テスト技術者資格制度

Advanced Level Syllabus
アジャイルテクニカルテスト担当者

Version 1.1.J01

International Software Testing Qualifications Board

著作権について

改訂履歴

バージョン	日付	備考
シラバス v0.1	2017 年 1 月 11 日	個別セクション
シラバス v0.2	2017 年 5 月 24 日	v01 に対する WG レビューコメントの反映
シラバス v0.3		v02 に対する WG レビューコメントを反映
シラバス v0.7		v03 に対するアルファレビューコメントを反映
シラバス v0.71		ワーキンググループによる v07 のアップデート
シラバス v0.9	2017 年 12 月 30 日	アルファ版候補
シラバス v0.91	2018 年 1 月 6 日	アルファ版リリース
シラバス v0.98	2019 年 1 月 12 日	ベータ版候補
シラバス v0.99		ベータ版リリース
シラバス 2018		GA 版
シラバス 2019	2019 年 2 月 12 日	文章の軽微な変更 いくつかの LO を追加（K1、K2 レベル） 著作権情報の追加 全章に学習時間を追加
シラバス 2019	2019 年 3 月 14 日	公式ショートネーム「ATT」に合わせて LO を変更 「謝辞」を修正
シラバス 2019	2019 年 05 月 06 日	複数の軽微な修正
シラバス 2019	2019 年 06 月 07 日	軽微な修正
シラバス 2019	2019 年 07 月 08 日	改訂
シラバス 2019	2019 年 7 月 10 日	ベータ版レビュー提供
シラバス 2019	2019 年 9 月 14 日	ベータレビュー後の修正および追記
シラバス 2019 V1.0	2019 年 11 月 14 日	リリース
シラバス 2019 V1.1	2019 年 12 月 09 日	ISTQB のロゴマーク変更 K レベルのトレーニング期間を変更（Chapter 0.6）

JSTQB

バージョン	日付	備考
Version 1.1.J01	2023 年 7 月 9 日	Advanced Level Syllabus Agile Technical Tester（ATT） Version 1.1 の日本語翻訳版

目次

謝辞

このドキュメントは、Rex Black（chair）、Michaël Pilaeten（Vice-chair and chair a.i.）、Renzo Cerquozzi （Product Owner）が率いる International Software Testing Qualifications Board の Agile Working Group で作成された。

Advanced Level アジャイルチームは、レビューチームとメンバー委員会の提案や意見に感謝する。

Advanced Level アジャイルテクニカルテスト担当者のシラバスの完成時点における、ワーキンググループのメンバー：Michaël Pilaeten（Chair a.i.）、Renzo Cerquozzi（Product Owner）、Alon Linetzki（Marketing workgroup）、Leanne Howard（Glossary workgroup）、Klaus Skafte（Exam workgroup）。

著者：Leo van der Aalst、Renzo Cerquozzi、Bertrand Cornanguer、István Forgács、Jani Haukinen、Noam Kfir、Sammy Kolluru、Alon Linetzki、Tilo Linz、Michaël Pilaeten、Marie Walsh。

内部レビューアー：Michael Arefi、Vojtěch Barta、Renzo Cerquozzi、Graham Bath、Laurent Bouhier、Anders Claesson、Alessandro Collino、David Janota、David Evans、Leanne Howard、Matthias Hamburg、Kari Kakkonen、Tor Kjetil Moseng、Meile Posthuma、Salvatore Reale、Marko Rytkönen、Sarah Savoy、Klaus Skafte、Mike Smith、Chris van Bael。

サンプル問題の作成およびレビュー：Armin Born、Renzo Cerquozzi、Alon Linetzki、Tilo Linz、Jamie Mitchell、Jani Haukinen、Tobias Horn、Chris van Bael、Suruchi Varshney。

また、Foundation Agile Extension Syllabusのレビュー、コメント、投票に参加したメンバー委員会並びにやアジャイルエキスパートコミュニティの以下の方々にも感謝したい。Adam Roman、Armin Beer、Beata Karpinska、Chris Van Bael、Erwin Engelsma、Giancarlo Tomasig、Gary Mogyorodi、Ingvar Nordström、Jana Gierloff、Jörn Münzel、Jurian van de Laar、Kari Kakkonen、Laurent Bouhier、Marko Rytkönen、Martin Klonk、Matthias Hamburg、Meile Posthuma、Paul Weymouth、R.Green、Richard Seidl、Rik Marselis、Stephanie Ulrich、Stephanie van Dijck、Tal Pe'er、Tilo Linz、Veronica Seghieri、Wim Decoutere。

ISTQB® general secretary である Galit Zuckerのガイダンスとサポートに特に感謝したい。

このドキュメントは、2019 年 10 月 18 日に ISTQB®の総会で正式にリリースが承認された。

日本語訳については、Japan Software Qualifications testing 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：
浅黄 友隆（ヒューマンクレスト）
上村 真季（メドレー）
須原 秀敏（JSTQB）
長谷川 亮 （ベリサーブ）
早川 隆治（SCSK）
三浦 尚（ベリサーブ）
森 龍二（ベリサーブ）

0 イントロダクション

0.1 本シラバスの目的

本シラバスは、国際ソフトウェアテスト資格 Advanced Level アジャイルテクニカルテスト担当者のベースとなる。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、認定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 概要

Advanced Level アジャイルテクニカルテスト担当者 Overview のドキュメント[ISTQB_ATT_OVIEW]には、以下の情報が含まれる。

- シラバスのビジネス成果
- シラバスの概要
- シラバス間の関連性
- 知識レベル（K-レベル）の説明
- ソフトウェアやシステムのテストについての付録、および基礎資料としての書籍や論文

0.3 試験対象の学習の目的

学習の目的はビジネス成果を支援し、テスト技術者資格制度 Advanced Level アジャイルテクニカルテスト担当者の試験問題作成のために使用する。 全体を通して、本シラバスのすべての内容は、K1 レベルで試験対象である。つまり、受験志願者はキーワードと概念について認識し、記憶し、想起することになる。K2、K3、K4 レベルの学習の目的は、各章の先頭で示されている。

0.4 Advanced Level アジャイルテクニカルテスト担当者の認定試験

Advanced Level アジャイルテクニカルテスト担当者 の認定試験は、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を基にしている場合がある。本シラバスのすべての節は、

「イントロダクション」と付録を除いて試験対象である。標準、文献、および他の 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。

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よい。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。

0.5 認定審査

ISTQB®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。

0.6 本シラバスの構成

4 つの章で構成され、すべて試験対象である。

各章の一番上の見出しは、章の学習時間を指定している。章より下のレベルでは、時間は指定されていない。

認定トレーニングコースでは、本シラバスでは **16** 時間以上の講義を必要とし、以下のように各章に配分する。

- 第 1 章：180 分 要求工学
- 第 2 章：540 分 アジャイルにおけるテスト
- 第 3 章：135 分 テスト自動化
- 第 4 章：105 分 デプロイメントとデリバリー

それぞれの学習の目的をカバーするための最小時間が以下のように指定されている。
K2：15 分
K3：60 分
K4：75 分

1 要求工学 180 分

キーワード
受け入れ基準、エピック、ユーザーストーリー

「要求工学」の学習の目的

ATT-1.x （K1） キーワード

ATT-1.1.1-1 （K4）要求工学技法を用いて、ユーザーストーリーやエピックを分析することができる。

ATT-1.1.1-2 （K2）要求工学技法と、それがどのようにテスト担当者に役立つかを説明する。

ATT-1.1.2-1 （K4）要求工学とテスト技術を用いて、与えられたユーザーストーリーに対してテスト可能な受け入れ基準を作成し、評価する。

ATT-1.1.2-2 （K2）要求獲得（Elicitation）の技法について説明する。

1.1 要求工学技法

要求工学技法を適用することで、アジャイルチームはユーザーストーリー（『Foundation Level Extension シラバス アジャイルテスト担当者』[AgileFoundationExt]参照）やエピック（『Foundation Level Extension シラバス アジャイルテスト担当者』[AgileFoundationExt]参照）に磨きをかけ、コンテキストを加え、影響や依存関係を考慮し、非機能要求の欠落などのあらゆるギャップを特定することができる。
本節で取り上げた要求工学技法の大半は、従来の開発アプローチに由来するものであるが、それらはアジャイル開発においても有効である。
一般的に従来のプロジェクトでは、要求工学の活動や技法は形式化され、順次適用され、ビジネスアナリスト、ファンクショナルアナリスト、テクニカルアーキテクト、エンタープライズアーキテクト、プロセスアナリストなどのしかるべき人の責任となる。対照的にアジャイルプロジェクトでは、要求工学技法はあまり公式ではないアプローチによって、プロジェクト全体および各イテレーション中に適用される。これらの要求工学技法は、専任のビジネスアナリストやプロダクトオーナーだけでなく、すべてのアジャイルチームメンバーによって、継続的なフィードバックループを用いて、より頻繁に適用される。

1.1.1 要求工学技法を用いたユーザーストーリーやエピックの分析

テスト担当者として、ユーザーストーリーやエピック、その他のアジャイル要求の明確化（場合によっては改善）を支援できるようになるためには、これを支えるさまざまな要求工学技法を知り、理解し、選択し、使用する必要がある。
技法には、ストーリーボード、ストーリーマッピング、ペルソナ、ダイアグラム、ユースケースなどがある。

- **ストーリーボード**：ストーリーボード（アジャイルタスクボードやアジャイルユーザーストーリーボードと混同しないように）は、システムを視覚的に表現したものである。ストーリーボードはテスト担当者が以下のことをするのに役立つ。
  - ユーザーストーリーの背景にある思考プロセスや全体的な「ストーリー」を見て、コンテキストを提供することで、システムの機能的な流れを素早く確認し、ロジックとのギャップを特定することができる
  - システムの共通部分に関連するユーザーストーリーのグループ（テーマ）を可視化する。これらのユーザーストーリーは、コードの同じ部分を実行している可能性が高いため、同じイテレーションに含めてもよい
  - プロダクトバックログにある、エピックと関連するユーザーストーリーに対してストーリーマッピングを作ることと優先度付けを支援する
  - ユーザーストーリーやエピックの受け入れ基準の特定を支援する
  - システム設計で視覚的側面に基づいて、適切なテストアプローチの選択を支援する
  - ストーリーマッピングとともに、テストの優先順位付けや、スタブ、ドライバー、モックの必要性の検討を支援する

