circleci

# DevOps 駆動チームのソフトウェアテスト

**著者:エンジニアリングマネージャー、**
**JUNE JUNG**
github.com/junejung

DevOps という用語を最近よく耳にしているものの、DevOps が何であるか、また、自分の組織がどのように DevOps に関わっていくのかよくわからない方にとって、本書はきっと役立つでしょう。本書では、DevOps の詳細について説明します。本書をご覧いただくと、DevOps がエンジニアリングチーム、そして最終的にはエンドユーザーに利益をもたらすことができるかどうかを判断できるようになります。

DevOps については後で、詳細に説明しますが、簡単に紹介すると、DevOps は開発者と運用者の活動分野とスキルセットを統合し、高速で効率的な開発インフラストラクチャを促進するソフトウェアエンジニアリングの手法です。DevOps は、基本的に反復プロセスであり、テストを早期かつ頻繁に実施できるようになり、ライフサイクルの延長とコードの運用について早期の段階で検討し、問題を特定して迅速に解決できるようにします。DevOps の最終的な目標は、製品の生産性と品質、およびエンドユーザーエクスペリエンスの両方を向上させることにあります。

現行のワークフローが適切に機能している企業であれば、DevOps への移行は不要になる場合があります。製品によっては、悪手になる場合もあります。結局、反復のプロセスは、プライバシーを含む何らかの懸念事項に対応する必要があり、リリースを大規模にせざるをえない企業には役に立ちません。

しかし、DevOps の人気が高まっているのには確固たる理由があります。この eBook では、DevOps とは何かを詳しく説明し、DevOps の採用が合理的である場合、自社のプロセス全体で DevOps をテストする方法とタイミングについて説明します。

# 目次

# DevOps への移行：本当に必要なツールとは？

問題解決のプロセスには長い歴史があり、各プロセスでさまざまなツールが利用されますが、DevOps はその中の最新のプロセスです。CI/CD システム、テストフレームワーク、監視ツール、セキュリティ監査ツールなどのツールがこれらのプロセスで利用されます。

組織が DevOps について検討するときには、多くの疑問が生じます。これらのツールのどれが必要か？組織が直面している問題、弱点、およびスローダウンを解決するのはどのツールか？ DevOps を支援するために、どのような組織構造が必要か？どのツールを実装する必要があるか？

これらはすべて大切な質問なのですが、個別に考えると、重要なポイントを逃すことになります。これらの質問だけに回答しようとすると、解決するべき課題を実際に評価する前に、解決策を検討してしまうことになります。

組織は、トップダウンモデル（「これを使用しないさい。これを行いなさい」）によって、チームのイノベーションを推進できると考えがちです。意欲的なチームリーダーは、新しい CI/CD ツールを導入し、関係者全員をプロジェクトに取り込んで、運用しようとします。採用した手法に従ってもらうために、適切なツールを提供することは大切ですが、チームリーダーがツールの価値や使用する理由を十分に理解せずに導入すると、問題が発生します。変更を指示した人でさえ、どのような理由で変更を開始したのか、何を解消したいと考えていたのかを忘れることも多くあります。悲しいことに、最初の理由が曖昧になるとツールを間違った方向で利用し始めることが多くあり、その結果、その価値が失われたり、負の価値を生み出したりすることもあります。

## DevOps を導入したい！

多くの組織が、DevOps をすばやく導入して適切に運用できるのであれば、DevOps があらゆる問題の解決策になると確信しています。DevOps の推進に関わっているのであれば、「なぜ DevOps が必要なのか？どのような価値をもたらすのか？」という問いについて考えてみましょう。

では、DevOps の概念について簡単に説明しましょう。

DevOps は、開発者と運用チーム間の連携（コラボレーション）を意味します。実質的には、インフラストラクチャに開発手法の文化を取り入れ、開発サイクルに運用の手法を取り入れることを示しています。これは実際にはどのように見えるのでしょうか？これは、インフラストラクチャをコードとして管理したり、再利用可能なコンポーネントを構築してイミュータブルなインフラストラクチャを作成し、必要なときにいつでも破棄または生成できるようになり、バージョン管理と変更の履歴を提供できるようになることを意味します。

また、製品に携わるすべての関係者が、取り組んでいる製品の最終的な結果をさらに意識するようになり、実環境でどのように機能するのか、ユーザーはどのように製品を利用しているのかなどを考えるようになります。真剣に品質向上に取り組むことは、ビジネスバリューとユーザビリティの両方を向上することにつながります。製品に携わるすべての関係者が、開発と運用の両方に関心を持って取り組むようになれば、真の意味で DevOps を取り入れたことになります。

CircleCI の経験では、このような広範な合意形成は、さまざまなスキルと専門知識を持つ関係者の多くの協力を必要となるため、ソフトウェアチームにとって特に困難になります。これらの目標を実現するには、部門横断的なチーム構造と思慮深いコミュニケーションスキルの両方が必要となります。たとえば、エンジニアがビジネス部門の誰かにデータベースの問題について説明しなければならない場合、取り組んでいる作業のデータを表示するだけでなく、重要だと考えている対象とその理由について、ビジネス部門の担当者が関心を持って聞いてくれるように、必要なコンテキストを提供する必要があります。

新しいツールは簡単で有用に見えるかもしれませんが、万能なソリューションではありません。新しい DevOps ツールを導入する前に次の事項に留意することで、成功を収める可能性が高まります。

1. **新しい変革によって達成しようとしている目標について、全員が同じ認識を共有していることを確認します。**すべての関係者が、自分が解決しようとしている問題に同意しており、この課題について協調して取り組む必要があります。

2. **小規模なプロジェクトから着手します。**組織全体を一晩で理想的な DevOps チームにしようと考えないでください。代わりに、1 つのチームから始めて、そのグループでプロセスの変更が上手くいくかを確認します。改善が見られる場合は、段階的に移行していきます。

3. **自分に役立つことをしましょう。**DevOps は組織にとって適切な解決策にならない場合もあることを理解しましょう。DevOps を取り入れることなく長い間成功を収めている企業もあります。企業文化や製品のニーズによっては、必ずしも DevOps を導入することが適切ではない企業もあります。大きな成功を収めている組織で、ウォーターフォールが非常に機能している場合もありました。たとえば、会社の製品戦略において機密性が重要視される場合、大規模な製品リリースまでにすべての製品の詳細を非公開にしておく必要があります。そのため、フィードバックを得るために段階的にリリースするプロセスは適切ではありません。このような環境では、DevOps の文化を構築することは非常に難しく、逆効果になります。

4. **常に測定しましょう。**改善プランを開始する前に、現状について正確な指標を取得します（つまり、「開発のサイクルには X 時間かかっている、などの情報」）。変更の前後に測定し、改善の状況を確認します。次のような失敗例があります。アジャイルが大流行していたときに、多くの企業がスタンドアップ（簡単な毎日のステータスミーティング）を行っていましたが、この本当の理由を理解せず、チームにプラスの効果があるかどうかを測定していませんでした。おそらく、節約した時間よりも多くの時間を無駄にしたはずです。

