
品質向上！Android アプリケーションテスト



品質向上！ Android アプリケーション テスト

2014-03-07 版 発行

トレーニング概要

トレーニングの目的

- テスティングフレームワークを使って効率のよい単体テスト方法の習得
- テストツールを活用して各種テスト（ストレステスト、シナリオテスト、カバレッジテスト、など）を効率よい実行方法の習得

受講前提条件

- 基本的な Android アプリケーション開発ができること。もしくは、Android アプリケーション開発、Android プログラミング入門コースのいずれかを受講されている方
- JUnit の基本的な知識

トレーニングスケジュール

1日目

- 第 1 章 テストの自動化
- 第 2 章 JUnit の基本
- 第 3 章 チュートリアル
- 第 4 章 Unit Test

2日目

- 第 5 章 JUnit UI Testing
- 第 6 章 JUnit の応用技術
- 第 7 章 ストレステスト
- 第 8 章 シナリオテスト
- 第 9 章 コードカバレッジ
- 第 10 章 ビルドツールと CI ツール
- 第 11 章 まとめ

開発環境

本トレーニングで使用する Android 開発環境は以下のとおりである

表 1 開発環境

ソフトウェア	バージョン
Eclipse	Eclipse IDE with built-in ADT Ver.22
Java SDK	JDK.6
Android Pratform SDK	Android 4.2.2 (API 17)

表 2 ディレクトリ構成

ディレクトリ名 (C:\android_training_test 以下)	説明
adt-bundle-windows-x86\sdk	AndroidSDK
adt-bundle-windows-x86\ eclipse	Eclipse
workspace	Eclipse のワークスペース
answer_docs\html	実習の解答ドキュメント

実習の注意事項

- テスト実行時はエミュレータ、デバイスのロック画面は解除しておく（一部のテスト API はロック解除状態でないと成功しません）
- 次のテストではエミュレータで実行する
 - DatabaseTesting
 - ProviderTesting
 - monkeyrunner uiautomator
 - emma

目次

トレーニング概要	i
トレーニングの目的	i
受講前提条件	i
トレーニングスケジュール	i
開発環境	ii
実習の注意事項	ii
第1章 テストの自動化	1
1.1 概要	2
1.2 テストとは？	3
1.2.1 テストの必要性	3
1.2.2 テストの問題点	4
1.2.3 テストの自動化	5
1.3 技術的側面からみたテストの自動化	6
1.3.1 テスト技法	7
1.3.2 テスト自動化の注意事項	9
1.3.3 豊富なライブラリ	10
第2章 JUnit の基本	11
2.1 概要	12
2.2 ユニットテスティングフレームワーク	13
2.3 JUnit とは	14
2.3.1 TestCase クラス	14
2.4 JUnit の基本的な使い方	15
2.4.1 説明	16
2.5 Android のテスティングフレームワーク	17
2.6 Android の JUnit	18
2.6.1 Android Test Project	18
2.6.2 Android の TestCase	19
2.6.3 主要な TestCase と API	19
2.6.4 Test API	20
2.6.5 TestCase	20

目次

2.6.6 Assertion	20
第3章 チュートリアル	21
3.1 概要	22
3.2 【実習】HelloJUnit	23
3.2.1 ターゲットアプリケーションの説明	24
3.2.2 テスト内容	25
3.2.3 テストプロジェクトを作成する	25
3.2.4 手順 1. 新規テストプロジェクトの作成	26
3.2.5 手順 2. テストケースの作成	28
3.2.6 手順 3. テストメソッドの作成	31
3.2.7 実行	33
3.2.8 実行結果の確認	33
第4章 Unit Test	36
4.1 概要	37
4.2 主要な TestCase	38
4.2.1 AndroidTestCase	39
4.2.2 InstrumentationTestCase	39
4.2.3 ApplicationTestCase	40
4.2.4 コンポーネント固有のテストケース	40
4.3 Activity Testing	41
4.3.1 Activity の特徴	42
4.3.2 Activity 用の主な TestCase	45
4.4 Activity Testing Sample	46
4.4.1 ライフサイクルのテスト	46
4.4.2 手順 1. ActivityUnitTestCase のサブクラスを作成する	47
4.4.3 手順 2. 初期化 (コンストラクタ、 setUp、 tearDown など)	47
4.4.4 手順 3. テストメソッドを作成する	49
4.4.5 MainActivityLifecycleTest.java	50
4.5 【実習】Activity Testing	51
4.5.1 実習	51
4.5.2 確認	51
4.5.3 解答	51
4.6 Service Testing	52
4.6.1 Service の特徴	53
4.6.2 サービスのライフサイクル	54
4.6.3 ServiceTestCase	54
4.7 Service Testing Sample	56
4.7.1 アプリケーションの説明	57
4.7.2 テストケースの作成	58

目次

4.7.3	ServiceTestCase の作成方法	58
4.7.4	手順 1.TestCase の作成	59
4.7.5	手順 2. コンストラクタの作成	59
4.7.6	手順 3. テストメソッドの作成	59
4.7.7	動作確認	61
4.8	【実習】Service Testing	62
4.8.1	実習	62
4.8.2	確認	62
4.8.3	解答	62
4.9	Database Testing	63
4.9.1	データベーステストの特徴	64
4.9.2	独立したテスト環境	65
4.10	Database Testing Sample	68
4.10.1	アプリケーションの説明	69
4.10.2	テスト内容	69
4.10.3	プロジェクト概要	70
4.10.4	テストプロジェクト概要	72
4.10.5	データ取得テストを作成する	72
4.10.6	手順 1. Fixture の準備	73
4.10.7	手順 2. BookDaoTest クラスの作成	73
4.10.8	手順 3. テストメソッドの作成	74
4.10.9	全体ソース	75
4.10.10	テストの実行	76
4.11	【実習】Database Testing	77
4.11.1	実習	77
4.11.2	確認	77
4.11.3	解答	77
4.12	【補足実習】Database Testing	78
4.12.1	確認	79
4.12.2	解答	79
4.13	【補足実習 2】テストコンテキストを使った Database Testing	80
4.13.1	テストコンテキストを取得する	80
4.13.2	テストケースの作成	81
4.13.3	手順 1. Fixture の作成 <実装済み>	81
4.13.4	手順 2. Fixture ユーティリティの作成 <実装済み>	81
4.13.5	手順 3. TestCase の作成 <一部未実装>	83
4.13.6	手順 4. テストメソッドの作成	85
4.13.7	確認	87
4.13.8	解答	87
4.14	Provider Testing	88

目次

4.14.1 Provider テストの特徴	89
4.14.2 ProviderTestCase2	89
4.14.3 MockContentResolver	90
4.15 Provider Testing Sample	91
4.15.1 アプリケーションの説明	92
4.15.2 アプリケーションの使い方	94
4.15.3 ContentProvider のテスト方法	94
4.15.4 手順 1. テストケースの作成	95
4.15.5 手順 2. コンストラクタの作成	96
4.15.6 手順 3. setUp メソッドのオーバライド	96
4.15.7 手順 4. テストメソッドの作成	96
4.15.8 テストの実行	99
4.16 【実習】Provider Testing	100
4.16.1 実習	100
4.16.2 確認	100
4.16.3 解答	100
4.17 【補足資料】テストコンテキストを使った Provider Testing	101
4.17.1 テストコンテキストを取得する	101
4.17.2 テストメソッドの作成	101
第 5 章 JUnit UI Testing	103
5.1 概要	104
5.2 UI テストの特徴	105
5.2.1 UI を操作する	105
5.2.2 UI テストの必要性	107
5.3 UI Testing Sample	108
5.3.1 アプリケーションの説明	109
5.3.2 テストプロジェクト概要	109
5.3.3 手順 1. ActivityInstrumentationTestCase2 のサブクラスを作成する	110
5.3.4 手順 2. テストメソッドを作成する	110
5.3.5 確認	114
5.4 【実習】UI Testing	115
5.4.1 実習	115
5.4.2 確認	115
5.4.3 解答	115
第 6 章 JUnit の応用技術	116
6.1 概要	117
6.2 テストレポート	118
6.2.1 テストレポートを作成する	118
6.2.2 テストレポートの作成方法	118

目次

6.2.3 【実習】テストレポート	120
6.3 TestSuite	121
6.3.1 TestSuite Sample	122
6.3.2 指定したパッケージ配下のテストケースをまとめて実行する	122
6.3.3 TestSuite を実行する	124
6.3.4 【実習】TestSuite 1	125
6.3.5 複数の TestSuite をまとめて実行する	126
6.3.6 実行	126
6.3.7 【実習】TestSuite 2	127
6.3.8 指定した TestCase をまとめて実行する (APIlevel 16 以降)	128
6.3.9 実行	128
6.3.10 【実習】TestSuite 3	130
6.4 TestSize	131
6.4.1 TestSize の指定	131
6.4.2 TestSize の使い方	132
6.5 【実習】TestSize	135
6.5.1 実習	135
6.5.2 確認	135
第 7 章 ストレステスト	136
7.1 概要	137
7.2 ストレステストとは	138
7.3 Monkey ツール	139
7.3.1 Monkey ツールの使い方	140
7.3.2 Monkey ツールの実行	141
7.3.3 【実習】Monkey ツール	142
7.3.4 Monkey ツール実習	142
第 8 章 シナリオテスト	144
8.1 概要	145
8.2 シナリオテストとは	146
8.2.1 テストシナリオ	146
8.2.2 Android UI Testing	146
8.2.3 ツールの紹介	147
8.3 monkeyrunner	148
8.3.1 monkeyrunner が提供している機能	148
8.3.2 monkeyrunner の API	149
8.4 monkeyrunner Sample	150
8.4.1 テストシナリオの作成	150
8.4.2 テストスクリプトの作成	151
8.4.3 monkeyrunner の実行	154

目次

8.5 【実習】monkeyrunner	156
8.6 uiautomator	157
8.6.1 uiautomator	158
8.6.2 操作をシミュレートする	158
8.7 uiautomator Sample	160
8.7.1 uiautomator の使い方	160
8.7.2 準備	160
8.7.3 UI Automator Viewer とは	161
8.7.4 UI の解析	164
8.7.5 プロジェクトの作成	166
8.7.6 テストケースの作成	168
8.7.7 プロジェクトのビルド	172
8.7.8 実行 uiautomator ツールを使ってテストを実行する	174
8.8 【実習】uiautomator	175
第9章 コードカバレッジ	176
9.1 概要	177
9.2 カバレッジとは	178
9.2.1 カバレッジの基準	178
9.2.2 emma とは	179
9.3 emma Sample	180
9.3.1 emma の使い方	180
9.3.2 手順 1. 対象プロジェクトのビルドとインストール	180
9.3.3 手順 2. テストプロジェクトのビルドとインストールとテスト実行	181
9.3.4 手順 3. 実行結果レポートの確認	182
9.3.5 レポートの見方	184
9.4 【実習】カバレッジ	187
第10章 ビルドツールと CI ツール	188
10.1 概要	189
10.2 ビルドプロセスの自動化	190
10.2.1 ビルドプロセスとは	190
10.2.2 ビルドツール	191
10.3 繙続的インテグレーション	192
10.3.1 繙続的インテグレーションとは	192
10.3.2 CI ツール	192
10.3.3 CI ツールの導入を検討する	194
第11章 まとめ	196
11.1 トレーニングの振り返り	196
11.2 Android に関する書籍	197

目次

11.3	Android に関する情報提供元	197
11.4	OESF 公認 Android トレーニング	197
11.4.1	Android プログラミング入門	199
11.4.2	Android アプリケーション開発入門	200
11.4.3	Android アプリケーション開発応用	201
11.4.4	Android 応用 WebAPI 開発	202
11.4.5	Android タブレット開発コース	203
11.4.6	Android UI デザイン入門	204
11.4.7	品質向上！ Android アプリケーションテスト	205
11.4.8	Android 組み込み開発 基礎コース "Armadillo-440 編"	206

第1章

テストの自動化



1.1 概要

概要

- テストとは？
- テストのメリット
- 技術的側面からみたテストの自動化



iv - 2

1.2 テストとは？

テストとは

- コンピュータのプログラムを実行し、正しく動作するかどうか確認する作業のこと
- プログラムが正しく動作するか評価すること



iv - 3

テストとは作成したプログラムを実行し、システムが正しく動作しているか、クライアントの要件を満たしているかなどの検証することです。

- コンピュータのプログラムを実行し、正しく動作するかを確認する作業のこと
- プログラムが正しく動作するか評価をつけること
- クライアントの要件を満たしているか確認

1.2.1 テストの必要性

問題を含んだままシステムを出荷してしまうと、要求された機能が実行されないなどの故障が発生し、個人や会社の信用を失い、経済的損失を招く恐れがあります。

テストをすることによって以下のような効果が期待できる

- システムの問題を見出し、修正することで品質向上が期待できる
- テスト結果が数値化されることにより、アプリケーションの品質が分かる
- 問題のあるシステムを出荷してしまうことで発生する損失を避ける事ができる

1.2.2 テストの問題点

テストの問題点

- テストするタイミング
 - 適切なタイミングと定期的なテスト
- デグレード対応
 - テストの自動化
- 人件費
- テストの精度



iv - 4

テストによって様々な問題を取り除くことができるが、人為的なミスを避けるのは非常に困難であり、リソースやコストなどの理由で人力テストには限界があります。

<問題点>

- テストするタイミング
 - テストフェーズに検証を集中すると、問題が発見された時の後戻りの範囲が大きくなる
- デグレード対応
 - バグ対応やマイナスマージなどによるデグレード対応の発生と検証のため、繰り返しテストする必要がある
- 人件費
 - テスターが必要になる
 - テストの工数に比例して人件費が多くなる
- テストの精度
 - 単純な操作ミスや不慣れな操作のため、テストの精度が低くなる
 - レアケースや複雑な手順が必要であるなど、問題の再現が困難となる

1.2.3 テストの自動化

テストを自動化することによって、先に挙げた問題点の多くを解決することができます。また、テストそのものがプログラマ化されているため、スケジューラなどと連携して定期的に行うことができます。

- テストの自動化
 - 様々なテストツールの登場で、テストを自動化することができるようになる
 - 評価を自動でおこなうプログラムを作成することで、様々な問題点の解決が期待できる
- 利点
 - システムの問題を早期発見できる
 - テスト時に発生する人為的なミスを防ぐことができる
 - 同じテストを繰り返し実行が可能なため、品質の担保に役立つ
 - 複雑な演算処理を高速に実行できる
 - テストにかかる工数と人件費を減らすことができる
 - テスト結果が数値化されることにより、アプリケーションの品質が分かる
 - ほとんどのテストツールが、何件テストを行ったかなどの集計やレポートにも対応している

1.3 技術的側面からみたテストの自動化

技術的側面からみたテストの 自動化

- テスト技法
- テスト自動化の注意事項
- 豊富なライブラリ



iv - 5

1.3.1 テスト技法

テスト技法



- テスト計画
 - ブラックボックステスト
 - ホワイトボックステスト

- テストの種類
 - ユニットテスト
 - シナリオテスト
 - ストレステスト
 - パステスト

iv - 6

テストには以下の様な種類があり、工程や着目点などに応じて使い分ける

<テスト計画>

何をテストするのかなど、テスト項目を決めるための抽出方法として「ブラックボックステスト」 「ホワイトボックステスト」があります。

- ブラックボックステスト
 - 外部からの動作に着目したテスト
 - システムの構造には着目せず、インプットとアウトプットにのみ着目したテスト技法
- ホワイトボックステスト
 - ブラックボックステストとは対照的にプログラムの中身に着目したテスト
 - プログラムのアルゴリズムやコード内の処理を、どのくらい網羅しているかなどの確認をする

<テストの種類>

テスト項目が決まったら、項目に合わせたテストを行います。テストの種類には以下のようなものがあります。

- ユニットテスト
 - 単一のモジュールを対象としたテスト

- モジュールが期待した動作を行うかを、メソッド単位でテストする
- シナリオテスト
 - テストシナリオを用意し、システムが一連の動作を正しく行うかをテストする（テストシナリオ：アプリケーション内の一連の動作のこと）
- ストレステスト
 - 大きな負荷をかけても、システムが正常に機能するかをテストする

1.3.2 テスト自動化の注意事項

テスト自動化の注意事項

- 繰り返し実行できる
- 順番に依存しない
- 外部に影響を与えない
- 簡単に実行できる



iv - 7

テストコードは、アプリケーションコードとは違う役割を持っています。動作を検証するためのプログラムであり、製品としてのアプリケーションとは違ったコーディングスタイルになります。動作確認のため、繰り返しテストすることができなくてはいけません。また、各テストが独立して実行出来るように順番に依存したテストコードを作成してはいけません。

繰り返し実行できる

問題が発見したときにそれが再現できないと、問題が解消されたかどうかが分からなくなってしまいます。そのため、何度も繰り返し実行出来るようにする必要があります。

- 簡単に問題の再現ができる
- 問題が解消されたかどうか、いつでも確認できるようにする

順番に依存しない

JUnitなどの自動テストは順番が保証されません。そのため、あるテストを実行しないと成功しない、またはその逆のようなテストは作ってはいけません。いつでも対象のテストだけ実行できるようにしておきます。

- 対象のテストだけ実行することができる
- あるテストを実行しないと成功しない、またはその逆のようなテストを作ってはいけない

外部に影響をあたえない

テストによっては、データベースや外部ファイル、サーバ連携など、外部の環境に依存するケースが存在します。また、それらのテストを行うことで実際のデータの変更が発生してはいけません。外部アプリケーションに影響を与えないような、独立したテストアプリケーションのための環境が必要になります。モックオブジェクトやテストデータなどを使って、テストのための独立した環境を用意する必要があります

- 例)
 - DB の id=1 のデータを削除する
 - SD カードには hoge.txt を変更する
 - 午前 0 時になるとデータをサーバにアップロードする
 - Web サービスと連携して、データを取得する

簡単に実行できる

テストを実行する人はエンジニアとは限りません。テスターだったり、CI ツールかもしれません。複雑な前提手順や外部ツールの設定などに依存してはいけません。その場合は前提条件のクリアや依存関係の解決も自動化します。

- テストするための前提手順が、多数用意されていないこと
- テストを実行する前に、あれも必要、これも必要となってはいけない
- 外部ツールと連携する場合は、依存関係の解決をしておく

1.3.3 豊富なライブラリ

有名なテスティングフレームワーク「JUnit」では、様々なサードパーティ製のライブラリが提供されています。Android では Android 用のテストライブラリも複数登場しています。

- 豊富なテストツールとライブラリ
 - JUnit
 - * EasyMock
 - * Mockito
 - * PowerMock
 - JUnit for Android
 - * Robotium
 - * Mockito
 - * Robolectric
 - * AndroidMock

第2章

JUnit の基本



2.1 概要

概要

- ユニットテスティングフレームワーク
- JUnitとは
- JUnitの基本的な使い方
- Androidのテスティングフレームワーク
- AndroidのJUnit



2.2 ユニットテスティングフレームワーク

ユニットテスティングフレームワーク

- 単体テストを自動で行うテストプログラム作成用フレームワーク
- JavaではJUnitが一般的
- Android用にカスタマイズされたJUnitが提供されている



ユニットテスティングフレームワークとは、単体テストを自動で行うテストプログラム作成用フレームワークです。Java では JUnit が一般的です。Android テスティングフレームワークは、JUnit を Android 用に拡張した android.test パッケージ群や、その他 UI 操作やストレステストなどシーンに合わせていくつかのフレームワークが提供されています。

2.3 JUnit とは

JUnitとは

- Javaプログラムのユニットテスト(単体テスト)を行うための テスティングフレームワーク
- JUnit3、JUnit4が用意されている
- テストプログラムはクラス単位で用意する



*AndroidではJUnit3を元に拡張しているため、本トレーニングではJUnit3を対象に説明します

Java プログラムのユニットテスト (単体テスト) を行うための テスティングフレームワークです。 JUnit3、JUnit4 が用意されています。 JUnit 用の拡張ライブラリも豊富で、ホワイトボックステストに対応した強力なものも存在します。

テストプログラムはクラス単位で用意するのが一般的です。 テストクラスを作成するには、 junit.framework.TestCase クラスを継承する必要があります。 Eclipse では標準でインストールされており、外部の jar ファイルを参照する必要はありません。

2.3.1 TestCase クラス

TestCase クラスとは、対象のクラスをテストするためのクラスです。

対象のクラスの動作を確認するコードを記述します。

JUnit のテストクラスは、junit.framework.TestCase クラスを継承する必要があります。

2.4 JUnit の基本的な使い方

JUnitの基本的な使い方

- サンプルプログラムを使って説明
- サンプルプロジェクトは解答ドキュメントより提供済
- プロジェクト名 : JUnitSampleForJava



サンプルプログラムを使って JUnit の基本的な使い方を説明します。

表 2.1 プロジェクト概要

項目	説明
プロジェクト名	JUnitSampleForJava
クラス名	Foo
処理	2つの引数の合計を返す add メソッドが定義されている

リスト 2.1: 【テスト対象のコード】 Foo.java

```
1: package com.example.junitsample;
2:
3: public class Foo {
4:     public int add(int x, int y){
5:         return x + y;
6:     }
7: }
```

リスト 2.2: 【テストコード】FooTest.java

```
1: package com.example.junitsample;
2: import junit.framework.TestCase;
3:
4: public class FooTest extends TestCase {           // ...
5:
6:     private Foo foo = null;
7:
8:     protected void setUp() throws Exception {    // ...
9:         super.setUp();
10:        foo = new Foo();
11:    }
12:
13:    protected void tearDown() throws Exception {  // ...
14:        super.tearDown();
15:    }
16:
17:    public void testAdd() {                      // ...
18:        int actual = foo.add(4, 5);
19:        assertEquals(9, actual);                  // ...
20:    }
21: }
```

2.4.1 説明

- クラス定義
 - junit.framework.TestCase を継承し、クラスを作成する
 - クラス名は「テスト対象のクラス名」 + “Test” とすることが通例
- setUp メソッド
 - 各テストメソッド実行前に、毎回実行される
 - テストの準備（データの整備、オブジェクトの生成等）を、このメソッドで行う
- tearDown メソッド
 - 各テストメソッド終了後に、毎回実行される
 - テスト後のクリーンアップ（データ復元等）をこのメソッドで行う
- テストメソッド
 - 対象のメソッドの動作を確認するためのコードを記述する
 - メソッド名は testXXX とし、public void で引数なしとする
- assertEquals メソッド
 - 実行したテスト対象メソッドの結果が、期待値と一致することを確認する
 - 第一引数に期待値、第二引数にテスト対象メソッドの結果をセットする。2つの値が正しい場合はテスト通過、そうでない場合はエラーとなる
 - assertEquals の他に assertTrue, assertFalse 等がある

2.5 Android のテスティングフレームワーク

Androidのテスティングフレームワーク

- 様々なシーンに対応したテスティングフレームワークが提供されている
 - 単体テスト
 - UIテスト
 - シナリオテスト
 - ストレステスト



Android には単体テスト、シナリオテスト、UI テスト、ストレステストなど様々なシーンに対応したテスティングフレームワークが提供されています。

表 2.2 Android のテストフレームワーク

フレームワーク	種類	役割
JUnit	単体テスト	Android 用拡張 TestCase が複数用意されている
Monkey	ストレステスト	ランダムなクリック、タッチイベントやジェスチャーなどのイベントを実行
monkeyrunner	シナリオテスト	位置指定でタッチ、クリック、ドラッグを Android デバイスに送信する。 Python で記述する
uiautomator	シナリオテスト	UI テストケース作成ツール。 monkeyrunner 同等の機能を持っているが、java で記述する。 表示テキストを指定することで、画面の UI パーツを取得できる

2.6 Android の JUnit

AndroidのJUnit

- JUnit3を元にして、Android用にカスタマイズされ 提供されている
- テストプロジェクト(Android Test Project) を作成する必要がある
- 画面操作やライフサイクルのテストが可能
- 作成したテストアプリケーションをインストールして実行する



Android 用の JUnit は android.test パッケージに配置されています。これらのクラスには、以下の特徴があります。

- JUnit3 を元にして、Android 用にカスタマイズされている
- ユーザーの画面操作や、ライフサイクルのイベント (onCreate 等) を呼び出すことが可能
- テスト用アプリケーションとして、プロジェクトを用意する
 - Android の JUnit ランナーはエミュレータや実機に Test 用アプリケーションをインストールし実行する

2.6.1 Android Test Project

Android Test Project とはテスト対象のアプリケーションをテストするためのプロジェクトのことです。Java では、ターゲットアプリケーションのプロジェクトにテストディレクトリを用意することで、テストを作成していました。Android ではテスト用の独立したプロジェクトを作成します。

Android Test Project の特徴

- テスト対象のアプリケーションをテストするためのプロジェクト
- テストプロジェクトは、テストアプリケーションとして実行される
- テストプロジェクトのパッケージ名は、ターゲットアプリケーションのパッケージ名に.testを追加することが推奨されている
 - 例)
 - * ターゲット : com.mydomain.myapp
 - * テスト : com.mydomain.myapp.test
- テストプロジェクトへ、機能に対応した TestCase を追加する
- Android のテスト用に提供された API を利用して、テストコードを作成する

2.6.2 Android の TestCase

テスト対象に合わせて、Android 用に拡張された TestCase クラスが提供されています。用途に合わせてこれらのクラスを利用します。

- Activity
- Service
- ContentProvider

2.6.3 主要な TestCase と API

- Test API
 - Instrumentation
 - InstrumentationTestRunner
- TestCase
 - AndroidTestCase
 - InstrumentationTestCase
 - ActivityInstrumentationTestCase2
 - ActivityUnitTestCase
 - SingleLaunchActivityTestCase
 - ServiceTestCase
 - ProviderTestCase2
- Asserstion
 - MoreAsserts
 - ViewAsserts

2.6.4 Test API

Instrumentation

Instrumentation は Android のライフサイクルなどのコールバックを呼び出すことができます。Instrumentation を使うことにより、ライフサイクルを段階的に繰り返して実行することができるようになります。

特徴

- 操作系のテストの自動化をサポートするクラス
- ライフサイクルなどのコールバックメソッドを直接呼び出すことが可能になる。
- UI スレッドでの実行や、ブロック (wait) を可能にする。

InstrumentationTestRunner

InstrumentationTestRunner は、主要な Android テストランナークラスです。Android 用に拡張された TestCase クラスを実行することができます。

特徴

- Android 用に拡張された TestCase を実行するためのランナークラス
- テストケースクラスのロード、セットアップ、実行、および破棄を行う
- JUnit の TestRunner クラスを拡張し、Android で提供されている全ての TestCase を実行することが可能

2.6.5 TestCase

Android 用に拡張された TestCase が用意されています。(詳細については「4 章 Unit Test」で説明します)

2.6.6 Assertion

JUnit の Assert クラスを、Android 用に拡張された API が提供されています。

MoreAsserts

正規表現によるマッチングや、コレクションおよび配列などのマッチングに対応しています。

ViewAsserts

View に関するアサーションクラスです。画面の x,y 座標を指定した View の存在確認など、UI の配置、位置関係によるマッチングに対応しています。

第3章

チュートリアル



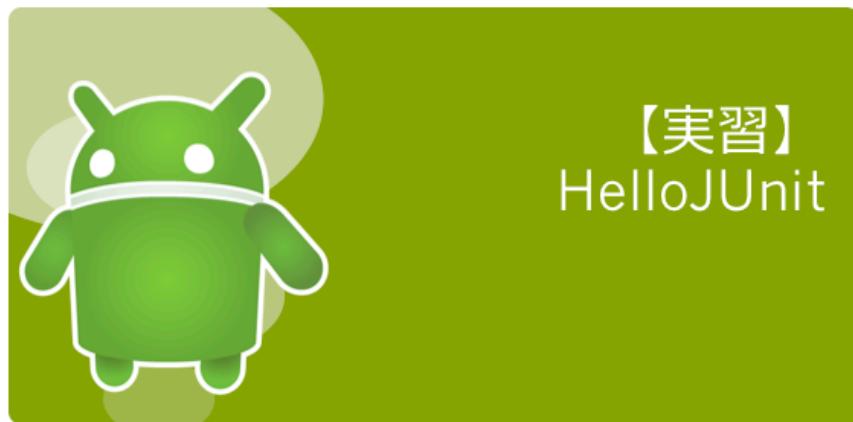
3.1 概要

概要

- テストプロジェクトの作成実習



3.2 【実習】HelloJUnit



HelloWorld アプリケーションのテストを作成し、テストプロジェクトの作成から実行までの一連の手順を習得します。

3.2.1 ターゲットアプリケーションの説明



ターゲットアプリケーションの説明

- HelloJUnit
 - 画面に「Hello world!」を表示するシンプルなアプリケーション

※トグルボタン押下時の動作は後述

アプリケーション名：「HelloJUnit」

画面に「Hello world!」の文字列を表示するアプリケーション。

トグルボタンを押したときの動作については後述します。

「HelloJUnit」アプリケーションは解答ドキュメントよりダウンロードして実行してください。

3.2.2 テスト内容



Hello world!と表示されている

■ 表示されているテキストが「Hello world!」になっていることを確認する

■ 対象プロジェクト

プロジェクト名	説明
HelloJUnit	テストターゲット
HelloJUnitTest	テストプロジェクト (新規作成)

HelloJUnit アプリケーションが正しく動作しているかをテストします。

- 確認項目
 - テキストの文字列が「Hello world!」になっていること

3.2.3 テストプロジェクトを作成する

HelloJUnit アプリケーションをテストするプロジェクト「HelloJUnitTest」を作成します。

テスト対象のアプリケーション「HelloJUnit」はあらかじめインストールしておく必要があります。

表 3.1 プロジェクト概要

項目	設定値
Project Name	HelloJUnitTest
Build Target	トレーニングで指定したバージョン
Test Target	HelloJUnit
Package	com.example.hellojunit.test

作成手順

1. 新規テストプロジェクトの作成
2. テストケースの作成
3. テストメソッドの作成

本トレーニングでは TestCase のサブクラスのことを「テストケース」と呼び称し、"test" から始まるメソッドのことを「テストメソッド」と呼び称します。

3.2.4 手順 1. 新規テストプロジェクトの作成

1. メニューバーから「File」、「New」、「Project」を選択
2. 「Android」、「Android Test Project」を選択し、「Next」ボタンをクリック

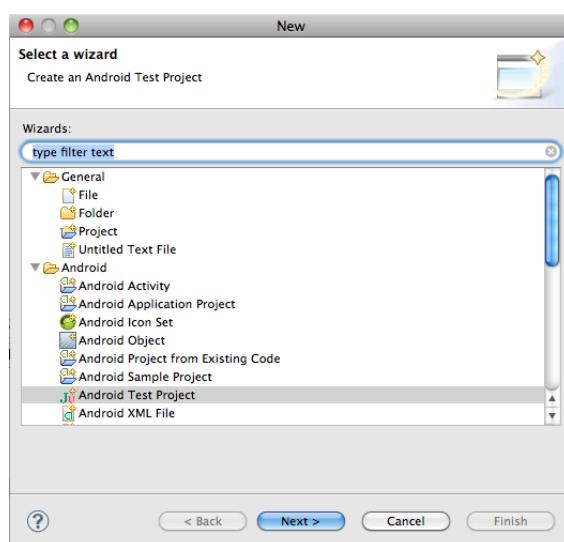


図: Select a wizard

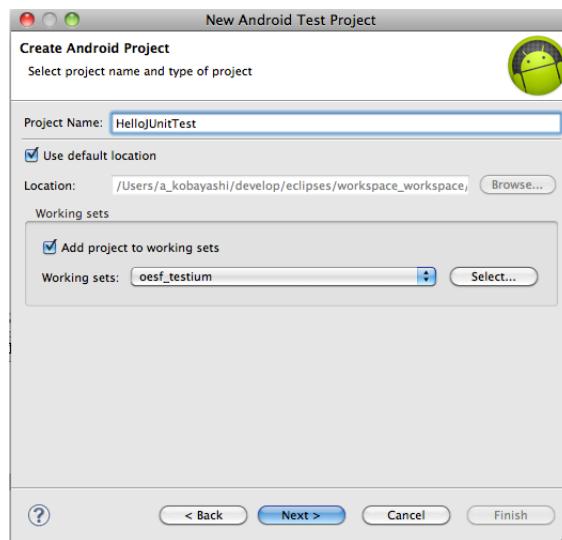
3. プロジェクト名に「HelloJUnitTest」と入力して「Next」ボタンをクリック

