

# Swift Testing を活用する

---



宇佐見公輔

2024-10-18

株式会社ゆめみ

# 自己紹介

- 宇佐見公輔
- 株式会社ゆめみ
  - ▶ iOS テックリード
- 50 歳になりました
  - ▶ 節目なので、健康診断で MRI とか CT スキャンとか

# 最近のアウトプット

- Swift Testing
  - ▶ 関モバ #5
- 3次元回転とクォータニオン / Accelerate
  - ▶ iOSDC ポスターセッション / Mobile Act OSAKA 13
- iOS アプリ開発の知識
  - ▶ iOSDC パンフレット記事
- Apple Vision Pro の UI / フォーカス操作
  - ▶ 関モバ #4 / YUMEMI.grow Mobile #15

# Swift Testing とは

---

# Swift Testing とは

## Swift 用の単体テストフレームワーク

- Swift 公式の GitHub リポジトリで公開されている
  - <https://github.com/swiftlang/swift-testing>
- Xcode 16 に統合された
- XCTest との関係
  - 従来の XCTest に比べて、Swift の機能をより活用
  - XCTest と同じプロジェクトで混在可能

# Swift Testing の構成要素

- テスト関数
  - `@Test` 属性
- 期待値の確認
  - `#expect` マクロ
- テストスイート
  - `@Suite` 属性
- トレイト
  - `TestTrait` / `SuiteTrait`

# テスト関数

---

# テスト関数定義：XCTest

```
import XCTest

class FoodTruckTests: XCTestCase {
    func testEngineWorks() {
        // ...
    }
}
```

- XCTestCase のサブクラス内で定義する必要がある
- メソッド名を test 始まりで命名する必要がある



# テスト関数定義：Swift Testing

```
import Testing

struct FoodTruckTests {
    @Test
    func engineWorks() {
        // ...
    }
}
```

- テスト関数はどこで定義してもよい
- メソッドに `@Test` 属性をつければテスト関数になる

# 期待値の確認

---


# 期待値の確認

```
// XCTest
func testEngineWorks() throws {
    XCTAssertNotNil(engine.parts.first)
    XCTAssertGreaterThan(engine.batteryLevel, 0)
    XCTAssertTrue(engine.isRunning)
}

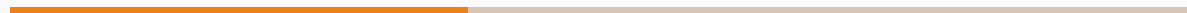
// Swift Testing
@Test func engineWorks() throws {
    try #require(engine.parts.first != nil)
    #expect(engine.batteryLevel > 0)
    #expect(engine.isRunning)
}
```

# Xcode 上の表示

#expect マクロはテスト失敗時の結果をきれいに表示する

```
@Test func videoMetadata() {  
9     let expectedMetadata = Metadata(duration: .seconds(90))  
10    #expect(video.metadata == expectedMetadata)  Expectation failed: (video.metadata → duration: 0.0...  
  
    Results  
    ▾ video.metadata: duration: 0.0 seconds, resolution: 3840.0 × 2160.0  
      Metadata  
      > duration: 0.0 seconds  
        Duration  
      > resolution: 3840.0 × 2160.0  
        Resolution  
  
    ▾ expectedMetadata: duration: 90.0 seconds, resolution: 3840.0 × 2160.0  
      Metadata  
      > duration: 90.0 seconds  
        Duration  
      > resolution: 3840.0 × 2160.0  
        Resolution  
  
11 }
```

# テストスイート



# テストスイート

```
@Suite
struct FoodTruckTests {
    @Test
    func engineWorks() {
        // ...
    }
}
```

- テスト関数を含む型が、自動的にテストスイートになる
- 後述するトレイトを使う場合は `@Suite` 属性を指定する

# Swift の機能の活用

```
final class FoodTruckTests {  
    init() async throws {  
        // ...  
    }  
}
```

- 専用の setUp の代わりに、通常の init が使える
- actor や @MainActor などにも使える

トレイト





テスト関数やテストスイートの振る舞いを指定する

- テスト関数のタイムアウト時間を指定する例

```
@Test(.timeLimit(.seconds(30)))  
func serveLargeOrder() {  
    // ...  
}
```

# 用意されているトレイト

- `.enabled / .disabled`
- `.timeLimit`
- `.serialized`
- `.tags`
- `.bug`
- `.isRecursive`

# Swift Testing の機能

---

# Swift Testing の機能

Swift Testing ならではの機能をいくつか紹介

- エラーのテスト
- パラメトライズテスト
- テストの並列実行

# エラーのテスト

---

# エラーのテスト

```
var order = PizzaToppings(bases: [.calzone, .deepCrust])
#expect(throws: PizzaToppings.Error.outOfRange) {
    try order.add(topping: .mozzarella,
                  toPizzasIn: -1..0)
}
```