5. **すべてを自動化しようとしないでください。**少なくとも一度にすべてを自動化しようとしないでください。DevOps に関する誤解の 1 つは、すべてのインフラストラクチャのプロビジョニングと構成管理を自動化しなければならないというものです。これは「インフラストラクチャ構成のコード化（Infrastructure as Code)」と呼ばれます。しかし、手動で行った方がうまく機能するものもあります。自動化は万能のソリューションではありません。また、作成した自動化スクリプトを何回実行するか、その設定にどれくらいの時間がかかるかを検討しましょう。スクリプトの使用するのが数千回になるのか、数回なのかによって状況は変わります。さらに、自動化の対象として最適なソリューションを把握する作業を手動で開始しなければならない場合もあります。

それでも自動化が必要かどうかを見極めてください。アプリケーションを Docker 対応にすることは、実行した作業を再利用できる可能性が高いため、自動化するのに最適な方法です。本稼働前の環境の作成を自動化することは、自動化を実装するための優れた方法の 1 つです。ファイアウォールのセットアップを自動化する別の例を考えてみましょう。現在の多くのファイアウォールソフトウェアでは API がサポートされていないことを考えると、この自動化に価値はないかもしれません。災害に備えることは賢明なことですが、過剰なまでの対策には意味はありません。

DevOps のトランスフォーメーションを組織で検討している場合は、デリバリーのスピードと製品の品質について検討する必要があります。今障害になっているのは何か？これらの質問への回答を把握できれば、組織にいる全員が、あなたが抱えている問題点を理解できるようになり、あなたは、この問題の改善に取り組むための最適な立場にあると言えます。

## 本稼働への工程：<br>テスト環境を分離する方法と場所

DevOps への移行を決定したら、組織に新しいツールを導入することは簡単な作業ではないことを念頭に移行を進めてください。CI/CD ツールまたはその他のツールを採用する場合は、調査、分析、調整のための期間を設けてください。

ツールの採用を成功させるための最初に重要なのは人です。目的に沿って調整し、期待される結果を設定し、評価をサポートするための「過去の状況」に関する指標（メトリクス）を取得します。

次のステップは、分析です。最優先で解決する必要があるパイプラインの問題を正確に説明します。開発プロセス（「本稼働への工程」と呼びます）を検証することで、最大の問題の原因を正確に特定できます。解決するべき問題を把握できなければ、ツールに関する合理的な決定を下すことはできません。

本稼働への工程を検証する目標は、チェックポイントとして機能する明確なステージを作成することです。ある段階から次の段階に移行するときに、ビルドはこれらの品質ゲートを通過する必要があります。ゲートは、さまざまな種類のテストを使用して区分されます。テストに合格すると、ゲートで定められているレベルの品質が保証され、ビルドを続行できます。

これらのパイプラインステージ（つまり、ビルド環境）は、手元の PC から、チームスペース、アプリケーションのコードベースに至るまで、開発者が担う責任の範囲の広さによって分離されます。責任の範囲が拡大するにつれて、ミスによるコストは高くなります。ビルドを次の後続のレベルに渡す前に段階的にテストするのはこのためです。

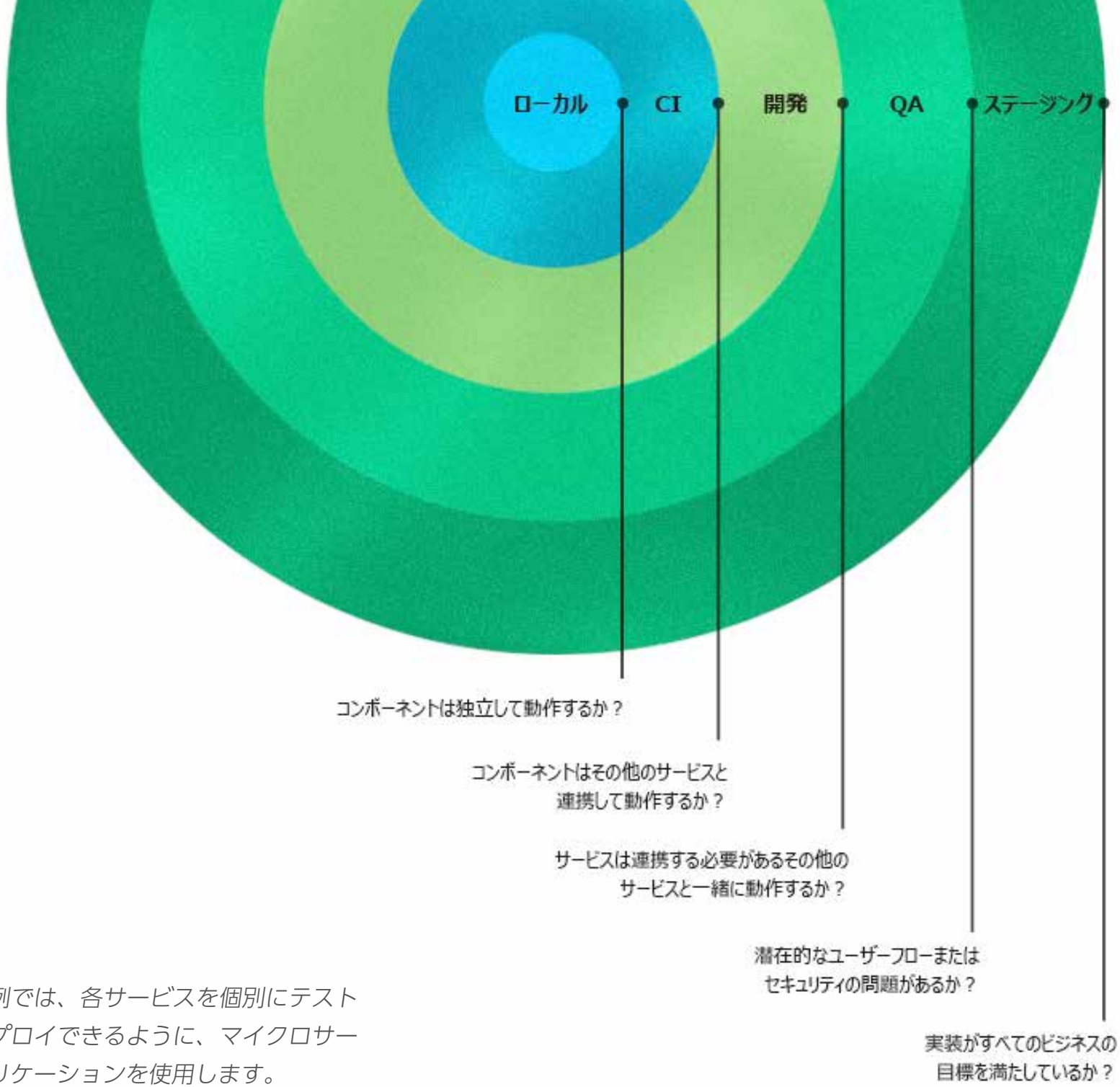
さらに、各ステージで必要となるテストのタイプは異なります。ステージング環境から本稼働に移行すると、テストはライトウェイトからヘビーデューティになります。リソースのコストも各段階で増加します。ヘビーデューティテストは、フルテクノロジースタックまたは外部の依存関係を含む、本稼働に近い環境で実行されます。これらのテストを適切に実行するには、さらに多くの作業が必要であり、より高価なマシンが必要です。そのため、前の環境で可能な限り多くのテストを行うことができれば、テストのコストを抑えることができます。