図: Create Android Project

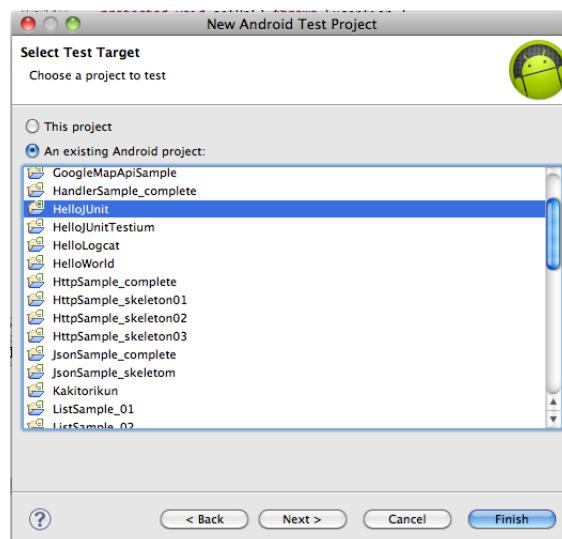
4. 目的のプロジェクト「HelloJUnit」を選択して「Finish」ボタンをクリック

図: Select Test Target

5. Package Explorer から、「HelloJUnitTest」プロジェクトが作成されていることを確認

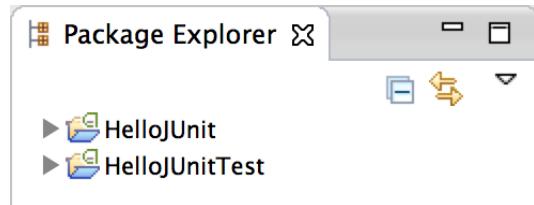


図: HelloJUnitTest プロジェクトが作成される

6. AndroidManifest.xml を確認します。

- instrumentation
 - "android:name" に "android.test.InstrumentationTestRunner" が指定されている
 - "android:targetPackage" に 単体テストをするプロジェクト「com.example.hellojunit」が指定されている
- uses-library
 - "android:name" に "android.test.runner" が指定されている

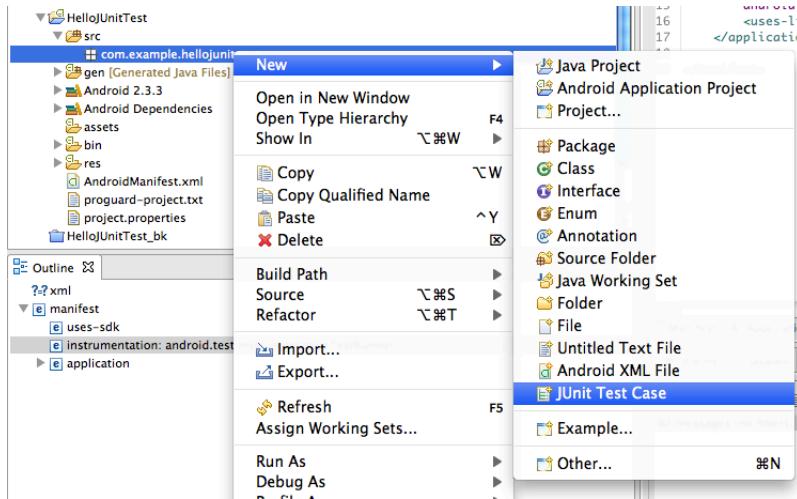
リスト 3.1: AndroidManifest.xml

```
1: <?xml version="1.0" encoding="utf-8"?>
2: <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3:   package="com.example.hellojunit.test"
4:   android:versionCode="1"
5:   android:versionName="1.0" >
6:
7:   <uses-sdk android:minSdkVersion="10" />
8:
9:   <instrumentation
10:     android:name="android.test.InstrumentationTestRunner"
11:     android:targetPackage="com.example.hellojunit" />
12:
13:   <application
14:     android:icon="@drawable/ic_launcher"
15:     android:label="@string/app_name" >
16:     <uses-library android:name="android.test.runner" />
17:   </application>
18:
19: </manifest>
```

3.2.5 手順 2. テストケースの作成

ActivityInstrumentationTestCase2 クラスのサブクラスを作成します。

1. src > "com.example.hellojunit.test" パッケージを選択して右クリックメニュー > 「New」 > 「JUnit Test Case」を選びます

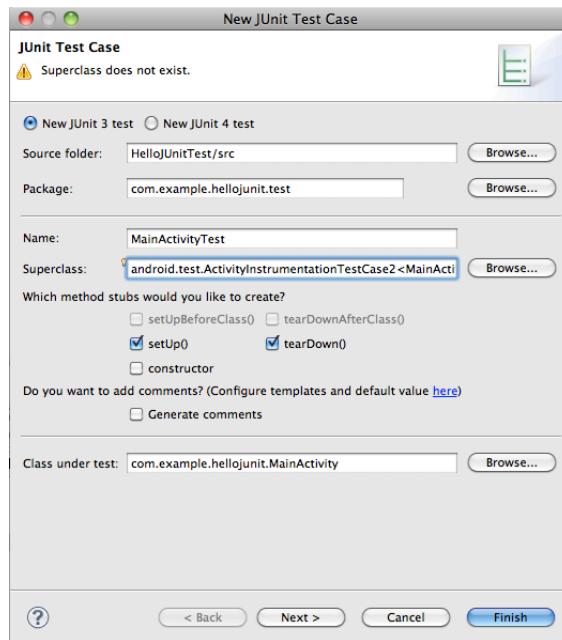


図：テストケースの作成 1

2. 各値を以下のように設定し「Finish」ボタンをクリック

表 3.2 MainActivityTest の設定値

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	HelloJUnitTest/src
Package	com.example.hellojunit.test
Name	MainActivityTest
Superclass	android.test.ActivityInstrumentationTestCase2<MainActivity>
setUp	チェック
tearDown	チェック
Class under test	com.example.hellojunit.MainActivity



図：テストケースの作成 2

3. テストケースが作成されていることを確認し、初期状態のエラーを修正する

- MainActivity の import
 - ActivityInstrumentationTestCase2 のジェネリクスにはテスト対象のクラスを指定するため import する
- コンストラクタの作成
 - コンストラクタでは対象のクラスの class を指定する

リスト 3.2: MainActivityTest クラスの作成

```
1: package com.example.hellojunit.test;
2:
3: import android.test.ActivityInstrumentationTestCase2;
4:
5: import com.example.hellojunit.MainActivity;
6:
7: public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {
8:
9:     public MainActivityTest() {
10:         super(MainActivity.class);
11:     }
12:
13:     protected void setUp() throws Exception {
14:         super.setUp();
15:     }
16:
17:     protected void tearDown() throws Exception {
```

```
18:         super.tearDown();
19:     }
20:
21: }
```

3.2.6 手順 3. テストメソッドの作成

TextView の値を確認するテストを作成します

1. メンバ変数の追加

テスト対象の Activity を保持するメンバ変数「activity」を定義する

リスト 3.6: メンバ変数の追加

```
1:     private MainActivity activity;
2:     private TextView textView;
```

2. setUp メソッドの修正

- getActivity メソッドを使って対象 Activity のオブジェクトを取得します
- TextView オブジェクトの取得
 - findViewById でリソース ID を指定して取得する
 - リソース ID のパッケージは"com.example.hellojunit"なので、import が必要

リスト 3.6: setUp メソッド

```
1:     import com.example.hellojunit.R;
2:
3:     public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {
4:
5:         //... 略...
6:
7:         protected void setUp() throws Exception {
8:             super.setUp();
9:             activity = getActivity();
10:            textView = (TextView) activity.findViewById(R.id.text_hello);
11:        }
12:    }
```

3. testText メソッドの追加

TextView の文字が正しいか確認するためのコードを追加します。

1. 期待する文字列をリソースから取得する

2. assertEquals メソッドを使って、表示文字列の正否を確認する

リスト 3.5: testText メソッド

```
1:     public void testText(){
2:         String expected = activity.getString(R.string.hello_world); // ... 1
3:         assertEquals(expected, textView.getText().toString());           // ... 2
4:     }
```

全体ソース

リスト 3.6: MainActivityText.java

```
1: package com.example.hellojunit.test;
2:
3: import android.test.ActivityInstrumentationTestCase2;
4: import android.widget.TextView;
5:
6: import com.example.hellojunit.MainActivity;
7: import com.example.hellojunit.R;
8:
9: public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {
10:
11:     private MainActivity activity;
12:     private TextView textView;
13:
14:     public MainActivityTest() {
15:         super(MainActivity.class);
16:     }
17:
18:     protected void setUp() throws Exception {
19:         super.setUp();
20:         activity = getActivity();
21:         textView = (TextView) activity.findViewById(R.id.text_hello);
22:     }
23:
24:     protected void tearDown() throws Exception {
25:         super.tearDown();
26:     }
27:
28:     public void testText(){
29:         String expected = activity.getString(R.string.hello_world);
30:         assertEquals(expected, textView.getText().toString());
31:     }
32: }
```

3.2.7 実行

ユニットテストはメソッドごと、またクラス単位での実行が可能です。ここでは、クラス単位でテストを実行します。(クラスの右クリックで、クラス単位で実行します。)

1. MainActivityTest 右クリック > 「Run As」> 「Android JUnit Test」を選択

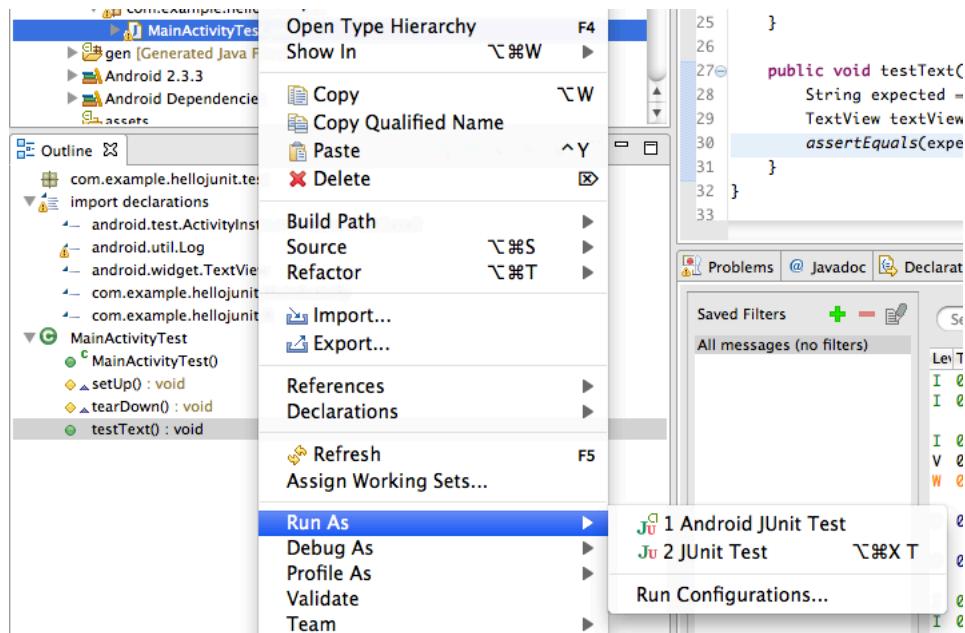


図: テストプロジェクトの実行

2. テストおよび対象アプリケーションが端末にインストールされる。テストの進捗に合わせ、結果を表すバーの表示が増えて行く。

3.2.8 実行結果の確認

実行結果が表示され、バーが緑になっているのを確認します。失敗した場合は、バーが赤くなります。

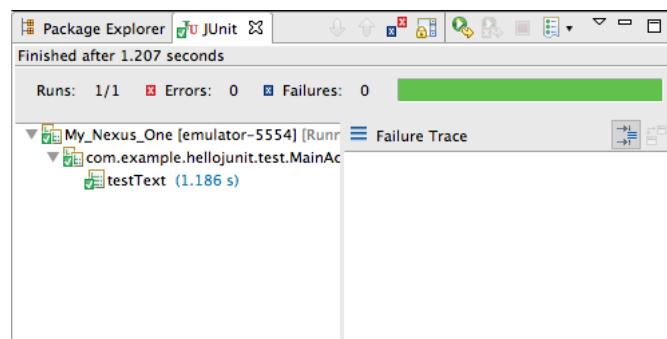


図: 成功

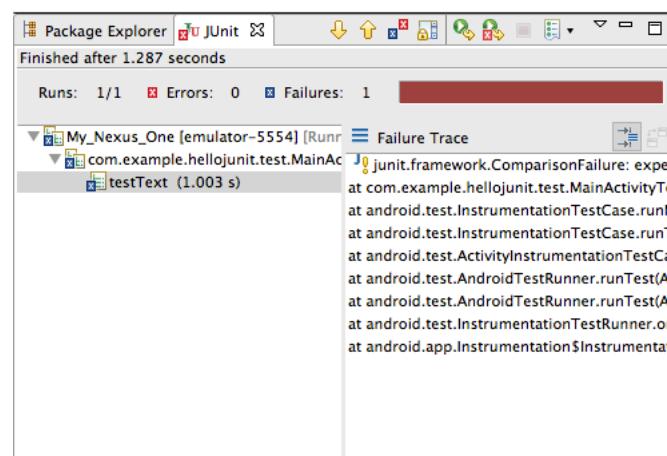


図: 失敗

実行結果画面の見方

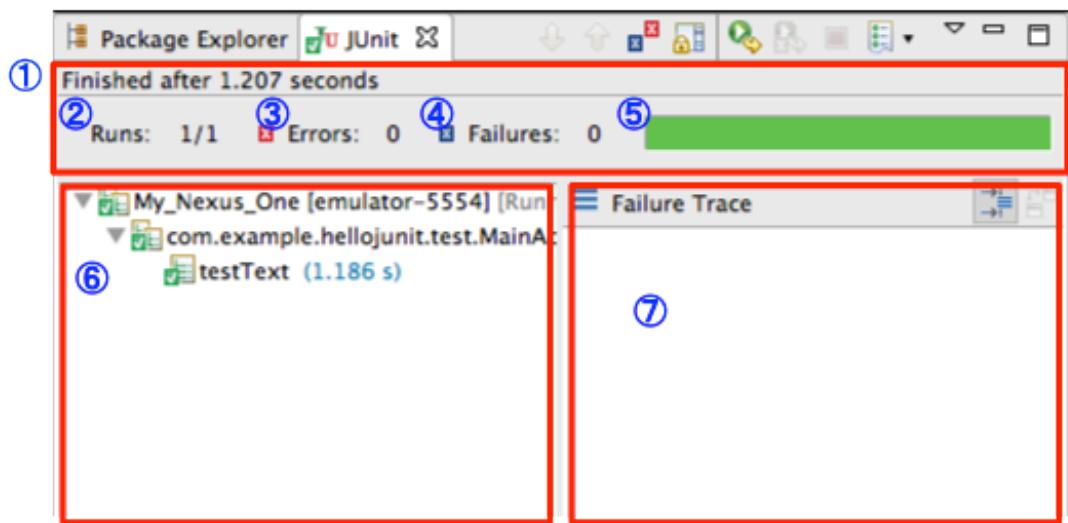
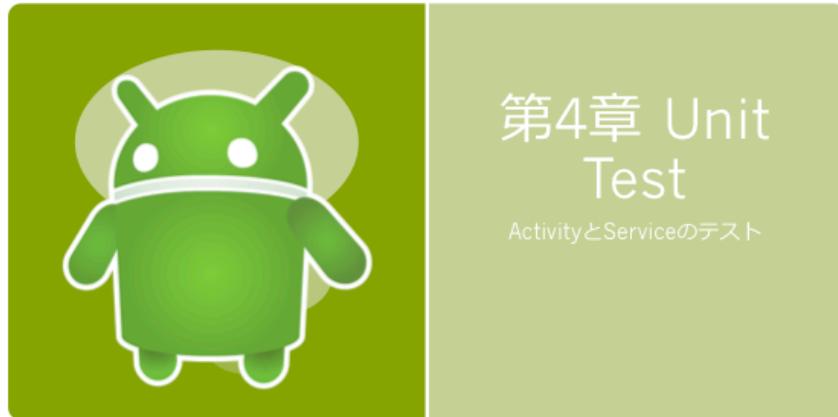


図: JUnit View

1. テストの実行にかかった時間です。
2. 実行の数: 実行したテストケースの数です。
3. エラーの数: テストの実行中に発生したプログラムエラーと例外の数です。
4. 失敗の数: テストの実行中に発生したテストの失敗の数です。これはアサーション失敗の数です。つまり、期待した結果と実際の結果が合わなかったということです。
5. プログレスバー: テストの実行とともにプログレスバーが左から右に伸びていきます。すべてのテストに成功すると、バーは緑色のままです。テストに失敗するとバーは緑色から赤色に変わります。
6. テストメソッドの要約: 実行したテストケースの一覧が表示されます。
7. Failure Trace: アサーションの失敗や例外が発生した場合は、ここにスタックトレースが表示されます。すべてのテストに成功した場合は、この枠は空になります。

第4章

Unit Test



4.1 概要

概要

- 主なTestCase
- Activity Testing
- Service Testing
- Database Testing
- Provider Testing



4.2 主要な TestCase

主要な TestCase

- 単体テスト用に提供された主要な TestCase
 - AndroidTestCase
 - InstrumentationTestCase
 - ApplicationTestCase
 - コンポーネント固有のテストケース



テスト対象に合わせて Android 用に拡張された TestCase クラスが提供されています。用途に合わせてそれらのクラスを利用します。

TestCase クラスと Android が提供しているテスト用クラスの関係

```
junit.framework.TestCase  
    InstrumentationTestCase  
        ActivityTestCase  
            |  
            ActivityInstrumentationTestCase2  
            |  
            ActivityUnitTestCase  
            SingleLaunchActivityTestCase  
        AndroidTestCase  
            ServiceTestCase  
            ProviderTestCase2  
            ApplicationTestCase
```

4.2.1 AndroidTestCase

基本的な Android の TestCase と Assert の両方を拡張した Android の汎用テストケースクラスです。リソースにアクセスする場合などに使用します。

特徴

- 基本的な Android の TestCase と Assert の両方を拡張した Android の汎用テストケースクラス
- リソースのアクセスなどの機能を提供している

4.2.2 InstrumentationTestCase

テストケースクラス内で Instrumentation クラスのメソッドを使用したい場合 (getActivity メソッドなど) に使用します。

主に Activity テストのための基底クラスとして用意されているが、テストケースクラス内で、Instrumentation クラスが提供しているメソッドを使用することができます。

特徴

- Instrumentation クラスのメソッドを使用することができます
- Activity のテストケースはこのクラスを拡張している
- Instrumentation クラスが提供しているメソッドの呼び出しに対応している
- ライフサイクルのようなコールバックイベントの呼び出しに対応している

4.2.3 ApplicationTestCase

ApplicationTestCase テストケースクラスを使って、 Application オブジェクトの setUp と tearDown をテストします。

このテストケースは、マニフェストファイルの <application> 要素が正しくセットアップされているかどうかを検証するのに役立ちます。

特徴

- アプリケーション全体のテストに有効
- <application>要素のセットアップの検証に有効

4.2.4 コンポーネント固有のテストケース

Activity, Service, ContentProvider などコンポーネント固有のテストケースが用意されています。

BroadcastReceiver, Fragment 用のテストケースクラスは用意されていません。

ActivityInstrumentationTestCase2

画面操作のテストを行うときに使います。

ActivityUnitTestCase

Activity の単体テストを行うときに使います。Activity のライフサイクルである onResume や onPause を適宜呼び出すことが出来ます。

SingleLaunchActivityTestCase

各テストケースが一つのアクティビティインスタンスとして使いまわしたい場合に利用します。

ServiceTestCase

サービスの単体テストを行うときに使います。

ProviderTestCase2

コンテンツプロバイダの単体テストを行う際に使います。

4.3 Activity Testing

Activity Testing

- Activityの特徴
- Activityのための主なTestCase
- Activity Testing Sample
- 実習



4.3.1 Activity の特徴

Activityの特徴

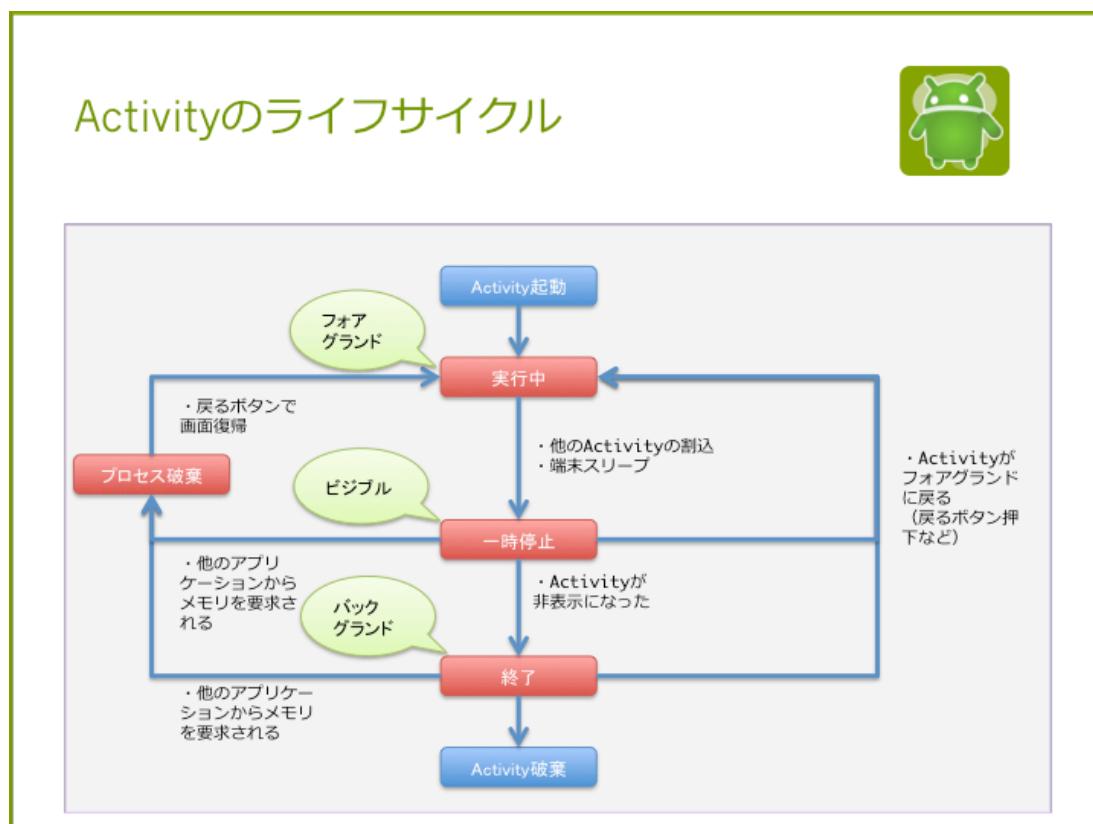
- ライフサイクルを持っている
- 直接ライフサイクルを呼び出すことはできない
- UIの操作はUIスレッドのみが許可されている



Activity は他のコンポーネントとは異なり、以下のような特徴があります。そのため、TestCase から UI 操作やライフサイクルの呼び出しを可能にした、Activity のためのテスクラスが提供されています。

- コールバック・メソッドに基づいたライフサイクルを持っている
- ライフサイクルは直接呼び出すことはできない
- UI の操作は UI スレッドのみが許可されている

Activity のライフサイクル



アクティビティにはオブジェクト生成～破棄に至るまでのライフサイクルが定義されています。
アクティビティの3種類の状態遷移が定められています（フォアグラウンド、バックグラウンド、
ビジブル）。

状態の変化が発生した時、イベントが通知されます。

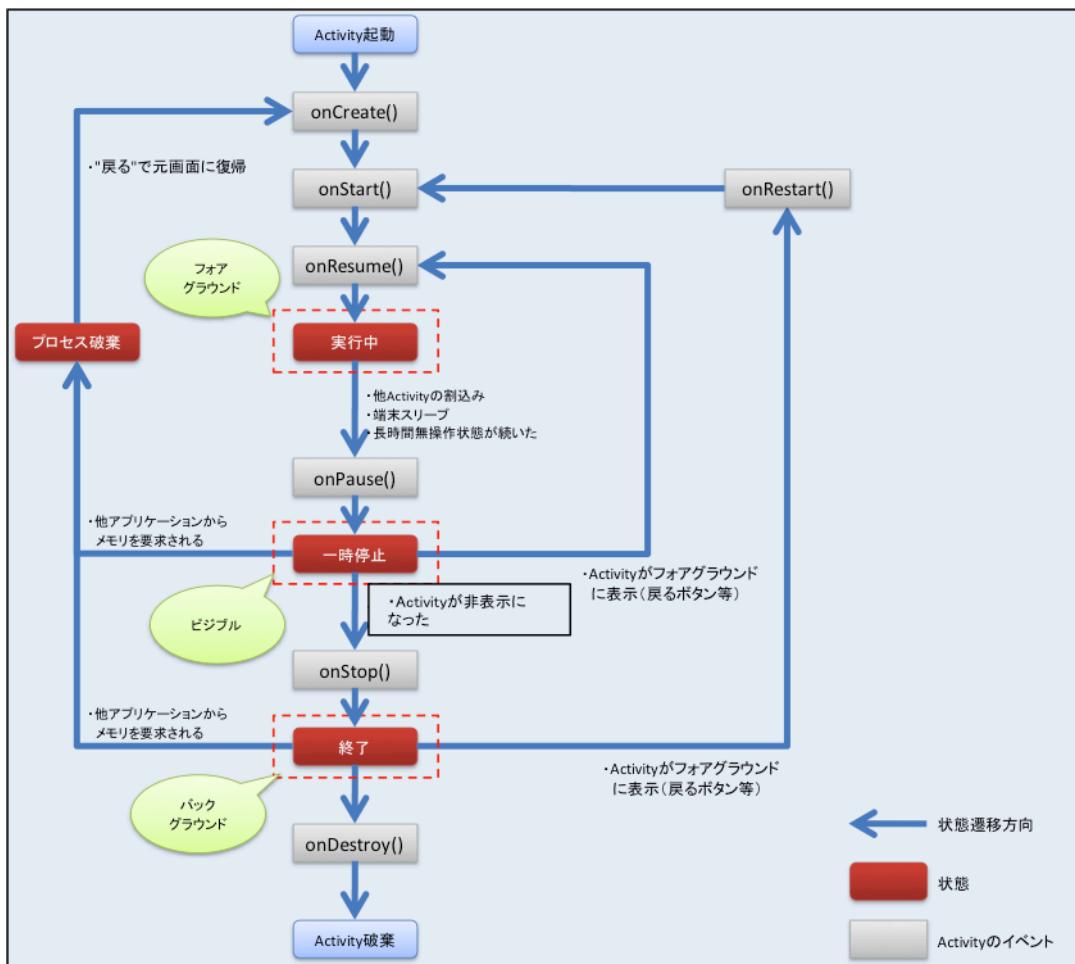


図: ライフサイクル

#	イベント名	内容
1	onCreate	最初の起動時に発生するイベント
2	onStart	アクティビティが表示される直前に発生するイベント
3	onResume	アクティビティが利用可能な状態(アクティブ状態)になる直前に発生するイベント
4	onPause	アクティビティがビジュアルになる直前に発生するイベント
5	onStop	onPauseの後、Activityが非表示の状態になった場合に発生するイベント (メモリ不足の際、イベントが発生しない場合がある)
6	onRestart	終了状態のアクティビティが再度表示される際に発生するイベント
7	onDestory	アクティビティが破棄される直前に発生するイベント (メモリ不足の際、イベントが発生しない場合がある)

図: 通知されるイベント

4.3.2 Activity 用の主な TestCase

ActivityInstrumentationTestCase2

- UI 操作のテストに使用する
- Button のクリックなど UI の操作イベントを処理することができる

ActivityUnitTestCase

- 画面表示や UI 操作に依存しないテストやライフサイクルのテストに使用する
- Activity のライフサイクルイベントを発行できる

SingleLaunchActivityTestCase

- 同一の Activity のインスタンスを利用したいときに使用する
- launchMode が standard 以外の Activity をテストすることができる

4.4 Activity Testing Sample

Activity Testing Sample

- HelloJUnitアプリケーションにライフサイクルのテストを追加する



- 対象プロジェクト

プロジェクト名	説明
HelloJUnit	テストターゲット
HelloJUnitTest	テストプロジェクト

4.4.1 ライフサイクルのテスト

HelloJUnit アプリケーションにライフサイクルのテストを追加します。

テスト対象のアプリケーション "HelloJUnit" の MainActivity には、変数 : lifecycle が定義されています。この変数の値の変化を検証します。

手順

1. ActivityTestCase のサブクラスを作成する
2. 初期化 (コンストラクタ、 setUp、 tearDown など)
3. テストメソッドを作成する

リスト 4.1: HelloJUnit アプリケーションの MainActivity

```
1: public class MainActivity extends Activity {  
2:  
3:     public String lifecycle;  
4:  
5:     @Override
```

```
6:     protected void onCreate(Bundle savedInstanceState) {
7:         super.onCreate(savedInstanceState);
8:         setContentView(R.layout.activity_main);
9:         lifecycle = "onCreate";
10:    }
11:
12:    @Override
13:    protected void onStart() {
14:        super.onStart();
15:        lifecycle = "onStart";
16:    }
17:
18:    @Override
19:    protected void onResume() {
20:        super.onResume();
21:        lifecycle = "onResume";
22:    }
23:
24:    ... 略...
25: }
```

4.4.2 手順 1. ActivityTestCase のサブクラスを作成する

ActivityTestCase を継承したクラス「MainActivityLifecycleTest」クラスを作成します。

1. src > "com.example.hellojunit.test" パッケージを選択して右クリックメニュー > 「New」> 「JUnit Test Case」を選びます
2. 各値を以下のように設定し「Finish」ボタンをクリック

表 4.1 MainActivityLifecycleTest

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	HelloJUnitTest/src
Pakage	com.example.hellojunit.test
Name	MainActivityLifecycleTest
Superclass	android.test.ActivityUnitTestCase<MainActivity>
setUp	チェック
tearDown	チェック
Class under test	com.example.hellojunit.MainActivity

4.4.3 手順 2. 初期化(コンストラクタ、setUp、tearDown など)

