Android テスト全書

白山 文彦
外山 純生
平田 敏之
菊池 紘
堀江 亮介

PEAKS

はじめに

本書を手にしてくださってありがとうございます！

この本を手にされたということは、Androidアプリの品質に強い関心をお持ちであることは間違いないでしょう。本書は「全書」というそのタイトルが示すとおり、Android開発におけるテストについて文字どおり「すべて」を網羅すべく企画しました。

すなわち、各モジュールが適切に動作しているか検証する**ユニットテスト**の書き方に始まり、UI操作によって引き起こされる画面の変化を検証する**UIテスト**、**JUnit 5**のような最新かつ今後主流になりうるテストフレームワーク、果ては開発現場で運用されている**CIシステム**や、日々のリリースを便利にする**CD**の知見などを惜しげもなくカバーしています。

筆者がこの本を最初に構想したきっかけは、これら有益な情報がいまだに各社または特定個人に「秘伝のタレ」のように受け継がれるのみでAndroidコミュニティ全体に還元されている状況とはいえない現状を何とかしたいという思いからでした。Androidは本書執筆の2018年にちょうど誕生10周年を迎えます。この間Androidのテスト手法は着実に進化してきましたが、ウェブで情報を検索するとまだまだ黎明期のノウハウがヒットすることもあるなど、これからテストを書いていこうという人たちには若干情報の取捨選択に苦労を伴う側面があるのも事実です。

そこで普段からテストやCI/CDに対して問題意識を持っている著者陣が集まって、この節目の2018年時点での決定版をここにまとめようではないかという結論に至りました。したがってこの本を読んでいただければ、今までユニットテストを書いたことのない人も、UIテストが1つもなかったプロジェクトでも、最先端のAndroidアプリ開発現場で実際に使われている生きた知見を手にすることができます。本書では各章の著者がこれまでの豊富な経験を基に、「単に公式サイトのチュートリアルをまとめただけ」から一歩踏み込んだ「かゆいところに手が届く」内容をみなさまにお届けします。楽しみにしていてください。

対象読者

本書は文字どおりAndroidのテストに関心のあるすべての方を対象としています。

- テストを書いた方がいいと思いつつ書けていない方
- テストを書いているが、もっといい書き方があると思っている方
- さらに発展的な内容を求める方
- 安定したアプリを継続的にリリースしたいすべてのAndroidエンジニア

本書の手引き

本書は全部で8章構成ですが、大まかに次のように成り立っています。

第1章ではこの後に続くすべての章の礎となるべく、Androidのテストにはどういったものが含まれるのか、それらの守備範囲はどこからどこまでなのかといった基礎的な内容をカバーしています。

第2章と第3章では、主にユニットテストにテーマを絞ってはじめてテストを書くところから始まり、実際の開発現場で使える**アサーション**や**モック**などのテクニックについて実践的な内容を紹介しています。

第4章〜第6章では、主にUIテストをテーマにしてUIテストを自動化する際の心構えに始まり、**Espresso**や**Appium**といったUIテストフレームワークの生きた知見が得られます。

第7章では、今後主流になっていくであろう**JUnit 5**を取り上げ、より便利になった**パラメタライズドテスト**などの最新情報を学ぶことができます。

最後に第8章では**CI/CD**をテーマに、これまでの章で紹介した内容を開発フローに組み込んで継続的に実行したり、ビルドフローを自動化したりするといった内容で締めくくっています。

特に理由がない場合は第1章から順にお読みいただくことを想定していますが、これらはまとまりごとに独立して読むことができますので、すでにAndroidのテストに精通した読者の方は目次を見て好きな部分から読んでいただいても構いません。

想定環境

本書では表1の開発／実行環境を想定しています。

● 表1　想定環境

Android Studio	3.1.4
Android Plugin for Gradle	3.1.4
JDK	Android Studio ビルトインの JDK[注1)]

Googleの開発者カンファレンスGoogle I/O 2017でKotlinがAndroidの公式開発言語に加わり、Google I/0 2018ではプロのアプリ開発者の35%以上がすでにKotlinを使っていることが発表されました[注2)]。このような現状を踏まえ、第2章や第3章、第7章など一部の章ではサンプルコードに積極的にKotlinを採用しています。その際に利用したKotlinのバージョンは次のとおりです（表2）。

● 表2　Kotlin のバージョン

Kotlin	1.2.51
Kotlin Gradle Plugin	1.2.51

サンプルコードにKotlinを利用した章でも、現在もJavaを使っている読者が問題なく読み進められるように適宜注釈を入れるなどして読みやすい書面になるように心がけました。また、第5章のようにライブラリや周辺ツールとの相性に鑑み、Javaのまま説明した章もあります。詳しくは各章の解説をご覧ください。

サンプルコード

次のリポジトリに本書のサンプルコードが掲載されています。

- https://github.com/peaks-cc/android_testing_samplecode

各章でサンプルコードの指定がある場合は、そちらも合わせて確認ください。

注1）OpenJDK 1.8.5_152
注2）Developer Keynote (Google I/O '18) https://youtu.be/flU42CTF3MQ?t=7m29s

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、850人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、かなら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。

はじめに 3

対象読者 4
本書の手引き 4
想定環境 5
サンプルコード 5
クラウドファンディングとPEAKS 6
免責事項 6

第1章　テスト入門 15

1.1 なぜテストが必要なのか 15
1.2 Androidのテストの種類と手法 16
1.2.1 ブラックボックステストとホワイトボックステスト 16
1.2.2 Androidのテストの実行方法 16
1.3 自動化されたテストの分類と役割 17
1.3.1 ユニットテスト 17
1.3.2 インテグレーションテスト 18
1.3.3 UIテスト 19
1.4 テストのピラミッド 20
1.5 テストがもたらすさまざまなメリット 21
1.5.1 テストは動く仕様書 21
1.5.2 急がば回れ 22
1.5.3 テストは資産 22
1.6 まとめ 23

第2章　ユニットテスト実践入門 25

2.1 ユニットテストとテスティングフレームワーク 25
2.2 JUnit 4を使ったはじめてのユニットテスト 26
2.2.1 テスト対象 26
2.2.2 テストクラスとテストケース 27
2.2.3 アサーション 29
2.2.4 マッチャー 29
2.2.5 はじめてのユニットテスト 29
2.2.6 テスト対象メソッドの実装 31

2.2.7 テストケースの追加 32
2.2.8 テストケースごとの初期化・後処理 34
2.2.9 テストケースの例外を検証する 35
2.2.10 テストケースを一時的にスキップする 36
2.2.11 JUnit 4のテストランナー 37
2.2.12 JUnit 4のテストルール 37
2.3 テストケースを減らすためのテクニック 38
2.3.1 同値分割 38
2.3.2 境界値 38
2.3.3 単項目チェック 39
2.4 AssertJを使ったアサーション 41
2.4.1 AssertJとは 41
2.4.2 はじめてのAssertJ 41
2.4.3 文字列のアサーション 42
2.4.4 数値のアサーション 45
2.4.5 IterableやCollectionのアサーション 45
2.4.6 AssertJの詳細な例外検証 47
2.4.7 まとめ 47
2.5 テストダブルを使った依存コンポーネントの差し替え 48
2.5.1 テストダブル 49
2.6 Mockitoでテストダブルを便利に扱う 58
2.6.1 Mockitoとは 58
2.6.2 Mockitoを使ったスタブ 59
2.6.3 Mockitoを使ったモックに対するメソッド呼び出しの検証 65
2.6.4 Mockitoを使ったスパイ 67
2.6.5 Mockito Tips 71
2.7 まとめ 74

第3章 ユニットテスト応用編 75

3.1 Androidフレームワークのコードを使ったユニットテスト 75
3.1.1 はじめてのRobolectric 76
3.1.2 RobolectricとAPIレベル 77
3.1.3 RobolectricとAndroid Jetpack 78
3.2 SQLiteを利用したモジュールのLocal Unit Test 80
3.2.1 Roomの利用例 80
3.2.2 RoomとLocal Unit Test 82
3.3 テストの構造化による可読性の向上 85

3.3.1 Enclosedランナー ... 85
3.4 非同期処理のユニットテスト ... 88
3.4.1 非同期処理のユニットテストが困難である理由 ... 89
3.4.2 CountDownLatchを使った非同期コールバックの待ち合わせ ... 92
3.4.3 コールバック内でアサーションしない改善手法 ... 95
3.5 RxJavaのテスト ... 99
3.5.1 シンプルなObservableのテスト ... 99
3.5.2 内部にRxJavaを使ったモジュールのテスト ... 101
3.6 OkHttpとRetrofitを使ったモジュールのユニットテスト ... 106
3.6.1 OkHttpとRetrofitの準備 ... 106
3.6.2 MockWebServer ... 108
3.7 MVPアーキテクチャのテスト ... 112
3.7.1 MVPアーキテクチャとは ... 112
3.7.2 MVP各層のテスト ... 114
3.8 テストのないプロジェクトにテストを導入する ... 117
3.9 まとめ ... 125

第4章 UIテスト（概要編） 127

4.1 UIテストの自動化をはじめる前に ... 127
4.1.1 目的を整理する ... 128
4.1.2 自動化する範囲を決める ... 129
4.1.3 テストツールを選択する ... 131
4.1.4 スケジュール・予算・体制を決める ... 131
4.2 テストツール選択のポイント ... 134
4.2.1 EspressoとAppiumの概要 ... 134
4.2.2 EspressoとAppiumの観点別比較 ... 135
4.2.3 自動化の難易度を決定付けるポイント ... 137
4.2.4 自動化が不可能、または非常に困難なもの ... 137
4.2.5 自動化できるが、Appiumでは工数がかかるもの ... 138
4.3 長くテストコードを利用し続ける ... 141
4.3.1 テストコードにPageObjectデザインパターンを適用する ... 141
4.3.2 CI環境でテストコードを動かし続ける ... 146
4.3.3 検証結果をわかりやすく共有する ... 147
4.3.4 不安定なテストを対策する ... 147
4.3.5 実行時間を短縮させる ... 148
4.3.6 開発やリリースのフローに組み込む ... 148
4.4 まとめ ... 149

5.1 概要 152
5.2 セットアップ 152
5.2.1 ビルドスクリプトの修正 152
5.2.2 必要に応じて設定するもの 154
5.2.3 端末のアニメーション設定 155
5.3 Espresso Test Recorderを使ってみる 156
5.3.1 サンプルアプリの準備 156
5.3.2 Espresso Test Recorderの実行 157
5.3.3 操作の記録 157
5.3.4 アサーションの記録 158
5.3.5 テストコードの生成と実行 160
5.4 Espresso APIの基本 163
5.4.1 テストクラスの作成と構造 163
5.4.2 Espressoテストコードの基本構造 165
5.4.3 レイアウトインスペクタ 166
5.4.4 Viewの特定、検証でよく使うAPI 168
5.5 画面更新を待ち合わせる 173
5.5.1 自動同期機能が提供してくれること・してくれないこと 174
5.5.2 自動同期機能だけではうまく動作しないときは 174
5.5.3 IdlingResourceの登録・解除 175
5.5.4 Espressoが提供しているIdlingResource実装を活用する 178
5.5.5 サードパーティーが提供しているIdlingResource実装を活用する 181
5.5.6 明示的な待ち合わせ処理を行う 182
5.5.7 まとめ 187
5.6 PageObjectデザインパターンの実践：長く使われ続けるテストにするために 187
5.6.1 操作の記録とテクニック 188
5.6.2 自動生成のテストコードを手直しする 190
5.6.3 生成されたテストコードをリファクタリングする 197
5.6.4 メソッドを抽出する 198
5.6.5 PageObjectを形作る 201
5.6.6 テストケースを実装する 203
5.7 目的別テクニック 205
5.7.1 RecyclerViewを操作する 205
5.7.2 WebViewを操作する 209
5.7.3 Viewの状態を知る 211
5.7.4 パーミッション要求ダイアロに対応する 212
5.7.5 カスタムマッチャー・カスタムアクションを書く 214
5.7.6 自分でIdlingResourceを実装する 217
5.7.7 外部アプリとのIntent送受信をテストする 220