- **ストーリーマッピング**：ストーリーマッピング（またはユーザーストーリーマッピング）は、2 つの独立した次元を使用してユーザーストーリーを並べることで構成される技法である。マップの横軸は各ユーザーストーリーの優先順位を表し、縦軸は実装の複雑さを表す。ストーリーマッピングを使うことで、テスト担当者は以下のことができる。
  - スモークテストを抽出するために、システムの最も基本的な機能を決定する

- テストの優先順位を決めるために、機能の順序づけを行う
- システムのスコープを可視化する
- 各ユーザーストーリーのリスクレベルを決定する

- **ペルソナ**：ペルソナは、典型的なユーザーがどのようにシステムを操作するかを示す、架空のキャラクターや代表例を定義するために使用される。ペルソナを使用することで、テスト担当者は以下のことができる。
  - システムを使用する異なるタイプのユーザーを特定することで、ユーザーストーリーのギャップを識別する
  - 他のタイプのユーザーと比較して、特定のタイプのユーザーがどのようにシステムを使用するかに基づいて、ユーザーストーリー間の不整合を特定する
  - ユーザーストーリーの受け入れ基準を導き出す
  - 探索的テスト中に追加のテストすべき経路（test path）を見いだす
  - テスト条件、特に特定のユーザーグループに関連する条件を明らかにすることで、ユーザーグループに対して十分な網羅性を確保し、ユーザーグループ間の違いをテストすることができる

- **ダイアグラム**：ER 図や、クラス図などの UML ダイアグラムは、データの構造や流れ、システムの機能的属性やふるまいを示すことができ、システム機能のギャップの特定に使用することができる。

- **ユースケース：**ユースケース（ユースケース図とユースケース記述）『Foundation Level シラバス』[Foundation]は、テスト担当者が以下のことをするのに役立つ。
  - ユーザーストーリーがテスト可能かつ適切なサイズであることを確実にする
  - ユーザーストーリーを改良、または分解する必要があるのかを決定する
  - 忘れられたステークホルダーを明らかにする
  - テスト設計時に考慮すべきインターフェースと統合ポイントを特定する
  - エピックとユーザーストーリーの関係から、エピックに欠けているユーザーストーリーがないかどうかを確認する

1.1.2 要求工学とテスト技法を用いた受け入れ基準の特定

要求工学は、次のようなステップで構成されるプロセスである。

- **要求獲得（Elicitation）：**システムに対するユーザーのニーズと制約を発見し、理解し、文書化し、検討するプロセス。受け入れ基準を導き出し、評価し、強化するために、要求獲得の技法を用いるべきである。
- **文書化（Documentation）：**ユーザーのニーズと制約を明確かつ正確に文書化するプロセス。ユーザーストーリーと受け入れ基準は、アジャイルマニフェストの原則に対するチームの忠実さに応じたレベルで文書化する必要がある。文書の種類は、チームやステークホルダーのアプローチによって異なる。受け入れ基準は、自然言語や、モデル（状態遷移図など）、実例を用いて文書化することができる。
- **交渉と検証（Negotiation and Validation）：**各ユーザーストーリーに対して、複数のステークホルダーが、別の見解や好みを持っている場合がある。これらの見解や好みに一貫性が

ないばかりか相反することもあるので、各ステークホルダーの受け入れ基準も同じようになる可能性がある。これらの相反は、影響を受けるすべてのステークホルダーの間で、特定・交渉・解決する必要がある。忘れられたり、解決されないままの相反は、プロジェクトの成功を危うくする。このステップの最後に、各ユーザーストーリーの内容は、影響を受けるステークホルダーによって検証される（例：準備完了（Ready）の定義）。

- **マネジメント（Management）：**プロジェクトの進行に伴い、意見や状況が変化することがある。受け入れ基準を適切に獲得し、文書化し、交渉し、検証したとしても、受け入れ基準は変更される可能性がある。変更の可能性があるため、適切な構成管理、および変更管理プロセスを用いてユーザーストーリーを管理する必要がある。

受け入れ基準を特定するために、テスト担当者が利用できる複数の要求獲得の技法を以下に挙げる。

- **定量的アンケート：クローズドクエスチョン**から得られる定量的なデータを利用することは、様々なデータの間で明確な比較を行う優れた方法である。これにより、多くの場合、受け入れ基準の数値的根拠になるデータが得られる。定量的なアンケートは、多数のステークホルダーのための、特に非機能面の受け入れ基準のための要求獲得の技法として使用することができる。
- **定性的アンケート：オープンクエスチョン**は、定量的な調査に加えて、さらに質を高めるための非常に効果的な方法である。オープンクエスチョンは、重要な質問のフォローアップとして使用することが最適である。これにより、新しいユーザーストーリーを作成したり、既存のユーザーストーリーに追加したりするための情報を得ることができる。定性的アンケートは、処理に時間がかかるため、より少数のステークホルダーに対する要求獲得の技法として使用することができ、機能面の受け入れ基準に適している。
- **定性的インタビュー：**定性的インタビューは、定量的な質問よりも柔軟性が高く、主に背景、コンテキスト、原因に関する情報を得るために使用される。確かなデータが得られる可能性は低いが、ユーザーストーリーのコンテキストに関する回答から受け入れ基準を導き出すことができる。定性的インタビューは、導き出された受け入れ基準を深めるために、両方のタイプのアンケートの補完となりうる。

観察技法（例：見習い）、創造性技法（例：6 つの帽子思考法）、支援技法（例：ローファイプロトタイピング（low-fi prototyping））など、他にも様々な要求獲得の技法がある。テスト担当者が持つ技法のツールセットは、獲得した受け入れ基準の品質に影響を与える。

INVEST および SMART[INVEST]や、同値分割法、境界値分析、デシジョンテーブル、状態遷移テスト『Foundation Level シラバス』[Foundation]などのテスト技法は、同様に受け入れ基準の特定と評価に使用できる。

2 アジャイルにおけるテスト 540 分

キーワード

テスト駆動開発（Test-driven development（TDD）），振る舞い駆動開発（behavior-driven development（BDD）），受け入れテスト駆動開発（acceptance test-driven development（ATDD）），実例による仕様（specification by example（SBE）），テストチャーター（test charter）

「アジャイルにおけるテスト」の学習の目的

ATT-2.x （K1） キーワード

2.1 アジャイルソフトウェア開発とテスト技法

ATT-2.1.1-1 （K3）アジャイルプロジェクトにおいて、与えられた実例のコンテキストでテスト駆動開発（TDD）を適用する。

ATT-2.1.1-2 （K2）ユニットテストの特徴を理解する。

ATT-2.1.1-3 （K2）「FIRST」という語呂合わせの意味を理解する。

ATT-2.1.2-1 （K3）アジャイルプロジェクトにおいて、与えられたユーザーストーリーのコンテキストで振る舞い駆動開発（BDD）を適用する。

ATT-2.1.2-2 （K2） シナリオ策定のためのガイドラインの管理方法を理解する。

ATT-2.1.3 （K4）アジャイルプロジェクトのプロダクトバックログを分析し、受け入れテスト駆動開発（ATDD）を導入する方法を決定する。

2.2 アジャイルにおける経験ベースのテスト

ATT-2.2.1-1 （K4）テスト自動化、経験ベースのテスト、ブラックボックステストを使ったテストアプローチについて、もしくは、アジャイルプロジェクトにおいて、与えられたシナリオに対して（リスクベースドテストを含む）他の取り組み方法で作成されたテストアプローチについて分析する。

ATT-2.2.1-2 （K2）ミッションクリティカルとノンクリティカルの違いを説明する。

ATT-2.2.2-1 （K4）テストチャーターを作成するために、ユーザーストーリーとエピックを分析する。

ATT-2.2.2-2 （K2）経験ベースの技法の使い方を理解する。

2.3 コード品質の側面

ATT-2.3.1-1 （K2）アジャイルプロジェクトにおけるテストケースのリファクタリングの重要性を理解する。

ATT-2.3.1-2　（K2） テストケースのリファクタリングのための実践的なタスクリストを理解する。

ATT-2.3.2-1　（K4） 欠陥や技術的負債を特定するために、コードレビューの一環としてコードを分析する。

ATT-2.3.2-2　（K2）静的コード解析を理解する。

2.1 アジャイルソフトウェア開発とテスト技法

ソフトウェア開発では、書き方のまずいコードや、顧客ニーズの獲得失敗により、欠陥が発生する可能性がある。これらの問題を解決するために、アジャイル開発手法では、テスト駆動開発（TDD）、振る舞い駆動開発（BDD）、受け入れテスト駆動開発（ATDD）の考え方を取り入れている。TDD はソフトウェアの製品品質を向上させるための手法であり、BDD や ATDD はソフトウェアの利用時品質（フィーチャー・機能）を向上させるための手法である。

2.1.1 テスト駆動開発（TDD）

テスト駆動開発とは、設計、テスト、コーディングを迅速な反復プロセスの中で組み合わせて行うソフトウェア開発手法である。TDD は機能をコーディングする前に、そのコードの意図する機能を表現するテストを作成するという、厳格なテストファーストのアプローチを採用している。テストはコードが書かれる前に実行される。それはテストが失敗することを確認するためである。一旦コードが書かれると、テストは再び実行され、合格するはずである。

TDD は一般的に、最初の主要なテストファーストプログラミング方法論と考えられており、そこから振る舞い駆動開発（BDD）、受け入れテスト駆動開発（ATDD）、実例による仕様（SBE）などの他の方法論が派生している。

TDD は、ソフトウェア開発において、設計を継続的な進行中のプロセスとして扱う進化的アプローチを提供する。各イテレーションは、プロダクションコードへの小さな変更を行い、開発者が設計をわずかに改善する機会を提供する。この変更と慎重に検討された設計上の決定が一歩一歩積み重なることで、堅牢で十分にテストされた設計が生まれる。