コンストラクタで親のコンストラクタを呼び出しテスト対象のクラス情報を与え、setUp メソッドでメンバ変数の初期化を行います。

1. メンバ変数の追加

テスト対象のクラス「MainActivity」をメンバ変数として保持する

リスト 4.7: メンバ変数の追加

```
1: public class MainActivityLifecycleTest extends ActivityUnitTestCase<MainActivity> {  
2:  
3:     private MainActivity activity;
```

2. コンストラクタの追加

コンストラクタで親のコンストラクタを呼び出し、引数にテスト対象のクラス情報を与える

リスト 4.7: コンストラクタの追加

```
1:     public MainActivityLifecycleTest() {  
2:         super(MainActivity.class);  
3:     }
```

3. setUp メソッドの修正

メンバ変数 activity の初期化を行う

ActivityUnitTestCase では対象 Activity をモックオブジェクト^{*1}として扱います。

startActivity メソッドを使うとテスト対象の Activity のインスタンスを取得することができます。

```
protected T startActivity (Intent intent,  
                           Bundle savedInstanceState,  
                           Object lastNonConfigurationInstance)
```

- startActivity メソッドは、onCreate だけ実行される
- Activity のインスタンスは、getActivity メソッドで、取得できる。
- getActivity メソッドは、startActivity を実行していないと null が返ってくる

リスト 4.7: setUp メソッドの修正

```
1:     protected void setUp() throws Exception {  
2:         super.setUp();  
3:         Intent intent = new Intent();
```

^{*1} モックオブジェクト： テスト対象のオブジェクトを意図したとおりに動かすための検証用オブジェクト

```
4:         activity = startActivity(intent, null, null);
5:     }
```

リスト 4.7: 手順 2 の実装コード

```
1: package com.example.hellojunit.test;
2:
3: import com.example.hellojunit.MainActivity;
4:
5: import android.content.Intent;
6: import android.test.ActivityUnitTestCase;
7:
8: public class MainActivityLifecycleTest extends ActivityUnitTestCase<MainActivity> {
9:
10:     private MainActivity activity;
11:
12:     public MainActivityLifecycleTest() {
13:         super(MainActivity.class);
14:     }
15:
16:     protected void setUp() throws Exception {
17:         super.setUp();
18:         Intent intent = new Intent();
19:         activity = startActivity(intent, null, null);
20:     }
21:
22:     protected void tearDown() throws Exception {
23:         super.tearDown();
24:     }
25: }
```

4.4.4 手順 3. テストメソッドを作成する

取得したモックオブジェクトを使ってライフサイクルのテストを行います

- ライフサイクルの呼び出しを確認するためのテストメソッド「testLifecycle」を作成します
- ライフサイクルを実行するには getInstrumentation().callActivityOnXXX メソッドの引数に Activity を指定します
 - イベントに応じた callActivityOnXXX が提供されています
 - 例)
 - * getInstrumentation().callActivityOnStart(activity);
 - * getInstrumentation().callActivityOnResume(activity);
- 各ライフサイクルのタイミングで、Activity のメンバ変数 lifecycle の値が正しく変化しているかを確認する

リスト 4.7: testLifecycle メソッド

```
1: public void testLifecycle(){  
2:     assertEquals("onCreate", activity.lifecycle);  
3:     getInstrumentation().callActivityOnStart(activity);  
4:     assertEquals("onStart", activity.lifecycle);  
5:     getInstrumentation().callActivityOnResume(activity);  
6:     assertEquals("onResume", activity.lifecycle);  
7: }
```

4.4.5 MainActivityLifecycleTest.java

リスト 4.7: MainActivityLifecycleTest.java

```
1: package com.example.hellojunit.test;  
2:  
3: import com.example.hellojunit.MainActivity;  
4:  
5: import android.content.Intent;  
6: import android.test.ActivityUnitTestCase;  
7:  
8: public class MainActivityLifecycleTest extends ActivityUnitTestCase<MainActivity> {  
9:  
10:    private MainActivity activity;  
11:  
12:    public MainActivityLifecycleTest() {  
13:        super(MainActivity.class);  
14:  
15:    }  
16:  
17:    protected void setUp() throws Exception {  
18:        super.setUp();  
19:        Intent intent = new Intent();  
20:        activity = startActivity(intent, null, null);  
21:    }  
22:  
23:    public void testLifecycle() {  
24:        assertEquals("onCreate", activity.lifecycle);  
25:        getInstrumentation().callActivityOnStart(activity);  
26:        assertEquals("onStart", activity.lifecycle);  
27:        getInstrumentation().callActivityOnResume(activity);  
28:        assertEquals("onResume", activity.lifecycle);  
29:    }  
30:  
31: }
```

4.5 【実習】Activity Testing



4.5.1 実習

サンプルで説明したテストを作成する

4.5.2 確認

テストを実行し、バーが緑になっていることを確認する

4.5.3 解答

解答ドキュメントを参照

4.6 Service Testing

Service Testing

- Serviceの特徴
- ServiceTestCase
- Service Testing Sample
- 実習



4.6.1 Service の特徴

Serviceの特徴

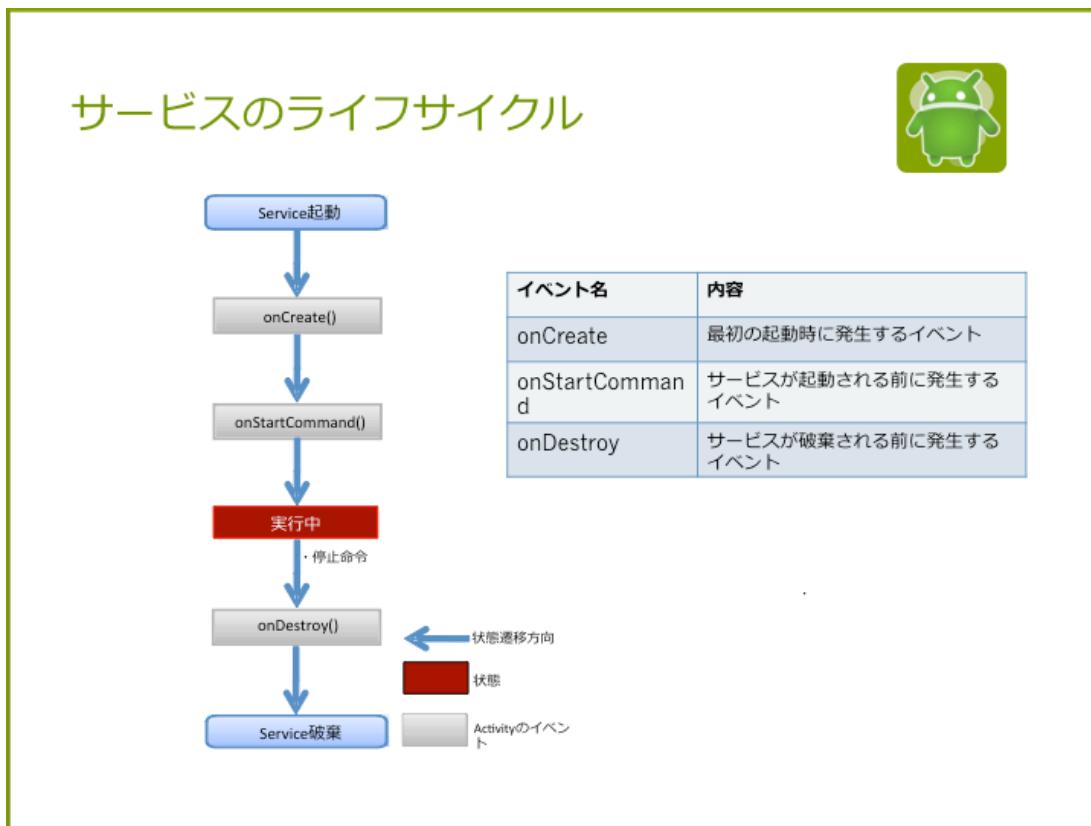
- UIを持たないコンポーネント
- ライフサイクルを持っている
- バックグラウンドで動作する



サービスは長時間バックグラウンドで動き続ける UI を持たない Android コンポーネントです。ユーザがアプリケーションを切り替えたり、終了してもサービスは起動し続けることができます。常にバックグラウンドで動き続けるため常駐プログラムとして使用することができます。

- サービスの例
 - 音楽再生
 - ダウンロード
 - タイマー、アラーム
 - ロケーション情報
 - センサー情報

4.6.2 サービスのライフサイクル



サービスにはオブジェクト生成～破棄に至るまでのライフサイクルが定義されています。

- ライフサイクル
 - onCreate
 - onStartCommand
 - onDestroy

4.6.3 ServiceTestCase

Service のテストでは ServiceTestCase を使用します。

- サービスのテスト用に拡張された TestCase クラスです。
- ServiceTestCase クラスのメソッドを使うことでサービスのモックオブジェクトを取得することができる

ServiceTestCase の機能

ServiceTestCase クラスが提供している以下のメソッドを使うことで、サービスの起動、停止イベントを呼び出すことが出来ます

- startService
 - サービスを起動する
 - onCreate、 onStartCommand が呼ばれる
- getService
 - startService で起動したサービスオブジェクトを取得する
- shutDownService
 - サービスを停止する
 - onDestroy が呼ばれる

4.7 Service Testing Sample

Service Testing Sample

- HelloJUnitアプリケーションにServiceのテストを作成する



- 対象プロジェクト

プロジェクト名	説明
HelloJUnit	テストターゲット
HelloJUnitTest	テストプロジェクト

4.7.1 アプリケーションの説明



画面のトグルボタンが ON になると、サービスが起動し、1秒毎にカウント UP します。現在のカウント値をログで出力します。

トグルボタンが OFF になると、サービスは破棄され、ログ出力が止まります。

HelloJUnit の HelloJunitService クラス

リスト 4.8: HelloJunitService.java

```

1: public class HelloJunitService extends Service {
2:
3:     private static final String TAG = "HelloJunitService";
4:     public AtomicBoolean running = new AtomicBoolean(false);
5:     LogThread thread = new LogThread();
6:
7:     @Override
8:     public IBinder onBind(Intent intent) {
9:         return null;
10:    }
11:
12:    @Override
13:    public int onStartCommand(Intent intent, int flags, int startId) {
14:        Log.v(TAG, "onStartCommand");

```

```
15:         Toast.makeText(this, "Service Start", Toast.LENGTH_SHORT).show();
16:         running.set(true);
17:         thread.start();
18:
19:         try {
20:             Thread.sleep(1000);
21:         } catch (InterruptedException e) {
22:             e.printStackTrace();
23:         }
24:         return START_STICKY;
25:     }
26:
27:     @Override
28:     public void onDestroy() {
29:         running.set(false);
30:         Toast.makeText(this, "Service end", Toast.LENGTH_SHORT).show();
31:     }
32:
33: /**
34: * 1秒毎にログを出力する Thread
35: *
36: */
37: class LogThread extends Thread {
38:     @Override
39:     public void run() {
40:         for (int i = 0; running.get(); i++) {
41:             Log.v(getClass().getSimpleName(), "i:" + i);
42:             try {
43:                 Thread.sleep(1000);
44:             } catch (InterruptedException e) {
45:                 Log.v(TAG, e.getMessage());
46:             }
47:         }
48:     }
49: }
50: }
```

4.7.2 テストケースの作成

HelloJUnitService クラスのテストを作成します。

HelloJUnitService クラスは変数 `running` で、サービス起動状態を管理しています。この変数の状態をチェックし、サービスの起動と停止の検証を行います。

4.7.3 ServiceTestCase の作成方法

手順

1. ServiceTestCase のサブクラスを作成する
2. コンストラクタを作成する
3. テストメソッドを作成する

4.7.4 手順 1. TestCase の作成

ServiceTestCase のサブクラスを作成する。

1. src > "com.example.hellojunit.test" パッケージを選択して右クリックメニュー > 「New」> 「JUnit Test Case」を選びます
2. 各値を以下のように設定し「Finish」ボタンをクリック

表 4.2 HelloJunitServieTest

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	HelloJUnitTest/src
Pakage	com.example.hellojunit.test
Name	HelloJunitServieTest
Superclass	android.test.ServiceTestCase<HelloJUnitService>
Class under test	com.example.hellojunit.HelloJUnitService

4.7.5 手順 2. コンストラクタの作成

1. HelloJUnitService の import
2. コンストラクタの作成

リスト 4.12: HelloJunitServieTest.java

```
1: public class HelloJunitServieTest extends ServiceTestCase<HelloJUnitService> {  
2:  
3:     public HelloJunitServieTest() {  
4:         super(HelloJUnitService.class);  
5:     }  
6:
```

4.7.6 手順 3. テストメソッドの作成

サービスの起動と停止を確認するための以下のテストメソッドを作成します

- testOnStartCommand
 - サービスの起動確認
- testOnDestroy
 - サービスの停止確認

testOnStartCommand

1. サービスを起動するための Intent を生成する
2. ServiceTestCase#startService メソッドを使って、テスト環境下でのモックサービスを実行する
3. getService メソッドを使って、テスト対象の Service のインスタンスを取得する
4. running の値が true になっているのを確認する

リスト 4.10: testOnStartCommand メソッド

```
1:     public void testOnStartCommand(){  
2:         Intent intent = new Intent(mContext, HelloJUnitService.class);  
3:         startService(intent);  
4:         assertNotNull(getService());  
5:  
6:         HelloJUnitService service = getService();  
7:         assertTrue("running フラグが true", service.running.get());  
8:     }
```

testOnDestroy

1. サービスを起動する処理の追加
2. ServiceTestCase#shutdownService() メソッドを使って、サービスを停止する
3. running の値が、false になっているのを確認する

リスト 4.11: testOnDestroy メソッド

```
1:     public void testOnDestroy(){  
2:         Intent intent = new Intent(mContext, HelloJUnitService.class); // 1  
3:         startService(intent);  
4:         HelloJUnitService service = getService();  
5:  
6:         shutdownService(); // 2  
7:         assertFalse("running フラグが false", service.running.get()); // 3  
8:     }
```

HelloJunitServieTest.java

リスト 4.12: HelloJunitServieTest.java

```
1: package com.example.hellojunit.test;  
2:  
3: import android.content.Intent;  
4: import android.test.ServiceTestCase;
```

```
5:
6: import com.example.hellojunit.HelloJUnitService;
7:
8:
9: public class HelloJunitServieTest extends ServiceTestCase<HelloJUnitService> {
10:
11:     public HelloJunitServieTest() {
12:         super(HelloJUnitService.class);
13:     }
14:
15:     public void testOnStartCommand(){
16:         Intent intent = new Intent(mContext, HelloJUnitService.class);
17:         startService(intent);
18:         assertNotNull(getService());
19:
20:         HelloJUnitService service = getService();
21:         assertTrue("running フラグが true", service.running.get());
22:     }
23:
24:     public void testOnDestroy(){
25:         Intent intent = new Intent(mContext, HelloJUnitService.class);
26:         startService(intent);
27:         HelloJUnitService service = getService();
28:
29:         shutdownService();
30:         assertFalse("running フラグが false", service.running.get());
31:     }
32:
33: }
```

4.7.7 動作確認

テストを実行し、バーが緑になっているのを確認する

4.8 【実習】Service Testing



4.8.1 実習

サンプルで説明したテストを作成する

4.8.2 確認

テストを実行し、バーが緑になっていることを確認する

4.8.3 解答

解答ドキュメントを参照

4.9 Database Testing

Database Testing

- Databaseテストの特徴
- 独立したテスト環境
- Database Testing Sample
- 実習



4.9.1 データベーステストの特徴

データベーステストの特徴

- Contextが必要
- テスト用の独立した環境を用意する
- テスト環境にアクセスする



テストの特徴

- データベースのテストでは、テスト専用の DB を準備する
- テスト対象のアプリが使用する DB の状態を、変更してはいけない
- 繰り返し実行出来るようにするために、実行前にはテストに適した状態を整え、テスト実行後に元の状態を復元する

テストケースクラスの特徴

- SQLiteOpenHelper を呼び出すため、TestCase で Context が必要になる
- TestCase で Context を使用する場合は InstrumentationTestCase または AndroidTestCase のサブクラスを作成する
- RenamingDelegatingContext を使ってテスト DB にアクセスする
- テストプロジェクトのリソースにアクセスする場合はテストプロジェクトのコンテキストが必要になる

3つの Context

Android では、データベースの操作に SQLiteOpenHelper を使います。SQLiteOpenHelper クラスは、インスタンスの生成に Context が必要です。そのため、TestCase で Context を取得する必要があります。

TestCase で Context を扱うためには、InstrumentationTestCase または AndroidTestCase などの TestCase を継承したクラスを作成します。

状況に応じて以下の 3 つ Context を使い分けます。

- ターゲットアプリケーションの Context
- RenamingDelegatingContext (後述)
- TestContext (後述)

以降、本トレーニングでは 3 つの Context を以下の用語を用いて表記する

表 4.3 3 つの Context

Context	ドキュメントでの表記	変数名
ターゲットアプリケーションの Context	ターゲットコンテキスト	targetContext, context
RenamingDelegatingContext (後述)	RenamingDelegatingContext, モックコンテキスト	mockContext
TestContext (後述)	TestContext, テストコンテキスト	testContext

4.9.2 独立したテスト環境

データベースのようなテストでは、テスト対象のアプリケーションのデータを操作してはいけません。そのためにテスト専用の DB を用意します。

また、データベースを扱ったテストは常に同じ結果でなくてはいけません。テストの再帰性や信頼性を保つため、テスト用のダミーデータの登録や、テスト専用 DB の準備などを行う必要があります。

- テスト環境に必要なもの
 - テスト用 DB
 - テストデータ
 - 検証用データ
 - テスト環境にアクセスするためのモックオブジェクト

RenamingDelegatingContext



RenamingDelegatingContext

- テスト専用のモックコンテキストを提供する
- テストDB操作用のContextを取得する
 - コンストラクタを使ってRenamingDelegatingContextのオブジェクトを取得する

コンストラクタ	説明
RenamingDelegatingContext (Context context, String filePrefix)	テスト専用のDBを作成するため のContext 引数 <ul style="list-style-type: none">• Context: ターゲットアプリの Contextを指定する• String: テスト用DBのprefix テスト用DBは prefix + "実際 のDBの名前"になる

RenamingDelegatingContext クラスをつかってテスト専用の環境を実現させます。テスト毎に DB の初期化を行うことができるため、データの状態を常に一定にすることができます。

RenamingDelegatingContext には以下のような特徴があります

- テスト用の独立した DB を使用するために用意する Context
- SQLiteOpenHelper のコンストラクタに、このクラスを引数に指定するとテスト DB が毎回 作成される。
- テスト用の DB はオリジナル DB を元に任意の prefix を追加したものが作成される
- コンストラクタを使ってテスト DB 操作用の Context を取得する

以下の例では test. + [DB の名前] でテスト DB が作成されます。(DB が MyDB の場合は test.MyDB という名前で作成されます。)

リスト 4.14: RenamingDelegatingContext

```
1:     private Context mockContext;
2:     private DBOpenHelper helper;
3:
```

```
4:     protected void setUp() throws Exception {
5:         mockContext = new RenamingDelegatingContext(mContext, "test.");
6:         helper = new DBOpenHelper(mockContext);
7:     }
```

BaseContext の取得

テストデータや、検証データなどを外部ファイルで用意する場合は、テストプロジェクトのリソースにアクセスする必要があります。テストプロジェクト固有のリソースにアクセスするために、テストアプリケーションの Context が必要です。

- TestContext^{*2}
 - テストアプリケーションのリソースにアクセスすることができる

^{*2} TestContext という名前のクラスは存在しませんが、本トレーニングではテストアプリケーション専用の Context のことを TestContext と表記する

4.10 Database Testing Sample

Database Testing Sample

- JUnitDatabaseSampleアプリケーションのデータベーステストを作成する
- アプリケーションには以下の機能がある
 1. 書籍データの登録をする
 2. 書籍データの一覧を表示する
- 対象プロジェクト



プロジェクト名	説明
JUnitDatabaseSample	テストターゲット
JUnitDatabaseSampleTest	テストプロジェクト

サンプルを使ってデータベーステストの作成方法を説明します。

4.10.1 アプリケーションの説明



書籍データを管理するアプリケーションです。データの登録と一覧表示に対応しています。

サンプルアプリケーションは解答ドキュメントで提供済です。

4.10.2 テスト内容

DB からデータを 1 件取得し、データが期待値と一致していることを確認します。

表 4.4 取得データ

_id	title	price
1	図解 Android プラットフォーム開発入門	3780

4.10.3 プロジェクト概要

プロジェクト構成

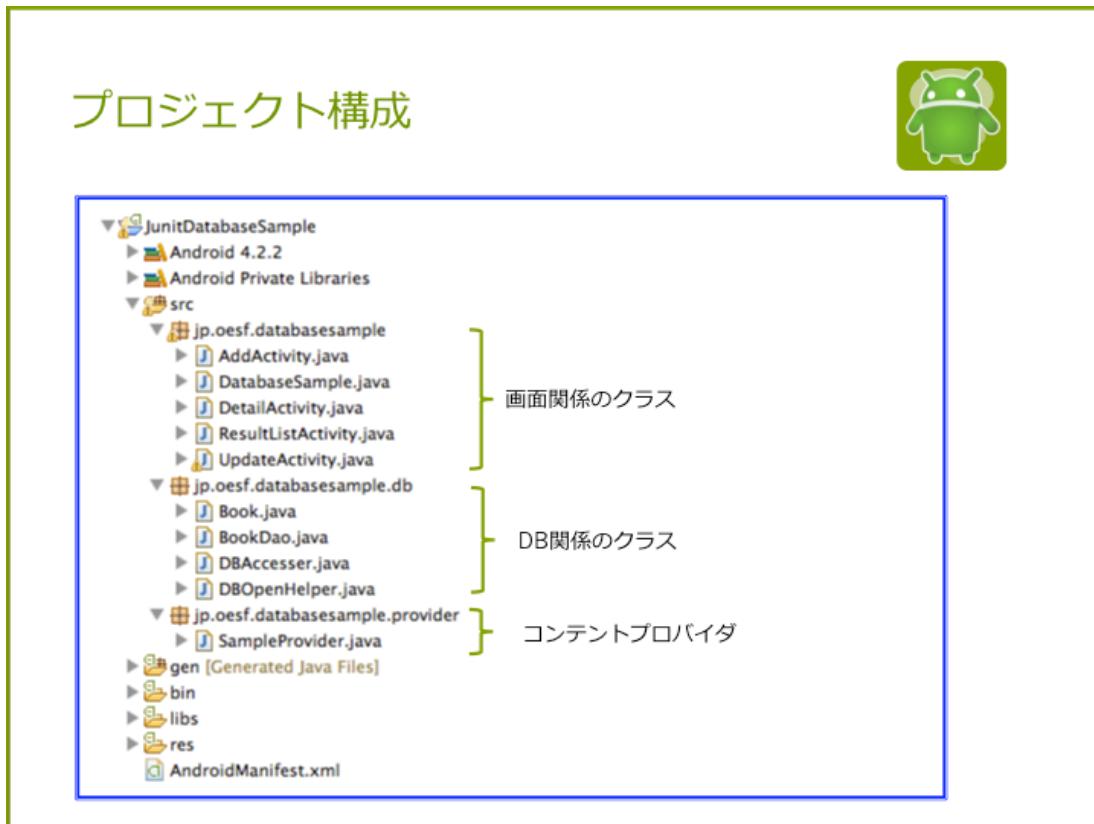


表 4.5 プロジェクト概要

項目	設定値
Project Name	JUnitDatabaseSample
Build Target	トレーニングで指定したバージョン
Package	jp.oesf.databasesample

データベース構成

- データベース名 : sample_db
- テーブル名 : book

テスト対象のクラス

- BookDao
 - book テーブル操作系クラス

表 4.6 book テーブル

カラム	型	その他
_id	INTEGER	主キー、auto increment
title	TEXT	Not Null
value	INTEGER	Not Null

表 4.7 主なメソッド

メソッド名	説明
get	_id を指定して、データを 1 件取得する
getAll	データを全件取得する
add	データを追加する
update	データを更新する
delete	データを削除する

サンプルでは get メソッドのテストを作成します。

4.10.4 テストプロジェクト概要

テストプロジェクト構成

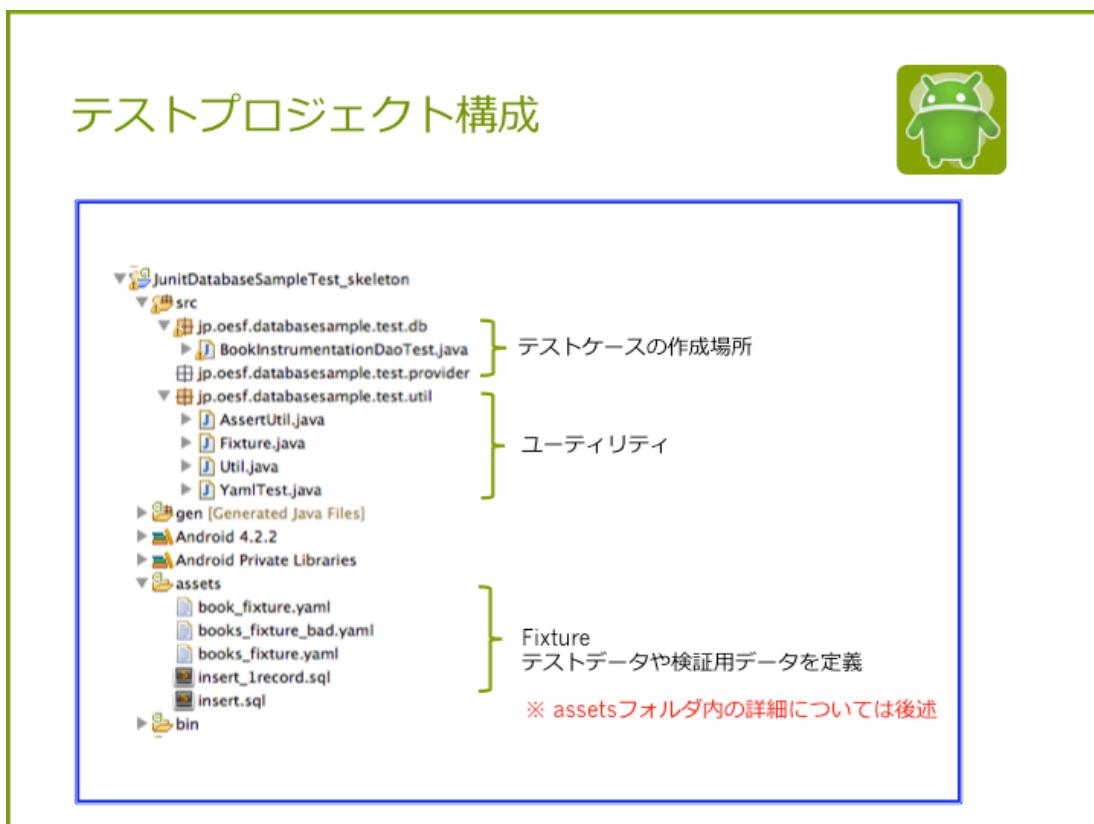


表 4.8 テストプロジェクト概要

項目	設定値
Project Name	JUnitDatabaseSampleTest
Test Target	JUnitDatabaseSample
Package	jp.oesf.databasesample.test

4.10.5 データ取得テストを作成する

手順

1. Fixture の準備
2. BookDaoTest クラスの作成
3. テストメソッドの作成

Fixture とは

- テストするために必要なデータや前提条件の集合のこと
- データベースのテストでは、繰り返しテストできるように、Fixture を使ってテスト毎にデータの初期化する
- Fixture の定義はコード内にインラインで記述する方法や、外部ファイルで定義するなどの手法がある
 - Android の場合は、テストプロジェクト以下にある assets やリソースファイルにアクセスすることができないため、工夫が必要になる
 - サンプルアプリケーションではインライン形式でテストデータを用意する

4.10.6 手順 1. Fixture の準備

テストデータの登録 sql を用意します。サンプルでは、よりシンプルな作りにするためインラインコードでデータの登録します。準備段階では直接クエリを発行しエラーが起きないことを確認します。

以下のクエリは「テストメソッドの作成」の手順で使用します。

```
INSERT INTO book ('_id', 'title', 'price') VALUES(1, '図解 Android プラットフォーム開発入門', 3780);
```

4.10.7 手順 2. BookDaoTest クラスの作成

データベースのテストでは Context が必要なので AndroidTestCase を継承したクラスを作成します。

1. AndroidTestCase クラスのサブクラスを作成する

表 4.9 BookDaoAndroidTest

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	JUnitDatabaseSampleTest_skeleton/src
Pakage	jp.oesf.databasesample.test.db
Name	BookDaoTest
Superclass	android.test.AndroidTestCase
setUp	チェック
tearDown	チェック
Class under test	jp.oesf.databasesample.db.BookDao

2. メンバ変数の定義

以下のメンバ変数を定義します。

- モックコンテキスト
- DBOpenHelper

3. setUp のオーバライド

setUp で以下の処理を実装します。

- メンバ変数の初期化
- テストデータベースの作成
 - ターゲットコンテキストから RenamingDelegatingContext を取得する
 - RenamingDelegatingContext を使ってとテスト専用の DB を作成する

4. tearDown のオーバライド

データベースをクローズします。

リスト 4.14: BookDaoTest の作成

```
1: package jp.oesf.databasesample.test.db;
2:
3: import jp.oesf.databasesample.db.DBOpenHelper;
4: import android.content.Context;
5: import android.test.AndroidTestCase;
6: import android.test.RenamingDelegatingContext;
7:
8: public class BookDaoTest extends AndroidTestCase {
9:
10:    private static final String TAG = "BookDaoAndroidTest";
11:    private Context mContext;
12:    private DBOpenHelper helper;
13:
14:    protected void setUp() throws Exception {
15:        mContext = new RenamingDelegatingContext(mContext, "test.");
16:        helper = new DBOpenHelper(mContext);
17:    }
18:
19:    protected void tearDown() throws Exception {
20:        helper.close();
21:    }
22:
23: }
```

4.10.8 手順 3. テストメソッドの作成

BookDao クラスの get メソッドのテストメソッドを作成します。

1. testGet メソッドを定義する
2. インライン Fixture で DB にテストデータを登録する
3. get メソッドを実行し、結果を取得する
4. 取得結果の確認をする

表 4.10 testGet 用登録データ

_id	title	price
1	図解 Android プラットフォーム開発入門	3780

リスト 4.15: testGet

```

1: public void testGet() { // 1. テストメソッドの定義
2:     String sql1 = "INSERT INTO book ('_id', 'title', 'price') VALUES(1, '図解 Android プラットフォーム開発入門', 3780);";
3:
4:     SQLiteDatabase database = helper.getWritableDatabase();
5:     database.execSQL(sql1); // 2. テストデータの登録
6:
7:     BookDao dao = new BookDao(helper.getReadableDatabase());
8:     Book actual = dao.get("1"); // 3. ターゲットメソッドの実行
9:
10:    Book expected = new Book("1", "図解 Android プラットフォーム開発入門", "3780");
11:    assertEquals("データチェック", expected, actual); // 4. 取得結果の確認
12: }
```

