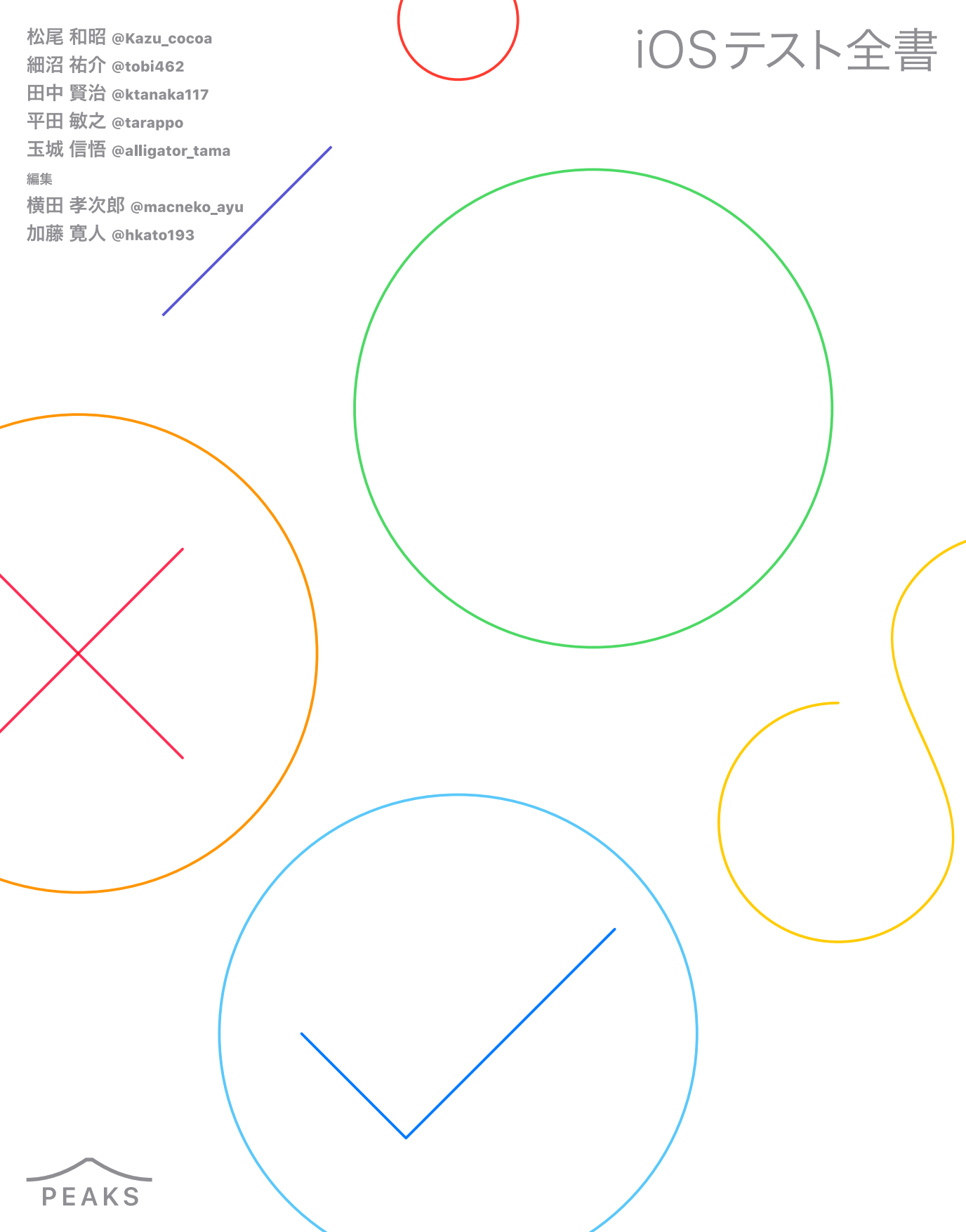
iOSテスト全書

松尾 和昭 @Kazu_cocoa
細沼 祐介 @tobi462
田中 賢治 @ktanaka117
平田 敏之 @tarappo
玉城 信悟 @alligator_tama
編集
横田 孝次郎 @macneko_ayu
加藤 寛人 @hkato193



PEAKS

はじめに

iOS アプリを App Store 経由で配布することができるようになった 2008 年から、約 10 年が経過しました。

この間に iOS アプリで実現できることは着実に増え、今や生活において欠かせないものとなっています。その iOS アプリを開発する環境も変化し続け、テスト周りにおいても進化してきています。

多様化されたアプリや端末により、iOS アプリ開発を行う上で自動テストの重要さは増しています。Apple も WWDC でテストピラミッドに触れるなど、テストの重要さについて語るようになってきました。

自動テストに関する書籍は増えてきていますが、iOS アプリ開発におけるテストについての情報はあまり世に公開されておらず、まとまった形ではないといってもいい状況です。また、公式のドキュメントは情報が少なく、どのように自動テストをするべきか試行錯誤されているのではないでしょうか。

そこで、iOS アプリ開発におけるテストの入門書として、テストに関する内容を幅広く扱った『iOS アプリ開発 自動テストの教科書』を執筆しました。そして、入門書のさらに一歩先の情報も必要だろうと考え、本書を企画しました。本書の企画をすすめるにあたり、iOS アプリ開発におけるテストや CI ／ CD に対して知見を持っている方々に執筆を依頼し、本書の執筆がスタートしました。

本書では、iOS アプリ開発におけるユニットテストや UI テスト、それらを活用するための CI ／ CD に関する情報まで、幅広くカバーしています。また、公式のテスティングフレームワークだけでなく、サードパーティのテスティングフレームワークも扱っています。各章は単なる公式情報のリファレンスなどの内容に留まらず、著者がそれぞれ培ってきた知見をもとに実践的に利用できる内容に仕上げています。

iOS アプリ開発に関わる人であれば、なにかしら知見が得られるような一冊になっていると思います。本書から得た知識をもとに実際のプロジェクトで実践をしてみてください。そして、そこで得た活きた知見を是非広めてくれれば、著者としては嬉しい限りです。

（著者代表 平田 敏之／@tarappo）

対象読者

本書は iOS アプリ開発のテストに関心のある次のような方を対象としています。

- テストについて興味はあるがどうしたらいいかわからない方
- もっとテストを書くためのヒントが欲しいと悩んでいる方
- テストにもっと強くなりたい方
- CI ／ CD サービスをもっと活用したいと思っている方

本書の手引き

本書は全部で 9 章構成になっています。大きく分けると「テスト自動化入門」「ユニットテスト」「UI テスト」「CI ／ CD」という 4 部構成になっています。

第 1 章では、自動テストをおこなっていく上で知っておくとよいことを、より深く学びたいときに参照できる情報とあわせて説明しています。まずこれらを知った上で、どのように自動テストを取り組んでいくかを考えていくことが重要です。

第 2 章〜第 5 章では、ユニットテストをテーマにしています。「アサーション」や「テストダブル」などといったユニットテストを行う上で大切な基礎知識から、XCTest や Quick ／ Nimble といったフレームワークの知見と、実際にどのように開発していくかまで知ることができます。

第 6 章〜第 8 章では、UI テストをテーマにしています。UI テストを扱う上で知っておくと効果的なことを、「導入前」「実装中」「運用中」といったフェーズに分けて説明してから、広く使われる XCUITest とサードパーティ製の Appium について、フレームワークを利用する上での知識を得ることができます。

最後に第 9 章で、CI ／ CD をあつかっています。これまでの章で紹介した内容を活用しつつ、プロジェクトのリリースをすばやく、そして確実・安全におこなうための内容で締めています。

本書は第 1 章から順に読んでいただくことを想定しています。多少の前後関係はありますが、上記のまとまりや各章単位で読むこともできます。

想定環境

本書では表 1 の環境で動作確認をしています。

●表 1　想定環境

Xcode	11.0
macOS	Mojave

利用しているライブラリのバージョンについては各章に記載しています。

クラウドファンディングと PEAKS

本書は技術書クラウドファンディング・サービスである「PEAKS」のプロジェクトとして開始され、926 人もの支援者のサポートによって作られました。出資者特典である「アーリーアクセス」でいただいたご意見も反映されております。

PEAKS ではこんな本を作りたい！という方を募集しています。次の窓口からご連絡いただければ幸いです。

- https://peaks.cc/requests

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第 1 章
テスト自動化入門

松尾 和昭／@Kazu_cocoa

1.1 はじめに

「テスト全書」ということで、みなさんはどのような「テスト」を思い浮かべながら本書を手に取られたでしょうか？開発者やテスターなどの役割により、この「テスト」という言葉に対して異なるものを想像することでしょう。昨今、開発に関わる人たちの間で「テスト」という言葉に含まれる意図が異なるために、話が噛み合わない経験があったなど耳にするようになりました。本書はそのような差を調整するきっかけとしたいという声がもととなり、執筆が始まりました。

本章では、まずは本書で扱う「テスト」が主に動的な、自動化されるテストであることを説明していきます。第 2 章からは具体的な**ユニットテスト**に関わる話、第 6 章からは **UI テスト**（ユーザーインタフェーステスト）、第 9 章ではそれらを実行するための **CI**（継続的インテグレーション）／ **CD**（継続的デリバリー）環境に関して説明します。

また、主に iOS クライアント側の話をしますので、通信相手となる API サーバーといったサーバー側の話題には触れません。

1.2 本書におけるテスト

1.2.1 ソフトウェアテストとは

JSTQB の定義[注1)]によると、ソフトウェアテストとは「ソフトウェアテストはソフトウェアの品質を評価し、運用環境でソフトウェアの故障が発生するリスクを低減する 1 つの手段である」です。その中で、テストは動的テスト・静的テストと区別されます。

動的なテストは実際にテスト対象を実行しながら行うテストです。実装されたテストコードを実行したり、手動でテスト対象を操作しながら行ったりする形式を指します。

静的テストとはレビューや静的解析ツールにより、テスト対象を実行することなく行う類のテストを指します。

動的テストは自動・手動と実行形態で分けることも可能です。たとえば手順書を用意し動作確認していく手動テストや、テストコードを実装し実行することで結果を得る自動テストです。第 2 章以降で扱う「テスト」はこの中で特に自動テストを暗に意味します。

注1)　テスト技術者資格制度 Foundation Level シラバス Version 2018.J02

1.2.2 Apple から公開されているテスト

iOS アプリ開発におけるテストは Apple が『**XCTest**』[注2] を公開しています。Apple は全体のテストの指針は明示的には提示せず、XCTest という枠の中でさまざまな区分を説明しています。たとえば『User Interface Tests』[注3] という区分の中では、UI に関わるテストをひとつの主題として扱っています。以前は『About Testing with Xcode』[注4] を公開し、現在とは少し異なる形でテストを説明していました。現在はアーカイブとなっています。

1.2.3 確認のためのテスト

本書の主な対象が動的・自動テストになることに触れました。本項ではテストの役割を少し掘り下げます。『Testing and Checking Refined』[注5] を参考に「**Testing**」と「**Checking**」という言葉の対比を持ち出します。

次の文章は同記事より引用したものです。

> Testing is the process of evaluating a product by learning about it through exploration and experimentation,which includes to some degree: questioning, study, modeling, observation, inference, etc.
>
> Checking is the process of making evaluations by applying algorithmic decision rules to specific observations of a product.

Testing には探索（exploration）や実験（experimentation）という言葉があるように、学びながらプロダクトを評価していく流れと定義しています。Checking は特定のアルゴリズムの決定規則を、プロダクトにおける特定の観測ポイントに適用することで確認・評価する流れを指します。Testing、Checking ともに広く引用される日本語訳はありませんが、大まかな日本語訳を脚注に載せています[注6][注7]。

この言葉を借りると、本書における「テスト」は「Checking」の意味を大きくもつ役割のテストとなります。その中の、特にリグレッション[注8]（ISTQB[注9] によると「変更による構成要素やシステムの品質の低下」[注10] を意味します）に効果を発揮するものが多くを占めます。この区分

注2） https://developer.apple.com/documentation/xctest
注3） https://developer.apple.com/documentation/xctest/user_interface_tests
注4） https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/01-introduction.html
注5） https://www.satisfice.com/blog/archives/856
注6） 「テスティングは、探索や実験をとおして製品を評価するプロセスです。それらの探索や実験は、疑問、学習、モデル化、観測や推測などの試行錯誤を含みます。」
注7） 「チェッキングは、プロダクトの特定の観測点にアルゴリズムを適用することで評価を行うプロセスです。」
注8） 日本語では、デグレードやエンハンスバグと表現されることもあります
注9） https://glossary.istqb.org/en/search/regression
注10） A degradation in the quality of a component or system due to a change.