次に、いくつかの一般的なタイプのテストを見てみましょう。

**単体テスト / コンポーネントテスト：**これらは、可能な限り最小のコンポーネント、ユニット、または機能を対象としたテストです。多くの依存関係やモックが不要であるため、最も安価に実行できる最速のテストです。これらが障害にならないようにするには、これらのテストを早期に実行する必要があります。

**統合テスト：**これらは、前の段階の各ユニットが他のコンポーネント、ユニット、および機能とどの程度うまく連携するのかをチェックします。広い意味では、サービス (API など) がどのように統合されるかをテストできます。

**UI レイヤーのテスト：**これは、基本的なユーザーフローをテストする自動化されたブラウザベースのテストです。セットアップに費用がかかり、実行に時間がかかるため、パイプラインの後半で実行する必要があります。




*注：この例では、各サービスを個別にテストおよびデプロイできるように、マイクロサービスアプリケーションを使用します。*

次に、これらのテストがソフトウェア開発パイプラインにどのように適合するかについて説明します。各テストには特定の役割と実施する場所があります。

## ローカル環境

これはプライベートな環境であり、開発者 1 人とそのラップトップに制限されます。最も簡単に変更したり、独自の実装をテストしたりできます。

ここでは「ローカル」という用語を使用していますが、実際には単に「個人の環境」です。ラップトップでテストする場合もありますが、アプリケーションが大きすぎてローカルマシンで実行できない場合は、クラウド環境も使用できます。ここで重要なのは、ローカル環境は自分だけの小規模なインスタンスであり、チーム内の他の開発者の作業を中断することなく、開発中に実装をテストおよびデバッグできることです。自分のローカル環境にあるものは他のユーザーには見えないため、チームは Git pre-push hook を自分のリポジトリに統合して、ローカル環境が使用されていることを確認し、コードがリモートつまり共有リポジトリにプッシュされる前に自動テストを実行できます。

**テスト:** ローカル環境から移行する前に、単体テスト、モックコンポーネントを使用した統合テスト、および可能な限りの UI テストを実施することが推奨されます。この環境でより多くのテストを実行できるほど、統合環境で成功する可能性が高まります。

**責任の範囲：** ここで説明する範囲は、自分が構築している実装または機能のみを対象としています。コンポーネントが単独で機能するかを確認します。テスト対象は比較的小さいのですが、これは共有環境にマージする前の最後の環境です。そのため、後で共有環境に混乱を持ち込み、他のユーザーの開発労力を妨げることがないように、責任を持ってテストする必要があります。

## CI 環境

これは最も短命な環境であり、ビルドと共に利用されます。ビルドがトリガーされると作成され、ビルドが完了すると破棄されます。また、最も不安定な環境でもあります。開発者はコードを CI 環境にチェックインします。他の開発者が同時にデプロイする可能性があるため、CI 環境には多数のデプロイアクティビティが同時に発生しています。その結果、CI は破損する可能性があり、多くの場合、実際に破損します。ただ、破損することは問題ではありません。重要なのは破損した環境を修正することです。

ローカル環境でアプリケーション全体を起動できなかった場合（これはよくあることです）、ブラウザ駆動のテストを実行できる最初の環境は CI 環境になります。また、ローカル環境で実行できなかった UI テストを実行することもできます。

この環境では、モックの外部サービスとデータベースを使用して、迅速にテストを実行する必要があります。

コードをマージしたらすぐに、CI 環境が自動的に起動し、そのコードを実行し、安全かどうかを通知してから、CI 環境を破棄するように、CI 環境のライフタイムを自動化することをお勧めします。Docker を使用して CI 環境を自動的に起動すれば、時間を節約できます。自動ビルドと環境作成の包括的なプロセスがあれば、チームメンバーはより頻繁にコミットできるようになります。

## 開発環境

この開発環境は、他の開発者と共有される環境です。この環境では、アプリケーション内のすべてのサービスが毎回デプロイされます。これらの環境は、さまざまなチームの影響を受けており、絶えず変化しているため、非常に不安定です。CI 環境でテストが合格した場合でも、統合およびブラウザベースのテストが失敗する可能性があります。なぜなら、外部サービスと完全に統合され、他のサービスも現在このデプロイ環境にあるためです。

CI 環境は自分のチーム（または製品の 1 つのサービス）専用であり、ビルド間で破棄できるのに対して、この開発環境は製品コードベース全体に対応します。自分のチーム専用の開発環境にマージすることを選択した場合、すべてのチームが共有する完全な開発環境にマージするよりも影響範囲は小さくなります。開発環境には、さまざまなコンポーネントがあり、変化が速いため、システムヘルスチェックの監視が必須となります。この環境は本稼働への工程上にあるため、この環境で問題が発生すると、後続の環境にも悪影響を及ぼします。この段階で発生する障害はすべての変更の妨げとなり、この開発環境で問題が発生すると、コードを次の段階に進めることはできません。

開発環境では、各サービスがテストの対象に対してどれほど重要であるか、また、その外部サービスに接続する費用に応じて、一部の外部サービスはモックされたり、一部はモックされなかったりします。ここでモックデータを使用している場合は、十分な量のテストデータが利用可能であることを確認してください。

組織内のすべての関係者がこの環境を見ることができます。開発者は誰でもログインして、アプリケーションとして実行できます。この環境は、すべての開発者がテストとデバッグに使用できます。

## QA 環境

QA 環境は、このシナリオでは、最初に手動でデプロイされる環境です。QA チームは、変更の構造に基づいて、どの機能をテストする価値があるかを自ら決定する必要があるため、この環境は手動でデプロイされます。変更がスタック（変更 A、B、および C）になっている場合があり、各変更を個別にテストする場合があります。このシナリオでは、変更 A をテストし、期待どおりに機能することを確認したら、次に進み、変更 B をテストします。エラーが発生する場合、問題が変更 B にあることが分かります。変更 C だけをテストする場合にエラーが発生すると、変更 C、B、および A のどこにエラーがあるかは分かりません。

QA 環境は、制御された統合環境です。この環境では、QA チームがどの変化が発生するかを管理しています。対照的に、開発環境では、いつでも変更が発生する可能性があります。ここではビルドは、アプリケーションとやり取りするサービスと統合されます。

この環境では、本番環境と同じインフラストラクチャとアプリケーションを使用できます。ここでは、本番データに十分近いテスト用の本稼働データの代表的なサブセットを使用しています。QA エンジニアまたはテスト担当者は、これらのテストをフォーカスする内容を理解する必要があります。小さな変更を隔離された環境でテストしてバグを特定できるように、このステージでは手動でデプロイすることをお勧めします。QA 環境が本番環境に近づくほど、テスト結果に対する信頼性は高まります。