4.10.9 全体ソース

リスト 4.16: BookDaoTest

```

1: package jp.oesf.databasesample.test.db;
2:
3: import jp.oesf.databasesample.db.Book;
4: import jp.oesf.databasesample.db.BookDao;
5: import jp.oesf.databasesample.db.DBOpenHelper;
6: import android.content.Context;
7: import android.database.sqlite.SQLiteDatabase;
8: import android.test.AndroidTestCase;
9: import android.test.RenamingDelegatingContext;
10:
11: public class BookDaoTest extends AndroidTestCase {
12:
13:     private static final String TAG = "BookDaoAndroidTest";
14:     private Context mockContext;
15:     private DBOpenHelper helper;
16:
17:     protected void setUp() throws Exception {
```

```
18:         mContext = new RenamingDelegatingContext(mContext, "test.");
19:         helper = new DBOpenHelper(mContext);
20:     }
21:
22:     public void testGet() { // 1. テストメソッドの定義
23:         String sql1 = "INSERT INTO book ('_id', 'title', 'price') VALUES(1, '図解 Android プラットフォーム開発入門', 3780);";
24:
25:         SQLiteDatabase database = helper.getWritableDatabase();
26:         database.execSQL(sql1); // 2. テストデータの登録
27:
28:         BookDao dao = new BookDao(helper.getReadableDatabase());
29:         Book actual = dao.get("1"); // 3. ターゲットメソッドの実行
30:
31:         Book expected = new Book("1", "図解 Android プラットフォーム開発入門", "3780");
32:         assertEquals("データチェック", expected, actual); // 4. 取得結果の確認
33:     }
34:
35:     protected void tearDown() throws Exception {
36:         helper.close();
37:     }
38:
39: }
```

4.10.10 テストの実行

テストを実行し、バーが緑になっているのを確認します。

テストデータベースの確認

テストを実行するとテスト用のデータベースが作成されているのが確認できます。

- 作成場所
 - /data/data/[ターゲットアプリパッケージ]/databases/[prefix][データベース名]
 - * prefix: RenamingDelegatingContext のコンストラクタの第 2 引数で指定

サンプルでは以下の DB が作成されます

- /data/data/jp.oesf.databasesample/databases/test.sample_db

```
root@android:/ # cd /data/data/jp.oesf.databasesample
root@android:/data/data/jp.oesf.databasesample # cd databases
root@android:/data/data/jp.oesf.databasesample/databases # ls
sample_db
sample_db-journal
test.sample_db
test.sample_db-journal
```

4.11 【実習】Database Testing



4.11.1 実習

サンプルで説明したテストを作成する

4.11.2 確認

テストを実行し、バーが緑になっていることを確認する

4.11.3 解答

解答ドキュメントを参照

4.12 【補足実習】Database Testing



以下のテストデータを登録し、全件検索のテストを作成します。

表 4.11 test GetAll 用登録データ

_id	title	price
1	図解 Android プラットフォーム開発入門	3780
2	進撃の巨人 (10)	450
3	テラフォーマーズ (1)	540

手順

1. test GetAll メソッドを定義する
2. インライン Fixture で DB にテストデータを登録する
3. getAll メソッドを実行し、結果を取得する
4. 取得結果の件数の確認をする
5. 取得データの確認をする

補足実習では細かい手順の詳細は省略します。

4.12.1 確認

テストを実行し、バーが緑になっていることを確認する

4.12.2 解答

解答ドキュメントを参照

4.13 【補足実習2】テストコンテキストを使ったDatabase Testing

補足実習で作成した全件検索のテストと同じテストケースを作成します。Fixtureは外部ファイルで定義します。

4.13.1 テストコンテキストを取得する

テストコンテキストを使うことで、外部ファイルで定義したFixtureを参照することができます。テストコンテキストは、AndroidTestCaseクラスまたは、InstrumentationTestCaseを使って取得できます。

AndroidTestCase クラスを使った取得例

AndroidTestCaseクラスにはテストコンテキストを取得するためのメソッドgetTestContextメソッドが定義されていますが、非公開APIとなっているため、直接呼び出すことはできません。以下のような、強引な方法で取得します。

リスト 4.17: AndroidTestCase クラスを使った取得例

```
1: public class BookDaoAndroidTest extends AndroidTestCase {  
2:  
3:     private Context testContext;  
4:  
5:     @Override  
6:     protected void setUp() throws Exception {  
7:         Method method = getClass().getMethod("getTestContext");  
8:         testContext = (Context) method.invoke(this);  
9:     }  
}
```

スケルトンプロジェクトのUtilクラスに、上記テストコンテキストの取得メソッドが実装されています。

InstrumentationTestCase のクラスを使った取得例

Instrumentationクラスのを使うことで、テストコンテキストを取得できます。InstrumentationTestCaseクラスは、Instrumentationクラスのメソッドを呼び出すことができるため、テストコンテキストを取得することができます。

リスト 4.18: InstrumentationTestCase のクラスを使った取得例

```
1: public class BookInstrumentationDaoTest extends InstrumentationTestCase {  
2:  
3:     private Context testContext;  
}
```

```

4:
5:     @Override
6:     protected void setUp() throws Exception {
7:         testContext = getInstrumentation().getContext();
8:     }
9:

```

4.13.2 テストケースの作成

未実装の処理を実装し、プログラムを完成させます。

修正対象：BookInstrumentationDaoTest

BookInstrumentationDaoTest クラスはスケルトンプロジェクトで提供済
手順

1. Fixture の作成 <実装済>
2. Fixture 制御クラスの作成 <実装済>
3. TestCase の作成 <一部未実装>
4. テストメソッドの作成

4.13.3 手順 1. Fixture の作成 <実装済み>

テストデータ定義ファイルは assets ディレクトリ以下に保存しています

今回のテストでは、「insert.sql」、「books_fixture.yaml」を使用します。

表 4.12 MainActivityTest

ファイル名	説明
insert.sql	テスト用の登録データクエリが記述されたテキスト 3件のテストデータを登録する
insert_1.sql	テスト用の登録データクエリが記述されたテキスト 1件のテストデータを登録する
books_fixture.yaml	DB から全件取得用の確認データ
book_fixture.yaml	DB から 1 件取得用の確認データ
books_fixture_bad.yaml	DB 全件取得用の失敗用確認データ

4.13.4 手順 2. Fixture ユーティリティの作成 <実装済み>

Fixture.java は外部ファイルで用意した Fixture を操作するためのユーティリティクラスです。

テストケースに合わせた Fixture を制御するメソッドを用意しています。

insertBooks(Context context, Context testContext, String filename):

第3引数に指定したファイルに記述されているクエリを発行しテストデータをDBに登録する

List<Book> getBooks(Context testContext, String filename)

第2引数で指定したYAMLファイルに定義してあるデータセットをList<Book>に変換する

Book getBook(Context testContext, String filename):

第2引数で指定したYAMLファイルに定義してあるデータセットをBookに変換する

リスト 4.19: Fixture.java

```
1: package jp.oesf.databasesample.test.util;
2:
3: import java.io.IOException;
4: import java.io.InputStream;
5: import java.util.ArrayList;
6: import java.util.List;
7:
8: import org.yaml.snakeyaml.Yaml;
9:
10: import jp.oesf.databasesample.db.Book;
11: import jp.oesf.databasesample.db.DBOpenHelper;
12: import android.content.Context;
13: import android.database.sqlite.SQLiteDatabase;
14: import android.util.Log;
15:
16: public class Fixture {
17:
18:     private static final String TAG = "Fixture";
19:     public static final String INSERT_SQL = "insert.sql";
20:     public static final String INSERT_SQL_1 = "insert_1.sql";
21:     public static final String BOOK_FIXTURE = "book_fixture.yaml";
22:     public static final String BOOKS_FIXTURE = "books_fixture.yaml";
23:     public static final String BOOK_FIXTURE_BAD = "book_fixture_bad.yaml";
24:
25:
26:     public static int insertBooks(Context context, Context testContext, String filename){
27:         String sql = Util.getAssetsData(testContext, filename);
28:
29:         DBOpenHelper helper = new DBOpenHelper(context);
30:         String[] ar = sql.split(";");
31:         SQLiteDatabase database = helper.getWritableDatabase();
32:         for(String s:ar){
33:             Log.v(TAG, s);
34:             database.execSQL(s);
35:         }
36:         helper.close();
37:         return ar.length;
38:     }
39:
40:     public static List<Book> getBooks(Context testContext, String filename){
41:         Yaml yaml = new Yaml();
42:
43:         try {
44:             InputStream in = testContext.getAssets().open(filename);
```

```

45:         @SuppressWarnings("unchecked")
46:         ArrayList<Book> list = (ArrayList<Book>) yaml.load(in);
47:         for(Book book:list){
48:             Log.v(TAG, "load :" + book.toString());
49:         }
50:
51:         return list;
52:     } catch (IOException e) {
53:         Log.e(TAG, e.getMessage(), e);
54:         return null;
55:     }
56:
57: }
58:
59: public static Book getBook(Context testContext, String filename){
60:     Book item = null;
61:     try {
62:         InputStream in = testContext.getAssets().open(filename);
63:         item = (Book) new Yaml().load(in);
64:         Log.v(TAG, item.toString());
65:         return item;
66:     } catch (IOException e) {
67:         Log.e(TAG, e.getMessage(), e);
68:         return null;
69:     }
70: }
71: }
```

4.13.5 手順3. TestCase の作成 <一部未実装>

データベースのテストでは Context が必要なので InstrumentationTestCase のサブクラスを作成します。

1. InstrumentationTestCase クラスのサブクラスを作成する

表 4.13 BookDaoInstrumentationTestCase クラスの作成

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	JUnitDatabaseSampleTest_skeleton/src
Pakage	jp.oesf.databasesample.test.db
Name	BookDaoInstrumentationTestCase
Superclass	android.test.InstrumentationTestCase
setUp	チェック
tearDown	チェック
Class under test	jp.oesf.databasesample.db.BookDao

2. ソースコードの修正

1. メンバ変数の定義
2. setUp メソッドの実装
3. tearDown メソッドの実装

手順2 までの BookInstrumentationDaoTest クラス

リスト 4.20: BookInstrumentationDaoTest.java

```
1: package jp.oesf.databasesample.test.db;
2:
3: import java.util.ArrayList;
4:
5: import jp.oesf.databasesample.db.Book;
6: import jp.oesf.databasesample.db.BookDao;
7: import jp.oesf.databasesample.db.DBOpenHelper;
8: import jp.oesf.databasesample.test.util.Fixture;
9: import jp.oesf.databasesample.test.util.Util;
10: import android.content.Context;
11: import android.database.sqlite.SQLiteDatabase;
12: import android.test.AndroidTestCase;
13: import android.test.RenamingDelegatingContext;
14: import android.test.suitebuilder.annotation.SmallTest;
15: import android.util.Log;
16:
17: public class BookInstrumentationDaoTest extends InstrumentationTestCase {
18:
19:     private static final String TAG = "BookInstrumentationDaoTest";
20:     private Context context;
21:     private Context mockContext;
22:     private Context testContext;
23:     private DBOpenHelper helper;
24:
25:     @Override
26:     protected void setUp() throws Exception {
27:         context = getInstrumentation().getTargetContext();
28:         // TODO TestContext を取得する
29:         testContext = getInstrumentation().getContext();
30:         mockContext = new RenamingDelegatingContext(context, "test.");
31:     }
32:
33:     @Override
34:     protected void tearDown() throws Exception {
35:         helper.close();
36:     }
37: }
```

4.13.6 手順4. テストメソッドの作成

test GetAll メソッドの作成

1. test GetAll メソッドを定義する
2. 外部ファイルで定義した Fixture で DB にテストデータを登録する
3. ターゲットメソッドを実行し、結果を取得する
4. 取得結果の確認をする

表 4.14 test GetAll 用登録データ

_id	title	price
1	図解 Android プラットフォーム開発入門	3780
2	進撃の巨人 (10)	450
3	テラフォーマーズ (1)	540

リスト 4.22: test GetAll

```

1:     public void test GetAll() {
2:         String filename = "insert.sql";
3:         String filenameExpected = "books_fixture.yaml";
4:
5:         ArrayList<Book> actual = null;
6:         ArrayList<Book> expected = null;
7:
8:         // TODO テストデータを取得
9:         Fixture.insertBooks(mockContext, testContext, filename);
10:
11:        // TODO getAll メソッドのテスト
12:        BookDao dao = new BookDao(helper.getReadableDatabase());
13:        actual = dao.getAll();
14:
15:        // TODO 検証データを取得
16:        expected = (ArrayList<Book>) Fixture.getBooks(testContext, filenameExpected);
17:        helper.close();
18:
19:        // Assertion
20:        assertEquals("件数チェック", expected.size(), actual.size());
21:        for (int i = 0; i < expected.size(); i++) {
22:            Log.v(TAG, "expected:" + expected.get(i) + " actual:" + actual.get(i));
23:            assertEquals("", i + "件目のデータ - タチエツク", expected.get(i), actual.get(i));
24:        }
25:    }

```

BookInstrumentationDaoTest.java

リスト 4.22: test GetAll

```
1: package jp.oesf.databasesample.test.db;
2:
3: import java.util.ArrayList;
4:
5: import jp.oesf.databasesample.db.Book;
6: import jp.oesf.databasesample.db.BookDao;
7: import jp.oesf.databasesample.db.DBOpenHelper;
8: import jp.oesf.databasesample.test.util.Fixture;
9: import android.content.Context;
10: import android.test.InstrumentationTestCase;
11: import android.test.RenamingDelegatingContext;
12: import android.util.Log;
13:
14: public class BookInstrumentationDaoTest extends InstrumentationTestCase {
15:
16:     private static final String TAG = "BookInstrumentationDaoTest";
17:     private Context context;
18:     private Context testContext;
19:     private Context mockContext;
20:     private DBOpenHelper helper;
21:
22:     @Override
23:     protected void setUp() throws Exception {
24:         context = getInstrumentation().getTargetContext();
25:         // TODO TestContext を取得する
26:         testContext = getInstrumentation().getContext();
27:         mockContext = new RenamingDelegatingContext(context, "test.");
28:         helper = new DBOpenHelper(mockContext);
29:     }
30:
31:     public void testGetAll() {
32:         String filename = "insert.sql";
33:         String filenameExpected = "books_fixture.yaml";
34:
35:         ArrayList<Book> actual = null;
36:         ArrayList<Book> expected = null;
37:
38:         // TODO テストデータを取得
39:         Fixture.insertBooks(mockContext, testContext, filename);
40:
41:         // TODO getAll メソッドのテスト
42:         BookDao dao = new BookDao(helper.getReadableDatabase());
43:         actual = dao.getAll();
44:
45:         // TODO 検証データを取得
46:         expected = (ArrayList<Book>) Fixture.getBooks(testContext, filenameExpected);
47:         helper.close();
48:
49:         // Assertion
50:         assertEquals("件数チェック", expected.size(), actual.size());
51:         for (int i = 0; i < expected.size(); i++) {
52:             Log.v(TAG, "expected:" + expected.get(i) + " actual:" + actual.get(i));
53:             assertEquals("", i + "件 の データチェック", expected.get(i), actual.get(i));
54:         }
55:     }
56:
```

```
57: }
```

4.13.7 確認

テストを実行し、バーが緑になっていることを確認する

4.13.8 解答

解答ドキュメントを参照

4.14 Provider Testing

Provider Testing

- Providerテストの特徴
- Provider Testing Sample
- 実習



ContentProvider もデータベース同様にアプリケーションのデータにアクセスします。そのためテストするときは専用の環境が必要になります。ContentProvider のテストではどのような方法で独自のテスト環境作成するのか、ContentProvider の呼び出しはどのように行われるのかをこの章で学びます

4.14.1 Provider テストの特徴

Providerテストの特徴

- ProviderTestCase2を使用する
- テスト用のDBが必要
- MockContentResolverクラスを使用する



ContentProvider のテストでは以下のクラスを使用して独立した環境とテスト用の DB を提供します。

- ProviderTestCase2
- MockContentResolver

4.14.2 ProviderTestCase2

ProviderTestCase2 クラスを使って ContentProvider のテストをすることができます。

- ContentProvider のテスト用に拡張された TestCase クラス
- ProviderTestCase2 クラスを使うことでテスト用に独立した環境でデータの操作が可能
 - 前回試験したデータによる影響は受けない
 - 実データの読み取りや変更操作を許可しない
- コンストラクタの引数にターゲットアプリのクラス情報と Authority を指定する
- MockContentResolver クラスを使用することによって、テスト用の DB を作成することができる

- 専用 DB は test. が prefix で追加される

4.14.3 MockContentResolver

MockContentResolver クラスを使用することによって、テスト用の DB を作成することができます。

MockContentResolver の特徴

- MockContentResolver は Provider フレームワークのモック化を実現している
- このクラスを使ってデータベースを作成すると、test.[データベース名] という名前のテスト用データベースが作成される
- ProviderTestCase2#getMockContentResolver メソッドで MockContentResolver オブジェクトを取得することができる

4.15 Provider Testing Sample

Provider Testing Sample

- JUnitDatabaseSampleアプリケーションのコンテンツプロバイダのテストを作成する
- コンテンツプロバイダには以下の機能がある
 - 書籍データを公開している（検索のみ可）
- データのアクセスは、DBResolverSampleアプリケーションを使う
- 対象プロジェクト



プロジェクト名	説明
JUnitDatabaseSample	テストターゲット
JUnitDatabaseSampleTest	テストプロジェクト
DBResolverSample	データアクセスのためのサンプルプロジェクト

4.15.1 アプリケーションの説明

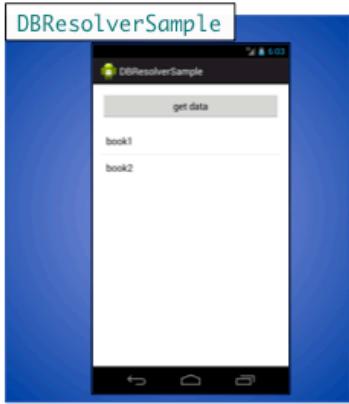
アプリケーションの説明



- 2つのアプリケーションを使って説明
 - JUnitDatabaseSample
 - ContentnProviderを公開
 - DBResolverSample (※ テストでは使いません)
 - ContentnProviderを使ってデータを取得する



JUnitDatabaseSample



DBResolverSample

データを公開する側と、公開データにアクセスする側の2つのサンプルアプリケーションを使って説明します。

サンプルアプリケーションは解答ドキュメントより提供済です。

- JUnitDatabaseSample
 - ContentProvider を公開している
 - 独自の ContentProvider を提供し管理しているデータのアクセスを許可する
- DBResolverSample (テストでは使いません)
 - ContentProvider を使ってデータを取得する

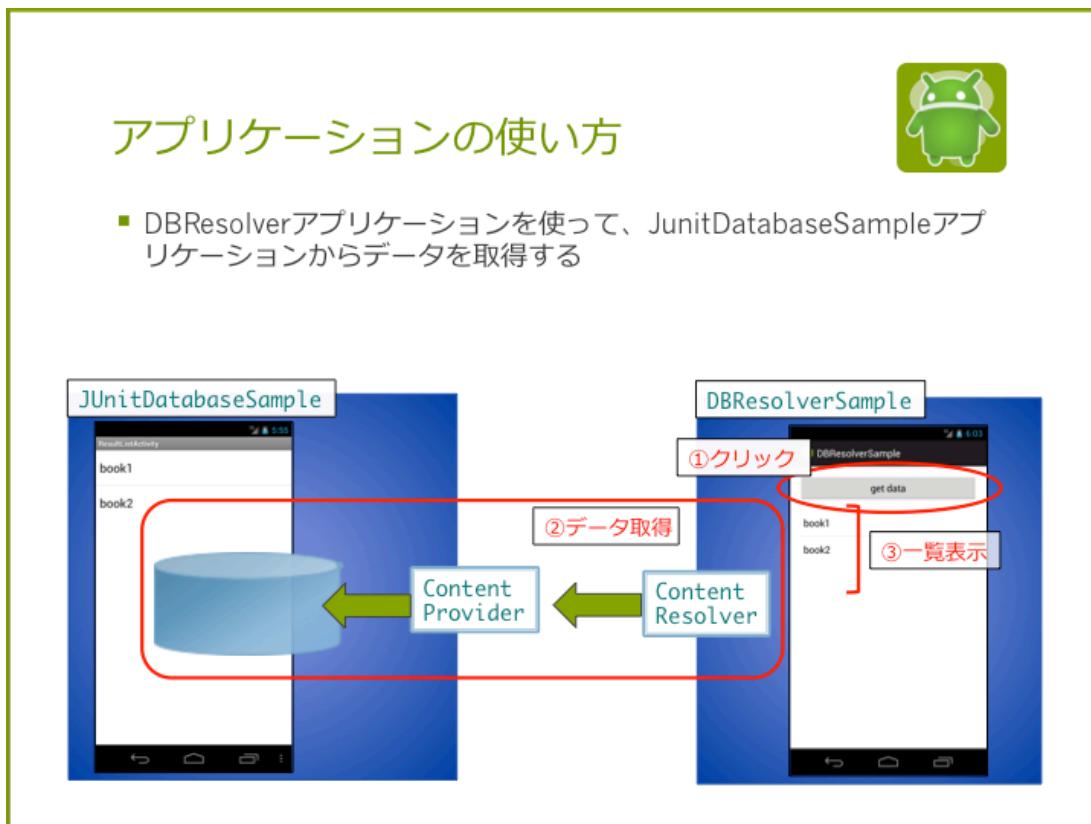
DatabaseTesting で使用した JUnitDatabaseSample アプリケーションの ContentProvider 機能を使ってテストプログラムを作成します。

ContentProvider のテストは2つのサンプルアプリケーションを使って説明します。

- JunitDatabaseSample
 - ContentProvider 提供側
 - Database Testing で使用したアプリケーションと同じものを使用する
- DBResolverSample

- 提供された ContentProvider を使ってデータを取得する（このアプリケーションは ContentProvider のアプリ使用例です。解説用に使っているだけで、実際の UnitTest では使用しません。）

4.15.2 アプリケーションの使い方



事前準備

1. 端末、エミュレータに JUnitDatabaseSample をインストールする
2. JUnitDatabaseSample アプリケーションを起動し、データの登録をする

データ取得手順

1. DBResolverSample を起動し、「get data」ボタンをクリックする
2. アプリケーション内で ContentResolver 経由で JUnitDatabaseSample のデータを取得する
3. 画面に取得したデータが表示される

4.15.3 ContentProvider のテスト方法

JUnitDatabaseSample アプリケーションに定義してある "jp.oesf.databasesample.provider.SampleProvider" のテストを作成します。今回はテストデータが 1 件登録された環境を作成し、全件検索を行います。Fixture はインラインで定義します。

手順

1. テストケースの作成
2. コンストラクタの作成
3. setUp メソッドのオーバーライド
4. テストメソッドの作成

4.15.4 手順 1. テストケースの作成

テストケースはデータベースのテストで作成した JUnitDatabaseSampleTest プロジェクトを使います。親クラスには、"ProviderTestCase2" を指定します。

1. ProviderTestCase2 のサブクラスを作成する

表 4.15 SampleProviderTest

項目	設定値
JUnit Version	New JUnit 3 test
Source folder	JunitDatabaseSampleTest_skeleton/src
Pakage	jp.oesf.databasesample.test.provider
Name	SampleProviderTest
Superclass	android.test.ProviderTestCase2
setUp	チェック
tearDown	オフ
Class under test	jp.oesf.databasesample.provider.SampleProvider

2. 定数とメンバ変数を定義する

以下の定数と変数を用意します。

- AUTHORITY
- AUTHORITY_URI
- mockContext

リスト 4.23: SampleProviderTest クラスの作成

```

1: public class SampleProviderTest extends ProviderTestCase2<SampleProvider> {
2:
3:     private static final String AUTHORITY = "jp.oesf.databasesample.test.provider";
4:     private static final Uri AUTHORITY_URI = Uri.parse("content://" + AUTHORITY);
5:
6:     private Context mockContext;
7:

```

4.15.5 手順 2. コンストラクタの作成

コンストラクタを作成します。コンストラクタでは親のコンストラクタを呼び出し、第2に引数にテスト用の Authority を指定する。この処理で、テスト用のデータベースにアクセスする ContentResolver と Context のモックオブジェクトを取得することができます。

- コンストラクタでは親のコンストラクタを呼び出す。
- 親のコンストラクタ引数に、ターゲットアプリのクラス情報とテスト用の Authority を引数に指定する

リスト 4.24: コンストラクタ

```
1:     public SampleProviderTest() {  
2:         super(SampleProvider.class, AUTHORITY);  
3:     }
```

4.15.6 手順 3. setUp メソッドのオーバライド

setUp メソッドをオーバライドします。ProviderTestCase2 では、モックオブジェクトを setUp メソッドで作成しているため、メソッド内では必ず super.setUp メソッドを呼び出す必要があります。そのため Context を必要とする処理は、setUp メソッドに記述します。ここでは、テスト Context の取得処理を以降で行っています。

- setUp をオーバライドする
- setUp メソッドで super.setUp メソッドを呼び出す
- テスト Context を取得する

リスト 4.25: setUp メソッド

```
1:     @Override  
2:     protected void setUp() throws Exception {  
3:         super.setUp();  
4:         mockContext = getMockContext();  
5:     }
```

4.15.7 手順 4. テストメソッドの作成

ContentProvider の query のテストを作成します。

1. testQuery メソッドを定義する
2. ContentResolver のモックオブジェクトを取得する

3. インライン Fixture で DB にテストデータを登録する
4. query メソッドを実行し、結果を取得する
5. 取得結果の件数の確認をする
6. 取得結果のデータの確認をする（AssertUtil の assertSameBook メソッドを使います）

表 4.16 テストデータ

_id	title	price
1	図解 Android プラットフォーム開発入門	3780

リスト 4.26: testQuery メソッド

```

1:     public void testQuery() { // 1. testQuery メソッドを定義する
2:
3:         // 2. ContentResolver のモックオブジェクトを取得する
4:         ContentResolver resolver = getMockContentResolver();
5:
6:         // 3. インライン Fixture で DB にテストデータを登録する
7:         String sql1 = "INSERT INTO book ('_id', 'title', 'price') VALUES(1, '図解
Android プラットフォーム開発入門', 3780);";
8:         DBOpenHelper helper = new DBOpenHelper(mContext);
9:         SQLiteDatabase database = helper.getWritableDatabase();
10:        database.execSQL(sql1);
11:        helper.close();
12:
13:        // 4. query メソッドを実行し、結果を取得する
14:        Cursor cursor = resolver.query(AUTHORITY_URI, null, null, null, null);
15:
16:        // 5. 取得結果の件数の確認をする
17:        assertEquals("件数チェック", 1, cursor.getCount());
18:
19:        // 6. 取得結果のデータの確認をする（AssertUtil の assertSameBook メソッドを使いま
す）
20:        Book expected = new Book("1", "図解 Android プラットフォーム開発入門", "3780");
21:        int index = cursor.getColumnIndex(BookDao.COLUMN_ID);
22:        assertEquals(-1, index);
23:        AssertUtil.assertSameBook(expected, cursor);
24:        cursor.close();
25:    }

```

リスト 4.27: SampleProviderTest.java

```

1: package jp.oesf.databasesample.test.provider;
2:
3: import jp.oesf.databasesample.db.Book;
4: import jp.oesf.databasesample.db.BookDao;
5: import jp.oesf.databasesample.db.DBOpenHelper;
6: import jp.oesf.databasesample.provider.SampleProvider;
7: import jp.oesf.databasesample.test.util.AssertUtil;

```

```
8: import android.content.ContentResolver;
9: import android.content.Context;
10: import android.database.Cursor;
11: import android.database.sqlite.SQLiteDatabase;
12: import android.net.Uri;
13: import android.test.ProviderTestCase2;
14:
15: public class SampleProviderTest extends ProviderTestCase2<SampleProvider> {
16:
17:     private static final String AUTHORITY = "jp.oesf.databasesample.test.provider";
18:     private static final Uri AUTHORITY_URI = Uri.parse("content://" + AUTHORITY);
19:
20:     private Context mockContext;
21:
22:     public SampleProviderTest() {
23:         super(SampleProvider.class, AUTHORITY);
24:     }
25:
26:     @Override
27:     protected void setUp() throws Exception {
28:         super.setUp();
29:         mockContext = getMockContext();
30:     }
31:
32:     public void testQuery() { // 1. testQuery メソッドを定義する
33:
34:         // 2. ContentResolver のモックオブジェクトを取得する
35:         ContentResolver resolver = getMockContentResolver();
36:
37:         // 3. インライン Fixture で DB にテストデータを登録する
38:         String sql1 = "INSERT INTO book ('_id', 'title', 'price') VALUES(1, '図解 Android プラットフォーム開発入門', 3780);";
39:         DBOpenHelper helper = new DBOpenHelper(mockContext);
40:         SQLiteDatabase database = helper.getWritableDatabase();
41:         database.execSQL(sql1);
42:         helper.close();
43:
44:         // 4. query メソッドを実行し、結果を取得する
45:         Cursor cursor = resolver.query(AUTHORITY_URI, null, null, null, null);
46:
47:         // 5. 取得結果の件数の確認をする
48:         assertEquals("件数チェック", 1, cursor.getCount());
49:
50:         // 6. 取得結果のデータの確認をする ( AssertUtil の assertSameBook メソッドを使います )
51:         Book expected = new Book("1", "図解 Android プラットフォーム開発入門", "3780");
52:         int index = cursor.getColumnIndex(BookDao.COLUMN_ID);
53:         assertEquals(-1, index);
54:         AssertUtil.assertSameBook(expected, cursor);
55:         cursor.close();
56:     }
57: }
```

4.15.8 テストの実行

テストを実行し、バーが緑になっているのを確認します。

4.16 【実習】Provider Testing



4.16.1 実習

サンプルで説明したテストを作成する

4.16.2 確認

テストを実行し、バーが緑になっていることを確認する

4.16.3 解答

解答ドキュメントを参照

4.17 【補足資料】テストコンテキストを使った Provider Testing

実習で作成したテストと同じテストケースを作成します。

Fixture は外部ファイルで定義します。

4.17.1 テストコンテキストを取得する

ProviderTestCase2 クラスは、そのままではテストコンテキストを取得できません。そのため、Util クラスで提供している getTestContext メソッドを使用します。

リスト 4.28: Util クラスの getTestContext メソッド

```
1:     @SuppressWarnings("rawtypes")
2:     public static Context getTestContext(Class clz, Context context, Object instance) {
3:         Context testContext = null;
4:         try {
5:             @SuppressWarnings("unchecked")
6:             Method method = clz.getMethod("getTestContext");
7:             testContext = (Context) method.invoke(instance);
8:         } catch (Exception e) {
9:             Log.e(TAG, e.getMessage(), e);
10:        }
11:        return testContext;
12:    }
```

4.17.2 テストメソッドの作成

手順

1. testQueryUseFile メソッドの作成
2. テストコンテキストの取得
3. 実行結果の確認

リスト 4.29: testQueryUseFile メソッド

```
1:     public void testQueryUseFile() {
2:         String filename = "insert_1record.sql";
3:         String filenameExpected = "book_fixture.yaml";
4:         Context testContext = Util.getTestContext(getClass(), getContext(), this);
5:
6:
7:         ContentResolver resolver = getMockContentResolver();
8:         int cnt = Fixture.insertBooks(getContext(), testContext, filename);
9:
10:        //query メソッドの実行
11:        Cursor cursor = resolver.query(AUTHORITY_URI, null, null, null, null);
12:    }
```

```
13:     assertEquals("件数チェック", cnt, cursor.getCount());
14:     Book expected = Fixture.getBook(testContext, filenameExpected);
15:     AssertUtil.assertSameBook(expected, cursor);
16:     cursor.close();
17: }
```