5.8 まとめ 223

第6章 UIテスト（Appium編） 225

6.1 Appiumの概要 225
6.1.1 アーキテクチャ 226
6.1.2 Desired Capabilities 227
6.1.3 利用する言語・テスティングフレームワークの選定 228
6.1.4 Appiumを知るための参考資料 229
6.2 Appiumを利用する 230
6.2.1 Appiumのセットアップ 230
6.2.2 Appiumを用いたサンプルコード 231
6.3 UIテストを実装する 235
6.3.1 利用するライブラリの例 235
6.3.2 テストコードのディレクトリ構成 236
6.3.3 Appium Desktopを利用する 237
6.3.4 UI要素のみつけ方と操作 241
6.3.5 UIテストのサンプルコード 244
6.4 UIテストをより良くする 250
6.4.1 実行する環境を変更できるようにする 250
6.4.2 テストコード失敗時に調査しやすいようにする 251
6.4.3 スクリーンショットを撮る 252
6.5 プロダクトにUIテストを導入するときに決めること 254
6.5.1 UIテストを利用する目的 254
6.5.2 UIテストで自動化する範囲の方針 255
6.5.3 UIテストの運用 255
6.6 実プロダクトへのUIテストの導入 256
6.6.1 プロダクトの説明 256
6.6.2 プロジェクト内で決めたこと 257
6.6.3 テストケースを考える 258
6.6.4 テストを実行する対象を決める 259
6.7 導入したテストコードの実装例とその説明 260
6.7.1 既存会員のテスト 260
6.7.2 新規会員のテスト 270
6.7.3 補足：WebViewを扱うテスト 277
6.8 長くテストコードを利用しつづけるための実践編 278
6.8.1 自身の環境以外でも動かせるようにする 279
6.8.2 実行時間を短縮させる 279
6.8.3 各フローへの組み込み 280
6.9 まとめ 281

第7章　JUnit 5　283

7.1 AndroidとJUnit 5 283
7.1.1 android-junit5プラグイン 284
7.1.2 導入要件 284
7.1.3 Local Unit TestとInstrumented Testに導入する 285
7.1.4 Java 8言語機能に対応する 288
7.2 テストの実行方法 288
7.2.1 Android Studioからの実行方法 288
7.2.2 Gradleのタスクとしての実行方法 289
7.2.3 テストコードの配置と注意点 290
7.3 JUnit 4からの変更点 291
7.3.1 APIのパッケージ名（名前空間） 291
7.3.2 アノテーション 291
7.3.3 テストケースの分類 292
7.3.4 テストケースのネスト 297
7.4 パラメタライズドテスト 298
7.4.1 特定の固定値を使用する（@ValueSource） 299
7.4.2 指定したEnumのメンバーを使用する（@EnumSource） 300
7.4.3 指定したメソッドの戻り値を使用する（@MethodSource） 302
7.4.4 CSV形式の文字列、またはCSVファイルの内容を使用する（@CsvSource、@CsvFileSource） 304
7.4.5 引数を提供する特別なクラスを作成して使用する（@ArgumentsSource） 306
7.4.6 引数を提供する特別なクラスにパラメータを与える（@ArgumentsSource） 308
7.5 JUnit 4 APIで記述したテストコードとの互換性 310
7.5.1 カテゴリからタグへの自動変換 310
7.5.2 Ruleの使用制限 310
7.6 よく使用されるライブラリとの相性 311
7.6.1 Robolectric 311
7.6.2 Espresso 312
7.7 まとめ 314

第8章　CI/CD　315

8.1 継続的インテグレーション 315
8.1.1 導入のメリット 316
8.1.2 CIに必要な機能 317
8.1.3 CIツール・サービスの選定基準 318
8.1.4 プロジェクトへの導入 320
8.2 継続的デリバリーとデプロイメントパイプライン 321
8.2.1 開発からリリースまでの流れの見える化 322
8.2.2 適切なフィードバックの実現 322

8.2.3 反復可能で信頼できるリリースプロセスの確立 ... 322
8.2.4 デプロイメントパイプラインを実現するには ... 323
8.3 デプロイメントパイプラインの構築 ... 324
8.3.1 テスト戦略 ... 325
8.3.2 CircleCIの導入 ... 326
8.3.3 config.ymlの作成 ... 327
8.3.4 セットアップと初回ビルド ... 327
8.3.5 デプロイメントパイプラインの各ステージの定義 ... 328
8.3.6 Workflowによるシンプルなデプロイメントパイプラインの構築 ... 329
8.4 コミットステージの構築 ... 330
8.4.1 Androidプロジェクトのビルド設定 ... 330
8.4.2 ビルドスクリプト：コンパイル ... 331
8.4.3 ビルドキャッシュの利用 ... 332
8.4.4 インスペクション：ツールの導入 ... 333
8.4.5 継続的テスト ... 335
8.4.6 環境変数の利用 ... 339
8.4.7 DeployGateによるアプリ配信：継続的デプロイ ... 341
8.4.8 作業環境の共有化 ... 343
8.5 受け入れテストステージの構築 ... 344
8.5.1 UIテストの実行手順 ... 344
8.5.2 Firebase Test Labのセットアップ ... 345
8.5.3 gcloudを利用したCIテスト実行環境の整備 ... 346
8.5.4 デバッグビルドで利用する署名を指定 ... 347
8.5.5 UIテストの実行 ... 348
8.5.6 テストパラメータの整理 ... 350
8.5.7 CircleCIへのテスト結果の保存 ... 351
8.5.8 config.ymlの完成 ... 353
8.6 手動テストステージの構築 ... 354
8.6.1 手動テストステージの準備 ... 354
8.6.2 Instabugの導入 ... 355
8.6.3 アプリ配布の実行 ... 358
8.6.4 完了処理 ... 359
8.7 リリースステージの構築 ... 360
8.7.1 リリース署名を行う：鍵を守る ... 360
8.7.2 Google Play Developerコンソールへのアップロードの自動化 ... 363
8.7.3 Workflowへの組み込み ... 363
8.7.4 アプリの公開 ... 364
8.8 デプロイメントパイプラインのステージ進行 ... 365
8.8.1 filtersによるステージの進行制御 ... 366
8.8.2 Manual Approvalsによるステージの手動進行制御 ... 368
8.9 まとめ ... 370

付録A　APPENDIX　371

A.1 Kotlinを使うための設定 371
A.1.1 新規プロジェクトでKotlinを使う 371
A.1.2 JavaプロジェクトをKotlin対応させる 372
A.2 JUnit 4のテストクラスごとの初期化と後処理 373
A.3 Mockito-Kotlinのセットアップ 374
A.3.1 Mockito-Kotlinのインストール 374
A.3.2 kotlin-allopenを使ってモックする 374
A.3.3 mock-maker-inlineを使ってモックする 375
A.4 呼び出し元と同じスレッドでタスクを処理するExecutorService 376
A.5 RxJava/RxAndroidのPluginでスケジューラ差し替え 377

おわりに　381

謝辞 381
権利表記 382
著者紹介 387
白山 文彦 @fushiroyama 397
外山 純生 @sumio_tym 397
平田 敏之 @tarappo 397
菊池 紘 @kikuchy 397
堀江 亮介 @Horie1024 397
日高正博 @mhidaka 397

第 1 章

テスト入門

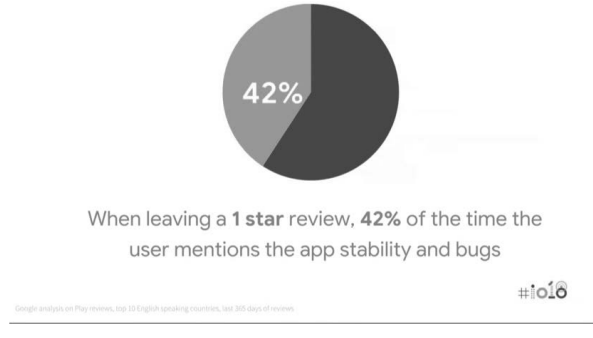
白山 文彦／@fushiroyama

本章では他のすべての章に先立ち、そもそもなぜテストを書くのか、どうしてAndroidアプリ開発においてテストが重要なのかをデータを元に確認します。また、Android開発ではどのようなテストが活用できるのか、それらをどのくらいの割合で作成・実行していけばよいのかについても解説します。これからAndroidのテストを書いていきたいけれどもどうしてよいか分からない方は、ぜひこの章から読み始めてください。

1.1 なぜテストが必要なのか

2018年現在、ネイティブアプリは今もって絶頂期にあり、主要なビジネスでネイティブアプリを提供していないものは少数派といえます。アプリとウェブの橋渡し役としてPWA[注1)]などのハイブリッドテクノロジーも出てきましたが、広く一般に使われるにはもう少し時間がかかるでしょう。

このような状況にあって、アプリの品質はそのアプリの生死を左右する重要な要素です。GoogleはGoogle I/O 2018のセッションで、Play Storeで星1レビューのついた原因分析を公開しています[注2)]。図1.1が示すように、実に42%もの割合がアプリのクラッシュを含む安定性の問題に起因していることを明らかにしています。



● 図 1.1　出典：Improve app performance and stability with Firebase (Google I/O '18)

注 1）Progressive Web Apps https://developers.google.com/web/progressive-web-apps/
注 2）Improve app performance and stability with Firebase (Google I/O '18) https://youtu.be/gpoqgQPwu_k?t=1m10s

さらにGoogleはそのような品質の低いアプリをPlay Storeのランキングや検索結果の上位には表示されにくくするようアルゴリズムを改良したことも報告[注3]しています。

したがってアプリ開発者はユーザーにすばやく価値を届けるのみならず、その価値提供を安全かつ安定的に成し遂げることを求められています。そのためにテストは必要不可欠であり、開発プロセスに強固なテスト基盤を構築することは品質を確保するための強力な処方せんとなります。

1.2 Androidのテストの種類と手法

本書ではAndroidにおけるさまざまなテストの種類や手法を紹介しています。まずAndroid開発ではどのような種類のテストが行われるのか、それらをどのように実行するのかを解説します。さらに各々のテストが紹介されている章を示します。これから本書を読み進める際の手引きとして利用してください。

1.2.1 ブラックボックステストとホワイトボックステスト

テスト手法のうち、決められた操作や入力に対して期待どおりの結果や出力が得られることを確認するテストを**ブラックボックステスト（Black-box Testing）**と呼びます。ブラックボックステストでは、外部から観測できる結果が仕様書に沿っていることが重要であり、内部的にどのように実装されているかは考慮されません。

これに対して、プログラムの内部構造に着目し、処理や分岐が正しく行われているかを確認するテストを**ホワイトボックステスト（White-box Testing）**と呼びます。ホワイトボックステストでは内部的なロジックのチェックに重きが置かれているため、仕様を勘違いしていた場合にホワイトボックステストとしては正しいがブラックボックステストとしては期待どおりではないということが起こりえます。

Androidのテストにはブラックボックステストとホワイトボックステストの両方が用いられます。

1.2.2 Android のテストの実行方法

Androidのテストの実行方法は次の2つに大別できます。すなわち人間が手動で行うテストと自動化されたテストです。

「Androidのテスト」と聞いて最初に思いつくテストは、人間が手動で行うテストでしょう。開発者自らが実機やエミュレーターにアプリをインストールして確認することもあるでしょうし、専任のQA[注4]エンジニアに総合的な動作確認を依頼している会社も多いでしょう。

注3） How we're helping people find quality apps and games on Google Play https://android-developers.googleblog.com/2017/08/how-were-helping-people-find-quality.html

注4） Quality Assurance の略。品質保証のこと

人間が手動で行うテストでは、検証者がアプリのいちユーザーとして仕様書どおりに動作しているかを横断的に確認します。したがって一般的にブラックボックステストとして実施されることが多いです[注5)]。この方法のメリットは、人間の目でより細かいケアができるので思いもよらない難解な手順のバグを発見できることがある点や、よりユーザーに近い目線から品質をチェックできることから結果的にアプリそのもののユーザー体験を向上できるチャンスが得られる可能性がある点などが考えられます。