## ステージング環境

これは、本番稼働前の最後のテスト環境です。ステージングの目的は、本番環境とほぼ同じ環境にすることです。ステージングに何かをデプロイし、それが機能する場合、そのバージョンは本番環境で失敗して停止することがないことが合理的に保証されます。すべての環境は、潜在的な問題を検出するのに役立ちますが、ステージングは最終的な確証を得るためのチェックとなります。ここでは、本番環境とほぼ同量のデータを保持することが重要です。これにより、負荷テストを実行し、本番環境でアプリケーションのスケーラビリティをテストできます。

本番環境向けのコードをこの環境に展開する必要があります。繰り返しますが、ステージング環境は本番環境とほぼ同じです。インフラストラクチャ、データベース、および外部サービスとの統合は、本番環境とまったく同じである必要があります。メンテナンスとビルドには費用がかかりますが、より高価となるのは、メンテナンスもビルドも行わず、生産を中断する場合です。スケールダウンは可能ですが、セットアップと構成は同じにする必要があります。

これは、実装、移行、構成、およびビジネス要件をテストする本番前の最後のゲートになります。そのため、ステージングに進む前にビジネス部門の承認を取得しておくことを強くお勧めします。承認を取得しておかないと、変更に多額の費用がかかります。開発中は、製品または要求された機能のオーナーが、自分がビルドしている内容を完全に理解していることを確認してください。同じ認識を常に共有するようにしてください。

ローカル環境にコードを実装しているすべての開発者は、本稼働環境にコードがどのようにデプロイされるかを可視化できる必要があります。コードがどのようにデプロイされているかを認識することによって、開発者は後の段階に進む前に可能な限りエラーを低減できるようになります。

# ソフトウェアのテスト方法、パート I：モック、スタブ、およびコントラクトテスト

このセクションでは、各テストレイヤーを支援する**モックとスタブ**の手法、および**コントラクトテスト**について説明します。上記のテストピラミッドをもう一度参照してください。テストピラミッドは、テストの種類と、各テストを実施すると有利になる場合の違いについて理解するのに役立ちます。

前述したように、単体テストまたはコンポーネントテスト（ピラミッド下部に表示）は、安価にすばやく実行できます。これらのテストは非常に重要です。これらのテストをできることを確認したら、統合や UI レイヤーテストなど、時間とリソースを集中的に使用するテストに進みましょう。

## モッキングとスタブ

多くの人は、モックとスタブは単体テストとコンポーネントテストにのみ使用されると考えていますが、モックオブジェクトまたはスタブを他のテストレイヤーでも使用できる方法をお見せします。

いくつかの定義を確認することから始めましょう。

**モック**とは、実際のサービスの代用となる外部または内部サービスの偽のバージョンを作成することを意味し、テストを迅速かつ確実に実行できるようにするものです。実装で機能や動作ではなくオブジェクトのプロパティとやり取りする場合、モックを使用できます。

スタブは、モックと同様に、代用を作成することを意味しますが、スタブはオブジェクト全体ではなく動作のみを模倣します。これは、実装がオブジェクトの特定の動作に対してのみ相互作用する場合に使用されます。モックとスタブの違いの詳細については、Martin Fowler 氏が執筆した記事「Mocks Aren't Stubs（モックはスタブではない）」をご覧ください。上記のピラミッドのすべてのレベルでテストを改善するためにこれらの方法を適用する方法について説明しましょう。

# 単体テスト＋コンポーネントテストにおけるモックとスタブ

## 外部機能のモック

コードでシステムコールやデータベースへのアクセスなどの外部との依存関係を使用する場合は、モックまたはスタブを使用することをお勧めします。たとえば、テストを実行するたびに、実装した機能を実行する場合があります。そのため、削除や作成の機能があると、ファイルの作成またはファイルの削除が許可されます。この作業は効率的ではなく、作成および削除されるデータは実際には役に立ちません。さらに、毎回手動で作成したファイルを削除する必要があるため、クリーンアップにもコストがかかります。このような状況では、モックとスタブが非常に役立ちます。

モックとスタブを使用して外部機能を模倣すると、独立したテストを作成できます。たとえば、テストで /tmp/test_file.txt にファイルが書き込まれ、テスト対象のシステムがそのファイルを削除するケースを考えてみましょう。

テストが独立していないことが問題ではなく、システムコールに多くの時間がかかることが問題です。この場合、ファイルシステムコールの応答をスタブにすることができます。応答がすぐに返されるため、時間がかかりません。

もう 1 つの利点は、複雑なシナリオでも簡単に再現できることです。たとえば、ファイルシステムで発生する可能性のある多くのエラー応答をテストしてから、実際に条件を作成する方がはるかに簡単です。破損したファイルのみを削除する場合を考えてみましょう。破損したファイルを書き込むことはプログラム的に難しい場合がありますが、破損したファイルに関連するエラーコードを返すことは、スタブが返すものを変更するだけで簡単に実行できます。

次のサンプルコードを参照してください。

```
def read_and_trim(file_path)
    return os.open(file_path).rstrip("\n")
```

*このメソッドは、システムコールを呼び出して、指定されたファイルパスからファイルを検索し、そこからコンテンツを読み取り、改行を削除します。*

上記のコードは、指定されたファイルパスからファイルを実際に検索するシステムコールと対話する Python の組み込みオープン関数と対話します。つまり、その関数のテストをいつでもどこでも実行できます。

1. テストで検索されるファイルが存在することを確認する必要があります。このファイルが存在しないと、テストは失敗します。

2. テストは、システムコールの応答を待つ必要があります。システムコールがタイムアウトになると、テストは失敗します。

上記の両方の失敗は、実装でジョブを実行できなかったことを意味するわけではありません。現在、これらのテストはシステムコールの応答に依存しているため分離されておらず、システムコールの接続で、要求と応答の配信に時間がかかるため、効率的でもありません。

上記の実装のテストコードは次のようになります。

```
from unittest.mock import patch

content = "fake file content\n"
trimed_content = content.rstrip("\n")
@patch("builtins.open", new_callable=mock_open, read_data=content)
def test_read_trim_content(self, mock_object):
    file_path = "/fake/file/path"
    self.assertEqual(read_and_trim(file_path), trimed_content)
    mock_object.assert_called_with(file_path)
```

ここでは、Python のモックパッチを使用して、組み込みのオープンコールを模倣しています。このようにして、実際に構築した内容だけをテストしています。

ユニットテストでモックとスタブを使用するもう 1 つの好例は、データベースコールの模倣です。たとえば、関数がデータベースからエンティティを削除するかどうかをテストする場合を考えてみましょう。最初のテストでは、手動でファイルを作成し、削除するファイルが存在するようにします。このテストは合格します。しかし、2 回目のテストでは、他のユーザー（自分ではない）は、エンティティを手動で作成する必要があることを知りません。このテストは失敗します。エンティティを作成する必要があることを知らなかったため、削除するファイルが存在していません。そのため、これは独立したテストではありません。