も議論できる余地はあるものの、本書を読み進める上での範囲の限定という意味では十分かと思います。なお、Checking と Testing は自動・手動のような実行手段の区分ではありません[注11)]。Checking、Testing のそれぞれの範囲で自動・手動と実現可能です。

1.2.4 テストを継続する

『実践的プログラムテスト入門』[注12)]によると、テストという行為は

> 品質保証を支援することにある。すなわち、品質に関連した情報を収集し、プログラマにその情報をフィードバックすることで、過去に発生した不良の再発を防止し、ソフトウェアの品質をよい方向へと導くことである。

とあります。また、『ソフトウェアテスト技法』[注13)]では

> 優秀なプログラマはバグを少ししか作らないのではなく、絶え間なくテストする習慣によってバグを局所化し、これが結果的にコスト低下にも繋がっているのである

という記述もあります。

本書におけるテストもその引用にあるように、第 2 章から第 9 章まで含めてテストを継続して実行できるところまでの話を扱います。

Column. テストをより広く学んでみたい場合に

本書は基本的には動的な自動化可能な類のテストの話をします。その目的も多くは Checking です。一方で、開発全体の中でどうテストを取り入れるかは話として出てきません。

「テスト」を広く学ぶ場合は、『ASTER セミナー標準テキスト』[注14)]が役に立ちます。アジャイルという文脈における話を手に入れたい場合は、『実践アジャイルテスト』[注15)]もよいでしょう。どのような類のテストをリリースまでに実施するかなどの、テスト実装以外の面に関しても学べます。

注 11）『Testing and Checking Refined』においても Chekcing は自動化することが可能とだけ言及しています。Testing はより人の活動によった活動と表現されています。

注 12）『実践的プログラムテスト入門　～ソフトウェアのブラックボックステスト～』　ボーリス・バイザー（著）、小野間 彰（翻訳）、山浦恒央（翻訳）、石原成夫（翻訳）、「1.3.2 ソフトウェアテストの理由づけ」より

注 13）『ソフトウェアテスト技法』（1994）　ボーリス バイザー（著）、Boris Beizer（原著）、小野間 彰（翻訳）、山浦恒央（翻訳）

注 14）http://aster.or.jp/business/seminar_text.html

注 15）『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』（2009）　Janet Gregory（著）、Lisa Crispin（著）、榊原 彰（監修、監修、訳）、増田 聡（翻訳）、山腰直樹（翻訳）、石橋正章（翻訳）

1.3 自動化

テストを継続して実行させていきたい、と前項にて述べました。本項では、それらを継続して実行させるための「自動化」に触れます。

「自動化」に期待できることとして、素早いフィードバック、安全網などの言葉が思い浮かぶかもしれません。繰り返される人の手作業の誤り・失敗を減らすなどもあるかと思います。

一方、何もかもが自動化できるわけではありません。我々が普段開発する場合、ある入力とそれに対する出力を想定して実装を行います。自動化を行う場合でも入力と出力を気にしておく必要があります。より大きな一連の流れを自動化するには、**PFD**（Process Flow Diagram）のように一連の流れを入力と出力で整理することに慣れておく必要があります。そのような要件が見えない場合、役立つ自動化ができないかもしれません。

定義した一連の流れを再現可能な透明性のある環境で繰り返し実施できるように自動化しておくことで、実装内容に関して大きく期待と異なる状態になることが防げます。

1.3.1 手作業による誤り

読者の多くは人による手作業は「注意」している場合も間違えることがあると、経験的に知っていることでしょう。さらに睡眠不足や疲労が溜まっているなどの体調の悪化が重なるとなおのこと顕著になるかと思います。

人は（結果的に）確率的に誤る可能性があります。本項では、そのような事象を『ヒューマンエラー』[注16)]を参考に少し深掘りしてみます。

『ヒューマンエラー』にでてくる **GEMS**（Generic Error-Modeling System）に沿った定義では、失敗には主に実行、もしくは計画が不適切という分け方があります。ソフトウェア開発を例に挙げると、そもそもの仕様やその設計が誤っているという状態が「計画が不適切」、実装を誤った状態が「実行時の失敗」というような対応づけが想像できます。

『ヒューマンエラー』では、人の活動を技術、知識、規則ベースで区分しています。それぞれに対して、「失敗」に繋がる要因を分析しています。「失敗」を観察するため、ある決まった手順をもつ一連の作業を観察していました。

その結果、人はいずれの活動においても手順の飛ばし、抜け、漏れを検出することができなかったようです。つまるところ、人は手順の抜け漏れを探すようなことが苦手で、「気をつける」や「手順を守る」では完全に予防できる類のものではありません。

失敗を検出するため、「自己を監視する」「なんらかの環境の手がかりに気づく」「誰か他の人がそれを発見することが有効で必要である」という言葉が出てきます。読者の身近なソフトウェア開発の話では、レビューというような活動にも繋がる話ですね。

注16）『ヒューマンエラー 完訳版』（2014）　ジェームズ リーズン（著）、James Reason（原著）、十亀 洋（翻訳）

人は警戒状態（vigilance）を有効に発揮し続けられるのは、ごく短時間でしかないことも知られています。滅多に発生しないことを効率的に発見することも人には不向きです。

たとえば、ほとんど開発メンバーが操作することはない一方で重要な場所、本来は変化がほとんどなく安定して実行可能な場所は自動化を進めるよい始点になります。たとえば、「iOS アプリでサービスのメンテナンス発生時の振る舞いが期待どおりできているか」などです。

人の誤りを軽減する手法として、自動化を進めることは有効な手段のひとつでもあります。ただし、『ヒューマンエラー』の中でも言及されていますが、

> 自動化に成功したシステムは、手動で介入する必要はほとんどなくなり、だからこそオペレータの訓練に膨大な投資を必要とする

という側面もあります。

自動化が進み、自動化により予防したいことを防ぐことができるようになります。その自動化された環境が常になると、自動化されたこと柄が盲目的に行われる状態になり、形骸化に向かうことがあります。自動化に関わる人たちは自動化に対する知見を忘れないように、もしくは異なる形で蓄積しておく必要があります。

1.3.2 集中するということ

「1.3.1 手作業による誤り」で、「人は警戒状態を有効に発揮し続けられるのは、ごく短時間でしかないことが知られている」と書きました。

静的テストの話に触れることになるのですが、読者にも馴染みのあるコードレビューを題材にします。『Making Software』[注17)]の 18 章で、

- 人は 45 分から 1 時間程度しか集中できない
- 行数でいうと 200 行付近

が、問題を発見できるレビューという行為の限界と実証されています[注18)]。

人は集中期間を越えると、集中時に見つけることができていた指摘事項を見つけられないことがある、と指摘されています。『TSPi ガイドブック』[注19)]でもこの行数が言及されています。フロー理論を参考にすると、その状態になるまでに 15 分程度の時間が必要ともいわれています。

現在は開発時に使用するプログラミング言語や統合開発環境などの環境的な変化はありつつも、人の集中力というものが飛躍的に変化していることはないでしょう。手作業の負荷や誤りを減らし、持っている資源をより活用するために適切に自動化を進めることは、とても有効に機能していきます。

注17）『Making Software What Really Works, and Why We Believe It』（2010）　Andy Oram（著）、Greg Wilson（著）
注18）集中状態になるための期間やその継続時間には個人差があります
注19）『TSPi ガイドブック』（2008）　ワッツ・S・ハンフリー（著）、秋山義博（監修、監修）、JASPIC TSP 研究会（翻訳）

第 2 章

ユニットテスト（概要）

細沼 祐介／@tobi462

本章では、iOS アプリ開発に限定されない、ユニットテストの一般的な考え方やテクニックについて解説していきます。世には多くのテスティングフレームワーク／ライブラリが存在しますが、全体を俯瞰して見ることで、そうした個々のツールや技法に振り回されにくくなります。

なお説明の都合上、必ずしもユニットテストに限定されない解説を行っている箇所もありますが、そうした箇所については、必要に応じて補足を入れています。

2.1 ユニットテストとは

ユニットテストは、システムにおける最小単位のコンポーネントに対して行うテストのことです。具体的には、構造体やクラス、関数などの単位に対して行います。「ユニットテスト」と表現した際は、**「1. 自動化」され「2. 繰り返し実行可能」で「3. 統一的な仕組み」に支えられたテスト**のことを指します。

2.1.1 テスティングフレームワークはなぜ必要か

iOS アプリ開発において、ユニットテストは XCTest や Quick ／ Nimble といったテスティングフレームワークを利用します。本項ではそれらの必要性について、あらためて考えてみます。

例として、足し算を行う add 関数について考えます（リスト 2.1）。

●リスト 2.1　足し算を行う add 関数

```
func add(_ x: Int, _ y: Int) -> Int {
    return x + y
}
```

この関数の動作が正しいことをテストするだけであれば、アプリケーションのエントリポイント[注1)]などで、一時的に動作確認用のコードを書けば十分です。

次のコードは、アプリ起動時に表示される UIViewController の viewDidLoad メソッド[注2)]において、動作確認用のコードを記述した例です（リスト 2.2）。

注1)　プログラム実行時に開始されるコードのことで、iOS アプリにおいては AppDelegate になります。なお、他の多くのプログラミング言語においては main 関数がエントリポイントにあたります。

注2)　紙面的な都合で AppDelegate ではなく UIViewController を例に解説しています。

●リスト 2.2　アプリの起点となる UIViewController で関数の挙動をテストする

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        // テスト対象の関数を呼び出し、結果をコンソールに出力する
        print("add(1, 1) = \(add(1, 1))")
        print("add(1, 2) = \(add(1, 2))")

        // 本来の処理がつづく...
    }
}
```

このコードを実行するとコンソールに次の出力がされ、関数が正しく機能していることを確認できます。

```
add(1, 1) = 2
add(1, 2) = 3
```

今回のケースでは、動作確認用のコードを記述しましたが、デバッガ（iOS アプリ開発においては LLDB）などを用いて動作確認することも可能です。

これは add 関数をテストしているという意味においては、ユニットテストと表現して差し支えないのかもしれません。しかし、この方法はその場かぎりのもので「繰り返し実行可能」なテストではありません。ソースリポジトリにコミットすることもできず、資産としても残りません。

この状態から一歩進めて、次のようなコードにしたらどうでしょうか（リスト 2.3）。

●リスト 2.3　デバッグビルド時においてテストを自動で実行する

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        // デバッグビルド時のみに有効な動作確認用のコードを記述しておき、
        // 期待する結果が得られなかった場合に`fatalError`で強制終了させる。

        #if DEBUG

        let x = add(1, 1)
        if x != 2 {
            fatalError("add(1, 1)の結果が2ではなかった。(実際の値：\(x)) ")
        }

        let y = add(1, 2)
        if y != 3 {
            fatalError("add(1, 2)の結果が3ではなかった。(実際の値：\(y)) ")
        }

        #endif
```

```
    }
}
```

このコードではプリプロセッサ（#if ／#endif）を利用し、デバッグビルド時のみ動作確認用のコードが有効になるようにしています。そして、関数の「呼び出し結果」と「期待値」を比較し、それらが異なっている場合には fatalError 関数を呼び出すことでアプリを強制終了するようにしています。

このコードはデバッグビルド時以外は無効なため、ソースリポジトリにコミットしておくことも可能です。

そうすることで、アプリ起動時に関数が正しく機能していなければ、fatalError によるメッセージとともにアプリが強制終了し、バグを検出することができます。これは**「1. 自動化」され「2. 繰り返し実行可能」な**状態になっています。

しかし他の多くの関数についても、同様にテストしたい場合に課題が残ります。ここにすべてのテストコードを書くのは現実的とはいえません。また、期待する結果が得られなかった場合にアプリが強制終了する作りなので、ひとつのテストが正しく動作しないだけで、アプリの動作確認ができない状態になります。開発における生産性を考えた場合、極めて非効率なのは間違いありません。

この状況を改善するためには「統一化された」動作確認の仕組みが必要です。

こうした背景から生まれたのが、xUnit ファミリー[注3)]に代表されるテスティングフレームワークです。Xcode に用意された標準のテスティングフレームワークである XCTest を利用すると、リスト 2.3 のコードを次のように記述できます（リスト 2.4）。

●リスト 2.4　テスティングフレームワークを利用してテストを記述

```
class AddTests: XCTestCase {
    func test_足し算が行えること() {
        XCTAssertEqual(add(1, 1), 2)
        XCTAssertEqual(add(1, 2), 3)
    }
}
```