反面、時間と手間をかけてビルド・配布したにもかかわらず画面を開いただけでクラッシュするような凡ミスを犯しているようなケースでは無駄が発生してしまいますし、単調で機械的な検証を人間がしらみつぶしに行うことはコストが高く、必ずしも効率的とはいえません。また、仕様書から漏れている部分に存在するバグの発見はQAエンジニアの職人技に依存することになります。したがって仕様書が不十分だと属人性が高く、品質のばらつきが発生する可能性があります。

一方、Androidのテストでは自動化されたテストも広く行われています。本書で扱う内容はこの自動化されたテストです。さきほどブラックボックステストとは決められた操作や入力に対して期待どおりの結果や出力が得られることを確認するテストと述べました。この一連の操作をプログラムに落とし込んで自動化できれば、人間がやるには単調でモチベーションを維持しづらいような確認事項も属人性がなく品質も均質なチェックを実現できます。

ホワイトボックステストも自動化されたテストとして広く実施されています。たとえばユーザーの目からはシンプルな1つの操作にみえることが、実際には複数のプログラムの相互作用によって実現されていることは日常茶飯事です。これらの個々のプログラムが正しく機能することを確認するテストは、内部実装を知っている開発者自らが書く必要があります。

1.3 自動化されたテストの分類と役割

前節でみたように自動化されたテストといってもその目的や粒度によってさらに細分化できることがわかります。本節ではさまざまな種類の自動化されたテストを紹介し、その分類や役割について解説します。

1.3.1 ユニットテスト

ユニットテスト（Unit Testing）はソースコードの個々の**ユニット（構成単位）**が意図した振る舞いをしているか検証するテストです。ソフトウェアを開発する際には一連のデータや処理を**モジュール**と呼ばれる意味のあるまとまりに分割しながら構築します。ユニットテストはこのモジュールが想定するとおりに動作していることを確認するために開発者自らが記述します。

注5) 人間が手動でホワイトボックステストを行うには、デバッガを使って無理やり通常通らない分岐を実行させる必要があるなどの理由から、それほど広くは行われていません

よりAndroidの実際に即した説明をするならば、クラスごとにメソッドが一定の入力に対して期待どおりの値を返すか、あるいは呼び出しに応じてプロパティが適切に変化するかといった内容を保証するために書かれるプログラムのことを指します。

Androidアプリは膨大な数のクラスの相互作用によって成り立っているので、安定したアプリ開発には個々の構成単位が信頼の置けるモジュールであることが必要不可欠です。したがってモジュールの動作を保証するユニットテストはすべての自動化されたテストの基礎であり、もっとも頻繁に実行されるテストであることを暗黙的に示唆します。

Android開発におけるユニットテストは実行環境の違いからさらに2つに細分化されます。

Local Unit Test

Local Unit Testは実機やエミュレーターを利用せずに開発マシンから直接高速に実行できるユニットテストの手法です。ユニットテストを書く対象のモジュールがAndroidフレームワークのAPIに依存していない[注6)]場合、Local Unit Testの採用を検討すべきです。普段からユニットテストを書きつつコードの構造化を意識しながら開発しているとAndroidフレームワークに依存したコードを書かざるを得ないケースは思ったほど多くありません。

またAndroidフレームワークに依存したコードもLocal Unit Testとして実行するためのテクニックやノウハウがありますので、詳しくは第2章「ユニットテスト実践入門」や第3章「ユニットテスト応用編」を確認してください。

Instrumented Unit Test

Instrumented Unit Testは実機やエミュレーターを使ってユニットテストを実行する手法です。

Android開発では、実機やエミュレーターを使った自動テスト全般のことを**Instrumented Test**と呼びます。その中で実行するテストの種類をユニットテストに絞ったものがInstrumented Unit Testです。

Instrumented Testは開発したプロダクションコードと検証のためのテストコードを、いずれも実際にユーザーが触れる形式に変換してAndroid上で実行します。そのため本番に近い信頼性の高いテスト結果を得ることができます。

反面、ビルド・配布に時間がかかる上にモバイルという実行速度も限られた状況で実施するテストなので時間的コストが高いことが最大のネックです。

1.3.2 インテグレーションテスト

インテグレーションテストは結合テストとも呼ばれ、複数のモジュールを組み合わせたときに正しく動作することを検証するために行うテストです。

注6) `android.*` 名前空間のクラスを利用していないコード。`android.content.Context` を使ってリソースをロードしている箇所などは Android フレームワークに依存しています

インテグレーションテストがカバーする内容は組織やコンテキストによって広範にわたります。Androidデベロッパー向け公式サイトの「テストの基礎（Fundamentals of Testing[注7]）」によると、たとえばServiceやContentProviderのようにユーザーが直接触れることができず、かつ複数の画面にまたがって協調して動作するようなコンポーネントのテストが例としてあげられています。

インテグレーションテストはテストのしづらさや複数コンポーネントをテストしたいという要求上、より本番に近い結果を求められることから、実機やエミュレーターを使ってテストすることが一般的です。

1.3.3 UIテスト

UIテストは文字どおりユーザーが実際にアプリの**UI（ユーザーインターフェイス）**を操作したときに画面が正しく反応できるかといった内容を検証するためのテストです。このようなテストを表す用語はいくつかありますが、本書ではAndroidの公式ドキュメント[注8]で使われている**UI Testing**にならってUIテストと表記します。

UIテストのもっとも基本的な形態は人間が実際にアプリをビルド・インストールして自ら触ってみるというものですが、これは時間がかかるだけでなくミスが多くなりがちで信頼性が十分とはいえません。したがってミスを減らす意味でもUIテストの自動化は重要です。また、画面単位で「ボタンをクリックするとポップアップが表示される」とか「規定文字数以上タイプすると送信ボタンが有効になる」といったシンプルな操作を自動化できるとQAコストの削減にもつながるでしょう。

自動化されたUIテストの実行環境は、おのずと実機またはエミュレーターとなるためInstrumented Testに分類されます。

UIテストも1つの画面をテストする場合と複数画面の連携やシステムアプリ[注9]との相互作用といった複雑な動作の検証[注10]とでは、テスト戦略も使用するツールも変わってきます。本書では第4章、第5章、第6章でUIテストの実行戦略からツールの使用方法まで詳細に解説しています。

注7） Fundamentals of Testing https://developer.android.com/training/testing/
注8） https://d.android.com/training/testing/ui-testing/
注9） Android搭載端末にプリインストールされているアプリ
注10） このようなテストはE2E（エンドツーエンド）テストと呼ばれることも多いです

第 2 章

ユニットテスト実践入門

白山 文彦／@fushiroyama

いよいよ本章から実際にユニットテストを書いていきます。本章では特にAndroid実機やエミュレーターを利用せず開発マシンから直接実行するLocal Unit Testに的を絞って解説します。Androidフレームワークのコードを利用したモジュールのユニットテストに関しては第3章「ユニットテスト応用編」を参照してください。

「はじめに」の「想定環境」で触れたように、本章と第3章ではサンプルコードにKotlinを用います[注1)]。

2.1 ユニットテストとテスティングフレームワーク

ユニットテストとはソースコードの個々のユニットが意図した振る舞いをしているか検証するテストです。具体的にどのように記述し、どのように実行するのでしょうか。一例としてKotlinのassert()関数を使った方法が考えられます（リスト2.1）。

● リスト 2.1　assert() 関数を使ったユニットテスト

```
assert(1 + 1 == 2)
assert("foo" + "bar" == "foobar")
```

assert()関数は、引数の条件式が真の場合は正常終了し、偽の場合はjava.lang.AssertionErrorでエラー終了します。

これをコンパイルして実行すると、確かにユニットテストとしての目的は果たしますが、この方法はいくつも問題を抱えています。

- 検証に失敗した場合にjava.lang.AssertionErrorとしか表示されず、エラー理由がわかりにくい
- 途中でAssertionError が発生すると、それ以降の行は評価されない
- 管理が煩雑で、コンパイルや実行も手動で行う必要がある

注 1)　これまで Kotlin に触れたことがない読者も読み進められるよう注意しました。より詳しく Kotlin について知りたい方は次の専門書を参考にしてください。長澤太郎 (2016)『Kotlin スタートブック』リックテレコム

実際の開発現場では、ユニットテストを快適に記述し、実行し、結果を検証するためのフレームワーク[注2)]を用います。それらは**テスティングフレームワーク（Testing Framework）**と呼ばれます。

2.2 JUnit 4を使ったはじめてのユニットテスト

JUnitはJavaやAndroid開発で広く用いられているテスティングフレームワークで、次のような特徴を備えています。

- アノテーションを用いたわかりやすい記法でユニットテストを快適に記述できる
- テストの検証結果をわかりやすく伝える仕組みをもつ
- Maven[注3)]やGradle[注4)]といった主要なビルドシステムと連携し、簡単にビルド・実行できる

Android開発では原稿執筆時点の2018年9月現在、**JUnitバージョン4.12（JUnit 4）**がデファクトスタンダード[注5)]です。本章で解説するLocal Unit Testでも、第4章以降で解説するUIテストを含むInstrumented TestでもJUnit 4を利用することができます。

それでは早速JUnit 4でユニットテストを書いてみましょう！

2.2.1 テスト対象

テストの対象となるプロダクションコードを本書では**テスト対象**と呼び、クラスを指す場合はテスト対象クラス、メソッドを指す場合はテスト対象メソッドと呼ぶことにします。ここでは入力ボックスを想定し、正しい文字列が入力されたかを真偽値で返すInputCheckerクラスを作成します（リスト2.2）。

● リスト2.2 テスト対象クラスの作成

```
class InputChecker {
  fun isValid(text: String?): Boolean {
    return false
  }
}
```

いまのところ、InputChecker#isValid()メソッドは単にfalseを返す仮実装で構いません。

注2） 必要とされる機能をまとめて提供し、アプリケーションの土台となるソフトウェアのこと
注3） Apache Maven Project https://maven.apache.org/
注4） Gradle Build Tool https://gradle.org/
注5） 事実上の標準のこと

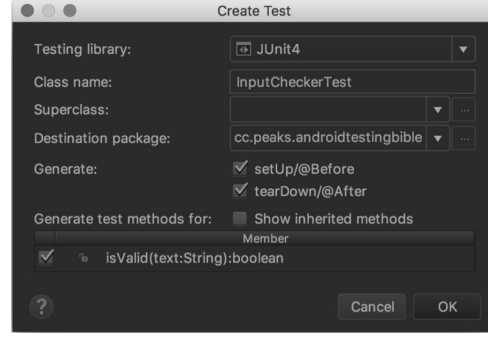
2.2.2　テストクラスとテストケース

JUnit 4では決められたルールにそって書かれたクラスをJUnitで実行可能な**テストクラス**として扱い、テストクラスに書かれた検証用のテストメソッドを順次実行して結果をレポートします。テストメソッドのことを**テストケース**と呼びます。

早速テストクラスを作ってみましょう。Android Studioのコードエディタ上でテスト対象クラス名の上にカーソルを合わせてCommand+Shift+T（WindowsではCtrl+Shift+T）と入力すると図2.1のようなダイアログが表示されます。画面上のCreate New Test...を選択してください。

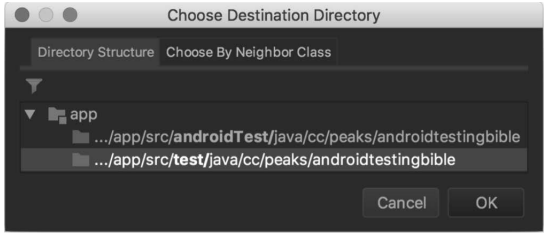
● 図 2.1　Create New Test

次に図2.2を参考にsetUp/@Before、tearDown/@Afterにチェックを入れます。ダイアログ下部にさきほど作成したisValid(text:String):Booleanメソッドがあるので、これにもチェックを入れてからOKを選択します。テストクラス名は任意に決めて構いませんが、慣習としてInputCheckerTestのように「テスト対象クラス + Test」とすることが多いです。