TDD の実践者は、高品質なコードを書き、技術的負債の蓄積を避けるために、以下のような数多くのプラクティスと技法を用いている。

- ユニットテストと、あらかじめ決められたテストファーストアプローチ
- 短いイテレーションサイクル
- 即時かつ頻繁なフィードバック
- ツール、構成管理、継続的インテグレーションの有効活用
- プログラミングの原則やデザインパターンのガイドとしての適用

TDD の実践者は、プロダクトコードの期待される動作をチェックするためにユニットテストを書く。ユニットテストは、特定の条件下で関数（function）を実行し、期待結果が得られるかどうかをチェックする。期待結果は、システムの状態が期待通りに変化するかどうか、あるいは特定の動作が現れるかどうかをチェックするアサーションで記述される。アサーションは、例えば以下のようなことをチェックできる。

- 関数（function）が計算を行い、期待通りの結果を返すこと
- 関数（function）がシステムの状態を特定の方法で変更すること
- 関数（function）が別の関数（function）を特定の方法で呼び出すこと

ユニットテストは、開発者が簡単に実装・保守できるものでなければならない。ユニットテストは自動化されており、多くの場合、プロダクトコードと同じ言語で書かれている。ユニットテストは以下のような特徴を持つべきである。

- 決定論的（Deterministic）：ユニットテストを同じ条件で実行すると、毎回同じ結果が得られるはずである。
- アトミック（Atomic）：ユニットテストは、そのユニットテストに関連する機能のみをテストする必要がある。
- 独立（Isolated）：ユニットテストは、最初に意図された特定のコードのみを実行するように努めなければならない。ユニットテストは相互に依存してはならず、プロダクトコードへの依存を可能な限り避けるべきである。
- 高速（Fast）：ユニットテストは、すぐにフィードバックを提供できるように、小さくて速いものでなければならない。短時間で多くのユニットテストを実行できるようにする必要がある。

優れたユニットテストを表すもうひとつの語呂合わせ（mnemonic）は、FIRST: 高速（Fast）, 独立（Isolated）, 繰り返し可能（Repeatable）, 自律的検証（Self-Validating）, 完全（Thorough） である。

様々な言語に対応したユニットテストフレームワークが多数存在する。それらは、API、アプローチ、用語などが異なるが、すべてに共通する要素がある。言語やユニットテストフレームワークにかかわらず、ユニットテストは通常、次の 3 つのステップのパターンに従う。

1. 準備（Arrange/Setup）：実行環境を準備する。このステップでは、必要に応じてオブジェクトをインスタンス化し、システムの状態を初期化し、データをセットする。
2. 実行（Act）：テストを実行し、その結果を確認する。この重要なステップは、テスト対象の操作を起動するためのたった 1 行のコードで構成されることもある。
3. アサート（Assert ）：期待結果やその他の事後条件を実際にチェックする。

イテレーティブなプロセスは TDD の基礎となるものである。各イテレーションの目的は、プログラムに小さな、局所的な、熟慮された変更を加えることである。色分けの習慣に従い、イテレーティブサイクルはしばしば「レッド／グリーン／リファクタリング」サイクルと呼ばれる。

- レッド（Red）：未実装の期待値を記述して、テストを失敗させる。テストを実行して、失敗することを確認する。
- グリーン（Green）：テストによって記述された期待値のみを満たすプロダクトコードを書き、それが合格するようにする。関連するすべてのテストを実行して、それらがすべて合格することを確認する。失敗した場合は、そのテストがすべて合格するように必要な変更を行う。
- リファクタリング（Refactor）：機能性を変更することなく、テストコードとプロダクトコードの両方の設計と構造を改善するとともに、関連するすべてのテストが引き続き合格することを保証する。開発者は、コードが最適化されるまで、一通りのリファクタリングを適用して、振る舞いを変えずにコードを変更することができる。最近の統合開発環境（IDE）や多くのコードエディタでは、自動的かつ安全にリファクタリングを適用することができる。

2.1.2 振る舞い駆動開発（BDD）

ISTQB® Agile Foundation シラバスによると、振る舞い駆動開発（BDD）とは、開発者、テスト担当者、ビジネス代表者が協力して、ソフトウェアシステムの要件を分析し、共通の言語を使って要件を策定し、自動的に検証する手法である。

BDD は、テスト駆動開発（TDD）とドメイン駆動設計（DDD）という 2 つの異なる方法論から強い影響を受けている[Evans03]。BDD は、コラボレーション、デザイン、ユビキタス言語、自動テスト

実行、短いフィードバックサイクルなど、2 つの方法論の中核となる原則の多くを取り入れている。TDD は実装の詳細を検証するためにユニットテストに依存しているのに対し、BDD は振る舞いを検証するために実行可能なシナリオなどに依存している。BDD は、大抵は規定された手順に従う。

- 共同でユーザーストーリーを作成する。
- ユーザーストーリーを実行可能なシナリオと検証可能な振る舞いとして策定する。
- 振る舞いを実装し、それを検証するためにシナリオを実行する。

BDD を適用するチームは、各ユーザーストーリーから 1 つ以上のシナリオを抽出し、それを自動テストにする。シナリオとは、特定の条件下でのひとつの振る舞いを表す。
シナリオは通常、ユーザーストーリーの受け入れ基準、例、ユースケースから成る。BDD シナリオは、技術者、非技術者を問わず、すべてのチームメンバーが理解できるような言語を使って書く必要がある。
そのため、シナリオの表現には英語などの自然言語を使うことが強く推奨されている。さらに、BDD を適用するチームは、ユビキタス言語[Evans03]を定める。これは、基本的にチームの誰にとっても明確で曖昧さのない用語で構成され、あらゆる箇所で使用される。

BDD シナリオは通常、3 つの主要セクションで構成されている[Gherkin]。

1. Given：振る舞いが引き起こされる前の環境の状態（事前条件）を記述する
2. When：振る舞いを引き起こすアクションを記述する
3. Then：振る舞いの期待結果を記述する

ユーザーストーリーからシナリオを抽出するには以下のように行う。

- ユーザーストーリーで指定されたすべての受け入れ基準を特定し、基準ごとにシナリオを書く。複数のシナリオを必要とする場合もある。
- 機能的なユースケースや例を特定し、それぞれのシナリオを書く。多くのユースケースや例は条件付きのため、それぞれの条件を特定することが重要である。
- 探索的テスト中にシナリオを作成することは、関連する既存の振る舞い、潜在的な矛盾する振る舞い、最小限の状況、代替フローなどを特定する助けになる。
- 反復作業を避けるために、繰り返し使っているステップを探す。
- ランダムデータや合成データを必要とする領域を特定する。
- モック、スタブ、ドライバーを必要とするステップを特定することで、独立性を保ったまま、環境の統合やコストのかかるプロセスを回避することができる。
- シナリオがアトミックであり、お互いの状態に影響を与えない（独立している）ことを確認する。
- テストは 1 つの期待結果をチェックするという原則に基づき、When セクションを 1 つのステップに制限することを決定する。または、テスト実行速度など他の考慮事項に合わせて最適化する。

シナリオを策定するために推奨されるガイドラインは以下の通りである。

- シナリオは、特定のユーザーの視点から、システムがサポートする特定の振る舞いを記述する必要がある。
- シナリオでは、ステップ（Given、When、Then）を記述する際に三人称を使い、ユーザーの視点から状態やインタラクションを表現しなければならない。

- シナリオは独立かつアトミックなものとし、どのような順序でも実行でき、お互いに影響したり依存したりしないようにする。Given のステップは、When のステップが期待通りに一貫して実行されるために必要な状態にシステムを設定する。
- 特定のアクションをテストする特別な必要性がない限り、When ステップは、特定の技術的なアクションではなく、ユーザーが実行するような意味のあるアクションを記述する必要がある。例えば、ボタン自体をテストする必要がない限り、「ユーザーはオーダーを確定する」（意味のあるアクション）の方が「ユーザーが確定ボタンをクリックする」（技術的なアクション）よりも一般的に好ましい。
- Then のステップでは、特定の観察結果や状態を記述する必要がある。一般的な成功やエラーの状態を指定しない。

2.1.3 受け入れテスト駆動開発（ATDD）

受け入れテスト駆動開発（ATDD）は、顧客のニーズに基づいたデリバリーに対応するためにソフトウェアプロジェクトの調整をサポートするものである。受け入れテストは、システムに求められる振る舞いや機能を仕様化したものである。ユーザーストーリーは、顧客にとって価値のある機能の一部を表している。受け入れテストでは、この機能が正しく実装されているかどうかを検証する。このユーザーストーリーは、開発者によって、その機能を実装するために必要な一連のタスクに分割される。開発者はこれらのタスクを TDD で実装する。あるタスクが完了すると、開発者は次のタスクに移り、ユーザーストーリーが完了するまで、受け入れテストが正常に実行されたかどうかを確認する。BDD と ATDD はどちらも顧客に焦点をあてており、TDD は開発者に焦点をあわせている。BDD と ATDD は、何を構築するか、どうすれば正しく構築できるかについての理解を共有するという点で、同じ結果を生むということが似ている。BDD は、前述の Given/When/Then 構文を使用してテストケース（例：受け入れテスト）を構造的に書く手法である。
ATDD では、プレーンテキストなどの人間が読める形式で表現されるテストの狙いと、テスト実装（これは自動化されていることが望ましい）とを分離している。この重要な分離は、プロダクトオーナー、アナリスト、テスト担当者など、ビジネス担当チームやその他の非技術的なチームのメンバーが、開発を推進するためのテスト可能な例（または受け入れテスト）の記述に積極的な役割を果たすことを意味する。
顧客、開発者、テスト担当者など、異なる役割や視点を持つステークホルダーが集まり、ユーザーストーリー（またはその他の形式の要求）の候補を共同で議論し、解決すべき問題について共通の理解を得て、機能に関するオープンな質問を互いに投げかけ、要求される振る舞いの具体的でテスト可能な例を検討する。