XCTest は第 3 章で詳しく取り扱うので、ここでは説明を割愛しますが、このコードがどのような意味をもつのかは直感的に理解できると思います。

テストの実行および結果のレポートは、テスティングフレームワークによって行われるため、自分で用意する必要がありません。また、テストケースを増やしたい場合も、テスト用のクラスやメソッドを増やすことで対応できます。これによって**「3. 統一的な仕組み」に支えられた**状態になります。

注3)　Java のテスティングフレームワークである JUnit に代表されるものです。数多くのプログラミング言語に移植されており、それらを総じて「xUnit ファミリー」と呼ぶことがあります。ただし、XCTest は公式に xUnit ファミリーであるとは言及されていません。

ここまで、add 関数をテストする方法について、もっとも原始的なやり方から順に見てきました。

1. 一時的なコードまたはデバッガでの動作する方法
2. **「自動化」**されており**「繰り返し実行可能」**だが統一化されていない方法
3. テスティングフレームワークという**「統一的な仕組み」**に支えられた方法

1. の課題は、テストを資産として残せないことにありました。2. の課題は、「自動化」し「繰り返し実行可能」であるが、統一化されていないためスケールしないことにありました。3. では、ユニットテストのための統一化された仕組みを用意することで、これらの課題を解決しました。

ソフトウェア開発においてテスティングフレームワークは一般的なものになっており、あらためて「なぜ」それが必要なのかは語られることは少ないように感じますが、**「1. 自動化」され「2. 繰り返し実行可能」で「3. 統一的な仕組み」に支えられたテスト**のために、テスティングフレームワークが必要とされることが理解できるかと思います。

2.1.2 目的

ユニットテストを行うメリットはいくつか存在しますが、大きく分けて次の 3 つがあります。

1. より早い段階で不具合に気づく
2. コードや設計を継続的に改善できる状態にする
3. API の利用方法をドキュメント化する

それぞれについて解説していきます。

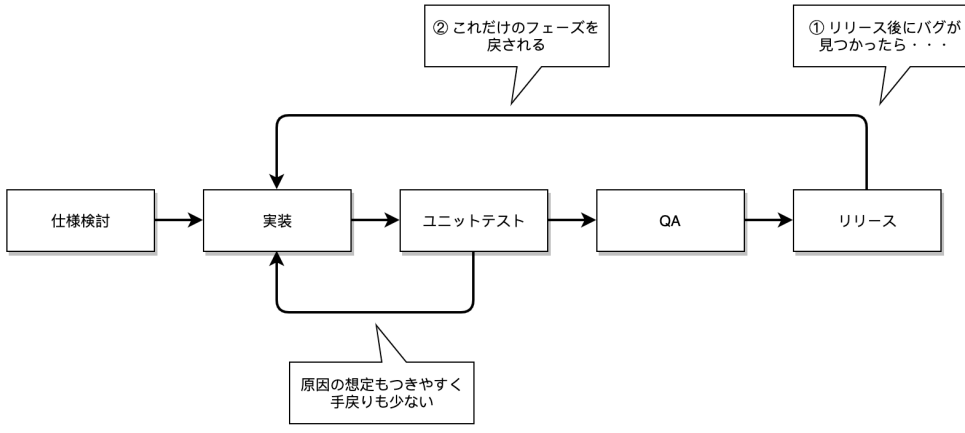
より早い段階で不具合に気づく

ユニットテストを行う最大の利点として、より早いタイミングで不具合に気づけるという点があげられます。一般的に、不具合はより早いタイミングで見つかったほうが手戻りが少なくなります（図 2.1）。

たとえば、リリース前の手動テストにおいて、検索機能における不具合を見つけたとします。その場合、不具合の原因がどこに存在するのか、すぐに見当をつけることは難しいでしょう。原因は、入力コンポーネントとのデータバインディングかもしれませんし、サーバサイド側の API の不具合かもしれませんし、あるいはクライアント側の View 構築に問題があるかもしれません。

もし、それぞれの処理の単位（ユニット）で正しく動くことを検証できていたら、この不具合はずっと早いタイミングで見つかっていたかもしれません。

仮にそこで見落としていたとしても、正しく動くことが保証されているテストコードが存在することによって、テストできていない箇所に原因があると判断できます。



●図 2.1　早いタイミングで不具合に気づけたほうが手戻りが少ない

コードや設計を継続的に改善できる状態にする

ユニットテストを整備しておくことで、それらが安全網として働き、コードや設計を継続的に改善できる状態になります。テストが自動化されていれば、動作確認が簡単に行えるので、コードの可読性向上や設計の改善が安全に行えます。

逆にユニットテストがまったく存在しなければ、テストは毎回手動で時間をかけて行う必要があります。それはとてもコストがかかる作業ですし、心理的にも「面倒」な作業です。

その結果としてコードや設計の改善は行われず、近い将来には修正するのが怖く、読んで理解するのも難しいコードができあがることでしょう。これは素早く継続的なリリースが求められる、現代のソフトウェアにとって致命的となりかねません。そうした状況を作らないためにも、ユニットテストで安全網を整備するのはとても価値があります。

API の利用方法をドキュメント化する

これは前述のケースと似ていますが、ユニットテストを記述しておくことで、対象の API の利用方法が分かる仕様書としての役割を果たします。

API の利用方法はドキュメンテーションコメントなどでも残せますが、実際に動くコードによって API の利用方法が分かるのはメリットとして大きいといえます[注4)]。また、細かくドキュメンテーションコメントを書くと、冗長な文章になりがちで、場合によってはノイズになりかねません。

たとえば、安全な割り算を行う次のコードを見てみます（リスト 2.5）。

注4)　一部のプログラミング言語ではドキュメンテーションコメントと一緒にユニットテストを書けるものも存在します。たとえば Python では doctest（https://docs.python.org/ja/3/library/doctest.html）というものが用意されています。

第 3 章

ユニットテスト（XCTest編）

田中 賢治／@ktanaka117、細沼 祐介／@tobi462

本章では Apple から公式に提供されているテスティングフレームワークである、XCTest について解説します。

また、簡単な XCTest を使ったユニットテストの書き方と、いくつかのユニットテストにおける頻出パターンについても解説します。

3.1 XCTest とは

XCTest は、ユニットテストや UI テストの実行環境を提供する、Swift および Objective-C で利用可能なテスティングフレームワークです。Xcode プロジェクトとシームレスに統合されたユニットテストを作成するために用いられます。

XCTest は、Xcode 上でテストターゲットが特定の条件を満たしているかを検証するために、さまざまな情報を記録します。iOS アプリ開発におけるユニットテストは、これらの情報をもとに設計されていきます。

- テストの成功／失敗
- テストが失敗した理由
- テストの実行時間
- etc

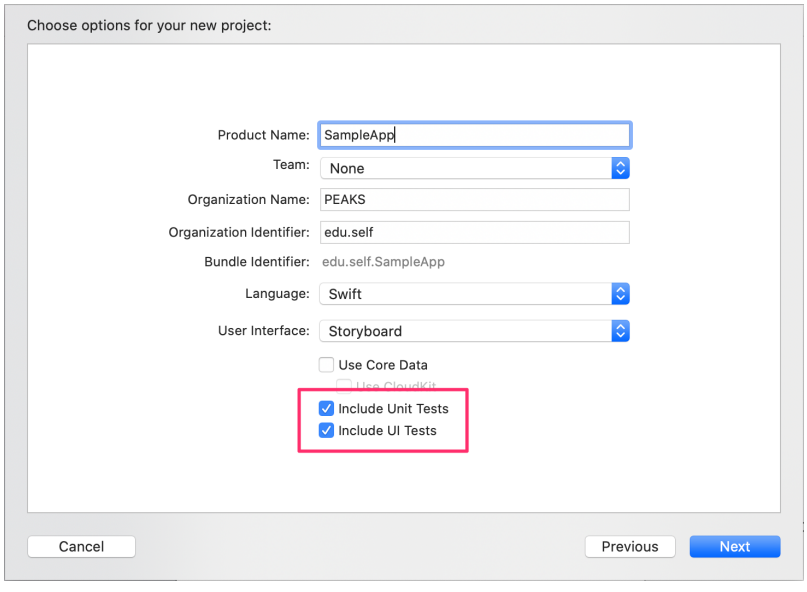
3.2 プロジェクトで利用するための準備

プロジェクトで XCTest を利用するためには、専用のターゲットを追加する必要があります。Xcode では、ユニットテスト用の「**iOS Unit Testing Bundle**」と UI テスト用の「**iOS UI Testing Bundle**」の 2 つが、ターゲット作成時のテンプレートとして用意されています（表 3.1）。

●表 3.1　テスト用のターゲットを作成する際に利用できるテンプレート

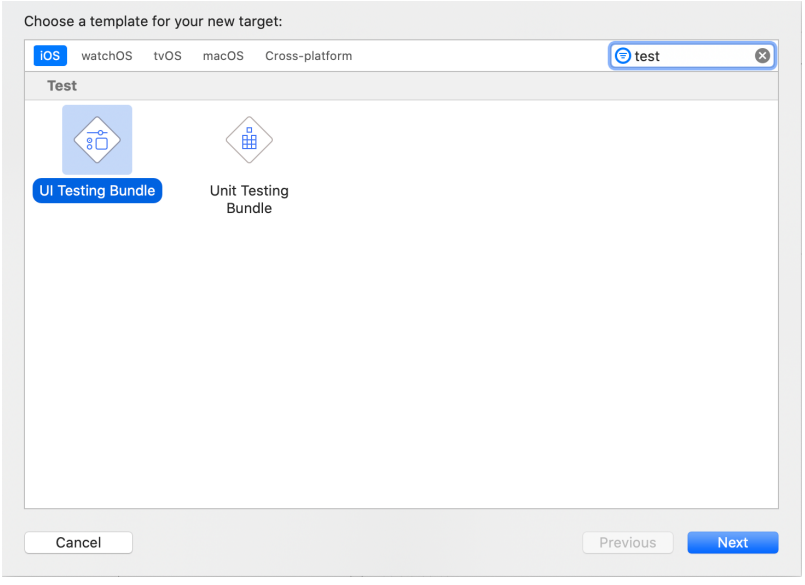
テンプレート名	利用目的
iOS Unit Testing Bundle	ユニットテスト用
iOS UI Testing Bundle	UI テスト用

新規プロジェクト作成時においては、プロジェクト作成ダイアログにおいて「Include Unit Tests」および「Include UI Tests」のチェックボックスを ON にすることで、それぞれのターゲットが追加されたプロジェクトを作成できます（図 3.1）。



●図 3.1　新規プロジェクト作成時にチェックをつけると自動で追加される

また、既存プロジェクトに追加する場合は、ターゲット追加ダイアログでそれぞれのテンプレートを選択してターゲットを作成することで追加できます（図 3.2）。



●図 3.2　ターゲット追加ダイアログでテスト用のテンプレートを選択して追加できる

本章ではユニットテストについて解説していくので、サンプルコードを試す場合は「iOS Unit Testing Bundle」を追加した Xcode プロジェクトを作成しておいてください。

なお、UI テスト用の「iOS UI Testing Bundle」については、XCTest による UI テストを扱った第 7 章で利用していきます。

3.3 最初の XCTest

まずは XCTest を使ってユニットテストを書くイメージをつかんでいただくために、簡単な例題を用いて説明していきます。

次の仕様をもつパスワード入力のバリデーションを考えてみます。

- ８文字以上であること
- 数字が２文字以上利用されていること
- 上記を満たさない場合は NG

この仕様を満たすテスト対象のコードは次のとおりです。

●リスト 3.1　パスワードバリデーションのテスト対象のコード

```
static func validate(password: String) -> Bool {
    if password.count <= 7 {
        return false
    }

    let numString = password.components(
      separatedBy: CharacterSet.decimalDigits.inverted).joined()
    return numString.count >= 2
}
```

先述した仕様に対して、「2.2.2 境界値」を参考に次のケースのユニットテストを書いてみましょう。

- ■ ８文字以上であること
  - ・数字が２文字含まれており、合計７文字入力された場合に false が返されること
  - ・数字が２文字含まれており、合計８文字入力された場合に true が返されること
  - ・数字が２文字含まれており、合計９文字入力された場合に true が返されること
- ■ 数字が２文字以上利用されていること
  - ・数字以外を７文字と、数字が１文字入力された場合に false が返されること
  - ・数字以外を７文字と、数字が２文字入力された場合に true が返されること
  - ・数字以外を７文字と、数字が３文字入力された場合に true が返されること

