテストを再利用して実装することができるので、2 つ目以降のテストの自動化には 1 つ目ほど手間はかからない。

欠点

共有スクリプトを作成する最初の作業は欠点に思われがちだが、この初期投資を適切に行うことができれば、大きな利益となって返ってくる。すべてのスクリプトを作成するには、単純な記録だけでは十分ではなく、プログラミングのスキルが必要になる。スクリプトライブラリは、スクリプトを文書化し、テクニカルテストアナリストが必要なスクリプトを容易に見つけられるように適切に管理される必要がある（そのため、適切な命名規約が役に立つ）。

データ駆動テスト

主な概念

データ駆動スクリプティング技法は、構造化スクリプティング技法がベースとなっている。最も重要な違いは、テスト入力を扱う方法にある。入力はスクリプトから切り離され、1 つまたは複数のファイルに格納される（通常はデータファイルと呼ばれる）。

つまり、主なテストスクリプトを再利用して（1 つのテストではなく）多くのテストを実装できる。通常は、「再利用可能な」主なテストスクリプトを「制御」スクリプトと呼ぶ。制御スクリプトにはテストを実行するために必要な一連の命令が含まれる。ただし、入力データはデータファイルから読み取られる。1 つの制御スクリプトを多くのテストで使用することも可能だが、幅広いテストを自動化するには不十分であることが多い。そのため、多数の制御スクリプトが必要になるが、自動化するテストの数全体に比べればわずかな数にすぎない。

利点

このスクリプティング技法を使用すれば、新しい自動テストを追加するコストを大幅に削減できる。この技法によって、有用なテストを数多く自動化でき、特定の領域を深くテストすることができるので、テストカバレッジの増加につながる可能性がある。

テストがデータファイルで「記述」されるので、テストアナリストは 1 つまたは複数のデータファイルを投入するだけで「自動」テストを指定できる。そのため、テストアナリストが自動テストを指定する自由度が増し、（希少なリソースである可能性がある）テクニカルテストアナリストへの依存度を低くできる。

短所

データファイルを管理し、TASが読み取れるようにしなければならない点は短所だが、これに対しては適切なアプローチをとることができる。

また、重要な否定テストケースが見逃される可能性がある。否定テストはテスト手順とテストデータの組み合わせである。主にテストデータを対象としたアプローチでは、「否定テスト手順」が見逃される可能性がある。

キーワード駆動テスト

主な概念

キーワード駆動スクリプティング技法は、データ駆動スクリプティング技法がベースとなっている。主な違いは、以下の 2 点である。(1)データファイルは「テスト定義」ファイル（またはアクションワードファイルなど）と呼ばれる。(2)制御スクリプトは 1 つしかない。

テスト定義ファイルには、テストアナリストが、同じ内容のデータファイルより簡単に理解できる方法で表現されたテストが含まれる。通常はデータファイルと同じようなデータが含まれるが、キーワードファイルには高レベルな命令（キーワードまたは「アクションワード」）も含まれる。

キーワードは、テストアナリスト、記述されるテスト、テストされるアプリケーションにとって意味があるものを選択する。キーワードは、例外はあるがほとんどの場合、ビジネスとシステムの高位レベルの相互作用（たとえば、注文の実行）を表すために使用する。それぞれのキーワードは、テスト対象システムとのさまざまな相互作用を表す。キーワード（関連するテストデータを含む）の配列は、テストケースを指定する

ために使用する。検証ステップに特殊なキーワードを使うこともできる。また、キーワードに操作ステップと検証ステップの両方を含めることもできる。
テストアナリストの責任範囲には、キーワードファイルの作成と保守が含まれる。つまり、補助スクリプトが一度実装されれば、テストアナリストは（データ駆動スクリプティングと同様に）キーワードファイルに記述するだけで「自動」テストを追加できる。

利点

このスクリプティング技法では、制御スクリプトとキーワード用の補助スクリプトを記述できれば、新しい自動テストを追加するコストを大きく削減できる。

テストがキーワードファイルで「記述」されるので、テストアナリストはキーワードと関連するデータを使用してテストを記述するだけで「自動」テストを作成できる。そのため、テストアナリストが自動テストを作成する自由度が増し、（希少なリソースである可能性がある）テクニカルテストアナリストへの依存度を低くできる。この点に関してデータ駆動アプローチと比較した場合のキーワード駆動アプローチの利点は、キーワードの使用にある。それぞれのキーワードは、意味のある結果を生成する詳細な一連のアクションを表すものにする。例えば、「アカウントの作成」、「注文の実行」、「注文ステータスのチェック」はすべて、オンラインショッピングアプリケーションでのアクションとして考えられる。これらのアクションには、複数の詳細な手順が含まれている。あるテストアナリストが別のテストアナリストにあるシステムテストについて説明する場合、詳細な手順ではなく、このような高レベルのアクションについて述べることが多い。キーワード駆動アプローチのねらいは、このような高レベルのアクションを実現し、詳細な手順について言及せずに高レベルのアクションだけでテストを定義できるようにする点にある。

このようなテストケースでは複雑さがキーワード（構造化スクリプティングアプローチの場合はライブラリ）で隠蔽されるので、保守性が高く、読み取り/書き込みも容易になる。キーワードを使用することで、SUTのインターフェースの複雑度を抽象化することができる。

短所

キーワードの実装はテスト自動化エンジニアにとって大きな負担であることには変わりない。このスクリプティング技法をサポートしないツールを使用する場合は、特にそれが当てはまる。小規模なシステムには実装のオーバーヘッドが大きすぎ、コストが利点を上回る可能性がある。

正しいキーワードが実装されるように十分注意を払う必要がある。優れたキーワードは数多くのテストで使用されるが、適切でないキーワードは 1 度または数回しか使われない可能性が高い。

プロセス駆動スクリプティング

主な概念

プロセス駆動アプローチは、キーワード駆動スクリプティング技法がベースとなっている。違いは、シナリオ（SUTのユースケース、すなわちバリエーションを表す）がスクリプトで構成される点にある。このスクリプトは、パラメーターとしてテストデータを受け取るか、ハイレベルのテスト定義と組み合わされている。

このテスト定義はアクション間の論理関係として扱いやすい。例えば、フィーチャーテストで「注文の実行」の後に「注文ステータスのチェック」を行うのか、またはロバストネステストで「注文の実行」を行わずに「注文ステータスのチェック」を行うのか、決めることができる。

利点

プロセス的でシナリオベースのテストケース定義を使用することで、ワークフローからテスト手順を定義することができる。プロセス駆動アプローチのねらいは、詳細なテスト手順を表すテストライブラリを使用して高レベルワークフローを実現する点にある（キーワード駆動アプローチも参照）。

短所

テクニカルテストアナリストがSUTのプロセスを理解するのは容易ではない。プロセス指向スクリプトの実装も同様である。ツールがビジネスプロセスのロジックをサポートしていない場合、特にそれが当てはまる。

正しいキーワードを使って正しいプロセスが実装されるように十分注意を払う必要がある。優れたプロセスは他のプロセスから参照され、多くの関連テストで活用されるが、適切でないプロセスは妥当性やエラー検知能力などの点で労力に見合わない。

モデルベースドテスト

主な概念

モデルベースドテストとは、キャプチャ/プレイバック、線形スクリプティング、構造化スクリプティング、データ駆動スクリプティング、プロセス駆動スクリプティングを使用してテストケースを自動生成することを指し（ISTQB-MBT Foundation Level Model-Based Tester Extension Syllabusも参照）、テストケースを自動実行することとは異なる考え方である。モデルベースドテストには、TAAのスクリプティング技法を抽象化した（準）形式モデルを使用する。前述の任意のスクリプティングフレームワーク用のテストを導出するために、異なるテスト生成手法を使用することもできる。

利点

モデルベースドテストでは、抽象化によりテストの本質（テスト対象のビジネスロジック、データ、シナリオ、構成などに関すること）に集中できる。テスト生成に使用するモデルが将来を見据えたテストウェアを表現したものになるように、異なる対象システムや対象技術のテストを生成できる。これは技術の進化に合わせて再利用したり保守したりすることもできる。

要件が変更された場合でも、テストモデルを対応させるだけで済む。すべてのテストケースは自動的に生成される。テストケース設計技法は、テストケースジェネレータに組み込まれる。

短所

モデルベースドテストアプローチを効果的に実行するには、モデルに関する専門知識が必要になる。SUTのインターフェース、データおよび/または振る舞いを抽象化してモデリングを行うタスクは困難になる場合もある。さらに、モデリングやモデルベースドテストのツールはまだ主流ではなく、成長途中にある。モデルベースドテストアプローチには、テストプロセスの調整が必要になる。例えば、テスト設計者の役割を確立する必要がある。また、テスト生成に使用するモデルがSUTの品質保証の主な成果物となるため、品質保証や保守を行う必要がある。

3.2.3 SUTの技術的考慮事項

さらに、TAAの設計にあたっては、SUTの技術的側面を考慮すべきである。以下でその一部を説明する。完全なものではないが、重要な側面の一例となるはずである。

SUTのインターフェース
SUTには内部インターフェース（システムの内部）と外部インターフェース（システム環境やユーザーに対するもの、またはコンポーネントによって外部に公開されているもの）がある。SUTのインターフェースは、テスト手順による影響を受ける可能性があるため、TAAはこれらすべてのインターフェースの制御および/または観測を行える必要がある（すなわち、インターフェースはテスト可能でなければならない）。さらに、SUTとTASとの相互作用をさまざまな詳細レベルで記録しなければならない場合もあり、これには通常、タイムスタンプが含まれる。

プロジェクト開始時のアーキテクチャ定義の際に（アジャイル環境の場合は継続的に）、個々のテストにスコープを当てることが必要である。これは、SUTがテスト可能であるために必要なテストインターフェースやテスト設備の可用性を検証するためである（試験性を考慮した設計）。

SUTのデータ
SUTは構成データを使って具現化、構成、管理などを制御する。さらに、SUTはユーザーデータの処理も行う。SUTは他のシステムからの外部データを使用してタスクを完了させることもある。SUTのテスト手順によっては、これらすべてのデータが定義可能かつ構成可能であり、さらにTAAから具現化できる必要がある。SUTデータを扱う具体的な方法は、TAAの設計で決定する。アプローチによっては、データをパラメーター、テストデータシート、テストデータベース、実データなどとして扱うことができる。

SUTの構成
SUTは、異なるオペレーティングシステム、異なる対象デバイス、異なる言語設定など、さまざまな構成で導入される可能性がある。テスト手順によっては、TAAが複数のSUT構成に対処しなければならない場合がある。テスト手順には、指定されたSUT構成とともに、異なるTAAのテスト設定（ラボ内）や仮想テスト設定（クラウド内）が必要になる場合がある。さらに、選択したSUTの側面について、選択したSUTコンポーネントのシミュレーターおよび/またはエミュレーターの追加が求められる場合もある。

SUTの標準と法的設定
互換性のある方法でTAAを設計できるように、TAAの設計は、SUTの技術的側面に加えて法律や標準に関する要件を考慮しなければならない場合がある。例えば、テストデータのプライバシー要件や、TAAの記録およびレポート機能に影響する機密性要件などがある。

SUTの開発に使用されるツールおよびツール環境
SUTの開発に合わせて、SUTの要件エンジニアリング、設計およびモデリング、コーディング、統合、導入のためにさまざまなツールが使われる場合がある。独自のツールを持つTAAは、ツールの互換性やトレーサビリティ、、成果物の再利用を実現するために、SUTツール群を考慮する。

ソフトウェアプロダクトのテストインターフェース
プロダクトリリースの前にすべてのテストインターフェースの削除を行わないことを強く推奨する。ほとんどの場合、最終的なプロダクトに影響を与えることなく、これらのインターフェースをSUTに残しておくことができる。残しておくと、サービスエンジニアやサポートエンジニアが問題の診断や保守リリースのテストにこれらのインターフェースを使用することができる。インターフェースがセキュリティリスクとならないことを検証することが重要である。必要があれば、通常、開発担当者は開発部門以外がテストインターフェースを使うことができないように無効化できる。

3.2.4 開発/QA プロセスの考慮事項

TAAを設計する際には、SUTの開発および品質保証のプロセスについて考慮すべきである。以下でその一部を説明する。完全なものではないが、重要な側面の一例となるはずである。

テスト実行制御要件
TAAで必要とされる自動化のレベルによっては、TAAが対話的テスト実行、バッチモードテスト実行、完全自動テスト実行をサポートしなければならない場合がある。

レポート要件
レポートの種類や構造などのレポート要件によっては、形式やレイアウトが異なる固定のテストレポート、パラメーター化されたテストレポート、定義済みのテストレポートをTAAがサポートできなければならない場合がある。

役割とアクセス権
セキュリティ要件によっては、TAAが役割やアクセス権のシステムを提供しなければならない場合がある。

確立されたツール群
確立されたツール群を構成するツールでは、SUTプロジェクトマネジメント、テストマネジメント、コードおよびテストのリポジトリ、欠陥追跡、インシデントマネジメント、リスク分析などがすべてサポートされている場合がある。TAAは、ツール群の中の他のツールとシームレスに統合されているツールまたはツールセットによってもサポートされている。また、テストスクリプトは、SUTのコードと同じプロセスに従って改訂されるように、どちらも同じ方法で保存およびバージョン管理すべきである。

3.3 TASの開発

3.3.1 TASの開発の概要

TASの開発は、他のソフトウェア開発プロジェクトに類似する。開発担当者とテスト担当者によるピアレビューなどを含め、同じ手順やプロセスに従うことができる。TASに固有な点は、SUTとの互換性と同期である。この点は、TAAの設計（3.2節を参照）およびTASの開発で検討する必要がある。さらに、TASがテストインターフェースを利用できるようにしなければならないなど、SUTはテスト戦略の影響を受ける。

本節では、ソフトウェア開発ライフサイクル（SDLC）を使用して、TAS開発プロセスおよびプロセスに関連するSUTとの互換性や同期の特徴について説明する。これらの特徴は、SUTおよび/またはTASの開発にあたって選択および実施した他の開発プロセスにとっても同じように重要であり、適切に適合させる必要がある。

TASの基本的な SDLC を図 2 に示す。

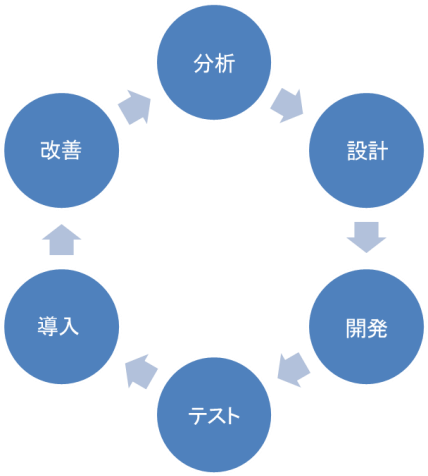

図 2: TASのための基本的な SDLC

TASの一連の要件を分析および収集する必要がある（図2参照）。TAAで定義されているように（3.2節を参照）、この要件はTASの設計の指針となる。ソフトウェアエンジニアリングのアプローチにより、この設計がソフトウェアになる。なお、TASは専用のテストデバイスハードウェアも使用する場合があるが、この点は本シラバスの対象には含まれない。他のソフトウェアと同じように、TASはテストする必要がある。通常、これはTASの基本的な機能をテストすることによって行われる。その後、TASとSUTは相互に影響する。TASを導入し使用した後は、さらにテスト機能の拡張をしたり、テストを変更したり、SUTの変更に合わせてTASを更新したりするために、TASを改修する必要が生じることが多い。TASを改善するには、SDLC に基づく新たな TA 開発が必要になる。