このような場合、データを変更したり、ファイルを削除するオペレーティングシステムコールを作成したりすることは避けることが推奨されます。これにより、別のユーザーが誤ってテストデータを作成できなかった場合、テストが不安定になることを防止できます。

## 内部関数のモックとスタブ

モックとスタブは、単体テストに非常に便利です。これらは、機能や実装を個別にテストするのに役立ちますが、単体テストを効率的かつ安価に維持するためにも役立ちます。

実装が別のメソッドやクラスやり取りする場合、単体テスト / コンポーネントテストでモックとスタブを効果的に応用できます。クラスオブジェクトをモックにしたり、実装がやり取りするメソッドの動作をスタブにしたりできます。他の機能またはクラスをモックやスタブにし、実装ロジックだけをテストすることが、単体テストの主要な利点であり、単体テストを実行することの最大の利点を得る方法となります。

## 統合テストのモック

統合テストでは、サービス間の関係をテストします。すべての依存サービスをテスト環境で起動して稼働させる方法もありますが、これは不要です。自分が管理していないサービスでは、多くの障害が発生する恐れがあり、これらの障害によって、テストで多くの時間が必要となったり、テストが複雑化したりする恐れがあります。モックとスタブを使用していくつかのサービス統合テストを作成し、テストの範囲を絞り込み、テスト全体の信頼性を高めます。

統合テストでは、ルールは単体テストとは異なります。ここでは、自分が編集できる実装と機能のみをテストする必要があります。モックとスタブはこの目的に使用できます。まず、どの統合が重要かを特定します。その後、モックにできる外部サービスや内部サービスを決定できます。

以下の例のように、コードが GitHub API とやり取りする場合を考えてみましょう。リクエストコールから GitHub API が応答する方法は個人では変更できないため、テストする必要はありません。予想される GitHub API の応答をモックにすることで、内部コードベース内における相互作用のテストに注力できます。

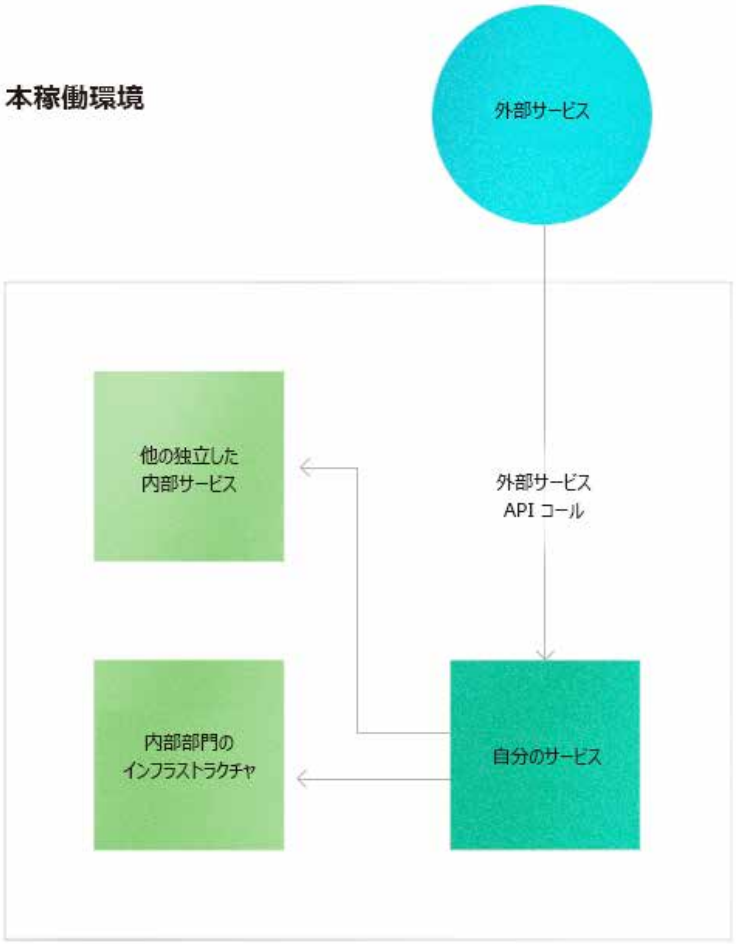
```
@unittest.mock.patch('Github')
def test_parsed_content_from_git(self,
mocked_git):
   expected_decoded_content = "b'# Sample Hello World\n\n> How to run this app\n\n- installation\n\n dependencies\n"
   mocked_git.get_repo.return_value =
expected_decoded_content
   parsed_content = read_parse_from content(repo='my/repo',
file_to_read='README.md')
   self.assertEqual(parsed_content['titles'], ['Sample Hello World'])
```

上記のテストコードでは、**read_parse_from_content** メソッドは、GitHub API コールから JSON オブジェクトを解析するクラスと統合されています。このテストでは、2つのクラス間の統合をテストしています。

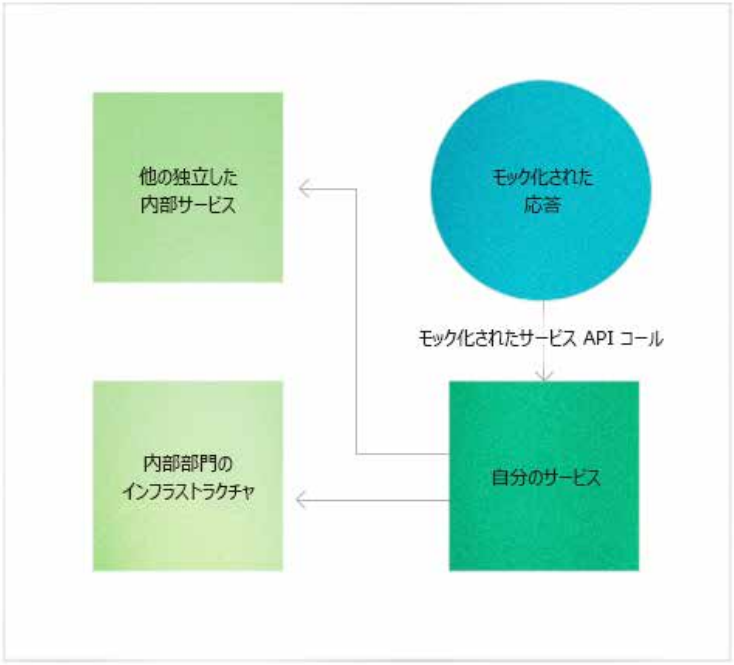
上記のテストではモックを使用しているため、GitHub API を呼ぶことを回避することで、テストの速度が向上し、外部サービスとの依存度が軽減されます。これにより、テストの実行環境でインターネットアクセスが不要になるため、時間と労力が軽減されます。ただし、依存する外部サービスをモックにしながら信頼性の高いテストを実施するには、外部との依存関係が実際の環境でどのように動作するかを把握することが非常に重要となります。