はじめにコードを紹介し、そのあとに書き方の解説をしていきます。

●リスト 3.2　テストケースの書き方

```
import XCTest
@testable import PeaksiOSTesting03 // ①

class PasswordValidatorTests: XCTestCase { // ②

    // ③"test"はじまりの命名
    // ④日本語で名付けられたテストケース名。句読点は利用できないため、_で代用
    func test数字が2文字含まれており_合計7文字入力された場合にfalseが返されること() {

        // ⑤"XCTAssert" の頭文字で始まるXCTestのアサーションメソッド
        XCTAssertFalse(validate(password: "abcde12"))
    }
  }
}
```

①の testable import は、プロジェクトのメインターゲットにバンドルされているクラスなどにアクセスするためのインポートです。このキーワードでテストターゲットからインポートしたターゲットは、internal のアクセスレベルにあるクラスなどにアクセスすることができます。

②の XCTestCase は、テストケースを定義していくためのクラスです。このクラスにテストケー

スを定義していくことで、特定のライフサイクルにしたがって XCTest がテストを評価していきます。

③は、XCTest で書くテストケースは関数名のはじまりを `test` とすることで、テストケースとして判別されます。

④は、XCTest のテストケースは日本語で名前をつけることができるため、ここではわかりやすさの観点から日本語で命名していきます。また句読点が利用できないので、代わりに_を入れることで代用します。

⑤は、テスト対象の検証は `XCTAssert` の頭文字で始まる XCTest のアサーションメソッドを使って行います。今回はバリデーション関数の戻り値が Bool で、その真偽値が期待する結果と等価かを確認したいので、XCTest に定義されているいくつかのアサーションメソッドのうち、`XCTAssertTrue` ／ `XCTAssertFalse` の２つのアサーションメソッドを使います。

これらの書き方を踏まえてテストケースを追加していくと、次のようなパスワードバリデーションのテストコードが完成します。

●リスト 3.3　パスワードバリデーションのテストコード

```
import XCTest
@testable import PeaksiOSTesting03

class PasswordValidatorTests: XCTestCase {
    // 8文字以上であること
    func test数字が2文字含まれており_合計7文字入力された場合にfalseが返されること() {
        XCTAssertFalse(validate(password: "abcde12"))
    }

    func test数字が2文字含まれており_合計8文字入力された場合にtrueが返されること() {
        XCTAssertTrue(validate(password: "abcdef12"))
    }

    func test数字が2文字含まれており_合計9文字入力された場合にtrueが返されること() {
        XCTAssertTrue(validate(password: "abcdefg12"))
    }

    // 数字が2文字以上利用されていること
    func test数字以外を7文字と_数字が1文字入力された場合にfalseが返されること() {
        XCTAssertFalse(validate(password: "abcdefg1"))
    }

    func test数字以外を7文字と_数字が2文字入力された場合にtrueが返されること() {
        XCTAssertTrue(validate(password: "abcdefg12"))
    }

    func test数字以外を7文字と_数字が3文字入力された場合にtrueが返されること() {
        XCTAssertTrue(validate(password: "abcdefg123"))
    }
  }
}
```

XCTest によるユニットテストはこのようにテストケースごとにテストメソッドを記述していくのが基本となります。なんとなくユニットテストが書けそうなイメージがつかめたでしょうか？

重複する部分もありますが、続く各節でより詳しく説明していきます。

3.4 アサーションメソッド

ユニットテストを書いていくにあたって、結果と期待値を比較する必要があります。その比較を行うのが**アサーションメソッド**です。テスティングフレームワークには書き方の差はあれど、必ずこのアサーションメソッドが定義されています。

3.4.1 種類（一覧）

XCTest で定義されているアサーションメソッドとその説明を一覧にしました。紙面の関係上、API リファレンスから参照している引数目 `expression` を `exp` と省略し、型情報も省略しています。

●表 3.2　XCTest のアサーションメソッド一覧

関数名	説明
XCTAssert(exp:)	※ XCTAssertTrue と同様
XCTAssertTrue(exp:)	true であればテスト成功
XCTAssertFalse(exp:)	false であればテスト成功
XCTAssertNil(exp:)	nil であればテスト成功
XCTAssertNotNil(exp:)	nil でなければテスト成功
XCTAssertEqual(exp1:exp2:)	等しければテスト成功
XCTAssertNotEqual(exp1:exp2:)	等しくなければテスト成功
XCTAssertGreaterThan(exp1:exp2:)	exp1 が exp2 より大きければテスト成功
XCTAssertGreaterThanOrEqual(exp1:exp2:)	exp1 が exp2 以上であればテスト成功
XCTAssertLessThan(exp1:exp2:)	exp1 が exp2 より小さければテスト成功
XCTAssertLessThanOrEqual(exp1:exp2:)	exp1 が exp2 以下であればテスト成功
XCTFail()	テストを失敗させる
XCTAssertThrowsError(exp:)	例外をスローすればテスト成功
XCTAssertNoThrowError(exp:)	例外をスローしなければテスト成功

結果によってはいくつかのアサーションメソッドで検証を表現できますが、なるべく簡潔なアサーションメソッドを選択して、より意図を明確に示したテストを書くように心がけましょう。

たとえば `XCTAssertEqual(testResult, true)` のようなアサーションメソッドの活用は `XCTAssertTrue(testResult)` を使うことで同じ検証を行えますが、`true` との比較であることがより明確になります。

第4章

ユニットテスト（Quick／Nimble編）

細沼 祐介／@tobi462

本章では、iOS アプリ開発において XCTest と並んでよく利用されるテスティングフレームワークである、Quick ／ Nimble[注1)][注2)] について解説していきます。

Quick ／ Nimble は、Swift と Objective-C の両方の言語をサポートしていますが、本書では Swift での書き方のみを解説していきます。Objective-C での記述方法については、公式リポジトリに含まれるドキュメントなどを参照してください。

なお、本書では執筆時点の最新バージョンを利用しています（表 4.1）。

●表 4.1　本書で利用する Quick ／ Nimble のバージョン

ライブラリ	バージョン
Quick	8.0.4
Nimble	2.2.0

4.1 Quick ／ Nimble とは

Quick ／ Nimble は、Swift および Objective-C から利用可能な BDD（Behavior Driven Development）フレームワークです。プログラミング言語 Ruby における **BDD フレームワーク**である RSpec などにインスパイアされて開発されています。XCTest 上で動作するように設計されており、Xcode から XCTest と同じように実行できます。

Quick ／ Nimble と一単語で呼ばれることが多いですが、実際には BDD フレームワークとしての「Quick」と、アサーションを提供する「Nimble」の 2 つのライブラリから構成されています。

それぞれの具体的な説明に入る前に、まず BDD フレームワークとは何かを説明していきます。

4.1.1 Quick ／ Nimble の基本

「2.5.2 振舞に着目せよ - 振舞駆動開発（BDD）」で触れたように、Quick ／ Nimble は振舞駆動開発（BDD）の枠組みを提供するテスティングフレームワークです。テストコードを用いて対象のプログラムが正しく動作するか検証する点は XCTest と同じですが、対象のプログラムの「振る舞い」に注目し、より「要求仕様」に近い形でテストコードを記述しようとする点が異なります。

注1）　https://github.com/Quick/Quick
注2）　https://github.com/Quick/Nimble

期待する振る舞いを記述する - Given・When・Then

BDD によるテストコードでは「振る舞い」を表現するために、**Given・When・Then** という 3 つの要素に分けて仕様を記述します。

- Given - あるコンテキストで
- When - 何かが起こると
- Then - ある結果が期待される

例として、次の仕様をもつパスワード入力のバリデーションを考えてみます。

- ８文字以上であること
- 数字が２文字以上利用されていること
- 上記を満たさない場合は NG

Quick ／ Nimble を利用したテストコードは次のようになります（リスト 4.1）。

●リスト 4.1　Quick ／ Nimble によるバリデーションのテストコード

```
class ValidatorSpec: QuickSpec {
    override func spec() {

        // ①Quickによるテストの構造化
        describe("validatePassword関数") {
            context("8文字未満") {
                it("falseを返す") {

                    // ②Nimbleによるアサーション
                    expect(validatePassword("1234567")).to(beFalse())
                }
            }
            context("8文字以上") {
                context("含まれる数字が2文字未満") {
                    it("falseを返す") {
                        expect(validatePassword("abcdefg1")).to(beFalse())
                    }
                }
                context("含まれる数字が2文字以上") {
                    it("trueを返す") {
                        expect(validatePassword("abcdef12")).to(beTrue())
                    }
                }
            }
        }
    }
}
```

コード中の `describe` は「Given（概要）」、`context` は「When（条件）」、`it` は「Then（期待する結果）」にそれぞれ対応しています（①）。

１つ目のテストについて順に読むと次のようになり、Given・When・Then の構造となっていることが分かります。

- validatePassword 関数について（Given）
- ８文字未満の時に（When）
- false を返すこと（Then）

テストを階層化して表現することで、テスト対象がどのように「振る舞う」べきかを分かりやすく表現できています。

expect から始まるコードは期待どおりの結果が得られるか検証するアサーションコードで、比較的自然な英文として読める書き方になっています（②）。また、テスト失敗時に分かりやすいエラーメッセージが生成されるほか、組み込みのマッチャー API によって XCTest よりも柔軟な判定も可能となっています。

このように BDD フレームワークを利用することで、テスト対象の「振る舞い」に着目し、より「要求仕様」に近い形でテストを記述できます。

XCTest との比較

XCTest でも XCTContext を利用してテストのグルーピングや、アサーション関数の説明文を記述、あるいはコメントを工夫することによって似たようなことができます。しかし、フレームワークにより提供された DSL[注3)] により、それが強制される点が大きく異なります。また、XCTest に比べてより強力なテストの構造化が可能になっています。

ここまで書くと XCTest に比べて高機能でメリットばかりに見えますが、デメリットもあります。

高機能であるがゆえに自然と学習コストは高くなります。またサードパーティという位置づけのため、Xcode と統合された機能は利用できません。たとえば、テストナビゲータ上で DSL の階層構造に対応した表示はされませんし[注4)]、１つのメソッドをオーバーライドして実装する仕組み上、シンボル一覧の表示からテスト一覧を確認することもできません。

Quick ／ Nimble に限った話ではありませんが、機能的に強力なライブラリが常に選択として正しいわけではありません。プロジェクト（やメンバー）の状況に合わせて適切な技術選定を行うことが重要です。

4.1.2 Quick によるテストの構造化、Nimble によるアサーション

Quick は、前述したとおり BDD フレームワークとしてテストの枠組みを提供するライブラリで

注3） Domain Specific Language の略。日本語ではドメイン特化言語と訳され、プログラムコード内にドメインに沿った独自の API を用意することです。

注4） 実際にどのように表示されるかは図 4.5 で触れています。

す。リスト 4.1 では describe、context、it の部分が Quick に含まれる API です。XCTest においては、XCTestCase により提供される setUp ／ tearDown による前後処理、XCTContext により提供される runActivity を利用したテストの構造化の部分にあたるといえます。

Nimble は、**マッチャー**によるアサーションを提供するライブラリです[注5)]。マッチャーは、対象の値が指定した条件に一致（マッチ）するかを指定する API となっており、リスト 4.1 では it 内に記述された expect や to、beTrue などが Nimble に含まれるアサーション API です。XCTest においては、XCTAssertEqual に代表されるアサーション API の部分にあたるといえます。

●表 4.2　Quick ／ Nimble と XCTest の対応関係

ライブラリ	役割	XCTest における対応
Quick	テストの構造化	XCTestCase の setUp ／ tearDown、XCTContext.runActivity
Nimble	アサーション	XCTAssertEqual などのアサーション関数

個別のライブラリとして提供されていることから分かるとおり、Quick と Nimble はそれぞれ独立して使用できます。

たとえば、テストの構造化には Quick を利用しつつ、アサーションには XCTest に含まれるアサーション関数を利用することができます。逆に XCTest でテストの枠組みを作成しつつ、アサーションに Nimble を利用するといったことも可能です。

本書では、Quick と Nimble の両方を使用したテストコードを記述していきますが、利用している API がどちらのライブラリによって提供されているものなのかを理解しつつ、読み進めるとよいでしょう。

4.1.3　導入手順

Quick ／ Nimble の導入方法について、CocoaPods と Carthage を使った方法について記載します。それ以外の手順については割愛しますが、公式ドキュメントに記載されているので、必要に応じて参考にしてください[注6)]。

CocoaPods

CocoaPods でインストールする場合、Podfile のテスト用のターゲット[注7)]に定義を追加します（リスト 4.2）。

注5）　マッチャー自体は厳密にはアサーションではなく、あくまで値が一致するかを検証する API ですが、本書では話をシンプルにするためマッチャーを含めてアサーションと表現します。
注6）　https://github.com/Quick/Quick/blob/master/Documentation/en-us/InstallingQuick.md
注7）　アプリ本体のターゲットに追加すると、アプリの実行バイナリに含まれてしまうためです。