なお、SDLC にはTASのバックアップ、アーカイブ、廃棄は示されていない点に注意すること。TASの開発と同様、これらの手順は組織で確立された手法に従うべきである。

3.3.2 TASとSUTの親和性

プロセスの親和性
SUTのテストはSUTの開発と同期すべきである。また、テスト自動化を行う場合は、TASの開発とも同期させる必要がある。そのため、SUTの開発プロセス、TASの開発プロセス、テストプロセスの整合性をとることが望ましい。プロセス構造、プロセスマネジメント、支援ツールの面でSUTとTASの開発に親和性があると、大きな利点となる。

チームの親和性
チームの親和性は、TASとSUTの開発間の親和性のもう 1 つの側面である。TASとSUTの開発のアプローチやマネジメントに親和性のマインドセットを取り入れると、お互いの要件や設計および開発の成果物のレビュー、問題についての議論、互換性のあるソリューションの発見という点で両方のチームにメリットがある。チームの親和性があると、お互いのコミュニケーションや対話の際にも役立つ。

技術の親和性
さらに、TASとSUTとの技術の親和性も考慮すべきである。最初からTASとSUTとの間のシームレスな相互作用を設計および実装しておくことにはメリットがある。TASにもSUTにも技術的ソリューションが利用できないなどの理由でそれが不可能だとしても、アダプターやラッパーなど何らかの仲介するものを使用することで、シームレスな相互作用を実現できる可能性がある。

ツールの親和性

TASとSUTのマネジメント、開発、品質保証の間のツールの親和性を考慮する必要がある。例えば、要件マネジメントおよび/または課題マネジメントに同じツールを使用している場合、TASとSUTの開発に関する情報交換や調整が容易になる。

3.3.3 TASとSUTの同期

要件の同期
要件を明確にすることが、SUTとTASの両方の要件の開発につながる。TASの要件は、次の 2 つの主な要件グループに分けることができる。(1)ソフトウェアベースのシステムとしてTASの開発に対処する要件。これには、テスト設計、テスト仕様、テスト結果分析などのためのTASのフィーチャー要件が含まれる。(2)TASを用いてSUTのテストに対処する要件。これらはテスト要件と呼ばれ、SUTの要件に対応する。また、TASがテストするすべてのSUTのフィーチャーや特性を反映する。SUTやTASの要件が更新されるときは、必ず両者の一貫性を検証し、TASによってテストされるすべてのSUT要件がテスト要件で定義されていることを確認することが重要である。

開発フェーズの同期
SUTのテストが必要になるときにTASの準備が完了しているよう、開発フェーズを調整する必要がある。最も効率がよいのは、SUTとTASの要件、設計、仕様、実装が同期していることである。

欠陥追跡の同期
欠陥は、SUTに関連するもの、TASに関連するもの、要件/設計/仕様に関連するものである可能性がある。2 つのプロジェクトには関連性があるため、片方で欠陥を修正するともう片方に影響が及ぶ可能性がある。欠陥追跡と確認テストは、TASとSUTの両方に対処しなければならない。

SUTとTASの改善の同期
SUTとTASは、新規フィーチャーの組込み、フィーチャーの無効化、欠陥の修正、環境の変更(SUTとTASは、片方がもう片方のコンポーネントとなっているので、それぞれの変更も含まれる)に対応するように、どちらも改善できる。SUTとTASのどちらかに変更が適用されると、もう一方が影響を受ける場合があるので、変更のマネジメントはSUTとTASの両方に対処すべきである。

SUTとTASの開発プロセス間を同期させる 2 つのアプローチを図 3 および図 4 に示す。

図 3 は、SUTとTASの 2 つの SDLC プロセスが主に次の 2 つのフェーズで同期されるアプローチを示している。(1)TASの分析はSUTの設計に基づく。SUTの設計はSUTの分析に基づく。(2)SUTのテストは導入済みのTASを使用する。

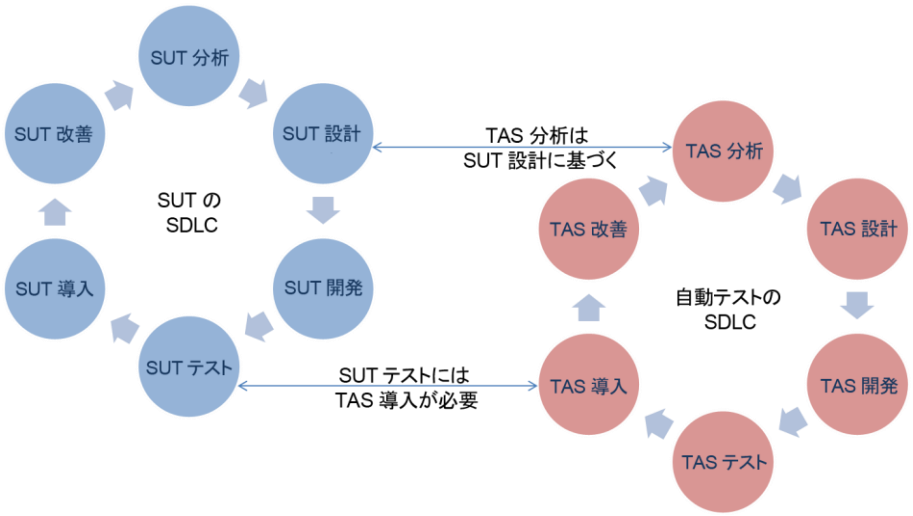

図 3: TASとSUTの開発プロセスの同期例 1

図 4 は、手動および自動の両方のテストを使用したハイブリッドアプローチを示している。テストを自動化する前に手動テストを用いる場合や、手動および自動のテストを併せて用いる場合、TASの分析はSUTの設計と手動テストの両方に基づくべきである。こうすることで、TASは両方に同期される。このようなアプローチで次によく使われる同期のポイントは、既に述べたとおりである。SUTのテストには導入済みのテストが必要であり、手動テストの場合は単に従うべき手動テスト手順である可能性がある。

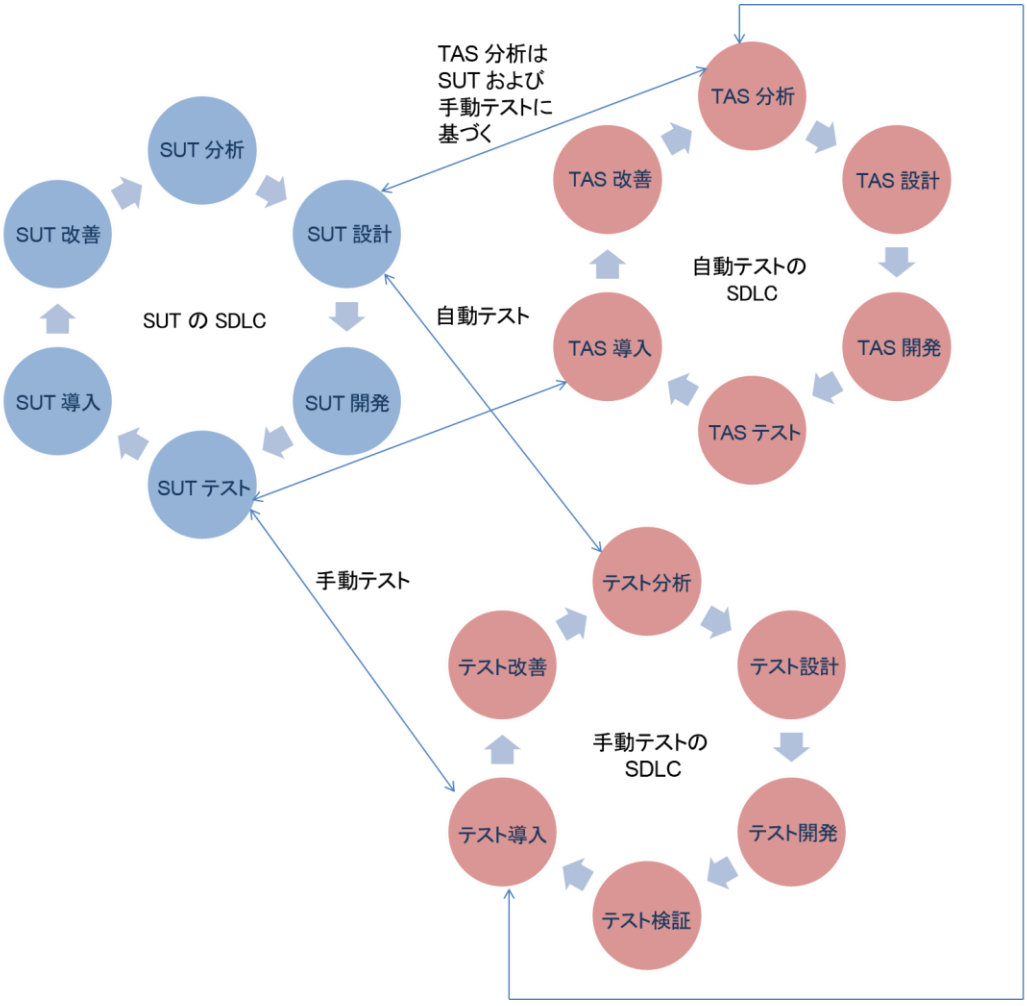

図 4: TASとSUTの開発プロセスの同期例 2

3.3.4 TASの再利用性の構築

TASの再利用とは、製品ライン、製品フレームワーク、製品ドメイン、プロジェクトファミリーで（任意のレベルのTASアーキテクチャの）TAS成果物を再利用することを指す。再利用の要件が生じるのは、他のプロダクトバリエーション、プロダクト、プロジェクトとの間でTAS成果物に関連性があるためである。再利用可能なTAS成果物には、以下が含まれる。

- モデル化されたテストの目標、テストシナリオ、テストコンポーネント、またはテストデータ（の一部）
- テストケース、テストデータ、テスト手順、またはテストライブラリ自身（の一部）
- テストエンジンおよび/またはテストレポートフレームワーク
- SUTのコンポーネントやインターフェースのアダプター

再利用の方向性はTAAが定義されるときに既に確定しているが、TASは以下によって再利用性を高めることができる。

- TAAに従う、または必要に応じて改訂または更新する
- TAS成果物を文書化することによって、わかりやすくし、新しい状況に組み込めるようにする

- すべてのTAS成果物の正確性を保証し、品質の高さによって新しい状況での使用が促進されるようにする

重要な点として、再利用性を高める設計は主にTAAに関することであるが、TASのライフサイクル全般にわたっては、再利用性を向上させるための保守や改善を考慮すべきである。再利用の実現、再利用による付加価値の測定や提示、既存のTASの再利用促進には、継続的な検討と作業が必要である。

3.3.5 さまざまな対象システムのサポート

TASによるさまざまな対象システムのサポートとは、TASがさまざまな構成のソフトウェアプロダクトをテストできることを指す。さまざまな構成とは、以下のいずれかを指す。

- SUTコンポーネントの数や相互連携
- SUTコンポーネントが実行される環境（ソフトウェアとハードウェアの両方）
- SUTコンポーネントを実装するために使われる技術、プログラミング言語、オペレーティングシステム
- SUTコンポーネントが使用するライブラリやパッケージ
- SUTコンポーネントを実装するために使用するツール

最初の 4 つの側面はTASのどのテストレベルにも影響するが、最後の側面は主にコンポーネントレベルと統合レベルのテストに適用される。

TASがさまざまなソフトウェアプロダクト構成をテストする能力は、TAAの定義の際に決定される。ただし、TASは技術的な相違に対応する機能を実装し、ソフトウェアプロダクトのさまざまな構成に必要なTASのフィーチャーやコンポーネントのマネジメントを実現しなければならない。

ソフトウェアプロダクトの多様性に関するTASの多様性に対処するには、次のような異なる方法で対応することもできる。

- TASやSUTのバージョン管理/構成管理を使用して、お互いに適合するTASとSUTのバージョンや構成を提供する
- TASのパラメーター化を使用して、TASがSUT構成に一致するように調整する

重要な点として、TASの変動性の設計は主にTAAに関することであるが、変動性を向上させるための保守や改善はTASのライフサイクル全般にわたって考慮すべきである。変動性の選択肢や形態を改訂、追加、削除するには、継続的な検討と作業が必要である。

4 導入のリスクとリスクヘッジ計画 - 150 分

キーワード

リスク、リスク軽減、リスクアセスメント、プロダクトリスク

「導入のリスクとリスクヘッジ計画」の学習の目的

4.1 テスト自動化アプローチの選択と導入/展開の計画

ALTA-E-4.1.1 （K3）効果的なテストツールパイロットと導入作業を支援するガイドラインを適用する

4.2 リスクアセスメントと軽減戦略

ALTA-E-4.2.1 （K4）導入のリスクを分析し、テスト自動化プロジェクトの失敗につながる可能性のある技術的な問題を特定し、軽減戦略を計画する

4.3 テスト自動化の保守

ALTA-E-4.3.1 （K2）TASの保守性を支える要素や、TASの保守性に影響を与える要素について理解する

4.1 テスト自動化アプローチの選択と導入/展開の計画

TASの実装と展開には、パイロットと導入という 2 つの主要な作業が関連する。これら 2 つの作業を構成するステップは、TASの種類とさまざまな状況によって異なる。

パイロットでは、少なくとも以下のステップについて考慮すべきである。
- 適切なプロジェクトの選定
- パイロットの計画
- パイロットの実行
- パイロットの評価

導入では、少なくとも以下のステップについて考慮すべきである。
- 最初の対象プロジェクトの選定
- 選択したプロジェクトへのTASの導入
- 計画した期間の後のプロジェクトのTASを監視および評価
- 他の組織/プロジェクトに展開