合意された例（およびそれに至るまでの議論）は、それ自体が価値のあるアウトプットであり、受け入れテストとして自動化することもできるが、そうすることが必ずしも必要ではなく、費用対効果も高くないことを念頭に置くことが重要である。このように、*テスト*ではなく価値のある*例*という考え方を強調するのは意図的なものであり、チームのすべてのメンバーに合意された例（およびそれに至るまでの議論）を促すための重要な仕掛けとなっている。

これは、この状況におけるアジャイルテスト担当者の重要な役割を示している。議論の間、テスト担当者は、同値分割法や境界値分析などのテスト技法の観点で考えているかもしれない。彼らは、興味深い振る舞いやエッジケースがどこにあるのかについてプロダクトオーナーに質問することを促すために、この分析的なアプローチを使うことができる。意図するところは、常にグループ全体で*重要な例*を検討し、発見を促すことである。興味深い入力の組み合わせを提示して、予想されるアウトプッ

トについてグループの意見が一致しているかどうかを尋ねることは、不確実性や不完全な分析の潜在的な領域を探るための良い方法である。

実例による仕様（SBE）とは、アジャイルチームがビジネスのステークホルダーが要求する重要な成果と、その成果に到達するために必要なソフトウェアの振る舞いを発見し、議論し、確認するための有用なパターン集のことである。この用語が広く使われてきた結果、受け入れテスト駆動開発（ATDD）の意味を SBE という用語が包含しつつ、ATDD という用語の意味を拡張してきている。

2.2 アジャイルにおける経験ベースのテスト

アジャイルプロジェクトのさまざまな特徴（アプローチ、イテレーションの長さ、適用可能なテストレベル、プロジェクトや製品のリスクレベル、要件の質、チームメンバーの経験・専門知識のレベル、プロジェクト組織など）は、アジャイルプロジェクトにおける自動テスト、探索的テスト、手動ブラックボックステストのバランスに影響を与える。

2.2.1 経験ベースの技法とブラックボックステストの組み合わせ

リスク分析では、個々のシステムのフィーチャーと機能について、リスクレベル（例：高、中、低）が決定される。次のステップは、特定のリスクレベルに対して、自動テスト、探索的テスト、手動のブラックボックステストの適切な組み合わせとバランスを見つけることである。以下は、この考え方を説明した表である。この表では、リスクレベルを縦に、3 つのテストアプローチを横に並べている。以下の記号を使って説明する。
++（強く推奨）
+（推奨）
o（ニュートラル）
-（非推奨）
--（使用しない）

以下の表は、セーフティクリティカルなシステムで使用できる、さまざまなテスト技法（自動テスト、探索的テスト、手動のブラックボックステスト）を組み合わせた例である。この表は、他の特定のプロジェクトにも適用できる。

リスクレベル	自動テスト	探索的テスト	ブラックボックステスト
高	++	+	++
中	+	+	+
低	o	++	+

表 1：ミッションクリティカル、セーフティクリティカルなシステム

表 1 の最初の行を見ると、このような状況では、探索的テストのアプローチに加えて、自動化されたテストとブラックボックステストを組み合わせることが強く推奨されることがわかる。
自動化するかしないかの判断は、他の多くの要因にも影響される。

以下は、非セーフティクリティカルなシステムに使用する場合の、異なるテスト技法の組み合わせの例である。

リスクレベル	自動テスト	探索的テスト	ブラックボックステスト
高	+	++	+
中	o	++	o
低	--	++	--

表 2：非ミッションクリティカル、非セーフティクリティカルなシステム

上記の表 2 の最後の行を見ると、このような状況では、探索的テストのアプローチが強く推奨され、他のアプローチは使用しないことを示唆している。
どのような状況でも（セーフティクリティカルかどうかに関わらず）、具体的なテスト技法の組み合わせは、プロジェクトの特性に依存する。

2.2.2 テストチャーターの作成とその結果の解釈

適切なテストチャーターを作成する前に、まず既存のエピックとユーザーストーリーを評価する必要がある。技法については第 1 章を参照のこと。

テストチャーターを作成するためにエピックやユーザーストーリーを分析する際には、以下の点を考慮する必要がある。

- このエピックやユーザーストーリーのユーザーは誰か？
- エピックやユーザーストーリーの主な機能は何か？
- ユーザーが実行できるアクションは何か？（ユーザーストーリーで定義される受け入れ基準リストから得られる。）
- ユーザーストーリーのゴールは、そのフィーチャーまたは機能が完了した時点で実現されているか？（つまり、完了（done）の定義に影響を与える他のテストタスクがあるか？）

テストチャーターの粒度は重要である。識別された問題の周辺（その問題への対処、リグレッション）や、ユーザーストーリーやエピックの周辺（周辺への影響の予見、潜在バグの顕在化）を調査する必要があるため、小さすぎてはいけない。

60 分から 120 分のタイムボックスに収まるように、テストチャーターはあまり大きなものにしてもいけない。探索的テストのセッションを実行する目的は、発生事象やバグの偏在する領域などに注意を払いながら、質の高い決定を下すのに役立てることである。探索的テストの結果は、その決定を下すのに十分な情報を与えてくれるはずである。

テストチャーターは、フリップチャート、スプレッドシート、ドキュメント、既存のテスト管理システム、ペルソナ、マインドマップ、チーム全体でのアプローチなどを用いて作成することができる。探索的テスト担当者は、探索的テストセッションの作成や実施において、創造性を発揮するためにヒューリスティックスを使用する。これらは、テストチャーターの作成や、ユーザーストーリーやエピ

ックを分析する際の創造的な思考にも使用できる。ヒューリスティックスの例は、[Whittaker09]や[Hendrickson13]に記載されている。

探索的テストで発見した全ての事象は、文書化する必要がある。探索的テストの結果は、より良いテスト設計のための洞察、製品をテストするためのアイデア、さらに改善するためのアイデアを提供しなければならない。探索的テストで文書化すべき事象には、欠陥、アイデア、質問、改善提案などがある。

探索的テストのセッションを文書化するために、ツールを使用することができる。これには、ビデオキャプチャおよびロギングツール、プランニングツールなどが含まれる。ドキュメントには、期待結果を含める必要がある。収集する情報の量によっては、ペンと紙で十分な場合もある。

探索的テストセッションを要約する場合、報告会では情報を収集して集計し、セッションの進捗、カバレッジ、効率性などの状況を提示する。この集計情報は、管理レポートとして使用したり、あらゆるレベル、あらゆる規模（単一のチーム、複数のチーム、大規模なアジャイル実装）のふりかえりで使用することができる。しかし、探索的テストセッションに関連する正確なテストメトリクスを決定することは非常に困難である。

2.3 コード品質の側面

アジャイルプロジェクトでは、技術的負債のコントロール、特にリリース全体を通して高レベルのコード品質を維持するという点が非常に重要である。この目標を達成するために、様々な技法が使われている。

2.3.1 リファクタリング

リファクタリングとは、振る舞いを変えずに既存のコードやテストケースの設計を明確にし、シンプルにすることで、効率的かつ制御された方法でコードをクリーンアップする方法である。アジャイルプロジェクトでは、イテレーションが短いため、チームメンバー全員に短いフィードバックループが発生する。また、短いイテレーションは、適切なカバレッジを達成しようとするテスト担当者にとってもチャレンジとなる。イテレーションの性質上、機能は成長し、時間の経過とともにフィーチャーが追加・強化されていくため、初期のイテレーションで書かれた機能テストは、後のイテレーションで保守や完全な再設計が必要になることがよくある。進化的なテスト設計のアプローチを使用して、テストの更新とリファクタリングを行うことで、フィーチャーの変更を取り入れ、プロダクトの機能に沿ったテストを維持することができる。

ユーザーストーリーが理解され、それぞれについて受け入れ基準が書かれた後、現在のイテレーションの機能が既存のリグレッションテスト（手動および自動）に与える影響が分析できる。その結果、テストのリファクタリングや強化が必要になる場合がある。チームは、イテレーションからイテレーションへと、広範囲にわたってコードを維持・拡張しているが、継続的なリファクタリングがなければ、これを行うことは困難である。

テストケースのリファクタリングは、以下のように行う。

- **識別**：レビューや原因分析により、リファクタリングが必要な既存のテストを特定する。
- **分析：**変更したテストがリグレッションテストセット全体に与える影響を分析する。

- **リファクタリング**：テストの内部構造の変更により、観測可能な振る舞いを変えずに理解しやすく、修正を簡単にする。
- **再実行**：テストを再実行し、その結果を確認し、必要に応じて欠陥を文書化する。リファクタリングは、テスト実行の結果に影響を与えてはならない。
- **評価**：再実行されたテストの結果を確認し、チームが定義して受け入れた品質基準にテストが合格した時点でこのフェーズを終了する。

2.3.2 欠陥や技術的負債を特定するコードレビューと静的コード解析

コードレビューとは、2 人以上の参加者（そのうちの 1 人は通常、開発担当者）によるコードの体系的な検査のことである。静的コード解析とは、ツールを使ったコードの体系的な検査である。どちらも、コードの品質に影響を与える問題点を明らかにし、それを特定するための効果的な手法であり、広く普及している。コードレビューと静的コード解析は、欠陥を特定し、技術的負債を管理するために役立つ建設的なフィードバックを提供する。

リソースの制約、予想以上の技術的な複雑さ、急速に変化する優先順位、技術的な制限などの障害は、質の高いコードを書く努力を妨げ、プログラマーに、すぐに得られる結果を優先してコードの質を下げるような妥協を強いることがある。このような妥協により、欠陥が発生し、技術的負債を抱えることになる。

技術的負債とは、劣っていても簡単に実装できるソリューションを選択した結果、将来、より良いソリューション（潜在的な欠陥の除去を含む）を実装するために必要となる労力が増加することを指す。技術的負債は、ソフトウェアの進化に伴うわずかな妥協や、小さな、あるいは気づかれないような変更の積み重ねによって、意図せずに発生することが多い。コードレビューや静的コード解析は、技術的負債のさまざまな原因を特定するのに役立つ。例えば、複雑さの増大、循環する依存関係、異なるモジュール間の競合、コードカバレッジの不足、セキュアではないコードなどである。他にも、テストの成果物、インフラ、CI パイプラインなどにも技術的負債が発生する可能性がある。