たとえば、上記のコード例の **expected_decoded_content** が GitHub がレポジトリファイルのコンテンツを返す正しい方法ではない場合、モックを使用したテストの誤った仮定が、予期しない障害につながる恐れがあります。モック化した応答を含むテストを作成する前に、外部との依存関係コールの実際のスナップショットを作成し、それをモック化した応答として使用することをお勧めします。スナップショットを使用してモックにした応答を一度作成すれば、アプリケーションプログラミングインターフェイスは多くの場合、下位互換性があるため、頻繁に変更する必要はありません。ただし、偶発的な予期しない変更に備えて、API を定期的に検証することが重要です。

**本稼働環境**


外部サービス
他の独立した
内部サービス
外部サービス
API コール
内部部門の
インフラストラクチャ
自分のサービス

**テスト環境**


他の独立した
内部サービス
モック化された
応答
モック化されたサービス API コール
内部部門の
インフラストラクチャ
自分のサービス

## コントラクトベースのテストにおけるモックとスタブ（マイクロサービスアーキテクチャ）

2 つの異なるサービスが相互に統合されるときには、それぞれのサービスに「期待」、つまり何を提供し、その代わりに何を得るのかについて、標準が設定されることになります。これらは、統合されたエンドポイント間のコントラクト（契約）と考えることができます。この標準により、コントラクトテストを使用して統合をテストできます。

例を見てみましょう。バージョンのタグが付けられる API は、頻繁に変更されるべきではなく、可能であれば一度も変更されるべきではありません。選択した API については、通常、その API で期待される機能など、その API に関するドキュメントを見つけることができます。特定のバージョンの API を使用することに決めた場合、その API コールが戻す内容を利用できます。これは、API を提供するエンジニアとそのデータを使用するエンジニアとの間で推定されるコントラクトです。

コントラクトの概念を使用して、内部サービスをテストすることもできます。マイクロサービスアーキテクチャを使用して大規模なアプリケーションをテストする場合、システム全体とインフラストラクチャをインストールするコストが高くなる可能性があります。このようなアプリケーションでは、コントラクトテストを使用すると、大きなメリットが生まれます。テストピラミッドでは、システムのコントラクトテストの対象範囲に応じて、単体テスト / コンポーネントテストのレイヤーと統合テストレイヤーの間にコントラクトテストは配置されます。エンドツーエンドまたは機能テストを完全にコントラクトテストに置き換えている組織も存在します。

コントラクトベースのテストは、次の 2 つの重要事項に対応します。

1. コントラクトがあるエンドポイントの接続性を確認する。

2. 特定の引数を使用してエンドポイントからの応答を確認する。

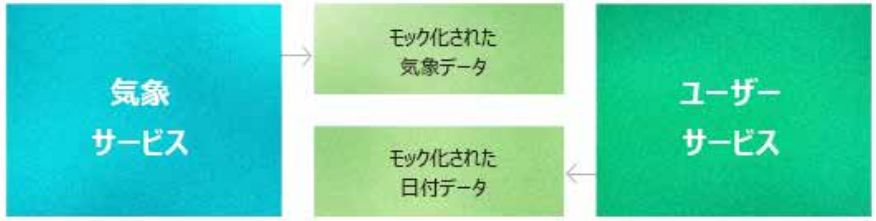
例として、ユーザーサービスと連携する気象サービスがある気象レポートアプリケーションについて考えてみましょう。ユーザーサービスが日付（要求）を使用して気象サービスのエンドポイントに接続すると、ユーザーサービスは日付データを処理してその日付の天気を取得します。これら 2 つのサービスにはコントラクトがあります。気象サービスは、エンドポイントを常にユーザーサービスからアクセスできるように維持し、ユーザーサービスが要求している有効なデータを同じ形式で提供します。

それでは、コントラクトテストでモックとスタブを利用する方法を見てみましょう。テストで気象予報サービスを実際に呼び出すコールの代わりに、モック応答を作成できます。2 つのサービス間にはコントラクトがあるため、エンドポイントと応答は変更されません。これにより、テスト中に両方のサービスが相互に依存することがなくなり、高速で信頼性の高いテストが可能になります。

異なるコンフィグの異なる環境で同じテストを実行すると便利な場合があります。コントラクトテストは、後者の場合の優れた例の 1 つです。コンフィグが異なるさまざまな環境でコントラクトテストを実行すると、さまざまな目標を達成できます。Dev や CI などの下位レイヤー環境の場合、モックを使用するコントラクトテストを実行すると、環境の制約がある中で内部実装をテストする目的に役立ちます。ただし、QA やステージングなどの上位レイヤー環境にアクセスする場合、実際の外部依存サービスとの関係を使用して、モックコントラクトなしで同じテストを使用できます。Mbtest は、上記で説明したタイプのコントラクトテストとモック応答を支援するツールの 1 つです。

**本稼働環境**


**テスト環境**




モックとスタブを使用したさまざまなテストレイヤーの例を見てきました。では、モックとスタブが有用である理由をおさらいします。

1. モックとスタブを使用したテストは、外部サービスに接続する必要がなく、高速で実行できます。外部サービスの応答を待機する必要はありません。
2. テスト範囲を柔軟に設定して、制御および変更できる部分のみを対象にできます。外部サービスを使用すると、外部サービスが不正であったり、テストが失敗したりする場合、テストの意味が失われることになります。モックを使用することで、実際に実行できる作業範囲をテストでき、自ら修正できない問題を抱え込まないようにします。
3. 外部 API コールのモックを使用することで、テストの信頼性が向上します。
4. コントラクトテストにより、サービスチームはより自律的な開発が可能になります。

次に、テスト駆動開発（TDD）およびビヘイビア駆動開発（BDD）の原則について説明し、機能テストからユニットテストまで、すべてのテスト結果をどのように改善できるかを見ていきます。

# ソフトウェアのテスト方法、パート II：TDD および BDD

これまでは、モックとスタブとは何か、さらに、さまざまなテストシナリオでモックとスタブを使用して、テストの柔軟性を高め、テストを高速化し、問題を切り分けながら効果的にテスト全体を進める方法について説明してきました。

次に、テストを念頭に置いてソフトウェアを開発する 2 つの方法であるテスト駆動開発（TDD）とビヘイビア駆動開発（BDD）について説明します。これらの手法を使用すると、ソフトウェア開発に関する考え方が改善され、テストの有効性が大幅に向上します。では、詳しく説明していきましょう。

## TDD：テスト駆動開発

TDD（テスト駆動開発）は、単体テストを作成する方法として知られています。機能テストからユニットテストに至るまで、TDD の原則を使用することについて説明します。