4.1.1 パイロットプロジェクト

通常、ツールの実装はパイロットプロジェクトから始まる。パイロットプロジェクトの狙いは、TASを使用して意図したメリットを実現できることを確認することにある。パイロットプロジェクトの目的には、以下が含まれる。
- TASに関する理解を深める。
- TASが既存のプロセス、手順、ツールにどのように適合するかを確認し、変更が必要になる点を特定する。(通常、既存のプロセス/手順に適合するようにTASを変更することが優先される。既存のプロセス/手順をTASをサポートするように変更する必要がある場合、少なくともプロセス自体を改善するべきである)
- テスト担当者のニーズに一致するように自動化のためのインターフェースを設計する。
- 構成管理や変更管理との統合を含むTASやテストのアセット(例えば、ファイルやテスト成果物の命名規約の決定、テストスイートのライブラリの作成と部品の定義など)の使用、管理、格納、保守の標準的な方法を決める。
- 使用性、保守性、拡張性など、テスト自動化を監視するためのメトリクスや測定手法を選定する。
- 期待するメリットが妥当なコストで実現可能かどうかを見極める。TASを使ってみた後で期待を見直すこともある。
- どのスキルが必要か、そのうちどれが満足し、どれが不足しているのかを判断する。

適切なプロジェクトの選定

パイロットプロジェクトは、以下のガイドラインに基づいて慎重に選択すべきである。
- 重要なプロジェクトを選ばない。TASの導入によって遅延を引き起こす場合、それが重要なプロジェクトに大きな影響を与えてはならない。TASの初期導入には時間がかかる。プロジェクトチームはその点を認識すべきである。
- 単純なプロジェクトを選ばない。単純なプロジェクトは適切な候補ではない。その導入に成功したとしても、単純でないプロジェクトで成功するとは限らず、導入に必要な情報もあまり得られないためである。
- 選定プロセスに必要なステークホルダー(マネジメント層を含む)を関与させる。
  - パイロットプロジェクトのSUTは、組織の他のプロジェクトの適切な基準となるべきである。例えば、SUTには自動化の必要がある代表的な GUIコンポーネントを含むべきである。

パイロットの計画

パイロットは通常の開発プロジェクトとして扱い、計画の作成、予算およびリソースの確保、進捗の報告、マイルストーンの定義などを行うべきである。そのうえで留意すべき点は、他のプロジェクトで作業のリソースが求められているとしても、TASの導入に携わる人員（スペシャリスト）が導入作業を十分に行えるようにすることである。特に共有リソースに関するマネジメント層の関与は重要である。これらの人員は、すべての時間を導入に充てられないことが多い。

TASがベンダーから提供されたものではなく、社内で開発したものである場合、その開発担当者を導入作業に関与させる必要がある。

パイロットの実行
導入のパイロットを実行し、以下の点に注意する。

- TASが期待どおりの機能を提供したか（また、ベンダーが述べたとおりであったか）。そうでない場合、できる限り早く対処する必要がある。社内でTASを開発した場合は、その開発担当者が不足している機能を提供し、導入を支援する必要がある。
- TASと既存のプロセスはうまく連携できているか。そうでない場合、連携させる必要がある。

パイロットの評価
評価にはすべてのステークホルダーを関与させる。

4.1.2 導入

パイロットの評価が完了し、それが成功と見なされた場合、TASを他の部署/組織に導入すべきである。展開は段階的に実施し、適切に管理する必要がある。導入の成功要因には、以下が含まれる。

- 段階的に展開する：他の組織に段階的に展開することで、新規ユーザーのサポートは一斉にではなく波状的に発生する。これにより、TASの使用を段階的に増やすことができる。ボトルネックの可能性を特定し、実際の問題が生じる前に対処できる。ライセンスは必要に応じて追加できる。
- TASの使用に適したプロセスの適合と改善を行う：異なるユーザーがTASを使用する場合、異なるプロセスがTASに関与する。そのため、プロセスをTASに合わせて調整したり、TASをプロセスに合わせて（わずかに）変更したりする必要がある。
- 新規ユーザーに対し、トレーニングやコーチング/メンタリングを行う：新規ユーザーには、新しいTASを使用するためのトレーニングやコーチングが必要になる。これらは確実に行われるようにする。トレーニング/ワークショップは、ユーザーが実際にTASを使用する前に提供すべきである。
- 利用ガイドラインを定義する：TASの使用に関する、ガイドライン/チェックリスト/FAQ を作成する。これにより、サポートにあたっての質問を大幅に減らすことができる。
- 実際の使用状況に関する情報収集の方法を実装する：実際のTASの使用状況についての情報を自動的に収集する方法を確立すべきである。使用状況だけではなく、TASのどの部分（特定の機能）が使用されているかの情報も得られることが理想的である。これによって、TASの使用状況を監視しやすくなる。
- TASの使用、メリット、コストを監視する：一定期間TASの使用状況を監視することで、TASが実際に使用されているかどうかがわかる。この情報は、ビジネスケース（どのくらいの時間を節約できたか、どのくらいの問題を防げたかなど）の再計算にも使用できる。
- TASのテストチームと開発チームを支援する。
- すべてのチームから、得られた教訓を集める：TASを使用した各チームと評価/振り返りミーティングを行う。このようにすることで、得られた教訓を特定できる。チームは、TASの使用法を改善するために、自分たちの情報提供が重要であり、その必要性を理解できる。
- 改善点を特定し、実装する：チームからのフィードバックとTASの監視に基づき、改善のステップを特定して実装する。また、この点をステークホルダーに明確に伝達する。

4.1.3 ソフトウェアライフサイクル内でのTASの導入

TASの導入は、TASを使用してテストを行うソフトウェアプロジェクトの開発フェーズに大きく依存する。

通常、新しいTASや新しいバージョンのTASは、プロジェクトの開始時か、コードフリーズやスプリント終了時などのマイルストーン到達時に導入される。これは、テストや修正を伴う導入作業には時間と労力がかかるためである。さらに、TASが動作しないためにテスト自動化プロセスに影響が及ぶというリスクを軽減する有効な手法でもある。ただし、TASに修正が必要な重要な問題がある場合や、TASが実行されている環境のコンポーネントを置き換える必要がある場合は、SUTの開発フェーズとは別に導入を行うことになる。

4.2 リスクアセスメントと軽減戦略

技術的な問題がプロダクトまたはプロジェクトのリスクにつながる可能性がある。一般的に見られる技術的な問題には、次のようなものがある。

- 過度な抽象化により、実際に何が起きているのかがわかりにくくなる（キーワード駆動など）
- データ駆動: データテーブルが大きすぎる/複雑すぎる/扱いにくい
- TASがオペレーティングシステムライブラリなどのコンポーネントに依存しているが、それらがSUTのすべての対象環境で利用できるとは限らない

一般的に見られる導入プロジェクトのリスクには、次のようなものがある。

- 要員の問題: コードベースを保守する適切な人員を確保するのが難しい
- 新たな SU 機能によってTASが正常に動作しなくなる
- 自動化の実現に遅れが生じる
- SUTの変更によってTASのアップデートに遅れが生じる
- TASで追跡が必要な（非標準）オブジェクトをキャプチャできない

TASプロジェクトで考えられる失敗する場面には、次のようなものがある。

- 異なる環境への移行
- 対象環境への導入
- 開発からの新しいデリバリー

このようなリスク領域に対処するために利用できるリスク軽減戦略が多数存在する。以下では、これらについて説明する。

TASには、社内で開発したものか獲得したソリューションかによらず、それぞれのソフトウェアライフサイクルがある。留意すべき点は、TASは他のソフトウェアと同じく、バージョンコントロールを行い、フィーチャーを文書化する必要があることである。そうしない場合、TASの一部を導入したり、連携して動作させたり、一部の環境で動作させたりすることが非常に難しくなる。

また、文書化された明確で使いやすい導入手順が必要である。この手順はバージョンに依存するため、バージョンコントロールに含める必要がある。

TASの導入には、2 つの異なる状況が存在する。

1. 初期導入
2. 保守 - TASが既に存在し、保守が必要な場合

初めてTASを導入する前に、その環境でTASが動作すること、予期せぬ変更の影響を受けないこと、テストケースを更新および管理できることを確認することが重要である。TASとそのインフラストラクチャの両方を保守しなければならない。

初回導入の場合は、以下の基本ステップが必要になる。
- TASを動作させるインフラストラクチャの定義
- TAS用のインフラストラクチャの構築
- TASおよびインフラストラクチャの保守手順の作成
- TASが実行するテストスイートの保守手順の作成

初回導入に関連するリスクには、次のようなものがある。
- テストスイートの合計実行時間がテストサイクルで計画した実行時間よりも長い。この場合、次に予定されたテストサイクルが始まる前に、テストスイートを実行する十分な時間を確保することが重要である。
- テスト環境のインストールや構成に関する問題が存在する（データベースのセットアップや初回ロード、サービスの起動/停止など）。通常、TASにはテスト環境での自動テストケース実行に必要な事前条件を設定する効率的な方法が必要である。

保守の場合は、追加で考慮すべき点がある。TAS自体の改善が必要であり、TASの更新を本番環境に適用する必要がある。他のソフトウェアと同じように、更新したTASを本番環境に適用する前にはテストを行う必要がある。そのため、新機能のチェックに加え、更新したTASでテストスイートが実行できること、レポートが送信できること、性能の問題や機能の回帰が起きていないことを検証する必要がある。場合によっては、新しいバージョンのTASに適用させるためにテストスイート全体の変更が必要になることもある。

保守には、以下のステップが必要になる。
- 新しいバージョンのTASと古いバージョンのTASを比べ、変更点を評価する
- 新機能と回帰の両方についてTASをテストする
- 新しいバージョンのTASにテストスイートを適合させる必要があるか確認する

更新には、以下のリスクおよび対応する軽減活動が伴う。
- 更新したTASで実行するために、テストスイートの変更が必要: テストスイートに必要な変更を行い、TASに導入する前にテストする。
- 更新したTASに適用させるために、テストで使われているスタブ、ドライバ、インターフェースに変更が必要: テストハーネスに必要な変更を行い、TASに導入する前にテストする。
- 更新したTASを組み込むためにインフラストラクチャに変更が必要: 変更が必要なインフラストラクチャコンポーネントを評価し、変更を行い、更新したTASを使用してテストする。
- 更新したTASに欠陥や性能の問題がある: リスクとメリットの分析を行う。見つかった問題によってTASの更新が不可能になる場合は、更新を行わない、あるいは次のバージョンのTASまで待機すべきである。問題がメリットに比べて無視できる程度なら、TASを更新してもよい。既知の問題を記載したリリースノートを作成し、テスト自動化エンジニアおよびステークホルダーに通知し、いつ問題が修正されるか見積るようにする。

4.3 テスト自動化の保守

TASの開発は単純なことではない。TASは部品化が可能であり、拡張性があり、わかりやすく、信頼性が高く、テスト可能でなければならない。さらに複雑になるが、TASは他のソフトウェアシステムと同様、改善し続けなければ

ならない。内部的な変更によるものか動作環境の変更によるものかを問わず、保守はTASのアーキテクチャにおいて重要な側面である。TASの保守では、テスト対象となる新しいタイプのシステムに適応させ、新しいソフトウェア環境をサポートし、新しい法律や規制に準拠させることにより、TASの動作の信頼性と安全性を高める。さらに、TASの寿命や性能も最適化する。

4.3.1 保守の種類

保守は運用中のTASで行われ、システムの変更、移行、廃止によって発生する。このプロセスは、以下のカテゴリに分けられる。

- 予防保守 - TASにより多くのテストタイプをサポートしたり、複数のインターフェースでのテストをサポートしたり、複数のバージョンのSUTでのテストをサポートしたり、新しいSUTでテスト自動化をサポートしたりするための変更を行う。
- 修正保守 - TASの故障を修正するための変更を行う。運用中のTASを保守しながらTAS使用におけるリスクを軽減する最適な方法は、定期的な保守テストを行うことである。
- 完成度を高めるための保守 - TASが最適化され、非機能的な問題が修正される。TASの性能、使用性、堅牢性、信頼性について対処する。
- 適応保守 - 市場に新しいソフトウェアシステム（オペレーティングシステム、データベースマネージャー、Web ブラウザなど）を投入する際、TASによるサポートが求められる場合がある。また、TASが新しい法律、規制、業界固有の要件に準拠しなければならない場合もある。その場合、TASが適切に適応できるように変更が行われる。注: 法律や規則に準拠する場合は、固有の規則や要件、場合によっては監査要件を伴う保守が必要になるのが一般的である。また、統合ツールが更新されて新しいバージョンが作成される場合、ツールの統合エンドポイントを保守して機能を保つ必要がある。

4.3.2 スコープとアプローチ

保守は、TASのすべてのレイヤーおよびコンポーネントに影響を及ぼす可能性のあるプロセスである。その範囲は以下の要因によって決定される。

- TASの規模と複雑度
- 変更の規模
- 変更のリスク

保守が運用中のTASを変更する点を考慮すると、変更によってシステムにどのような影響が及ぶかを判断するには、影響度分析が必須である。影響によっては、TASの機能を確実に継続させるため、段階的に変更を導入し、それぞれのステップ後にテストすることが必要である。注: 仕様や文書が古いと、TASの保守が困難になる場合がある。

テスト自動化の主な成功要因は時間効率であるため、以下のような手法を用いてTASの保守を行うことが重要である。

- TASの導入手順および使用方法は必ず明確化かつ文書化する
- サードパーティへの依存性、欠点、既知の問題は必ず文書化する
- コンポーネントを簡単に交換できるよう、TASは必ず部品化する
- TASは必ず移行可能な環境か、そのコンポーネントが移行可能な環境で実行する
- TASは必ず TAF自体からテストスクリプトを分離する
- TASの変更によってテスト環境が悪影響を受けないように、必ず開発環境から分離した状態でTASを実行する
- TASおよび環境、テストスイート、テストウェアは、必ず構成管理の対象とする

サードパーティコンポーネントやその他のライブラリの保守には、以下の点も考慮する。

- TASがテストを実行するためにサードパーティ製のコンポーネントを使用することは非常に多い。TASがサードパーティライブラリ（UI自動化ライブラリなど）に依存している場合もある。TASのサードパーティ製のコンポーネント部分は、必ずすべて文書化し、構成管理の対象とする。
- これらの外部コンポーネントの変更や修正が必要になった場合の計画が必要である。TASの保守責任者は、連絡先や問題の報告先を知っておく必要がある。
- サードパーティ製のコンポーネントを使用するライセンスについては文書化されており、そのコンポーネントが変更可能か、どの程度変更できるかという情報が必要になる。
- サードパーティ製のコンポーネントのそれぞれについて、更新や新バージョンについての情報を取得する必要がある。サードパーティのコンポーネントやライブラリを最新の状態に保つことは、長期的な投資に見合った予防措置である。

命名規約やその他の規約については、以下の点を考慮する。

- 命名規約やその他の規約を導入するメリットは単純である。テストスイートおよびTAS自身は、読みやすく、わかりやすく、変更や保守が簡単でなければならない。これによって保守プロセスにかかる時間を短縮でき、回帰や誤った修正が発生するリスクも最低限にとどめ、発生した場合は容易に取り除くことができる。
- 命名規約が使用されていれば、新しい人員がテスト自動化プロジェクトに参画しやすくなる。
- 命名規約は、変数やファイル、テストシナリオ、キーワード、キーワードパラメーターの名前を定義する。その他の規約は、テスト実行の前提条件や事後処理、テストデータの内容、テスト環境、テスト実行のステータス、実行結果記録、レポートを定義する。
- 命名規則やその他の規約は、テスト自動化プロジェクトの開始時にすべて同意され、文書化されていなければならない。

