

JUnit 実践入門

体系的に学ぶユニットテストの技法

Watanabe Shuji
渡辺修司
〔著〕

4フェーズテスト

```
public class FourPhaseTest {  
    @Test  
    public void testCase()  
        throws Exception {  
        // 1.初期化  
        Calc sut = new Calc();  
        int expected = 7;  
        // 2.テストの実行  
        int actual = sut.add(3, 4);  
        // 3.アサーション  
        assertThat(actual,  
            is(expected));  
        // 4.（必要ならば）終了処理  
        sut.shutdown();  
    }  
}
```

アサーション

```
assertThat(actual, is(expected));  
fail("未実装");
```

CoreMatchers

is	equalsによる比較
nullValue	null
not	Matcherの否定
notNullValue	非null
sameInstance	インスタンスの比較
instanceOf	型の比較

JUnitMatchers

hasItem	要素の比較
hasItems	要素の比較（複数）

ルール

TemporaryFolder	一時ファイル
ExternalResource	外部リソース
Verifier	事後検証
ErrorCollector	エラーの収集
ExpectedException	例外の検証
Timeout	タイムアウト
TestWatcher	テストの監視
TestName	テストケース名

```
public class RuleTest {  
    @Rule  
    public TestRule rule  
        = new SomeRule();  
    @ClassRule  
    public static TestRule RULE  
        = new SomeRule();  
}
```

アノテーション

@Test	テストケース
@Ignore	テストの実行を除外
@Before	初期化
@After	後処理
@BeforeClass	初期化／クラスごと
@AfterClass	後処理／クラスごと
@RunWith	テストランナー

巻頭付録 JUnitチートシート

構造化テスト

```
@RunWith(Enclosed.class)
public class EnclosedTest {
    public static class XXの場合 {
        @Before
        public void setUp() {
        }
        @Test
        public void testCase() {
        }
    }
    public static class YYの場合 {
        @Before
        public void setUp() {
        }
        @Test
        public void testCase() {
        }
    }
}
```

パラメータ化テスト

```
@RunWith(Theories.class)
public class ParameterizedTest {
    @DataPoints
    public static int[] PARAMS
        = {1, 2, 3, 4};
    @Theory
    public void test(int x) {
    }
}
```

カテゴリ化テスト

```
@Category(DbTests.class)
public class CategorizedTest {
    @Test
    @Category(SlowTests.class)
    public void testCase() {
    }
}
```

mockitoを利用したモック／スタブ

```
@Test
public void 戻り値を返すスタブ() throws Exception {
    List<String> stub = mock(List.class);
    when(stub.get(0)).thenReturn("Hello");
    assertThat(stub.get(0), is("Hello"));
}

@Test(expected = IndexOutOfBoundsException.class)
public void 例外を送出するスタブ() throws Exception {
    List<String> stub = mock(List.class);
    when(stub.get(0)).thenThrow(new IndexOutOfBoundsException());
    stub.get(0);
}

@Test
public void モックによる検証() throws Exception {
    List<String> mock = mock(List.class);
    mock.clear();
    mock.add("JUnit");
    mock.add("JUnit");
    verify(mock).clear();
    verify(mock, times(2)).add("JUnit");
    verify(mock, never()).add("mockito");
}
```

本番は、小社刊『WEB+DB PRESS Vol.69』の特集2「JUnit実践入門」をもとに、大幅に加筆と修正を行いました。

本書記載の内容に基づく運用結果について、著者、ソフトウェアの開発元および提供元、株式会社技術評論社は一切の責任を負いかねますので、あらかじめご了承ください。

本書に記載されている会社名・製品名は、一般に各社の登録商標または商標です。本文中では、™、©、®マークなどは表示しておりません。

推薦のことば

みなさんは、JUnitでユニットテストを書いていますか？

- ・大事だとはわかっているんだけど、面倒そうだし、最初の一歩が難しい
- ・JUnitを調べ始めるとき、情報の多くは古いバージョンのものばかり
- ・JUnit 4になって以降、どんな機能が追加されたのか、よくわかっていない

そういう人も多いのではないでしょうか。Javaのテスティングフレームワークとして事実上のデファクトスタンダードの地位を占めているJUnitは、なにぶん長い歴史を持っていますので、触れたことのある人は膨大な数になります。しかし、日本語のまとまった情報は意外と少なく、本家JUnitが進化するに従って経年劣化を起こしてしまっているというのが現状です。

そのような状況が変わります。渡辺さんの「JUnit便り」が札幌から全国へ届く日がやってきたのです。

本書は、初学者の方でも、上級者の方でも読み応えがあるような構成になっています。

初学者にとって、本書はJUnitを使ったユニットテストの良質な入門書となります。自習用の問題も付いているので、理解の段階を一步一步上り、一人からでも学習を始めることができます。新人エンジニアを教育する立場にある人にとっては、「これを読んでおいて」と渡せる本にもなるでしょう。

上級者にとって、本書はJUnit 4の基礎から応用への扉を開く、JUnit 4の俯瞰的な情報源という側面を見せます。JUnit 4の新機能、拡張方法、設計思想を知ることができます。しかも、DBのテスト、Androidのテスト、カバレッジなど応用的なトピックも扱っています。

さらに注目すべきは、ユニットテスト設計の語彙の豊富さです。「フレッシュフィックスチャ」「スタブ」「モック」「スパイ」といったユニットテストの語彙が登場する本は、まだ日本には少ないのではないでしょうか。

良い仕事は、良い道具を知ることから始まります。

そして、Javaのとびきり良い道具のひとつがJUnitです。

JUnitはJavaエンジニアが時間の大部分を共に過ごす相棒なのです。
JUnitの世界へようこそ。

テスト駆動開発者 和田卓人

はじめに

本書はJavaプログラマを対象としたJUnitによるユニットテストの実践ガイドです。

ユニットテストの基本概念から、テストコードの記述方法や拡張機能まで、JUnitに関する内容をほぼ網羅しました。そして、テスト駆動開発などユニットテストを基盤とする開発手法やツールも紹介します。さらに、本書で解説するユニットテスト技法を実践し、身に付けるための演習問題を多く収録しました。

このような本を執筆した背景には、テスト駆動開発(TDD)を体験して学ぶワークショップ「TDD Boot Camp」との関わりがあります。TDD Boot Campを地元札幌で開催した経験から、ユニットテストの概念から実践するまでをまとめた書籍の必要性を強く感じました。それは、ユニットテストを最大限に活用する方法がテスト駆動開発であると気付いたからです。

プログラミングは楽しい活動です。「Hello World」でも、業務アプリケーションでも、プログラムが意図したように動くことがプログラマにとっての喜びです。それが仕事となるプログラマは、幸せな職業だと思います。

同時にプログラマは職人でもあります。仕事としてプログラムを作るならば、その品質に責任を持たなければなりません。その品質、特に内部品質を支える技術がユニットテストです。ユニットテストとリファクタリングを実践することで、自信を持ってプログラムをリリースできます。そして、動作するきれいなコードを手に入れることができます。

筆者もテストが嫌いでした。テストは面倒で単調な作業というイメージから、苦手意識を持っていました。しかし、テストを学び、テストを書けるようになるとテストも楽しくなりました。プログラミングと同じです。

ユニットテストは、プログラマが、プログラマ自身のために行う、プログラムを使ったテストです。きれいなコードを書きたい、価値のあるソフトウェアを提供したいと思うならば、ユニットテストを行ってください。

本書が多くのプログラマに読まれ、ユニットテストが楽しく行われることを願います。

2012年10月 渡辺修司

本書について

対象とする読者

本書で対象とする読者は次のような Java プログラマです。

- ・ユニットテストや JUnit をこれから学びたい方
- ・より実践的な JUnit の使い方を学びたい方
- ・JUnit の最新機能を学びたい方
- ・ユニットテストを活用した開発手法を学びたい方

前提となる知識

ユニットテストや JUnit に関して、前提となる知識はありません。ただし、基本的な文法、クラス、インターフェース、アノテーションなど、Java の基礎的な知識があることを前提としています。また、一部の章では、データベースと JDBC や Android アプリケーション開発などを扱います。それらの章では、該当する技術の基礎的な知識が必要です。

環境とバージョン

本書では、JDK 6、JUnit 4.10、Eclipse 3.7 Indigo を基本的な開発環境としています。これらの開発環境のセットアップ方法については、巻末の付録 A および付録 B を参照してください。

その他、本書で利用する開発環境やライブラリに関するバージョンの詳細については、付録 F を参照してください。なお、本文中に登場する Eclipse のショートカットコマンドは Windows 環境を前提として記述しています。

本書の構成と利用方法

本書は 5 パート 20 章構成となっています。章ごとに独立した内容を扱っているため、興味のある章から読むことができます。

● Part 1 JUnit入門

Part 1では、本書で初めてユニットテストを学ぶ方が、JUnitとユニットテストの基礎を習得できるように構成されています。

第1章では、チュートリアルを進めながら JUnitの基礎を学びます。JUnitでのユニットテストが初めての方はこの章からお読みください。

第2章では、ユニットテストの位置付けと目的について解説します。また、ユニットテストを実践するうえで重要なパターンを解説します。

第3章では、JUnitのしくみとテストコードの書き方について解説します。

● Part 2 JUnitの機能と拡張

Part 2では、JUnitの持つさまざまな機能を解説します。

第4章では、JUnitにおける値の比較検証について解説します。Matcher APIを活用することで、テストコードの表現力が高まるでしょう。

第5章では、ユニットテストの実行方法について解説します。テストランナーを活用することで、テストの実行方法を柔軟に制御できます。

第6章では、コンテキスト(前提条件や流れ)に着目してテストを分類・整理する方法を解説します。共通の値や状態に着目し、テストクラスを分割して整理します。

第7章では、テストを構成するさまざまな要素を管理する方法について解説します。テストで利用する入力値や期待値を効率良く整理しましょう。

第8章では、テストケースとテストデータを分離し、さまざまなパターンでテストを行う方法について解説します。

第9章では、ルールを使ったJUnitの拡張について解説します。カスタムルールの作り方についても解説します。

第10章では、テストをカテゴリに分け、実行するテストの実行を制御する方法を解説します。

● Part 3 ユニットテストの活用と実践

Part 3では、ユニットテストを支援するライブラリやツールについて紹介します。

第11章では、モックやスタブとして知られるテストダブルについて解説します。モックやスタブを利用することでテストしにくいクラスをテスト

したり、オブジェクトの相互作用に着目したテストを行うことができます。

第12章では、データベースを利用するクラスのテストを扱います。データベースのテストで注意すべき点と、DbUnitについて紹介します。

第13章では、GUIアプリケーションのテストを扱います。Androidを例に、テストしやすいクラス設計とユーザ視点のテストについて紹介します。

第14章では、ユニットテストでコードの実行網羅率(カバレッジ)を測定する方法と、その目的や効果について解説します。

◎ Part 4 開発プロセスの改善

Part 4では、ユニットテストを効果的に行うためのプラクティスを紹介します。

第15章では、自動化されたユニットテストを継続的に行う方法について解説します。ビルドを自動化し、ソースコード管理システムと連動し、ユニットテストが行われる環境を構築します。

第16章では、テストコードを先に書き、インクリメンタルにソフトウェアを開発するテスト駆動開発について解説します。テスト駆動開発を学ぶことで、テスタビリティを意識したクラス設計を行えます。

第17章では、ユーザ視点での受け入れテストを先に書き、ソフトウェアを開発する振舞駆動開発について解説します。ユニットテストだけではカバーできない大きな粒度のテストを自動化できます。

◎ Part 5 実践問題

Part 5では、実践的なユニットテストの演習問題です。さまざまなテストコードを書くことで、本書で学んだことを再確認してください。

第18章は、シンプルな状況でのユニットテストを扱います。JUnitの基本的な機能を使えば、テストコードを書くことができる問題です。

第19章は、値の比較検証やテストデータのセットアップに工夫が必要なテストを扱います。

第20章は、モックやスタブを駆使して行うユニットテストを扱います。

サンプルコードについて

本書では多くのサンプルコードを掲載しています。その多くは、テストコードです。

紙面の都合上、プロダクションコードはほとんど掲載できませんでした。テストコードを実行するためにも、プロダクションコードを含む完全なサンプルコードを、本書のサポートサイトよりダウンロードしてください。サンプルコードは、Eclipseプロジェクトとなっています。

- ・サンプルコードのダウンロード先

<http://www.gihyo.jp/book/2012/978-4-7741-5377-3>

なお、サンプルコードのライセンスはMITライセンスです。サンプルコードは、原則として無償で無制限に利用することができますが、筆者および出版社は利用に関してなんら責任を負いません。あらかじめご了承ください。

謝辞

本書の執筆にあたって、多くの方々に協力していただきました。

推薦のことばを書いていただいた和田卓人さんには、TDD Boot Campを通じてテスト駆動開発の「こころ」を学ばせていただき、本書執筆のきっかけを作っていただきました。本書の企画を相談した後、いろいろとご助言をいただき、ありがとうございます。

また、レビューに参加していただいた阿部慎也さん、井芹洋輝さん、太田健一郎さん、大中浩行さん、末吉剛さん、其田沙梨さん、高谷征芳さん、中山裕貴さん、野崎啓史さん、林田直樹さん、原知愛さん、矢野勉さん、@kyon_mmさん、@shinlogawaさん、ありがとうございました。さまざまな立場からの本書をテストしていただいたおかげで、本書を書き上げることができました。

そして、本書の出版は技術評論社の稻尾さんと緒方さんのお力がなければ、ありませんでした。初めての執筆でいろいろとご迷惑をおかけしましたが、最後までお付き合いいただき、ありがとうございます。

JUnit実践入門[目次]

推薦のことば 和田卓人	iii
はじめに	iv
本書について	v

Part 1 JUnit入門 1

第1章

JUnitチュートリアル

ユニットテストの作成から実行まで	2
------------------------	---

1.1 なぜ、ユニットテストを行うのか?	2
1.2 JUnitとは?	3
バージョン	3
コラム 企業文化と「ユニットテスト」の定義	3
ライセンス	4
1.3 JUnitテストを始めよう	4
チュートリアルの概要	5
プロジェクトを作成する	5
• 新規Javaプロジェクトの作成	5
• ライブラリの追加	6
コラム junit-4.x.jarとjunit-dep-4.x.jar	6
テスト対象クラスを作成する	8
テストクラスを作成する	9
• testソースフォルダの作成	9
• テストクラスの作成	9
• Quick JUnitの利用	11
テストを実行する	11
1.4 テストコードの記述	12
乗算メソッドのテストを作成する	12
• 日本語のメソッド名でわかりやすくする	12
• 値を比較検証する	13
コラム 日本語のメソッド名を使うメリット	14
• 乗算メソッドテストの実行	15
乗算メソッドのテストを追加する	16
• 障害トレースとエラーメッセージの読み方	16
• テストコードの修正	18
除算メソッドのテストを作成する	18
• Calculatorクラスの設計変更	18
ゼロ除算を例外として送出するテストを作成する	20
• 除算メソッドのゼロ除算対応	20
• 例外の送出を検証するテスト	20
チュートリアルの完了	21