### Red（レッド）-Green（グリーン）-Refactor（リファクタ）：テスト駆動開発の原則

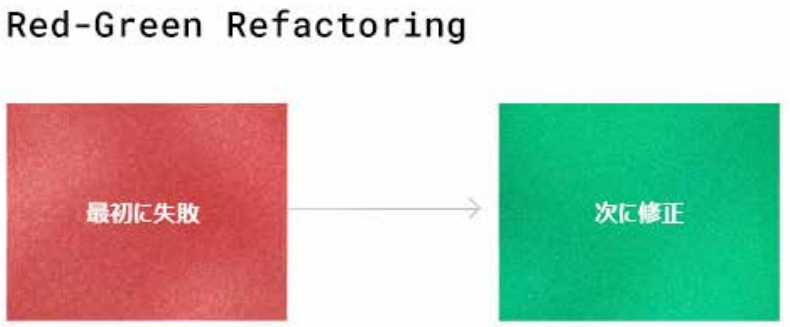
TDD では、コードを実装する前に設計します。そのため、TDD を使用する場合、コンポーネントを記述する前にコンポーネントのビヘイビアを考慮する必要があります。また、これはデリバリーしようとしている対象に注力するのにも最適な方法です。TDD では、メソッドまたは実装のテストを作成して、実装で求められる機能をテストします。

TDD では、テストは最初に必ず失敗します。コードをまだ書いていないので、機能はありません。これは良いことなのです。実装が無効な場合に、テストが合格しないことを証明しています。

次に、実装が有効で目的を達成している場合に、テストに合格することを証明します。実装が正常に機能しない場合にテストが失敗し、実装が正常に機能する場合に合格することを確認したら、コードをリファクタリングし、コードを明確にクリーンにすることができます。テストを実行したばかりであるため、リファクタリングが極めて簡単であり、コードのビヘイビアを正しく変更したかどうかをテストで確認することが可能です。これは、**red-green refactoring** と呼ばれます。

**Red：**最初に、テストを失敗させます。**Green：**次にテストを合格させます。

**最初にテストし、その後でリファクタリングします。**これにより、コードがクリーンになり、本番環境で実行する準備が完了します。完全なコードを作成にする前に、実際に Red と Green を通過することが重要です。Red ステージでは、テストが常に合格しないことだけを確認します。より複雑になったときに、後で、決定論的であ




ると確信できます。後の Green ステージでは、「最適なコードを作成するにはどうすればよいか？」ではなく、「要件を満たすコードを作成するにはどうすればよいか？」という課題に注力します。これは、テストが有効なユースケースに合格していることを証明するステージと考えてください。

次の段階であるリファクタリングでは、コードを再検討して変更します。リファクタリングのステージでは、さらにクリーンでインテリジェントなコードを記述し、改善できます。エンジニアは最初の段階からコードの記述を始めてしまい、本来デリバリーするべきであった内容に注力できなくなることが多くあります。また、エンジニアが実装する機能を作成すると同時にテストを作成し、予期しないバグを作り出す場合もあります。TDD はこれらの間違いを避けるために有用です。

作成するテストは次のようになります。

```
describe('sum()', function () {

    it('should return the sum of given numbers', function () {
      expect(simpleCalculator.sum(1,2)).to.equal(3);
      expect(simpleCalculator.sum(5,5)).to.equal(10);
    });
  })
```

1. **Red**：現在、実装する機能は空です。まだ実装していないため、テストは失敗します。テストが決定論的であることを確認する必要があります。テストがいつ失敗するか、また、いつ合格するのかを伝えることができる必要があります。

```
var Calculator = function () {
    return true // implementation goes here
}
```

2. **Green**：機能を実装します。テストを合格させます。ここで、テストに合格するコードを記述します。

```
var Calculator = function () {
    return{
          sum: function(number1, number2){
                return number1 + number2;
          }
    };
}
```

これで、機能を実装したため、これでテストの条件は満たされます。

**コードのリファクタリング**：次に、コードをリファクタリングして、より明確で読みやすくします。最初の 2 つのステップでコードが明確で信頼できるものとなり、テストを信頼でき、コードのビヘイビアを偶然変更する心配がなくなります。 Red と Green のステージをテストが通過してコードの機能が検証されたことが重要です。コードがリファクタリングステージに入ったら、テストを変更しないでください。ここで変更すると、ソースコードで機能の問題が発生する可能性が高くなります。 テストを変更する必要がある場合は、必ず Red/Green のステージでテストを変更するようにします。

有効なテストを設定したら、コードをさらにクリーンに、スタイルやクラス全体に適合するようにリファクタリングできます。

## すべてのテストレイヤーへの TDD の適用

上記の例は単体テストに関するものでした。では、テストピラミッドの他のレイヤーで TDD を使用するにはどうすればよいでしょうか？


機能を実装するときには、最初に UI レイヤーのテストから始めます（BDD を使用しますが、BDD については次のセクションで説明します）。これらの UI テストは長期間合格することはありませんが、テストから取り組みを開始することで、実際、何をビルドしようとしているのか、また、バックエンドコードがフロントエンドレイヤーとどのように対話するかにフォーカスできるようになります。このアプローチにより、開発者は記述する前に実装する機能を設計できます。

そこから、作業に応じて、単体 / コンポーネントテストまたは統合テストの作業を開始します。コードベースを詳細に決定する前にアーキテクチャの設計が明確であれば、統合テストの作成を開始します。これらのテストもしばらくは失敗します。作成している時点では、これらのテストは完全ではないかもしれませんが、何をビルドしようとしているのか、そして、初期設計はどうなるかを考えるのに役立つ機能となります。

単体 / コンポーネントテストに移行すると、単体テストレイヤーで最終的に Red-Green- リファクタリングステージを開始し、UI レイヤーと統合レイヤーを「Red」ステージのままにして、単体テストをリファクタリングステージまで完了します。次に、統合テストに戻り、テストを Green にしてからリファクタリングします。その後、同じ手順が UI テストに適用されます。

ご覧のように、テストのすべてのレイヤーで TDD の原則が適用されています。原則は同じですが、スケール（規模）だけが異なります。

# BDD：ビヘイビア駆動開発

## ユーザージャーニーストーリー、前提（Given）、条件（When）、結果（Then）

新機能の要望があると、企業の製品部門の担当者は、ユーザーストーリーやユーザー受け入れ基準など、ストーリーレベルのタスクをエンジニア向けに作成します。この方法により、エンジニアは、企業に対する価値を把握でき、ユーザーの観点から実装する機能について考えることができます。ユーザーストーリーを見ることで、エンジニアは作業範囲をより明確に理解できるようにもなります。

ユーザー受け入れテスト（UI 駆動テスト）は、このユーザー受け入れ基準とユーザーストーリーに基づいています。UI 駆動テストでは、通常、稼働しているサイトでユーザージャーニーをテストするのに役立つ Selenium や Cucumber などの自動テストツールを使用します。

ユーザージャーニーストーリーとは、ユーザービヘイビアを示します。開発者は、提供されたビジネス要件を使用して、ユーザーがこの新しい機能をどのように使用するかというシナリオを考えることができます。そして、これらのシナリオを使用してテストを作成できます。これは、ビヘイビア駆動開発（BDD）と呼ばれます。

BDD は、UI 駆動テストで広く使用されている方法です。BDD は、「前提、条件、結果」という構造で作成されます。

**前提：**ビヘイビア / アクションを受け取るシステムの状態

**条件：**最終的に発生し結果を引き起こすビヘイビア / アクション

**結果：**その状態のビヘイビアによって引き起こされた結果

最初にユーザージャーニーとユーザーのビヘイビアについて考えることは良いアイデアです。これにより、機能を実装するときに、ユーザーがどのように機能を利用するのかを検討できます。