文書化については、以下の点を考慮する。

- テストシナリオとTASの両方について十分な最新の文書が必要となる点は極めて明らかであるが、これに関して 2 つの問題がある。文書を作成し、保守しなければならない。
- テストツールのコード設計書は自動または半自動で文書化することができるが、設計、コンポーネント、サードパーティ製のコンポーネントとの統合、依存性、導入手順については、誰かが文書化する必要がある。
- 適切な方法としては、開発プロセスの一部に文書作成を含めることができる。文書の作成または更新が行われるまでは、タスクは完了したとはみなされない。

トレーニング教材については、以下の点を考慮する。

- TASの文書が十分記述されている場合、これをTASのトレーニング教材のベースとして使用できる。
- トレーニング教材とは、TASの機能仕様、TASの設計およびアーキテクチャ、TASの導入および保守、TASの使用方法（ユーザーマニュアル）、実例や練習問題、使い方に関するヒントなどを組み合わせたものである。
- トレーニング教材の保守には、最初の作成と定期的な見直しが含まれる。見直しの作業は実際にはTASの講師となるチームメンバーが担当し、SUTのイテレーションの終盤（スプリントの終了時など）で行われる可能性が高い。

5 テスト自動化のレポートとメトリクス - 165 分

キーワード

自動テストコードの欠陥密度、カバレッジ、トレーサビリティマトリクス、同等の手動テスト工数、メトリクス、テスト結果記録作業、テストレポート作業

「テスト自動化のレポートとメトリクス」の学習の目的

5.1 TASメトリクスの選択

ALTA-E-5.1.1 （K2）テスト自動化戦略と有効性を監視するために使用できるメトリクスを分類する

5.2 測定の実施

ALTA-E-5.2.1 （K3）技術的およびマネジメント的な要件をサポートするメトリクス収集手法を実施する。テスト自動化のメトリクス測定をどのように実施できるかを説明する。

5.3 TASおよびSUTの結果記録

ALTA-E-5.3.1 （K4）TASおよびSUTのテスト結果記録を分析する

5.4 テスト自動化のレポート

ALTA-E-5.4.1 （K2）テスト実行レポートの作成および公開の方法を説明する

5.1 TASメトリクスの選択

本節では、テスト自動化戦略とTASの有効性および効率性を監視するために使用できるメトリクスに着目する。このメトリクスは、SUTおよびSUTの（機能および非機能）テストを監視するために使用するSUT関連のメトリクスとは異なり、プロジェクト全体のテストマネージャーが選択する。テスト自動化メトリクスを使用することで、TAM およびTAEはテスト自動化の目的に対する進捗を追跡し、TASに対して行われた変更の影響を監視することができる。

TASメトリクスは、外部および内部の 2 つのグループに分割できる。外部メトリクスは、他の作業（とりわけテスト活動）に対するTASの影響を測定するために使用される。内部メトリクスは、目的達成におけるTASの有効性および効率性を測定するために使用される。

通常、測定後のTASメトリクスには以下が含まれる。

- 外部TASメトリクス
  - 自動化のメリット
  - 自動テストを構築する工数
  - 自動テストで検出された故障を分析する工数
  - 自動テストを保守する工数
  - 故障と欠陥の比
  - 自動テストの実行時間
  - 自動テストケースの数
  - 成功結果および失敗結果の数
  - 誤った失敗結果および誤った成功結果の数
  - コードカバレッジ
- 内部TASメトリクス
  - ツールスクリプトのメトリクス
  - 自動テストコードの欠陥密度
  - TASコンポーネントのスピードと効率性

上記のそれぞれについて、以下で説明する。

自動化のメリット

TASのメリットを測定し記録することはとりわけ重要である。その理由は、コスト（一定期間に関与した人員の数）は目に見えやすいからである。テスト以外の作業に携わっている人員はコストの全体像をイメージできるが、達成できたメリットはイメージできない場合がある。

メリットの測定は、TASの目的によって異なる。通常は、時間や工数の節約、実行するテストの数の増加（カバレッジの広さや深さ、実行の頻度）、または再実行性の増加、リソースの効率的な活用、あるいはヒューマンエラーの削減といったメリットである。メリットの測定項目には次のようなものがある。

- 手動テスト工数を節約できた時間
- 回帰テストの実行時間の削減
- 実現できたテスト実行の追加サイクル数
- 実行された追加テストの数や割合
- テストケース全体に対する自動テストケースの比率（ただし、自動テストケースと手動テストケースの比較は容易ではない）
- カバレッジの増加（要件、機能、構造）

- TASによって早期に発見できた欠陥の数（欠陥の早期発見による平均的な利益がわかる場合、これを「計算」して予防されたコストの合計を導き出すことができる）
- TASによって発見できた欠陥のうち、手動テストでは見つからなかったと思われるものの数（信頼性の欠陥など）

なお、一般的にはテスト自動化によって手動テストの工数は節約できる。この工数は、その他の種類の（手動）テスト（探索的テストなど）に充てることができる。テスト自動化によって、このような手動テストを実行できるようになったため、追加テストで見つかった欠陥もTASの間接的なメリットと見なすことができる。TASがなければこれらのテストは行われず、追加の欠陥は見つからないはずである。

自動テストを構築する工数
テストを自動化する工数は、テスト自動化に関わる主なコストの 1 つである。これは同じテストを手動で実行するコストよりも高くなることが多く、テスト自動化の利用を拡大する上で障害となる場合がある。特定の自動テストを実装するコストはテストの内容に依存するところが大きいが、使われるスクリプティングアプローチ、テストツールの習熟度、環境、テスト自動化エンジニアのスキルレベルといったその他の要素にも左右される。

通常、規模が大きく複雑なテストの自動化は、短いテストや単純なテストの自動化よりも長い時間がかかるため、テスト自動化の構築コストの計算には平均構築時間が使われることがある。これは、同じ機能を対象としたテストや、あるテストレベルのテストなど、特定のテストセットの平均コストを考慮することにより、さらに精度を上げることができる。もう 1 つのアプローチは、テストを手動で実行する際に必要になる工数（同等の手動テスト工数、EMTE）の係数として構築コストを表現することである。例えば、テストケースの自動化には手動テスト工数の 2 倍（すなわち EMTE の 2 倍）が必要になることがある。

SUTの故障を分析する工数
自動テスト実行で検出されたSUTの故障を分析することは、手動で実行したテストで検出された故障を分析するよりも大幅に複雑になることがある。これは、手動テストで検出する故障に至るまでの事象は、テスト実行者が認識していることが多いためである。この点は、3.1.4 節に記載されている設計レベルと、5.3 節および 5.4 節に記載されているレポートレベルによって緩和できる可能性がある。この測定値は、失敗したテストケースあたりの平均または EMTE の係数として表現できる。後者は、自動テストの複雑度や実行時間が大きく異なる場合に特に適している。

利用できるSUTとTASの記録は、故障の分析において重要な役割を果たす。記録は、この分析を効率的に行うために十分な情報を提供すべきである。重要な記録機能には、次のようなものがある。

- SUTの記録とTASの記録は同期しているべきである
- TASは期待される振る舞いと実際の振る舞いを記録すべきである
- TASは実行されるアクションを記録すべきである

一方、SUTは実行されるすべてのアクションを記録すべきである（アクションが手動テスト結果でも、自動テスト結果でも構わない）。内部エラーはすべて記録し、クラッシュダンプやスタックトレースも利用できるようにすべきである。

自動テストを保守する工数
SUTと自動テストを同期するために必要な保守工数は、非常に大きくなる可能性があり、最終的には、TASによって実現できるメリットを上回ってしまうこともある。この点は、多くの自動化作業が失敗する原因となっている。そのため、保守工数を削減する必要がある場合、または少なくとも保守工数が制限なく増加することを防ぐ措置が必要になる場合は、保守工数の監視が重要になる。

保守工数の測定値は、SUTの新しいリリースごとに保守が必要になるすべての自動テストの合計で表現できる。または、更新された自動テストごとの平均または EMTE の係数として表現することもできる。
関連メトリックとして、保守作業が必要なテストの数や比率がある。

自動テストの保守工数がわかる場合（または算出できる場合）、その情報は特定の機能を実装するかどうか、または特定の欠陥を修正するかどうかを判断する際に重要な役割を持つ。ソフトウェアの変更によるテストケースの保守に必要な工数は、SUTの変更と合わせて考慮する。

故障と欠陥の比
自動テストで発生する一般的な問題は、多くの自動テストが同じ理由、すなわちソフトウェアの 1 つの欠陥のために失敗するということである。テストの目的はソフトウェアの欠陥を明らかにすることであるが、同じ欠陥を複数のテストで検出しても無駄になる。自動テストでは失敗した各テストを分析する工数が多くなる可能性があるため、特にその点が重要である。ある欠陥が原因で失敗した自動テストの数を測定することで、問題の発生箇所を明らかにできる。この点は、自動テストの設計と実行する自動テストの選択によって解決できる。

自動テストの実行時間
判断しやすいメトリクスの 1 つに、自動テストの実行にかかる時間がある。これはTASの最初の段階では重要でないこともあるが、自動テストケースの数が増えるにつれて、非常に重要なメトリクスになる。

自動テストケースの数
このメトリクスは、テスト自動化プロジェクトの進捗を示すために使用できる。しかし、自動テストケースの数だけでは、多くの情報を明らかにできないことを念頭に置く必要がある。例えば、自動テストケースの数はテストのカバレッジが増加したことを示すものではない。

成功結果および失敗結果の数
このメトリクスは、成功した自動テストの数と、期待結果を満たせずに失敗した自動テストの数を追跡する一般的なものである。失敗を分析し、SUTの欠陥による失敗なのか、環境やTAS自体の問題などの外部の問題による失敗なのかを判断する必要がある。

誤った失敗結果および誤った成功結果の数
上記のメトリクスのいくつかにも記したように、テストの失敗の分析にはかなりの時間がかかる場合がある。誤警告であることがわかった場合は、さらにもどかしさを感じることになる。時間がかかるのは、問題がSUTではなくTASまたはテストケースにある場合である。重要なのは、誤警告の数（および無駄になる可能性のある工数）を低く抑えることである。誤った失敗が発生すると、TASの信頼性が低下する。一方、誤った成功はさらに危険なものになる可能性がある。誤った成功とは、テスト自動化がSUTに存在する故障を特定できず、成功という結果が報告されることを言う。つまり、欠陥候補を検知できなかったことになる。これは結果の検証が適切に行われなかった場合、無効なテストオラクルが使用された場合、またはテストケースが誤った結果を期待していた場合に発生することがある。

なお、誤警告はテストコードの欠陥によって発生する場合もある（「自動テストコードの欠陥密度」メトリクスを参照）が、SUTが不安定で予想できない振る舞いをすることが原因となる場合もある（タイムアウトなど）。テストフックの干渉のレベルによって誤警告が起きる場合もある。

コードカバレッジ
さまざまなテストケースによってカバーされるSUTのコードカバレッジを把握することで、有用な情報を得ることができる。これは、ハイレベルの自動テストスクリプトでも測定することができる（回帰テストスイートのコードカバレッジなど）。カバレッジが十分であることを示す絶対的な比率は存在せず、相当単純なソフトウェアアプリケーション

でない限り、コードカバレッジが 100% になることはない。しかし、カバレッジが十分であるほど全般的なソフトウェア展開時のリスクを軽減できるため、カバレッジが大きいほどよいというのは一般的な共通認識である。このメトリクスは、SUTの活動を示す可能性もある。例えば、コードカバレッジが低下した場合、SUTに機能が追加されたにもかかわらず、対応するテストケースが自動テストスイートに追加されていない可能性が高いことを示す。

ツールスクリプトのメトリクス
自動スクリプトの開発を監視するために使用できるメトリクスは多数ある。そのほとんどは、SUTのソースコードのメトリクスと同様である。コードの行数（LOC）やサイクロマティック複雑度は、肥大したスクリプトや複雑なスクリプトを検知するために使用できる（再設計が必要な可能性を示す）。

コメントと実行ステートメントの比は、スクリプトの文書化や注釈の程度を示す可能性があるものとして使用することができる。スクリプト標準への準拠違反の数は、どの程度標準に準拠しているかを示す。

自動テストコードの欠陥密度
自動テストコードは、ソフトウェアであり欠陥を含むという点でSUTのコードと変わらない。自動テストコードは、SUTのコードより重要度が低いとは限らない。優れたコーディング慣習やコーディング標準を適用し、その結果をコードの欠陥密度などのメトリクスで監視すべきである。構成管理システムのサポートがあれば、これらの収集は容易になる。

TASコンポーネントのスピードと効率性
同じ環境で同じテスト手順を実行するためにかかる時間の差から、SUTの問題が明らかになる場合がある。SUTが同じ経過時間で同じ機能を実行できない場合、調査が必要である。これは許容範囲外のシステムの性能のばらつきを示している可能性があり、負荷が増加するとさらに悪化する可能性がある。TASはSUTの性能を妨げないように、十分な性能を発揮できる必要がある。性能がSUTの要件にとって重要な場合は、その点を考慮してTASを設計する必要がある。

傾向メトリクス
これら多くのメトリクスとともに重要なのが傾向（測定値の経時変化）の報告である。これは、特定の時間の測定値よりも重要になる場合がある。例えば、保守が必要な自動テスト当たりの平均保守コストが前の2つのSUTリリースの平均保守コストよりも多いことがわかれば、増加の原因を判断するために迅速な行動を取り、その傾向を改善させるステップを実行することができる。

測定コストはできる限り低く抑える必要があるが、多くの場合、収集とレポートを自動化することで実現できる。

5.2 測定の実施

テスト自動化戦略の中核には自動化テストウェアが存在するので、自動化テストウェアを拡張して使用状況に関する情報を記録することができる。抽象化と構造化されたテストウェアを組み合せていれば、基になるテストウェアに対して行われた拡張をすべてのハイレベル自動テストスクリプトで活用できる。例えば、テスト実行の開始時間と終了時間を記録するために基となるテストウェアを拡張すると、すべてのテストに適切に適用できる可能性がある。

測定とレポート生成をサポートする自動化の機能
多くのテストツールのスクリプト言語は、個々のテストや一連のテスト、テストスイート全体の実行前、実行中、実行後に情報を記録する機能によって、測定とレポートをサポートしている。

傾向(テスト成功率の変化など)を明らかにできるように、一連のテスト実行のそれぞれに対するレポートは、前のテスト実行結果を考慮した分析機能を備える必要がある。

通常、テストを自動化するには、テスト実行とテスト検証の両方を自動化する必要がある。後者は、テスト結果の特定の要素と事前に定義された期待結果を比べることによって実現できる。この比較はテストツールで行うのが一般的である。この比較の結果としてレポートされる情報のレベルを考慮する必要がある。テストのステータスを正確に判断することが重要である(成功、失敗など)。ステータスが失敗の場合、失敗の原因についての詳しい情報が必要になる(スクリーンショットなど)。