第2章

ユニットテスト

何のためにテストするのか	23
--------------	----

2.1 ソフトウェアテストとは?.....23

ソフトウェアテストの特徴	24
テストケースとテストスイート	24
コラム 製造業と建築業とソフトウェア開発	25
ソフトウェアテストの目的	26
ソフトウェアテストの限界	26

2.2 テスト技法.....27

ホワイトボックステストとブラックボックステスト	27
同値クラスに対するテスト	28
境界値に対するテスト	28

2.3 ユニットテストとは?.....29

ユニットテストの特徴	30
ユニットテストの目的	31
ユニットテストのフレームワーク	32

2.4 ユニットテストのパターン.....33

自動化されたテスト——繰り返しつつでも実行できること	33
不安定なテスト——結果が一定でないテストを避けること	34
コラム 実行環境とテスト	34
ドキュメントとしてのテスト——仕様書として読めること	35
コラム テストケースとドキュメント	35
問題の局所化——テスト失敗時に問題を特定しやすいこと	36
不明瞭なテスト——可読性の低いテストコードは避けること	36
独立したテスト——実行順序に依存しないこと	37
コラム シングルトンオブジェクトとユニットテスト	38

第3章

テスティングフレームワーク

ユニットテストを支えるしくみ	39
----------------	----

3.1 テスティングフレームワークとは?.....39

xUnitフレームワーク	39
--------------	----

3.2 JUnitによるユニットテストの記法.....40

テストメソッドのthrows句	40
テストメソッドを簡単に挿入する	41

3.3 可読性の高いテストコードの書き方.....42

テストケース	42
テスト対象	43

コラム 不完全なテストケース

実測値と期待値	44
メソッドと副作用	45

4 フェーズテスト.....	47
テストフィクスチャ.....	47
3.4 比較検証を行うアサーション.....	48
JUnitのアサーション.....	49
3.5 JUnitが提供するアノテーション.....	50
@Test —— テストメソッドを宣言する.....	51
● expected.....	51
● timeout.....	52
@Ignore —— テストの実行から除外する.....	52
@Before —— テストの実行前に処理を行う.....	53
@After —— テストの実行後に処理を行う.....	54
@BeforeClass —— テストの実行前に一度だけ処理を行う.....	54
コラム JUnit 3スタイルのテスト.....	55
@AfterClass —— テストの実行後に一度だけ処理を行う.....	56
3.6 JUnitのテストパターン.....	56
標準的な振る舞いを検証するテスト.....	56
例外の送出を検証するテスト.....	57
コンストラクタを検証するテスト.....	58

Part 2 JUnitの機能と拡張.....59

第4章

アサーション

値を比較検証するしくみ.....	60
------------------	----

4.1 Assertによる値の比較検証.....	60
assertThat —— 汎用的な値の比較検証.....	61
fail —— テストを失敗させる.....	62
そのほかのアサーションメソッド.....	63
4.2 Matcher APIによるアサーションの特徴.....	63
可読性の高い記述.....	64
柔軟な比較.....	65
● カスタムMatcherによる、より柔軟な比較.....	66
詳細な情報の提供.....	66
4.3 Matcher APIの使用.....	67
CoreMatchersが提供するMatcher.....	67
● is.....	67
● nullValue.....	68
● not.....	68
● notNullValue.....	69
● sameInstance.....	69
● instanceOf	70

JUnitMatchersが提供するMatcher.....	70
● hasItem.....	70
● hasItems.....	70
そのほかHamcrestが提供するMatcher.....	71
4.4 カスタムMatcherの作成.....	71
日付の比較検証を行うカスタムMatcherの要件.....	72
カスタムMatcherの作成手順.....	73
● IsDateクラスを作成する.....	73
● ファクトリメソッドを作成する.....	74
● コンストラクタを定義する.....	74
● matchesメソッドを実装する.....	74
● describeToメソッドを実装する.....	76
● テスト失敗メッセージの確認.....	76
カスタムMatcherのメリット.....	77
IsDateによる比較の実行結果.....	78

第5章

テストランナー

テスト実行方法の制御.....	80
5.1 コマンドラインからのJUnitの実行.....	80
テストクラスからテストを実行する.....	81
JUnitCoreクラスのしくみ.....	81
5.2 テストランナーとは?.....	82
テストランナーの設定.....	82
5.3 JUnitが提供するテストランナー.....	83
コラム ユニットテストの実行結果をカスタマイズする.....	83
JUnit4——テストクラスの全テストケースを実行する.....	84
Suite——複数のテストクラスをまとめて実行する.....	84
● テストスイートクラスでテストクラスを束ねる.....	84
● テストスイートクラスの利用.....	85
● テストスイートとビルド支援ツール.....	85
Enclosed——構造化したテストクラスを実行する.....	86
Theories——パラメータ化したテストケースを実行する.....	88
Categories——特定カテゴリのテストクラスをまとめて実行する.....	89
● カテゴリクラスの作成.....	89
● カテゴリクラスの指定.....	89
● カテゴリ化テストの実行.....	89
コラム JUnitのorg.junit.experimentalパッケージ.....	91

第6章

テストのコンテキスト

テストケースの構造化.....	92
6.1 テストのコンテキストとは?.....	92

6.2 テストケースの整理	92
テストクラスの構造化	93
テストケースをグループ化する	94
Enclosedによるテストクラスの構造化	96
6.3 コンテキストのバターン	98
共通のデータに着目する	99
共通の状態に着目する	101
コンストラクタのテストを分ける	103
6.4 テストクラスを横断する共通処理	104

第7章

テストフィクスチャ

テストデータや前提条件のセットアップ	106
--------------------	-----

7.1 テストフィクスチャとは?	106
ユニットテストのフィクスチャ	106
フレッシュフィクスチャ	107
フィクスチャとスローテスト問題	108
共有フィクスチャ	108
7.2 フィクスチャのセットアップバターン	109
インラインセットアップ	110
暗黙的セットアップ	111
生成メソッドでのセットアップ	114
外部リソースからのセットアップ	114
• YAMLを使ったセットアップ	116
コラム Javaと宣言的記法	118

第8章

パラメータ化テスト

テストケースとテストデータの分離	120
------------------	-----

8.1 テストデータの選択	120
どのくらいのテストデータが必要か?	121
• 同値クラスによるテストデータの選択	121
• 組み合わせによるテストデータの選択	122
どのテストデータを選択するか?	124
コラム 妥当な値の範囲を制限する	125
8.2 入力値と期待値のパラメータ化	126
Theories —— パラメータ化テストのテストランナー	126
@Theory —— テストメソッドに指定するアノテーション	128
@DataPoint —— パラメータを定義するアノテーション	129
コラム Parameterizedテストランナー	129
• 複数のテストメソッドがある場合	130
• 複数の引数が定義されている場合	132

● 同じ型の引数が複数定義されている場合	132
● パラメータをifikスチャオブジェクトにまとめる	132
@DataPoints —— 複数のパラメータを定義するアノテーション	134
8.3 組み合わせテスト	136
Assumeによるパラメータのフィルタリング	137
8.4 パラメータ化テストの問題	138
データの網羅性	138
パラメータに関する情報の欠落	140

第9章

ルール

テストクラスを拡張するしくみ

9.1 ルールとは?	141
ルールの宣言	142
ルールのしくみ	142
複数のルールの宣言	143
9.2 JUnitが提供するルール	143
TemporaryFolder —— 一時ファイルを扱う	144
● TemporaryFolderの拡張	145
ExternalResource —— 外部リソースを扱う	145
Verifier —— 事後検証を行う	146
ErrorCollector —— エラーを収集する	149
ExpectedException —— 詳細な例外を扱う	150
Timeout —— テストのタイムアウトを設定する	151
TestWatcher —— テストの実行を監視する	152
TestName —— テスト名を扱う	152
9.3 カスタムルールの作成	154
TestRuleインターフェース	154
事前条件をチェックするカスタムルール	155
OSに依存したテストを行うカスタムルール	156
9.4 RuleChainによるルールの連鎖	158
9.5 ClassRuleによるテストクラス単位でのルールの適用	160
コラム JUnitのバージョンとルール	161

第10章

カテゴリ化テスト

テストケースのグループ化	162
10.1 スローテスト問題	162
スローテスト問題とは?	162
対策	163
● テストの実行時間を短くする	163

目次

● テストの実行環境を強化する	164
● テストを並列で実行する	164
● 実行するテストを絞り込む	164
10.2 カテゴリ化テスト	165
カテゴリ化テストとは?	165
カテゴリ化テストの実行	165
● カテゴリクラスを作成する	166
● テストケースにカテゴリを指定する	166
● テストスイートクラスを作成する	166
● Eclipseからカテゴリ化テストを実行する	168
10.3 カテゴリ化テストのパターン	169
データベーステスト	169
通信処理を伴うテスト	169
GUIを伴うテスト	170
10.4 ビルドツールによるカテゴリ化テスト	170

Part 3 ユニットテストの活用と実践171

第11章

テストダブル

テスタビリティ、モック／スタブによるテスト	172
11.1 テスタビリティを高めるリファクタリング	172
テスタビリティとは?	172
リファクタリングとは?	174
コラム ユニットテストを前提としたリファクタリング	174
メソッドの抽出	175
● 処理をメソッドに抽出する	175
● 抽出メソッドをテストでオーバーライドする	175
委譲オブジェクトの抽出	177
● 処理を委譲オブジェクトに抽出する	177
コラム メソッドヒッププロジェクト	177
● 委譲オブジェクトをテストで置き換える	178
コラム DIヒュニットテスト	180
11.2 テストダブルとは?	181
スタブ —— 依存オブジェクトに予測可能な振る舞いをさせる	181
● 固定値を返すスタブ	182
● 例外を送出するスタブ	185
モック —— 依存オブジェクトの呼び出しを検証する	186
● モックとスタブの違い	186
スパイ —— 依存オブジェクトの呼び出しを監視する	187
コラム 状態に着目するテストと相互作用に着目するテスト	188

13.5 GUIアプリケーションのテストにおける注意点 254

コラム 正常系／準正常系／異常系 255

コラム Webアプリケーションの機能テスト 255

第14章

コードカバレッジ

テスト網羅率の測定 256

14.1 コードカバレッジとは? 256

カバレッジの基準 256

• C0(命令網羅) 257

• C1(分岐網羅) 257

• C2(条件網羅) 258

カバレッジ測定の効果 258

14.2 カバレッジツールの利用 258

ユニットテストの漏れを監視する 259

テストケースの問題を検知する 259

コードの実行をトレースする 259

カバレッジツールの選択肢 260

14.3 EclEmmaによるカバレッジ測定 260

EclEmma プラグインのインストール 261

コラム そのほかのカバレッジツール 261

EclEmma プラグインの実行 262

コラム EclEmma プラグインと JaCoCo エンジン 262

カバレッジビューアとカウンタ 263

• 命令カウンタ 263

• ブランチカウンタ 264

• 行カウンタ 264

• メソッドカウンタ 265

• 型カウンタ 265

• 複雑度 265

コードハイライト 266

測定結果のクリア 266

14.4 カバレッジに関するよくある疑問 266

カバレッジとは何か? 267

カバレッジ測定の目的は何か? 267

コラム デフォルトコンストラクタとカバレッジ 267

何%のカバレッジを目標とするべきか? 268

ユーティリティクラスのコンストラクタをどう扱うか? 268

再現困難な例外処理をどうするか? 269

いつカバレッジを測定するか? 270

カバレッジは本当に役に立つのか? 270

● テストの実行環境を強化する	164
● テストを並列で実行する	164
● 実行するテストを絞り込む	164
10.2 カテゴリ化テスト	165
カテゴリ化テストとは?	165
カテゴリ化テストの実行	165
● カテゴリクラスを作成する	166
● テストケースにカテゴリを指定する	166
● テストスイートクラスを作成する	166
● Eclipseからカテゴリ化テストを実行する	168
10.3 カテゴリ化テストのパターン	169
データベーステスト	169
通信処理を伴うテスト	169
GUIを伴うテスト	170
10.4 ビルドツールによるカテゴリ化テスト	170

Part 3 ユニットテストの活用と実践 171

第11章

テストダブル

テスタビリティ化、モック／スタブによるテスト	172
11.1 テスタビリティを高めるリファクタリング	172
テスタビリティとは?	172
リファクタリングとは?	174
コラム ユニットテストを前提としたリファクタリング	174
メソッドの抽出	175
● 処理をメソッドに抽出する	175
● 抽出メソッドをテストでオーバーライドする	175
委譲オブジェクトの抽出	177
● 処理を委譲オブジェクトに抽出する	177
コラム メソッドとオブジェクト	177
● 委譲オブジェクトをテストで置き換える	178
コラム DIとユニットテスト	180
11.2 テストダブルとは?	181
スタブ —— 依存オブジェクトに予測可能な振る舞いをさせる	181
● 固定値を返すスタブ	182
● 例外を送出するスタブ	185
モック —— 依存オブジェクトの呼び出しを検証する	186
● モックとスタブの違い	186
スパイ —— 依存オブジェクトの呼び出しを監視する	187
コラム 状態に着目するテストと相互作用に着目するテスト	188

11.3 Mockitoによるモックオブジェクト	190
Mockitoとは?	190
Mockitoを利用する準備.....	191
モックオブジェクトの作成.....	191
スタブメソッドの戻り値.....	192
スタブメソッドの定義.....	192
• 例外を送出するスタブメソッド.....	193
• void型を返すスタブメソッド.....	193
• 任意の引数に対するスタブメソッド.....	194
スタブメソッドの検証.....	194
コラム Mockitoの後置記法と前置記法.....	195
部分的なモックオブジェクト	196
スパイオブジェクト.....	197
• メソッド実行時のコールバック	197