技術的負債が意図的に（他の決定の結果として、あるいは妥協して）作り込まれた場合、またはコードレビューや静的コード解析によって特定された場合、技術的負債を減らす努力をしなければならない。技術的負債にはすぐに対処することが望ましい。すぐに対処できない場合は、技術的負債に対処するためのタスクを（プロダクト）バックログに追加すべきである。

コード解析とレビューに必要な時間を増やすことと、技術的負債を抱えることのトレードオフにおいては、ほとんどの場合、コード解析とレビューが優先される。欠陥や技術的負債のあるコードが増加すると、システムの他の部分に悪影響を与えずに修正することがますます困難になり、コストと時間がかかるようになる。コード解析とレビューは、コードの品質を向上させ、全体的な時間の浪費を減らすことができる。

不具合の発見や技術的負債の管理に役立つだけでなく、コード解析とレビューにはさらなる利点がある。

- トレーニングと知識の共有
- コードの堅牢性、保守性、読みやすさの向上
- 一貫性のあるコーディング基準を保守し、見落としを減らす

コードレビュー

コードレビューに参加することで、テスト担当者ならではの視点を活かして、プログラマーと協力して潜在的な欠陥を特定し、技術的負債を非常に早い段階で回避することで、コードの品質向上に価値ある貢献をすることができる。テスト担当者は、レビューするコードで使用されているプログラミング言語を読む能力が必要であるが、コードレビューに効果的に参加するために高度なコーディングスキルを持っている必要はない。テスト担当者は、コードの動作に関する質問をしたり、考慮されていない可能性のあるユースケースを提案したり、品質上の問題を示すコードメトリクスを監視するなど、さまざまな方法でその専門知識を活かすことができる。また、コードレビューは、プログラマーとテスト担当者との間で知識を共有する機会にもなる。アジャイルプロジェクトにおけるコードレビューの主な難題の１つは、イテレーションが短いにもかかわらずコードレビューに時間がかかることである。コードレビューの計画を立て、各イテレーションにおいてコードレビューを行うために必要な時間を確保することが重要である。

コードレビューは、開発担当者以外の他の人が、あるいは開発担当者が他の人と一緒に、場合によってはツールの支援の元に行う自動化されていない活動である。通常、チームリーダーや経験豊富なプログラマーがコードレビューを行うが、他のチームメンバーと一緒に行うこともできる。テスト担当者やその他の非プログラマーがコードレビューに参加することは、有益であることが多い。

コードレビューのアプローチは、その形式と厳密さのレベルによって異なる[Wiegers02]。形式的で厳密なアプローチは、より完全性が高いものになるが、時間もかかる。一方、形式的でなく、厳密でもないアプローチでは、完全性はやや劣るが、スピードは格段に向上する。ピアレビューにはさまざまな種類があるが、アジャイルチームは、通常はコードの統合前に頻繁に行われる、より迅速なレビューを好む傾向がある。

コードレビューは、レビュアーと開発担当者が並んで座って行うこともできる。この形式は、アドホックレビューやペアプログラミングでよく見られるもので、優れたコミュニケーションを促進し、より深い分析とより良い知識の共有を促す。また、チームの結束力や士気の向上にもつながる。

分散したチームや、より独立したアプローチを好むチームでは、コードレビューのプロセスは構成管理システムによって支援される。このプロセスは通常、継続的インテグレーションのプロセスの一部として部分的に自動化されている。これらのプロセスは、回覧レビューにおいてそれぞれのレビュアーによるコードレビューをサポートする場合もあれば、チームによる共同レビューをサポートする場合もある。

静的コード解析

静的コード解析では、ツールがコードを解析し、コードを実行することなく特定の問題を検出する。静的コード解析の結果は、コードの明確な問題を指摘する場合もあれば、さらなる調査を必要とすることを間接的に示す場合もある。

多くの開発ツール、特に統合開発環境（IDE）では、コードを書きながら静的なコード解析を行うことができる。これにより、すぐにフィードバックが得られるという利点があるが、継続的インテグレーションの際に実行される解析の一部でしかない。

3 テスト自動化 135 分

キーワード

データ駆動テスト, キーワード駆動テスト, テスト手順, テストアプローチ

「テストプロセス」の学習の目的

ATT-3.x （K1） キーワード

3.1 テスト自動化技法

ATT-3.1.1 （K3）自動テストスクリプトを開発するために、データ駆動およびキーワード駆動のテスト技法を適用する。

ATT-3.1.2 （K2）アジャイル環境に求められるテストアプローチにテスト自動化を適用する方法を理解する。

ATT-3.1.3-1 （K2）テスト自動化を理解する。

ATT-3.1.3-2 （K2）さまざまなテストアプローチの違いを理解する。

3.2 自動化のレベル

ATT-3.2.1-1 （K2）デプロイメントのスピードに追随するために、必要なテスト自動化のレベルを決定する際に考慮すべき要素を理解する。

ATT-3.2.1-2 （K2）アジャイルの背景におけるテスト自動化の課題を理解する。

3.1 テスト自動化技法

3.1.1 データ駆動テスト

モチベーション

データ駆動テストとは、同じテストステップを持ちながら、テストデータの入力値の組み合わせが異なるテストケースを開発したり、保守したりするのに必要とされる労力を最小限に抑えるテスト自動化技法である。データ駆動テストは、すでに確立された技法でアジャイルプロジェクト特有ではない。テストの自動化や保守の労力を軽減することができるため、この技法はすべてのアジャイルプロジェクトにおいてテスト自動化戦略の一部として考慮されるべきである。

コンセプト

データ駆動テストの基本的な考え方は、テストデータからロジックを分離することである。別のテストデータのリストまたはテーブルで提供される、テストデータの値をテスト手順で指定する。そうすることで、テスト手順は、異なるテストデータのセットを使用して繰り返し実行することができる。詳細と例については、『Advanced Level Specialist シラバス テスト自動化エンジニア』[AdvancedTestAutomationEngineer] を参照されたい。

アジャイルチームにとっての利点

- アジャイルチームは、イテレーションごとに製品の機能を変更、あるいは追加しても、迅速に対応することができる。なぜなら、新しいデータの組み合わせの変更や追加が容易で、既存の自動テストにほとんど、あるいは全く影響を与えないからである。
- アジャイルチームは、テストデータのテーブルの行を追加、変更、または削除することで、必要なテストカバレッジを簡単に拡げたりまたは狭めたりできる。これは、アジャイルチームがテスト実行時間をコントロールし、継続的デプロイメントの制約条件を満たすのに役立つ。
- テストデータのテーブルはテストスクリプトよりも理解しやすく、技術力の低いチームメンバーでも効果的に参加できる。よって、この技法はクロスファンクショナルな作業を行うという考えに従っているといえる。
- この技法を使うことでテストデータのテーブルがより容易に理解できるため、技術者ではないチームメンバーや顧客／ユーザーからのテストケースや受け入れ基準に対する、早期のフィードバックが可能になる。
- この技法は、新しいテストを開発するための労力を軽減するだけでなく、既存のテストの保守の労力も軽減するため、アジャイルチームがテスト自動化タスクをより効率的に完了するのに役立つ。

アジャイルチームの制約

アジャイルコンテキストでこの技法を使うことに特別な制約はないが、アジャイルチームはこの技法を使うにあたっての一般的な制約を知っておく必要がある。詳細は、『Advanced Level Specialist シラバス テスト自動化エンジニア』[AdvancedTestAutomationEngineer]を参照されたい。

ツール

- テストデータの編集や保存、管理は、通常、スプレッドシートまたはテキストファイルを利用して行われる。
- テスト自動化に使われるほとんどのツールや言語には、スプレッドシートやテキストファイルからテストデータを読み込むためのコマンドが用意されている。

3.1.2 キーワード駆動テスト

モチベーション

テスト自動化の欠点のひとつは、自動化されたテストスクリプトが、自然言語で記述された手動のテスト手順よりもはるかに理解しにくいことである。キーワード駆動型のテスト自動化技法を適用することで、自動化されたテストスクリプトの可読性、理解容易性、および保守性を向上させることができる。

コンセプト

キーワード駆動テストの基本的な考え方は、プロダクトのユースケースや顧客の事業ドメインから得られた一連のキーワードを定義し、テスト手順を表現するためのボキャブラリーとしてそれらを使用する。自然言語に近いため、結果として得られる自動テストのスクリプトは、平易なプログラムやスクリプト言語と比べても理解しやすくなる。詳細は、『Advanced Level Specialist シラバス テスト自動化エンジニア』[AdvancedTestAutomationEngineer] を参照されたい。

キーワードを使用してテスト手順を記述する方法には、以下に示すさまざまな形式がある（[Linz 14] 6.4.2 キーワード駆動テスト参照）：

- リスト形式：テスト手順は、キーワードを使って表現された複雑ではないシーケンスやリストである。
- 振る舞い駆動開発（BDD）形式：テスト手順（またはシナリオ）は、「実行可能な」キーワードを含んだ自然言語に近い文で記述される（2.1.2 項 振る舞い駆動開発（BDD）参照）
- ドメイン固有言語（DSL）形式：「ドメイン固有言語」（DSL）の概念（[Fowler/Parsons10]参照）に基づいた形式的な文法により、キーワード（および、その他の言語要素）をどのように組み立てるかを定義する。

アジャイルチームにとっての利点

- アジャイルチームは、キーワードや DSL の定義によってドメインに関連したチームの用語集の作成と標準化をおこなう。これにより、チームメンバーはより明確で正確なコミュニケーションが可能となり、認識のずれを避けることができる。
- アジャイルチームは、より質の高いフィードバックを顧客やユーザーから素早く集めることができる。キーワードで構成されたり、BDD/DSL の形式で書かれたり、またはその両方を取り入れたテスト手順は、プログラム言語だけのテストコードよりも、顧客の理解を深めることができる。そして形式的でない受け入れ基準を、「自然言語による読みやすさ」を失うことなく形式化できる。これにより、ビジネスロジック自体の理解が深まり、受け入れ基準の解釈の助けになる。
- テストケース作成のこの方法は、リビングドキュメントの作成と管理であると考えられる。