●リスト 4.2　Podfile

```
target 'App' do
  use_frameworks!

  target 'AppTests' do
    inherit! :search_paths

    # 次の行を追加
    pod 'Quick'
    pod 'Nimble'
  end
end
```

あとはターミナルから pod install コマンドを実行すれば導入は完了です。

```
$ pod install
```

Carthage

次の手順で行います。

1．次のコードを Cartfile.private[注8)] に追加します。

●リスト 4.3　Cartfile.private に追加するコード

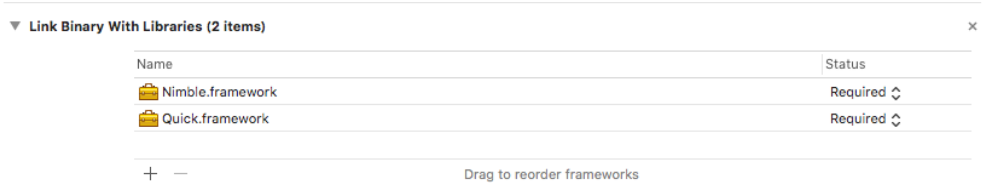
```
github "Quick/Quick"
github "Quick/Nimble"
```

2．ターミナルから次のコマンドを実行します。

```
$ carthage update --platform iOS
```

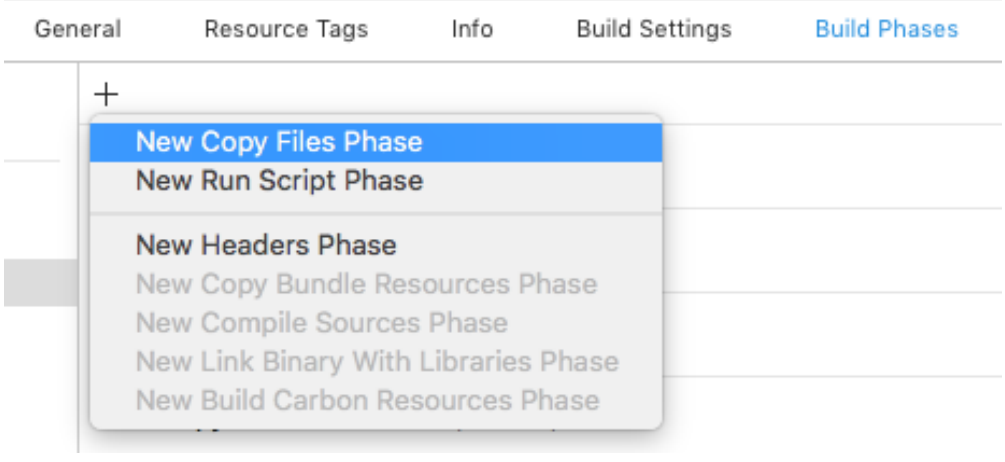
3．「Carthage/Build/iOS/」フォルダから「Quick.framework」と「Nimble.framework」をテストターゲットの「Build Phaeses」の「Link Binary With Libraries」に追加します[注9)]（図4.1）。

注8)　アプリの実行バイナリに含めないものは Cartfile.private に記述します。
注9)　ファイルをドラッグ＆ドロップするか「＋」ボタンから選択します。



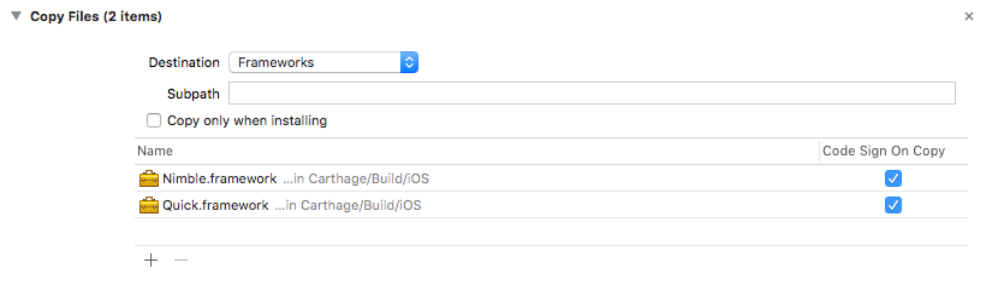
●図 4.1　「Link Binary With Libraries」に 2 つのフレームワークを追加

4．同じく「Build Phases」の左上の「＋」ボタンから「New Copy Files Phase」を追加します（図 4.2）。



●図 4.2　Build Phases の＋ボタンから「New Copy Files Phase」を選択

5．追加した「Copy Files」にて Destination プルダウンから「Frameworks」を選択し、左下にある「＋」ボタンから 2 つのフレームワークを追加します（図 4.3）。



●図 4.3　Copy Files の設定を終えたところ

第 5 章

BDDによるアプリ開発

細沼 祐介／@tobi462

第 2 章から第 4 章をとおして、ユニットテストの概要、XCTest および Quick ／ Nimble の利用方法について解説してきました。本章では総集編的な位置づけで、BDD を利用したサンプルアプリの開発を行っていきます。

実際にテストコードを書き進めながら、TDD の原則である「テストを先に書く」とはどういうことか、BDD における「振る舞いに着目する」とはどういうことか、について理解を深めていきます。

5.1 本章の構成

本章は大きく次のような構成となっています。

- ①品質を「外側」と「内側」に分ける考え方を理解する
- ② BDD による開発フローを理解する
- ③アプリの仕様や実装の流れを把握する
- ④ BDD のための土台を構築する
- ⑤実際にアプリを開発する
- ⑥全体を振り返る

最初に、BDD による開発がどのようなものか全体を見ていきます。①品質について「外側」と「内側」の 2 つに分ける考え方を解説した上で、② BDD による開発サイクルの全体像について見ていきます。

③次に本章で開発するアプリの仕様について説明し、具体的にどのような流れで開発していくかを示します。

④アプリの開発を始める前段階として、BDD のための土台を構築します。そして土台が整えられたところで、⑤実際にアプリの開発を進めていきます。

⑥最後に、本章の全体をとおしてどのようなことを学べたかを振り返ります。それぞれのトピックについて、さらに深く学びたい方に向けた参考文献の紹介も行います。

■ 本章における BDD について

第 2 章で触れたとおり BDD については大きく、Quick ／ Nimble などのコードスタイルを指す場合と、顧客に対する受け入れテストも含めた開発プロセス全体を指す場合があり、BDD という用語が何を指

すのかはとても曖昧になっています。
本章では両方について取り上げていきますが、「厳密な BDD の定義」について解説することが目的ではなく、一連の流れをとおして「考え方」や「テクニック」を示すことを主軸においています。

■ 本章における UI テストについて

本章では ViewController に対してテストも作成していきますが、これは広い意味では「UI テスト」に分類することもできます。
UI テストについては第 6 章から第 8 章で解説しますが、それらの内容を理解せずとも読み進められる内容となっています。

5.2 外側の品質と内側の品質

ソフトウェアにおける品質を**外側の品質**と**内側の品質**の２つにわける考え方があります。
「外側の品質」というのは**利用者から見たソフトウェアの品質**のことです。アプリでいえば実際に利用するユーザから見た次のような観点です。

- 必要な機能が満たせているか
- 操作性は問題ないか
- 画面の表示崩れは無いか
- クラッシュしないか

一方「内側の品質」というのは、**開発者から見たソフトウェアの内部構造における品質**のことで、具体的には次のようなものです。

- アーキテクチャやクラス設計は適切か
- 利用しやすい API 設計になっているか
- コードの可読性は保たれているか

ソフトウェア業界において単に「品質」と表現した場合、「外側の品質」について言及されることが多いように感じます。利用者にとっては目の前のソフトウェアが動くかどうかがすべてであり、それがどのようなプログラミング言語やアーキテクチャで実現されているかは関係ありません
また、アプリの開発に携わる「企画」「デザイナー」「マネージャー」なども同様のはずです。
しかし、開発者にとっては「内側の品質」も同じくらい大切です。一度完成したら「それで終わり」というソフトウェアは稀で、アプリを含めて多くのソフトウェアは機能追加・変更を繰り返し、

とても長い期間にわたって保守・運用を続けていくはずです[注1)]。

アーキテクチャやクラス設計・コードの可読性といった「内側の品質」が悪いと、機能追加や変更などにより多くの時間がかかる、リグレッションが多発するといった問題に繋がります。場合によってはリリースしたアプリに致命的な不具合が見つかるなど、「外側の品質」にまで影響を及ぼすケースもあります[注2)]。

品質のよいソフトウェアを安定して継続的にリリースするためには、「外側の品質」と同様に「内側の品質」も考えていく必要があるのです。

5.3 BDD による開発フロー

本章における BDD では次のステップで開発を進めます（図 5.1）。

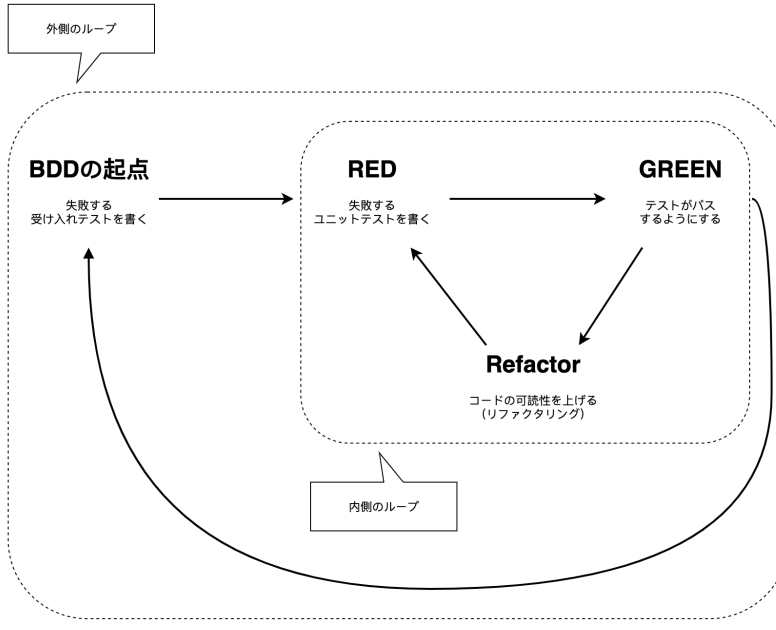
1. 失敗する「受け入れテスト」[注3)]を作成する
2. TDD プラクティスを利用して機能を実装する[注4)]
3. 「受け入れテスト」が成功するまで 2. を繰り返す
4. 1. に戻って繰り返す

注1) 一度完成したらそれで終わるソフトウェアとして自身の作業効率化のための「使い捨てのスクリプト」などが考えられます。ただ、その場合でもコードとしては残しておき、あとから流用できるようにするケースが多いはずです。

注2) iOS アプリの保守に携わったことのあるほとんどの人は、こうした問題に直面したことがあるのではないでしょうか。

注3) 本章では ViewController を介したテストのことを、それ以外のテストと区別する意味で「受け入れテスト」と表記します。「UI テスト」とも表現できますが、BDD における文脈を意識して「受け入れテスト」という単語を選んでいます。

注4) ここでは RED、GREEN、Refactor からなる一連のサイクルを意図しています。



●図 5.1　BDD による開発フローの全体像

5.3.1　失敗する受け入れテストを作成する - 「外側の品質」

最初に、これから実装しようとしている機能について、外側から見てどのように振る舞うべきかという視点で受け入れテストを作成します。

これはそもそも「どういったものを作るべきか」を明確にする目的と、ユーザから見て期待どおりに動作するかをテストするもので、ソフトウェアにおける「外側の品質」を作り込むものです。

たとえば、Twitter クライアントアプリであれば次のようなものです。

- アプリ起動時にタイムラインが表示されること
- ツイート機能からタイムラインに投稿できること

最初に受け入れテストを作成するということは、テスト対象の機能は実装されていないため、テストは失敗から始まることになります。これが**失敗する受け入れテストから始める**という意味です。

Column.　受け入れテストはエンドツーエンドであるべき？

受け入れテストは **E2E テスト**（エンドツーエンドテスト）[注5] に近いほうが望ましいとされます。ユーザから見て期待どおりに動作するか、関連システムを含めた全体をテストしたいためです。