第12章

データベースのテスト

テストコードで外部システムを制御する.....199

12.1 データベースに依存するユニットテスト	199
データベースを扱うソフトウェアの設計	199
• Web三層構造アーキテクチャ.....	200
• パーシステンス層のスタブによる置き換え.....	200
• プロダクション環境とユニットテスト	200
コラム 代替データベースを利用したテスト	201
データベースの状態とユニットテスト	202
• 参照系のテスト.....	202
• 更新系のテスト.....	202
12.2 ユニットテストの自動化とH2 Database	203
H2 Databaseサーバの起動／停止を行うルール	203
12.3 DbUnitによるデータベースのテスト	204
DbUnitとは?	204
DbUnitのJUnit 4対応	204
• DbUnitのルールクラスの作成	206
• DbUnitのルールクラスの利用	210
データベースのセットアップ	211
• DbUnitのデータセット	211
• データセットの外部定義	212
データベースのアサーション	213
• JUnit 4スタイルでのアサーション	213
コンテキストによるテストケースの整理	214
DbUnitのそのほかの機能	214

第13章

Androidのテスト

UIとロジックを分けてテストする.....	219
13.1 GUIアプリケーションの設計.....	219
GUIアプリケーション開発のポイント.....	220
MVCパターン.....	220
• モデル.....	220
• ビュー.....	221
• コントローラ.....	221
イベント駆動.....	222
• イベントハンドラ／イベントリスナー.....	223
GUIアプリケーションのスレッドモデル.....	223
コラム なぜGUIはシングルスレッドモデルなのか?.....	224
13.2 MVCパターンによるAndroidアプリケーションの作成.....	225
HelloAndroidの概要.....	225
Androidのプロジェクトの作成.....	226
エミュレータの起動.....	227
レイアウトを作成する.....	228
アクティビティを実装する.....	230
• コンポーネントの参照.....	231
• イベントハンドラの追加.....	231
モデル用プロジェクトの作成.....	233
モデルの定義.....	235
モデルのスタブ実装.....	237
コラム 独立したモデルプロジェクトのメリット.....	237
スタブ実装によるイベントの実装.....	238
13.3 モデルのテスト.....	238
認証ユーザクラスのテスト.....	240
ユーザ認証クラスのテスト.....	240
13.4 ビューとコントローラのテスト.....	243
Androidアプリケーションの機能テスト.....	243
Android SDKが提供するテスティングフレームワーク.....	244
アクティビティのテスト.....	245
• テスト用プロジェクトの作成.....	245
コラム JUnit 3.8におけるテストを書くルール.....	246
• MainActivityTestの作成.....	247
• アクティビティテストの初期化.....	248
• 初期状態のテスト.....	250
• アクティビティテストの実行.....	250
シナリオテスト.....	251
• ハッピーパスのテスト.....	251
• 拡張シナリオのテスト.....	252
• エラーシナリオのテスト.....	253
• シナリオの優先順位.....	253

13.5 GUIアプリケーションのテストにおける注意点 254

コラム 正常系／準正常系／異常系 255

コラム Webアプリケーションの機能テスト 255

第14章

コードカバレッジ

テスト網羅率の測定 256

14.1 コードカバレッジとは? 256

カバレッジの基準 256

• C0(命令網羅) 257

• C1(分岐網羅) 257

• C2(条件網羅) 258

カバレッジ測定の効果 258

14.2 カバレッジツールの利用 258

ユニットテストの漏れを監視する 259

テストケースの問題を検知する 259

コードの実行をトレースする 259

カバレッジツールの選択肢 260

14.3 EclEmmaによるカバレッジ測定 260

EclEmma プラグインのインストール 261

コラム そのほかのカバレッジツール 261

EclEmma プラグインの実行 262

コラム EclEmma プラグインとJaCoCoエンジン 262

カバレッジビューアとカウンタ 263

• 命令カウンタ 263

• ブランチカウンタ 264

• 行カウンタ 264

• メソッドカウンタ 265

• 型カウンタ 265

• 複雑度 265

コードハイライト 266

測定結果のクリア 266

14.4 カバレッジに関するよくある疑問 266

カバレッジとは何か? 267

カバレッジ測定の目的は何か? 267

コラム デフォルトコンストラクタヒカバレッジ 267

何%のカバレッジを目標とするべきか? 268

ユーティリティクラスのコンストラクタをどう扱うか? 268

再現困難な例外処理をどうするか? 269

いつカバレッジを測定するか? 270

カバレッジは本当に役に立つか? 270

Part 4 開発プロセスの改善 271

第15章

継続的テスト

すばやいフィードバックを手に入れる 272

15.1 継続的テスト 272

開発チームへのすばやいフィードバック	272
● 早期からテストする.....	273
● 自動的にテストする	273
● 繰り返しテストする	273
継続的テストとは?	274
継続的テストを行うための3つの要素	274
● ユニットテスト	275
● 自動化	275
● バージョン管理	275
継続的テストの運用	275
継続的テストとリファクタリング	276
● ユニットテストとリファクタリング	276
● 積極的なリファクタリング	277

15.2 Mavenによるビルドプロセスの自動化 277

Mavenとは?	278
コラム Mavenのバージョン	278
Mavenの準備	279
MavenプロジェクトとPOM	279
Mavenプロジェクトの作成	280
Mavenのプロジェクト構成	282
依存ライブラリの管理	283
● Mavenのリポジトリ	283
● 依存ライブラリの追加	284
ソースコードのエンコーディング設定	286
Mavenのプラグイン	286
● プラグインの設定	287
コラム プラグインのconfigurationセクション	288
Mavenによるビルドの実行とフェーズ	289
● テストをスキップする	289
● Eclipseから実行する	290
プラグインとゴール	291
● ゴールを実行する	291
Mavenによるカテゴリ化テスト	292
Mavenによるカバレッジレポート	293
● JaCoCo プラグインの導入	294
● JaCoCoによるカバレッジレポート	294

15.3 バージョン管理システムによる継続的テストの運用

バージョン管理システムとは?	296
バージョン管理システムへのコミット	296
Subversionでのテストサイクル	297
GitやMercurialでのテストサイクル	298
15.4 Jenkinsによる継続的インテグレーション	299
継続的インテグレーションとは?	299
Jenkinsとは?	300
Jenkinsの導入	300
Jenkinsのジョブ	301
ジョブの作成	302
ジョブの設定	302
● バージョン管理システムの設定	303
● ビルドゴールの設定	303
● ジョブの実行	304
ビルドトリガーの設定	304
● Jenkinsのリモートビルド機能	304
● バージョン管理システムのフック機能	305
継続的インテグレーションの効果	306
● テストの実行コスト削減	306
● すばやいフィードバック	306
● テスト結果の履歴	306
コラム 失敗するテストの扱い	307

第16章

テスト駆動開発

テストファーストで設計する	308
16.1 テスト駆動開発とは?	308
16.2 テスト駆動開発のサイクル	309
設計する	309
テストコードを書く	310
プロダクションコードを書く	310
リファクタリングを行う	311
次のサイクルを始める	311
16.3 Calculatorクラスのテスト駆動開発	311
テストファースト	312
● テストクラスの作成	312
● 設計とテストケースの作成	312
● テストケースの実装	313
コンパイルエラーの解決	314
コラム テスタビリティクラス設計	314
● クイックフィックスの活用	315
仮実装	316
● 仮実装の目的	317
リファクタリング	318

• リファクタリングの目的.....	319
三角測量.....	319
• 計測点の追加	320
• プロダクションコードの修正	321
16.4 テスト駆動開発の目的.....	322
ステップバイステップ	322
自分が最初のユーザ.....	322
動作するきれいなコード	323
すばやいフィードバック	323
メンテナンスされたドキュメント	324

第17章**振舞駆動開発**

ストーリーをテスト可能にする	325
17.1 受け入れテストとは?.....	325
受け入れテストの自動化	326
17.2 振舞駆動開発とは?.....	327
ソフトウェアの振る舞いを記述するシナリオ	327
• 「~するべき」とアサーション	327
振舞駆動開発と受け入れテスト	328
テスト駆動開発との違い	328
• テストコードと自然言語によるシナリオ	329
コラム ユースケースシナリオと振舞駆動開発のシナリオ	329
• API設計のタイミング	330
• 検証する対象の粒度	330
振舞駆動開発の語彙	330
• Given/前提	331
• When/もし	331
コラム 振舞駆動開発の生まれた背景	331
• Then/ならば	332
17.3 cucumber-junitによる振舞駆動開発.....	332
cucumber-junitのしくみ	333
cucumber-junitの準備	333
ポーカーアプリケーションの概要	334
フィーチャーファイルの作成	334
シナリオの作成	335
• ランダム性とテスト	336
シナリオの実行	336
ステップ定義の作成	337
• コードスニペットの利用	337
• ステップ定義ファイルの作成	338
ステップ定義の実装	339
• ポーカーアプリケーションの設計	339
• プロダクションコード	340

コラム PokerGameクラスの設計	341
ワンペアシナリオの作成	345
シナリオテンプレートの利用	348
アウトサイドインの開発	348

Part 5 演習問題 351

第18章

ベーシックなテスト	353
18.1 状態を持たないメソッドのテスト	353
18.2 例外を送出するメソッドのテスト	355
18.3 副作用を持つメソッドのテスト	357
18.4 同値クラスに対するテスト	359
18.5 void型を戻り値とするメソッドのテスト	361
18.6 マルチスレッドのテスト	365

第19章

アサーションとフィクスチャ	368
19.1 リストのアサーション	368
19.2 JavaBeansのアサーション	373
19.3 複数行テキストのアサーション	378
19.4 境界値のテスト	381
19.5 フィクスチャを用いたパラメータ化テスト	385
19.6 組み合わせテスト	387

第20章

テストダブルの活用	392
20.1 システム時間に依存するテスト	392
20.2 例外ハンドリングのテスト	395
20.3 外部システムに依存するテスト	398
20.4 インタフェースとスタブによるテスト	401
20.5 サーブレットのテスト	405
20.6 Hello Worldのテスト	408

付録.....411

付録A

開発環境のセットアップ.....412
A.1 JDKのセットアップ.....412
Javaのバージョン.....412
JDKのインストール.....413
A.2 Eclipseのセットアップ.....416
Eclipseのバージョン.....416
Eclipseのインストール.....416
Eclipseの日本語化.....417
Mac OS X環境におけるデフォルト文字コードの設定.....418
Eclipseは英語版のまま使う.....419

付録B

Eclipseの便利機能と設定.....420
B.1 EclipseのJUnitサポート.....420
Quick JUnitプラグインをインストールする.....423
テスティングペアを開く.....424
テストを実行する.....424
B.2 テキストファイルのエンコーディング設定.....421
B.3 staticインポートのワイルドカード.....421
B.4 Quick JUnit.....422
Quick JUnitプラグインをインストールする.....423
テスティングペアを開く.....424
テストを実行する.....424
B.5 コンテンツアシストとお気に入り.....425
B.6 テンプレート.....426
test —— テストメソッド.....428
at —— アサーション.....429
setUp —— 初期化メソッド.....429
tearDown —— 後処理メソッド.....430
setUpClass —— クラスの初期化メソッド.....430
tearDownClass —— クラスの後処理メソッド.....430
when —— ネストしたテストクラス.....431
instantiation —— インスタンス化テスト.....431
B.7 次の注釈.....432
B.8 クイックフィックス.....432
B.9 クイックアウトライン.....433

付録C

H2 Databaseのセットアップと使い方	435
C.1 H2 Databaseの特徴	435
C.2 H2 Databaseのセットアップ	435
C.3 H2 Databaseの起動と停止	436
サーバモード	436
組込みモード	437
C.4 H2 Databaseの起動オプション	438
C.5 H2 Console	438

付録D

Android開発環境のセットアップ	440
D.1 ADTのインストール	440
D.2 AVDの設定	442

付録E

Jenkinsの設定	444
E.1 JDKの設定	444
E.2 Mavenの設定	444
E.3 プラグインの設定	444

付録F

本書利用環境のバージョン	446
F.1 開発環境	446
F.2 Eclipseプラグイン	446
F.3 外部ライブラリ	447
F.4 Mavenプラグイン	447

巻頭付録

JUnitチートシート	表2
--------------------	----

巻末付録

Eclipseのショートカット	表3
------------------------	----

API索引／索引	448
----------	-----

JUnit入門

Part 1

- 第3章
JUnitの構造 —— JUnitアーキテクチャとフレームワーク
- 第2章
JUnitテスト —— テスト用アノテーションとアダプタ
- 第1章
JUnitとは —— Java言語の特徴とJUnitの開発歴

第1章

JUnitチュートリアル ユニットテストの作成から実行まで

本章では、JUnitによるユニットテストの進め方をチュートリアル形式で解説します。ユニットテストが初めて、または不慣れな人は一読してから次の章へお進みください。JUnitを使ったユニットテストの経験のある人は読み飛ばしていただいてもかまいません。

1.1 なぜ、ユニットテストを行うのか？

プログラマは人間であり、プログラミングは人間が行う知的で創造的な活動です。どんなプログラマであっても、人間である限り、間違いを犯します。良いプログラマは自分の書いたコードが本当に正しく動くか、不安に思います。この不安を安心に変える方法がユニットテストです。

ユニットテスト(*unit testing*)は、プログラマにとって重要なスキルです。ユニットテストでは、クラスやメソッドがプログラマの期待した振る舞いをすることを検証します。このような検証を、JavaプログラムのユニットテストではJUnitを使って行います。また、継続的インテグレーションやテスト駆動開発、振舞駆動開発などはすべて、ユニットテストが基盤です。

一方で、ユニットテストは万能というわけではありません。ソフトウェアは多くのクラスやメソッドの相互作用で動作しています。こうした相互作用やソフトウェア機能の確認、品質の保証については、機能テストや受け入れテストなどで行わなければなりません。

良いプログラマがユニットテストを実践する理由は単純です。それは、自分の書いたコードに責任を持ち、不安なくソフトウェア開発を行うためです。また、良いプログラマのテストコードは、変更に強く、読みやすいものです。

さあ、JUnitによるユニットテストを始めましょう。

1.2 JUnitとは？

JUnit^{注1}は、Kent Beck や Erich Gamma によって開発されたJavaのテスティングフレームワークです。主に次のような機能を持ちます。

- テストの実行フレームワーク
- テストの期待値と実測値の検証 API
- テストケースのフォーマット

簡単に言えば、「どのようにテストを記述して、実行し、検証するか」についてサポートしています。

バージョン

本書執筆時点でJUnitの最新バージョンは4.10です。最新のJUnitでは、ユニットテストを続けていく中で生まれた問題への解決手段や、さらに便利に使うための拡張が多く含まれています。本書のPart 2では、それらの最

注1 <http://www.junit.org/>

Column

企業文化と「ユニットテスト」の定義

開発プロセスなどと同様に、企业文化によって「テスト」の定義は大きく異なります。開発者として現場でテストを行う場合、その企業やチームでの「テスト」の定義を確認したうえで行わなければなりません。