- 既存のキーワードからテスト手順を作成する際にプログラミングのスキルを必要としないため、組織横断的なやり方で技術的な知識を持たないメンバーがいる場合でも、アジャイルチームがテスト自動化のタスクを達成するのに役立つ。
- 定義されたキーワードの動作の変更は、複数のテスト手順にまたがって同じ動作を変更するよりもはるかに少ない労力で済む。これにより、保守の手間が大幅に軽減され、新しいテストの実装に時間を割くためのリソースが確保される。
- アジャイルチームは、この技法をテストピラミッド全体に適用することができる（結果として、その利点を実感することができる）。なぜなら、キーワードは、ユーザーインターフェースレベルのテストのレベルからより具体的な API レベル（API コール、REST コール、SOAP コールなど）まで、任意のインターフェースでテスト対象を操作するように記述できるからである。つまり、このコンセプトは、ユニットテストや統合テストのテストケースにも適用可能であり、よく想定されるようなユーザーインターフェースでのシステムテストに限定されるものではない。しかし、ユニットテストには不必要な抽象度を加えてしまう可能性がある。

アジャイルチームの制約

アジャイルチームは、『ISTQB テスト技術者資格制度 Advanced Level シラバス 日本語版 テスト自動化エンジニア』で説明されているキーワード駆動テストの一般的な制約の他に、以下の制約と潜在的な落とし穴に注意する必要がある。

- キーワード駆動型のテスト手順を実行するには、適切な実行フレームワーク（例えば、キーワードインタープリタを含む）が必要である。フレームワークを一から作り直すことは推奨されない。キーワード駆動テストをサポートする既存のフレームワークやツールを使うことで、チームはより高いベロシティを得ることができる。これは特に、チームが BDD や DSL 形式を採用している場合に当てはまる。
- 永続的に保守可能な新しいキーワードを実装するのは難しく、経験と優れたプログラミングスキルが必要である。また、キーワードの実装作業は、プロダクトのコーディング作業と同時に行われる。そのため、結果として得られるテスト自動化のベロシティは、予想よりも低くなる可能性がある。
- キーワードのセット（ネーミング、抽象度）や DSL（文法ルール）がうまく設計されている必要がある。そうでなければ、理解しやすさやスケーラビリティが期待に添うことはない。
- また、キーワードのセットも適切に管理しなければならない。これができていないと、類似のキーワードが複数回実装されたり、キーワードがテストケースで全く、あるいはめったに使用されなかったりするため、キーワードの実装が無駄になる。これらの落とし穴を避けるために、チームではキーワードのボキャブラリーを管理する責任者を決めておく必要がある。
- キーワード駆動テストを適用するには、ある程度の先行投資が必要であり（例：キーワードやドメイン言語の定義、適切なフレームワークの選択、最初のキーワードセットの実装）、プロジェクトの初期段階ではテスト自動化のベロシティが低下する可能性があることを、チームは認識しておく必要がある。

ツール

- データ駆動テストと同様に、キーワード駆動テストの編集、保存、管理には、スプレッドシート、テキスト、フラットファイルを使用するのが一般的であるが、限界のあるアプローチである。
- いくつかのテストフレームワーク、テスト実行ツール、テスト管理ツールには、キーワード駆動テスト（ツールベンダーやフレームワークによって、「キーワード駆動テスト」、「イ

ンタラクションベーステスト」、「ビジネスプロセステスト」、「振る舞い駆動テスト」と名付けられている）が組み込まれて提供されている。利用可能なツールは[ToolList]で確認できる。

3.1.3 特定のテストアプローチへのテスト自動化の適用

テスト自動化は、テストの目的ではない。テスト自動化は戦略である。適切に推進すれば、テストの効率が高まり、特定の種類の欠陥（性能や信頼性に関する欠陥など）に対してテストが有効なものになり、欠陥の早期発見を可能にできる。その結果、より広く戦略的なテスト目的の達成に寄与する。戦略は状況に応じて継続的に進化していくものである。

テストの自動化は、チームの状況やニーズに応じて、さまざまな形態やツールを用いて行われる。大規模なプロジェクトでは、通常、すべてのニーズに適合する単一のソリューションは存在しないため、複数のテスト自動化戦略を用いる。テスト自動化の適用は、組織のテスト戦略や、特定のプロジェクトにおけるテストアプローチに適したものでなければならない。

テスト自動化は、単にテストの実行を自動化するだけではない。いくつか例を挙げると、テスト自動化は、テスト環境の設定、テスト対象の取得、テストデータの管理、テスト結果の比較などで重要な役割を果たすことができる。このようなツールの使用を計画し設計する際には、テストアプローチ、アジャイルライフサイクル採用の意図、テスト自動化ツールの能力、およびこれらのテスト自動化ツールと他のさまざまなツールとの統合（例：継続的インテグレーションフレームワーク内でのテスト自動化）を考慮する。

多くの場合、テスト自動化がイテレーションの目標、すなわち新しい機能の開発に直接貢献する。一方で、イテレーションの目標に、テスト自動化が間接的に貢献する場合もある。例えば、システムへの変更に関連するリグレッションリスクを減らすことができる。『Foundation Level Extension シラバス アジャイルテスト担当者』[AgileFoundationExt]で説明されているように、組織によっては、これらの支援的なテスト自動化の取り組みを、イテレーションチーム以外のチームに任せることを選択している。そのような組織では、例えば、複数のイテレーションチームに対するサービスとして、リグレッションテストの自動化フレームワークの作成と保守を提供する別のチームが存在することがある。このアプローチは、外部チームがイテレーションチームに有益なサービスを提供し、イテレーションチームが直近のイテレーションの目標に集中できるようになれば成功である。外部チームへ依存することは、チームのコミットメントに対するコントロール（の一部）を移転することであり、チームのコミットメントに影響する。

以下は、Foundation Level シラバス、Advanced Level シラバス テストマネージャ、Expert Test Manager のシラバスに記載されている主要なテストアプローチに対するテスト自動化の考慮点の例である。

分析的：振る舞い駆動開発（BDD）と受け入れテスト駆動開発（ATDD）は、アジャイル開発における分析的アプローチで使用できる技法であり、テスト自動化にも適用できる。BDD と ATDD は、ユーザーストーリーの実装と並行して（あるいはその前に）自動テストを作成するために使用できる。

モデルベースド：機能的振る舞いのモデルベースドテストは、効率的なテストの情報源を迅速に提供することで、ユーザーストーリーの実装時にテストの自動作成をサポートする。モデルベースドテストは、ユーザーストーリーの作成にも使用できる。モデルを使用して要件をテストし、静的なレビュ

ーに役立てることができるからである。モデルは、システム全体を対象として導出され、多くのシステムにとって重要な特性である信頼性や性能などの非機能的な振る舞いのテストにもよく使用される。

系統的：アジャイルプロジェクトでは短期間で複数回のイテレーションが行われるため、自動テスト向けのチェックリスト（すべてのアクティビティを含む）は、安定したテストセットを効率的に実行するための系統的アプローチとして使用できる。

プロセス準拠：外部で定義された標準や規制に準拠しなければならないプロジェクトでは、これらの標準や規制が、自動テストの使用方法や自動テストの結果の取得方法に影響を与えることがある。例えば、FDA（米国食品医薬品局）の規制（高リスク）を受けるプロジェクトでは、自動テストとその結果は要件に遡って追跡可能でなければならず、結果にはテストの合格を証明する十分な詳細情報が含まれていなければならない。

対処的（ヒューリスティック）：アジャイルテストでは、ほとんどの自動テストが主に検証の役割を果たしているのに対し、対処テストは重要な妥当性確認の役割を果たしている。対処的な戦略は主に手動で行われるが（例：探索的テスト、エラー推測など）、自動テストのカバレッジの割合が増えると、前もって準備できるテストの多くが自動化されるため、対処的な戦略に従う手動テストの割合が増えることが多い。さらに、残った手動テストで、よりリスクの高い領域をカバーできる。

指導ベース（コンサルテーションベース）：外部のステークホルダーからテストカバレッジとテスト自動化の適用が指示される場合、テストチームがその要求に応えられるかどうかが重要になる。したがって、指導ベースのテスト戦略をとるテストチームは、イテレーション内でタスクを完了するために必要な時間とスキルの両方を考慮する必要がある。

リグレッション回避：アジャイルプロジェクトにおいて、リグレッション回避戦略の主な特徴は、自動化されたリグレッションテストのセットが大きく、安定しており、かつ増加していることである。特にリグレッションテストの数が増えると、十分なカバレッジ、保守性、効率的な結果分析が重要になる。リグレッション回避戦略で成功するには、増え続けるリグレッションテストのセットに着目するのではなく、作成したテストの継続的な改善とリファクタリングに着目する。

3.2 自動化のレベル

3.2.1 必要なテスト自動化のレベルを理解する

アジャイルプロジェクトにおいて自動化は重要な要素である。なぜなら、テストの自動化のみならず、デプロイメントプロセスの自動化もカバーしているからである（第 4 章を参照）。継続的デプロイメントとは、新しいバージョンを本番環境に自動的にデプロイすることである。継続的デプロイメントは、定期的で短い間隔で行われる。

継続的デプロイメントは自動化されたプロセスである以上、求められているレベルのコード品質を維持するのに十分な自動テストを用意しなければならない。単に自動ユニットテストを実行するだけでは、十分なテストカバレッジを得ることはできない。デプロイメントプロセスの一環として実行される自動テストスイートには、統合テストとシステムレベルのテストを含めければならない。実際には、手動テストが必要な場合もある。