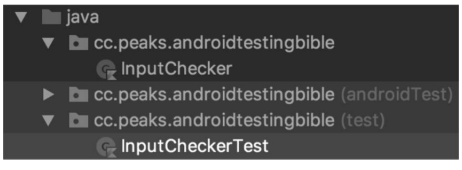
● 図 2.2　Create Test

つづいて図2.3のように.../app/src/test/javaを選択してください。Local Unit Testではこのようにtestディレクトリ配下にテストクラスを配置します。対してInstrumented Testの場合はandroidTest配下にテストクラスを配します。



● 図 2.3　Choose Destination Directory

ここまでで図2.4のようにInputCheckerと同一パッケージでtestディレクトリ配下にInputCheckerTestクラスが生成されていることを確認してください。



● 図 2.4　InputCheckerTest

InputCheckerTestクラスの中身がリスト2.3のようになっていれば準備完了です。

● リスト 2.3　生成された InputCheckerTest クラス

```
class InputCheckerTest {
  @Before
  fun setUp() {
  }

  @After
  fun tearDown() {
  }

  @Test
  fun isValid() {
  }
}
```

JUnit 4ではテストケースとして実行したいメソッドをorg.junit.Testアノテーションで修飾します。InputCheckerTestテストクラスでは@Test fun isValid() {}がテストケースとして実行されます。

@Beforeと@Afterアノテーションについては「2.2.8　テストケースごとの初期化・後処理」で後述します。

2.2.3　アサーション

テストケースでは、テスト対象メソッドを実行した結果が意図した状態であるかを確かめます。このとき、メソッドを実行した結果実際に得られた値を**実測値（actual）**、あらかじめ期待していた値を**期待値（expected）**、これらが意図した状態か確かめることを**アサーション（検証）**といいます。

JUnit 4ではorg.junit.Assert.assertThat()メソッドを使ってassertThat(実測値，マッチャー(期待値))という形式でアサーションを行います。

2.2.4　マッチャー

アサーションに利用するassertThat()メソッドは第2引数に「期待する状態がどのような条件か」を定義した**マッチャー**を指定します。JUnit 4には**Hamcrest**[注6]というマッチャーライブラリが組み込まれており、追加でインストールすることなく豊富なマッチャーを利用できます。

たとえばorg.hamcrest.CoreMatchers.is()は実測値と期待値が「同値である」という条件を定義したマッチャーです[注7]。ほかにも、実測値の整数が期待値「より大きい」という条件を定義したorg.hamcrest.Matchers.greaterThan()や、実測値のCollection[注8]が期待値を含んでいることを定義したorg.hamcrest.Matchers.hasItem()など、目的に応じて数多くのマッチャーが提供されています（リスト2.4）。

● リスト2.4　Hamcrestが提供するマッチャー

```
assertThat(1 + 1, `is`(2))
assertThat(100, greaterThan(50))
assertThat(listOf("for", "bar", "baz"), hasItem("bar"))
```

リスト2.4のassertThat(1 + 1, `is`(2))をみると、assert that 1 + 1 is 2（1足す1は2である）と自然言語のように読むことができます。このようにアサーションとマッチャーを使うと人間の読みやすい形で実測値と期待値が意図した状態であるかを検証できます。

2.2.5　はじめてのユニットテスト

前提知識がそろったのでInputCheckerTestクラスを使ってはじめてのユニットテストにチャレンジしてみましょう。

前項の内容を参考にimport文や`is`(true)のバッククォートなどに注意して入力してみてください（リスト2.5）。

注6）Hamcrest http://hamcrest.org/

注7）正確にはorg.hamcrest.CoreMatchers.is()自体はマッチャーではなく、org.hamcrest.core.Isマッチャー本体を返すファクトリメソッドです。なおバッククォートで囲む必要があるのはKotlinではisが予約語であるからです

注8）java.util.Collection

第 3 章

ユニットテスト応用編

白山 文彦／@fushiroyama

本章ではAndroidフレームワークのコードを利用したテストの書き方を確認し、実践的なユニットテストを書くための下地を作っていきます。筆者が開発現場でユニットテストを書いていて工夫している点や、新人教育を通じて得られた「つまずきやすい部分」の知見から培われた**ユニットテストのエッセンス**ともいうべき応用的な実践例を厳選してご紹介します。

3.1 Androidフレームワークのコードを使ったユニットテスト

第2章「ユニットテスト実践入門」で紹介した内容はすべてKotlin、Javaの標準クラス群のみを利用したモジュールのユニットテスト（Local Unit Test）に限られていました。Androidフレームワークを使ったコードも同様にテストできるのでしょうか。

確かめるために第2章でも取り上げた入力文字列チェックのためのモジュール`InputChecker`を、単純な`null`チェックから`android.text.TextUtils.isEmpty()`を使って空文字判定するように修正してみます（リスト3.1）。

● リスト 3.1　Android フレームワークのコードを使ったモジュール

```
import android.text.TextUtils

class InputChecker {
  fun isValid(text: String): Boolean {
    if (TextUtils.isEmpty(text)) throw IllegalArgumentException("Cannot be blank")
    return text.length >= 3 && text.matches(Regex("[a-zA-Z0-9]+"))
  }
}
```

次に修正した`isValid()`メソッドに空文字列を渡して、意図どおり`IllegalArgumentException`が発生することをテストケースで確認します（リスト3.2）。

● リスト 3.2　Android フレームワークのコードを使ったモジュールのユニットテスト

```
@Test
fun isValid_givenBlank_throwsIllegalArgumentException() {
  assertThatExceptionOfType(IllegalArgumentException::class.java)
      .isThrownBy { target.isValid("") }
      .withMessage("Cannot be blank")
}
```

テストを実行すると次のエラーが発生してテストは失敗します。

```
Method isEmpty in android.text.TextUtils not mocked.
See http://g.co/androidstudio/not-mocked for details."
```

これは「実行しようとしたAndroidフレームワークのコードが適切にモックされていないので失敗した」というメッセージです。Local Unit Testではモジュールに`android.*`の名前空間のクラス群が含まれる場合、Android SDKによって提供される`android.jar`でフレームワークの依存関係を解決しますが、これはあくまで名前解決の便宜上提供されているダミーに過ぎず、実機やエミュレーター以外で実行しようとすると実行時例外が上がってテストが失敗する仕様になっています。

そこで今回利用するのが「フレームワークのコードをモックする」アプローチです。モックといえば第2章「ユニットテスト実践入門」で解説したMockitoでAndroidフレームワークの依存関係をひとつひとつスタブ化してもよいのですが、本節ではそれよりもずっと便利な**Robolectric**を使った方法を解説します。

- https://github.com/robolectric/robolectric

RobolectricはAndroidフレームワークのコードをシミュレートしてJVM上で高速に実行できるテスティングフレームワークです。あくまでシミュレートであるため本物の挙動とは異なりますが、大部分のケースで申し分ないレベルでフレームワークのコードをシミュレートしてくれるので、Local Unit Testで大活躍します。

3.1.1　はじめての Robolectric

Robolectricを使うためにアプリケーションモジュールの`build.gradle`にリスト3.3の依存関係を設定します。

● リスト 3.3　app/build.gradle

```
dependencies {
  testImplementation "org.robolectric:robolectric:4.0-alpha-2"
}
```

次にGradleのandroid.testOptionsにunitTests.includeAndroidResources = trueを追加します(リスト3.4)。

● リスト 3.4　app/build.gradle

```
android {
  testOptions {
    unitTests.includeAndroidResources = true
  }
}
```

最後にテストクラスをRobolectricTestRunnerランナーで実行するように設定します（リスト3.5)。

● リスト 3.5　Android フレームワークのコードを使ったユニットテスト

```
import org.junit.runner.RunWith
import org.robolectric.RobolectricTestRunner
@RunWith(RobolectricTestRunner::class)
class InputCheckerTest {
  /* 省略 */
}
```

これで先ほどは失敗したリスト3.2のテストケースを実行し、無事成功することを確認してください。Robolectricはテスト実行時にフレームワークのコードをシミュレートするので、TextUtils.isEmpty()は意図どおり引数の文字列を評価します。

3.1.2　Robolectric と API レベル

Robolectricでは何も指定しない場合はAndroidマニフェストのtargetSdkVersionのAPIレベルとしてフレームワークのコードをシミュレートします。これを制御したい場合はorg.robolectric.annotation.Configアノテーションを利用します（リスト3.6)。

● リスト 3.6　@Config アノテーションの sdk オプション

```
import android.os.Build.VERSION_CODES.*
import org.robolectric.annotation.Config

@Config(sdk = [LOLLIPOP, LOLLIPOP_MR1])
@RunWith(RobolectricTestRunner::class)
class InputCheckerTest { }
```

sdkオプションには対象とするAPIレベルを配列で指定できます。この場合、指定したすべてのバージョンですべてのテストケースが実行されます。リスト3.7のようにminSdk，maxSdkで指定する方法もあります。

● リスト 3.7　@Config アノテーションの minSdk、maxSdk オプション

```
@Config(minSdk = 16, maxSdk = 27)
```

この場合も指定した範囲のAPIレベルのテストケースがすべて実行されるので注意してください。なお、APIレベルの差が結果を左右するほどAndroidフレームワークのコードに依存したテストコードはLocal Unit Testの範囲外だと考え、本章では特定のAPIレベルに依存した構成にしていません。

3.1.3　Robolectric と Android Jetpack

Google I/O 2018のFrictionless Android testing: write once, run everywhere[注1)]というセッションで、既存のAndroid Testing Support Libraryが**Android Test**という名称になり、Android開発ライブラリ群**Android Jetpack**[注2)]ファミリーの一員になることが発表されました。

この結果として大きく次の2点が便利になります。

1. **これまでバラバラであったLocal Unit TestとInstrumented Testの実行方法（ランナー）が共通化**
2. **フレームワークのコード（たとえばandroid.content.Context）へのアクセス方法が共通化**

これまではLocal Unit TestでAndroidフレームワークのContextオブジェクトが必要な場合、Robolectricを使ってRobolectricTestRunnerランナーでテストケースを実行し、Robolectric固有の方法でContextを取得する必要がありました。これがInstrumented Testと統一されるのは大きな進歩です。本節では今後を見据え、Android Jetpackの作法でContextを使ったテストケースを実行する例を紹介します。

まずは新しいランナーを利用するためにアプリケーションモジュールの依存関係にandroidx.test:runnerを追加します（リスト3.8）。

注 1）Frictionless Android testing: write once, run everywhere https://youtu.be/wYMIadv9iF8
注 2）Android Jetpack https://developer.android.com/jetpack/

● リスト 3.8　app/build.gradle

```
dependencies {
  testImplementation 'androidx.test:runner:1.1.0-alpha3'
  testImplementation "org.robolectric:robolectric:4.0-alpha-2"
}
```

テストクラスでは新しいandroidx.test.runner.AndroidJUnit4ランナーを使ってテストケースを実行します。また、Contextの取得はandroidx.test.InstrumentationRegistry.getTargetContext()メソッドを使って行います。いずれも、androidx.testという新しいパッケージである点に注意してください（リスト3.9）。

● リスト 3.9　Jetpack で Local Unit Test を実行

```
import androidx.test.InstrumentationRegistry
import androidx.test.runner.AndroidJUnit4

@RunWith(AndroidJUnit4::class)
class JetpackTest {
  @Test
  fun gettingContextTest() {
    val context = InstrumentationRegistry.getTargetContext()
    val appName = context.getString(R.string.app_name)
    assertThat(appName).isEqualTo("AndroidTestingBible")
  }
}
```

この方法の素晴らしい点は、@RunWith(AndroidJUnit4::class)指定しておくだけでInstrumented TestではAndroid環境の標準のランナーであるAndroidJUnit4ランナーが使われ、Local Unit Test環境では自動的にRobolectricTestRunnerランナーが使われてRobolectric上でユニットテストが実行されることです。

Contextを取得するために使ったInstrumentationRegistry.getTargetContext()もInstrumented Testとまったく同じ作法なので、今後はLocal Unit TestなのかInstrumented Testなのかを意識することなくテストを書けるケースが増えていくでしょう。

第 4 章

UIテスト（概要編）

外山 純生／@sumio_tym

皆さんはUIの操作が発生するテスト（UIテスト）を自動化したいと考えたことはありませんか？

UIテストの自動化では、ユニットテストと比較してテストを書くコストやメンテナンスするコストが大きくなりがちです。そのため、事前検討なしに自動化に着手するとコスト面で行き詰まることになりかねません。