「ユニットテスト」は「単体テスト」とも訳されますが、A社での「単体テスト」とB社の「ユニットテスト」、はたまたC社で「UT」と呼んでいるテストがまったく同じものを指すこともあれば、ある企業における「単体テスト」と、ある企業での「単体テスト」がまったく異なることもあります。また、開発標準などでテストに関するフォーマットや規約が定められている場合もあります。

特に「単体テスト」と呼ぶ場合、システムの単機能(画面単位や機能単位など)に関するテストを指す場合があります。これは、本書で扱う「クラスやメソッド単位でプログラムの仕様を検証する自動化されたテスト」とは異なりますのでご注意ください。

近追加された機能を紹介しています。

JUnitには、大きく分けて2つの系統があります。Javaのバージョンの大きな節目であるJava 5のアノテーション機能に対応したJUnit 4と、それ以前のバージョンでも動作するJUnit 3です。JUnit 4では、テストメソッドの記述ルールがアノテーションベースとなり、より宣言的にテストを記述できるようになりました。

本書では、明示的にバージョンを指定しない限り、JUnit 4について記述してあります。また、サンプルコードはJUnit 4.10にて動作確認を行っています。

ライセンス

JUnitのライセンスは、JUnit 4.10の時点ではCPL(Common Public License)を採用しています^{注2}。このライセンスはEPL(Eclipse Public License)とほぼ同等のオープンソースライセンスです。歴史的には、IBMの作ったライセンスであるIPL(IBM Public License)を汎用的にしたものであり、過去にはEclipseも採用していました^{注3}。

1.3 JUnitテストを始めよう

それでは、EclipseとJUnitを使ってJavaのテストプログラムを書いてみましょう。乗算と除算を行う簡単な計算機クラスと、そのユニットテストを書いて実行しながら、ユニットテストの雰囲気をつかんでください。

このチュートリアルを進めるには、次の環境がセットアップされている必要があります。

- JDK 6 (*Java Development Kit 6*)^{注4}
- Eclipse 3.7.2 Indigo以降
- Quick JUnit (Eclipse プラグイン)

注2 <http://www.junit.org/license>

注3 Eclipseのライセンスは、バージョン3.1でCPLからEPLへ移行しています。

注4 JDK 7でもサンプルコードを実行できますが、Eclipseでコードを開いた際に警告が出る場合があります。

環境のセットアップが済んでいない方は、付録A「開発環境のセットアップ」および付録B「Eclipseの便利機能と設定」を参考に、開発環境を準備してください。以下、本書ではMac OS X版のEclipse 3.7.2 IndigoにPleiadesプラグインで日本語化したものをベースに画面や操作の解説を進めていきます^{注5}。

準備ができましたら、Eclipseを起動して次のステップに進んでください。

チュートリアルの概要

このチュートリアルでは簡単な計算を行うCalculatorクラスを作成し、そのユニットテストを作成します。

Calculatorクラスは、乗算を行うmultiplyメソッドと除算を行うdivideメソッドを持つ、状態を持たないシンプルなクラスです。

JUnitによるユニットテストでは、動作するテストプログラムを作成して実行します。このとき、テストプログラムを記述するクラスをテストクラスと呼びます。そして、リリース予定のソフトウェアのコード(プロダクションコード)に含まれる、テストされるクラスをテスト対象クラスと呼びます。

プロジェクトを作成する

Eclipseでは、Javaのプログラムを「プロジェクト」という単位で管理します。プロジェクトは、ソースコードやライブラリなどを含んだフォルダ(ディレクトリ)です。

●新規Javaプロジェクトの作成

はじめにチュートリアル用のプロジェクトを作成しましょう。パッケージエクスプローラで右クリックしてコンテキストメニューを表示し、[新規]→[Javaプロジェクト]を選択します。

「新規Javaプロジェクト」ダイアログが表示されるので、[プロジェクト名]に「junit-tutorial」と入力し、[完了]をクリックしてください(図1.1)。そ

^{注5} Windowsで「Pleiades All in One」を利用している場合は、若干手順が異なる場合があります。

のほかの項目はデフォルトのままでかまいません。これで、いま作成した `junit-tutorial` プロジェクトがパッケージエクスプローラに表示されるようになります。

● ライブラリの追加

次に JUnit の JAR ファイルをプロジェクトのライブラリに追加します。公式サイト^{注6}から JUnit の JAR ファイル (`junit-4.10.jar`) をダウンロードし、プロジェクトフォルダにコピーしてください。続けてコピーした JAR ファイルを右クリックしてコンテキストメニューを表示し、「[ビルド・パス] - [ビルド・パスに追加]」を選択します(図 1.2)。

これで JUnit 4 がライブラリとして `junit-tutorial` プロジェクトに追加されました(図 1.3)。

以上でプロジェクトの作成は完了です。

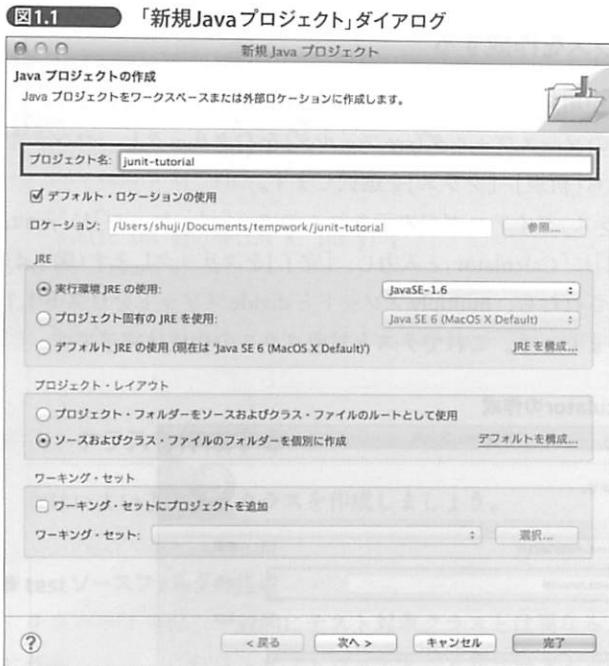
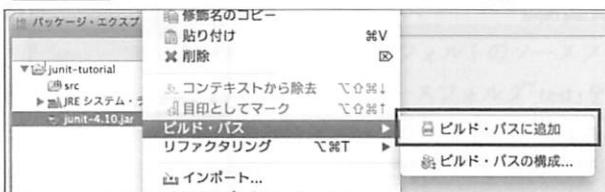
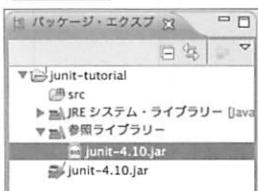
注6 <http://www.junit.org/>

Column

`junit-4.x.jar`と`junit-dep-4.x.jar`

JUnit の JAR ファイルには、「`junit-4.x.jar`」と JUnit の拡張ライブラリである Hamcrest を除外した「`junit-dep-4.x.jar`」があります。通常は `junit-4.x.jar` を使用しますが、Hamcrest を併用する場合は競合を避けるために `junit-dep-4.x.jar` を使用してください。

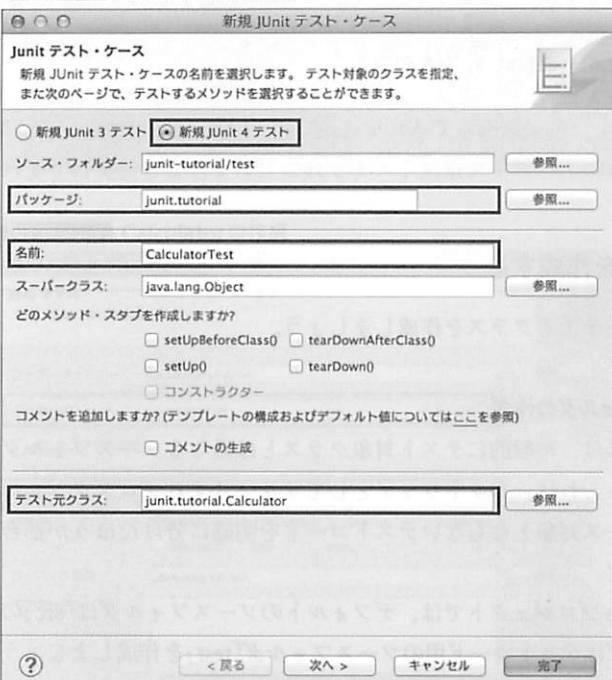
Hamcrest は、以前までは第 4 章で解説するアサーションや Matcher API などが提供されていましたが、現在、その主要な機能は JUnit に組み込まれています。より多くの機能を利用したい場合のみ、ライブラリとして追加してください。

**図1.2 JUnit 4.10ライブラリのビルドパスへの追加****図1.3 ライブラリの追加**

リックしてください(図 1.5)。

これで、リスト 1.2 のようなテストクラスが生成されます。Test アノテーションが付与されたメソッドが自動的に定義されています。

図 1.5 テストクラスの追加



リスト 1.2 初期の CalculatorTest

```
package junit.tutorial;

import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void test() {
        fail("まだ実装されていません"); ①
    }
}
```

図1.1 「新規Javaプロジェクト」ダイアログ



図1.2 JUnit 4.10ライブラリのビルドパスへの追加

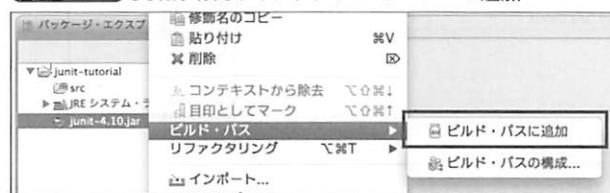
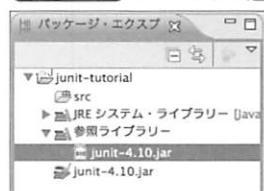


図1.3 ライブラリの追加



テスト対象クラスを作成する

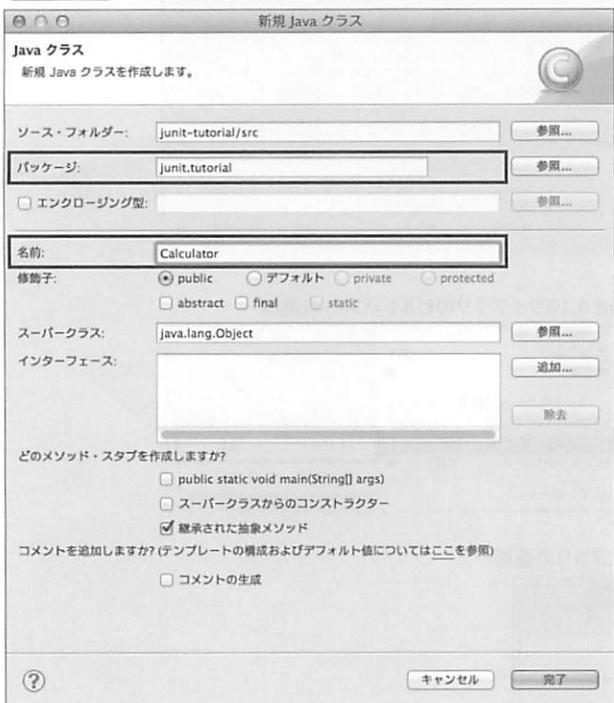
プロジェクトが用意できたら、次にテスト対象クラスを作成します。

プロジェクトのソースフォルダ(src フォルダ)を右クリックし、コンテキストメニューから[新規]-[クラス]を選択します。

「新規 Java クラス」ダイアログが表示されるので、[パッケージ]に「junit.tutorial」、[名前]に「Calculator」と入力し、[完了]をクリックします(図 1.4)。

クラスが作成されたら、multiply メソッドと divide メソッドをリスト 1.1 のように実装しましょう。これでテスト対象クラスの作成は完了です。

図 1.4 Calculator の作成



リスト1.1 Calculatorクラス

```
package junit.tutorial;

public class Calculator { ❶
    public int multiply(int x, int y) {
        return x * y;
    }
    public int divide(int x, int y) {
        return x / y;
    }
}
```

テストクラスを作成する

次はいよいよテストクラスを作成しましょう。

● testソースフォルダの作成

テストクラスは、一般的にテスト対象クラスとは異なるソースフォルダに作成します。これは、ソフトウェアとしてリリースするプロダクションコードとリリース対象とならないテストコードを明確に分けたほうが都合が良いからです。

EclipseのJavaプロジェクトでは、デフォルトのソースフォルダは「src」のみなので、まずはテストコード用のソースフォルダ「test」を作成しましょう。ソースフォルダを追加するには、プロジェクトのコンテキストメニューから[ビルド・パス]→[新規ソース・フォルダー]を選択します。「新規ソース・フォルダー」ダイアログが表示されたら、「フォルダー名」に「test」と入力して[完了]をクリックします。

● テストクラスの作成

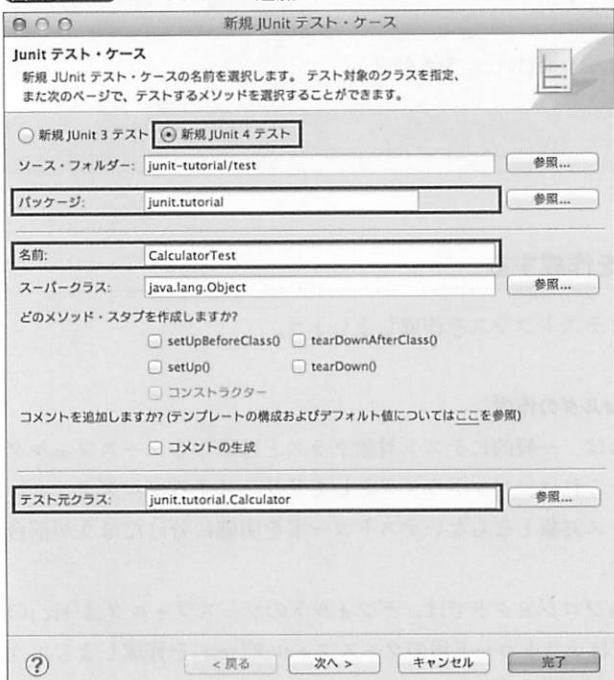
続けてテストクラスを作成します。ソースフォルダ「test」のコンテキストメニューを開き、[新規]→[JUnit テスト・ケース]を選択します。

「新規JUnit テスト・ケース」ダイアログが表示されるので、[新規JUnit 4 テスト]を選択し、[パッケージ]に「junit.tutorial」、[名前]に「Calculator Test」、[テスト元クラス]に「junit.tutorial.Calculator」と入力して[完了]をク

クリックしてください(図1.5)。

これで、リスト1.2のようなテストクラスが生成されます。Testアノテーションが付与されたメソッドが自動的に定義されています。

図1.5 テストクラスの追加



リスト1.2 初期のCalculatorTest

```
package junit.tutorial;

import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void test() {
        fail("まだ実装されていません"); ❶
    }
}
```