- 自前で do ~ catch するテスト実装が不要に
- throw されなければテスト失敗になってくれる
  - ▶ 自前でテスト実装したときにやりがちなミス

# エラーのテストのカスタマイズ

```
#expect {  
  FoodTruck.shared.engine.batteryLevel = 0  
  try FoodTruck.shared.engine.start()  
} throws: { error in  
  return error == EngineFailureError.batteryDied  
    || error == EngineFailureError.stillCharging  
}
```

- 単純な等価判定では都合が悪い場合は `errorMatcher` を実装
  - ▶ エラーが `Equatable` でない場合など

# パラメトライズテスト

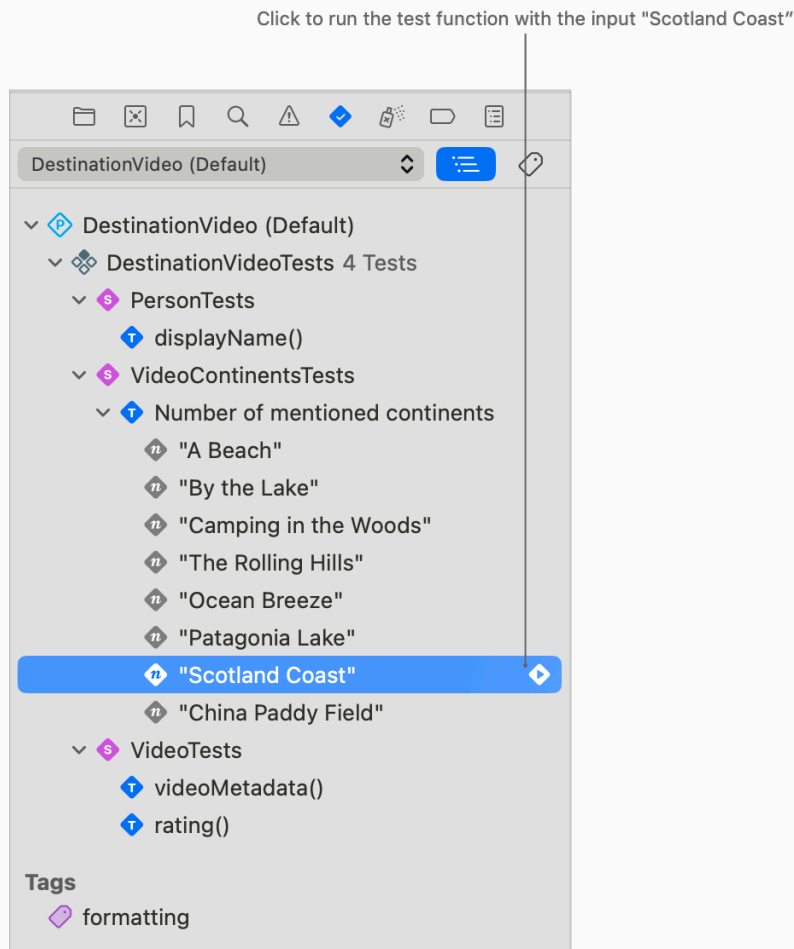




# パラメトライズテスト

```
enum Food {  
    case burger, iceCream, burrito, noodleBowl, kebab  
}  
  
@Test(arguments: [  
    Food.burger, .iceCream, .burrito, .noodleBowl, .kebab  
])  
func foodAvailable(_ food: Food) async throws {  
    let foodTruck = FoodTruck(selling: food)  
    #expect(await foodTruck.cook(food))  
}
```

# Xcode 上の表示



Xcode のテストナビゲータで、パラメータごとに分かれて表示される

- それぞれのテスト結果がわかる
- 特定パラメータだけで実行できる

# テストの並列実行

---

# テストの並列実行

- デフォルトで、テストは並列実行される
- テスト関数
  - パラメトライズテストの場合、各テストが並列実行
- テストスイート
  - 各テスト関数やサブスイートが並列実行

# テストの直列実行

グローバルな状態を操作するテストは並列実行だと困る

```
@Suite(.serialized)
struct FoodTruckTests {
    @Test func refill() { ... }
    @Test func startEngine() async throws { ... }
}
```

- `.serialized` トレイトで並列でなく直列実行にできる

# 複数テストスイートの直列実行

```
@Suite(.serialized) struct MultipleTests {}  
  
extension MultipleTests {  
    struct FoodTruckTests { ... }  
}  
  
extension MultipleTests {  
    struct OtherTests { ... }  
}
```

- サブスイートも直列実行にできる
- なお、`.serialized` トraitはサブスイートにも適用される

まとめ



- Swift Testing の構成要素
  - テスト関数、期待値の確認、テストスイート、トレイト
- Swift Testing の機能
  - エラーのテスト
  - パラメトライズテスト
  - テストの並列実行