本章では、そのような悲しい出来事が起きることを少しでも防ぐため、UIテストの自動化をするにあたって事前に検討したほうがよいことを解説します。

次にAndroidで広く使われているUIテストツール[注1)]である**Espresso**と**Appium**をさまざまな観点で比較し、テストツール選択の手助けをします。

最後に自動化したUIテストを長く運用し続けるためのポイントを紹介します。

4.1 UIテストの自動化をはじめる前に

UIテストの自動化では、何らかの方法でUIの操作をテストコードに書き下します。この逃れることのできない特徴から次のような課題と向き合わなければなりません。

- **画面デザインが変更される度に、テストコードの修正が必要**
- **画面の操作があるため個々のテストケースの実行時間が長くなりやすい**
- **UIに起因する理由により、テスト結果が安定しない**
  - **UIを操作してから画面が変化するまでの時間が一定ではない**
  - **操作を中断するダイアログが突然表示されることがある**

操作を中断するダイアログとはログイン画面やパーミッションダイアログといった表示タイミングが異なる種類のものです。

上述の課題はテストの作成、運用コストの増加に直結します。そのため、ある程度の準備をした上で自動化にとりかからないと、せっかく自動化したテストを運用できずに放棄せざるを得なくなることがあります。

注 1)　正確には UI テストの自動化を実現するツール（またはフレームワーク）ですが、本章では単にテストツールと表記しています

そのようなリスクを少しでも減らすために、次のような典型的なプロジェクトを想定（以降、想定プロジェクトと表記）して自動化に着手する前に最低限の整理が必要なことを解説します。

- **プロジェクトメンバーが一堂に会することができる程度の構成人数である**
- **すでにサービスはローンチ済みである**
- **リリース前のQA作業は次のような状態である**
  - **新機能の追加が多く、新機能の動作確認が作業の大半を占めている**
  - **開発スピードが求められ、既存機能に対する回帰テスト[注2)]が手薄い**
- **リリース時にサービス停止につながる既存機能の不具合がたびたび発覚する**
- **回帰テストを増やしたいがQA期間をこれ以上延ばしたくない**

テスト自動化研究会（STAR）による「テスト自動化の８原則」[注3)]にも、テスト自動化に取り組む前に留意しておくべき項目が簡潔にまとまっています。合わせて参考にしてください。

本章の解説はプロダクトオーナーを含む周囲のメンバーが、UIテストの自動化に前向きであることを前提にしています。そうではない場合や、導入対象の組織の規模が大きい場合は、より大規模な準備が必要です。難易度が高いケースについて詳細を知りたい場合は参考文献[6]の10章・11章が参考になります。

4.1.1 目的を整理する

まずはUIテストを自動化する目的が何なのか、関係者で議論するところから開始しましょう。

想定プロジェクトでは既存機能の不具合多発を防ぐため回帰テストを増やしたいという動機が自動化を考えはじめたときにありました。この場合は「QA期間は延ばさずに回帰テストを増やして既存機能の不具合発生を抑えたい」が大きな目的になります。

自動化というキーワードから**検証コストの削減**が目的と決め付けてしまいがちですが、このケースではそうではなさそうです。

また、関係者が口々に自動化したいと言っているとしても、その目的は人によって違うことがあります。プロダクトオーナーだけでなく、開発メンバー、QAチームも交えて何のために自動化したいのかを議論してみてください。

議論を行うと意外と次のようなコスト削減ではない目的が隠れていることがあります。

注2） テスト対象アプリを変更したときに、未変更箇所に意図せずバグが混入していないか検証するテストのこと

注3） https://sites.google.com/site/testautomationresearch/test_automation_principle

テスト担当者の精神的負担を軽減したい

画面上のメッセージの正しさを目視でひとつひとつ検証するなど、退屈な（精神的に疲れる）テストから解放されたい

リリースまでに必要な日数を短縮したい

自動化の結果、プロジェクト全体のスケジュールを短くし、開発コストを圧縮したい

既存機能のバグを予防したい

プロダクトコードを修正するときに思わぬ箇所にバグを入れてしまう不安から解放されたい

想定プロジェクトのケースでは３つ目の既存機能のバグを予防したいが大きな目的で、そこにQA期間を延ばしたくないという条件が付いていると考えるとよさそうです。

UIテストの自動化では、自動化にかかる投資が検証コストの減少だけでは回収できないことがあります。そのような状況になったとしても検証コスト削減以外の目的があれば失敗したとはいえません。

意識が合っていない関係者から「投資が回収できそうにないから自動化は中止だ」といわれたり効果を疑問視されたりしないように事前に合意しておくことが大切です。

4.1.2　自動化する範囲を決める

明確になった目的を達成するために、どのテストを自動化すべきなのか検討します。

最初は自動化できるものをすべて自動化しようと考えてしまいがちですが、かけられる予算は有限です。すべて自動化しようとは考えずに目的に合うものだけを選択することがポイントです。

まずは手動で行うか、自動で行うかとは関係なく、どのような切り口でテストをするべきかを整理し、そこから自動化する候補を選択するのがよいでしょう。

どのような手段であれ目的を達成できる候補を選択することが一番大切です。想定プロジェクトの場合だと、手薄だった回帰テストのうち過去に不具合が発覚した箇所を重点的に自動化対象にするとよさそうです。

一般的には次のような選択基準が考えられます。

検証コスト削減が目的の場合

何度も行うテストや時間（テスト工数）がかかるテストを重点的に自動化する

QAエンジニアの精神的負担の軽減が目的の場合

QAエンジニアが負担に思っているテストを重点的に自動化する

これだけを条件に選択してしまうと自動化が困難・不可能なものが入り込んだり、自動化コストが予算を超えたりする可能性があります。次の点を考慮しながら自動化の優先度を付けてください。

- 採用を考えているテストツールで操作（実装）しやすいものを優先的に選択する
- 操作する画面（ActivityやFragmentなど）の数が少なくなるようにする

後者をいいかえると、他の自動テストでは操作しない画面を含むテストは自動化の優先度を下げるということです。理由は自動テストの作成コストに影響するからですが、画面の数とコストの関係はわかりづらいかもしれませんので、ここで詳しく説明します。

画面の数とコストの関係

UIテストでは、UIコンポーネントを操作するために検索するための条件を指定します。たとえばLinearLayoutの直下の階層にあるEditTextで、hintに「名前」と表示されているものといった検索条件です。

この条件を見つけて意図どおり操作できるか動作確認する作業がもっとも時間がかかります。すなわち次に挙げる２つのテストを比較すると、後者のほうがテスト作成コストが大きくなるのです。

- 他の自動テストで、すでに操作実績のあるUIコンポーネントを操作するテスト
- 他の自動テストで操作実績がないUIコンポーネントを操作するテスト

当然ながら、いままで操作したことのない画面には、操作実績がないUIコンポーネントしかありません。そのため、テスト対象の画面を増やすときには大きなテスト作成コストがかかることになります。

具体例として次の３つのテストを考えてみます。

1. 画面A、画面Bを操作するテスト
2. 画面Bを操作するテスト
3. 画面A、画面B、画面Cを操作するテスト

これら３つをすべて自動化する場合、操作の自動化が必要な画面はA、B、Cの３つです。

ここでテスト１やテスト２の自動化を諦めても、操作が必要な画面数は３つのまま変化しません。一方でテスト３の自動化を諦めると画面Cの操作は自動化する必要がなくなり、画面数を２つに減らせます。

このように自動化コストを抑えるために自動化対象を減らすのであれば、他で操作しない画面を操作するテストを減らすのが効果的なのです。

なお、検討の時点ではテストツールも予算も決まっていないかもしれません。その場合は当初の自動化範囲を元にテストツールを決めてください。削減はテストツールとの相性をみながらで問題ありません。テストツールが決まれば自動化にかかる見積もり、予算の調整が可能です。ここまでくれば改めて自動化する範囲を絞り込めます。

4.1.3　テストツールを選択する

一般的には、次のような観点でテストツールを選択していきます。

- 自動化したい内容がツールを使って簡単に実現できそうか？
  - トラブルなく、安定してテストを実行できそうか？
- 低コストで学習できるか？
  - （自動化担当者にとって）テストコードが書きやすいか？
  - 使い方に困ったときに近くに質問できる人がいるか？
- 開発コミュニティは活発か？
  - バグ修正やAndroid新バージョンへの対応など、今後も開発が続きそうか？
  - 使い方やトラブル時の対応方法などについて情報が充実しているか？

本書で扱うテストツールであるEspressoやAppiumの場合、開発コミュニティはどちらも活発ですので、それ以外の観点について検討すればよいでしょう。

それぞれのテストツールを比較した「4.2　テストツール選択のポイント」も、あわせて参照してください。有料のテストツールを検討する場合は、保守料金やサポートの充実度などの考慮も必要ですが詳細は提供元に依存するため割愛します。

4.1.4　スケジュール・予算・体制を決める

UIテスト自動化のスケジュール、予算、体制を決めていきましょう。通常のソフトウェア開発と同じように決めればよいのですが特に注意が必要な点を説明します。

スケジュール

UIテストの自動化ではテスト実装時よりも、運用を開始してからのほうが大きな課題に直面しやすいという特徴があります。まずは一番簡単にできそうなところを数ケース作り、運用を開始するスケジュールを立ててください。

たとえば想定プロジェクトのリリース間隔が1ヶ月で、既存機能の回帰テストを自動化する場合を考えてみます。図4.1のようにすべての自動化が完了してから運用を開始しようとすると、リリースまでの運用期間は1週間です。そうではなく、簡単な自動化が1つ完了したらすぐに運用を開始するようにすれば、図4.2のようにリリースまでの運用期間は2．5週間に延びます。

第5章

UIテスト（Espresso編）

外山 純生／@sumio_tym

第4章「UIテスト（概要編）」では、UIテストの自動化着手前に必要な準備について紹介しました。本章では、いよいよEspressoを使ったテストの書き方について具体的に説明します。

前半となる「5.1　概要」から「5.5　画面更新を待ち合わせる」までEspressoの基本的な使い方を説明します。Espressoにはじめて触れる場合は、ここから読み進めてください。

続いて「5.6　PageObjectデザインパターンの実践：長く使われ続けるテストにするために」ではメンテナンス性の高いテストを書くためのテクニックを説明します。実際のプロダクトに適用する場合には、メンテナンス性の高さはとても大事ですのでポイントとなる内容です。

最後の「5.7　目的別テクニック」は、Espressoでテストを書く上で直面しやすい問題をピックアップし、それぞれの解決方法を紹介しています。必要に応じて参照してください。

本章の解説では、プログラミング言語にJavaを使っています。次のようにEspresso Test RecorderのKotlin対応が十分でないことが理由です。

- Android Studio 3.1.4では、テストコードをJava言語でしか生成できない
- 生成されたJavaコードをKotlinに変換するとコンパイルエラーとなる部分がある

2つ目の問題はHamcrestのMatcherを使っている部分で、コンパイルエラーが発生しています。ただしGradleからビルド、実行することは可能です。

Android Studio 3.2（執筆時点の最新版は3.2 Beta 5）ではテストコード生成時にKotlinを選択できるようになっていますが、生成コードがコンパイルエラーになる点は変わりませんでした。

このような状況なので現状ではEspressoのテストコードはJavaで書くほうが無難といえます。テストの価値は言語に依存しません。今後、Kotlin対応が進んだ時点でKotlinに変換する方針が良いでしょう。

5.1 概要

EspressoはAOSPが開発し、Android Testの一部として提供されているUIテストフレームワークです。Android端末・エミュレーターの中でテストコードを実行するInstrumented Testの仕組みを使って動作します。

Espressoの公式ドキュメント[注1)]によると、Espressoは簡潔で、美しく、信頼性の高いUIテストを書けることを目指して開発されました。そのためEspressoを使ったテストコードは、簡潔さを追求した独特なスタイルになっています。

Espressoの動作条件は次のとおりです。

- UI Automatorと併用する場合：API Level 19（Android 4.4）以上
- UI Automatorと併用しない場合：API Level 16（Android 4.1）以上