第5章

JUnit UI Testing



5.1 概要

概要

- UIテストの特徴
- UIを操作できるAPI
- UI Testing Sample
- 実習



5.2 UI テストの特徴

UIテストの特徴

- UIの操作はUIスレッドからしか出来ない
- JUnitのテストケースからUIを操作するためのAPIが提供されている
- ActivityInstrumentationTestCase2クラスを使用する



Android では UI の操作は、UI スレッド^{*1}でしか実行できないという制限があります。アプリケーションで UI 操作が正しく動作したかをテストするには、UI 操作を UI スレッドに依頼するという処理が必要になります。ActivityInstrumentationTestCase2 クラスは UI スレッドで動作するための API を提供しています。

5.2.1 UI を操作する

JUnit のテストケースから UI を操作する方法として以下の手段が用意されています。

表 5.1 UI 操作が可能なクラス

API など	説明
sendKey	キーイベントをシミュレートする
TouchUtils	タッチイベントをシミュレートするユーティリティ
runTestOnUiThread	run メソッドに記述した処理を UI をスレッドで実行する
@UiThreadTest	テストメソッドを UI をスレッドで実行する
ViewAsserts	ビューをテストするためのアサーションクラス

^{*1} Android ではフォアグラウンドで動いている Thread をメインスレッドまたは、UI スレッドと呼びます。

ボタンクリックの例

TouchUtils、runTestOnUiThread、@UiThreadTest を使った、ボタンクリックの処理は以下のようになります。

runTestOnUiThread runTestOnUiThread を使って Button のクリックを実行する例です。UI スレッドで実行したいコードを記述した Runnable を用意し、runTestOnUiThread メソッドの引数に与えます。

リスト 5.6: runTestOnUiThread のサンプル

```
1: public void testOnButtonClick_RunTestOnUiThread() throws Throwable {
2:
3:     runTestOnUiThread(new Runnable() {
4:         @Override
5:         public void run() {
6:             button.performClick();
7:         }
8:     });
9:
10:    String expected = activity.getString(R.string.clicked);
11:    assertEquals("ボタンテキストチェック", expected, button.getText().toString());
12: }
```

@UiThreadTest @UiThreadTest を使って Button のクリックを実行する例です。メソッドの先頭に @UiThreadTest アノテーションを追加します。

リスト 5.2: @UiThreadTest のサンプル

```
1: @UiThreadTest
2: public void testOnButtonClick_UiThreadTest() {
3:     button.performClick();
4:     String expected = activity.getString(R.string.clicked);
5:     assertEquals("ボタンテキストチェック", expected, button.getText().toString());
6: }
```

TouchUtils TouchUtils を使って Button のクリックを実行する例です。TouchUtils にはたくさんの UI 操作系の API が用意されていますが、クリックイベントを扱うときは clickView メソッドを実行します

リスト 5.3: TouchUtil のサンプル

```
1: public void testOnButtonClick_TouchUtils() {
2:     TouchUtils.clickView(this, button);
3:     String expected = activity.getString(R.string.clicked);
4:     assertEquals("ボタンテキストチェック", expected, button.getText().toString());
```

```
5: }
```

- clickView (InstrumentationTestCase test, View v):
 - 第2引数で指定した View のクリック操作をシミュレートする
 - 引数
 - * InstrumentationTestCase
 - 実行中の TestCase クラスのインスタンスを指定する
 - * View
 - クリック対象の View

5.2.2 UI テストの必要性

全てのイベントを網羅することは、膨大な工数が必要になり、アプリケーションの変更に強く影響するような弱い実装になるおそれがあります。そのため、単体テストでは単純で必要なテストケースのみに抑えたほうが良い。

実装の必要性を検討する

- UI テストはビジネスロジックのテストと違い、内部ロジックに強く依存している
- JUnit4 のような強力なモックライブラリが存在しない
- 複雑な操作は実装コストも高く可動性も低く、仕様変更に弱い
- 手動テストの方が十分な効果が期待できる
- シナリオテストや結合テストで置き換えることができる

5.3 UI Testing Sample

UI Testing Sample

- テスト内容
 - Buttonが表示されているかを確認する
 - Buttonが押されたときのテキストチェック
 - 表示テキストが「clicked!」になっているかを確認する
- 対象プロジェクト



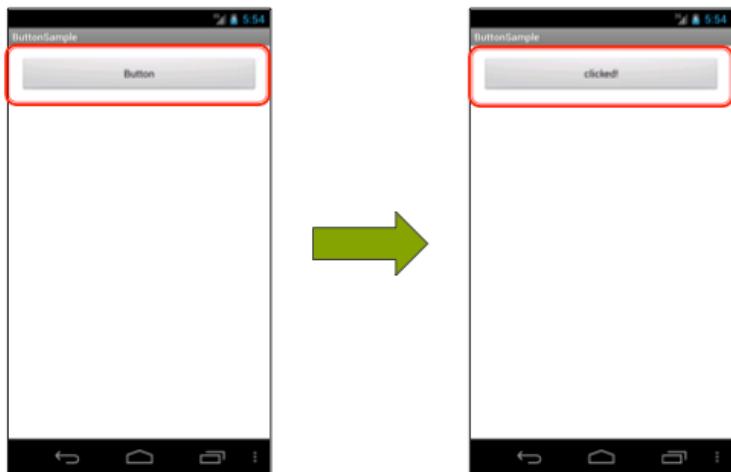
プロジェクト名	説明
ButtonSample	テストターゲット
ButtonSampleTest	テストプロジェクト

5.3.1 アプリケーションの説明

アプリケーションの説明



- ButtonSample
- ButtonをクリックするとButtonのTextが変わる



サンプルプログラムを使って UI テストの方法を説明します。

サンプルアプリケーション : ButtonSample

ButtonSample プロジェクトは、解答ドキュメントからダウンロードして下さい。

5.3.2 テストプロジェクト概要

以下の設定でテストプロジェクトを作成します。

表 5.2 プロジェクト概要

項目	設定値
Project Name	ButtonSampleTest
Build Target	トレーニングで指定したバージョン
Test Target	ButtonSample
Pakage	com.example.buttonsample.test

手順

1. ActivityInstrumentationTestCase2 のサブクラスを作成する

2. テストメソッドを作成する

5.3.3 手順 1. ActivityInstrumentationTestCase2 のサブクラスを作成する

1. ActivityInstrumentationTestCase2 のサブクラスを作成する
2. コンストラクタを作成する
3. setUp メソッドを作成する

リスト 5.4: MainActivityTest.java

```
1: public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {  
2:  
3:     private MainActivity activity;  
4:     private View rootView;  
5:     private Button button;  
6:  
7:     public MainActivityTest() {  
8:         super(MainActivity.class);  
9:     }  
10:  
11:    @Override  
12:    protected void setUp() throws Exception {  
13:        activity = getActivity();  
14:        rootView = activity.getWindow().getDecorView();  
15:        button = (Button) activity.findViewById(R.id.button1);  
16:  
17:    }  
18: }
```

5.3.4 手順 2. テストメソッドを作成する

動作検証用に以下の 4 つのメソッドを作成します

- testPreconditions
- testOnClickButton_runTestOnUi
- testOnClickButton_UiThreadTest
- testOnClickButton_TouchUtils

testPreconditions

アプリケーションの初期状態の確認をします。このメソッドでは Button が表示されていることを確認します。

Button の表示確認には ViewAsserts.assertOnScreen メソッドを使います

- assertOnScreen(View origin, View view)
 - 第 2 引数で指定した View が画面に表示されているか確認する。

- 引数
 - * View origin
 - Activity のルート View を指定する。タイトルバーを含む完全なコンテンツルートを指定するのが一般的であるが、setContentView で指定した View でも問題ない
 - * View view
 - 確認対象の View

リスト 5.5: testPreconditions メソッド

```
1: public void testPreconditions() {  
2:     ViewAsserts.assertOnScreen(rootView, button);  
3: }
```

testOnClickButton_runTestOnUiThread

Button クリック後に Text が「clicked!!」になっていることを確認するためのテストメソッドを作成します。

UI 操作には runTestOnUiThread を使います

リスト 5.6: testOnClickButton_runTestOnUiThread メソッド

```
1: public void testOnClickButton_runTestOnUiThread() throws Throwable {  
2:  
3:     runTestOnUiThread(new Runnable() {  
4:         @Override  
5:         public void run() {  
6:             button.performClick();  
7:         }  
8:     });  
9:  
10:    String expected = activity.getString(R.string.clicked);  
11:    assertEquals("ボタンテキストチェック", expected, button.getText().toString());  
12: }  
13:
```

testOnClickButton_UiThreadTest

Button クリック後に Text が「clicked!!」になっていることを確認するためのテストメソッドを作成します。

UI 操作には @UiThreadTest を使います

リスト 5.7: testOnClickButton_UiThreadTest メソッド

```
1: @UiThreadTest
2: public void testOnClickButton_UiThreadTest() {
3:     button.performClick();
4:     String expected = activity.getString(R.string.clicked);
5:     assertEquals("ボタンテキストチェック", expected, button.getText().toString());
6: }
```

testOnClickButton_TouchUtils

Button クリック後に Text が「clicked!」になっていることを確認するためのテストメソッドを作成します。

UI 操作には TouchUtils を使います

リスト 5.8: testOnClickButton_TouchUtils メソッド

```
1: public void testOnClickButton_TouchUtils() {
2:     TouchUtils.clickView(this, button);
3:     String expected = activity.getString(R.string.clicked);
4:     assertEquals("ボタンテキストチェック", expected, button.getText().toString());
5: }
6:
```

全体ソース

リスト 5.9: MainActivityTest

```
1: package com.example.buttonsample.test;
2:
3: import android.test.ActivityInstrumentationTestCase2;
4: import android.test.TouchUtils;
5: import android.test.UiThreadTest;
6: import android.test.ViewAsserts;
7: import android.view.View;
8: import android.widget.Button;
9:
10: import com.example.buttonsample.MainActivity;
11: import com.example.buttonsample.R;
12:
13: public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {
14:
15:     private MainActivity activity;
16:     private View rootView;
17:     private Button button;
18:
19:     public MainActivityTest() {
20:         super(MainActivity.class);
21:     }
22: }
```

```
23:     @Override
24:     protected void setUp() throws Exception {
25:         activity = getActivity();
26:         rootView = activity.getWindow().getDecorView();
27:         button = (Button) activity.findViewById(R.id.button1);
28:
29:     }
30:
31:     public void testPreconditions() {
32:         ViewAsserts.assertOnScreen(rootView, button);
33:     }
34:
35:     public void testOnClickButton_runOnUi() throws Throwable {
36:
37:         runTestOnUiThread(new Runnable() {
38:             @Override
39:             public void run() {
40:                 button.performClick();
41:             }
42:         });
43:
44:         String expected = activity.getString(R.string.clicked);
45:         assertEquals("ボタンテキストチェック", expected, button.getText().toString());
46:     }
47:
48:     @UiThreadTest
49:     public void testOnClickButton_UiThreadTest() {
50:         button.performClick();
51:         String expected = activity.getString(R.string.clicked);
52:         assertEquals("ボタンテキストチェック", expected, button.getText().toString());
53:     }
54:
55:     public void testOnClickButton_TouchUtils() {
56:         TouchUtils.clickView(this, button);
57:         String expected = activity.getString(R.string.clicked);
58:         assertEquals("ボタンテキストチェック", expected, button.getText().toString());
59:     }
60:
61: }
```

5.3.5 確認

テストを実行し、バーが緑になっていることを確認します。

5.4 【実習】UI Testing



5.4.1 実習

サンプルで説明したテストを作成します。

5.4.2 確認

テストを実行し、バーが緑になっていることを確認します。

5.4.3 解答

別ドキュメントを参照して下さい。

第6章

JUnit の応用技術



6.1 概要

概要

- テストレポート
- TestSuite
- TestSize



6.2 テストレポート

テストレポート

- JUnitの実行結果をxml形式で出力することができる



6.2.1 テストレポートを作成する

Eclipse では標準でテスト結果を xml 形式に出力することができます。

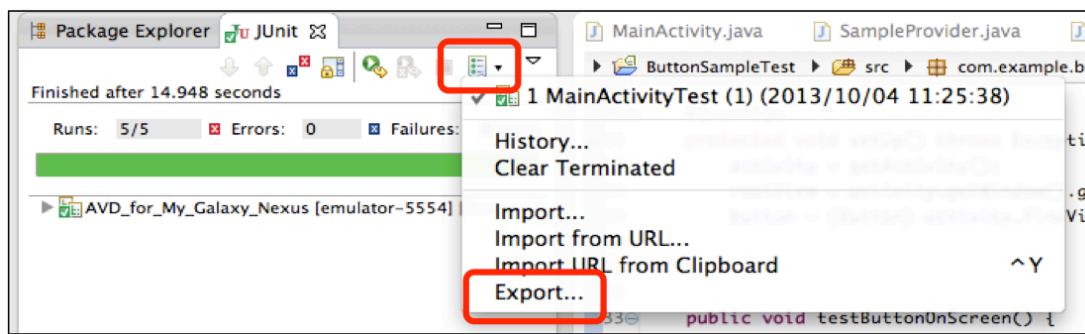
テストレポートを見ることで、テストにどのくらいの時間が必要か、何件テストしたかなどが確認できます。

- レポートには以下の情報が記録されます
 - 実行時間
 - 成功数、失敗数、全体の数
 - テスト対象のクラス
 - 実行したテストメソッド

6.2.2 テストレポートの作成方法

手順

1. JUnit ビューの「TestRunHistory」ボタンの「...」をクリック
2. 「Export...」を選択
3. ファイル名を指定して保存する



The screenshot shows the Eclipse IDE interface with the JUnit View expanded to show individual test cases. The tree view shows 'AVD_for_My_Galaxy_Nexus [emulator-5554] [Runner: JUnit 3] (11.422 s)' and 'com.example.buttonsample.test.MainActivityTest (11.422 s)'. Under 'MainActivityTest', four test cases are listed with their execution times: 'testButtonOnScreen (2.160 s)', 'testOnClickButton_TouchUtils (4.419 s)', 'testOnClickButton_UIThreadTest (2.534 s)', and 'testOnClickButton_runOnUi (2.308 s)'. A yellow arrow points from the JUnit View to a code block below containing the XML output.

```
<?xml version="1.0" encoding="UTF-8"?>
<testrn name="MainActivityTest (1)" project="ButtonSampleTest" tests="4" started="4" failures="0" errors="0" ignored="0">
  <testsuite name="AVD_for_My_Galaxy_Nexus [emulator-5554]" time="11.422">
    <testsuite name="com.example.buttonsample.test.MainActivityTest" time="11.422">
      <testcase name="testButtonOnScreen" classname="com.example.buttonsample.test.MainActivityTest" time="2.16"/>
      <testcase name="testOnClickButton_TouchUtils" classname="com.example.buttonsample.test.MainActivityTest" time="4.419"/>
      <testcase name="testOnClickButton_UIThreadTest" classname="com.example.buttonsample.test.MainActivityTest" time="2.534"/>
      <testcase name="testOnClickButton_runOnUi" classname="com.example.buttonsample.test.MainActivityTest" time="2.308"/>
    </testsuite>
  </testsuite>
</testrn>
```

JUnitView に表示されている内容が XML 形式で出力されます

6.2.3 【実習】テストレポート



実習

前項までの説明を参考にして、ButtonSampleTest のレポートを作成する。

- テストターゲット
 - ButtonSample
- テストプロジェクト
 - ButtonSampleTest

確認

テストレポートが作成されていることを確認する

6.3 TestSuite

TestSuite

- 複数のTestCaseをまとめて実行したい時はTestSuiteクラスを使用する
- 3つの手法
 - ① 指定したパッケージ配下のTestCaseをまとめて実行
 - ② 複数のTestSuiteをまとめて実行する
 - ③ 指定したTestCaseをまとめて実行する(APIlevel 16以降)



TestSuite クラスを使用してテストをパッケージごとに細分化することで、テスト実行時間の節約や機能ごとに分けてテストすることができます。

- TestSuite クラスを使用することで以下のことが可能
 - 指定したパッケージ配下のTestCaseをまとめて実行
 - 複数の TestSuite をまとめて実行する
 - 指定した TestCase をまとめて実行する (APIlevel 16 以降)

6.3.1 TestSuite Sample

TestSuite Sample

- サンプルを使って 3 つの手法で TestSuite の作成方法を説明
- 3 つの手法
 - ① 指定したパッケージ配下の TestCase をまとめて実行
 - ② 複数の TestSuite をまとめて実行する
 - ③ 指定した TestCase をまとめて実行する
- 対象プロジェクト

プロジェクト名	説明
JUnitDatabaseSample	テストターゲット
JUnitDatabaseSampleTest	テストプロジェクト

サンプルを使って、それぞれ手法で TestSuite の作成方法を紹介します。
サンプルプログラムは Database Testing で使用したテストプロジェクトを使用します。

6.3.2 指定したパッケージ配下の TestCase をまとめて実行する

指定したパッケージ配下の TestCase をまとめて実行する TestSuite を作成するところができます。

作成方法

ここではパッケージ「jp.oesf.databasesample.test.db」配下の TestCase をまとめて実行する TestSuite を作成します。

手順

1. TestSuite のサブクラスを作成
2. suite メソッドの作成
3. TestSuiteBuilder を使って TestSuite を作成する

手順 1. TestSuite のサブクラスを作成

以下の設定値で DBAllTests クラスを作成します

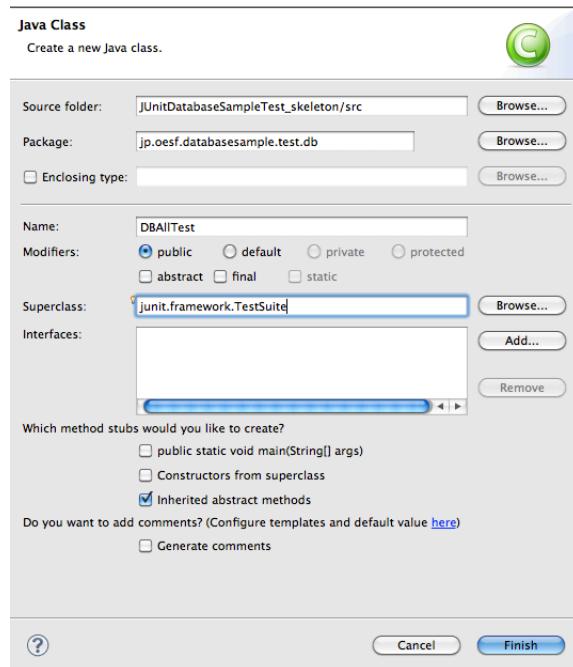


図: TestSuite の作成

表 6.1 新規 TestSuite 作成の設定値

項目	設定値
package	jp.oesf.databasesample.test.db
Name	DBAllTests
Superclass	junit.framework.TestSuite

リスト 6.1: DBAllTest クラスの作成

```
1: package jp.oesf.databasesample.test.db;
2:
3: import junit.framework.TestSuite;
4:
5: public class DBAllTests extends TestSuite {
6:
7: }
```

手順 2. suite メソッドの作成

public static 指定で suite メソッドを作成します。戻り値は Test です。

リスト 6.3: DBAllTests クラスの作成

```
1: public class DBAllTests extends TestSuite {  
2:     public static Test suite(){  
3:     }  
4:     }  
5: }
```

手順 3. TestSuiteBuilder を使って TestSuite を作成する

対象パッケージ配下の TestCase を TestSuite に追加します。

1. 引数に TestSuite のクラス情報を指定して TestSuiteBuilder のオブジェクトを作成する
2. 作成した TestSuiteBuilder の includeAllPackagesUnderHere メソッドを実行して TestCase を一括登録する
3. TestSuiteBuilder#build メソッドを実行して TestSuite を作成する
4. suite メソッドの戻り値に作成した TestSuite を指定する

リスト 6.3: DBAllTest クラスの作成

```
1: public class DBAllTests extends TestSuite {  
2:     public static Test suite(){  
3:         return new TestSuiteBuilder(DBAllTests.class).includeAllPackagesUnderHere().build();  
4:     }  
5: }
```

6.3.3 TestSuite を実行する

Eclipse からは通常の TestCase 同様の方法で実行することができます。

TestSuite クラスを右クリック > Run As > Android JUnit Test を選択すると対象の TestCase が実行されます

6.3.4 【実習】TestSuite 1



実習 1

以下のパッケージ配下のテストケースをまとめて実行する TestSuite を作成する

- jp.oesf.databasesample.test.db
 - クラス名 : DBAllTests
- jp.oesf.databasesample.test.provider
 - クラス名 : ProviderAllTests

確認

- テストを実行してバーが緑になっていることを確認
- 対象のパッケージ配下のテストケースが実行されていることを確認

(4 章 DatabaseTesting の補足実習 2 を完了していない場合はバーは赤くなります)

解答

別ドキュメント参照

6.3.5 複数の TestSuite をまとめて実行する

複数の TestSuite をまとめて実行することができます

作成方法

ここでは「jp.oesf.databasesample.test」パッケージに「AllTest」クラスを作成します
手順

1. TestSuite のサブクラスを作成
2. suite メソッドの作成
3. TestSuite に TestSuite を複数登録する

手順 1,2 までは前項で紹介した TestSuite の作成と同じです

手順 3. TestSuite に TestSuite を複数登録する

suite メソッド内で実行したい TestSuite の登録をします。TestCase の追加では TestSuiteBuilder を使っていましたが、TestSuite を追加するときは TestSuite オブジェクトを生成して、addTest メソッドで追加します。

1. TestSuite のオブジェクトを作成
2. 作成した TestSuite に addTest メソッドを使って Test を登録する
3. 引数に作成済 TestSuite#suite メソッドを指定する
4. suite メソッドの戻り値に作成した TestSuite を指定する

リスト 6.4: AllTest クラスの作成

```
1: public class AllTests extends TestSuite {  
2:     public static Test suite() {  
3:         TestSuite suite = new TestSuite();  
4:         suite.addTest(DBAllTests.suite());  
5:         suite.addTest(ProviderAllTests.suite());  
6:         return suite;  
7:     }  
8: }
```

6.3.6 実行

前項の手順と同じです。

- TestSuite クラスを右クリック > Run As > Android JUnit Test

6.3.7 【実習】TestSuite 2



実習 2

以下の TestSuite を実行する TestSuite 「AllTests」 を作成する

- DBAllTests
- ProviderAllTests

確認

- テストを実行してバーが緑になっていることを確認
- 対象のテストケースが実行されていることを確認

(4 章 DatabaseTesting の補足実習 2 を完了していない場合はバーは赤くなります)

解答

別ドキュメント参照

6.3.8 指定した TestCase をまとめて実行する (API level 16 以降)

API Lv16 より指定した TestCase を複数まとめて実行することが可能になりました。

作成方法

ここでは「jp.oesf.databasesample.test」パッケージに以下の TestCase を実行する「ChoicedTests」クラスを作成します

- BookDaoTest.class
- SampleProviderTest

手順

1. TestSuite のサブクラスを作成する
2. suite メソッドの作成
3. TestSuite に Test を複数登録する

手順 1,2 までは前項で紹介した TestSuite の作成と同じです

手順 3. TestSuite に Test を複数登録する

引数付きコンストラクタを使って、実行したい TestCase を複数指定します。また、API Lv16 以降がターゲットなため、suite メソッドまたは、TestSuite クラスの定義に @TargetApi を追加します。

- suite メソッドまたは、TestSuite クラスの定義に @TargetApi を追加する
 - 引数に「Build.VERSION_CODES.JELLY_BEAN」を指定する
- 引数付きコンストラクタを使って TestSuite のオブジェクトを作成する
 - 引数に実行したい TestCase のクラス情報を複数指定する

リスト 6.5: ChoicedTests クラスの作成

```
1: @TargetApi(Build.VERSION_CODES.JELLY_BEAN)
2: public class ChoicedTests extends TestSuite {
3:     public static Test suite() {
4:         TestSuite suite = new TestSuite(BookDaoTest.class, SampleProviderTest.class);
5:         return suite;
6:     }
7: }
```

6.3.9 実行

前項の手順と同じです。

- TestSuite クラスを右クリック > Run As > Android JUnit Test

6.3.10 【実習】TestSuite 3



実習 3

以下の TestCase を実行する TestSuite 「ChoisedTests」 を作成する

- BookDaoTest
- SampleProviderTest

確認

- テストを実行してバーが緑になっていることを確認
- 対象のテストケースが実行されていることを確認

解答

別ドキュメント参照

6.4 TestSize

TestSize

- TestSizeを使うことで規模によるテストの振り分けができる
- `@[Size]Test`を使ってテストメソッド単位でTestSizeを指定する



TestSize を使うことで、テストの大きさ、実行時間、依存関係などテストの規模で振り分けることができます。

TestSuite によるテストの振り分けではネットワーク通信するテストと、シンプルなロジックテスト、実行時間の短いテストと長いテストが同じレベルでテストされます。TestSize を使用するとテストの外部依存や規模感で振り分けすることができます。

どのテストを実行するとか、機能別にテストするという方法は TestSuite で行い、単純に規模やサイズによる振り分けは TestSize を使います。

参考 <http://googletesting.blogspot.jp/2010/12/test-sizes.html>

6.4.1 TestSize の指定

テストを振り分けるには、メソッド単位で TestSize を設定するためのアノテーションを追加します。

TestSize アノテーションは以下のものが用意されています。

- @SmallTest
- @MidiumTest
- @LargeTest

以下の表を参考にして TestSize を決定します。

表 6.2 TestSize の基準

機能	Small	Medium	Large
ネットワーク通信	No	localhost のみ	Yes
データベース	No	Yes	Yes
ファイルアクセス	No	Yes	Yes
外部システム	No	Yes	Yes
System properties	No	Yes	Yes
実行時間(秒)	60	300	900 以上

6.4.2 TestSize の使い方

JUnitDatabaseSampleTest プロジェクトを使って TestSize 使い方を説明します。

手順

1. @|[Size]Test を追加する
2. Eclipse メニューから TestSize を指定して実行する

手順 1. @|[Size]Test を追加する

BookDaoTest#test GetAll に@SmallTest を追加します。

リスト 6.6: BookDao の test GetAll メソッドに TestSize を追加

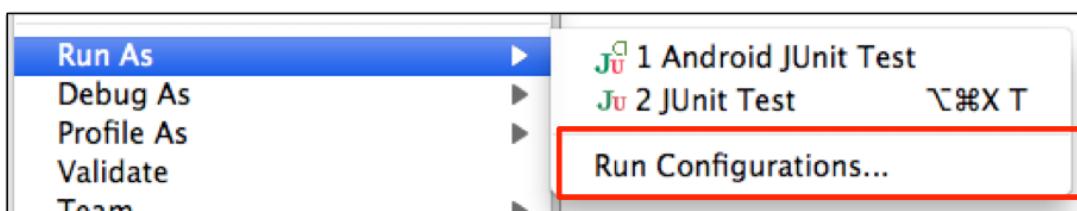
```

1:      @SmallTest
2:      public void test GetAll() {
3:          String filename = "insert.sql";
4:          String filenameExpected = "books_fixture.yaml";
5:
6:          ... 略...
7:      }

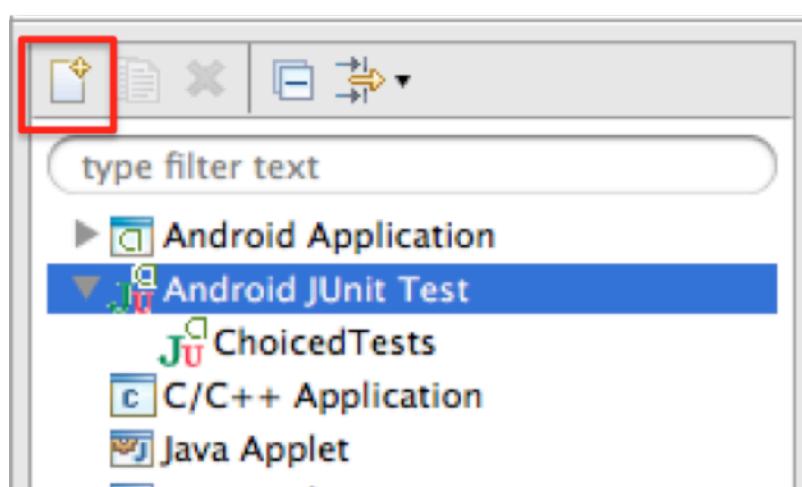
```

手順 2. Eclipse メニューから TestSize を指定して実行する

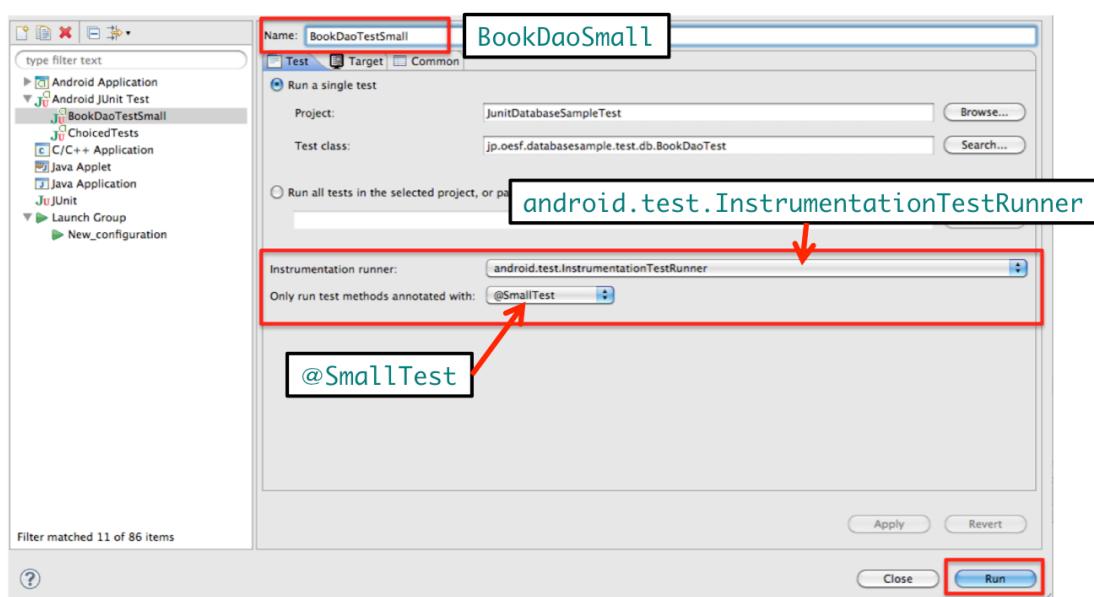
1. TestCase クラス右クリック > Run as > Run Configuration を選択



1. 追加ボタンをクリック



1. Configuration の設定値を以下のようにする
2. Run ボタンをクリックしてテストを実行する



確認

テストを実行すると対象の TestSize のみ実行される



6.5 【実習】 TestSize



6.5.1 実習

サンプルを参考に次のメソッドに @SmallTest を設定する

- BookDaoTest#get
- BookDaoTest#getAll (4 章の補足実習を完了している場合)

6.5.2 確認

テストを実行すると対象の TestSize のみ実行されるのを確認する

第7章

ストレステスト



7.1 概要

概要

- ストレステストとは

- Monkeyツール



7.2 ストレステストとは

ストレステストとは

- システムに必要以上の負荷をあたえても問題なく動作するかテストする



システムに過剰な負荷をあたえたり、でたらめな動作を実行したりして、システムの耐久性や安定性を検証するテストです。

- ストレステスト
 - システムに必要以上の負荷をあたえても問題なく動作するかを検証する
- モンキーテスト
 - ランダムにイベントを発生させてもシステムが問題なく動作するかを確認するストレステストの一種