テストの実行結果と期待結果で予測される違いが簡単に判別されるとは限らないが、予測される違い(日付や時間など)を無視しつつ、予測されない違いを明らかにする比較を定義するにあたっては、ツールのサポートが大いに役立つ可能性がある。

その他のサードパーティツールとの統合(スプレッドシート、XML、文書、データベース、レポートツールなど)
自動化されたテストケースを実行して得られた情報を他のツール(トレーサビリティマトリクス更新などの追跡やレポート)で使用する場合、これらのサードパーティツールに適切な形式で情報を提供することができる。多くの場合、これは既存のテストツールの機能(レポート用にフォーマットをエクスポート)によって実現できる。または、他のプログラムと互換性のある形式(Excel の「.xls」、Word の「.doc」、Web の「.html」など)で出力できるようにレポートをカスタマイズすることによって実現できる。

結果の視覚化(ダッシュボード、図、グラフなど)
テスト結果は、図を使って視覚化すべきである。テスト実行では、色を使って問題を示すことを考慮する。例えば、報告された情報に基づいて判定を行えるように、信号機の色でテスト実行/自動化の進捗を示す。特にマネジメントには、視覚的にまとめてテスト結果を一目で確認できることが求められている。さらに情報が必要な場合は、詳細を確認することもできる。

5.3 TASおよびSUTの結果記録

結果記録作業はTASにとって非常に重要である。これには、テスト自動化自身とSUTの両方の結果記録作業が含まれる。テスト結果記録は、問題候補を分析する際に頻繁に使われる情報源である。本節では、TASまたはSU に分類されるテスト結果記録作業の例を示す。

TASの結果記録作業(TAFまたはテストケース自体のどちらが情報を記録するのかはさほど重要ではなく、状況に依存する)は、次を含むべきである。

- 現在実行中のテストケースと、開始時間および終了時間。
- テストケース実行のステータス。失敗はログファイルで容易に特定できるが、フレームワーク自体にもこの情報があり、ダッシュボードを通して報告される。テストケースの実行ステータスは、成功、失敗、TASエラーのいずれかである。TASエラーの結果は、問題がSUTではない場合に使用される。
- タイミング情報など、テスト結果記録のハイレベルな詳細(重要な手順の結果記録)。
- サードパーティツールとテストケースによって明らかになったSUTについての動的な情報(メモリリークなど)。実行結果や動的測定の失敗は、インシデントが検知された際に実行されていたテストケースとともに記録する。
- 信頼性テスト/ストレステスト(多数のサイクルが実施される)の場合は、テストケースが何回実行されたかが容易に判断できるように、回数を記録する。

- テストケースにランダムな部分がある場合（ランダムパラメーター、ステートマシンテストのランダムステップなど）、乱数/選択値を記録する。
- テストケースが実行するすべてのアクションは、ログファイル（またはその一部）を再生し、テストの同じステップを同じタイミングで正確に再現できるように記録する。これは、特定した故障の再現性を確認して追加情報を得る際に便利である。テストケースのアクション情報は、顧客が特定した問題を再現する際に使えるように、SUT自体に記録することもできる（顧客がシナリオを実行し、記録されたログ情報を開発チームがトラブルシューティングを行う際に再生する）。
- 故障の解析の際に使用できるよう、テスト実行の際にスクリーンショットなどの視覚情報をキャプチャしたものを保存しておくことができる。
- テストケースの実行時に故障が発生した場合、TASは問題の分析に必要なすべての情報が保存されており、利用できることを保証する。また、該当する場合は、テストの継続に関連する情報も同様である。関連するクラッシュダンプやスタックトレースがある場合、TASは安全な場所に保存する。上書き可能なログファイル（SUTのログファイルには循環バッファが使われることが多い）についても、後の分析で利用できるように、この場所にコピーする。
- 記録された情報の種類を区別する場合は、色分けするとよい（エラーは赤、進捗情報は緑など）。

SUTの結果記録作業は、次を含むべきである。

- SUTが問題を特定した場合、問題の分析に必要なすべての情報を記録する。これには、日付とタイムスタンプ、問題の発生元の位置、エラーメッセージなどが含まれる。
- SUTはすべてのユーザー操作を記録することができる（直接利用可能なユーザーインターフェースの他に、ネットワークインターフェースなども含まれる）。こうすることで、顧客が特定した問題を適切に分析し、開発側で問題の再現を試みることができる。
- システムの起動時には、各ソフトウェア/ファームウェアのバージョン、SUTの構成、オペレーティングシステムの構成などの構成情報をファイルに記録する。

さまざまな記録情報は、すべて簡単に検索できるべきである。TASがログファイルで特定した問題は、SUTのログファイルで簡単に特定でき、その逆も同様である（追加ツールがある場合もない場合もある）。タイムスタンプによってさまざまな記録を同期させると、エラーが報告された際に何が起きているのかを関連付けやすくなる。

5.4 テスト自動化のレポート

テスト結果記録は、テストケースやテストスイートの実行手順やアクションと応答について詳細な情報を提供する。しかし、結果記録だけでは、すべての実行結果の全体像を適切に把握することはできない。そのため、レポート機能が必要になる。テストスイートが実行されるたびに、簡潔なレポートを作成して公開しなければならない。そのために、再利用可能なレポートジェネレータコンポーネントを使用できる場合がある。

レポートの内容

テスト実行レポートは、実行結果の概要、テスト対象のシステム、テストを実行した環境がわかるサマリーを含まなければならない。このサマリーは、各ステークホルダーに提供することが望ましい。

どのテストがどのような理由で失敗したかを明らかにしなければならない。トラブルシューティングを簡単にするには、テストの実行履歴とその責任者（通常はテストの作成者または最終更新者）を知ることが重要である。責任者は失敗の原因を調査し、関連する問題を報告し、問題の修正をフォローアップし、正しく修正されたことを確認する必要がある。

レポートは、TAFコンポーネントの故障を診断する際にも使用される（7 章を参照）。

レポートの公開
レポートは、実行結果に興味を持つ者全員に公開される。Web サイトへのアップロード、メーリングリストへの送信、テストマネジメントツールなどの別のツールへのアップロードを行うことができる。実際には、ほとんどの場合、実行結果に関心がある人に講読の仕組みが提供されて、メールでレポートを受信することができれば、実行結果を参照および分析することができる。

SUTの問題がある部分を特定したり、頻繁に起こる回帰を含めたテストケースやテストスイートの統計を集計できるようにレポートの履歴を保管したりすることも考えられる。

6 手動テストから自動化環境への移行 - 120 分

キーワード
確認テスト、回帰テスト

「手動テストから自動化環境への移行」の学習の目的

6.1 自動化の移行条件
ALTA-E-6.1.1 （K3）テスト自動化の合目的性を判断するための移行条件を適用する
ALTA-E-6.1.2 （K2）手動テストを自動テストに移行する要素を理解する

6.2 回帰テストを自動化する際に必要な手順の特定
ALTA-E-6.2.1 （K2）回帰テストを自動化する際に考慮する要素を説明する

6.3新規のフィーチャーのテストを自動化する際に考慮する要素
ALTA-E-6.3.1 （K2）新規のフィーチャーのテストを自動化する際に考慮する要素を説明する

6.4 確認テストを自動化する際に考慮する要素
ALTA-E-6.4.1 （K2）確認テストを自動化する際に考慮する要素を説明する

6.1 自動化の移行条件

従来より、組織は手動テストを実施してきた。自動テスト環境への移行を決断する際には、現在の手動テストの状況を評価し、そのテストアセットを自動化する最も効果的なアプローチを判断しなければならない。既存の手動テストの構造は、自動化に適する場合も適さない場合もある。適さない場合は、自動化をサポートするためにすべてのテストの再作成が必要になることもある。または、既存の手動テストから関連する内容（入力値、期待結果、ナビゲーションパス）を抽出し、自動化を行う際に再利用してもよい。手動テスト戦略が自動化を考慮したものになっていれば、自動化への移行が容易な構造にできる。

すべてのテストが自動化できるとは限らず、またすべてを自動化するべきでもない。場合によっては、最初のテストのイテレーションは手動となるかもしれない。そのため、移行には考慮すべき側面が 2 つある。それは、既存の手動テストを自動化する初回の変換と、その後に新しい手動テストを自動化するために行う移行である。

また、テストの中には、自動化することによって初めて（効果的に）実行できるタイプのテストがあることにも留意する。これには、信頼性テスト、ストレステスト、性能テストなどが含まれる。

テスト自動化を行うと、ユーザーインターフェースのないアプリケーションやシステムをテストできる。その場合、ソフトウェアのインターフェースを通して統合レベルでテストを行える。この種のテストケースは手動でも実行できる（手動でコマンドを入力してインターフェースを実行する）が、これは現実的ではない。例えば、自動化を行うと、システムのメッセージキューにメッセージを直接挿入することが可能になる。こうすることにより、まだ手動テストを行うことができない早い段階でテストを開始でき、欠陥を早期に検出することができる。

テスト自動化の作業を始める前に、自動テストと手動テストのどちらを作成するかの合目的性（適応性と実現性）を考慮する必要がある。合目的性の条件には以下の内容が含まれるが、これらに限定されるものではない。

- 使用頻度
- 自動化の複雑度
- ツールの適合性とサポート
- テストプロセスの成熟性
- ソフトウェアプロダクトライフサイクルの段階における自動化の合目的性
- 自動化環境の持続可能性
- SUTの操作特性

以下では、上記のそれぞれについて詳しく説明する。

使用頻度
テストをどの程度の頻度で実行するかは、自動化するか否かの実現性に関する考慮事項の 1 つとなる。
メジャーおよびマイナーリリースサイクルの一部として定期的に実行されるテストは他のテストより使用頻度が高いので、自動化の候補となりやすい。汎用的なルールとしては、アプリケーションのリリースが頻繁で、対応するテストサイクルが多いほど、テストを自動化するメリットは大きい。
機能テストを自動化すれば、以降のリリースで回帰テストの一部として使用できる。回帰テストで使用される自動テストの投資収益率（ROI）は高く、既存のコードベースのリスク軽減にもなる。

テストスクリプトが 1 年に 1 度実行され、1 年以内にSUTが変更される場合は、自動テストの作成が現実的でも効果的でもないだろう。年ごとに行われるテストをSUTに対応させるためにかかる時間を考慮すると、手動でテストを行う方が望ましい場合がある。

自動化の複雑度
複雑なシステムをテストする必要がある場合、自動化によるメリットが非常に大きくなる可能性がある。手動テスト担当者は、手間や時間がかかり、エラーも起きやすい複雑な手順を繰り返すタスクを行わずに済む。

ただし、一部のテストスクリプトは自動化が困難だったり、自動化のコスト効率がよくない場合がある。この点には、さまざまな要素が影響する。例えば、既存の利用可能な自動テストソリューションと互換性のないSUT、自動化のために大量のプログラムコードとAPI呼び出しを開発するための要件、テスト実行の一部として対処しなければならないさまざまなシステムの複雑性、外部インターフェースや独自システムとの連携、使用性テストのいくつかの側面、自動化スクリプトのテストに必要な時間などがある。

ツールの適合性とサポート
アプリケーションの作成には、さまざまな開発プラットフォームが使用される。テスト担当者にとっての課題は、あるプラットフォームをサポートするためのテストツールがあればどれを利用できるか、そのプラットフォームがどの程度サポートされているかを知ることである。組織は、商用ベンダー、オープンソース、社内開発などによるさまざまなテストツールを使用する。テストツールをサポートするニーズやリソースは、組織ごとに異なる。通常、商用ベンダーは有償サポートを提供する。一般に、大手の商用ベンダーであればテストツールの実装を支援できる専門家のサポート環境が存在する。オープンソースツールはオンラインフォーラムなどのサポートを提供していることがあり、ユーザーはそこから情報を得たり、そこに質問を投稿したりすることができる。社内で開発されたテストツールは、既存のスタッフがサポートを提供する。

テストツールの適合性の問題は過小評価すべきではない。テストツールとSUTの適合性レベルを完全に理解せずにテスト自動化プロジェクトに着手すると、壊滅的な結果となることがある。SUTのテストの大半が自動化できたとしても、最も重要なテストが自動化できない場合もある。

テストプロセスの成熟性
テストプロセスで自動化を効果的に実装するには、そのプロセスを構造化し、統制し、繰り返し可能なものにしなければならない。自動化によって、既存のテストプロセスは自動化コードや関連するコンポーネントの管理が必要となり、それは開発プロセス全体へと影響を及ぼす。

ソフトウェアプロダクトライフサイクルの段階における自動化の合目的性
SUTにはソフトウェア開発ライフサイクルが存在するが、これは数年から数十年に及ぶ可能性がある。システム開発が始まると、エンドユーザーのニーズを満たすために欠陥への対処や機能の追加が行われ、システムは変化し拡張される。システム開発の初期段階では、変更が早すぎて自動テストソリューションの実装が追いつかない場合もある。画面レイアウトや制御の最適化や拡張が行われるにつれて、動的に変化する環境で自動化を行うために、作業のやり直しが継続的に必要になる場合があり、効率的、効果的ではない。これは走行中の車のタイヤを交換するようなものであり、車が止まるのを待つ方が望ましい。シーケンシャル開発環境を利用する大規模システムでは、システムが安定し、コア機能を組み込んだタイミングで自動テストの実装を始めることが望ましい。

最終的に、システムはそのプロダクトライフサイクルが終了すると、廃棄されるか、新しく効率的な技術を使用して再設計される。ライフサイクルの終了が近いシステムの自動化は推奨しない。このような短命な活動にはほとんど価値がないためである。しかし、既存の機能を保持しつつ、異なるアーキテクチャを使用して再設計されているシステムの場合、データ要素を定義した自動テスト環境は、古いシステムでも新しいシステムでも同じように有用である。この場合、テストデータの再利用が可能であり、自動化環境の記録に新しいアーキテクチャとの互換性を持たせることが必要である。

環境の持続可能性

自動化のテスト環境は、時間の経過とともにSUTに起こる変化に柔軟に対応できる必要がある。これには、自動化の問題をすばやく診断して修正できること、自動化コンポーネントを容易に保守できること、自動化環境に簡単に新規のフィーチャーやサポートを追加できることなどが含まれる。これらの特徴は、gTAAの全般的な設計や実装として不可欠な部分である。

SUTの操作特性（事前条件、セットアップ、安定性）
TAEは効果的な自動テストを作成する際に役立つSUTの操作特性や視覚特性を識別する。識別できない場合、テスト自動化は UI操作にのみ依存することになり、テスト自動化ソリューションの保守性が低下する。詳細については、2.3節の「試験性と自動化を考慮した設計」を参照のこと。