パーミッション確認ダイアログを扱ったり、画面更新完了を明示的に待ち合わせたりなど、さまざまな場面でUI Automatorは必要になりますので本書ではAndroid 4.4以上の端末でテストを実行できる前提で解説します。

やむをえずAndroid 4.4未満の端末でテストを実行しなければならない場合は、以降の解説のうち、UI AutomatorのAPIが使われている箇所は動作しない点に注意して読み進めてください。

5.2 セットアップ

ここでは、Espressoを使ったテストを書き始める前に必要なセットアップ作業について説明します。

一般的なテストツールやライブラリと異なり、ビルドスクリプトの修正に加えてテスト対象端末の設定変更が必要です。順を追って説明していきます。

5.2.1 ビルドスクリプトの修正

Android Studioで新規にプロジェクトを作成すると、最初からEspressoが使えるようになっています。初期設定は必要最低限ですのでEspressoの機能をフル活用するには、ビルドスクリプトへの追記が必要です。

ここではEspressoを使うのであれば、かならず設定しておきたいものと、それ以外のものに分けて説明します。

注1） https://d.android.com/training/testing/espresso/

かならず設定しておきたいもの

モジュールレベルのビルドスクリプト（通常はapp/build.gradle）をリスト5.1のように修正してください。

この中には、Android Studioが最初から設定しているものもありますので、重複して追記しないように注意してください。

● リスト 5.1　Espresso を使うのに必要なビルドスクリプト修正箇所

```
apply plugin: 'com.android.application'

android {
  defaultConfig {
    ...
    // ↓この行を追加する
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
  }
  ...
}

dependencies {
  ...
  // ↓この 4 つの依存関係を追加する
  androidTestImplementation 'com.android.support.test:runner:1.0.2'
  androidTestImplementation 'com.android.support.test:rules:1.0.2'
  androidTestImplementation \
      'com.android.support.test.espresso:espresso-core:3.0.2'
  androidTestImplementation \
      'com.android.support.test.espresso:espresso-contrib:3.0.2', {
    exclude group: 'com.android.support'
  }
}

// ↓このブロックを追加する。
// ※ ${supportLibraryVersion} は、プロダクションコードで使っている
//   サポートライブラリのバージョンに置き換えること。
configurations.all {
    resolutionStrategy.force \
        "com.android.support:support-annotations:${supportLibraryVersion}"
}
```

Instrumented Testの制約で、プロダクションコード、テストコードの両方から依存しているライブラリはバージョンを揃えなければなりません。espresso-contribに対するexclude指定やconfigurationsのresolutionStrategy指定はそのためのものです。

また3.0.2などのバージョン指定部分は、原則として最新バージョンを指定してください。最新バージョンの一覧は公式ページ「Android Test release notes[注2)]」で確認できます。

5.2.2　必要に応じて設定するもの

Espressoでは、特定の用途でのみ利用する機能は、それぞれ独立したモジュールとして提供されています。Espressoが提供しているモジュールの一覧を表5.1にまとめました。

これらのモジュールはMavenのグループID「com.android.support.test.espresso」で提供されています。

● 表 5.1　グループ ID「com.android.support.test.espresso」提供モジュール一覧

アーティファクト ID	モジュールの概要
espresso-core	Espresso を使う場合には必須のコアモジュール
espresso-contrib	RecyclerView などの複雑なコンポーネントをテストする
espresso-idling-resource	Espresso の自動同期機能をカスタマイズする（「5.5　画面更新を待ち合わせる」参照）
espresso-web	WebView をテストする（「5.7.2　WebView を操作する」参照）
espresso-intents	外部アプリとの Intent 送受信をテストする（「5.7.7　外部アプリとの Intent 送受信をテストする」参照）
espresso-accessibility	アクセシビリティの整備状況をチェックする
espresso-remote	複数プロセスで構成されたアプリをテストする

一覧のうちespresso-coreとespresso-contribはリスト5.1ですでに追加済みのものです。

それ以外のモジュールのうち、本書で触れているものには参照先を併記しました。本書で触れていないespresso-accessibility・espresso-remoteについては、それぞれ公式ドキュメントの「Accessibility checking[注3)]」「Multiprocess Espresso[注4)]」を参照してください。

これらのモジュールを使うにはapp/build.gradleのdependenciesブロックに依存関係を追加します。たとえばespresso-webを使う場合はリスト5.2のようにします。

● リスト 5.2　espresso-web モジュールに対する依存関係を追加する例

```
dependencies {
  ...
  androidTestImplementation 'com.android.support.test.espresso:espresso-web:3.0.2'
}
```

注 2）https://d.android.com/training/testing/release-notes
注 3）https://d.android.com/training/testing/espresso/accessibility-checking
注 4）https://d.android.com/training/testing/espresso/multiprocess

この1行を追加するとWebViewのテストモジュールが組み込まれます。

5.2.3 端末のアニメーション設定

UIテストを実行するAndroid端末のアニメーションを無効化します。無効化しなくても問題が起きないケースもありますが、テストが不安定になるのを避ける効果[注5)]があるため設定しておくと安心です。

Android端末を次のように操作してください。

- 設定から開発者オプションを開く
- 次の3つの設定項目を「オフ」に変更する
  - ウィンドウアニメスケール
  - トランジションアニメスケール
  - アニメーター再生時間スケール

コマンドラインからしかUIテストを実行しないのであれば、自動的にアニメーションを無効にすることもできます。リスト5.3のように、app/build.gradleにtestOptions.animationsDisabledオプションを指定してください。

● リスト 5.3 アニメーションを無効化するビルドスクリプト設定

```
android {
  ...
  testOptions {
    animationsDisabled true
  }
}
```

このオプションはInstrumented Testを実行している間だけアニメーションを無効にします。とても便利なオプションなのですが次のような制限があるため注意して使ってください。

- API Level 26（Android 8.0）未満では一部のアニメーションしか無効にできない[注6)]
- Android Studioからテストを実行するときには効果がない[注7)]

注 5） https://d.android.com/training/testing/ui-testing/espresso-testing#setup
注 6） https://issuetracker.google.com/issues/69247502
注 7） 筆者は Android Studio のバージョン 3.1.4 と 3.2 beta 5 で確認してみましたが、効果はありませんでした

第 6 章

UIテスト（Appium編）

平田 敏之／@tarappo

本章ではAppiumを用いたUIテストを説明していきます。AppiumはAndroid/iOSアプリケーションなどを自動化するためのオープンソースのツールです。

前半はAppiumを知って自身の環境でAppiumを利用したコードを実装できるようになりましょう。後半は実際にテスティングフレームワークを利用してAppiumを用いたUIテストを実装・導入する手法を学びます。

さらにAppiumを用いたUIテストを導入した事例を紹介しています。実際の事例を知ることで自身のプロジェクトに導入する際の参考になればと考えています。

AppiumはJavaScript、Java、Rubyなど複数のプログラミング言語を利用できます。本章ではRubyを用いています。採用した理由は利用ライブラリの更新頻度が高く、筆者が所属するSWETグループでよく使っているからです。

動作確認バージョンは次のとおりです。

- macOS HighSierra 10.13.4
- Ruby 2.4.1
- Appium 1.9.0
- Appium Desktop 1.7.0

また利用するAndroid端末の言語設定は日本語にしています。

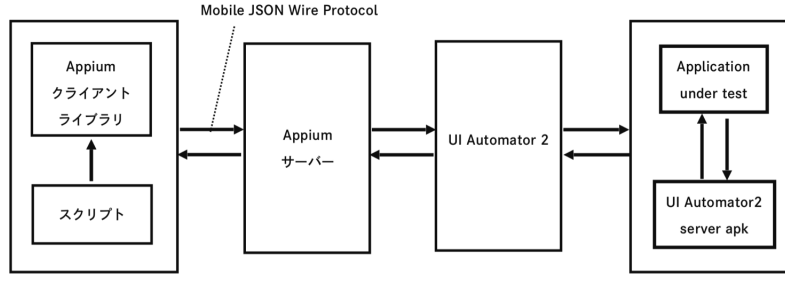
6.1 Appiumの概要

Appiumの利用方法を説明する前に次の項目を観点に概要を説明します。

- アーキテクチャ
- Desired Capabilities
- 利用する言語・テスティングフレームワークの選定
- Appiumを知るための参考資料

6.1.1　アーキテクチャ

Appiumのアーキテクチャを図6.1に図示します。



● 図 6.1　Appium アーキテクチャ

Appiumは基本的にクライアント・サーバー型の構成をとっています。

クライアントはスクリプトに動作を記述します。スクリプトにしたがってAppiumクライアントライブラリのAPIを利用した結果をAppiumサーバーに送ります。Appiumサーバーではドライバー経由でテスト対象となるアプリケーションを操作して結果をクライアントに返します。

スクリプト、Appiumサーバーは対象となるアプリケーションがインストールされている端末上では実行されていません。両方ともPCやサーバー上で実行します。さらにスクリプトとAppiumサーバーを同じ場所で動かす必要はなく、スクリプトとAppiumサーバーがHTTP通信ができれば問題ありません。

図のAppiumクライアントライブラリとAppiumサーバーについて解説していきます。

Appium クライアントライブラリ

Webブラウザの自動化のためにJSON Wire Protocol[注1)]が定められています。AppiumではJSON Wire Protocolを拡張したMobile JSON Wire Protocol[注2)]を実装してモバイルアプリケーションに対応しています。クライアント・サーバー間は本プロトコルで通信を行います。

プロトコルさえサポートしていれば、いろいろな言語でクライントライブラリを実装できます。クライアントライブラリは、アプリケーションや端末を操作するためのAPIを提供しています。

Appium サーバー

Appiumサーバーはクライントから受け取ったリクエストをもとに、プラットフォーム標準のテスティングフレームワークを利用してテスト対象のアプリケーションを操作します。そして操作結果をクライアント側に返却します。

注 1)　https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol
注 2)　https://github.com/SeleniumHQ/mobile-spec/blob/master/spec-draft.md

図中ではUI Automator 2とプラットフォーム標準のテスティングフレームワークを記載しています。Appiumが提供するドライバーがフレームワークの利用をサポートしています。本章では現時点で安定版であるUiAutomator2 Driverを利用します。テスティングフレームワークを変更するとドライバーも変わります。Espressoを利用するEspresso Driver[注3)]は現在開発中です。

6.1.2 Desired Capabilities

Desired Capabilitiesは、クライアントからサーバーに送信するJSONオブジェクトです。Appiumの利用ではDesired Capabilitiesの値が重要になります。

AppiumはJSONオブジェクトのkeyとvalueを設定してテストを実行する端末やUiAutomator2かEspressoといったドライバーを指定できます。Desired Capabilitiesの値を適切に設定していないと、求めている環境でテストが実行されません。

基本的な設定は次のとおりです（リスト6.1）。

● リスト 6.1　Desired Capabilities の設定例

```
{
    "platformName": "Android",
    "platformVersion": "8.0",
    "disableWindowAnimation": true,
    # Android では利用されていませんが、値としては必要になっています。
    "deviceName": "Android Emulator",
    "automationName": "uiautomator2",
    "unicodeKeyboard": true,
    "app": "/path/to/my.apk"
}
```

サンプルコード以外にも設定できるkeyはいろいろとあります。表6.1にDesired Capabilitiesで設定可能なkeyのうち主要なものを紹介します。

注 3）https://github.com/appium/appium-espresso-driver

● 表 6.1　Desired Capabilities の例

key	説明
platformName	モバイル OS プラットフォーム（iOS、Android など）
automationName	利用する自動化エンジン（UiAutomator2 や Espresso など）
unicodeKeyboard	Unicode 入力ができるキーボードを利用するかどうか
resetKeyboard	unicodeKeyboard を true にした時にキーボードを元の状態に戻すかどうか
app	apk ファイルの絶対パス
disableWindowAnimation	端末のウィンドウのアニメーションを無効にする
androidInstallTimeout	apk ファイルのインストールのタイムアウト値（未設定だと 90,000 ミリ秒）
newCommandTimeout	新しいコマンドが来るまでどれぐらい待つかどうか（秒設定）
fullReset	テスト後にアプリケーションのデータを消し、アプリケーションをアンインストールする
noReset	テスト後にアプリケーションのデータを消さず、アプリケーションもアンインストールしない
udid	接続されている端末のシリアル番号

newCommandTimeoutはテストコードにブレークポイントをはってなにかをするときは、長めに設定することでタイムアウトまでの余裕を確保できます。このように開発フェーズで使い分けるkeyも存在しています。