7.3 Monkey ツール

Monkeyツール

- モンキーテストを実行するツール
- AndroidではMonkeyツールを使ってストレステストを行う



AndroidSDKに同梱されているツールです。クリックやジェスチャーなどのイベントをランダムに発生させアプリケーションの安定性や耐久性のテストを行います。

- Monkey ツールの特徴
 - クリックやジェスチャーなどのイベントをランダムに発生させる
 - キーイベントの発生にも対応
 - 発生回数やインターバルを指定できる

7.3.1 Monkey ツールの使い方

Monkeyツールの使い方



- コマンドラインからMonkeyツールの起動コマンドを打って実行する
 - コマンド

```
$ adb shell monkey [オプション] <イベント回数>
```

- 主なオプション

オプション	説明
-v	出力レベルの設定 0-2(0:最小～2:詳細) の範囲で指定する。
-s	乱数のシード値。同じ値を指定すると同じイベントを発生させる。一度実行したMonkeyを再現させるときに有効なオプション
-throttle	イベントの発生間隔 (ミリ秒)
-p	パッケージ名を指定

コマンドラインから Monkey ツールの起動コマンドを打って実行します

```
$ adb shell monkey [オプション] <イベント回数>
```

例)

パッケージ名： your.package.name に 500 回のランダムイベントを発生させる
対象のアプリケーションがインストールされていること

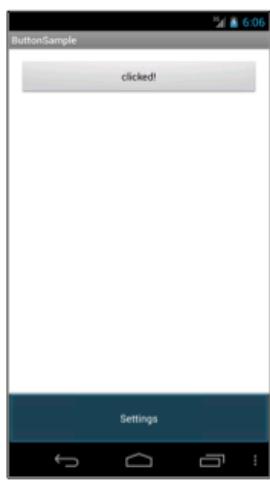
```
$ adb shell monkey -p your.package.name -v 500
```

7.3.2 Monkey ツールの実行

Monkeyツールの実行



- 画面ではイベントが実行され、ターミナルにはログが出力される



```
$ adb shell monkey -p com.example.buttonsample -v 500
:Monkey: seed=1376335967912 count=500
:AllowPackage: com.example.buttonsample
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: -0.0%
// 5: 25.0%
// 6: 15.0%
// 7: 2.0%
// 8: 2.0%
// 9: 1.0%
// 10: 13.0%
:Switch:
:Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;component=com.example.buttonsample/.MainActivity;end
    // Allowing start of Intent
{ act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
cmp=com.example.buttonsample/.MainActivity } in package com.example.buttonsample
:Sending Touch (ACTION_DOWN): 0:(121.0,487.0)
:Sending Touch (ACTION_UP): 0:(135.44057,509.5309)
:Sending Touch (ACTION_DOWN): 0:(249.0,7.0)
```

Monkey ツールを実行すると発生したイベントが端末に送信され、端末の画面では送信されたイベント実行されます。ターミナルにはログが出力されます

7.3.3 【実習】Monkey ツール



7.3.4 Monkey ツール実習

実習

Monkey ツールを使ってモンキーテストを実行します。

- 対象アプリケーション : ButtonSample
- 指定オプション
- パッケージ名 : com.example.buttonsample
- 回数 : 500

コマンド

```
$ adb shell monkey -p com.example.buttonsample -v 500
```

確認

アプリケーションが強制終了しないことを確認します。

実行例

```
$ adb shell monkey -p com.example.buttonsample -v 500
:Monkey: seed=1382220981560 count=500
:AllowPackage: com.example.buttonsample
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: -0.0%
// 5: 25.0%
.
.
.

:Sending Touch (ACTION_UP): 0:(184.8112,1020.64825)
:Sending Touch (ACTION_DOWN): 0:(445.0,486.0)
Events injected: 500
:Sending rotation degree=0, persist=false
:Dropped: keys=0 pointers=8 trackballs=0 flips=0 rotations=0
## Network stats: elapsed time=2456ms (0ms mobile, 2456ms wifi, 0ms not connected)
// Monkey finished
```

第8章

シナリオテスト



8.1 概要

概要

- シナリオテストとは
- monkeyrunner
- uiautomator



8.2 シナリオテストとは

シナリオテストとは

- シナリオテストとは
 - テストシナリオにしたがってシステム全体をテストする
- Andriod UI Testing
 - UIテスト用に提供されたツール
 - monkeyrunner
 - uiautomator



※ 本トレーニングでは、主にUI操作系のシナリオテストについて解説

- システム全体の各ユースケースを想定したシナリオを作成し実行するテスト
- モジュール化を結合し、モジュール間のインターフェースのテストを行う
- 一連の処理をブラックボックスとして扱い、入力に対する結果を確認する

本トレーニングでは主に、UI 操作系のシナリオテストについて解説します

8.2.1 テストシナリオ

シナリオテストで実行する、システムのユースケースをテストシナリオと呼びます。

8.2.2 Android UI Testing

コンポーネントのテストと同様に、UI のテストも重要です。UI テストでは、アプリケーションがユーザアクションのシーケンスに応答して適切な UI 出力を返すことが保証されなくてはいけません。Android では UI 操作をシーケンス化した TestingFramework が提供されており、シナリオテストとして利用することができます。

例) EditText にフォーカスを当てる > キーボード入力 > Button をクリックする

8.2.3 ツールの紹介

Android SDK では、シナリオテストを実行するためのツールとして以下のものが提供されています。

- monkeyrunner
- MonkeyRecorder
- uiautomator

MonkeyRecorder は Android SDK の公式リリースツールに含まれていないため、説明は省略します

8.3 monkeyrunner

monkeyrunner



- monkeyrunnerとは
 - pythonで記述されたテストスクリプトを実行するツール
- 提供されている機能
 - マルチデバイス制御
 - 機能テスト
 - 回帰テスト
 - 拡張オートメーション
- 主要なAPI
 - MonkeyRunner
 - MonkeyDevice
 - MonkeyImage

monkeyrunner はシナリオテスト用のツールで、Android SDK に同梱されています。[android-sdk]¥ tools 以下に実行ファイルが用意されています。monkeyrunner では、Android のコード外からアプリケーションを操作可能な API が提供されています。Python で記述されたテストスクリプトを実行して、デバイスへイベントを送信できます。Python の Java 実装である Jython を使用しているため、Python の構文でアプリケーションの API にアクセス可能です。

特徴

- Python で記述されたテストスクリプトを実行して、デバイスにイベントを送信する
- 画面キャプチャを保存することができる
- Jython により、Python の構文でアプリケーションの API にアクセス可能

8.3.1 monkeyrunner が提供している機能

monkeyrunner は、以下の機能を提供しています。

- マルチデバイス制御
 - テストスイートを、複数のデバイス上で実行することができる

- 機能テスト
 - キーストロークやタッチイベントなどの操作を実行し、スクリーンショットで結果を保存することができる
- 回帰テスト
 - テスト結果のスクリーンショットと期待される正しいスクリーンショットを比較し、アプリケーションの安定性を検証できる
 - 動的に変化する出力結果では、比較対象からパーセンテージを指定して安定性を検証する
- 拡張オートメーション
 - Python のモジュールと Android デバイスを制御する AndroidAPI と連携し、標準 Python モジュールから adb などの Android ツールを呼び出すことができ、より完全なプログラムを作成することができる
 - プラグインを作成し monkeyrunner を拡張することができる

8.3.2 monkeyrunner の API

monkeyrunner は以下の 3 つのクラスから構成されています

MonkeyRunner

- monkeyrunner を操作するための Utiltiy クラス
- デバイスやエミュレータの接続やインスタンスの取得などを行うことができる

MonkeyDevice

- monkeyrunner からデバイスやエミュレータを操作する
- UI イベントの発行やデバイス情報の取得、アプリケーションのインストールやアンインストールを行うことができる

MonkeyImage

- 画面キャプチャのイメージオブジェクト
- 様々なフォーマットの画像形式に変換でき、ファイル出力機能も提供する

8.4 monkeyrunner Sample

monkeyrunner Sample



- テスト内容
 - メニュー押下後に、"Setting"メニューが表示されるか確認する
- テスト対象

プロジェクト名	説明
ButtonSample	テストターゲット
monkeyrunner_sample.py	テストスクリプト

ButtonSample アプリケーションを使って monkeyrunner の使い方を説明します。

monkeyrunner のテスト手順

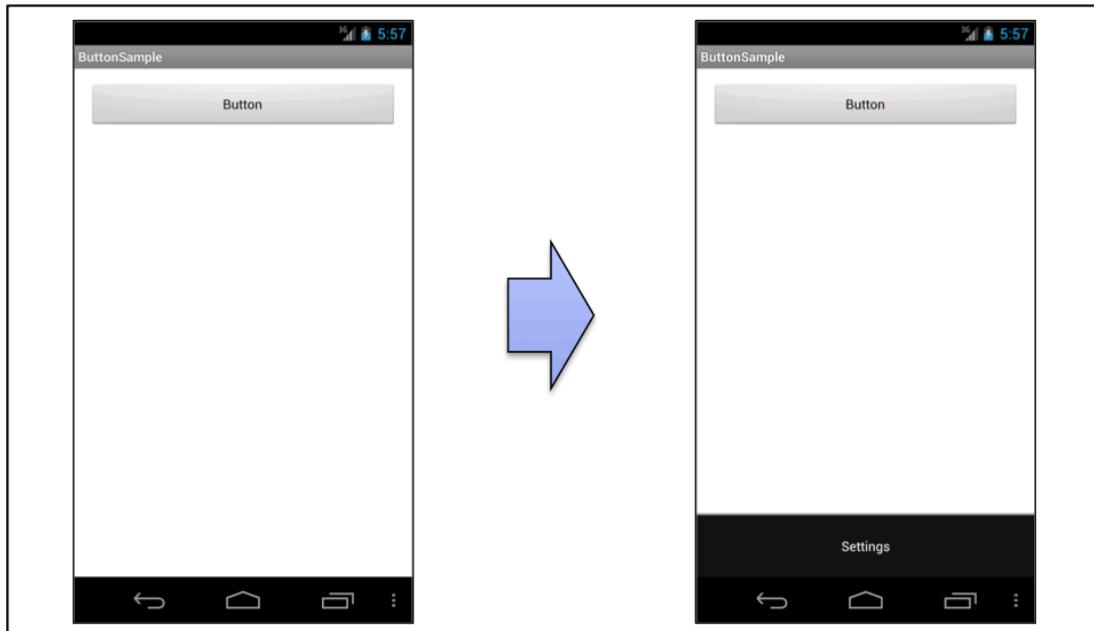
- 準備
 - テストシナリオの作成
 - 実行後の比較画像の準備（ 解答ドキュメントよりダウンロードする ）
- 作成
 - monkeyrunner のスクリプトを作成する
- 実行
 - monkeyrunner コマンドを実行する

8.4.1 テストシナリオの作成

テストシナリオ

1. ButtonSample アプリケーションを起動する
2. メイン画面でメニューキーを押下

3. 画面キャプチャを取得し、ファイル名「shot1.png」で保存する
4. 保存された画像をと「expect.png」を比較し、一致率が 70% 以上であることを確認する



8.4.2 テストスクリプトの作成

テストシナリオの動作手順に従い、monkeyrunner スクリプトを記述します。

手順

1. 端末に接続する
2. アプリケーションを起動する
3. メニューボタンを押す
4. スクリーンショットを取得する
5. 画像を比較する

リスト 8.1: サンプルプログラム monkeyrunner_sample.py

```
1: # -*- coding: utf-8 -*-
2:
3: # monkeyrunner モジュールのインポート
4: from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
5:
6: # 1. デバイスに接続し、MonkeyDevice オブジェクトを取得する
7: device = MonkeyRunner.waitForConnection()
8: print "waiting for connection...\n"
9:
10: # 2. 対象 Activity を起動する
```

```
11: package = 'com.example.buttonsample'
12: activity = 'com.example.buttonsample.MainActivity'
13: runComponent = package + '/' + activity
14: device.startActivity(component=runComponent)
15:
16: # 3. メニューボタンを押す
17: device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)
18:
19: MonkeyRunner.sleep(2)
20:
21: # 4. スクリーンショットの取得
22: result = device.takeSnapshot()
23: result.writeToFile('./shot1.png', 'png')
24:
25: # 5. 期待する画面のイメージを取得
26: expect = MonkeyRunner.loadImageFromFile('./expect.png')
27: expect.writeToFile('./shot2.png', 'png')
28:
29: # 6. 画面を比較する
30: print result.sameAs(expect, 0.5)
31:
32: if not result.sameAs(expect, 0.7):
33:     print "comparison failed!\n"
34:
```

手順 1. 端末に接続する

MonkeyRunner クラスの `waitForConnection` メソッドを実行して、デバイスまたはエミュレータに接続します。接続すると、`MonkeyDevice` オブジェクトが取得できます。

リスト 8.2: 手順 1. 端末に接続する

```
1: # 1. デバイスに接続し、MonkeyDevice オブジェクトを取得する
2: device = MonkeyRunner.waitForConnection()
3: print "waiting for connection...\\n"
```

手順 2. アプリケーションを起動する

`MonkeyDevice` クラスの `startActivity` メソッドを使って、`ButtonSample` の `MainActivity` を起動します。引数には "[アプリケーションパッケージ]/[Activity 名]" を指定します。アプリケーションが起動するまで、次の処理を実行しないように、5 秒間スリープします。

リスト 8.3: 手順 2. アプリケーションを起動する

```
1: # 2. 対象 Activity を起動する
2: package = 'com.example.buttonsample'
3: activity = 'com.example.buttonsample.MainActivity'
4: runComponent = package + '/' + activity
```

```
5: print runComponent  
6: device.startActivity(component=runComponent)  
7: MonkeyRunner.sleep(5)
```

手順 3. メニュー ボタンを押す

MonkeyDevice クラスの press メソッドを実行して、キーイベントを送信します。メニューが完全に表示されるまで、次の処理を実行しないよう、5 秒間スリープします。

リスト 8.4: 手順 3. メニュー ボタンを押下する

```
1: # 3. メニュー ボタンを押す  
2: print "press menu key"  
3: device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)  
4: MonkeyRunner.sleep(5)
```

手順 4. スクリーンショットを取得する

メニュー ボタン押下後の状態のスクリーンショットを取得します。MonkeyDevice クラスの takeSnapshot メソッドを実行すると、MonkeyImage オブジェクトが取得できます。MonkeyImage クラスの writeToFile メソッドを実行してファイルに書き出します。

リスト 8.5: 手順 4. スクリーンショットを取得する

```
1: # 4. スクリーンショットの取得  
2: print "take a screenshot"  
3: result = device.takeSnapshot()  
4: result.writeToFile('./shot1.png', 'png')
```

手順 5. 画像を比較する

今回のケースでは、メニュー アイテムに"Settings"が表示されているのが期待される状態です。あらかじめ用意した画像"expect.png"と、実際の実行後の画面の比較を行います。メソッドの第 2 引数に、画像の一致率を指定します。

今回のケースでは、期待する一致率は 70% 以上です。"expect.png"は、スクリプトファイルと同じディレクトリに配置されている必要があります。

リスト 8.6: 手順 5. 画像を比較する

```
1: # 5. 画面を比較する  
2: # 期待する画面のイメージを取得  
3: expect = MonkeyRunner.loadImageFromFile('./expect.png')
```

```
4: expect.writeToFile('./shot2.png','png')
5:
6: # 画像を比較し、一致率が 70% 以上か確認
7: if not result.sameAs(expect, 0.7):
8:     print "comparison failed!\n"
9: else:
10:    print "comparision ok!\n"
```

8.4.3 monkeyrunner の実行

1. テストスクリプトと確認用の画像ファイルの準備

確認用の画像ファイルを monkeyrunner 実行ディレクトリに配置します。

Linux, Mac Linux, Mac では eclipse の ButtonSample プロジェクトに配置すると便利です。

```
$ mv [eclipse workspace]/ButtonSample/bin/expect.png ./
$ ls
expect.png          monkeyrunner_sample.py
```

Windows windows の場合は monkeyrunner ツールと同じディレクトリに配置します。エクスプローラから、テストスクリプトと比較画像を配置します。

- monkeyrunner.bat の場所
 - [android-sdk]\tools

2. monkeyrunner コマンドを実行する

monkeyrunner を実行するにはコマンドラインから以下のコマンドを実行します。

1. テストスクリプト配置ディレクトリに移動
2. monkeyrunner コマンドで実行する

Linux, Mac monkeyrunner の実行コマンド

```
$ monkeyrunner [スクリプトファイル名]
```

今回の例では、以下のように入力します。また、スクリプトファイルと同じディレクトリに、画面キャプチャファイルが保存されます。

```
$ cd [eclipse workspace]/ButtonSample/bin/  
$ monkeyrunner monkeyrunner_sample.py
```

Windows monkeyrunner の実行コマンド

```
> monkeyrunner.bat [スクリプトファイル名]
```

今回の例では以下のように入力します。また、スクリプトファイルと同じディレクトリに画面キャプチャファイルが保存されます。

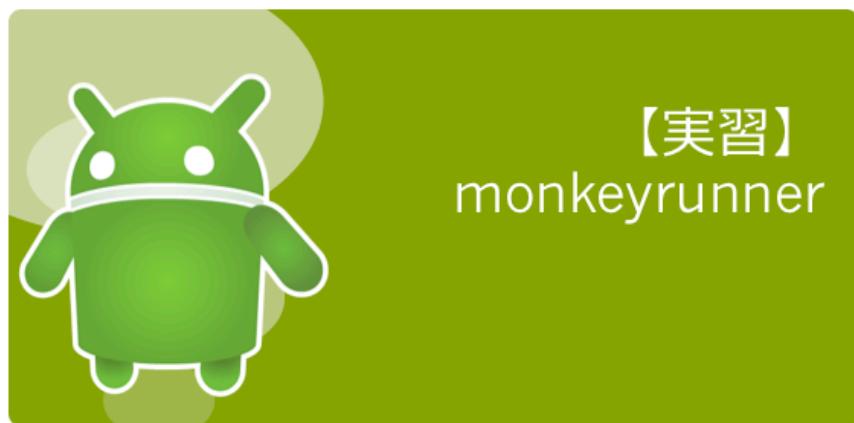
```
> cd [android-sdk]\tools  
> monkeyrunner.bat monkeyrunner_sample.py
```

実行結果の確認

- 出力された画像ファイルを目視し、メニューが起動していることを確認する
- エラーが起きないことを確認する
- コンソール上で、以下の内容が表示されていることを確認する

```
$ monkeyrunner monkeyrunner_sample.py  
waiting for connection...  
  
com.example.buttonsample/com.example.buttonsample.MainActivity  
press menu key  
take a screenshot  
comparision ok!
```

8.5 【実習】monkeyrunner



サンプルと同じテストを、実際に作成してください。

比較対象の画像は、解答ドキュメントより提供済です

8.6 uiautomator



8.6.1 uiautomator

uiautomator



- 効率的なユーザインターフェースのためのテスティングフレームワーク
- 以下のバージョンより対応
 - Android SDK 21以上
 - Android SDK Platform, API 16以上
- Javaプロジェクトとして作成する

効率的にユーザーインターフェイスをテストすることができる、テスティングフレームワークです。

自動化された機能的な UI のテストケースを作成することができます。

特徴

- 1つまたは複数のデバイス上で、アプリケーションに対して実行することができる
- 画面上の UI をスキャンできる
- 特定のアクションを実行したり、ユーザの操作をシミュレートするための API が提供されている
- 以下のバージョンより対応している
 - Android SDK 21 以上
 - Android SDK Platform, API 16 以上

8.6.2 操作をシミュレートする

uiautomator にはユーザの操作をシミュレートするための API が提供されています。uiautomator API は <android-sdk>/platforms/<target platform sdk>/uiautomator.jar に含まれて

います。

主な API

- Uidevice
- UiSelector
- UiObject
- UiCollection
- UiScrollable

UiDevice uiautomator からデバイスやエミュレータを操作するためのオブジェクトです。画面の向きや画面サイズ、Home や Menu キーの操作をシミュレートすることができます。

UiSelector 画面上の UI 要素をスキャンすることができます。スキャンする時に、いくつかの View プロパティの設定値を参照します。プロパティの設定に誤りがあると、正しくスキャンすることができないので注意が必要です。また、プロパティの設定値を確認するためのツールも提供されています。（ 必須プロパティについては後述 ）

UiObject UI 要素のオブジェクトです。Button や CheckBox などのような View が UIObject です。UiSelector を使ってオブジェクトを取得します。UIObject を使って View プロパティの情報を取得したし、クリックなどのアクションを実行することができます。

主な UiObject

- Button
- CheckBox

UiCollection Music Album やメール Box のように、複数のアイテムが含まれている UI 要素を扱うためのオブジェクトです。ListView や LinearLayout のような ViewGroup が UiCollection です。

主な UiCollection

- FrameLayout
- LinearLayout

UiScrollable スクロールに対応した UI 要素の集まりを UiScrollable といいます。水平方向または、垂直方向のスクロール操作をシミュレートします。

主な UiScrollable

- ListView
- Spinner

8.7 uiautomator Sample

uiautomator Sample



- テスト内容
 - Buttonをクリックし、テキストが「clicked!」になっていることを確認する
- 対象プロジェクト

プロジェクト名	説明
ButtonSample	テストターゲット
UIAutomatorSample	テストプロジェクト

8.7.1 uiautomator の使い方

uiautomator のテスト手順

- 準備 1. テストシナリオの作成 2. UI の分析
- 作成 1. UIAutomator のテストプロジェクトを作成する 2. テストケースの作成 3. テストプロジェクトのビルド 4. jar ファイルをデバイスに転送する
- 実行 1. uiautomator ツールを使ってテストを実行する

8.7.2 準備

テストシナリオの作成

テストシナリオ

1. ButtonSample アプリケーションを起動する
2. Button を押下する

3. Button のテキストが"clicked!"になっているか確認する

UI を分析する

対象の UI を操作するためには、その UI が画面のどこにあるのかを知っておく必要があります。uiautomator では View プロパティの設定値を使って画面上の UI を検索します。

テスト対象のアプリケーションの UI 情報を分析する

- どんな UI で構成されているか
- 対象となる UI コンポーネントの階層はどこか
- 設定情報は適切か

設定情報の確認

UI Automator Viewer を使って設定値を確認します。

uiautomator は UI の識別に特定のプロパティ (text, hint など) を参照するため、以下の値を設定する必要がある

- android:text
- android:contentDescription
- android:hint

8.7.3 UI Automator Viewer とは

UI Automator Viewer ツールは、レイアウト階層を検査し、個々の UI コンポーネントのプロパティを表示するためのツールです。

- UI のコンポーネントの解析ツール
- レイアウトの階層を検査し、UI コンポーネントのプロパティを表示することができる
- [android-sdk]/tools 以下に格納されている

UI Automator Viewer を起動する

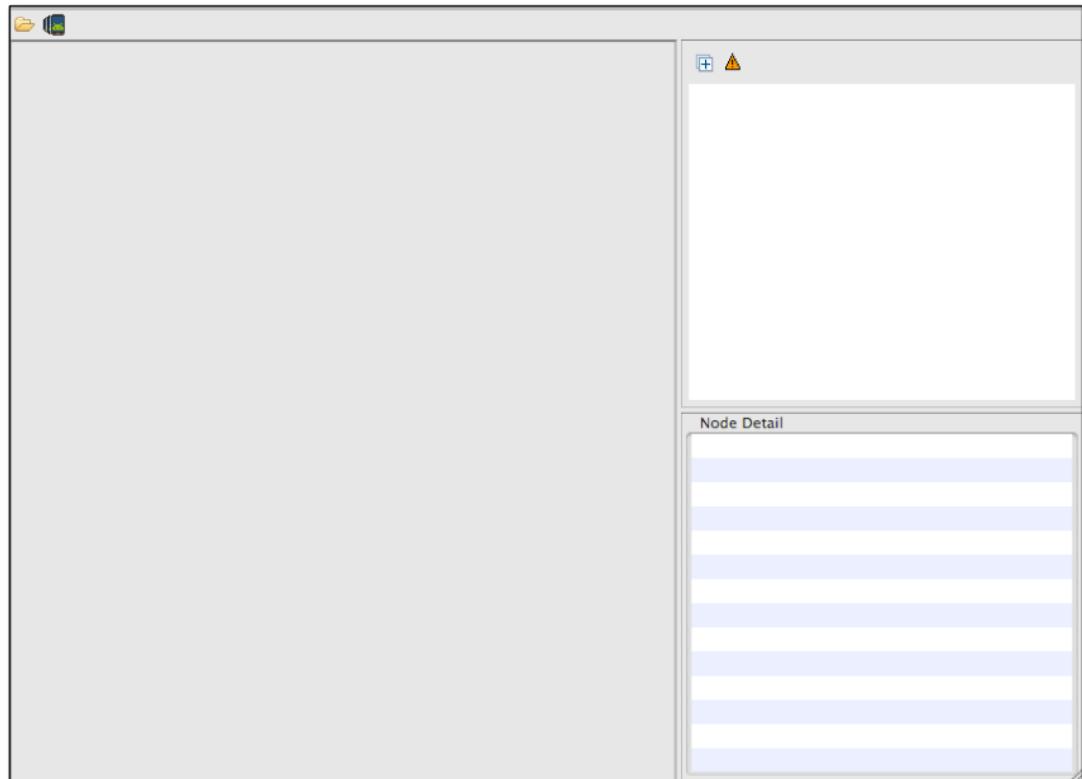
UI Automator Viewer を起動する 2 つの方法

- コマンドラインから起動する方法
- DDMS をつかって起動する方法

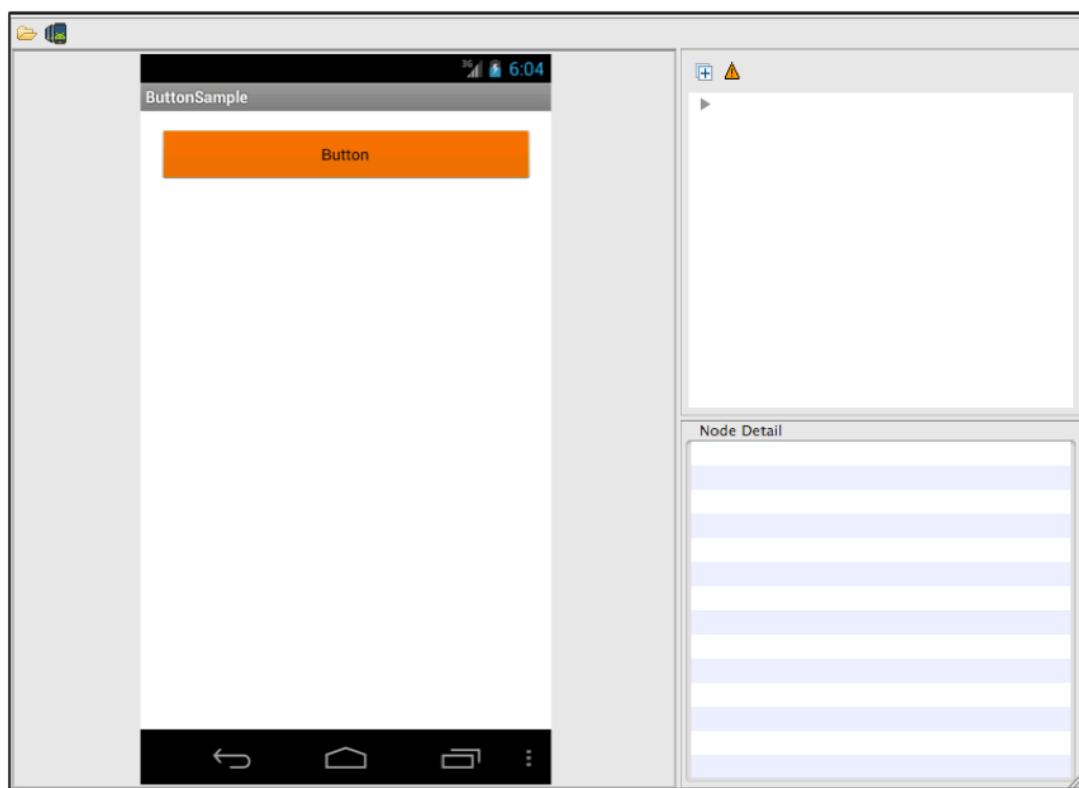
コマンドラインから起動する方法

- コマンドラインから、次のコマンドを実行する

```
$ uiautomatorviewer
```

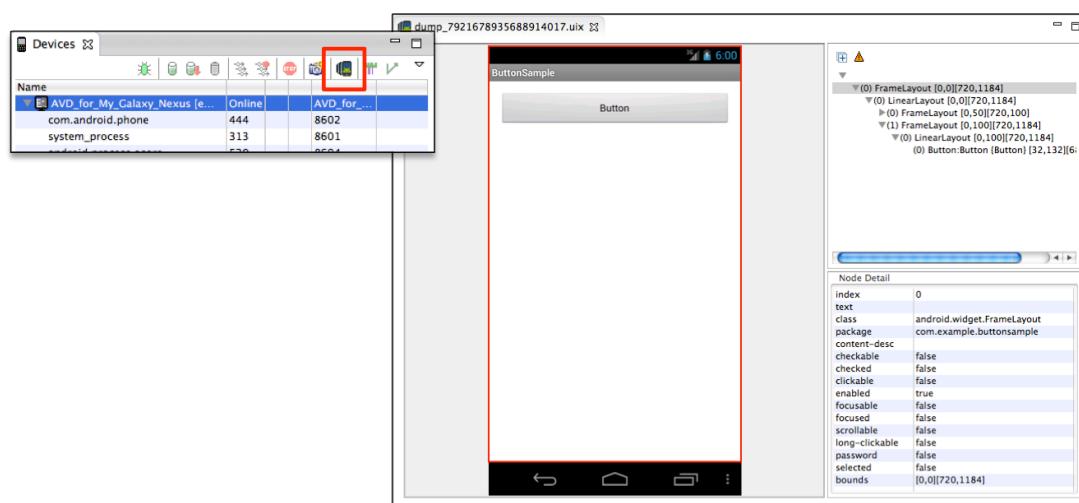


- 画面キャプチャを表示する
 - Device Screenshot ボタンをクリックし、現在の画面キャプチャを表示する



DDMS をつかって起動する方法

- DDMS パースペクティブ > Device View > Dump View Hierarchy for UI Automator ボタンをクリックする（Eclipse ではキャプチャされた状態で表示される）