なお、ソフトウェアテストでは、テストする項目をテストケースと呼びます。JUnitでのテストケースは、テストメソッドとして実装します。テストメソッドには、テスト対象となるクラスやメソッドの振る舞いを検証するテストコードを記述します。

● Quick JUnitの利用

Quick JUnit プラグインがインストール済みであれば、テストクラスを作成する手順も簡単にできます。

Quick JUnit を使ってテストクラスを作るには、テスト対象クラスである Calculator クラスをエディタで開き、Calculator クラスの宣言行(リスト 1.1 ①)にキャレット(I字のカーソル)がある状態^{注7}でショートカット [Ctrl]+[G] を入力します。すると、「テスティングペアがありません。作成しますか?」というダイアログが表示されます。ここで[はい]を選択すると、自動的にテストクラス名やテスト対象が設定された状態で図 1.5 のダイアログが表示されます。あとは[完了]をクリックするだけでテストクラスが作成できます。

テストを実行する

JUnit はユニットテストを行うためのフレームワークのひとつです。フレームワークとは、特定の目的で実行させるプログラムを作るための枠組みです。JUnit ではユニットテストを行うための枠組みを提供しており、JUnit を利用するにはテストプログラムを一定のルールに従って作成する必要があります。

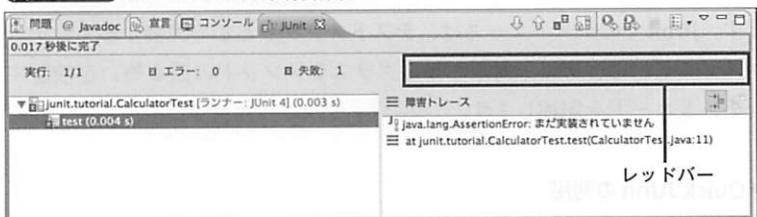
テストクラスを作成したときに、Test アノテーションが付与されたメソッドが自動的に定義されました。このメソッドこそ、JUnit を利用するためのルールに従ったテストメソッドです。

それでは、先ほど作成したリスト 1.2 を用いてテストを実行してみましょう。ユニットテストを実行するには、テストクラスのコンテキストメニューを開き、[実行]-[JUnit テスト]を選択します。

テストが実行されると、Eclipse の JUnit ビューに図 1.6 のような実行結果

注7 略密には、Calculator クラスが有効な範囲にキャレットがある状態であれば大丈夫です。

図1.6 ユニットテストの実行結果



が表示されます。

図1.6ではレッドバーとともに「実行: 1/1 エラー: 0 失敗: 1」、すなわち1件のテストを実行し、1件が失敗したことを確認できます。詳細は「障害トレイス」として表示されており、エラーメッセージが「まだ実装されていません」となっていることがわかります。このメッセージはリスト1.2の①に対応します。

1.4 テストコードの記述

乗算メソッドのテストを作成する

それでは、いよいよテストコードを書いていきましょう。まずはCalculatorクラスに対する乗算メソッドの検証を例にテストを作成していきましょう。

テストコードはJUnitのルールに従って書かなければなりません。JUnitでは、テストコードを次のルールで書く必要があります。

- テストクラスはpublicクラスとする
- テストメソッドは、org.junit.Testアノテーションを付与したpublicメソッドとする
- テストメソッドは、戻り値がvoidであり、引数を持たない
- throws句は自由に定義可能

なお、テストクラスにはいくつでもテストメソッドを定義できます。

●日本語のメソッド名でわかりやすくする

開発チームが全員日本人(あるいは日本語を読解できる)ということであ

れば、テストメソッド名は日本語とすることをお勧めします。なぜならば、テストコードはドキュメントとして読みやすいことが重要だからです。

リスト1.3は、デフォルトで生成されたテストメソッド(リスト1.2のtest)の名前を「multiplyで乗算結果が取得できる」に変更したものです。

日本語のメソッド名を使うメリットについては次ページのコラム「日本語のメソッド名を使うメリット」にまとめました。つまるところ、テストメソッド名がテストの概要を日本語で表していれば、各テストケースが明確となります。ただし、メソッド名の命名規則には従わなければならないため、句読点などの区切り文字は使用することができません。また、先頭文字としては数字を使用できません^{注8}。

● 値を比較検証する

続けてテストコードを記述します。ユニットテストは、「あるテスト対象クラスのメソッドを実行したときに、戻り値などで取得された実際の値と、期待される値が一致する(しない)はずである」という宣言で構成されます。このような値を比較検証する宣言はアサーションと呼ばれ、JUnitではorg.junit.AssertクラスのassertThatメソッドを使って実装します。

なお、アサーションについては第3章および第4章で詳しく扱いますので、ここではチュートリアルを進めるうえで必要な解説にとどめます。なお、Javaに組み込まれているassert構文とは振る舞いも使用目的も異なります

^{注8} 2バイト文字であっても使用できません。

リスト1.3 テストメソッド名を日本語に変更

```
package junit.tutorial;

import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void multiplyで乗算結果が取得できる() {
        fail("まだ実装されていません");
    }
}
```

ので注意してください。

ここで検証したいことは乗算メソッドの振る舞いとなるため、リスト1.4のようなテストコードとなります。リスト1.4では、「3と4を引数としてmultiplyメソッドを実行し、取得された値が12である」ことを検証しています。

リスト1.4❸の`assertThat`は`Assert`クラスに定義された`static`メソッドです。`assertThat`メソッドは引数を2つ持ち、1つ目の引数は実測値(*actual value*)で、2つ目の引数は期待値(*expected value*)です。2つ目の引数に表れる`is`メソッドは、`Matcher`と呼ばれる値の比較オブジェクトを生成するためのファクトリメソッド^{注9}で、`CoreMatchers`クラスに定義されています。これらの`assertThat`メソッドや`is`メソッドは、`static`インポートされて利用しています(同❶❷)。こうすることで、テストコードを自然言語(英語)に近い構文で記述できます。

注9 オブジェクトの生成を行うメソッドのこと。`Matcher`についての詳細は第4章で扱います。

Column

日本語のメソッド名を使うメリット

Javaの世界では標準APIはもちろん、ほとんどのライブラリでは、英語表記のメソッド名とクラス名が使用されています。また、一般的に、Javaの多くのコーディング標準(コーディング規約)では、「メソッド名には半角英数でわかりやすい英単語を使用すること」となっています。したがって、一部のAPIが日本語であると大きな違和感を感じるでしょう。

しかし、我々が今書こうとしているのはテストコードです。英語で格好よくテストの内容を記述でき、それを読み取ることができるのであれば、英語を使うほうが望ましいですが、日本語でソフトウェア開発をするならば、日本語でテストの内容を把握できるメリットのほうが大きくなります。

メソッドの一覧はJavadocに出力することでテスト項目の一覧となりますし、テストクラスのアウトラインを参照すればそのテストクラスで行っているテストが簡単に把握できます。テストが失敗したときのレポートも、失敗したテストメソッドが日本語で出力されることにより理解しやすくなります。

唯一の難点としては、日本語入力の切り替えを行わなければならないことがあります。テストコードの質が高くなるメリットのほうが大きいでしょう。はじめは違和感を感じるかと思いますが、テスト名を日本語で記述してみましょう。

リスト1.4 テストコードの追加

```
package junit.tutorial;

import static org.junit.Assert.*; ①
import static org.hamcrest.CoreMatchers.*; ②
import org.junit.Test;

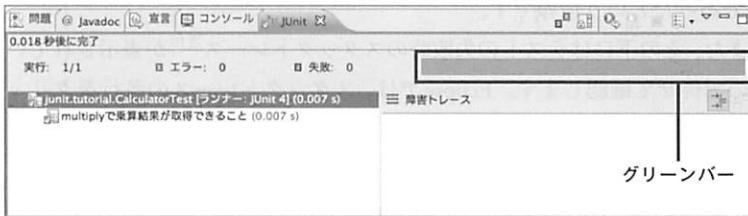
public class CalculatorTest {
    @Test
    public void multiplyで乗算結果が取得できる() {
        Calculator calc = new Calculator();
        int expected = 12;
        int actual = calc.multiply(3, 4);
        assertThat(actual, is(expected)); ③
    }
}
```

●乗算メソッドテストの実行

このテストは成功することが期待されるので、実行して確認してみましょう(図1.7)。

結果としては、「実行: 1/1 エラー: 0 失敗: 0」、すなわち1件のテストを実行し、エラーや失敗はありませんでした。また、今回は先ほどと異なり、グリーンバーが表示されます。JUnitをはじめ多くのテスティングフレームワークでは、ユニットテストの成功はこのようにグリーンで表現され、失敗はレッドで表現されます。このため、グリーンといえばテスト成功を、レッドといえばテスト失敗を意味します。

現時点では、乗算について1つのパターンしか検証していません。これでは、テストとしてはまだまだ不十分であり、さらにいくつかのパターン

図1.7 乗算メソッドテストの実行

について検証する必要があります。次のステップではテストのパターンを追加します。

乗算メソッドのテストを追加する

それでは2つ目のテストを追加しましょう。テストしたい内容は先ほどと変わらないため、「multiplyで乗算結果が取得できる」という名前でもう1つテストメソッドを作りたいところです。しかし、同じ名前のメソッドを2つ定義できませんし、できたとしても両者の区別がつきません。ここで安易に「multiplyで乗算結果が取得できる2」とするのではなく、具体的なテスト内容をメソッド名に反映しましょう。

1つ目のテストは、「 3×4 が12となる」ことを検証しているため「multiplyで3と4の乗算結果が取得できる」と修正し、新しく「multiplyで5と7の乗算結果が取得できる」を追加します。

なお、テスト失敗時の情報を確認するために、どちらも期待結果を12としておきます(リスト1.5)。

●障害トレースとエラーメッセージの読み方

テストを実行するとレッドバーになります(図1.8)。また、障害トレースには図1.9のようなメッセージが表示されています。

ユニットテストが失敗したならば迅速にその修正をする必要があります。その際、テストが失敗した原因に関する情報がより具体的で細かいほうが修正作業も楽です。もし、これが「テストは失敗しました」というメッセージだけだと、何が問題なのかがわからずに途方にくれてしまいます。

図1.9にはいくつもの重要な情報が含まれています。2行目と3行目は期待された値(Expected)が12であるのに対し、取得された実際の値(got)が35であったという情報です。

また、その下にはテストの失敗時のスタックトレース^{注10}が表示されているので併せて確認します。Eclipseでは、スタックトレースの各行をクリックすることで、その行に示されているクラスのソースコードへと移動でき

注10 ある時点での、プログラムが実行された過程の記録をプログラマが理解できる形式で出力したもの。

ます。この機能を使えば、より速いスピードで開発を進めることができます。

リスト1.5 乗算メソッドのテストを追加

```
package junit.tutorial;

import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void multiplyで3と4の乗算結果が取得できる() {
        Calculator calc = new Calculator();
        int expected = 12;
        int actual = calc.multiply(3, 4);
        assertThat(actual, is(expected));
    }
    @Test
    public void multiplyで5と7の乗算結果が取得できる() {
        Calculator calc = new Calculator();
        int expected = 35;
        int actual = calc.multiply(5, 7);
        assertThat(actual, is(expected));
    }
}
```

図1.8 乗算メソッドテストの実行結果(失敗)

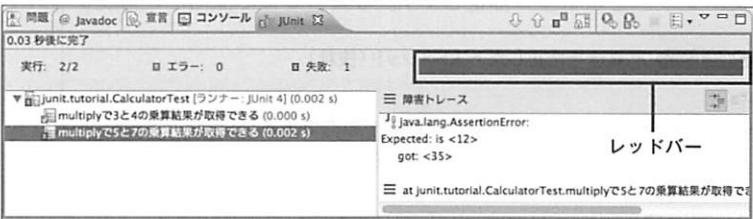


図1.9 テスト失敗時のエラーメッセージ

```
java.lang.AssertionError:  
Expected: is <12>  
got: <35>  
  
at junit.tutorial.CalculatorTest.multiplyで5と7の乗算結果が取得できる  
(CalculatorTest.java:20)
```

● テストコードの修正

このケースでは、テストコードに不具合がありました。期待する値を35に修正したテストコードはリスト1.6のようになります。

このように、ユニットテストでは1つのメソッドに対し複数のテストを行うことでテストの精度を高めていきます。まだ乗算メソッドのテストは不十分ですが、次のステップでは除算メソッドのテストを追加しましょう。

除算メソッドのテストを作成する

次に除算メソッドのテストを作成します。乗算メソッドのテストと同様に「divideで3と2の除算結果が取得できる」を最初のテストメソッドとしましょう。

● Calculatorクラスの設計変更

ここで、ひとつの問題に気づきます。リスト1.1を眺めると、Calculatorクラスは単純にint型の戻り値を返しています。しかし、「 $3 \div 2$ 」は整数で割り切れないため、このままでは期待される1.5を返せません(小数点以下は切り捨てとなっているようです)。Calculatorクラスの設計に欠陥があったと言えるでしょう。

そこで、リスト1.7のようにCalculatorクラスの除算メソッドの戻り値と計算ロジックをfloat型に修正します。なお、ここでは細かい精度について

リスト1.6 不具合を修正したテストメソッド(抜粋)

```
@Test
public void multiplyで5と7の乗算結果が取得できる() {
    Calculator calc = new Calculator();
    int expected = 35;
    int actual = calc.multiply(5, 7);
    assertThat(actual, is(expected));
}
```

リスト1.7 floatに対応した除算メソッド(Calculatorクラス)

```
public float divide(int x, int y) {
    return (float) x / (float) y;
}
```

は考慮しません。

続けてテストクラスに除算メソッドのテスト(リスト1.8)を追加します。

修正が終わったらさっそくテストを実行してみましょう。3件のテストが実行され、すべて成功しました(図1.10)。

このようにユニットテストを作り、具体的な値を当てはめることで、プログラムの潜在的な問題を見つけることもユニットテストの目的のひとつです。テスト対象クラスが完璧に設計されていることを前提とせずに、「何を検証するべきか?」を意識することで効果的なユニットテストを行うことができます。なお、このような考え方を強く意識する開発手法は、**テスト駆動開発**^{注11}と呼ばれます。



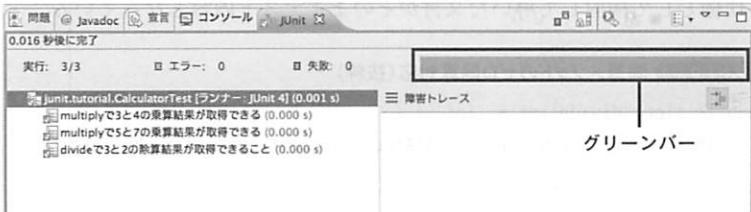
これで基本的なユニットテストの流れは理解できたと思います。しかしながら、実際の開発ではこのような単純なテストだけではなく、多くの検証しにくいパターンがあります。それらについては本書で少しづつ解説していきます。この章では最後に例外を扱うテストを紹介します。