アジャイル環境でのテスト自動化で直面する課題は以下の通りである。

- **テストスイートの量：**各イテレーション（保守やハードニングのイテレーションを除く）では、追加の機能が実装される。プロダクトの機能追加にテストスイートが追従し続けるために、アジャイルチームはイテレーションの中でテストスイートを増強しなければならない。これは、テストスイート内のテストケース数が、通常はイテレーションごとに増加することを意味する。テストケース数を大幅に増やさずにカバレッジを高めるため、テストのリファクタリングに注意した取り組みが必要である。いずれにせよ、完全なテストスイートを保守し、準備して実行するのに必要な労力と時間は、徐々に増加する。
- **テスト開発時間：**プロダクトの新機能や変更された機能を検証するために必要なテストを設計し、実装しなければならない。これには、必要なテストデータの作成や更新、テスト環境の更新の準備も含まれる。また、テストの保守性も開発に必要な時間に影響する。
- **テスト実行時間：**テストスイートの量が増えることによって、テストの実行に必要な時間の総計が増える。
- **スタッフの確保**：テストスイートの作成、保守、実行のためのスタッフはデプロイメントのたびに必要となる。プロジェクトの期間全体で、とりわけ休日や週末、あるいは時間外にデプロイメントが行われる場合には、これを確保することが困難または不可能なことがありえる。

これらの課題に対処する戦略のひとつは、テストの優先順位付けやリスク分析により、テストのサブセットのみを選択、準備、実行して、テストスイートを最小化することである。これは実行されるテストの数が減ることを意味するため、この戦略にはリスクが増大するという欠点がある。

可能な限り多くのテストを自動化することで、デプロイメントの頻度や速度を向上させることができる。つまり、デプロイメントの頻度やスピードを維持（あるいは向上）するためのもうひとつの戦略は、できるだけ多くのテストを自動化することである。

テスト自動化は、どのようなプロジェクトにおいても、デプロイメントのスピードを維持または向上させるための有効な手段であり、継続的デプロイメントの前提となるものである。デプロイメントのスピードを維持できるテスト自動化の適切な量を見つけるためには、以下の利点と制限事項を分析し、バランスを取る必要がある。

利点

- テスト自動化は、各デプロイメントサイクルにおいて、定義された繰り返し可能なレベルのテストカバレッジを確保することができる。
- テスト自動化は、テストの実行時間を短縮し、デプロイメントのスピードを上げるのに役立つ。
- テスト自動化によって、デプロイメント頻度を高めるためのボトルネックを緩和できる。
- 継続的デプロイメントは、市場投入までの時間とユーザーからのフィードバックループを短縮する。
- テスト自動化が進むと、毎回のビルドで全てのテストが実行されるので、ソフトウェアがメインラインにマージされる継続的インテグレーション（CI）環境では、安定したメインラインを提供できる。

制限事項

- テスト自動化は、テストの開発と保守に工数が掛かる。それにより、開発期間が長くなり、デプロイメント頻度を低下させる可能性がある。

- システムテストレベルのテスト、特に負荷テストや性能テスト（場合によってはその他の非機能テストも）は、自動化されていても実行に長時間を要することがある。
- 自動テストは様々な要因でうまくいかないことがある。合格した自動テストケースは、信頼できないこともある（偽陰性）。失敗した自動テストケースは、プロダクトの品質とは関係のないエラーや、偽陽性が原因かもしれない。

プロダクトのリリース頻度は、ユーザーのニーズに合わせる必要がある。短期間にあまりにも多くのバージョンを提供することは、提供頻度が下がることよりも顧客に不快感を与える可能性がある。そのため、技術的にはデプロイメント頻度をさらに上げることが可能であっても、顧客の視点ではそれ以上の利点がない場合もある。

4 デプロイメントとデリバリー 105 分

キーワード
サービス仮想化、継続的テスト

「デプロイメントとデリバリー」の学習の目的

ATT-4.x （K1） キーワード

4.1 継続的インテグレーション、継続的テスト、継続的デリバリー

ATT-4.1.1 （K3）継続的インテグレーション（CI）を適用し、そのテスト活動への影響を要約する。

ATT-4.1.2 （K2） 継続的デリバリ/継続的デプロイメント（CD） における継続的テストの役割を理解する。

4.2 サービスの仮想化

ATT-4.2.1-1 （K2）サービス仮想化の概念と、アジャイルプロジェクトにおけるその役割を理解する。

ATT-4.2.1-2 （K2） サービス仮想化の利点を理解する。

4.1 継続的インテグレーション、継続的テスト、継続的デリバリー

4.1.1 継続的インテグレーションとテストへの影響

継続的インテグレーション（CI）の目的は、迅速なフィードバックを提供することであり、コードに混入した欠陥を、できるだけ早く発見して修正することである。アジャイルテスト担当者は、CI フレームワークに適合した自動テストを作成・保守するだけでなく、テストの優先順位、必要な環境、構成などの観点から、効果的かつ効率的な CI プロセスの設計、実装、保守に貢献しなければならない。

CI の理想的な状況では、コードがビルドされると、その後、すべての自動テストが実行され、ソフトウェアが想定通りに動作し続け、コードの変更によって壊れていないことが検証される。しかし、そこには 2 つの相反するゴールがある。

1. CI プロセスを頻繁に実行し、コードについてのフィードバックを即座に得る
2. ビルドごとにできるだけ徹底的にコードを検証する

自動テストの設計、実装、保守に十分な注意を払わないと（前章で述べたように）、すべての自動テストの実行に時間がかかりすぎて、一日に何度も CI プロセスを完了させることができなくなる。慎重にテストを自動化したとしても、すべてのテストレベルですべての自動テストを完全に実行すると、CI プロセスが過度にスローダウンすることがある。それぞれの組織はそれぞれ異なる優先順位を持つ。そして、それぞれのプロジェクトには、上述したゴールの適切なバランスを見つけるためのそれぞれのソリューションが必要になる。例えば、システムが安定していて変更頻度が低いなら、より少ない CI サイクルを回すだけでよく、システムが常に更新されている場合は、より頻繁に CI サイクルを回す必要がある。

両方のゴールをサポートする複数のソリューションがあり、それらは補完的で、並行して使用することができる。

1 つ目のソリューションは、リスクベースドテストを用いて、基本的で最も重要なテストが常に実行されるように、テストに優先順位をつけることである。

2 つ目のソリューションは、異なる種類の CI サイクルに合わせて、異なるテスト構成の CI プロセスを利用可能にすることである。デイリーのビルドとテストのプロセスでは、決めておいた優先順位付けに基づいて基本的なテストのみを選択して実行する。ナイトリーの CI プロセスでは、本番前環境を必要としない機能テストをできるだけ多く、可能ならすべて実行する。リリース前には、データベース、異なるシステムやプラットフォームとの統合テストを含む、実際のユーザー入力を用いた、より徹底した機能テストおよび非機能テストが本番前環境で行われる。

3 つ目のソリューションは、ユーザーインターフェース（UI）テストの量を減らすことで、テストの実行を高速化することである。通常は、ユニットテストや統合テストは非常に高速なので、テストの実行完了について時間的な問題はない。ただし、ユニットテストの数が多すぎて、CI プロセスの中で完全に実行できないという状況もある。時間に関する本当の問題は、一般的に、CI プロセスの一部としてエンドツーエンドの UI テストケースを使用することに関連している。解決するには、API、コマンドライン、データ層、サービス層など、UI 以外のビジネスロジックのテストを増やし、UI のテストを減らして、テスト自動化のピラミッドの下の方に自動化の作業を移すとよい。こうすることで、より保守性の高いテストとなり、テスト実行時間も短縮される。

4 つ目のソリューションは、CI システムでテスト実行が非常に頻繁に行われ、すべてのテストケースを実行することが不可能な場合に使用できる技法である。コードの変更と、既存のテストケースの実行履歴のナレッジに基づいて、開発者やテスト担当者は、変更の影響を受けるテストケースだけを選択して実行することができる（すなわち、影響度分析を使用してテストを選択する）。短いサイクルの間に変更されるのはコードベース全体のごく一部なので、実行しなければならないテストケースの数は比較的少なくて済む。一方で、重大なデグレードを見逃してしまう可能性がある。

5 つ目のソリューションは、テストスイートを同程度の長さのまとまりに分割し、複数の環境（エージェント、ビルドファーム、クラウド）で並行して実行することである。このソリューションは、CI を使っている企業では、すでに大量のビルドサーバーのキャパシティを必要としているため、当たり前に適用されている。

継続的インテグレーションは、さまざまなテクノロジープラットフォームで機能する。開発ツールやデプロイメントツールのクラウド化に伴い、CI 製品もクラウドに移行した。ほとんどの CI 製品は、ダウンロードしてローカル環境で実行するように設計されているが、クラウドの登場により、チームがすぐに使い始めることができるように、ホストされたプラットフォーム上で CI サービスを提供する新種の製品が出現した。これでチームは、新しい環境の構築、CI ソフトウェアのダウンロード、インストール、設定といった余計なコストと時間をかけずに済む。クラウドに移行したチームは、設定だけで作業を開始できる。実用上も、クラウドは必要な時にテストのビルドと実行をスピードアップするための柔軟な方法である。

現在の CI ツールは、継続的インテグレーションだけでなく、継続的デリバリーや継続的デプロイメントもサポートしている。本番環境を完全に再現していない環境で自動テストを実行すると、欠陥の見逃しにつながる可能性があるが、本番環境を複製するのはコストがかかりすぎる。そのため、必要に応じて本番環境を再現したクラウドテスト環境を利用したり、本番環境を縮小した現実的なテスト環境を構築したりする。

優れた CI システムであるためには、より複雑な環境やさまざまなプラットフォームに自動的にデプロイできる必要がある。テスト担当者の仕事は、良好なカバレッジを維持するため、一部または全てのプラットフォームについて、含めるべきテストケースと実行すべきテストケースの優先順位を計画し、さらに、本番同様の環境でソフトウェアを効率的に検証するためのテストケースを設計することである。

4.1.2 継続的デリバリーと継続的デプロイメント（CD）における継続的テストの役割

継続的テストとは、ソフトウェアのリリース候補のビジネスリスクのフィードバックを可能な限り迅速に得るために、「早期にテストする」「頻繁にテストする」「どこでもテストする」「自動化する」というプロセスを含むアプローチである。継続的テストでは、システムに変更が加えられると、必要なテスト（変更点をカバーするテスト）が自動的に実行され、開発者に迅速なフィードバックがもたらされる。継続的テストは、さまざまな状況（IDE、CI、継続的デリバリー、継続的デプロイメントなど）に適用できるが、テストをできるだけ早い段階で自動的に実行するというコンセプトは同じである。