今回紹介した以外にも設定できるkeyはたくさんあるので、どのようなものがあるかはドキュメントを参考にしてください。

- Appium ドキュメント https://appium.io/docs/en/writing-running-appium/caps/

6.1.3　利用する言語・テスティングフレームワークの選定

Appiumは、仕様に基づいて実装すればどのようなプログラミング言語でも利用できます。公式ドキュメント[注4)]にはいろいろなプログラミング言語で実装されたクライアントライブラリの情報が載っています。

利用するプログラミング言語の選定基準は次のようなものがあります。

- テストコードを実装するメンバーのスキル
- テスト対象アプリケーション以外との連携の必要性
- クライアントライブラリの更新頻度や参考情報の多さ

注 4）https://appium.io/docs/en/about-appium/appium-clients/

Appiumではアプリケーションの実行ファイルがあればUIテストを行えます。このような特徴のおかげでアプリケーション開発者以外がテストコードを実装できます。アプリケーションとテストの実装者が異なる場合、テストコードを作るメンバーのスキルに合わせてテスティングフレームワークを採用したほうがよい結果を得られるでしょう。

テストコードはテスト対象となるアプリケーションを操作する以外にも、サーバーと連携したいことがあります。たとえば結果をアップロードするなど組織が求めるテスト利用方法はさまざまです。

組織に合わせてテストを構築する場合、すでに存在するライブラリを採用したいというモチベーションが生まれてきます。このときはライブラリにあわせてAppiumで使うプログラミング言語を決定することもありえます。

すべてのクライアントライブラリがメンテナンスされ続けているわけではありません。利用するプログラミング言語のクライアントライブラリの更新頻度や、参考情報の多少も重要な選定基準です。更新頻度はライブラリのGitHub Repoを確認しましょう。

6.1.4 Appiumを知るための参考資料

Appiumに関する情報は公式サイトにまとまっています。わからないことがあれば、まず公式サイトをチェックしてみてください。サンプルコードはプログラミング言語ごとに用意されているので、こちらもあわせて確認しておくことをお勧めします。Appiumをテーマにしたカンファレンスもあり、実際のノウハウなど役に立つトークが多くあります。

- Appium公式サイト https://appium.io/documentation.html
- サンプルコード https://github.com/appium/appium/tree/master/sample-code
- Appium conference https://appiumconf.com/

また、Appiumの情報を扱ったWebサイトはAppium Pro[注5)]などいろいろあります。

注5） https://appiumpro.com/

第7章

JUnit 5

菊池 紘／@kikuchy

当たり前のことではあるものの自動テストを行う上で避けて通れないことのひとつが、テストコードの記述です。

テストの関心ごとに集中してテストコードを記述できるツールとしてテスティングフレームワークがあります。テスティングフレームワークはテストケースの収集やレポートを行い、開発者を手助けします。テスティングフレームワークのひとつであるJUnitは登場以来、改善が続けられており、2017年10月10日に最新のメジャーバージョンであるJUnit 5がリリースされました。

現在のところAndroidアプリケーションのテストではひとつ前のバージョンであるJUnit 4が使われており、公式サポートされていないことから広くJUnit 5が使われるに至っていません。しかしJUnit 5にはJUnit 4の不便さを克服する便利な機能が搭載されており、特に個別のユースケースに合わせた拡張が簡単になっています。テストコードと長く付き合うためにも、新しいテスティングフレームワークでテストを書いてみたいものです。

本章ではJUnit 4とJUnit 5の機能を比較し、JUnit 5で新しく導入された機能と実現できることがらを解説します。またAndroidアプリケーションのテストにJUnit 5を使用する方法と注意点を合わせて紹介します。多少なりともJUnit 4でテストコード書いた経験があり、さらに楽にテストを書きたい開発者へ向けた内容です。

7.1 AndroidとJUnit 5

JUnitは著名なテスティングフレームワークです。Javaを使った開発ではデファクトスタンダードともいえる存在で、世界中にたくさんのユーザーが存在します。いわゆるxUnit[注1)]のJava版にあたります。

名称に**Unit**とついていますがユニットテスト（Unit Testing）のためだけのものではありません。

テストを記述したテストメソッドの探索と列挙、テストメソッドの実行前後での前準備や後始末の実行、実行結果の収集とレポート作成といった、テストにおける一連の流れをすべて行うフレームワークがJUnitです。

Androidアプリケーション開発においてもJUnitは重要な位置を占めます。Androidエミュレー

注1） こうしたユニットテスト用フレームワークは Kent Beck の SmallTalk 用フレームワーク SUnit が原型となっていて他の言語へも移植されていきました。総称して xUnit と呼びます

ターや実機でUIを操作し、結果を確かめるようなUIテストも実行可能です。

2018年9月の執筆時点では、Android Studioを使って新規プロジェクトを作成すると自動的にJUnit 4が依存ライブラリとして追加されます。Androidアプリケーション開発のためのライブラリであるAndroid Test（旧Testing Support Library）には、JUnit用に記述したテストをAndroidエミュレーターや実機上で実行するためのAndroid JUnit RunnerとJUnit 4でのテストを手助けするJUnit 4 Rulesが含まれています。これらからもJUnitはスタンダードなテスティングフレームワークとしてAndroidアプリケーション開発者に認知されており、重要であることがうかがえます。

2018年9月時点、Android TestはJUnit 5を正式にサポートしていません。Androidアプリケーション開発にJUnit 5を使用する場合は自力で環境を構築する必要があります。

将来的にJavaユーザーのJUnit 5への移行が進めば、Android TestがJUnit 5をサポートすることも考えられます。

7.1.1 android-junit5 プラグイン

執筆時点ではAndroidプロジェクトでJUnit 5を使ったテストを行う場合、サードパーティー製のGradleプラグインを使用する必要があります。

Gradle 4.6からJUnit 5が公式にサポートされていますが、純粋なJavaプロジェクトのためのものであってAndroidプロジェクトのようにBuild Valiantを複数所有できるようなプロジェクトには対応していません。

AndroidプロジェクトでもJUnit 5の使用を可能にするプラグインとしては**android-junit5**がデファクトスタンダードとなっていますので、このプラグインの使用方法を説明します。

- https://github.com/mannodermaus/android-junit5

以降はバージョン1.2.0.0をベースに解説を行います。

7.1.2 導入要件

android-junit5を導入するにあたって最低限、満たさなければならない動作要件を表7.1に示します。

● 表 7.1　導入する android-junit5 のバージョンと最低動作環境

android-junit5	Gradle	Android Gradle Plugin
1.2.0.0	4.7 以上	3.2.0 以上

まだAndroid Gradle Pluginのバージョンを上げられていないプロジェクトは、早めにバージョンを上げておきましょう。このプラグインだけでなくその他のプラグインやライブラリを導入できず、バージョンアップが困難になることがあります。

7.1.3 Local Unit Test と Instrumented Test に導入する

プロジェクトルートの build.gradle 内、buildscript ブロックに、次のように android-junit5 への依存関係を追加します（リスト 7.1）。

● リスト 7.1　android-junit5 を buildscript の依存関係に追加

```
buildscript {
  dependencies {
    classpath "de.mannodermaus.gradle.plugins:android-junit5:1.2.0.0"
  }
}
```

これでプロジェクトの Gradle スクリプトの依存関係に android-junit5 を追加し、Gradle プラグイン de.mannodermaus.android-junit5 を使用できるようになります。

もし、まだ正式にリリースされていないスナップショット版を使用したい場合は、Sonatype のスナップショットリポジトリを参照する必要があります（リスト 7.2）。2018 年 9 月 17 日執筆時点での最新のスナップショットは 1.2.0.1-SNAPSHOT でした。

● リスト 7.2　スナップショット版を使用する

```
buildscript {
  repositories {
    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
  }
  dependencies {
    classpath "de.mannodermaus.gradle.plugins:android-junit5:1.2.0.1-SNAPSHOT"
  }
}
```

これでスナップショット版の Gradle プラグイン de.mannodermaus.android-junit5 を使用できるようになります。

Sonatype はパッケージリポジトリ管理ツールを提供している企業です。その製品を使って Maven の Central リポジトリを公開しています。Central リポジトリにはリリース品質のパッケージをホストするためのものとスナップショット品質のパッケージをホストするものがあり、リスト 7.2 で参照を追加しているのは後者の方です。ちなみに Maven Central というと通常は前者の方を指します。

次に JUnit 5 を使用したいアプリケーションモジュール、もしくはライブラリモジュールの build.gradle に android-junit5 プラグインを適用し、設定と依存関係を記述します。

● リスト7.3　Local Unit Test で android-junit5 を使用可能にする

```
apply plugin: "de.mannodermaus.android-junit5"

dependencies {
  testImplementation "org.junit.jupiter:junit-jupiter-api:5.2.0"
  testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:5.2.0"

  testImplementation "org.junit.jupiter:junit-jupiter-params:5.2.0"

  testImplementation "junit:junit:4.12"
  testRuntimeOnly "org.junit.vintage:junit-vintage-engine:5.2.0"
}
```

リスト7.3はLocal Unit Testにおいて android-junit5を使用できるようにする設定です。dependenciesのはじめの２行はJUnit ５を使用する上で必須の依存関係です。これを追加することでJUnit ５の基本機能をUnit Testで使用できるようになります。

junit-jupiter-paramsはパラメタライズドテストを行いたい場合にのみ必要です。使用しない場合は書かなくても構いません。パラメタライズドテスト系のAPIはJUnit ５の標準機能と切り離されているため、使用したい場合は依存関係を別途記載します。

dependenciesの最後の２行はJUnit ４のAPIで書かれたテストを同時に存在させたい場合に必要です。JUnit ４のAPIで書かれたテストを使用しない場合は不要です。

次に示すリスト7.4はInstrumented TestでJUnit ５を使用可能にする設定です。

● リスト7.4　Instrumented Test で android-junit5 を使用可能にする

```
apply plugin: "de.mannodermaus.android-junit5"

android {
  defaultConfig {
    // (1)
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    testInstrumentationRunnerArgument "runnerBuilder",
      "de.mannodermaus.junit5.AndroidJUnit5Builder"

    // (2)
    minSdkVersion 26
  }

  // (2')
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
```

```
  // (3)
  packagingOptions {
    exclude 'META-INF/LICENSE.md'
    exclude 'META-INF/LICENSE-notice.md'
  }
}

dependencies {
  // (4)
  androidTestImplementation "org.junit.jupiter:junit-jupiter-api:5.2.0"
  androidTestImplementation
    "de.mannodermaus.junit5:android-instrumentation-test:0.2.2"
  androidTestRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:5.2.0"
  androidTestRuntimeOnly "org.junit.platform:junit-platform-runner:1.2.0"
  androidTestRuntimeOnly
    "de.mannodermaus.junit5:android-instrumentation-test-runner:0.2.2"

  androidTestImplementation "org.junit.jupiter:junit-jupiter-params:5.2.0"

  androidTestImplementation "junit:junit:4.12"
  androidTestRuntimeOnly "org.junit.vintage:junit-vintage-engine:5.2.0"
}
```

（1）ではテスト時にアプリケーションを外部から操作するためのInstrumentation RunnerとしてTesting Support Libraryの`AndroidJUnitRunner`を使い、`runnerBuilder`オプションに`android-junit5`の`AndroidJUnit5Builder`を指定しています。2018年9月時点で`android-junit5`のREADMEにはTesting Support Library、すなわち`android.support.test`パッケージを使った設定方法が記載されていますが、著者が試したところ、`android-junit5`のバージョン1.2.0.0を使用したInstrumented Testは`androidx.test`パッケージのAndroid Testライブラリ（`androidx.test:runner:1.1.0-alpha4`）でも動作するようです。しかし`android-junit5`は内部で`android.support.test`パッケージのTesting Support Libraryを使用しているようですので、なにかしらの不具合が生じる可能性はあります。`androidx.test`のAndroid Testライブラリと同時に使用して問題が起きた場合は`android.support.test`パッケージのTesting Support Libraryを使用してみてください。