注11 ➔ テスト駆動開発(p.308)

リスト1.8 除算メソッドのテスト(抜粋)

```
@Test
public void divideで3と2の除算結果が取得できる() {
    Calculator calc = new Calculator();
    float expected = 1.5f;
    float actual = calc.divide(3, 2);
    assertThat(actual, is(expected));
}
```

図1.10 除算メソッドテストの実行



ゼロ除算を例外として送出するテストを作成する

除算を行う場合には、ひとつ注意しなければならないことがあります。それはゼロ除算です。何らかの数を0で割るという計算は、数学的には「定義されていない」と見なします。すなわち、0で割った値は0でも無限大でもなく、0で割るという式自体に意味がないと見なします。

しかしながら、Javaプログラムとしてdivideメソッドの第2引数に0を与えることは防ぐことができません。したがって、第2引数に0が与えられた場合の振る舞いはdivideメソッドの設計者が決める必要があります。その際、どのような仕様にするかについての制限はありません。利用者にとって有益であれば、「第2引数に0を指定した場合には0を返す」でもかまわないのです。ですが、一般的にはゼロ除算をしようとしたということは利用者側の落ち度であり、メソッドは例外を送出するとよいでしょう。

● 除算メソッドのゼロ除算対応

それではさっそくゼロ除算に対応しましょう。

ここでは設計として「第2引数に0を指定した場合にはIllegalArgumentExceptionを送出する」とします。IllegalArgumentExceptionはJavaが提供する基本的なランタイム例外で、「引数が不正である場合に送出される例外」です。

これらのこと踏まえて、Calculatorクラスの除算メソッドを修正するリスト1.9のようになります。

● 例外の送出を検証するテスト

次に、このメソッドの振る舞いを検証します。検証する内容は「第2引数に0を指定した場合にはIllegalArgumentExceptionを送出する」です。先ほど仕様として検討して導いた文言がそのままテスト内容となっていること

リスト1.9 除算メソッドのゼロ除算対応(抜粋)

```
public float divide(int x, int y) {
    if (y == 0) throw new IllegalArgumentException("divide by zero.");
    return (float) x / (float) y;
}
```

を確認してください。

メソッドが例外を送出することを検証するには、リスト1.10のようにTestアノテーションのexpected属性を利用します。expectedの値として、IllegalArgumentExceptionのクラスオブジェクトを指定すると、指定したクラスの例外が送出された場合にテストが成功します。逆に、指定したクラスの例外が送出されないと、テストは失敗します。

チュートリアルの完了

以上でチュートリアルは完了です。JUnitによるユニットテストの基本を一通り確認できたと思います。次章以降では、ユニットテストの技法をJUnitの機能や周辺ライブラリを使いながら学んでいきましょう。

最後にチュートリアルで作成したテスト対象コード(リスト1.11)とテストコード(リスト1.12)全体を記載しておきます。

リスト1.10 例外の送出を検証するテスト

```
@Test(expected = IllegalArgumentException.class)
public void divideで5と0のときIllegalArgumentExceptionを送出する() {
    Calculator calc = new Calculator();
    calc.divide(5, 0);
}
```

リスト1.11 Calculator.java

```
package junit.tutorial;

public class Calculator {
    public int multiply(int x, int y) {
        return x * y;
    }
    public float divide(int x, int y) {
        if (y == 0) throw new IllegalArgumentException("divide by zero.");
        return (float) x / (float) y;
    }
}
```

リスト1.12 CalculatorTest.java

```
package junit.tutorial;

import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void multiplyで3と4の乗算結果が取得できる() {
        Calculator calc = new Calculator();
        int expected = 12;
        int actual = calc.multiply(3, 4);
        assertThat(actual, is(expected));
    }
    @Test
    public void multiplyで5と7の乗算結果が取得できる() {
        Calculator calc = new Calculator();
        int expected = 35;
        int actual = calc.multiply(5, 7);
        assertThat(actual, is(expected));
    }
    @Test
    public void divideで3と2の除算結果が取得できる() {
        Calculator calc = new Calculator();
        float expected = 1.5f;
        float actual = calc.divide(3, 2);
        assertThat(actual, is(expected));
    }
    @Test(expected = IllegalArgumentException.class)
    public void divideで5と0のときIllegalArgumentExceptionを送出する() {
        Calculator calc = new Calculator();
        calc.divide(5, 0);
    }
}
```

第2章

ユニットテスト

何のためにテストするのか

JUnitはJavaのユニットテストを支援する標準的なフレームワークです。JUnitの使い方を学べばユニットテストを書くことができます。しかしながら、効果的なユニットテストを行うためには、JUnitの使い方を知るだけでは不十分です。ユニットテストの基礎的な知識が必要となります。

この章では、ソフトウェアテストの概要とユニットテストの目的や基本となる概念を解説します。

2.1 ソフトウェアテストとは?

ソフトウェア開発ではさまざまなテストが行われます。製造業や建築業などでもテストは行われますが、ソフトウェア開発におけるテストは独特的な特徴を持ちます。

「テスト」という言葉からは、学生のころに苦しめられた中間テストや、大学入試などで行われる入学テストを思い出す人も多いかもしれません。それらの「テスト」は、問題として用意された質問に対して受験者が解答を行い、成績を出すプロセスです。成績は受験者がどの程度の知識を持っているかの指標であり、学生の評価などに利用されます。また、問題に対する解答が決められているか、模範解答が存在する場合がほとんどです。

一方、ソフトウェア開発におけるテストでは、検証する内容を定義し、ソフトウェアが期待どおりに動作するかを確認します。このような「テスト」では、テストの内容を作るのはテストを行うテスト自身であり、ソフトウェアの仕様や要件をもとにテスト項目を抽出します。満たすべき要件や機能などに対するテストであれば検証する項目は明確ですが、ユーザビリティテストのように検証する項目が明確ではなく、人間の感覚に依存するテストもあります。

ソフトウェアテストの特徴

ソフトウェア開発によるテストの定義は「ある条件下においてソフトウェアの振る舞いを記録し、その記録が期待される結果となることを検証するプロセス」です。この定義にはソフトウェアテストの重要なポイントが含まれています。

1つ目のポイントは、テストに「ある条件下」という制約があることです。これは「前提条件」や「事前条件」と呼ばれ、テストに使用するデータ、環境、事前の操作手順などが含まれます。前提条件が異なれば検証する内容も異なるため、テストでは前提条件が明確になっていなければなりません。

2つ目のポイントは、何らかの方法で「ソフトウェアの振る舞いを記録する」ことです。記録できなければ、期待される結果となることを検証できません。たとえば、出力されるデータやデータベースなどが、どのような状態であるかを確認できなければなりません。しかしながら、画面の表示を確認するなど人間が観測者となって変化を知ることができれば十分な場合もあります。

最後のポイントは、「期待される結果との検証」を行うことです。つまり、ランダム性の高い振る舞いであればあるほど、テストは難しくなります。

なお、ソフトウェア開発の大きな特徴として、プログラム上の問題や制約などが実際に作ってみなければわからないことが多いこと、そして一度作ったプログラムを破棄して作りなおすことが比較的に簡単であることが挙げられます。このため、製造業や建築業で行われているような設計段階で徹底的に検証して品質を高める手法は、ソフトウェア開発ではあまり効果的ではありません。ソフトウェア開発では、作りながら同時にテストを行ったり、実施したテストから得られたフィードバックからテストを追加したりする方法が有効です。

テストケースとテストスイート

ソフトウェアテストでは、1つのテスト項目をテストケースと呼びます。テストケースには、テストの前提条件、実行する操作、期待される値や状態がすべて含まれます。つまり、「どのような条件でどのような操作をした

ならば、こうなることが期待される」を記述したものがテストケースです。

通常、ソフトウェアテストは複数のテストケースで構成されています。テストケースの数が増えてくれば、何らかの形で似たテストケースをまとめる必要があります。このように、いくつかのテストケースをまとめたものをテストスイートと呼びます。

Column

製造業と建築業とソフトウェア開発

ソフトウェア開発では、製造業や建築業から多くの開発プロセスや考え方を取り入れています。これはテストでも同様ですが、ソフトウェア開発と製造業や建築業は性質が異なるため、まったく同じように行うことができません。

製造業では、設計と試作というプロセスでさまざまなテストを行います。しかし、工場などの製造ラインに乗って作られた製品について、全品検査を行うのは困難です。もし、全品検査を行ったとしても、検査を通らなかつた製品は作りなおすことが困難な場合、破棄されるでしょう。したがって、可能な限り設計と試作の段階で問題を減らし、製造後に問題を起こさないような工夫が行われています。

製造業における設計と試作のプロセスは、ソフトウェア開発では要件定義や設計プロセスに相当します。製造業と同様に設計段階で多くのレビューと検証を行い、プログラミングの段階に問題を持ち込まない方針をとる開発プロセスも存在します。しかしながら、小～中規模なプロジェクトで設計段階での過度な検証を行ってもコストを増やすだけで品質につながりません。本文中でも触れましたが、ソフトウェアへの要求の変化は大きく、作ってみなければ問題がわからないことが多いからです。

一方、建築物では、特殊な建物でない限りは長い歴史の中でパターンが構築されており、一定の規格の中で設計することで、強度、価格、品質などが予測できます。これに対して、ソフトウェアに対するニーズは、規模から目的まで多種多様であり、同じものが2つとない場合もあります。また、建築の分野などに比べれば歴史はありません。

ソフトウェア業界では、建築の世界を参考にして、ソフトウェアのモジュール化を進め、モジュールの組み合わせでソフトウェアを構築しようとする試みも行われてきました。しかしながら、特定の分野で特定の用途では効果があるものの、多様なニーズに対しては一定の効果しかないのでです。

そして、建築物では建築が始まると大きな手戻りはできないのですが、ソフトウェア開発ではいくらでも壊して作りなおすことができます。また、すべての建築物に対し、強度以上の負荷をかけて耐障害性を検証することはできませんが、ソフトウェアでは想定以上の負荷をかけて耐障害性を検証できます。

ソフトウェアテストの目的

ソフトウェアテストを行う主な目的は品質保証です。しかしながら、ソフトウェアテストには品質保証以外の目的もあります。たとえば、機能テストは設計時に考慮した仕様が不足なく実装されているかを検証するテストであり、ソフトウェアの完成度(進捗度)を計る目的で行われます。本書で扱うユニットテストも、クラスなどが正しく設計され動作することを検証するテストですが、品質を高めるためだけではなく、設計そのものが妥当であるかを検証する目的もあります。

このように、ソフトウェアテストではテストによって目的が異なるため、何のためのテストであるかを意識することは重要です。

特にユニットテストの目的は、ソフトウェア開発が進化する中で大きく変化してきました。1990年代以前のソフトウェア開発では、今よりも実現できることも少なく、ユニットテストといえば1本のプログラムを検証することとほぼ同義でした。しかし、現在のソフトウェアは、より複雑になり、多くのクラスやモジュールの相互作用として構築します。したがって、ソフトウェアの機能検証を目的としてクラスやモジュールを対象とするユニットテストを行うことはできません。

ソフトウェアテストの限界

ソフトウェア開発では、「完璧なテスト」を行うことはできません。製造業や建築業でも同様ですが、ソフトウェア開発では完全に不具合のないソフトウェアを作ることも「完璧なテスト」を行うこともできません。

たとえば、入力値がint型であるメソッドが1つあったとしても、すべての入力可能な値を網羅するテストケースを作ろうとすれば、約40億のテストケースが必要です。したがって、ソフトウェアテストを行う場合には、効果的な入力値を選択し、「十分に良いテスト」を行うしかありません。

なお、どの程度で「十分に良いテスト」とするかは、ユーザの要求や予算に依存する問題です。「完璧なテストは不可能である」という前提のもと、テストの方針を合意する必要があります。

2.2 テスト技法

ソフトウェア開発にはオブジェクト指向設計やアジャイル開発手法などさまざまな開発手法があります。その中でテストに関する技法は、**テスト技法**と呼ばれます。

テスト技法には境界値に対するテストのような有名なものから、探索的テストのような難易度の高いものまでさまざまなものがあります。ソフトウェア開発において、テストは非常に難しい技術であり、さまざまなテスト技法が研究されています。そして、どのようなソフトウェアでも効率良くテスト可能な「銀の弾丸」が存在しないのは、開発手法などと同様です。

ここではユニットテストで有効なテスト技法を紹介します。

ホワイトボックステストとブラックボックステスト

ソフトウェアテストは、テストケースの作り方によってホワイトボックステストとブラックボックステストに分類できます。

ホワイトボックステストは内部のロジックや仕様について考慮してテストケースを設計します。データの抽出時に内部ロジックの分岐などを考慮し、テストが可能な限りすべてのロジックを実行するデータを作成します。したがって、ソフトウェアの内部仕様とコードを理解しているプログラマがテストを行います。

一方、ブラックボックステストとは、文字どおりソフトウェアの内部仕様について考慮せず、外部仕様のみからテストケースを設計します。ブラックボックステストは、ユーザ受け入れテストや機能テストなど、比較的に大きな粒度でソフトウェアをエンドトゥエンドで検証するテストで採用されます。ブラックボックステストのテストケースを抽出するためにはプログラミングの知識よりも業務知識などが重要です。

本書で扱うユニットテストは、テスト対象の内部ロジックを考慮して行うため、どちらかといえばホワイトボックステストに分類されます。しかしながら、内部ロジックに依存しすぎたユニットテストは、変更に対して脆弱くなるため、テスト対象メソッドの外部仕様からテストデータを選択したほうがよいという側面もあります。このため、ユニットテストは、ホワ

イトボックステストとブラックボックステストを組み合わせて行います。

同値クラスに対するテスト

同値クラスに対するテストは、ソフトウェアが同様の結果をもたらす値を同値クラスとしてグループ化し、各同値クラスからテストデータを選択するテスト技法です。同値クラスに対するテストは、ブラックボックステストに分類されます。

同値クラスに対するテストは内部仕様や実装ではなく外部仕様、特に出力結果について着目してテストデータを選択します。

たとえば、自動販売機をシミュレートするソフトウェアがあるとします。自動販売機のあるボタンは、投入されているお金の合計が120円以上である場合には点灯し、120円未満の場合は点灯しません。ただし、投入金額が120円以上であってもお釣りが足りない場合は点滅します。また、自動販売機には10円硬貨、50円硬貨、100円硬貨、500円硬貨を投入できます。この自動販売機のボタンについてテストをするときに、投入されたお金の組み合わせをどう選択するとよいかを考えます。