継続的デリバリーには継続的インテグレーションが必要である。継続的デリバリーは継続的インテグレーションを拡張したもので、全てのコード変更をビルド後にテスト環境またはプリプロダクション

環境、かつ/またはプロダクションに近い環境へデプロイする。このステージングプロセスにより、実際のユーザー入力を適用した機能テストや、負荷テスト、ストレステスト、パフォーマンステスト、移植性テストなどの非機能テストを実行することが可能になる。反映されたすべての変更は、完全な自動化によってステージング環境に配信されるため、アジャイルチームは、完了（done）の定義が達成されたときにボタンを押すだけでシステムを本番環境にデプロイできるという確信を持つことができる。

継続的デプロイメントは、継続的デリバリーをさらに一歩進めたもので、すべての変更が自動的に本番環境にデプロイされるという考え方である。継続的デプロイメントは、新しいコードを書くという開発作業から、そのコードが本番で実際のユーザーに使われるまでの時間を最小限にすることを目的としている。
詳細は[Farley]を参照のこと。

4.2 サービスの仮想化

サービスの仮想化とは、接続されたシステムやサービスについて、適切にその振る舞いや、データ、性能をシミュレートする、共有可能なテストサービス（仮想サービス）を作成するプロセスである。この仮想サービスを利用することで、実際のサービスが開発中または利用できない場合でも、開発チームやテストチームが作業できる。

今日のような高度に連携し相互依存している開発チームやシステムに対して、開発サイクルの中で早期にテストを実施することは困難である。サービス仮想化によってチームとシステムを切り離すことで、より現実的なユースケースや負荷に基づいて、開発ライフサイクルの中で早期にソフトウェアをテストすることができる。

スタブ、ドライバー、モックは早期のテストを可能にする価値のあるものだが、手作りのスタブは通常ステートレスであり、固定された応答時間でリクエストに対する単純なレスポンスを返却する。仮想サービスは、ステートフルなトランザクションをサポートし、動的な要素（セッションや顧客 ID、日時など）のコンテキストを保持し、さらに複数のシステムを介したメッセージフローを模した、可変の応答時間を設定できる。

多くの場合、サービス仮想化ツールを用いて、以下のような方法で作成される。

- データを利用する方法：XML ファイル、サーバーログからの履歴データ、または単にスプレッドシートにあるサンプルのデータを読み取る。
- ネットワークトラフィックを監視する方法： SUT（System under test）の実行が、依存するシステムの関連する動作のトリガーになる。これをサービス仮想化ツールによって捕捉しモデル化する。
- エージェントによるキャプチャを行う方法：サーバーサイドまたは内部のロジックは通信メッセージから判断できない。そこで、依存するサーバーに該当するデータやメッセージをプッシュさせ、仮想サービスとして再作成する。

上記のいずれにも該当しない場合は、適切な通信プロトコルに基づいて、プロジェクトチーム内で仮想サービスを作成する必要がある。

サービス仮想化のメリットは以下の通りである。

- 開発中のサービスの、開発活動とテスト活動の並走

- サービスや API の早期テスト
- 早期のテストデータのセットアップ
- コンパイル、継続的インテグレーション、テスト自動化の並列化
- 欠陥の早期発見
- 共有リソース（COTS システム、メインフレーム、データウェアハウス）の過剰利用の削減
- インフラへの投資を抑えることで、テストコストを削減
- SUT の早期の非機能テストが可能
- テスト環境管理の簡素化
- テスト環境の保守作業の軽減（ミドルウェアの保守が不要）
- データ管理の低リスク化（GDPR 対応にも役立つ）

仮想サービスは、実際のサービスの機能やデータをすべて備えている必要はなく、SUT のテストに必要な部分だけで良い。

サービス仮想化の導入は、複雑でコストのかかる作業になる可能性がある。サービス仮想化ツールの導入は、チームや組織に新しいテストツールを導入するのと同様に扱う必要がある。このトピックは、『Foundation Level シラバス』[Foundation]および『Advanced Level シラバス テストマネージャ』でカバーされている。

5 リファレンス

[AgileFoundationExt] ISTQB テスト技術者資格制度 Foundation Level Extension シラバス アジャイルテスト担当者 日本語版 Version 2014.J02.

[Foundation] ISTQB テスト技術者資格制度 Foundation Level シラバス 日本語版 Version 2018V3.1.J03

[AdvancedTestAutomationEngineer] ISTQB 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.J01.

[ISO 29119-5], ISO/IEC/IEEE 29119-5:2016 システム及びソフトウェア工学ーソフトウェア試験ー第5部：キーワード駆動試験

書籍

[Adzic09] Adzic, Gojko."Bridging the Communication Gap" Neuri Ltd, 2009.

[Adzic11] Adzic, Gojko."Specification by Example" Manning, 2011.

[Beck02] Kent Beck, "テスト駆動開発", オーム社, 2017.

[Beck99] Beck, Kent. "エクストリームプログラミング", オーム社, 2015.

[Carkenord08] Barbara A. Carkenord, "Seven Steps to Mastering Business Analysis", J. Ross Publishing, 2008.

[Cohn 09] Mike Cohn: Succeeding with Agile: Software Development Using Scrum.Addison-Wesley Professional, 2009.

[Crispin08] Crispin, L. and Gregory, J., “実践アジャイルテスト: テスターとアジャイルチームのための実践ガイド”, 翔泳社, 2009.

[Elfriede99] Dustin Elfriede, “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.

[Evans03] Eric Evans, "エリック・エヴァンスのドメイン駆動設計", 翔泳社, 2011.

[Fewster12] Mark Fewster, Dorothy Graham, Experiences of Test Automation: Case Studies of Software Test Automation, Addison-Wesley Professional, 2012.

[Fowler/Parsons 10], Martin Fowler, Rebecca Parsons, “ドメイン特化言語 : パターンで学ぶ DSL のベストプラクティス 46 項目”, ピアソン桐原, 2012.

[Hendrickson13] Elisabeth Hendrickson, "Explore It!Reduce Risk and Increase Confidence with Exploratory Testing", Pragmatic Bookshelf, 2013.

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, "XP エクストリーム・プログラミング導入編", ピアソン・エデュケーション, 2001.

[Jorgensen13] Paul C. Jorgensen, “Software Testing: A Craftsman's Approach”, Auerbach Publications; 4th edition, 2013.

[Linz14] Tilo Linz, “Testing in Scrum, A Guide for Software Quality Assurance in Agile World”, Rocky Nook, 2014.

[Meszaros07] Gerard Meszaros, "xUnit Test Patterns: Refactoring Test Code", Addison-Wesley, 1st Edition, Apress, 2007.

[Michelsen12] John Michelsen and Jason English, "Service Virtualization: Reality is Overrated", Apress, 1st Edition, 2012.

[Osherove09] Roy Osherove, "The Art of Unit Testing", Manning, 2009.

[Paskal15] Greg Paskal, "Test Automation in the Real World: Practice Lessons for Automated Testing”, MissionWares, 2015.

[Smart15]; Smart, John Ferguson; "BDD in action", Manning, 2015

[Wein89] Weinberg, Gerald & Gause, Donald."要求仕様の探検学—設計に先立つ品質の作り込み—", 共立出版, 1993.

[Whittaker09] James A Whittaker, "Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design", Addison-Wesley Professional, 2009.

[Wiegers02] Karl Wiegers, "ピアレビュー： 高品質ソフトウェア開発のために", 日経 BP ソフトプレス, 2004.

アジャイル用語
ISTQB®用語集に掲載されているキーワードは、各章の冒頭に記載されている。アジャイルの一般的な用語については、定義を提供する次のよく知られたインターネットリソースに依拠している。
https://www.agilealliance.org/agile101/agile-glossary/
異なる定義がある場合は、ISTQB®用語集が有力な情報源となる。

その他の参考資料
以下の参考文献は、インターネットやその他の場所で入手可能な情報を指している。本シラバス発行時にこれらの参考文献を確認しているが、その文献が利用できなくなったとしても、ISTQB®は責任を負わない。

[Cohn09] The Forgotten Layer of Test Automation Pyramid,
https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[CyclomaticComplexity] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[Farley] http://www.davefarley.net/?cat=5

[DSL] https://en.wikipedia.org/wiki/Domain-specific_language

[Fowler07] https://martinfowler.com/articles/mocksArentStubs.html

[Fowler04] Fowler, Martin. https://martinfowler.com/bliki/SpecificationByExample.html

[Fowler03] Martin Fowler, https://martinfowler.com/bliki/TechnicalDebt.html

[Gherkin] https://docs.cucumber.io/gherkin/

[INVEST] Bill Wake, "INVEST in Good Stories, and SMART Tasks", http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[IQBBA] International Qualifications Board for Business Analysts, http://www.iqbba.org/

[IREB] 国際要求工学委員会、https://www.ireb.org/en

[IIBA] International Institute of Business Analysts (IIBA), https://www.iiba.org/

[Marick01] Marick, Brian, http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Marick03] Marick, Brian. http://www.exampler.com/old-blog/2003/08/22.1.html

[North06] Dan North, "Introducing BDD", blog post, 2006

[TESTDOUBLES] Wojciech Bulaty, Bill Wake, "Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams", https://www.infoq.com/articles/stubbing-mocking-service-virtualization-differences

[ToolList] テストツールレビュー、ソフトウェアテストツールの国際市場における情報プラットフォーム www.testtoolreview.de/en/

[xUnit] https://en.wikipedia.org/wiki/XUnit, https://en.wikipedia.org/wiki/Unit_testing

6 付録

6.1 アジャイルテクニカルテスト担当者特有の用語集

用語解説	定義
ペルソナ	ある種のユーザーを代表する架空の人物であり、彼らがどのようにシステムと関わっていくかを表している。
ストーリーボード	ビジネスプロセスを理解する目的で、ユーザーストーリーを文脈に沿って表現したシステムの視覚的表現。
ストーリーマッピング	ユーザーストーリーの優先順位を横軸に、実装されたプロダクトの複雑さを縦軸にとった二次元的な順序付けを行う手法。