ROI 分析を支援する技術計画
テスト自動化によって、テストチームが得られるメリットの程度はさまざまである。しかし、効果的な自動テストソリューションの実装は多大な作業とコストを伴う。自動テストの開発に時間と工数を費やす前に、テスト自動化を実装する際に目的や可能性としてあがった全般的なメリットと成果について、評価を行う必要がある。それが決定した後、計画を実現するために必要な作業を定義し、それに伴うコストを判断して ROI を計算すべきである。

自動化環境への移行準備を十分整えるために、以下の領域に対処する必要がある。

- テスト自動化のテスト環境で利用できるツール
- テストデータとテストケースの正確性
- テスト自動化作業の範囲
- パラダイムシフトを起こすためのテストチームの教育
- 役割と責任
- 開発担当者とテスト自動化エンジニアの連携
- 並行作業
- テスト自動化のレポート

テスト自動化のテスト環境で利用できるツール
選定されたテストツールを先行検証環境にインストールし、機能することを確認する必要がある。これには、サービスパックやリリースアップデートのダウンロード、SUTをサポートするために必要かつ適切なインストール構成の選択（アドインを含む）、検証環境と自動化の開発環境でTASが正しく機能することの確認などが含まれる。

テストデータとテストケースの正確性
自動化した際に確実に予測できる結果を出すには、手動テストデータとテストケースが正確かつ完全である必要がある。自動テスト実行には、明示的な入力データ、ナビゲーション、同期化、検証が必要である。

テスト自動化作業の範囲
自動化を早期に成功させ、進捗に影響する可能性がある技術的問題のフィードバックを得るには、限定された範囲から始め、その後の自動化タスクを容易にする。パイロットプロジェクトは、システム全体の相互運用性を代表するシステムの機能の 1 つの領域を対象とすることができる。パイロットから学んだ教訓は、今後の時間の見積りやスケジュールの調整、および特殊な技術を持つリソースが求められる領域の特定に役立つ。パイロットプロジェクトでは、早期に自動化を成功させる方法を特定でき、その成功によって、マネジメントからの支持も集まる。

このためには、自動化するテストケースを適切に選択する必要がある。自動化に手間がかからず、高い付加価値を提供するテストケースを選択する。自動での回帰テストやスモークテストを実装することもできる。通常、このようなテストは、非常に頻度が高く、ときには毎日実行されるので、かなりの付加価値となる。着手点となるもう 1 つの優れた候補は、信頼性テストである。このようなテストは複数のステップで構成されることが多く、何度も繰り返し実行され、手動では明らかにしにくい問題を顕在化させる。このような信頼性テストは、実装に手間をかけずに、短時間で付加価値を示すことができる。

このようなパイロットプロジェクトでは、手動テストの工数を節約したり、重大な問題を特定できるため、自動化に注目を集めることができる。これによって、工数や予算などの点から、今後の拡張への道が開かれる。

また、組織にとって重要なテストは最初から最大の価値が明らかであるため、優先度を高くすべきである。ただし、この状況では、パイロット作業の中で技術的に難易度が高すぎるテストの自動化を避けることが重要である。避けられない場合、自動化の開発に多大な工数がかかる割に、示すべき結果がほとんどないことになる。汎用的なルールとしては、アプリケーションの大部分に見られる特徴を共有するテストを特定することが、自動化の作業存続に必要な推進力となる。

パラダイムシフトを起こすためのテストチームの教育

テスト担当者にはさまざまな特徴がある。特定のドメインの専門家でエンドユーザーコミュニティに所属していたり、ビジネスアナリストとして関与していたりする場合もある。また、基になるシステムアーキテクチャを深く理解できる優れたな技術スキルを持つ者もいる。効果的なテストを実現するには、さまざまな背景を持つ人員を混在させることが望ましい。テストチームが自動化へとシフトするにつれ、役割はそれぞれに特化されたものになる。テストチームの構成を変えることは、自動化の成功にとって不可欠である。予定されている変更について早くからチームを教育することは、役割に関する不安や不必要な考えが生じる可能性を軽減するために役立つ。適切に対処すれば、テストチームの全員が自動化へのシフトに期待するとともに、組織や技術の変化に参画する準備が整うことになる。

役割と責任

テスト自動化は、全員が参画できる活動にすべきである。ただし、全員が同じ役割を担うということではない。設計や実装、自動テスト環境の保守は本質的に技術的な作業であるため、プログラミングのスキルに強く、技術的背景を持つ人員を割り当てる。自動テストの開発作業の結果として、技術者であっても非技術者であっても同じように使用できる環境としなければならない。自動テスト環境の価値を最大限に引き出すためには、適切なテストスクリプト（対応するテストデータを含む）を開発する必要があるため、ドメインの専門性とテストスキルを持つ人員が必要である。このような人員は、自動化環境を推進し、目標のテストカバレッジを実現するために動員される。ドメインの専門家はアプリケーションの機能を確認するためにレポートをレビューし、技術的な専門家は自動化環境が正常かつ効率的に運用されることを保証する。このような技術的な専門家は、テストに関心のある開発担当者である場合もある。保守性の高いソフトウェアの設計にソフトウェア開発の経験は不可欠であり、テスト自動化にとっては特に重要である。開発担当者は、テスト自動化フレームワークやテストライブラリに集中することができるよう、テストケースの実装は、引き続きテスト担当者が行うべきである。

開発担当者とテスト自動化エンジニアの連携

テスト自動化の成功には、テスト担当者に加え、ソフトウェア開発チームの関与も必要である。テスト自動化においては、開発担当者が開発の手法やツールについての支援要員や技術情報を提供できるように、開発担当者とテスト担当者がより密接に連携して作業する必要がある。システム設計や開発コードの試験性については、テスト自動化エンジニアが懸念を示す場合がある。これは特に、設計やコードが標準に従っていない場合や、開発担当者が見慣れないライブラリやオブジェクト、または内製のものや極端に新しいものを使用している場合などが考えられる。例えば、開発担当者が選定された自動化ツールと適合性のないサードパーティ製 GUIコントロールを

選ぶ可能性もある。最終的には、組織のプロジェクトマネジメントチームが自動化作業の成功に必要な役割の種類と責任を明確に理解していなければならない。

並行作業
移行活動の一部として、多くの組織が並行チームを編成し、既存の手動テストスクリプトを自動化する作業を始めている。その後、新しく自動化されたスクリプトがテスト作業に組み込まれ、手動スクリプトに代わるものとなる。ただし、ほとんどの場合は、その前に自動スクリプトと置換対象となる手動スクリプトを比較し、自動スクリプトが手動スクリプトと同じテストを実行して確認していることを検証することが推奨される。

自動化への移行前に手動スクリプトの評価を行う例も多数ある。このような評価の結果、自動化によってさらに効率的、効果的なアプローチを実現するために、既存の手動テストスクリプトを再構築する必要があると判断される場合もある。

テスト自動化のレポート
TASが自動生成できるレポートには、さまざまなものがある。これには、個々のスクリプトやスクリプト内の手順の合格/失敗のステータス、テスト実行全体の統計、TASの全般的な性能などが含まれる。報告されるアプリケーション固有の結果が正確かつ完全であると見なされるように、TASの正しい運用を可視化することも重要である（第 7 章「TASの検証」を参照）。

6.2 回帰を自動化する際に必要な手順の特定

回帰テストは、自動化を使用する絶好の機会である。今日の機能テストが翌日の回帰テストになるため、回帰テストのテストベッドは増加し続ける。回帰テストの数が、従来の手動テストチームが利用できる時間とリソースを超えるのは時間の問題である。

自動回帰テストを準備する段階においては、次のような問いに答える必要がある。
- どのくらいの頻度でテストを実行するのか。
- 個々のテストの実行時間、回帰テストスイートの実行時間はどのくらいか。
- テスト間で機能の重複はないか。
- テストでデータが共有されているか。
- テストはお互いに依存しているか。
- テスト実行の前にどのような事前条件が必要になるか。
- テストのSUTのカバレッジは何 % か。
- 現在のテストは失敗なく実行されるか。
- 回帰テストに時間がかかりすぎた場合、何が起きるか。

以下では、上記のそれぞれについて詳しく説明する。

テスト実行の頻度
回帰テストテストの一部として頻繁に実行されるテストは、自動化の候補として最も適している。このようなテストは既に開発されており、SUTの既知の機能性をテストするものである。そして、自動化によって実行時間を大幅に減らすことができる。

テスト実行の時間
あるテストやテストスイート全体を実行するために必要な時間は、回帰テストで自動テストを実装する価値を評価する上で重要なパラメーターである。選択肢の 1 つとして、時間のかかるテストから自動化の実装を始めることが挙げられる。これによって毎回のテストの実行が速く効率的になり、自動回帰テストの実行サイクルを増やすこともできる。このメリットは、SUTの品質に関するフィードバックの数と頻度が増加し、展開時のリスクが減ることである。

機能の重複
既存の回帰テストを自動化する場合、テストケース間に存在する機能の重複を見つけ、可能な場合は同等の自動テストで重複を減らすことが望ましい。これにより、自動テスト実行時間の面でさらに効率が上がる。実行される自動テストが増えるにつれ、この点の重要性は増していく。多くの場合、自動化を使用して開発するテストは新しい構造になる。再利用可能なコンポーネントと共有データリポジトリに依存するためである。既存の手動テストを分解して複数の小さな自動テストにすることは、それほど珍しくない。同じように、いくつかの手動テストを統合して大きな自動テストにすることも、適切な解決策になる場合がある。効果的な変換の戦略を実現するために、手動テストには個々の評価とグループとしての評価が必要になる。

データ共有
テストでは、頻繁にデータが共有される。これは、テストが同じレコードのデータを使用して別のSUT機能を実行する場合に行われる。この例としては、社員が利用できる休暇時間を検証するテストケース「A」と、社員がキャリアアップのために受講できるコースを検証するテストケース「B」などがある。どちらのテストケースも同じ社員を使用するが、異なるパラメーターを検証している。手動テスト環境では、社員データは手動テストケースごとに複数回コピーされ、その社員を使用して社員データを検証するのが一般的である。しかし、自動テストでは、重複やエラーを防ぐため、共有するデータをできる限り 1 つのソースに格納し、そこからアクセスするべきである。

テストの相互依存性
複雑な回帰テストシナリオを実行する場合、1 つのテストが 1 つまたは複数の他のテストに依存する場合がある。これは非常に発生頻度が高く、テストのステップを実行して新しい「注文 ID」が作成された場合などに発生する。後続のテストは、a）新しい注文がシステムに正しく表示される、b）注文の変更が可能である、c）注文の削除が成功することを検証する場合がある。どの場合も、最初のテストで動的に生成される「注文 ID」の値をキャプチャし、以降のテストで再利用しなければならない。TASの設計によっては、この点に対処できる。

テストの事前条件
多くの場合、初期条件を設定する前にテストを実行することはできない。このような条件には、テストを行う正しいデータベースやテストデータセットの選択、パラメーターの初期値の設定などが含まれる。テストの事前条件を確立するために必要な初期手順の多くは、自動化することができる。これによってテストの実行前にステップの漏れを防ぎ、さらに信頼できるソリューションを実現できる。回帰テストを自動化する際には、このような事前条件を自動化プロセスの一部に含める必要がある。

SUTのカバレッジ
テストが実行されるたびに、SUTの機能の一部が実行される。SUT全体の品質を確認するためには、カバレッジを最大限に広く、深くするようにテストを設計する必要がある。さらに、コードカバレッジツールを使用して自動テストの実行を監視し、テストの有効性の定量化を進めることもできる。自動回帰テストにより、時間の経過とともにテストが追加され、カバレッジを高めることが期待される。この測定が、テスト自体の価値を定量化する有効な手段となる。

実行可能なテスト
手動回帰テストを自動テストに変換する前に、手動テストが正しく動作することを検証することが重要である。これは、自動回帰テストへの変換を確実に成功させるための正しい開始点となる。テストの記述が適切でない、無効なデータを使用している、期限切れまたは現在のSUTと同期していない、あるいはSUTに欠陥があることが原因で手動テストが正しく実行されない場合、故障の根本原因の理解や解決が行われないまま自動テストに変換すると、機能しない自動テストが作成されることになる。これは無駄な行為であり、非生産的である。

大規模な回帰テストセット
SUTの回帰テストセットは巨大になりがちで、テストセットを夜間または週末のうちに実行できないほどの大きさになることもある。その場合、複数のSUTが利用できるなら、テストケースの同時実行などが考えられる（PC アプリケーションの場合はこれで問題になることはほぼないが、SUTが航空機や宇宙ロケットのシステムである場合は話が違ってくる）。SUTが高価で入手しにくく、並列化するのが現実的ではない場合もある。その場合、回帰テストの一部だけを実行することも考えられる。長期間（たとえば数週間）かけることで、最終的にセット全体が実行される。また、回帰テストスイートのどの部分を選んで実行するかをリスク分析（最近変更したのはSUTのどの部分か）によって決めることができる。

6.3 新規のフィーチャーのテストを自動化する際に考慮する要素

一般的には、新しい機能のテストケースの方が自動化しやすい。これは、実装がまだ完了していない、あるいは始まってすらいないためである。テストエンジニアは、テスト自動化ソリューションによって、効果的かつ効率的にテストできるようにするためには新機能に対して何が必要かを、自分の知識を使用して開発担当者やアーキテクトに対して厳密に説明できる。

SUTに新規のフィーチャーが導入されると、テスト担当者は新フィーチャーと対応する要件に関連する、新しいテストを開発する必要がある。TAEは、ドメインの専門知識を持つテスト設計者によるフィードバックを求め、現在のTASが新規のフィーチャーのニーズを満たすかどうかを判断しなければならない。この分析には、使用されている既存のアプローチ、サードパーティ製開発ツール、使用されているテストツールなどが含まれるが、これに限られるものではない。

TASの変更は既存の自動テストウェアコンポーネントに対して評価し、変更や追加が完全に文書化され、振る舞い（または性能）や既存のTASの機能に影響しないようにする必要がある。

新規のフィーチャーが別のクラスやオブジェクトなどで実装される場合、テストウェアコンポーネントの更新や追加が必要になることがある。さらに、既存のテストツールとの互換性を評価し、必要な場合は代替ソリューションを特定しなければならない。例えば、キーワード駆動アプローチを使用している場合、新機能に対応するために追加キーワードの開発や既存のキーワードの変更または拡張が必要になることがある。

新機能が存在する新しい環境をサポートするために、追加テストツールを評価する要件が生じることもある。例えば、既存のテストツールが HTML しかサポートしない場合、新しいテストツールが必要になる場合がある。

新しいテストの要件は、既存の自動テストやテストウェアコンポーネントに影響を与える可能性がある。そのため、変更を行う前に新しいSUTや更新されたSUTに対して既存の自動テストを実行し、既存の自動テストに対する正しい動作が変更されないかどうかを確認し、記録すべきである。これには、他のテストとの相互依存性のマッピングも含まれるべきである。技術面での新しい変更点では、現在のテストウェアコンポーネント（テストツール、機能ライブラリ、APIなど）と既存のTASとの互換性に関する評価が必要になる。