自動販売機のボタンの結果は「点灯」「点灯しない」「点滅」の3つです。したがって、それぞれの場合について条件を満たすお金の組み合わせが同値グループを形成します。しかし、この組み合わせは膨大な量となるため、すべてをテストすることはできません。このため、各同値クラスについて代表値を選びテストします。代表値で正しく動作することが確認できたら、ほかの値でも(その同値クラスについては)正常に動作するものと見なします。

境界値に対するテスト

境界値に対するテストは、ソフトウェアが異なる結果をもたらす値(境界値)に着目し、境界値の近傍からテストデータを選択するテスト技法です。

たとえば、あるソフトウェアで「年齢が20歳以下の場合、割引料金が適用される」という要件があったとします。このとき、ソフトウェアが異なる結果をもたらす年齢の境界値は「20」です。したがって、境界値に対するテ

ストでは「20」の近傍である「20」と「21」をテストデータとして選択します^{注1}。

これは、経験的に「プログラムが仕様上の境界値付近で不具合を生みやすい」という性質を利用したテスト技法です。たとえば、前述のソフトウェアに含まれるメソッドには、次のようなコードが含まれるでしょう。

```
if (age <= 20) {
    // ageが20以下の場合の割引処理
}
```

プログラマがこのコードを書くときに不具合を混入するならば、比較するための演算子を間違えるか、比較するための値を間違えるかのどちらかです。「20」と「21」をテストデータとして選択すれば、この分岐を最小限のテストデータで検証できます。

このように、境界値に対するテストはブラックボックステストとホワイトボックステストの性質を持ちます。一般的にはブラックボックステストとして分類されますが、テスト対象のプログラムとテストコードを同時に実装する場合は、ホワイトボックステスト的に扱われます。

2.3 ユニットテストとは？

ユニットテストは、クラスやメソッドを対象としたプログラムを検証するためのテストであり、ソフトウェアテストの中では最も小さい粒度のテストです。ユニットテストでは、対象のクラスやメソッドが期待された振る舞いをするかを検証し、テストが成功することによってそれを保証します。この「期待された振る舞い」とは、言い換えれば、対象のクラスやメソッドの仕様です。

第1章のチュートリアルで作成した乗算メソッドを思い出してください。乗算メソッドには「multiply」という適切な名前があり、引数として2つの整数を持ち、戻り値として乗算した値を整数として返しました。これが、この乗算メソッドの仕様となります。この仕様をJavadocとして記述すると、リスト2.1のようになります。

このメソッドに対し、次のテストケースを作成しました。

^{注1} 割引料金が適用される年齢で「20」に最も近い値「20」と、適用されない年齢で「20」に最も近い値「21」を選択しています。なお、厳密には年齢の上限下限を考慮すべきですが、ここでは割愛します。

リスト2.1 乗算メソッドのドキュメント

```
/*
 * 引数で与えられた2つの値を掛け合わせた値を返す
 * @param x 1つ目の引数
 * @param y 2つ目の引数
 * @return xとyを掛け合わせた値
 */
public int multiply(int x, int y);
```

- multiplyで3と4の乗算結果が取得できる
- multiplyで5と7の乗算結果が取得できる

これはテストケースとしては適切です。しかしながら、プログラムとしては実行できません。また、テストケースが自然言語(日本語)で記述されているため、解釈による誤解が生じる余地が残ります。

ユニットテストの特徴

ユニットテストはプログラムとして実行できる仕様書となることが特徴です。そして、ユニットテストが成功する限り、正確な仕様書となります。

クラスやメソッドの仕様を自然言語(日本語や英語)で記述してテストを行うことは不可能ではありません。しかしながら、自然言語ではあいまいさを完全に排除することは難しく、句読点やニュアンスで読み手が誤解するかもしれません。また、テストを実行するには手動でプログラムを実行し、手動で値を確認しなければならないため、大きなコストとなります。

一方、クラスやメソッドの仕様をプログラムとして記述するならば、あいまいさが含まれることなく明確に記述できます。プログラマがテストコードを誤解して読むことはあるかもしれません、コンパイラや実行環境が間違って実行することはありません。

第1章のチュートリアルでは、乗算メソッドのテストをリスト2.2のように記述しました。JUnitによるユニットテストでは、対象のクラスやメソッドの仕様を実行できる形式で記述できます。また、このときに作成したコードは、対象のクラスやメソッドの実行できるサンプルコードと考えることもできます。そして、プログラマ自身が、作成したクラスやメソッドの

リスト2.2 乗算メソッドのテストコード

```

@Test
public void multiplyで3と4の乗算結果が取得できる() {
    Calculator calc = new Calculator();
    int expected = 12;
    int actual = calc.multiply(3, 4);
    assertThat(actual, is(expected));
}

@Test
public void multiplyで5と7の乗算結果が取得できる() {
    Calculator calc = new Calculator();
    int expected = 35;
    int actual = calc.multiply(5, 7);
    assertThat(actual, is(expected));
}

```

最初のユーザです。もし、テストコードを書いているときに使いにくいと感じたならば、設計や仕様に問題があると気づくこともできます。

また、プログラムとしてユニットテストを行う場合、最初にテストコードを記述するコストはかかりますが、実行するコストはほとんど必要ありません。したがって、何度も繰り返し実行できますし、それを頻繁に行なうことも可能です。なお、半自動的に実行して常に検証結果を確認する手法は第15章で詳しく解説します。

一方、手動によるテストの場合はテストの実行に大きな労力が必要なため、簡単には何度も繰り返して実施できません。

ユニットテストの目的

ユニットテストではクラスやメソッドを対象として、仕様どおりの振る舞いをするかを保証します。しかしながら、直接の目的はソフトウェアの品質を高めることではありません。それでは、ユニットテストを行う目的はどこにあるのでしょうか？

ユニットテストは、テスト対象のクラスの仕様を明確にしなければ書くことはできません。また、ユニットテストを繰り返し何度も実行することで、プログラムに問題が発生したときに、早い段階で影響範囲などをチエ

ックできます。この仕様の明確化とすばやいフィードバックがあるため、プログラマはリグレッション^{注2}の恐れを最小限にして新しい機能や試みを行うことができるようになります。これまで「動作するコードはけっして触ってはいけない」とされ、リスク回避のために避けてきた機能拡張やコードの修正を、安心して行うことができます。

つまり、ユニットテストの目的は、対象のクラスやメソッドの仕様を動くプログラムとして記述することにより、仕様を明確にし、その仕様を保証することです。

したがって、ユニットテストを行うことにより、プログラマは円滑に、不安なく開発を進めることができます。このため、ユニットテストはデベロッパー・テストとも呼ばれます。

なお、ユニットテストをより強力に活用する開発手法として「テスト駆動開発」があります。テスト駆動開発ではクラスやメソッドの仕様をプロダクションコードの実装よりも先に検討し、その仕様をユニットテストとして記述します。詳しくは第16章で解説します。

もちろん、ユニットテストを実施しているソフトウェアの品質は結果的に高くなります。不具合は減り、変更に強くなるでしょう。それは、すばやいフィードバックと仕様の明確化の賜物です。

ユニットテストのフレームワーク

ユニットテストはプログラムとして実行できること、すなわち自動化されたテスト（後述）であることが重要です。本書では、JUnitを利用して自動化されたユニットテストを行っていますが、必ずしも JUnit を使う必要はありません。

JUnit以外のフレームワークとしては、TestNG^{注3}が有名です。TestNGは、Java 5でアノテーションによる宣言的な文法が導入されたころ、早い段階でアノテーションに対応していました。ほかにも JUnit にはなかった便利な機能が多く実装されていたため、JUnitよりも好んで使うプログラマもいま

注2 修正と無関係の個所が壊れること。デグレ(デグレーション)とも呼ばれます。

注3 <http://testng.org/>

す。EclipseなどのIDEやMavenなどのビルドツールも、TestNGに対応したプラグインを導入することで利用できます。

現在ではJUnitとTestNGの間に決定的な機能の違いはなく、どちらを使うかは好みと言えます。本書で紹介するJUnitの機能のほとんどはTestNGでも同等の機能が提供されているでしょう。

また、フレームワークを利用せずに自動化されたユニットテストを行うこともできます。つまり、通常のJavaプログラムとしてユニットテストを書くこともできます。重要なことは、ユニットテストがプログラムとして自動的に実行できることです。

しかしながら、通常はJUnitやTestNGなどのフレームワークを利用したほうが便利です。なぜならば、テストの実行、検証、テスト結果のレポートといった、テストケースとは直接関連しない面倒な部分を実装する必要がないため、テストケースの設計と実装に専念できるからです。

2.4 ユニットテストのパターン

ソフトウェアの設計にはさまざまなパターンがあるように、ユニットテストに関しても効果的に行うための有益なパターンが数多くあります。この節では、それらのパターンを多くまとめているWebサイト「xUnit Test Patterns」^{注4}から、基本的な概念を中心に、重要なパターンをいくつか紹介します。なお、「xUnit Test Patterns」は書籍^{注5}としても刊行されていますが、ほとんどの記事はWebサイトで参照可能です。

自動化されたテスト——繰り返しいつでも実行できること

ユニットテストに限らず、ソフトウェアテストは可能な限り自動化すべきです。つまり、テストがプログラムとして繰り返し何度も実行できる状態にします。

JUnitを使ってユニットテストを記述すれば、基本的には自動化されたテ

注4 <http://xunitpatterns.com/>

注5 Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley Professional, 2007.

ストとなります。ユニットテストが完全に自動化されることで、何度も繰り返してテストを実行することが容易となります。その結果、不具合などを早い段階でフィードバックでき、安心してリファクタリングや機能拡張を行うことができるようになります。

ただし、GUI(*Graphical User Interface*)のテストなど自動化が難しいテストに関しては、テストの自動化に対するコストとのトレードオフとなります。

なお、開発プロセスでテストの実行自体を自動化するとより効果が高まります。詳しくは第15章で解説します。

不安定なテスト——結果が一定でないテストを避けること

ユニットテストでは、常にすべてのテストが成功している状態を維持することが望ましいです。成功しないテストを放置する状態や、実行ごとに成功するか失敗するかわからないテスト(不安定なテスト)^{注6}がある状態は望ましくありません。

注6 現在時刻に依存するテストなど。これらをどのようにテストするかは第11章で解説します。

Column

実行環境とテスト

Javaの開発では、開発環境がWindowsでプロダクション環境^{注a}がLinuxであることは珍しいことではありません。そして、開発環境では問題なく動作するアプリケーションが、プロダクション環境で動かないことがあります。これは、ファイルのセパレータや絶対パスがハードコーディングされていることが主な原因です。

実行環境に依存する実装を避けることは基本です。これは、ユニットテストにも深く関係します。なぜならば、第10章で紹介する継続的テストを導入し、テスト用サーバを設置するならば、その構成はプロダクション環境に近いほうが効果的だからです。開発環境とテスト環境でユニットテストの結果が異なれば、テストを不安定と感じ、ユニットテストがサーフティネットとして機能しなくなるでしょう。逆に、プロダクション環境に近いテスト環境でユニットテストが通ることがわかれば、大きな安心につながります。

どうしても環境依存避けられない場合もありますが、可能な限り、実行環境に依存しない実装を行うべきです。

注a いわゆる本番環境。運用時にアプリケーションを実行する環境のこと。

そのようなテストが存在すると、ほかのテストが失敗しても誰も気にしなくなり、気が付けば多くのテストが失敗している状態となります。こうなってしまうと、ユニットテストがセーフティネットとして機能しなくなり、重大な不具合の発見が遅れてしまうでしょう。そして、修正しようとしたときには修正範囲が大きくなりすぎて手を付けられなくなってしまいます。

一時的なテストの失敗はしかたのないことですが、何日もテストを失敗した状態にしておくのは避けるべきです。

ドキュメントとしてのテスト——仕様書として読めること

テストケースは、テスト対象のクラスやメソッドに対し開発者が想定しているさまざまなユースケースを記述したものです。また、テストが成功する限り、テストケースで定義されている動きは保証されています。したがって、テストケースは最も正確なドキュメントであり、テスト対象のサンプルコードです。

また、読みやすいテストコードを記述することで、テストコードのメンテナンス性も高くなり、最高のドキュメントが維持できます。テストコー

Column

テストケースとドキュメント

筆者の体験談となりますが、可読性の高いユニットテストを書くことを心がけるようになってから、プロダクションコードのコメントが大幅に減りました。

Martin Fowlerの『リファクタリング』^{注b}にある「コードの不吉な匂い」のひとつに「コメントは消臭剤」というものがあります。これはプログラマが自信のないコードを書くときにコメントが多くなるため、コメントの多いコードには不具合が多いという傾向を表したものです。

また、ユニットテストを書くようになると、インターフェースに関する関心が高くなります。なぜならば、コメントを読まないと理解できないインターフェースはテストコードを書いていても苦痛だからです。

このように、ユニットテストにはコメントがなくとも使いやすいインターフェースに導く効果もあります。

^{注b} Martin Fowler著／児玉公信、友野晶夫、平澤章、梅澤真史訳『リファクタリング——プログラミングの体質改善テクニック』ピアソン・エデュケーション(2000年)

ドもプロダクションコードと同じようにメンテナンスし続ける必要があります。常にテストコードを読む人を意識してテストコードを書くように心がけましょう。

このように、ドキュメントを意識して記述したテストコードは、テスト対象クラスの仕様書にほかなりません。

問題の局所化—— テスト失敗時に問題を特定しやすいこと

テストケースは、十分に小さな単位で可能な限り多く作るべきです。

テストが十分に小さいならば、何らかの原因でテストが失敗したとしても、影響範囲と条件が絞り込みやすくなります。ある一定の条件でうまく動かなくなったのか、広範囲に影響を与えてしまったのかなどの情報も多く得られます。テストに失敗したとき、その原因の分析のコストはけっして小さくありません。原因を早く特定できるようにテストケースはなるべく小さくしてください。

不明瞭なテスト—— 可読性の低いテストコードは避けること

テストケースにおいて、前提条件、実行、検証などが複雑に入り組んでいると、テストコードが非常に読みにくくなります。読みにくいテストコードはメンテナンス性も悪く、テストの質にも大きく影響を与えます。したがって、テストコードは可能な限りシンプルにわかりやすく記述すべきです。特に、各ステートメント(行)が、前提条件、実行、検証のどこに属しているかが明確な短いコードであるべきです。