（2）と（2'）はソースコードをJava 8でコンパイルし、Java 8のAPIを使用可能にするための設定です。これはJUnit 5がJava 8の機能を使用しているためです。Androidのランタイム上でテストを実行するInstrumented TestにおいてはJava 8のAPIに対応したSDK Level 26以上のSDKと、Java 8の言語機能をAndroidで使用可能にするdesugarを動作させるための設定が必要になります。アプリケーションの要件によっては`minSdkVersion`を26以上にすることは難しいでしょう。その際は、Instrumented Test用のプロダクトフレーバーを用意して、そのフレーバーの`minSdkVersion`を26以上にするとよいでしょう。

（3）はapkを作成する際に余計なファイルを除外する設定です。

（4）はInstrumented TestでJUnit 5を使用する上で必須の依存関係です。

第 8 章
CI/CD

堀江 亮介／@Horie1024

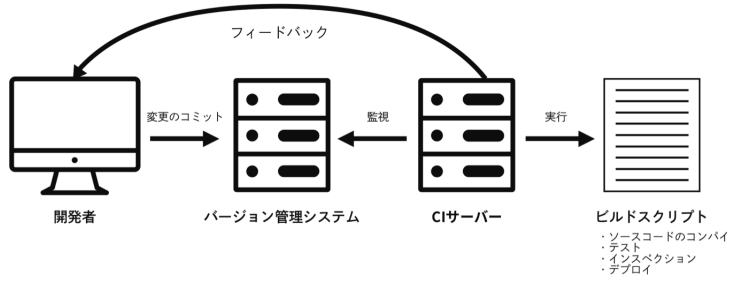
本章では自動化されたテストを開発フローに組み込んでより効果的に実行するプラクティスを扱います。**継続的インテグレーション（Continuous Integration:CI）**そして**継続的デリバリー（Continuous Delivery:CD）**をAndroidプロジェクトに導入し、開発からリリースまでのプロセスを統合的に管理します。開発速度と品質を両立する上でチームの助けとなるでしょう。

8.1 継続的インテグレーション

継続的インテグレーションの**インテグレーション**とは、個々のソースコードが全体として機能するかを検証するために統合することを指します。そして統合されたソースコードの検証は**ビルド**によって行います。

ビルドという言葉はコンパイルと同じ意味で使われる場合もありますが、CIの文脈でのビルドはソースコードのコンパイルだけでなく、インスペクション（静的解析）、テスティング、成果物のデプロイを含むことが往々にしてあります。

CIを自動的に実行する仕組みを**CIシステム**と呼びます。図8.1はCIシステムの全体像です。



● 図 8.1　CI システム

CIシステムはバージョン管理システムと連携してCIサーバーでビルドスクリプトを実行し、結果を開発者にフィードバックするといった機能を備えています。

8.1.1　導入のメリット

前述の内容をまとめるとCIは統合されたソースコードが機能するかをビルドによって継続的に検証するソフトウェア開発手法と言い換えてもいいでしょう。導入のメリットは次のとおりです。

- 不具合の早期発見
- プロジェクトの見える化
- 開発への安心と自信

以降、それぞれについて説明していきます。

不具合の早期発見

継続的な検証は不具合の早期発見につながり、プロジェクトのリスクを減らします。早期に発見することでソースコードの差分や影響範囲が小さい段階で修正に取りかかれるため、問題に対処しやすくなります。

CIシステムでは不具合を発見したことを即座に開発チームにフィードバックすることで問題の発見と修正がより効率的に行えます。その上でも自動化されたテストは重要な存在です。テストコードを書くことは高品質なソフトウェアを開発する近道であり、開発を支える資産であるとの主張は、本書を読み進めてきた読者には当然のことと受け入れられるでしょう。

プロジェクトの見える化

CIによる継続的な検証はプロジェクトの異変をとらえます。ビルドステータスや検証結果を確認することで、プロジェクトのさまざまな傾向と効果的な判断材料を提供しています。

たとえばインスペクションによってコーディング規約に反したコードやパフォーマンスに悪影響を与えるコードが増えていることに気づければソフトウェアの品質に与える影響を抑えることができます。

開発への安心と自信

CIの導入は常にソフトウェアをデプロイ可能な状態に保ちます。ソースコードの変化ごとにテストを実行し、インスペクションによって品質をチェックします。開発チームが自信を持ってプロジェクトを進められるようになり、開発終盤にでる不具合を予防し安心して開発を行えるようになります。

8.1.2　CI に必要な機能

CIが必要とする機能は次の４つです。

- バージョン管理システム
- ビルドスクリプト
- フィードバック手段
- 変更を起点とした実行

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機能するかを継続的に検証します。そのためにはソースコードが変更されるたびにCIを実行するのが効果的です。ソースコードの変更はバージョン管理システムを使用してください。

ビルドスクリプト

CIのインテグレーションプロセスは簡単に実行できる必要があり、ビルドスクリプトを記述することで自動化します。ビルドスクリプトでは常に同じ方法、順序でビルドプロセスが実行されることを保証します。

フィードバック手段

CIではインテグレーションを頻繁に行うことでソースコードへの問題の混入やプロジェクトの悪い兆候を捉えます。適切なフィードバック手段がない場合、プロジェクトの異変に気づくのが遅れ、対応や修正の遅れへと伝播します。メールやSlackといったコミュニケーションツールが使われています。

変更を起点とした実行

実行そのものは必ずしも自動化されている必要はありません。重要なのはソースコードを変更するたびにCIを実行することです。しかし、ソースコードの変更検知とCIの実行を自動化すれば実行忘れを防ぎ、より確実に成果を得られます。自動化によって得られた時間でより生産的な作業を行いましょう。

8.1.3　CI ツール・サービスの選定基準

本節では、Android プロジェクトへ CI を導入するための準備について解説します。

CI システム構築に必要な CI ツール・サービスは数多くありますが当然、それぞれで特徴が異なります。何を重視するかは企業や開発チームで異なるので各自が適したものを選ぶ必要があります。

CI サーバーの利用形態として自分たちで CI サーバーを管理する**セルフマネージド型**とクラウド上の CI サーバーをサービスとして利用する**SaaS 型**があります。セルフマネージド型の CI サーバーは、以前はオンプレミスな環境に構築していました。現在は AWS や GCP、Azure といったクラウドサービスが普及し、クラウド上に CI サーバーを構築することが増えています。SaaS 型の CI サーバーでは、CI サーバーの構築や運用を行わずに簡単に CI に必要な機能を利用できるようになりました。

表 8.1 は主要な CI ツール・サービスとその利用形態の一覧です。

● 表 8.1　主要な CI サーバー

CI ツール・サービス名	主な利用形態
Jenkins	セルフマネージド
CircleCI	SaaS/ セルフマネージド[注 1)]
Bitrise	SaaS
Travis CI	SaaS/ セルフマネージド
Wercker	SaaS
Gitlab CI/CD	SaaS/ セルフマネージド
Bitbucket Pipelines	SaaS
Visual Studio App Center	SaaS

導入コスト

導入コストを環境構築コストと利用を開始するまでのコストの２点に分けて考えます。

- 環境構築コスト
- 利用開始コスト

表 8.2 に示すように、セルフマネージド型の導入コストは SaaS 型と比較して高くなります。

● 表 8.2　CI サーバー利用形態によるコスト

コスト	セルフマネージド	SaaS
環境構築コスト	高	低
利用開始コスト	高	低

注 1)　CircleCI Enterprise をセルフマネージド型として分類しています。https://circleci.com/enterprise/

セルフマネージド型の環境構築コストは専用のサーバーを用意してCIサーバーを構築するためSaaS型と比較して高くなります。一方SaaS型はサービスへのサインアップを済ませればすぐに利用可能になるため環境構築コストはほとんどかかりません。

利用開始コストは実際にCIサーバーとしてビルドプロセスを実行できるようになるまでのコストです。SaaS型の代表となるCircleCI、Bitrise、Travis CIといったサービスでは設定ファイルの導入と管理コンソールからの数クリックで利用を開始できます。セルフマネージド型の代表となるJenkinsではSaaS型と比較すると設定が必要な項目数が多く利用開始までにはより多くの時間を要します。

運用コスト

運用コストはCIをプロジェクトに導入してから維持していくためのコストで次の２点を指します。

- 人的コスト
- 金銭的コスト

セルフマネージド型では構築したCIサーバーを運用し続ける必要があるため、運用を行う人的コストとサーバー自体の金銭的コストがかかります。SaaS型でもサービス使用料という形で金銭的コストを支払いますが、人的なコストを削減できます。

SaaS型CIサービスの料金プランはサービスによって異なりますが、おおむね利用できる機能やビルドマシンの性能が選ぶ料金プランによって変化します。より高額なプランを選択するほど潤沢なリソースを確保でき、ビルドを速く安定的に実行する余地が生まれます。

カスタマイズ性

CIツール・サービスを利用する際に影響を与えるカスタマイズ性について解説します。カスタマイズ性の指標を次の２つと定義します。

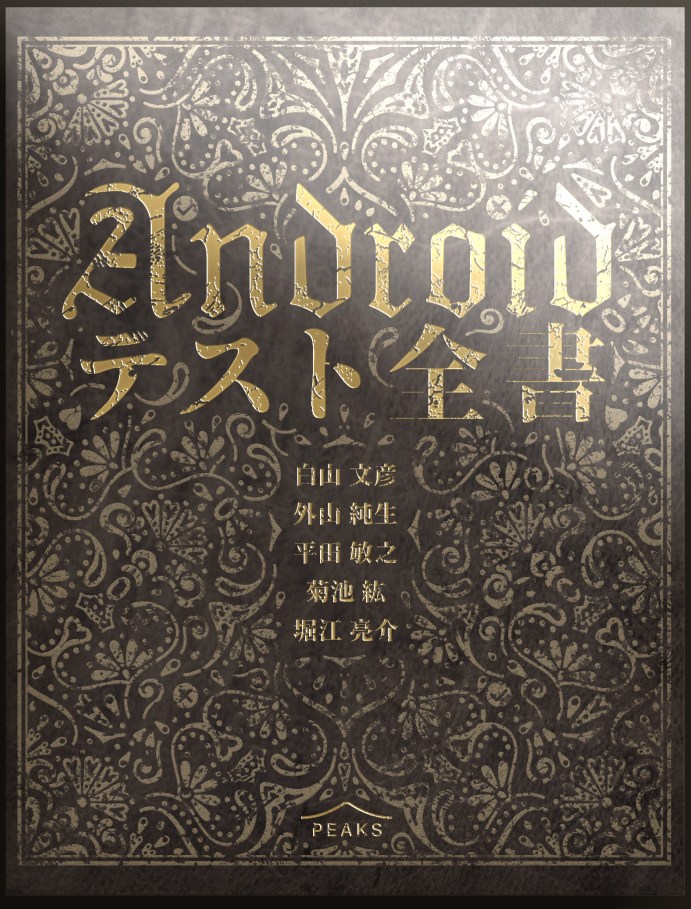
- CIツール・サービス自体の拡張性
- ビルドプロセス構成の柔軟性

CIツール・サービス自体の拡張性とは、プロジェクトや開発チームに合わせてCIツール・サービスを拡張できるかを指します。たとえばJenkinsでは、プラグイン機能によってJenkins自体の機能の拡張をできるようにしています。プラグインは必要に応じて作成可能であり、プロジェクトが必要とするものを何でも実現できます。

ビルドプロセス構成の柔軟性とはCIのビルドプロセスを開発フローに合わせてカスタマイズできることを指します。この仕組みはJenkinsではPipelinesと、CircleCIやBitrise、WerckerではWorkflowと呼ばれ、開発からリリースまでの流れを可視化して捉えられるようになります。

ビルドプロセス構成の柔軟性は、後述するデプロイメントパイプラインの実現に役立ちます。

この続きを読むには...



『Androidテスト全書』
電子版: ¥3,800　製本版: ¥4,200

購入して続きを読む

PEAKS