既存の要件が変更された場合、要件を検証するテストケースの更新作業をプロジェクトのスケジュール（ワークブレークダウンストラクチャー（WBS））に含めるべきである。要件からテストケースまでのトレーサビリティがあれば、どのテストケースを更新する必要があるかがわかるはずである。このような更新は、全体計画の一部に含めるべきである。

また、既存のTASが現在のSUTのニーズを満たし続けるかどうかを判断する必要がある。実装技術はまだ有効か、それとも新しいアーキテクチャが必要になるか、現在の機能を拡張して行うことができるか。

新機能が導入されるタイミングは、テストエンジニアにとって新しく定義された機能がテスト可能であることを確認する機会となる。設計フェーズの間にテストについても考慮しておくべきである。それは、スクリプト言語やテスト自動化ツールで新機能を検証できる、テストインターフェースの提供を計画することによって行う。詳細については、2.3節の「試験性と自動化を考慮した設計」を参照のこと。

6.4 確認テストを自動化する際に考慮する要素

確認テストは、報告された欠陥に対処するためにコードを修正した後に実行される。通常、テスト担当者は欠陥を再現するステップを実行し、欠陥が存在しなくなっていることを検証する。

欠陥は以降のリリースで再発する可能性があるため（これは構成管理に問題がある可能性を示している）、確認テストは自動化の主な候補となる。自動化により、確認テストの実行時間を削減できる。確認テストは、既存の自動回帰テストベッドに追加、補完することができる。

通常、自動確認テストの機能スコープは狭い。実装は、欠陥が報告され、それを再現するために必要なステップが理解された後にいつでも行うことができる。自動確認テストは、標準の自動回避スイートに組み込んだり、それが効果的な場合は、既存の自動テストに取り入れることができる。どちらのアプローチでも、自動欠陥確認テストの価値は保持される。

自動確認テストの追跡を行うと、欠陥の解決に費やされたサイクルの時間や回数を追加報告できる。

欠陥の修正の副作用として新しい欠陥が増えていないことを確認するには、確認テストに加えて回帰テストも必要になる。回帰テストの適切なスコープを決定するには、影響度分析が必要になる場合もある。

7 TASの検証 - 120 分

キーワード
検証

「TASの検証」の学習の目的

7.1 自動テスト環境のコンポーネントの検証
ALTA-E-7.1.1（K3）テストツールの設定を含む自動テスト環境の正当性を検証する

7.2 自動テストスイートの検証
ALTA-E-7.2.1（K3）自動テストスクリプトやテストスイートの振る舞いの正当性を検証する

7.1 自動テスト環境のコンポーネントの検証

テスト自動化チームは、自動テスト環境が期待通りに動作していることを検証する必要がある。この確認は、テストの自動化を始める前などに行われる。

自動テスト環境のコンポーネントを検証するステップは複数存在する。以下では、そのそれぞれについて詳しく説明する。

テストツールのインストール、セットアップ、設定およびカスタマイズ
TASは多数のコンポーネントで構成され、それぞれが信頼性と再現性のある性能を確保するように考慮される必要がある。TASの中核は、実行可能なコンポーネント、対応する機能ライブラリ、サポートデータ、設定ファイルである。TASの構成プロセスは、自動インストールスクリプトを使用するものから、対応するフォルダに手動でファイルを配置するものまで、さまざまである。オペレーティングシステムやその他のアプリケーションと同じように、一般的なテストツールにはサービスパックやオプションのアドインなどが存在し、これらによって、SUT環境との互換性が確保される。

セントラルリポジトリからの自動インストール（またはコピー）には、複数のメリットがある。該当する場合、異なるSUTのテスト を同じバージョンのTASや同じ構成のTASで確実に実行できる。TASのアップグレードもこのリポジトリを通して行える場合がある。リポジトリの使用方法とTASを新しいバージョンにアップグレードするプロセスは、標準開発ツールの使用方法およびプロセスと同じにすべきである。

合格と失敗が既知であるテストスクリプト
合格することがわかっているテストケースが失敗した場合、何かが根本的に間違っており、できる限り早急に修正する必要があることは明らかである。逆に、失敗するはずのテストケースが合格した場合、正しく機能していないコンポーネントを特定する必要がある。正しい世代のログファイルと性能メトリックを検証するとともに、テストケース/スクリプトの自動セットアップおよび終了処理を検証することが重要である。タイプやレベルが異なるテスト（機能テスト、性能テスト、コンポーネントテストなど）から少数のテストを実行することも有用である。これは、フレームワークのレベルでも実行すべきである。

テスト環境のセットアップ/終了処理における再実行性
TASは、さまざまなシステムおよびサーバーで実装される。TASを各環境で適切に動作させるには、任意の環境からTASをロードおよびアンロードするシステム的なアプローチが必要になる。TASの構築や再構築の際に、内部的な動作や複数の環境で目に見える差異が発生しない場合、これがうまく実現できていることになる。TASコンポーネントの構成管理を行えば、信頼できる形で構成を作成できるようになる。

テスト環境およびコンポーネントの構成
TASを構成するさまざまなコンポーネントを理解して文書化することで、SUT環境が変更された場合に 、TASのどの側面が影響を受けたり変更が必要になったりする可能性があるかについての知識を得ることができる。

内部および外部のシステム/インターフェースに対する接続性
SUT環境にTASをインストールしたら、実際にSUTを使用する前に、一連のチェックや事前条件を管理し、内部および外部のシステムやインターフェースなどに接続できるようにする。自動化のための事前条件を確立することは、TASを正しくインストールおよび構成するために不可欠である。

自動テストツールの干渉関係
多くの場合、TASはSUTと密接に結合している。これは、設計的に高いレベルの互換性を実現するためであり、特に GUIレベルの操作が関係する場合に当てはまる。ただし、この密接な統合には、マイナスの効果がある場合

もある。例えば、TASがSUT環境の中にある場合とない場合でSUTの動作が異なる、手動で使用する場合にSUTが異なる振る舞いを見せる、環境内のTASによって、またはSUTに対してTASを実行するときに、SUTの性能が影響を受ける、などである。

干渉のレベルまたは干渉関係は、選択された自動テストアプローチによって異なる。次に例を示す。

- 外部インターフェースからSUTに接続する場合、干渉のレベルは非常に低くなる。外部インターフェースは、電気信号（物理スイッチの場合）、USB 装置の場合は USB 信号（キーボードなど）が考えられる。このアプローチを使用すると、最も適切な形でエンドユーザーをシミュレートできる。このアプローチでは、SUTのソフトウェアをテストの目的によって変更することはまったくない。テストアプローチによってSUTの振る舞いやタイミングが影響を受けることはない。この方式によるSUTとのインターフェースは、非常に複雑になる可能性がある。例えば、専用ハードウェアが必要になる場合がある、SUTとのインターフェースにハードウェア記述言語が必要なことなどが考えられる。これはソフトウェアのみのシステムでは典型的なアプローチではないが、組込みソフトウェア製品ではよく使われる。
- GUIレベルでSUTと接続する場合、UIコマンドをインジェクションしてテストケースが必要とする情報を抽出できるように、SUT環境を変更する。SUTの振る舞いは直接的に変更されないが、タイミングには影響があり、それが結果的に振る舞いに影響する可能性がある。干渉のレベルは前述の項目よりも高いが、複雑さはこの方式でのSUTとのインターフェースの方が低くなる。この種の自動化には、市販ソフトウェアツールを使えることが多い。
- ソフトウェアのテストインターフェースを使うか、既にソフトウェアが提供している既存のインターフェースを使ってSUTとの接続を行う。これらのインターフェース（API）の可用性は、試験性を考慮した設計の重要な部分である。この場合、干渉のレベルは非常に高くなる可能性がある。自動テストは、システムのエンドユーザーにまったく使用されないインターフェース（テストインターフェース）を使用する場合がある。または、実際とは異なる状況でインターフェースが使われる場合がある。一方で、インターフェース（API）を経由して自動テストを行うのは、非常に簡単で安価である。潜在的なリスクを理解している限り、テストインターフェースを経由してSUTのテストを行う方が確実である。

干渉のレベルが高いと、実世界の使用条件では起こりえない故障がテストで起こる可能性がある。これによって自動テストが失敗した場合、テスト自動化ソリューションの信頼性が大幅に低下する可能性がある。開発担当者は、可能であれば分析を支援できるように、自動テストで特定された故障を手動で再現させることを最初に要求する場合がある。

フレームワークコンポーネントテスト

あらゆるソフトウェア開発プロジェクトと同じように、自動フレームワークコンポーネントは個々にテストし、検証する必要がある。これには、機能テストと非機能テスト（性能、資源効率性、使用性など）が含まれる場合がある。

例えば、GUIシステムでオブジェクト検証を行うコンポーネントは、そのオブジェクト検証が正しく機能するように幅広いオブジェクトのクラスに対してテストを行う必要がある。同様に、エラーログおよびレポートは、自動化の状況とSUTの振る舞いについて正確な情報を提供する。

非機能テストには、フレームワークの性能低下の把握、メモリリークなどの問題を示す可能性があるシステムリソースの使用状況、フレームワークの内外のコンポーネントの相互運用性などが含まれる。

7.2 自動テストスイートの検証

自動テストスイートに対して、完全性、一貫性、正しい振る舞いを確認するテストを行う必要がある。常に自動テストスイートが動作していることを確認したり、問題なく使用できることを判断したりするために、さまざまな種類の検証チェックを適用できる。

自動テストスイートの検証手順は多数存在する。以下のようなものが挙げられる。

- 既知の合格と失敗があるテストスクリプトの実行
- テストスイートの確認
- フレームワークの新フィーチャーに着目した新しいテストの検証
- テストの再実行性の考慮
- 自動テストスイートに十分な検証ポイントが存在あることの確認

以下では、上記のそれぞれについて詳しく説明する。

既知の合格と失敗があるテストスクリプトの実行
合格することがわかっているテストケースが失敗した場合、何かが根本的に間違っており、できる限り早急に修正する必要があることは明らかである。逆に、失敗するはずのテストスイートが合格した場合、正しく動作していないテストケースを特定する必要がある。ログファイルと性能データを正しく生成し、テストケース/スクリプトのセットアップおよび終了処理を検証することが重要である。タイプやレベルが異なるテストから少数のテスト（機能テスト、性能テスト、コンポーネントテストなど）を実行することも有用である。

テストスイートの確認
テストスイートの完全性（すべてのテストケースに期待結果、テストデータが存在すること）と、正しいバージョンのフレームワークとSUTがあることを確認する。

フレームワークの新フィーチャーに着目した新しいテストの検証
テストケースで初めてTASの新フィーチャーを使用する場合、細かく検証および監視を行ってそのフィーチャーが正しく動作することを確認する。

テストの再実行性の考慮
テストを繰り返して実行する場合、テストの結果/判定は常に同じになるべきである。テストセット内で信頼性のある結果にならない（競合状態など）テストケースは、アクティブな自動テストスイートから移動させて個別に分析し、根本原因を調査することができる。そうしないと、テスト実行の際に何度も問題を分析する時間を使うことになる。

断続的な失敗は分析する必要がある。問題はテストケース自身の中にある場合も、フレームワークにある場合もある（SUTの問題である可能性もある）。ログファイル（テストケース、フレームワーク、SUTのもの）の分析により、問題の根本原因を特定できる場合がある。デバッグが必要になる場合もある。根本原因を見つけるために、テストアナリスト、ソフトウェア開発担当者、ドメインの専門家の支援が必要になる場合がある。

自動テストスイートやテストケースに十分な検証ポイントが存在することの確認
自動テストスイートが実行されており、期待結果を実現していることを検証できなくてはならない。テストスイートやテストケースが期待通りに実行されていることを確認できる証拠を提供しなければならない。この証拠には、各テストケースの開始時および終了時の記録、完了したテストケースのテスト実行ステータスの記録、事後条件が実現されているかの検証などが含まれる。

8 継続的な改善 - 150 分

キーワード
保守

「継続的な改善」の学習の目的

8.1 テスト自動化の改善オプション

ALTA-E-8.1.1 （K4）導入したテスト自動化ソリューションの技術的側面を分析し、改善の推奨策を提供する

8.2 テスト自動化の環境およびSUTの変更への適応

ALTA-E-8.2.1 （K4）テスト環境やSUTの一連の変更の後、統合や更新を行うべき場所を把握するために、テスト環境コンポーネント、ツール、サポート機能ライブラリなどの自動テストウェアを分析する

8.1 テスト自動化の改善オプション

一般的に、TASを改善する機会は、TASとSUTを同期させるために必要となる継続的な保守作業以外にも多く存在する。TASの改善は、効率性の向上（手動による介入をさらに削減する）、使いやすさの向上、機能の追加やテスト活動のサポートの向上など、さまざまなメリットを実現するために行われる場合がある。TASをどのように改善するかの決断は、プロジェクトに最大の価値をどのようにもたらすかを左右する。

改善が考えられるTASの具体的な領域には、スクリプティング、検証、アーキテクチャ、事前および事後処理、文書化、支援ツールなどがある。これらについて、以降で詳細に説明する。

スクリプティング
3.2.2節に記載されているように、スクリプティングアプローチは単純な構造化アプローチからデータ駆動アプローチ、さらに洗練されたキーワード駆動アプローチまでさまざまである。すべての新しい自動テストのために、現在のTASのスクリプティングアプローチを変更・向上させることが適切な場合もある。このアプローチでは、既存の自動テストがすべて更新される場合もあり、その場合かなりの量の保守作業が発生する。

TASの改善では、スクリプティングアプローチをすべて変更するのではなく、スクリプトの実装に着目する場合もあり、次に例を示す。

- 自動テストを統合する作業の中で、自動テストケース/ステップ/手順の重複を評価する。
  同じ操作手順を含むテストケースでは、そのステップを複数回実装しない。このようなステップは再利用できるように関数化し、ライブラリに追加する。ライブラリ関数は、異なるテストケースで使用できるようにする。これにより、テストウェアの保守性が向上する。テストのステップが同一でなく似ている場合は、パラメータ化が必要になる場合がある。
  注: これは、キーワード駆動テストにおける典型的なアプローチである。
- TASとSUTのエラー回復プロセスを確立する。
  テストケースの実行中にエラーが発生した場合、TASはこのエラー条件から回復し、次のテストケースを継続できるようにする。SUTでエラーが発生した場合、TASはSUTに対して必要な回復操作を可能にする必要がある（SUTを完全に再起動するなど）。
- スクリプト実行の待機（Wait）のメカニズムを評価し、最適なタイプが使用されるようにする。
  一般的な待機メカニズムは 3 つある。
  1. ハードコードによる待機は（一定のミリ秒だけ待機する）、さまざまなテスト自動化の根本な問題となる可能性がある。
  2. ポーリングによる動的な待機は（所定の状態変化や行われる操作を確認する）、それよりもはるかに柔軟で効率的である。
     - 必要な時間だけ待機するため、テスト時間が無駄にならない。
     - 何らかの理由でプロセスに長い時間がかかる場合、ポーリング条件が true になるまで待機を続ける。タイムアウトのメカニズムを含めておかないと、問題が発生した場合にテストが永久に待機を続けることになる。
  3. さらに優れているものは、SUTのイベントメカニズムを監視する方式である。これは他の 2 つの選択肢よりも信頼性が高い。それらはテストスクリプトの言語でイベントの監視がサポートされており、SUTからテストアプリケーションに対してイベントが発行されなければならない。タイムアウトのメカニズムを必ず含めておかないと、問題が発生した場合にテストが永久に待機を続けることになる。