8.7.4 UI の解析

UI Automator Viewer は、次の 3 つの領域で構成されています。

- 画面キャプチャ領域
- 階層構造領域
- プロパティ領域



View プロパティの確認

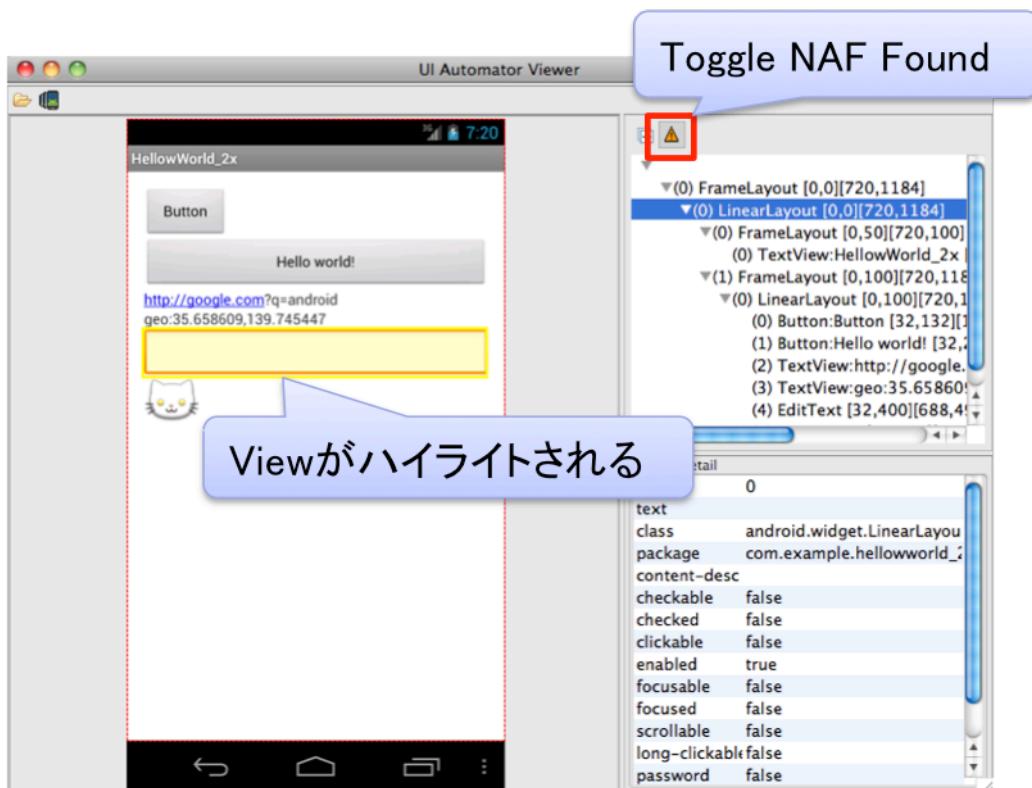
uiautomator の実行に必須の View プロパティを確認します。

今回の例では、Button の contentDescription が、"Button"になっていることを確認します。

index	0
text	Button
class	android.widget.Button
package	com.example.buttonsample
content-desc	Button
checkable	false
clickable	true
enabled	true
focusable	true
focused	true
scrollable	false
long-clickable	false
password	false
selected	false
bounds	[32,132][688,228]

設定漏れのチェック

Toggle NAF Found ボタンを押下すると、uiautomatorviewer の Toggle NAF Found ボタンを押すと、プロパティ設定されていない View がハイライトされます。

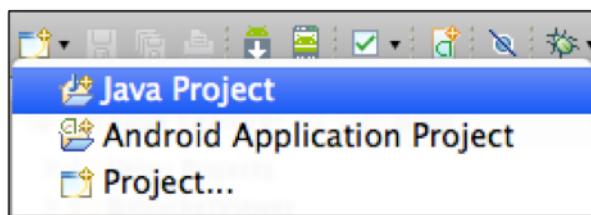


8.7.5 プロジェクトの作成

uiautomator のテストプロジェクトを作成します。

テストプロジェクト作成手順

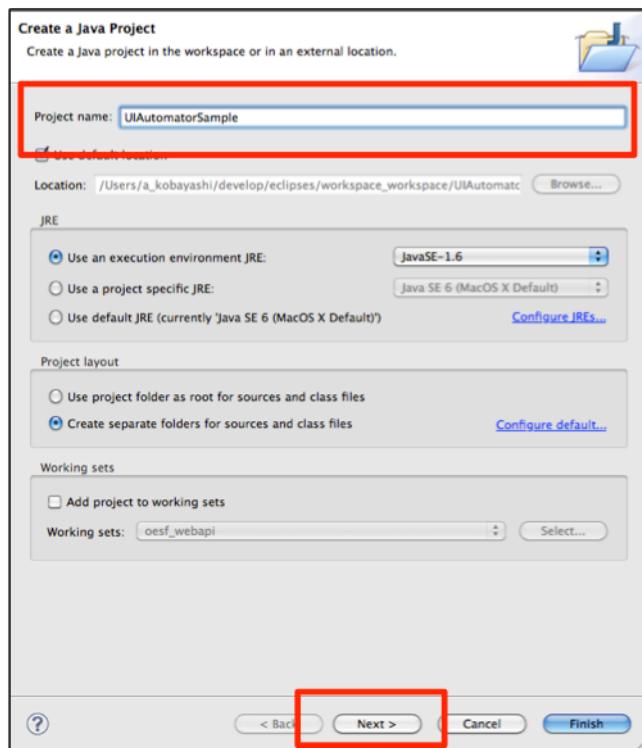
1. 新規 Java プロジェクトを作成



2. ProjectName にプロジェクト名を記入

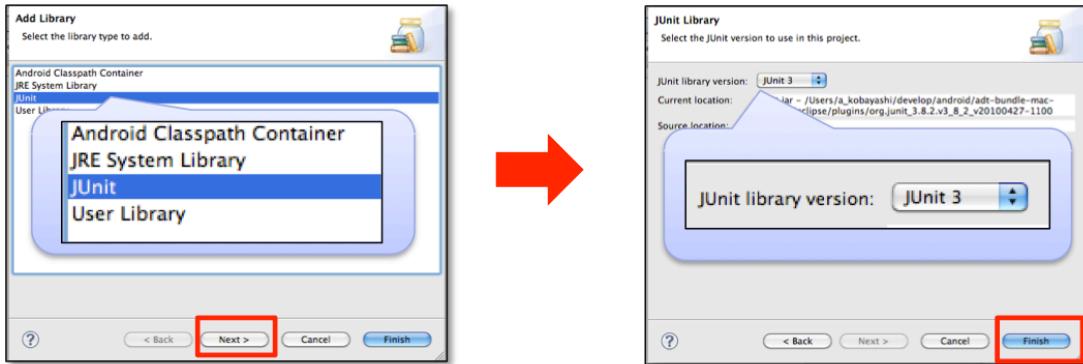
プロジェクト名: UIAutomatorSample

3. Next ボタンをクリック



4. Libraries に JUnit3 を追加する

Add Library > JUnit > JUnit 3 > Finish

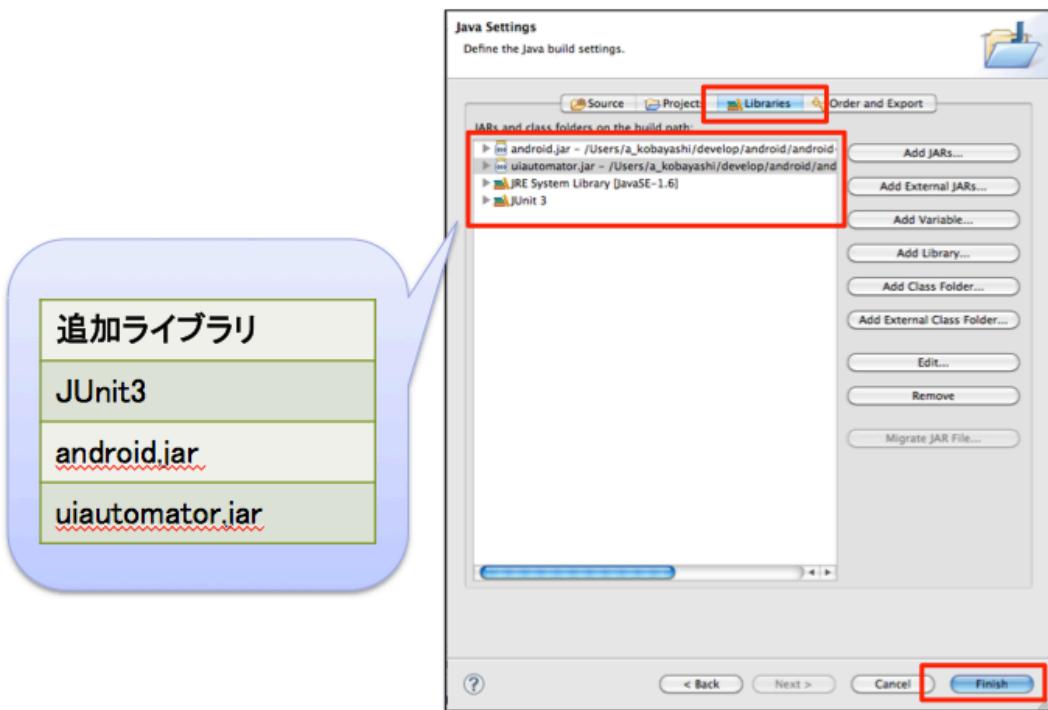


5. Libraries に以下の jar ファイルを追加する

パス : [android-sdk] ¥ platforms ¥ android-[API Lv] (API Lv は 17 以上を指定する)

- android.jar
- uiautomator.jar

6. Finish ボタンをクリック



8.7.6 テストケースの作成

手順

1. UIAutomatorTestCase クラスを継承したクラスの作成
2. テストメソッドの作成

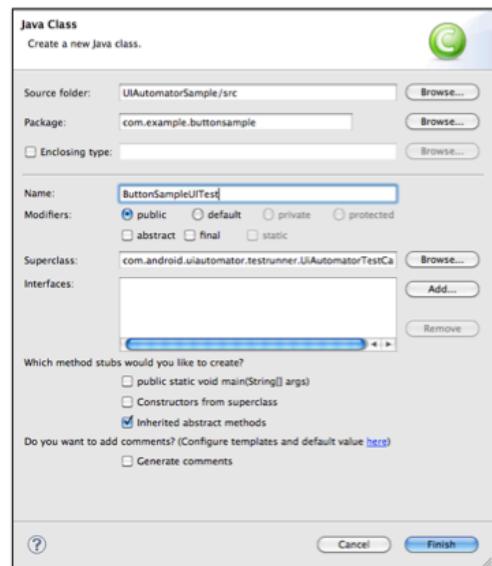
処理シーケンス

1. ホーム画面からアプリケーションの起動するまでの処理
2. アプリケーションが起動してからボタンをクリックするまでの処理
3. アサーション

1. UIAutomatorTestCase クラスを継承したクラスの作成

- uiautomator ツールを使ってテストするときは UIAutomatorTestCase クラスのサブクラスを用意する
- UIAutomatorTestCase クラスは通常のクラスを作成する方法で作成する

項目	値
Package	com.example.buttonsample
Name	ButtonSampleUITest
Superclass	com.android.uiautomator.testrunner.UiAutomatorTestCase

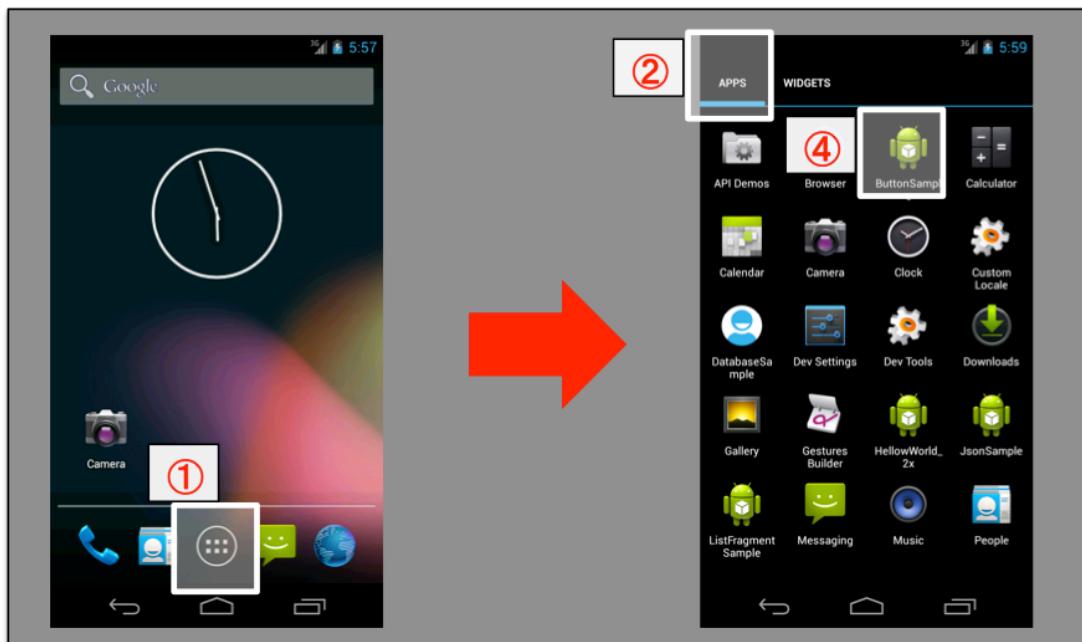


2. テストメソッドの作成

ボタンクリックのテストメソッドを作成します。

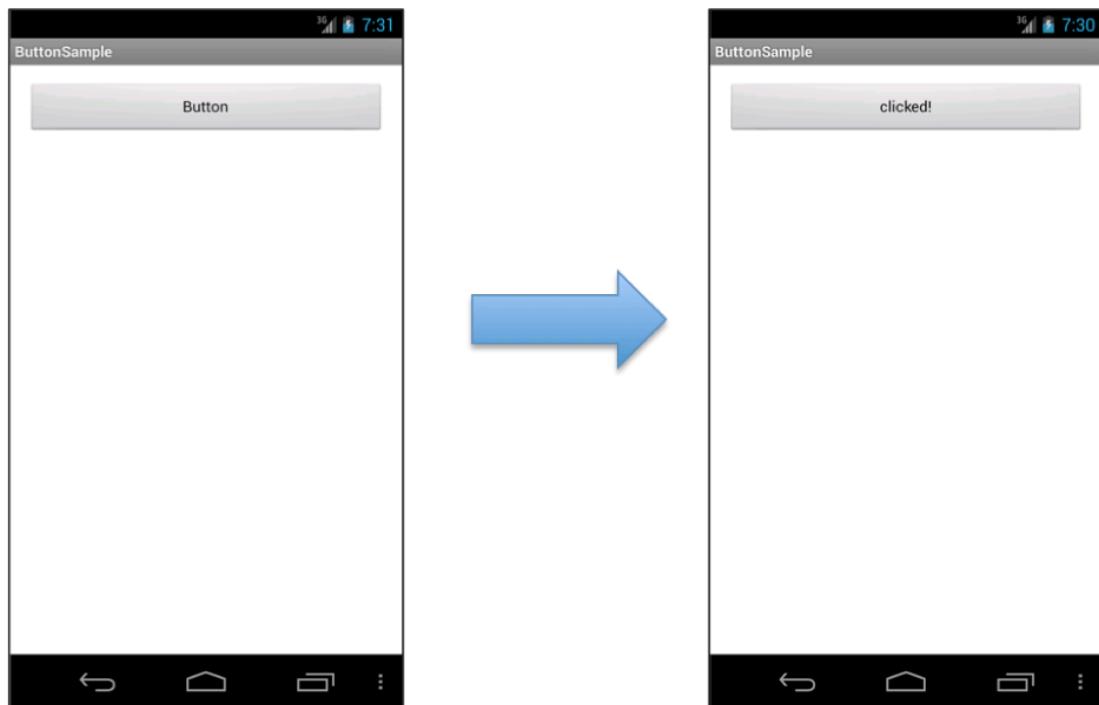
1. 対象アプリケーション起動までの処理

1. ホームボタン押下
2. Apps タブの選択
3. アプリケーション一覧画面をスクロールし、対象アプリケーションを探す
4. 対象アプリケーションをクリックし起動



2. 対象アプリケーション起動後の処理

1. Text プロパティが"Button"を指定して Button を取得する
2. Button クリック
3. Button の Text プロパティが"clicked!"になっていることを確認する



リスト 8.7: ButtonSampleUITest.java

```
1: package com.example.buttonsample;
2:
3: import com.android.uiautomator.core.UiObject;
4: import com.android.uiautomator.core.UiObjectNotFoundException;
5: import com.android.uiautomator.core.UiScrollable;
6: import com.android.uiautomator.core.UiSelector;
7: import com.android.uiautomator.testrunner.UiAutomatorTestCase;
8:
9: public class ButtonSampleUITest extends UiAutomatorTestCase {
10:
11:     private static final String APPS_TAB_NAME = "Apps";
12:     private static final String TARGET_APP_NAME = "ButtonSample";
13:     private static final String TARGET_APP_PACKAGE = "com.example.buttonsample";
14:     private static final String BUTTON_DESCRIPTION = "Button";
15:     private static final String BUTTON_LABEL_AFTER = "clicked!";
16:
17:     public void testButtonClick() throws UiObjectNotFoundException {
18:         // ホームボタンを押してアプリケーション一覧を表示
19:         getUiDevice().pressHome();
20:
21:         // APPS タブを選択
22:         UiObject allAppsButton = new UiObject(new UiSelector().description(APPS_TAB_NAME));
23:
24:         allAppsButton.clickAndwaitForNewWindow();
25:         UiObject appsTab = new UiObject(new UiSelector().text(APPS_TAB_NAME));
26:         appsTab.click();
27:
28:         // 対象アプリケーションのアイコンが表示するまでスクロールする
29:         UiScrollable appsView = new UiScrollable(new UiSelector().scrollable(true));
30:
31:         // 対象アプリケーションが見つかったら起動する
32:         UiObject targetApp = appsView.getChildByText(
33:             new UiSelector().className(android.widget.TextView.class.getName()),
34:             TARGET_APP_NAME);
35:         targetApp.clickAndwaitForNewWindow();
36:
37:         // 指定したパッケージ名のアプリの存在を確認する
38:         UiObject appValidation = new UiObject(new UiSelector().packageName(TARGET_APP_PACKAGE));
39:         assertTrue(appValidation.exists());
40:
41:         // Button 取得
42:         UiObject button = new UiObject(new UiSelector().description(BUTTON_DESCRIPTION));
43:
44:         // Button クリック
45:         button.click();
46:         sleep(1000);
47:
48:         String actual = button.getText();
49:         assertEquals("Button の Text 確認", BUTTON_LABEL_AFTER, actual);
50:
51:     }
52:
53: }
```

8.7.7 プロジェクトのビルド

ビルド手順

1. 開発環境の対象 API のターゲット ID を確認する
2. android コマンドで設定ファイルを作成する
3. ant コマンドでテストプロジェクトをビルドする
4. jar ファイルをデバイスに転送する

1. 開発環境の対象 API のターゲット ID を確認する

android list target コマンドで API の TargetID を確認します。

```
$ android list target
.
.
.
-----
id: 8 or "android-17"
Name: Android 4.2.2
Type: Platform
API level: 17
Revision: 2
Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default), WVGA854, WXGA720, W
ABIs : armeabi-v7a
```

2. android コマンドで設定ファイルを作成する

テストプロジェクトは ant build コマンドを使って作成します。そのために必要な構成ファイルを作成するためターミナルから android コマンドを実行します。

```
android create uitest-project -n <name> -t <target> -p <path>
```

表 8.1 コマンドオプション

項目	設定値
<name>	プロジェクト名
<target>	「android list target」で表示される対象 API の Target ID
<path>	対象プロジェクトのパス

例)

```
$ android create uitest-project -n UIAutomatorSample -t 8 -p .
Added file ./build.xml
$ ls
bin                      build.xml          local.properties      project.properties
```

3. ant コマンドでテストプロジェクトをビルドする

build.xml が保存されている場所に移動し ant build コマンドを実行する

```
$ ant build
```

例)

```
$ ant build
Buildfile: /Users/user/develop/eclipse/workspace/UIAutomatorSample/build.xml

-check-env:
[checkenv] Android SDK Tools Revision 21.1.0
[checkenv] Installed at /Users/user/develop/android/android-sdk-mac_x86
.

.

build:

BUILD SUCCESSFUL
Total time: 2 seconds
```

jar ファイルが作成されていることを確認する

```
$ ls bin/
UIAutomatorSample.jar    classes           classes.dex        classes.dex.d
```

4. jar ファイルをデバイスに転送する

ビルドしてできた jar ファイルを adb コマンドで Android 端末に転送します。
コマンド

```
$ adb push <ビルドして生成された jar ファイル> <転送先ディレクトリ>
```

例)

```
$ adb push ./bin/UIAutomatorSample.jar /data/local/tmp  
154 KB/s (1956 bytes in 0.012s)
```

8.7.8 実行 uiautomator ツールを使ってテストを実行する

adb コマンドで以下のオプションを指定しテストを実行します。

コマンド

```
$ adb shell uiautomator runtest <ビルドして生成された jar ファイル> -c <テストクラス>
```

例)

```
$ adb shell uiautomator runtest UIAutomatorSample.jar -c com.example.buttonsample.ButtonSampleUITest  
INSTRUMENTATION_STATUS: current=1  
INSTRUMENTATION_STATUS: id=UiAutomatorTestRunner  
INSTRUMENTATION_STATUS: class=com.example.buttonsample.ButtonSampleUITest  
.  
. .  
INSTRUMENTATION_STATUS_CODE: 0  
INSTRUMENTATION_STATUS: stream=  
Test results for WatcherResultPrinter=.  
Time: 16.711  
  
OK (1 test)  
  
INSTRUMENTATION_STATUS_CODE: -1
```

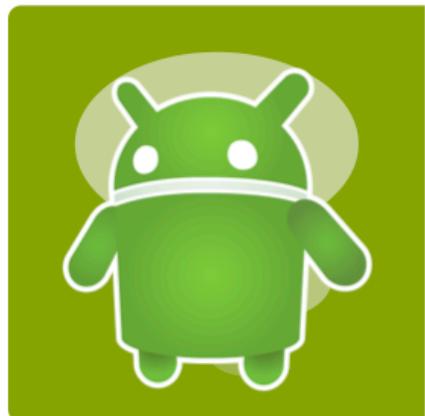
8.8 【実習】uiautomator



サンプルと同じテストを、実際に作成してください。

第9章

コードカバレッジ



第9章 コー
ド・カバレッ
ジ

9.1 概要

概要

- カバレッジとは

- emma



本章では以下のことについて学習します

- カバレッジとは
- emma

9.2 カバレッジとは

カバレッジとは

- テストの網羅率のこと
- テストコードが対象アプリケーションのプログラムコードを、どれだけ網羅したかを示す
- カバレッジの基準にはC0, C1, C2の三種類が用意されている



テストコードが対象アプリケーションのプログラムコードを、どれだけ網羅したかを示すコードの網羅率のことです。

カバレッジの基準には C0, C1, C2 の三種類が用意されています。

9.2.1 カバレッジの基準

C0: 命令網羅（ステートメントカバレッジ）

コード内の全てのステートメント（命令文）を少なくとも 1 回は実行する。

C1: 分岐網羅（ブランチカバレッジ）

コード内のすべての分岐（判定条件）で、少なくとも Yes/No が分かれるようにする。

C2: 条件網羅（コンディションカバレッジ）

全ての判定条件の組み合わせを実行する。C1 が判定結果の分岐なのに対し、C2 では判定条件の組み合わせを網羅する。

9.2.2 emma とは

AndroidSDK に標準で付属しているカバレッジツールです。emma は Android JUnit プロジェクトを実行し、カバレッジ計測をします。

- emma
 - AndroidSDK に標準で付属しているカバレッジツール
 - ant コマンドで Android JUnit プロジェクトを実行する
 - 計測結果は html、xml、text などの形式で出力
 - emma を実行するには、エミュレータまたは、root 取得済の端末が必要

9.3 emma Sample

emma Sample

■ ButtonSampleとButtonSampleTestを使ってemmaツールの使い方を説明



■ 対象プロジェクト

プロジェクト名	説明
ButtonSample	テストターゲット
ButtonSampleTest	テストプロジェクト

9.3.1 emma の使い方

emma は ant コマンドで実行されるため、ant によるビルドが必要です。ビルドはテスト対象のプロジェクトとテストプロジェクト両方行います。

手順

1. 対象プロジェクトのビルドとインストール
2. テストプロジェクトのビルドとインストールとテスト実行
3. 実行結果レポートの確認

9.3.2 手順 1. 対象プロジェクトのビルドとインストール

テスト対象のプロジェクトのビルドとインストールを以下の手順で行います。

1. android update コマンドを実行し、既存のプロジェクトの build.xml を作成
2. ant clean コマンドで、クリーンアップする
3. ant コマンドで対象プロジェクトのビルド、インストールを行う

build.xml の作成

```
$ android update project --path .
```

プロジェクトのビルド、インストール

```
$ ant clean  
$ ant emma debug install
```

```
$ cd [eclipse workspace]/workspace/ButtonSample  
$ android update project --path .  
$ ant clean  
$ ant emma debug install  
Buildfile: /Users/a_kobayashi/develop/work/test/ButtonSample/build.xml  
  
emma:  
  
-set-mode-check:  
  
-set-debug-files:  
.  
. .  
. .  
  
install:  
[echo] Installing /Users/a_kobayashi/develop/work/test/ButtonSample/bin/MainActivity  
[exec] 2096 KB/s (187661 bytes in 0.087s)  
[exec]      pkg: /data/local/tmp/MainActivity-debug.apk  
[exec] Success  
  
BUILD SUCCESSFUL  
Total time: 17 seconds
```

9.3.3 手順 2. テストプロジェクトのビルドとインストールとテスト実行

テストプロジェクトのビルドとインストールを以下の手順で行います。test オプションを追加しテストを実行すると、エミュレータの画面ではテストが実行され、カバレッジ計測のレポートファイルが出力されます。

1. android update コマンドを実行し、既存のプロジェクトの build.xml を作成
2. ant clean コマンドで、クリーンアップする
3. ant コマンドでテストプロジェクトのビルドとインストール
4. エミュレータ上で、テストが実行される

build.xml の作成

```
$ android update test-project -m ../ButtonSample/ -p .
```

プロジェクトのビルド、インストール、テスト

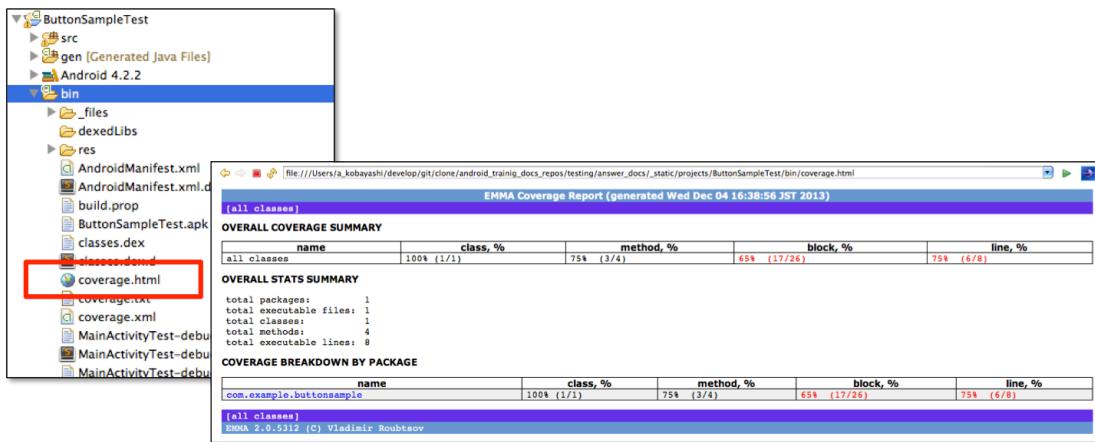
```
$ ant clean  
$ ant emma debug install test
```

```
$ cd [eclipse workspace]/ButtonSampleTest  
$ android update test-project -m ../ButtonSample/ -p .  
$ ant clean  
$ ant emma debug install test  
Resolved location of main project to: c:/android_training/workspace/ButtonSample  
Updated project.properties  
Updated local.properties  
Added file ./build.xml  
Updated file ./proguard-project.txt  
Updated ant.properties  
. . .  
[delete] Deleting: /Users/a_kobayashi/develop/work/test/ButtonSampleTest/bin/coverage.em  
[delete] Deleting: /Users/a_kobayashi/develop/work/test/ButtonSample/bin/coverage.em  
[echo] Saving the coverage reports in /Users/a_kobayashi/develop/work/test/ButtonSampl  
  
BUILD SUCCESSFUL  
Total time: 39 seconds
```

9.3.4 手順 3. 実行結果レポートの確認

実行結果レポートを確認します。

1. レポートファイル「coverage.html」がカレントディレクトリに保存される（今回のケースでは"ButtonSampleTest/bin"）
2. レポートファイルをブラウザで開くと、カバレッジ計測結果が確認できる

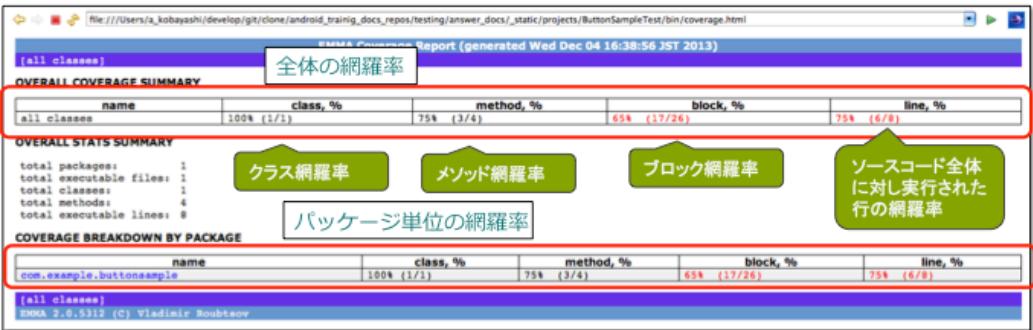


9.3.5 レポートの見方

レポートの見方（1）



- トップページ
 - アプリケーション全体の網羅率
 - パッケージ単位の網羅率



The screenshot shows the EMMA Coverage Report window. At the top, it displays 'EMMA Coverage Report (generated Wed Dec 04 16:38:56 JST 2013)' and '(all classes)'. A red box highlights the 'OVERALL COVERAGE SUMMARY' table:

name	class, %	method, %	block, %	line, %
all classes	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)

Below this, a green box labeled 'クラス網羅率' covers the 'OVERALL STATS SUMMARY' section, which includes statistics like total packages (1), total executable files (1), etc. A green box labeled 'メソッド網羅率' covers the 'METHOD COVERAGE' section. A green box labeled 'ブロック網羅率' covers the 'BLOCK COVERAGE' section. A green box labeled 'ソースコード全体に対し実行された行の網羅率' covers the 'LINE COVERAGE' section.

At the bottom, another red box highlights the 'COVERAGE BREAKDOWN BY PACKAGE' table for the package 'com.example.buttonsample':

name	class, %	method, %	block, %	line, %
com.example.buttonsample	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)

The bottom of the window shows the footer: '(all classes)' and 'EMMA 3.0.5332 (C) Vladimir Roubtsov'.

184



レポートの見方（2）

- coverage breakdown by XXXをクリックすると下位階層のレポートを確認できる

The screenshot shows the EMMA Coverage Report interface. At the top left, there's a table titled "COVERAGE BREAKDOWN BY PACKAGE" with a red box highlighting the row for "com.example.buttonsample". An arrow points from this row down to the main report area. The main area is titled "EMMA Coverage Report (generated Wed Dec 04 16:38:56 JST 2013)". It contains two tables: "COVERAGE SUMMARY FOR PACKAGE [com.example.buttonsample]" and "COVERAGE BREAKDOWN BY SOURCE FILE". Both tables have columns for name, class, %, method, %, block, %, and line, %. The "COVERAGE SUMMARY" table shows data for MainActivity.java, while the "COVERAGE BREAKDOWN" table shows data for com.example.buttonsample.