しかし、エンドツーエンドテストは実行時間や安定性の問題もあり、受け入れテストを必ずエンド

ツーエンドテストで作成するようにしても、必ずしも費用対効果が得られないというケースも考えられます。

ここまでの章でも何度か触れていますが、大切なのは何をテストしたいのかに着目することで、厳密な定義やプロセスにこだわることではないと筆者は考えます[注6)]。

受け入れテストにおいてもエンドツーエンドにこだわるのではなく、関連システムや HTTP 通信を置き換える方法も選択肢に入れるとよいでしょう。

5.3.2 TDD プラクティスを利用して機能を実装する - 「内側の品質」

失敗する受け入れテストが作成できたら、第 2 章で解説した TDD プラクティスである「RED → GREEN → Refactor」のサイクルを用いて実装を進めていきます。

1. 失敗するテストを書く（RED）
2. テストが成功するように実装する（GREEN）
3. リファクタリング（Refactor）

このサイクルを繰り返しながら実装を進めていくことで、「失敗するテスト」から始めることで自然とテストしやすい設計になり、リファクタリングにより自然と「内側の品質」が作り込まれます。

実装を完了すれば受け入れテストが成功することになり、それをもって機能が完成したことになります。

5.3.3 外側のループと内側のループ

最初に作った受け入れテストが成功することで機能が完成し、全体のループを一周したことになります。

図 5.1 で示したように、全体のサイクルは「外側の大きなループ」と「内側の小さなループ」の２つのループにより構成されます。

どちらも失敗するテストから始めることで、これから作るべきものを明確にしつつ、テストしやすい設計を導きます。これは開発においてありがちな「テストを書こうと思ったが設計が悪くて書けない」という事態を防ぐことに繋がります。

外側のループでは、最初になにを作るべきかを明確にし、外側からテストできる設計を考えることを強制させることで「外側の品質」を支えます。内側のループでは、最初に利用しやすい API を検討させ、リファクタリングによるコードの可読性や設計の改善により、「内側の品質」を支えます。

注5） End to End テスト。システムの端から端まで通してテストすること。一般的な通信を行うアプリにおいては「画面の操作」から「通信先のバックエンドサーバー」までテストすることにあたります。

注6） 本書の範囲外になるので詳細は割愛しますが、アジャイルソフトウェア開発宣言では「プロセスやツールよりも個人と対話を」という言葉があります。

第6章
UIテスト入門

平田 敏之／@tarappo

UI テストはユニットテストと比べると、実装・運用コストが大きくなってしまいがちです。あまり考えずに UI テストを実装していくと、あとあとコスト面だけでなくさまざまな面で問題となってしまうでしょう。しかし、うまく使えばプロダクトにとって強い武器となります。

本章では、UI テストを利用する上で知っておくとよいことを解説しています。まずは UI テストを実装しようとはやる気持ちを抑えて、本章を読んでみてください。そのあとに UI テストの実装方法などについて書かれた第 7 章、第 8 章を読めば、UI テストを有意義に活用できるでしょう。

6.1 UI テストとは

第 1 章で述べたように、本書で扱う UI テストは対象となるプロダクトの UI に対して、何らかの働きかけをする類のテスト[注1)]を指します。

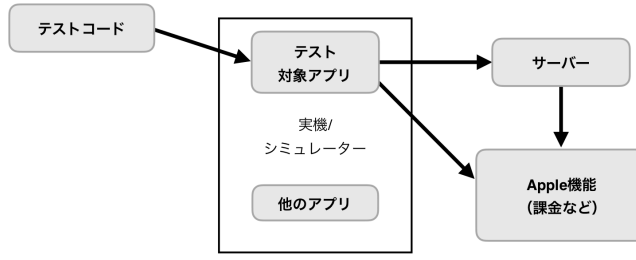
iOS の自動テストは、iOS シミュレーターや実機を通して実行します。さらに UI テストでは、テスト対象となるアプリを操作できることから、ユーザが行う手動テストと同等のことも行えます。このようなテストは、E2E テストとも呼ばれます[注2)]。

ユーザの操作と同等のことが行えるため、手動テストの代わりに利用したいと考えるかもしれません。しかし、手動テストで確認していることすべてが UI テストで行えるわけではありません。たとえば、いま見ている画面のレイアウトが崩れているといった、テストコードに落とし込めない確認まではできません。そのため、必ずしもいま行っている手動テストの代わりとなるとは限らない点には注意が必要です。

また、UI テストでは自動テストの実行中に関わるものが多くなります。たとえば図 6.1 にあるように、テストコードを通して複数のサービスと関わります。

注1) 本書での UI テストは単に UI 要素を操作するものだけをさしません。第 7 章や第 8 章では、UI 要素以外として Siri を操作する話もあります。

注2) アプリとサーバとの通信を HTTP スタブにすることにより、E2E テストではない UI テストを行うこともできます。



●図 6.1　関わるサービスが多いので不安定になりえる

これらの関わってくるサービスが何かしら正常に動作しないことや不安定の原因になることもあります。特に第三者の提供する API においては、自分たちで品質面を担保することはできません。その結果、実行の安定性という面において他の自動テストと比べても問題になりがちです。

このように書くと、UI テストを利用するのは難しそうに感じるかもしれません。本章では、UI テストを実際に長く利用していくために、UI テストの実装方法などを紹介する前に「導入前」「実装中」「運用中」といったフェーズにおいて、知っておくとよいことについて説明をします。

6.2　導入前に検討をしておくこと

UI テストを導入する前に知っておくとよいこと、決めておくとよいことがあります。そこで、本節では次のような点について説明をしていきます。

- UI テストを行う目的を考える
- UI テストで担保する範囲を考える
- UI テストにかかるコストを見積もる
- UI テストの実装をするのは誰かを決める

説明をした内容について検討をせずに進めてしまうと、あとで手戻りが発生することもありますし、UI テストの導入が失敗するかもしれません。

6.2.1　UI テストを行う目的を考える

さまざまな自動テストの中で、UI テストを導入する目的を考えます。UI テストを導入することによるメリットとしては次のようなものがあります。これらのメリットは UI テストを導入する目的となりえます。

- 手作業の誤り・失敗を減らす

- 退屈と感じやすい作業を自動化することにより、検証担当者の負担を減らす
- 開発のフィードバックを手作業よりも短時間で回すことができる

これらがどのようなメリットになるのか簡単に説明します。

手作業の誤り・失敗を減らす

次のような内容の作業を行っている場合、手作業の誤り・失敗が起きることがあります。

- 繰り返される作業
- 複雑な操作

繰り返しの作業を行い続けていると、たまに誤りや失敗を起こしてしまうこともあります。誤りや失敗が発生したときに「失敗をした箇所を入念にしたチェックリストをつくり、それをもとに人がチェックをする」といった対応をしていないでしょうか。

UI テストは、このような繰り返される手作業の全部または一部を自動化してくれます。UI テストに限らず自動テストは繰り返し作業が得意で、作業の途中でなにかしらの問題が発生したらそのことを教えてくれます。

複雑な操作においても、同様に誤りや失敗を生みやすくなるでしょう。たとえば、アプリ内でいくつもの設定をしてから、実際の操作を行うといったこともあります。このような複雑な操作の場合、手動だとひとつの手順を忘れてしまったがために本来求める状況になってなくて、1からやり直しといったことも起きます。

このような複雑な作業も、UI テストのほうが得意といえます。またどの状況まで進んだかもテストの実行ログからわかります。

退屈と感じやすい作業を自動化することにより、検証担当者の負担を減らす

繰り返される作業は誤りや失敗を起こしやすいだけでなく、退屈にも感じやすいです。そのような作業を UI テストで担保することは、検証担当者（以後、QA メンバーとします）のさまざまな負担を減らすことに繋がります。

減った負担で、QA メンバーが本来行うと効果を発揮する「Testing[注3)]」を行う工数をとることもできます。

開発のフィードバックを手作業よりも短時間で回すことができる

UI テストは実行時間がかかりますが、それでも手作業で行うよりかは短時間で実行できます。そのため、UI テストがあれば開発のフィードバックを短時間で回すことができます。つまり、開

注3）　Testing については第 1 章の「1.2.3 確認のためのテスト」で紹介をしています。

発した機能をユニットテスト、UI テストを通して問題がないことを比較的気軽に確認できます。その結果として、安心して開発を行うことができます。

6.2.2 UI テストで担保する範囲を考える

UI テストで何を担保するのか、どこまで担保するのかというのは、事前にある程度決めておきましょう。またこれは、一度決めたらそれを守り続ければよいわけではなく、プロジェクト状況の変化によって見直しが必要なこともあります。

前提として、自動テストを用意する上では第 1 章で紹介した「テストピラミッド」を考慮して行います。テストピラミッドの上層でなく、より下層で機能の担保をできるのであれば、そちらの自動テストを実装するように考えていきます。問題は早い段階で見つけることが重要です。したがって、ユニットテストやそれ以前で見つけられるのであれば、それに越したことはありません。

ユニットテストで担保ができず、手動テストで行うよりも UI テストで行ったほうが効果が高い箇所に対して、UI テストを利用していきます。たとえば、「6.2.1 UI テストを行う目的を考える」で説明をした次のような箇所が考えられます。

- 手作業で繰り返しアプリの操作を行っている箇所
- 複数のアプリをまたぐなど複雑な作業を要するものであり、手作業で行うには非常に手間と時間がかかる箇所

上述したような箇所を UI テストで担保するにあたっては、コスト面も考慮することは大事です。ただし、UI テストにより削減されるコストのほうが少ないからといって、担保する範囲外にするのは必ずしも正しいとはいえません。コストとして見えるもの以外にも繰り返し作業の削減に伴う心理的負担の削減や、UI テストがあることによる開発生産性の向上などといった効果もあります。

6.2.3 UI テストにかかるコストを見積もる

UI テストを実装する上でかかるコストとして、「実装にかかるコスト」と「運用時にかかるコスト」があります。コストを見積もれるように、これらのコストがかかりがちな点について説明していきます。

実装にかかるコスト

UI テストを実装するにあたり行うこととしては主に次になります。

1. アプリの画面にある UI 要素の特定
2. UI 要素を特定したあとのテストコードへの実装と動作確認

まずアプリの画面にある UI 要素の特定ですが、操作をしたい対象の画面の数と UI 要素の数が

多ければ多いほどコストがかかります。特に UI 要素を一意に特定しづらい場合は、プロダクトコード側で特定しやすいよう改修をするケースもあります。

UI 要素を特定した後は、その UI 要素に対してなにを行うかをテストコードに実装しつつ、動作確認をすることになります。これらの実装や動作確認がスムーズにいくということは少ないでしょう。それはなぜでしょうか。

通常の手動テストでは、人がよい感じにいろいろと補ってくれます。たとえば、通常３秒程度で表示されるようなページの操作をするとして、表示までの時間が６秒や７秒になったとしても人であれば待ってくれるでしょう。他にも、常に表示されるわけではないが特定の条件の場合だけ表示されるようなダイアログに出くわしたときも、人であれば自然と閉じるような操作をしてくれるでしょう。

しかし、UI テストではそういうわけにはいきません。いままで、手動テストのときに人が「よい感じに」補ってくれていた箇所に対して、ひとつひとつ対応が必要です。そのため、想定以上に実装コストがかかりがちです。

そこで、UI テストを始めるには、アプリの広範囲の画面を扱うテストコードを実装するよりも、まず少ない画面で問題なく動くかどうかから始めるのがよいでしょう。

運用時にかかるコスト

UI テストの初期の実装が終われば、それで完了でしょうか。UI テストを実装したあとに、何も対応をしなくてもテストが問題なく成功し続けるわけではありません。

なにかしらの対応が必要になるケースとしては次のようなものがあります。前者はかかる時期やコストが比較的読みやすいもので、後者は判断がしづらいものとなります。

- プロダクト側の変更
  - プロダクトの UI の変更
  - プロダクトの機能の変更
- テストのための基盤のバージョンアップに伴う変更
  - XCUITest の API の変更
  - Xcode のテストにかかわる機能の変更・追加
  - テストのための基盤自体に含まれる不具合の回避

１つ目の「プロダクト側の変更」においては、変更が発生した時点でテストコードを修正する必要があります。

UI の変更が多い箇所においては、対応コストがある程度かかります。そのため、そのような箇所において本当に UI テストで実装をするべきかどうかは考えておく必要があります。アプリの UI 側の変更が少ない箇所（たとえばログイン画面や設定画面など）であれば、運用時にかかるコストは少なくなるでしょう。