- テストウェアをソフトウェアとして扱う。
  テストウェアの開発と保守は、ソフトウェア開発の 1つの形態である。そのため、優れたコーディング慣習（コーディングガイドラインの使用、静的解析、コードレビューなど）を適用する必要がある。ライブラリなど、テストウェアの特定部分の開発を、テストエンジニアではなくソフトウェア開発担当者が行うことも優れたアイデアといえる。
- 既存のスクリプトの改訂/削除について評価する。
  スクリプトの一部には、よく失敗する、保守コストが高いなど、問題を起こす可能性のあるものがある。そのようなスクリプトは再設計する方が望ましい場合がある。付加価値のなくなった一部のテストスクリプトはスイートから削除できる。

テスト実行
自動回帰テストスイートが夜間のうちに終わらないことは例外的なことではない。テストに時間がかかりすぎる場合、別のシステムで同時にテストを行うべきであるが、これは常に可能とは限らない。テストに高価なシステム（ターゲット）が使われている場合、すべてのテストを 1 つのターゲットで行わなければならないという制約が生じることがある。回帰テストスイートを複数の部分に分割し、それぞれを定義した時間（ある日の夜など）に実行しなければならない場合がある。自動テストのカバレッジをさらに分析すると、重複が明らかになる場合がある。重複を削除すれば、実行時間を削減でき、効率性が上昇する可能性がある。

検証
新しい検証機能を作成する前に、すべての自動テストで使用でき一連の検証ができる標準的な手法を採用する。これにより、複数のテストをまたいだ検証アクションの再実装を避けることができる。検証手法が同一でなく似ている場合は、パラメーター化することで複数の種類のオブジェクトで機能を使用できるようになる。

アーキテクチャ
SUTの試験性を向上させるために、アーキテクチャの変更が必要になる場合がある。このような変更は、SUTのアーキテクチャや自動化のアーキテクチャに対して行われる可能性がある。これはテスト自動化の大きな改善となるが、SUT/TASに大幅な変更やそれに対しての投資が必要になる場合がある。例えば、テスト用の APIを提供するためにSUTを変更する場合、TASも適切にリファクタリングされる。後工程の段階でこのような種類のフィーチャーを追加すると、非常に高価になる可能性がある。この点は自動化の初期段階で（SUT開発の初期段階 – 2.3節の「試験性と自動化を考慮した設計」を参照）で考慮する方がはるかに望ましい。

事前および事後処理
標準のセットアップおよび削除タスクを提供する。これらは、事前処理（セットアップ）および事後処理（削除）とも呼ばれる。これにより、自動テストごとにタスクを繰り返し実装する手間を省くことができ、保守コストだけでなく新しい自動テストを実装するために必要な工数も削減できる。

文書化
これは、スクリプトの文書化（スクリプトが何をするか、どのように使うべきかなど）、TASのユーザー文書、TASが生成するレポートやログなど、あらゆる形態の文書が対象である。

TASのフィーチャー
詳細レポート、ログ、他のシステムとの統合などのTASのフィーチャーを追加する。新フィーチャーは実際に使用される場合にのみ追加する。使用しないフィーチャーを追加しても、複雑さが増して信頼性と保守性が低下するだけである。

TASの更新とアップグレード
新しいバージョンのTASに更新またはアップグレードすると、テストケースで使用できる新機能が提供される場合がある(または、故障が修正される場合がある)。そのリスクとしては、フレームワークを更新する(既存のテストツールをアップグレードする場合でも、新しいものを導入する場合でも)ことで、既存のテストケースに悪影響が生じる可能性がある。新しいバージョンを展開する前に、サンプルのテストを実行して新しいバージョンのテストツールをテストする。サンプルのテストとは、異なるアプリケーション、異なるテストタイプ、そして適切なさまざまな異なる環境での自動テストを代表するものである。

8.2 テスト自動化改善の実装計画

既存のTASの変更には、慎重な計画と調査が要求される。TAFとコンポーネントライブラリで構成される堅牢なTASの作成には、多大な作業が必要となる。どれほど些細なものであろうと、変更はすべてTASの信頼性や性能に幅広く影響する可能性がある。

テスト環境のコンポーネントの変更点を特定する
どのような変更や改善を行う必要があるかを確認する。テストソフトウェア、カスタマイズした機能ライブラリ、OSに変更が必要になるかを確認する。このそれぞれが、TASの動作方法に影響する。全般的には、自動テストが確実かつ効率的に動作し続けることを目標とする。変更は、テストスクリプトを限定的に実行してTASへの影響を計測できるように段階的に行う。悪影響がないことがわかったら、変更を完全に実装することができる。完全な回帰実行は、変更が自動スクリプトに悪影響を与えないことを検証する最終段階となる。この回帰スクリプトの実行の際に、エラーが発見される場合がある。レポート、ログ、データ分析などによってこのエラーの根本原因を特定すれば、エラーが自動化の改善活動によるものではないことを保証する。

TASのコア機能ライブラリの効率性と有効性を改善する
TASが成熟するにつれて、タスクをさらに効率的に実行する新たな方法が発見される。このような新しい技法(機能のコードの最適化、新しいオペレーティングシステムライブラリの使用など)は、現在のプロジェクトやすべてのプロジェクトで使用されるコア機能ライブラリに組み込む必要がある。

同じ制御タイプで動作する複数の機能を統合の対象にする
自動テスト実行の際に起きることの大部分は、GUIのコントロールの調査である。この調査は、コントロールに関する情報(表示/非表示、有効/無効、サイズ、データなど)を提供する際に役立つ。自動テストはこの情報を利用してドロップダウンリストから項目を選択したり、フィールドにデータを入力したり、フィールドから値を読み取ったりすることができる。一部の機能では、コントロールに対して操作を行い、この情報を取り出すことができる。機能の中には、極端に細分化されたものも、本質的に汎用的なものもある。例えば、ドロップダウンリストに対してのみ動作するものは、細分化された機能である。また、パラメーターの1つとして関数を指定することで、いくつかの機能と連動して動作する機能も存在する(またはTASの内部で作成され、使用される)。そのため、TAEが使用する一部の機能は、その数を統合によって削減できる場合がある。これにより、同じ結果を実現しつつ、保守要件を最小化することができる。

TAAをリファクタリングしてSUTの変更に対応する
TASの耐用期間を通して、SUTの変更に対応するための変更が必要になる。SUTが進化および成熟するにつれて、その基盤となるTAAも進化してSUTを支える機能を提供しなければならない。フィーチャーを拡張する場合は、実装を追加していくのではなく、自動化ソリューションのアーキテクチャレベルで分析および変更を行うように注意しなければならない。これにより、新しいSUT機能で追加スクリプトが必要になる場合、互換コンポーネントによって新しい自動テストに対応できるようになる。

命名規約と標準化

変更が行われる場合、新しい自動コードや機能ライブラリの命名規約は、以前に定義された標準（4.3.2節の「スコープとアプローチ」を参照）と一貫性を持たせる必要がある。

SUTの改訂/削除のための既存のスクリプトの評価

変更や改善のプロセスには、既存のスクリプトの使用方法、継続的な価値の評価も含まれる。例えば、あるテストが複雑で実行に時間がかかる場合、それを複数の小さなテストに分割する方が現実的かつ効率的になることがある。頻繁に実行されないテストやまったく実行されないテストを削除対象とすると、TASの複雑度を低減させ、保守が必要なものを明らかにすることができる。

9 参考文献

9.1 標準

テスト自動化の標準には、以下のものが含まれるが、これに限定されるものではない。

- ETSI（European Telecommunication Standards Institute）および ITU（International Telecommunication Union）による The Testing and Test Control Notation（TTCN-3）は、以下から構成される。
  - ES 201 873-1: TTCN-3 Core Language
  - ES 201 873-2: TTCN-3 Tabular Presentation Format（TFT）
  - ES 201 873-3: TTCN-3 Graphical Presentation Format（GFT）
  - ES 201 873-4: TTCN-3 Operational Semantics
  - ES 201 873-5: TTCN-3 Runtime Interface（TRI）
  - ES 201 873-6: TTCN-3 Control Interface（TCI）
  - ES 201 873-7: Using ASN.1 with TTCN-3
  - ES 201 873-8: Using IDL with TTCN-3
  - ES 201 873-9: Using XML with TTCN-3
  - ES 201 873-10: TTCN-3 Documentation
  - ES 202 781: Extensions: Configuration and Deployment Support
  - ES 202 782: Extensions: TTCN-3 Performance and Real-Time Testing
  - ES 202 784: Extensions: Advanced Parameterization
  - ES 202 785: Extensions: Behaviour Types
  - ES 202 786: Extensions: Support of interfaces with continuous signals
  - ES 202 789: Extensions: Extended TRI
- IEEE（Institute of Electrical and Electronics Engineers）による The Automatic Test Markup Language（ATML）は、以下から構成される。
  - IEEE Std 1671.1: Test Description
  - IEEE Std 1671.2: Instrument Description
  - IEEE Std 1671.3: UUT Description
  - IEEE Std 1671.4: Test Configuration Description
  - IEEE Std 1671.5: Test Adaptor Description
  - IEEE Std 1671.6: Test Station Description
  - IEEE Std 1641: Signal and Test Definition
  - IEEE Std 1636.1: Test Results
- The ISO/IEC/IEEE 29119-3:
- OMG（Object Management Group）による The UML Testing Profile（UTP）は、以下のテスト仕様の概念を記している。
  - テストアーキテクチャ
  - テストデータ
  - テストの振る舞い
  - テスト結果記録作業
  - テストマネジメント

9.2 ISTQBドキュメント

ID	参考資料
ISTQB-AL-TM	ISTQB Certified Tester, Advanced Level Syllabus, Test Manager, Version 2012、[ISTQB-Web]から入手可能
ISTQB-AL-TTA	ISTQB Certified Tester, Advanced Level Syllabus, Technical Test Analyst, Version 2012、[ISTQB-Web]から入手可能
ISTQB-EL-CEP	ISTQB Advanced Level Certification Extension、[ISTQB-Web]から入手可能
ISTQB-EL-Modules	ISTQB Advanced Level Modules Overview, Version 1.2, August 23, 2013、[ISTQB-Web]から入手可能
ISTQB-EL-TM	ISTQB Advanced Level – Test Management syllabus, Version 2011、[ISTQB-Web]から入手可能
ISTQB-FL	ISTQB Foundation Level Syllabus, Version 2011、[ISTQB-Web]から入手可能
ISTQB-Glossary	ISTQB Glossary of terms, Version 2.4, July 4, 2014、[ISTQB-Web]から入手可能

9.3 商標

本資料内で使用する登録商標およびサービスマークは次のとおりである。

ISTQB® は、International Software Testing Qualifications Board の登録商標である。

9.4 書籍

ID	参考文献
[Baker08]	Paul Baker, Zhen Ru Dai, Jens Grabowski and Ina Schieferdecker, “Model-Driven Testing: Using the UML Testing Profile”, Springer 2008 edition, ISBN-10: 3540725628, ISBN-13: 978-3540725626
[Dustin09]	Efriede Dustin, Thom Garrett, Bernie Gauf, “Implementing Automated Software Testing: how to save time and lower costs while raising quality”, Addison-Wesley, 2009, ISBN 0-321-58051-6
[Dustin99]	Efriede Dustin, Jeff Rashka, John Paul, “Automated Software Testing: introduction, management, and performance”, Addison-Wesley, 1999, ISBN-10: 0201432870, ISBN-13: 9780201432879 (日本語訳) 自動ソフトウェアテスト—導入から、管理・実践まで-効果的な自動テスト環境の構築を目指して, ピアソンエデュケーション, 2002
[Fewster&Graham12]	Mark Fewster, Dorothy Graham, “Experiences of Test Automation: Case Studies of Software Test Automation”, Addison-Wesley, 2012
[Fewster&Graham99]	Mark Fewster, Dorothy Graham, “Software Test Automation: Effective use of test execution tools”, ACM Press Books, 1999, ISBN-10:

0201331403, ISBN-13: 9780201331400
（日本語訳）システムテスト自動化 標準ガイド, 翔泳社、2014

［McCaffrey06］ James D. McCaffrey, “.NET Test Automation Recipes: A Problem-Solution Approach”, APRESS, 2006 ISBN-13:978-1-59059-663-3, ISBN-10:1-59059-663-3

［Mosley02］ Daniel J. Mosley, Bruce A. Posey, “Just Enough Software Test Automation”, Prentice Hall, 2002, ISBN-10: 0130084689, ISBN-13: 9780130084682

［Willcock11］ Colin Willcock, Thomas Deiß, Stephan Tobies and Stefan Keil, “An Introduction to TTCN-3” Wiley, 2nd edition 2011, ISBN-10: 0470663065, ISBN-13: 978-0470663066

9.5 Web 参考資料

ID	参考資料
ISTQB-Web	International Software Testing Qualifications Board の Web サイト。最新の ISTQB 用語集およびシラバスについては ISTQB の Web サイト（www.istqb.org）が参照可能。日本語版については、JSTQB の Web サイト（jstqb.jp）が参照可能。

10 教育 機関への通知

10.1 教育の時間

本シラバスの各章には、分単位で時間が割り当てられている。これは、認定コースで各節に割り当てる時間の相対的な比率と、各節を教育する際にふさわしい最小時間の両方を示すことを目的としたものである。

教育機関では、指定より長い時間がかかっても構わない。また、受験希望者は、読み直しや調査にさらに時間を費やしても構わない。コースのカリキュラムは、本シラバスの順序に従う必要はない。またコースを 1 回で完了する必要はない。

次の表は、各章の座学と演習の時間の目安である(時間はすべて分単位)。

章	分
0.イントロダクション	0
1.テスト自動化の概要と目的	30
2.テスト自動化の準備	165
3.汎用テスト自動化アーキテクチャ	270
4.導入のリスクと不確実性	150
5. テスト自動化のレポートとメトリック	165
6.手動テストからの自動化環境への移行	120
7.TASの検証	120
8.継続的な改善	150
合計:	1170

1 営業日を平均 7 時間とした場合の合計コース時間:
2 日と 5 時間 30 分

10.2 実環境での演習

実環境で行うことができる演習は定義されていない。

10.3 e ラーニングのルール

本シラバスのすべての部分は、e ラーニングとして適切に実施できると考えられている。

11 索引