COVERAGE BREAKDOWN BY PACKAGE				
name	class, %	method, %	block, %	line, %
com.example.buttonsample	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)

COVERAGE SUMMARY FOR PACKAGE [com.example.buttonsample]				
name	class, %	method, %	block, %	line, %
MainActivity.java	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)

COVERAGE BREAKDOWN BY SOURCE FILE				
name	class, %	method, %	block, %	line, %
MainActivity.java	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)

レポートの見方（3）



- ソースコードをクリックするとソースコード全体のうち、実行された行を確認することができる
 - 緑：実行したコード
 - 赤：実行されていないコード

COVERAGE SUMMARY FOR SOURCE FILE [MainActivity.java]				
name	class, %	method, %	block, %	line, %
MainActivity.java	100% (1/1)	75% (3/4)	65% (17/26)	75% (6/8)
COVERAGE BREAKDOWN BY CLASS AND METHOD				
name	class, %	method, %	block, %	line, %
class MainActivity	100% (1/1)	75% (3/4)	45% (17/38)	75% (6/8)
onCreateOptionsMenu (Menu): boolean		0%	(0/3)	0% (0/2)
MainActivity (): void		100% (1/1)	100% (3/3)	100% (1/1)
onClickButton (View): void		100% (1/1)	100% (7/7)	100% (2/2)
onCreate (Bundle): void		100% (1/1)	100% (7/7)	100% (3/3)
<pre>1 package com.example.buttonsample; 2 3 import android.app.Activity; 4 import android.os.Bundle; 5 import android.view.Menu; 6 import android.view.View; 7 import android.widget.Button; 8 9 public class MainActivity extends Activity { 10 11 @Override 12 protected void onCreate(Bundle savedInstanceState) { 13 super.onCreate(savedInstanceState); 14 setContentView(R.layout.activity_main); 15 } 16 17 @Override 18 public boolean onCreateOptionsMenu(Menu menu) { 19 getMenuInflater().inflate(R.menu.main, menu); 20 return super.onCreateOptionsMenu(menu); 21 } 22 23 public void onClickButton(View v){ 24 ((Button)v).setText(getString(R.string.clicked)); 25 } 26 27 }</pre>				

9.4 【実習】カバレッジ



サンプルと同じカバレッジレポートを、実際に作成して下さい。

第10章

ビルドツールとCIツール



10.1 概要

概要

- ビルドプロセスの自動化
- 繙続的インテグレーション



本章では以下のことを学習します

- ビルドプロセスの自動化
- 繙続的インテグレーション

10.2 ビルドプロセスの自動化

ビルドプロセスの自動化



- ビルドプロセスとは
 - アプリケーションをビルドし、実行ファイルを作成するまでの一連の作業のこと
- ビルドプロセスを自動化する
 - Apache Ant
 - Maven
 - Gradle

10.2.1 ビルドプロセスとは

アプリケーションをビルドし、実行ファイルを作成するまでの一連の作業のことをビルドプロセスといいます。広義の意味では、単純に実行ファイルを作成するだけでなくリリース作業や、開発環境の構築、検証も含まれます。

環境構築やリリースなどの作業には以下のようなものがあり、これらの作業は正しい手順で行う必要があります。

- セットアップ
- 依存ライブラリの解決
- コンパイル
- ビルド
- メイク
- テスト
- パッケージング
- デプロイ
- ドキュメンテーション

10.2.2 ビルドツール

ビルドプロセスを自動化するためのツールです。ソースファイルや依存ライブラリが多数存在すると、ビルド手順はさらに複雑化され、正しい順序での統合処理が困難になります。ビルドツールを使うことで、統一した開発環境の提供や、正しい手順でのビルド、デプロイなどができます。

代表的なビルドツール

- Apache Ant
- Maven
- Gradle

Apache Ant

- Java アプリケーションプロジェクト管理ツール
- ビルド手順を xml で記述する
- ライブラリ依存関係の解決はしない
- ソースコードのコンパイル、テスト、Javadoc 生成、テストレポート生成、プロジェクトサイト生成、JAR 生成、WAR ファイル、EAR ファイル生成など様々な機能が用意されている
- ビルド手順を xml で定義するため、記述量が膨大になり可読性が低い

Maven

- Apache ライセンスにて配布されているオープンソースソフトウェア
- Ant のようなビルド管理以外にもライブラリ依存管理に対応
- pom.xml というファイルでプロジェクトに関する必要な情報を管理している
- ライブラリのリモートリポジトリが提供されており、自動インストールが可能
- Ant と同様にビルド手順や依存関係解決を xml で定義するため、記述量が膨大になり可読性が低い

Gradle

- 設定ファイルを Groovy(JVM 上で動くスクリプト言語) で記述する
- 記述言語に Groovy を採用しているため、プログラミングの感覚でビルドロジックを組み立てることができる
- if 文などの構文が使えるため、複雑な環境構築にも柔軟に対応することができる
- プログラミング言語なので可読性が高い
- Groovy は Java の構文を扱えることもできるため、学習コストが低い
- ライブラリの管理に Maven リポジトリを採用しているため、多くのライブラリに対応している

10.3 継続的インテグレーション

継続的インテグレーション

- 継続的インテグレーション（以下CI）とは
 - ソフトウェア開発における一連の作業を定期的に行うこと
 - CIツールと使って自動化する
 - Jenkins
 - Buildbot
 - Bamboo
 - CruiseControl
 - Apache Continuum



10.3.1 継続的インテグレーションとは

ソフトウェア開発における一連の作業を定期的に行うことを継続的インテグレーションといいます。インテグレーション（統合）を継続的かつ定期的に行うことで以下のようないくつかの効果が期待できます。

- ビルドエラーが発生するコミットの早期発見
- テストのスケジューリングによるバグの早期発見
- デグレードの早期発見

10.3.2 CIツール

一般的にCIはCIツールを使って自動化し、ナイトリービルドや人為的なミスの防止に役立てます。

CIツールの利点

- バージョン管理サーバーと連携し、最新コードが取得できる

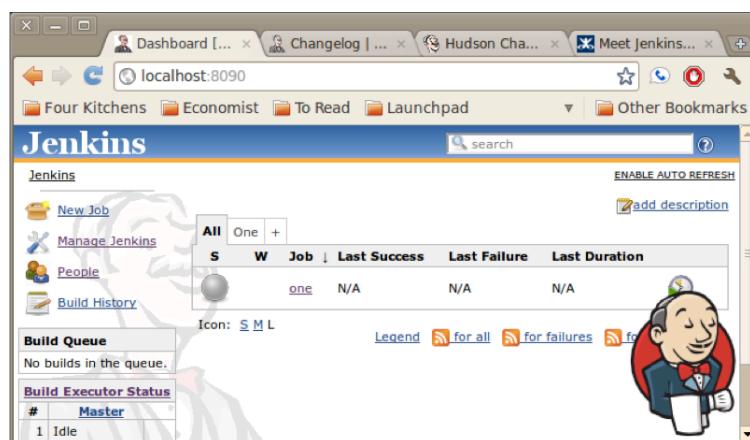
- コミット毎に自動ビルド、自動テストを実施する
- 結果レポートの作成
- 成果物を管理し、容易に取得できる

主なCIツール

- Jenkins
- Buildbot
- Bamboo
- CruiseControl
- Apache Continuum

Jenkins

- Javaで書かれたオープンソース継続的インテグレーションツール
- Apache TomcatなどのServletで動作しているサーバベースシステム
- Subversion、Git、Mercurialなどその他多くのバージョン管理システムツールに対応
- Apache AntやApache Mavenなどと連携することができる



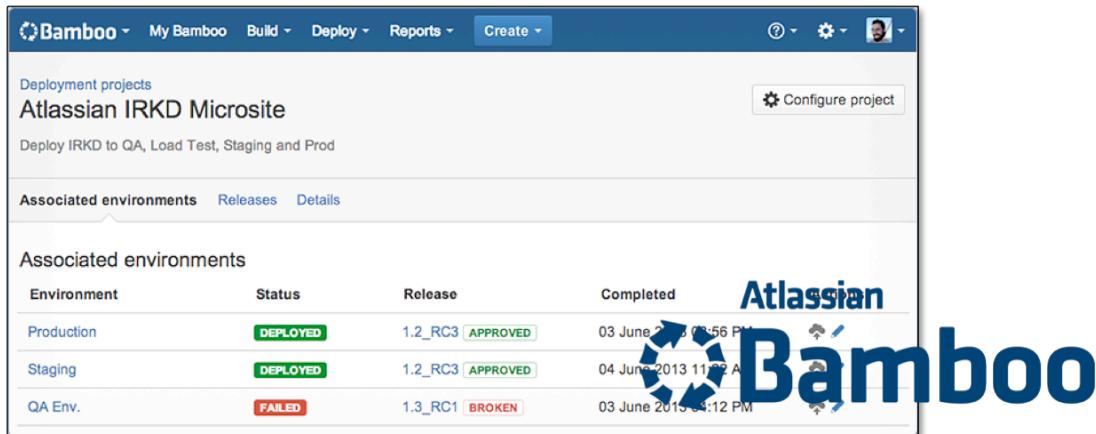
Buildbot

- Pythonで書かれたオープンソースの継続的インテグレーションツール
- Twistedフレームワークを使ったサーバアプリケーション
- Subversion、Git、Mercurialなどその他多くのバージョン管理システムツールに対応
- Mozilla、Chromiumなど多くのプロジェクトで採用されている



Bamboo

- Atlassian 社が提供する有償の CI サーバー
- 多くの言語とビルドツール、テストツールに対応している
- Atlassian 社が提供するその他の Web サービスと容易に連携できる
- クラウド版が用意されており、自前で CI サーバを建てる必要がない
- Jenkins からの移行にも対応している



10.3.3 CIツールの導入を検討する

上記のように様々な CIツールが存在しますが、提供される機能などはそれぞれ異なります。そのため、導入の際は事前に十分に検討する必要があります。

検討項目

- 対応言語
- BTS

- 対応 OS
- 有料・無料
- サポート
- バージョン管理対応 (Git、Mercurial、svn...)
- ビルドツール

第 11 章

まとめ

11.1 トレーニングの振り返り

このトレーニングでは以下のことについて、学習しました

- 第 1 章 テストの自動化
- 第 2 章 JUnit の基本
- 第 3 章 チュートリアル
- 第 4 章 Unit Test
- 第 5 章 JUnit UI Testing
- 第 6 章 JUnit の応用技術
- 第 7 章 ストレステスト
- 第 8 章 シナリオテスト
- 第 9 章 コードカバレッジ
- 第 10 章 ビルドツールと CI ツール
- 第 11 章 まとめ

11.2 Android に関する書籍

タイトル	著者	出版社
図解 Android プラットフォーム開発入門	橋爪香織 小林明大 …他	技術評論社
Android Hacks	株式会社ブリリアントサービス	オライリー
Android UI Cookbook for 4.0 ICS	あんざい ゆき	インプレス ジャパン
入門 Android 2 プログラミング	Mark Murphy	翔泳社
Android Layout Cookbook	あんざい ゆき	インプレス ジャパン

11.3 Android に関する情報提供元

- インターネット
 - 技術資料・ツール・ソース
 - * Android Developers
 - <http://developer.android.com/index.html>
 - * Open Handset Alliance
 - <http://www.openhandsetalliance.com/>
 - コミュニティ
 - Android SDK Japan
 - * <https://groups.google.com/group/android-sdk-japan>
 - Android Developers
 - * <http://groups.google.com/group/android-developers>

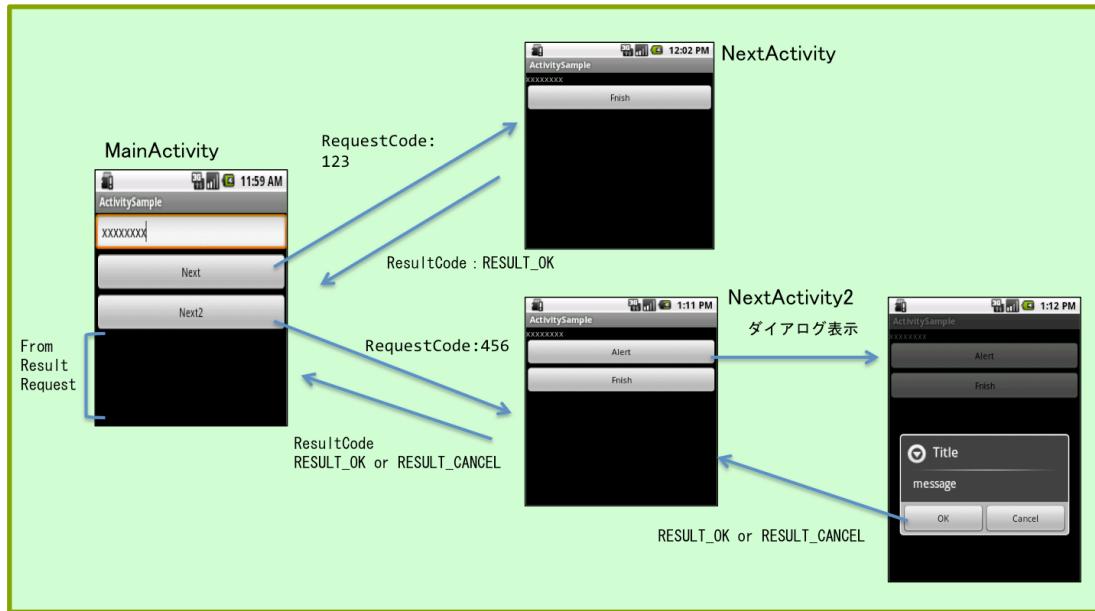
11.4 OESF 公認 Android トレーニング

- アプリケーション開発
 - Android プログラミング入門 (難易度 :)
 - Android アプリケーション開発入門 ()
 - Android アプリケーション開発応用 ()
 - Android 応用 WebAPI 開発 ()
 - Android 応用 タブレットアプリケーション開発入門 ()
 - Android UI デザイン入門 ()
 - 品質向上！ Android アプリケーションテスト ()

- 組み込み
 - Android 組み込み開発 基礎コース "Armadillo-440 編"

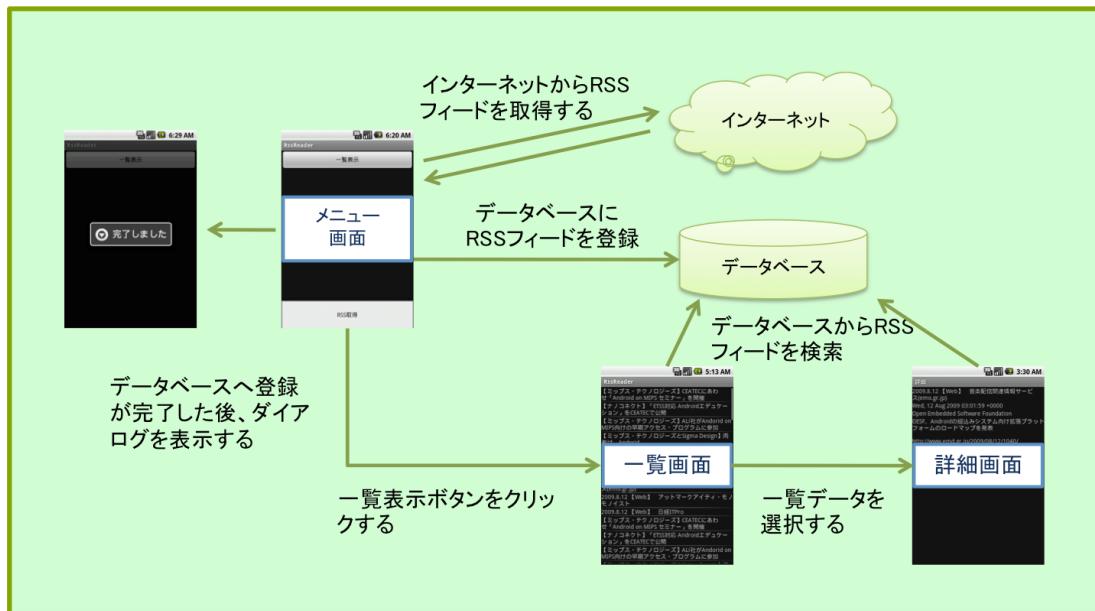
11.4.1 Androidプログラミング入門

- 基本的なアプリケーション開発に必要なプログラミング技術を習得する
- ユーザインターフェースの使い方や画面遷移の仕方



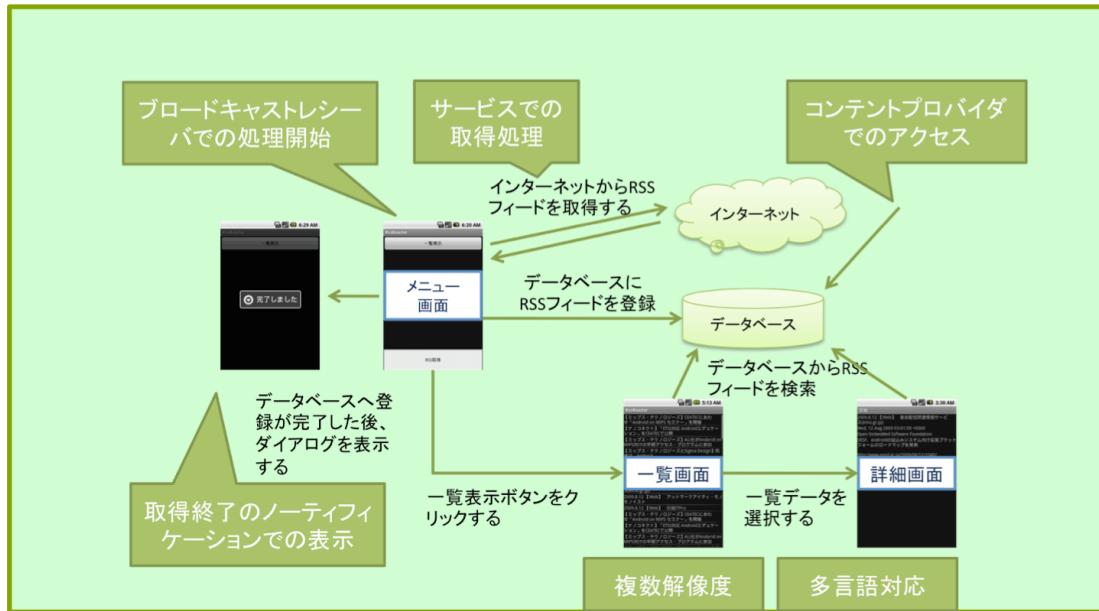
11.4.2 Android アプリケーション開発入門

- Android の基本的な知識から本格的なアプリケーション開発



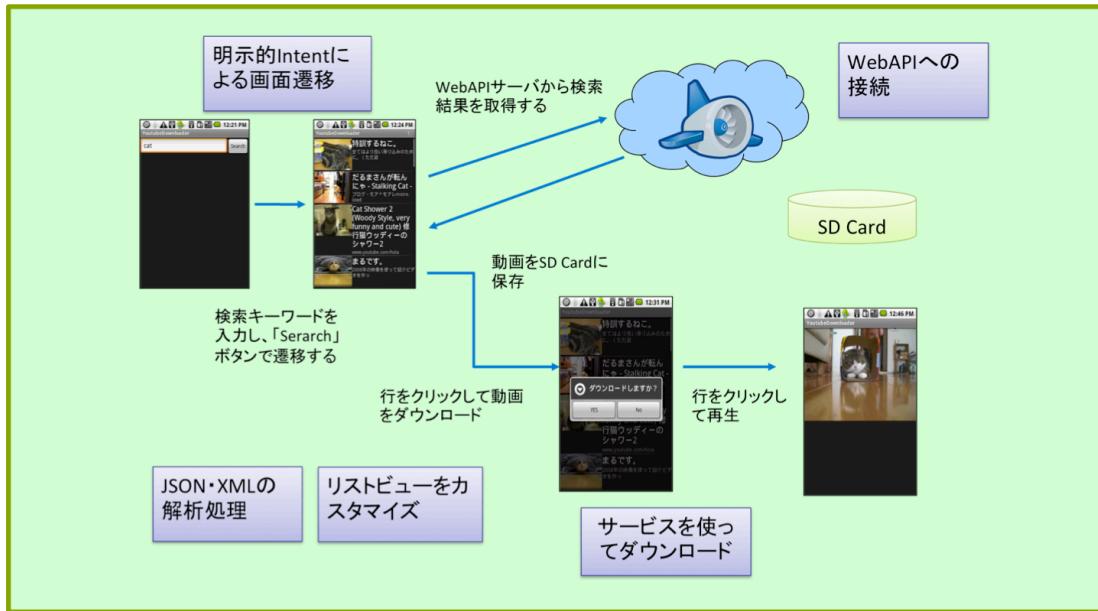
11.4.3 Android アプリケーション開発応用

- ・入門編で作成したアプリケーションに手を加え、より快適なものに仕上げる
- ・Activity のタスク管理やプロファイリングなど開発における高度なサイド技術の習得



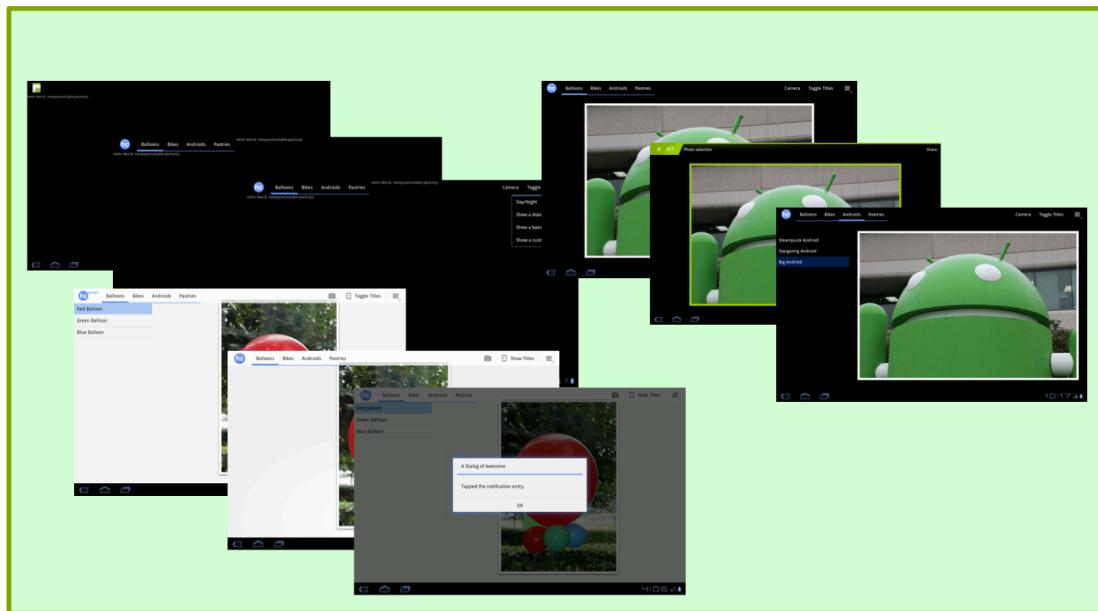
11.4.4 Android応用WebAPI開発

- 非同期処理、プロセス間通信など開発において重要で難易度の高い技術の習得
- GAEサーバを使用した動画ダウンロードアプリケーションの開発



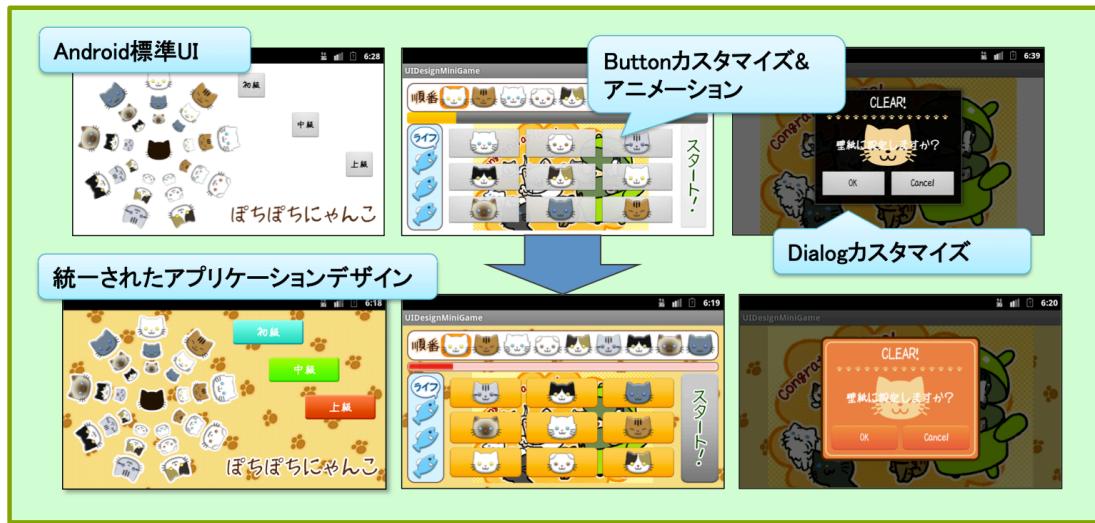
11.4.5 Androidタブレット開発コース

- タブレット基本的な知識、新機能、開発手法のベストプラクティスの習得
- ActionBar、Fragmentなどを駆使して画像ビューアを作成する



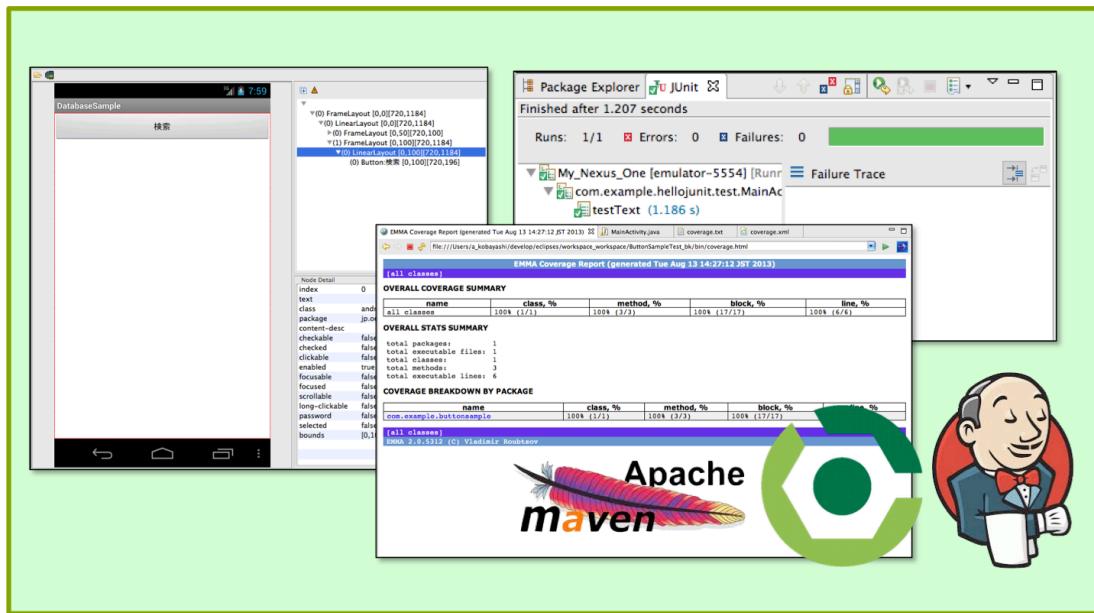
11.4.6 Android UI デザイン入門

- Androidで独自UIを実現させるためのカスタマイズ手法を習得する
 - UI単体のカスタマイズからアプリケーション全体で統一されたデザインの実現
 - ソースコードの修正は殆ど無いため、デザイナも受講生対象
- 実習用アプリケーションをGooglePlayで配布中
 - <https://play.google.com/store/apps/details?id=biz.oesf.pochipochinyanko>



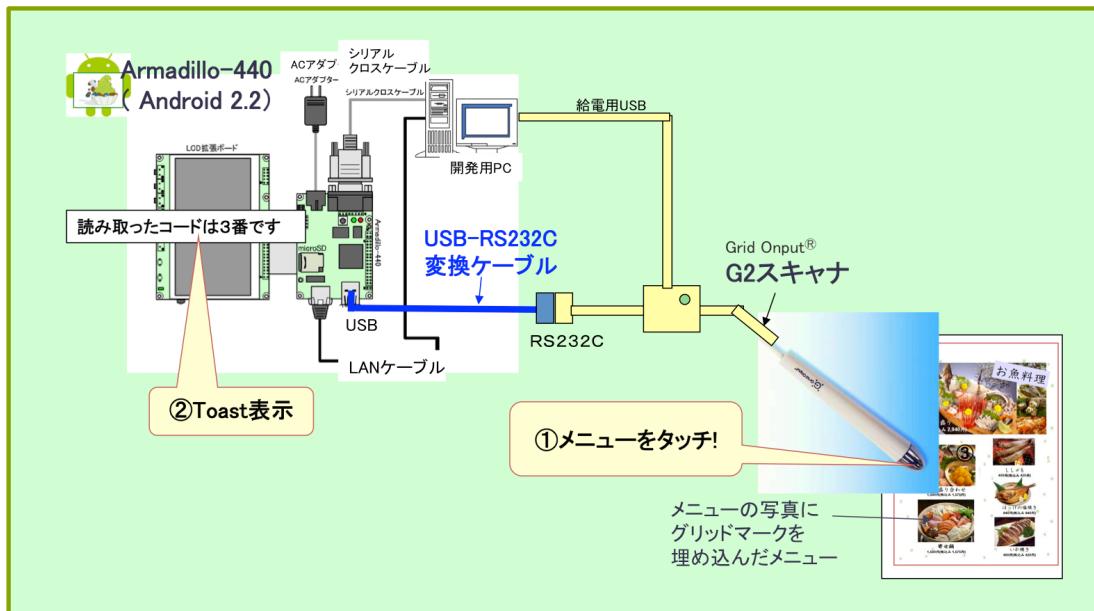
11.4.7 品質向上！Android アプリケーションテスト

- Androidで動作するアプリケーション開発に必要なテスティング技法を習得する
- JUnitベースのテストや、Android特有のテスティングフレームワークAPIと、各種テスト（ストレステスト、シナリオテスト、カバレッジテスト、など）の効率的な使い方を習得する



11.4.8 Android組み込み開発基礎コース "Armadillo-440編"

- Armadillo-440を使ったシリアルデバイスアプリケーション作成





本ドキュメントは株式会社リーディング・エッジ社が作成しています。
<http://www.leadinge.co.jp/>



本ドキュメントは株式会社リーディング・エッジ社及び株式会社カサレアルが
バージョンアップ対応のために改変しています。
<http://www.leadinge.co.jp/>
<http://www.casareal.co.jp/>



OESFトレーニングドキュメントはクリエイティブ・コモンズ
表示 3.0 非移植 ライセンスの下で提供されています。

このドキュメントの内容の一部は、Google が作成、提供しているコンテンツをベースに変更したもので、クリエイティブ・コモンズの表示 3.0 ライセンスに記載の条件に従って使用しています。

品質向上！ Android アプリケーションテスト

2013 年 08 月 03 日 v1.0.0 版発行

2014 年 03 月 07 日 v1.1.0 版発行

発行所

(C) 2013-2014 Open Embedded Software Foundation