また、UI の変更に効果的な UI テスト実装におけるデザインパターンもあります。実装方法の

第 7 章

XCUITestを使ったUIテスト

玉城 信悟／@alligator_tama

本章ではXCTestのUI Tesitngの機能を使ったUIテストの実行について説明します。XCTestのUI Testingの機能は通称XCUITestとも呼ばれることも多いため、本章ではXCUITestという呼称で説明を進めていきます。XCUITestは、Xcodeやxcodebuildといった Apple純正ツールとうまく統合されていて、開発者のローカル環境やCI環境のどちらでも使用しやすくなっています。

7.1 XCUITestとは

2015年のWWDCで発表されたXcode 7からXCTestに追加されたUIテストのための機能です。XCTestはユニットテストとUIテストのフレームワークを含んでいます。UIテストのフレームワークは**iOS UI Testing Bundle**で作成したターゲットにおいて利用できます。

XCUITestが登場する以前はUIAutomationという**Instruments**の一機能を使ってUIテストをサポートしていました。現在ではUIAutomationは利用できなくなっていて、UIテストを書くにはXCUITestを利用する必要があります。

またXCUITestでは**XCTest Runner**というUIテスト専用のアプリがテスト対象のアプリと一緒にインストールされます。このランナーアプリがXCTestフレームワークを介して**ターゲットアプリケーション**[注1)]の操作を行います。

注1)　ランナーアプリによって操作されるテスト対象のアプリケーション

●図 7.1　テストランナーがターゲットアプリケーションをユーザの代わりとなって操作する

表 7.1 は XCUITest の主な 3 つのクラスの説明です。

●表 7.1　XCUITest の主なクラス

クラス	役割
XCUIApplication	ターゲットアプリケーションの起動・終了を行うためのプロキシとなるクラス
XCUIElementQuery	ターゲットアプリケーションの要素を特定するための問い合わせ（クエリ）を表すクラス
XCUIElement	ターゲットアプリケーションの要素そのものを表現するクラス

これらのクラスをこの後説明する UI テストの 4 つのプロセスの中で使用していきます。

7.2　UI テストの流れ

UI テストの流れには次の 4 つのプロセスがあります。このプロセスは人がアプリを使う時のプロセスにもとづいていて、テストコードはユーザと同じ立場でテストターゲットのアプリを動かします。XCUITest や第 8 章で紹介するような iOS 向けの他の UI テストライブラリ、Web ブラウザを対象とした UI テストでもほぼ同様です。

1. ターゲットアプリケーションの「起動」
2. UI 要素の「検索」、対象の要素を絞り込むクエリの作成

3. UI 要素の「検証」、対象の要素の値を検証して検索が正しく実行されたかを検証する
4. UI 要素の「操作」、対象の要素に対するタップやスワイプといった操作の実行

UI テストの４つのプロセスについて XCUITest の主要なクラスを当てはめると表 7.2 のようになります。

●表 7.2　UI テストの各プロセスで使用する API

プロセス	API
ターゲットアプリケーションの「起動」	XCUIApplication
UI 要素の「検索」	XCUIElementQuery
UI 要素の「検証」	XCTAssert
UI 要素の「操作」	XCUIElement

ターゲットアプリケーションの起動以降は２～４のプロセスのいずれかを繰り返しながらテストを書き進めていくことになります。

なお XCTAssert を含むアサーション API は、XCTest に用意されたものをそのまま利用できます。詳細については第 3 章で解説しているので、必要に応じてご参照ください。

7.3　XCUITest を使ったテストコードはじめの一歩

それでは XCUITest を使った UI テストを始めましょう。この節ではログイン機能のみを持つサンプルアプリを用意し、まずは XCUITest の API の基本的な使い方を説明します（図 7.2）[注2)]。

API の基本的な使い方を理解したのち、あとの節でサンプルアプリの「正常にログインが完了するシナリオ」として UI テストを完成させます。

注2)　より詳細な API については後の節で説明します。

●図 7.2　UI テスト対象のサンプルアプリ

サンプルアプリは次のような機能が実装されています。

- ログイン画面はユーザ名とパスワードを入力する入力欄とログインボタンを持つ
- ログイン処理を行ったあと、次回起動時はログイン画面は表示せずトップ画面を表示する
- トップ画面はシンプルなリストを表示する画面

このサンプルアプリを次のような流れでテストの書き方を説明していきます。

1. ターゲットアプリケーションの起動
2. ユーザ名とパスワードの入力、ログインボタンのタップ
3. ログイン後の画面に遷移する

それでは、プロジェクトにファイルを追加してテストの準備をしましょう。リスト 7.1 は Xcode 標準のテンプレート「UI Test Case Class」から生成したコードからコメントなどの情報を削除したものです。

●リスト 7.1　UI Test Case Class テンプレートから作成されたコード

```
import XCTest

class FirstStep: XCTestCase {
  override func setUp() {
    continueAfterFailure = false
    XCUIApplication().launch() //起動
```

第 8 章

サードパーティ製ツールとAppium

松尾 和昭／@Kazu_cocoa

本章では特に UI テストに絡むサードパーティ製ツールを扱います。次節にていくつかサードパーティ製ツールに触れたのち、その後からは Appium を中心に取り上げます。Appium を取り上げる理由としては、いくつかのテスト実行基盤を提供する企業が公開する利用状況の中でも、Appium の利用率が XCUITest と比較しても高いためです[注1)]。

Appium の話では、Appium 自体の話やテスト実行環境の構築や基本 API といった基本的な利用方法を説明します。その後、Appium における CI ／ CD の環境構築例、並列実行を説明します。一部、XCTest フレームワークに関する話もでてきます。

8.1 XCUITest 以前から取り巻くサードパーティ製ツール

第 7 章において説明された XCUITest は、ブラックボックスタイルの UI テストを支援します。一方、テスト実行の安定性や効率を考えると、テスト対象に対してホワイトボックススタイルに寄った UI テストを実現したいという要求もあります。第 7 章にて触れられた XCUITest 以前の UI テストツールである UIAutomation では JavaScript が、XCUITest では Objective-C もしくは Swift によるテストコードが必要でした。それ以外の開発言語でテストを実装したいという要求もありました。iOS アプリ以外の既存環境との連携を重視しないとい目的もありました。

昨今では、XCTest の「iOS Unit Testing Bundle」ターゲットを拡張する形で、ホワイトボックステストの UI テスト実装を実現する KIF や EarlGrey（v1）などのツールがあります。Appium は多様な実装言語でテストコード実装を実現してました。UIAutomation や XCUITest 同様、ブラックボックステストの実現を志向してきました。

表 8.1 に現在も開発が続いているサードパーティ製 UI テストツールをテスト実装言語と、それらがどのような形式のテストを志向するかに分けてまとめました。本章ではすべては説明しませんが、たとえば Google が開発する EarlGrey、Unity アプリに焦点をあてている AirTest、React Native アプリを中心とした Detox などがあります。

注1） Bitbar の記事（https://bitbar.com/blog/speed-and-efficiency-of-mobile-test-automation-frameworks/）によると彼らの環境では半数以上、Kobiton の情報（https://twitter.com/KobitonMobile/status/1101168218490707968）によると Appium は XCUITest に比べて何十倍の利用者がいるそうです

●表 8.1　ホワイト／ブラックボックス

ツール	実装言語	スタイル
XCUITest	Objective-C ／ Swift	ブラックボックス
Appium	JavaScript、Ruby、Python など	ブラックボックス
AirTest[注2)]	Python	ブラックボックス
macacajs[注3)]	JavaScript、Java、Python	ブラックボックス
Detox[注4)]	JavaScript	グレーボックス
EarlGrey v1[注5)]	Objective-C ／ Swift	ホワイトボックス（XCTest ベース）
EarlGrey v2[注6)]	Objective-C ／ Swift	ホワイトボックス（XCUITest ベース）
KIF[注7)]	Objective-C ／ Swift	ホワイトボックス（XCTest ベース）

サードパーティ製ツールは、XCTest フレームワークをはじめとする各種フレームワークやそれらのプライベート API が活用されています。Xcode のメジャーアップデートによりコードの修正が必要になる懸念はあるものの、ある段階における XCUITest では実現できないことを実現しているなどの利点があります。

しかし、Xcode のバージョン更新に伴い、XCTest（XCUITest 含む）でも利用可能な機能が増えてきました。広く XCTest が提供する API も安定化が進んだため、サードパーティ製ツールも広く XCTest の API を活用する動きが増え、プライベート API の利用範囲も減ってきています。

XCUITest 含め、より安定したテスト実行やその環境構築を指すと、内部・周辺ツール群の理解が必要になります。本章では XCUITest の挙動についても必要に応じて触れていきます。

8.2　Appium とは

テストコードをさまざまな開発言語を用いて記述可能[注8)]な自動化ツールです。対象は iOS に限らず Android、Mac デスクトップアプリ、Windows アプリなど含めた多様な端末です。OSS として公開され開発が行われています。数年前までは SauceLabs のメンバーのみにより開発が行われていましたが、現在は OpenJS Foundation のもと、さまざまな人も参加しており活発に開発されています。

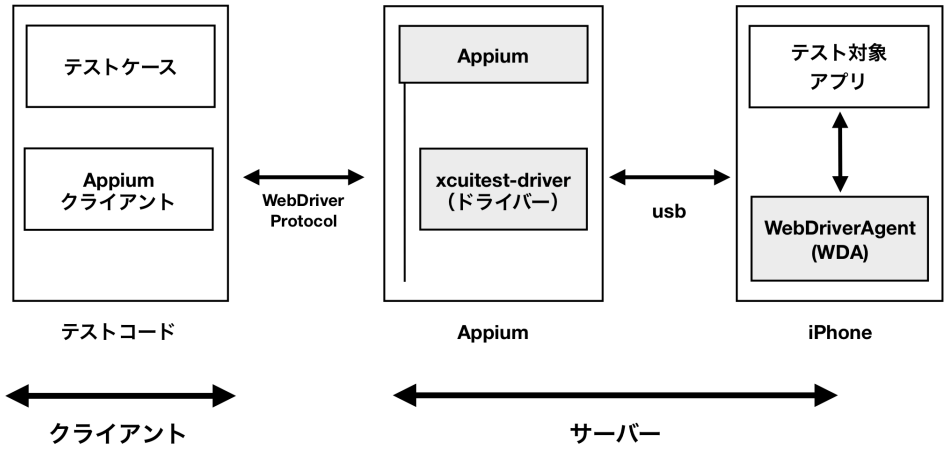
Appium は次の 4 つの大きな価値にもとづいて開発されています。

- ユーザーが利用する環境でテストを実行できること
- 自分たちで実装言語を選択できること（テストコードの実装言語を環境が強制しない）
- 標準化された API を経由し、ハブのように多様な端末を操作できること

注3）　http://airtest.netease.com/
注4）　https://macacajs.github.io/
注5）　https://github.com/wix/Detox/
注6）　https://github.com/google/EarlGrey/
注7）　https://github.com/kif-framework/KIF/
注8）　Ruby、Python、Java、JavaScript、Objective-C、C#（.NET）、PHP が公式ページには並べられています。Swift 実装はありません。

■ オープンソースコミュニティとして成長すること

Appium はクライアント・サーバー形式をとり、ユーザーは「クライアント」側としてテストコードを記述します。「サーバー」側の機構が iPhone や Apple TV などの機器とやり取りを行います。（図 8.1）



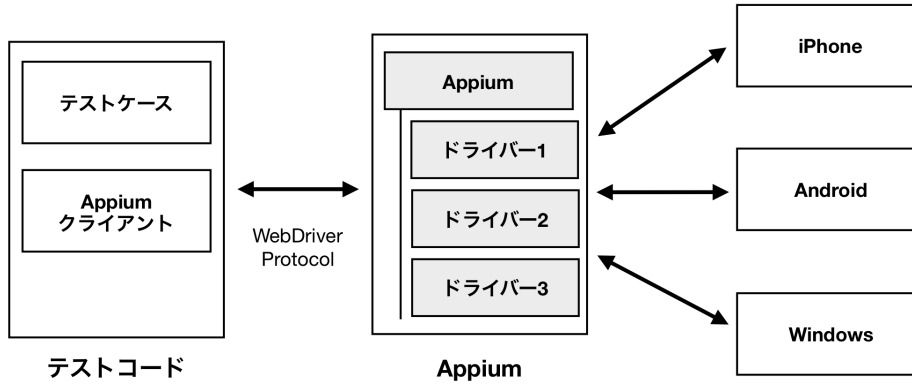
●図 8.1　Appium のアーキテクチャ

これらの「機器とのやり取り」を行う仕組みを「ドライバー」と呼んでいます。「xcuitest-driver」というような文言が Appium のドキュメントや issue で散見されます。これはこの「ドライバー」の役割をもつモジュールであることを意味します。