以下に例を示します。Cypress の簡単なテストコードの例を次に示します。

**シナリオ：**ユーザーがサイトにサインアップする

(Cypress orb を使用すると、このツールと簡単に統合できます)

**条件：**ユーザーがサイトにアクセスした

**結果：**ユーザーがサインアップボタンをクリックした

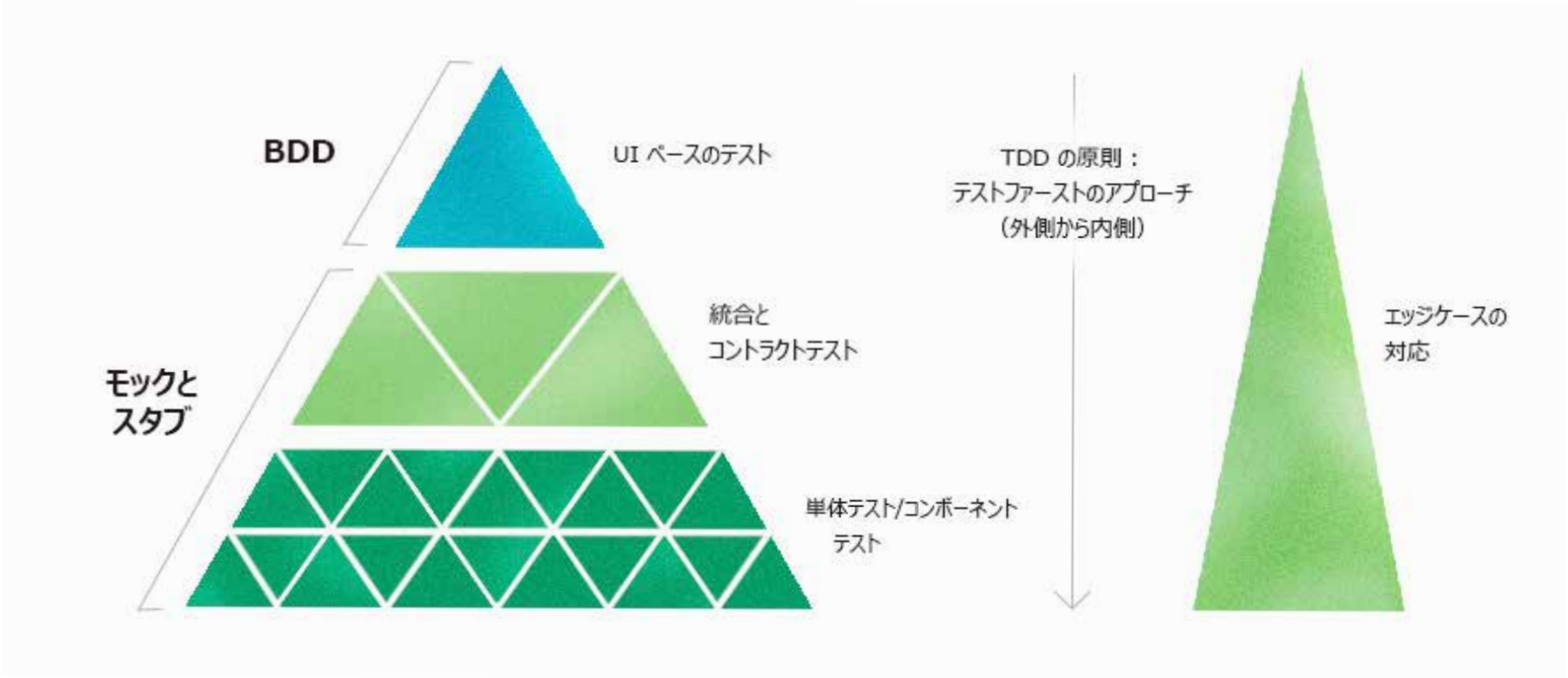
**次に、**ユーザーがサインアップページにアクセスできることを確認します

```
describe('User can signup to the test-example site', function() {

  it('clicking "signup" navigate to a signup url', function() {
    // Given
    cy.visit('https://test-example.com/')
    // When
    cy.contains('signup').click()
    //Then
    cy.url().should('include', '/signup')
  })
})
```

UI レイヤーのテストで BDD を使用することは、ユーザーがやり取りするアプリケーション領域が含まれるため、合理的です。他のテストレイヤーは、BDD の使用は適していません。BDD を使用した UI レイヤーテストは、高品質のソフトウェアを構築する非常に重要なプロセスになりますが、これは非常に高価で非効率的でもあります。

前に述べたように、さまざまなレイヤーと種類のテストを利用することは、何かがうまくいかない場合に、何が失敗しているかを正確に特定でき、迅速に修正できるようになります。これにより、デバッグ時間を短縮し、下位レベルの問題をより安価かつ迅速に検出できるようになります。



## 結論：ハッピーパスとエッジケース

テストを作成するとき、すべてがうまくいったときに何が起こるかについて考えることは簡単です。エッジケースについて考えることは、エンジニアにとって困難な場合があります。これは想定内です。

UI レイヤーのテストについて考えるとき、サイトのほぼ全体がすでに稼働中であり、テストはこの完成したサイトに対して実行されていると想定しています。現在、ユーザーが実行する可能性があるすべてのシナリオを想定することは困難であり、可能性のあるすべてのシナリオをテストする場合、そのコストも膨大になります。したがって、**ハッピーパスと主要な障害パスにフォーカスすることは良い習慣です。これらは、主なビヘイビアとワーストケースシナリオの両方に対応できます。**多くの場合、エッジケースのバグは、QA のような環境で QA 探索的テストで検出されます。このプロセスでは、QA はビジネスリスクを分析し、エッジケースシナリオをエンジニアに伝え、コードが本番稼働する前にバグを修正し、エッジケースシナリオをカバーする新しいテストを作成します。

エンジニアがシステムと状況をより明確に理解できれば、エッジケースについて考えることも容易になります。これにより、エッジケースのテストはテストピラミッドの下層でより適切にカバーされることになります。これは、より効率的であるため、常に望ましい方法です。とはいえ、テストの維持もエンジニアの業務の一部です。コードが進化したら、テストも変更する必要があります。有用なビヘイビア駆動テストを行うことは、実施するテストの数よりも重要です。テストの最終的な目的は、本番環境に高品質のソフトウェアを提供することです。

コードベースをよりテストしやすくすることは価値のある投資であり、長期的にビジネスとソフトウェアを拡張するのに役立ちます。この基本的な作業により、ソフトウェアを本番環境にリリースする工程を最適化でき、効率的にデプロイできている自信が持てるようになります。

# 結論

DevOps をいつどのように使用するかを学び、テストへのインプットとアウトプットをマスターすることは、最高品質のソフトウェアを確実にデリバリーするために大いに役立ちます。また、無駄な時間とリソースを削減し、効率性と結果の両方を最大化できます。ソフトウェアデリバリーをさらに高速かつ堅牢にする方法の詳細については、circleci.com をご覧ください。