たとえば、先ほどの図 8.1 は iOS 9.3 以上を対象とした iOS 向けテスト実行時の構成です[注9)]。テストコードで囲まれる範囲が「クライアント」に該当し、多様な言語で実装可能です。それ以外の Appium と **WDA**（WebDriverAgent）に含まれる範囲が「サーバー」と記されている範囲です。

このドライバーの対応が広がることで、「クライアント」は単一の実装言語で多様な端末（ドライバー）に対してテストコードを実行・操作できます。（図 8.2）

注9）　Appium では iOS9.2 以下を対象端末として操作するときの構成は異なります



●図 8.2　マルチプラットフォームな端末と Appium の接続

テストコード自体は Selenium ／ Appium に対応しているクライアントの範囲内で、任意の実装言語を選択することができます。それらのクライアントの内部実装は W3C WebDriver[注10)] に準拠している必要があります。操作の抽象化は Appium で行われるため、テストコードの実装者はドライバーごとに一から実装する必要はありません。

脚注[注11)] にて触れたとおり、Objective-C 実装の公式クライアントは存在しますが、Swift 実装は公式にはありません。Swift 言語を使い iOS アプリ開発をしている読者にとっては、単一言語でシナリオ記述まで実装できないことに難しさを覚えるかもしれません。

Column. Appium クライアントの Swift 実装

著者は Swift によるクライアント実装を試した過去があります。開発環境のバージョンや Swift の更新に伴うクライアントライブラリの更新、Swift 言語のテストコード記述の柔軟さを考えると、あまり追従のコストに見合わないと感じたので断念しました。

Swift 実装のクライアントをプロダクトコードと合わせて利用する場合、テストコード側に手を加えることなく、プロダクトコードだけ環境を更新して動作が正しいことを確認する。問題なければもう一方の環境も更新する、というお互いに補完し合う関係が実現しにくかったためです。

なお、モバイル端末向けテストサービス[注12)] の実行基盤として、Appium を活用する事例も散見されます。たとえば、国内外に存在する機械学習を活用した挑戦的なテスト自動化サービスでは主に Python や JavaScript と Appium を利用し、iOS ／ Android 問わずモバイル端末向けにサー

注10）https://www.w3.org/TR/webdriver1/
注11）Ruby、Python、Java、JavaScript、Objective-C、C#（.NET）、PHP が公式ページには並べられています。Swift 実装はありません。
注12）たとえば、１章でもとりあげた MagicPod（https://magic-pod.com/）のような、テスト実装とその実行を支援するようなサービスを指します

ビスが提供されています。このような用途にも使われているのが Appium です。

8.2.1 Appium の構成

図 8.1 のとおり、Appium はクライアント・サーバー形式の構成をとります。

Appium の利用を開始すると、WDA[注13] がシミュレーター・実機にインストールされます。その後、WDA が端末上で動作するサーバーとして仲介役となり、クライアントから送られてきたコマンドを解釈してテスト対象を操作します。この WDA が Appium において、第 7 章にて説明された XCTest Runner という UI テスト専用のアプリに該当します。

XCTest フレームワークが識別できない要素は、Appium も同様に識別できないことが多いです。要素探索などのおおもとは、XCTest フレームワークと同様の形で要素を参照・取得しているためです[注14]。

クライアントと Appium サーバー間は WebDriver[注15] 形式に沿った処理を行います。この W3C 規格に準拠しているクライアントであれば、さまざまな言語実装から利用可能です。

モバイル端末やその OS のさまざまな機能は、現在の W3C で定義されているものだけでは十分に表現できません。そのため、Appium は W3C に定義されていないもののうち、独自にコマンドを定義しているものもあります。クライアント・サーバー間の相互運用性を考え、新規に WebDriver 形式でコマンドを定義するのではなく、基本となるコマンドのパラメーターを操作する形で拡張しているものもあります。これらはあとで登場します。

XCTest フレームワークを利用するということは、Appium の挙動は Xcode のバージョンに依存することを意味します。Appium 実行時に利用する Xcode のバージョンを切り替えることで、新しい Xcode では操作対象から外れた古い iOS バージョンをテスト対象とすることも可能です。一方で、Xcode のバージョンに依存して挙動がわずかに変化したり、何らかの Xcode 側由来の不具合に出くわすこともあります。時には古い Xcode バージョンのほうが XCTest フレームワークが安定していることもあるので、使い分けも必要になります。Appium 実行時に利用する Xode バージョンの切り替えは「8.5.2 iOS のテスト実行環境管理」で説明します。

8.2.2 Appium のテストライフサイクル

XCTest では通常、テスト実行はシミュレーターの準備が終わり、テスト対象がそれらにインストールされたあとから、ユーザーが実装したテストコードが開始されます。`XCUIApplication().launch()` によりテスト対象の起動が行われ、記述されたテストコードが実行されます。テスト終了時にはシミュレーターの終了は行われません。そのため、テスト開始時には `setUp()` であったり、次回のテスト実行時のために `tearDown()` などに、プロダクトコー

注13） https://github.com/appium/WebDriverAgent
注14） WDA のコードを読むと XCTest フレームワークのプライベート API を多く参照していることがわかります
注15） https://www.w3.org/TR/webdriver1/

第9章
CI／CD

平田 敏之／@tarappo

マーケットにおける競争力や優位性を高めて維持し続けるためには、顧客からのフィードバックをはやく受けとり改善のサイクルをはやく回すことが必要です。それには、プロダクトをすやばく、そして確実・安全にリリースできることが非常に重要です。そのためにはCI（継続的インテグレーション）／CD（継続的デリバリー）が適切に実現できていることが大事です。

iOSアプリ開発におけるCI／CDとはどういうものでしょうか？

本章では、iOSアプリ開発におけるCI／CDを説明していきます。

前半ではCIとCDについて説明します。またCI／CDで利用するサービスについて紹介します。

後半ではCI／CDでおこなう一連の処理にまとまりである**ワークフロー**について、例をもとに説明していきます。さらに実際の**CI／CDサービス**である**Bitrise**について説明し、説明したワークフローの設定の仕方について紹介していきます。ワークフローと実際の設定例を知ることで自身のプロジェクトで導入する際の参考になればと思います。

9.1 CI（継続的インテグレーション）

CIとは、変更を**バージョン管理システム**にプッシュするたびにビルドを実行するものです。ここでいうビルドは、アプリのビルドだけでなく自動テストなどの手順すべてを含みます[注1)]。

本節では、そのCIについて説明するにあたり次のトピックについて順に説明します。

- CIを実施するメリット
- CIで必要な機能
- CIをおこなう上で守るべきこと

これらを知ることでCIとはなにかについて理解できます。その後に、CDについて説明していきます。

9.1.1 CIを実施するメリット

CIを実施するメリットには次のようなものがあります。

注1) 本章では単にビルドと書かれている場合は、CIでおこなう手順すべてを指します。

- 開発者の生産性の向上
- プロジェクトの見える化
- 環境に対するリスクの軽減

開発者の生産性の向上

開発中は自分の手元にだけコードがあり、他の箇所に影響を与えていても気づかないことはよくあります。その結果、他の開発者とのコードとの統合作業に時間がかかってしまうことがあります。

コードの変更単位で都度ビルドを実行することで、問題があればすぐに見つけられます。また問題を見つけるにはアプリのビルド以外にも自動テストなどの実施も重要です。早めに問題が見つかれば原因を把握するコストが少なくなるため、修正コストが低くなります。結果としてより迅速に開発ができることに繋がります。

ビルドが継続的に動き続けていることで、コードに問題が起きていないことがわかり開発者の自信にもつながります。このようなサイクルが揃った環境では、プッシュしたコードに問題があってもすぐに検知しフィードバックしてくれます。そのため開発者は安心して作業を続けられるため、生産性向上が見込めます。

プロジェクトの見える化

CI によって常にビルドを回し続けることは、その iOS プロダクトのプロジェクト（以後、プロジェクトと呼びます）における健康状態を可視化しているといえます。健康状態を知ることはプロジェクトにおいても重要です。

CI を活用することで、Lint をもちいたコードの静的解析、ビルドにかかった時間やテストのカバレッジ、そして次で紹介する CD まで含めればリリースまでにかかった時間など、プロジェクトの今の健康状態がわかります。

これらの情報を常に得ることでプロジェクトの健康状態の変化がわかり、異変がおきていないかがわかります。

環境に対するリスクの軽減

iOS アプリのビルドや自動テストを安定して実行するためには、再現性のある環境構築が必要です。再現性のある環境とは、誰でも同じようにビルドや自動テストが実行できる環境のことです。

ある人の環境では自動テストが動くけれど、他の人の環境では動かないといったことが起きたことはないでしょうか。必要なセットアップ作業が追加されていてもその情報が他の人には共有されていなかった、ということもあります。

特定の誰か以外の環境でもビルドができていることを確認できるのは、環境構築に対するリスク

を軽減しているといえます。

CI ／ CD 環境で継続的にビルドするには、このような再現性のある環境が必要です。クラウド型サービスとして CI ／ CD 環境を提供するサービスの多くはこの再現性のある環境が用意されています。ただし、自身でローカル環境のビルドマシンを CI ／ CD 環境として構築する場合は、環境構築手順漏れがないようセットアップに対して注意が必要です。

9.1.2 CI で必要な環境

CI 環境を構築するためには、CI ／ CD サービス以外に次が必要です。

- バージョン管理システム
- 自動ビルド（ビルドスクリプト）
- フィードバック

これらについて順に説明していきます。

バージョン管理システム

CI では、統合されたソースコードが問題なくビルドできるか確認します。そのためには、ソースコードが変更されたタイミングでビルドするとよいです。このタイミングを検知するには「バージョン管理システム」を利用します。

今では Git を利用しているケースが多く、特に「GitHub」や「GitLab」などのバージョン管理サービスを利用するのが一般的です。

自動ビルド（ビルドスクリプト）

通常の開発作業では Xcode などからアプリのビルドや自動テストを実行することがほとんどです。

しかし CI ／ CD サービス上のビルドマシンで CI をおこなうには、コマンドライン上からビルドや自動テストを実行できる必要があります。そのためには、**ビルドスクリプト**が必要になります。

用意したビルドスクリプトが実行する環境によって結果が変わったり、たまに失敗したりするようでは安心して継続的にビルドできません。そこでこのビルドスクリプトはプロダクトのコードと同じように扱う必要があります。問題なく動き、なおかつよくテストされているべきです。

ビルドスクリプトを用意するにあたって、自前で用意する以外に CI ／ CD サービスが機能として提供しているケースがあります。その提供された機能を利用するのもよいです。

自前で用意する場合は **xcodebuild** コマンドを利用したビルドスクリプトを 1 から用意することになります。ただし iOS プロダクトにおいては、この xcodebuild コマンドを内部で利用した

fastlane[注2)]がよく利用されます。こちらは Fastfile ファイルを用意し、設定などを書くだけで比較的かんたんにビルドスクリプトを用意できます。

どちらを利用するにせよ、自身のプロジェクトで実行できるビルドスクリプトを用意する必要があります。

フィードバック

CI の結果は、何かしらの手段をとおしてフィードバックできる必要があります。このフィードバックには次の 2 つの手段があります。

- Pull 型（自ら結果を取得しにいく）
- Push 型（特定サービスに結果がフィードバックされる）

Pull 型のようにフィードバックを得るために CI ／ CD サービスにアクセスするのは、アクセスした時点で結果が出ているかどうかもわからないことがあり面倒です。ただし、今までのビルド履歴が分かるようになっている便利な面もあります。

Push 型では普段利用しているようなサービスにたいして結果をフィードバックさせるのがいいです。たとえば、チャットツールである Slack などのサービスを利用している方は多いでしょう。

ただし Push 型を利用する場合は、過度なフィードバックに伴う通知に注意する必要があります。あまりにも通知が多いと無視されてしまうかもしれません。結果のフィードバックは必要ですが、フィードバックの回数や内容についてはよく吟味すべきです。

9.1.3 CI をおこなう上で守るべきこと

CI はあくまでも仕組みでありツールそのものをさすわけではありません。

コードが問題ないかを確認するためにも、手元にあるコードを定期的にバージョン管理システムにプッシュすることは重要です。またそれ以外にも次の守るべきことがあります。

- ビルドが失敗したら修正する
- テストが失敗したときにコメントアウトで対応しない
- ビルドの実行時間を意識する

これらを守らないと CI による恩恵を得られません。

ビルドが失敗したら修正する

継続的にビルドすることで問題が早期に発見できるようになります。しかしその問題は早めに解決しなければ意味はありません。

注2） https://github.com/fastlane/fastlane