前提条件は、同じコードが2回現れたならばsetUpメソッド^{注7}に抽出します。同じテストクラスの中で2つ以上の前提条件が重複して出現したならば、コンテキストごとのテストを検討してください。コンテキストごとにテストケースを整理する方法は第6章で扱います。

テストのための「実行」は、1つのテストケースで1つだけ行うようにしてください。ある操作をして、別の操作をして、また別の操作をする、とい

注7 ➡@Before — テストの実行前に処理を行う(p.53)

ったテストの実行フェーズを行うと、何が原因でテストが失敗したかがわかりにくくなります。このような場合は、前提となる操作は前提条件に含め、テストする実行は1つの操作としてください。

検証も、1つのテストケースで1つだけ行うのが理想ですが、現実的には検証は1つのテストケースでいくつか行うことになるでしょう。しかし、検証する項目は1つずつを行い、まとめて検証することは避けるほうがテストケースが明瞭になります。特にデータクラスなどの検証は、`equals`メソッドで行うことでのコードは読みやすくなりますが、検証に失敗したときにどのフィールドが一致しないのか分析に時間がかかります。フィールドごとに検証を行うか、カスタムMatcherを作成して検証しましょう^{注8}。

独立したテスト——実行順序に依存しないこと

テストケースは、可能な限りお互いに影響を与えないように定義すべきです。もし、あるテストの結果やテストの実行順番がテストの結果に影響を与えるならば、テストケースの追加や削除により予期しない形でテストが失敗することになります。

这样的ことは起こりそうにないと思いがちですが、シングルトン^{注9}として管理されているリソースやデータベースにアクセスするテストでは珍しいことではありません。また、Javaのコンパイラや実行環境が異なったときに、テストの実行順序が変わる場合もあります。

このため、原則としては各テストの初期化処理で必要なデータなどを用意し、テストの終了処理でデータやリソースをテスト前の状態に戻します。こうすることで、各テストがそれぞれ独立したものとなり、実行順番にテスト結果が依存しません。テストが独立しているのであれば、テストの実行時間が増えてしまった場合にも分散してテストを実行しやすくなります。

テストを行いにくい設計を避けることも重要です。ハードコーディングされた設定ファイルを読み込むリソースや、ミュータブルなオブジェクト^{注10}をシングルトンとして設計すると、この間に嵌まりやすいので注意しましょう。

注8 ⇒ カスタムMatcherの作成(p.71)

注9 詳しくは次ページのコラムを参照ください。

注10 作成後も状態を変えることができるオブジェクト。

Column

シングルトンオブジェクトとユニットテスト

シングルトンパターンは、クラスのインスタンスが1つしか生成されないことを保証するデザインパターンです。わかりやすく便利であるため、アプリケーションの設定クラスや状態を管理するクラスにしばしば適用されます。しかしながら、シングルトンオブジェクトは、ユニットテストを行うときには大きな障害となります。

シングルトンパターンを適用する場合は、そのオブジェクトが本当に1つであることが必要かどうかを確認してください。どこからでもアクセスしやすいようにシングルトンパターンとすることは、グローバル変数の置き換えでしかありません。

■テストの独立性を破壊する

ユニットテストでは、テストの実行ごとにテストクラスのインスタンスを作成し、テストメソッドを実行します。テストメソッドでは、テストデータやテスト対象クラスのインスタンスを生成し、検証を行います。そして、テストが成功したかどうかにかかわらず、テストに関連したすべてのインスタンスは破棄されます。ところが、シングルトンオブジェクトが存在する場合、各テストで同じオブジェクトを共有してしまいます。つまり、テストの独立性を破壊してしまいます。

■テストの後処理と初期化処理を複雑にする

ユニットテストでは、さまざまな前提条件において、テスト対象メソッドの検証を行います。したがって、設定クラスや状態を管理するクラスは、テストごとに前提条件に合わせてセットアップされる必要があります。テストごとにそれらの前提条件が破棄されるならば、初期化処理だけを考慮すれば十分です。しかしながら、シングルトンオブジェクトである場合、後処理も考慮しなければなりません。

■テストを不安定にする

ユニットテストでは、テストが独立しているのであれば、テストを並列に実行することができます。しかしながら、シングルトンオブジェクトがある場合、テストを並列に実行すると、お互いのテスト結果に影響を与える可能性があります。

■モック／スタブとの関係

シングルトンパターンでは、インスタンスが1つしか存在しないことを保証するために、コンストラクタをprivateとし、staticメソッドからそのクラスのインスタンスを取得します。このため、テスト用のモックやスタブを作成することができません。また、シングルトンオブジェクトを利用しているコードを、テスト用のコードに置き換えてテストを実行することが困難です。

第3章

テスティングフレームワーク ユニットテストを支えるしくみ

JUnitによるユニットテストでは、テストクラスに定義されたテストメソッドがテストの基本単位となります。各テストでは、テスト対象となるクラスやメソッドの振る舞いを検証します。そのためには入力値となるデータの準備や期待される結果の検証が必要です。もし、これらの手順が不完全であるならば、テスト自体が不完全なものとなってしまいます。特に複雑なクラスやメソッドのユニットテストでは、これらの手順も複雑になるため注意が必要です。また、テストコードはソフトウェアのライフサイクルと同じ期間使い続けるものであり、メンテナンス性も重要な要素です。

本章では、JUnitを使ったユニットテストにおける、基本的なテストの定義方法と、良いテストを書くためのガイドラインについて説明します。

3.1 テスティングフレームワークとは？

JUnitなどテストを支援するフレームワークは、**テスティングフレームワーク**と呼ばれます。

テスティングフレームワークでは「どのようにテストを記述して、実行し、検証するか」のしくみを提供します。テスティングフレームワークを利用すれば、一定のフォーマットでテストケースを記述できるため、テストコードは読みやすいものとなります。また、入力値の検証やテスト結果の収集などのしくみを提供するため、テストコードの記述に集中することができます。

xUnitフレームワーク

テスティングフレームワークにはいくつかの系統があり、JUnitは**xUnit**フレームワークと呼ばれます。「xUnit」は、Kent Beckにより開発されたSmalltalk

用のテスティングフレームワークである「SUnit」を源流としたフレームワークで、同氏と Erich Gamma により Java に移植されたものが「JUnit」です。

xUnit はさまざまな言語に移植されており、ユニットテスト用フレームワークの基本となっています。

3.2 JUnitによるユニットテストの記法

第1章のチュートリアルでも触れましたが、JUnit ではユニットテストをテストクラスにテストメソッドとして定義します。テストメソッドには、テスト対象となるクラスやメソッドの振る舞いを検証するテストコードを記述します。テストメソッドは次の制約で作成しなければなりません。

- public メソッドとする
- org.junit.Test アノテーションを付与する
- 戻り値を void とし、引数を持たない

メソッド名は、Java の言語仕様に従う範囲であれば、どのような名前でもかまいません。本書では理解しやすいように日本語のメソッド名を用います。また、テストクラスには複数のテストメソッドを定義することもできます。

テストメソッドのひな型は次のようにになります。

```
@Test
public void テスト名() throws Exception {
    // テスト対象となるクラスやメソッドの振る舞いの検証
}
```

テストメソッドの throws 句

JUnit ではテストメソッドの throws 句に制約はありません。JUnit ではテストメソッドが例外を送出したときにテスト失敗として扱うため、通常は throws 句に Exception クラスを指定します。

一般的な Java のコーディング標準では、メソッドの throws 句に Throwable や Exception などの基底となる例外クラスを指定することを禁止しています。

これは、例外処理を適切に行うために重要なルールであり、レビューや静的コード解析ツールなどでチェックされる項目となっています。

しかしながら、JUnitでユニットテストを書く場合は状況が異なります。テストメソッドはJUnitによって実行されるメソッドであり、例外処理はフレームワーク上で行われます。テストメソッドから期待されていない状態で例外が送出された場合、そのテストは失敗とし、エラーメッセージが出力されるしくみです。したがって、throws句に送出される可能性がある例外を詳細に列挙したかどうかは、テストの挙動に影響を与えません。

また、テストメソッド内ではさまざまなクラスが利用されます。テストコードを書いたあとにAPIの仕様を変更し、新しいタイプの例外が送出されるかもしれません。そのたびにテストメソッドのthrows句を修正するのには非常に面倒でしょう。

これらの理由から、コーディング標準に含まれる「throws Exceptionの禁止」は、テストコードには適用すべきではありません。

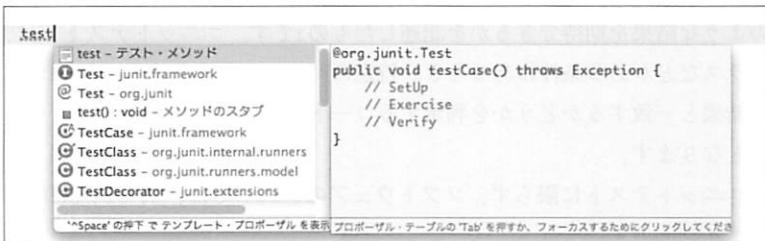
テストメソッドを簡単に挿入する

Eclipseのテンプレート機能を使えば、テストメソッドのひな型をテストクラスの中に簡単に挿入できます。やり方は、テストクラスの中のテストメソッドを書きたい場所で「test」と入力してショートカット **[Ctrl]+[Space]** を押します(図3.1)。

すると、コンテンツアシストが起動しますので、JUnit 4用のテンプレート(デフォルトでは「test - テスト・メソッド(JUnit4)」)を選択してください。

なお、ここで挿入されるコードスニペットはEclipseにデフォルトでも組

図3.1 コンテンツアシストとテンプレート



み込まれていますが、独自にカスタマイズすることもできます。詳細は付録B「Eclipseの便利機能と設定」を参照してください。

3.3 可読性の高いテストコードの書き方

テストのためのソースコードはテストコードと呼ばれます。これに対して、プロジェクトに含まれるソフトウェア本体のソースコードはプロダクションコードと呼ばれます。どちらも同じソースコードですが、目的が異なるため、コードを書くときに重視すべき点は異なります。

テストコードは、プロダクションコードに比べて、一定のパターンで同じようなコードが多くなる傾向があります。一見すると似たようなテストコードであっても、前提条件が少しだけ異なるということも珍しくはありません。

ソースコード上の重複はなるべく減らすように整理すべきです。しかしながら、テストコードについては、プロダクションコードと同じように整理するべきではありません。なぜならば、テストコードは個々の独立したテストケースに対して可読性が高いことが求められるからです。言い換えれば、あるテストコードを読むことにより、そのテストで何を前提として、何を行い、何を期待しているのかを理解しやすいテストコードが求められます。

本節では、可読性の高いテストコードを書くための基礎となる用語やポイントを紹介します。

テストケース

テストケース(テスト項目)は「ある状況である入力(操作)をしたときにどのような結果を期待できるかを記述したもの」です。ユニットテストでは、クラスなどがある条件にあるとき、特定のパラメータを入力し、予想される結果と一致するかどうかを判定するコードを記述したものがテストケースとなります。

ユニットテストに限らず、ソフトウェアのテストでは、次の3点が重要なポイントです。

- ・テストを行う前提条件
- ・テストに用いる入力値や操作
- ・テストを行ったときに期待する値や動作

もし、これらが不完全に定義されていたならば、そのテスト自体が不完全なものとなってしまいます。前提条件、入力、期待する結果が正しく定義されており、乱数や外部環境に依存しないテストであれば、誰がいつテストを実施しても、同じ結果となります。

テスト対象

テスト対象となるクラスやオブジェクトは、SUT(*System Under Test*)と呼ばれます。テストケースを作成するときには、そのテストの対象が何であるかについて明確にしなければなりません。また、1つのテストケースでは1つのSUTを対象としなければなりません。もし、複数のSUTを1つのテストケースで扱った場合、そのテストケースが何のテストをしているのかが不明瞭になります。

Column

不完全なテストケース

不完全なテストケースとして典型的なものは、確認項目に「正しく表示されること」などと書かれたテストケースです。

テストケースは、読んだ人によって解釈が変わらるようでは成立しません。「正しい表示」とは何か、テストをする人間によって基準が異なってしまえば、テスト結果に信頼性を期待できません。これは、システムテストや受け入れテストなどを手動で行う場合に発生しやすい状況です。

もちろん、ユーザビリティテストなど人間の感覚に依存するテストもあるため、すべてのテストケースについて明確な基準を設けることはできません。ですが、可能な限りは誰もが判断できる基準でテストケースを書くようにするべきです。

ただし、完璧なテストも完璧なソフトウェアも作ることができないことを忘れないでください。パフォーマンステストで5秒以内と基準を設けたとしても、特定の状況でやむを得ずパスできないのであれば、パフォーマンスを改善する前に基準の見直しも検討すべきです。

JUnitでは、通常はクラスごとに対応するテストクラスを作ります。また、慣習としてテストクラスはテスト対象と同じパッケージとし、クラス名を「テスト対象のクラス名+Test」とします(リスト3.1)。このため、テスト対象のクラスは命名規則から明確に1つに決まります。

テストコードでは、テスト対象クラスのインスタンスを生成し、テスト対象オブジェクト(SUT)とします。リスト3.2のように、インスタンスを「sut」という名前の変数で扱うことで、テスト対象オブジェクトがどれであるかが理解しやすいテストコードとなります。

実測値と期待値

ユニットテストでは、テスト対象となるオブジェクトやクラスにメソッド呼び出しなどの操作を行い、仕様どおりに動作することを検証します。このとき、メソッドが返す値やオブジェクトの変化する状態などは実測値(*actual value*)と呼びます。

一方、仕様として返すべき値や変化する状態である値を期待値(*expected value*)と呼びます。期待値は、原則として仕様から一意に決まるものであり、仕様変更がなければ不变な値です。

リスト3.1 テスト対象クラスとテストクラス

```
// テスト対象クラス (TargetClass.java)
public class TargetClass {
}

// テストクラス (TargetClassTest.java)
public class TargetClassTest {
}
```

リスト3.2 テスト対象オブジェクトと変数名

```
@Test
public void isValidはuserNameとpasswordが空でない場合にtrueを返す()
    throws Exception {
    UserForm sut = new UserForm("user01", "1234");
    assertThat(sut.isValid(), is(true));
}
```

可読性を高めるため、テストコードに含まれる変数名は、実測値に「actual」、期待値に「expected」とするとよいでしょう。

一般的なJavaのコーディング標準では、変数名として意味のある名前を付けることとなっています。しかしながら、テストコードで最も重要なことは、仕様上の意味ではなく、期待値であるか実測値であるかというテスト上の意味です。したがって、変数名を「errorMessage」などとするよりも、リスト3.3①のように「expected」とするほうが読みやすくなります。

なお、1つのテストメソッドに複数の実測値や期待値が必要となった場合、そのテストケースまたは対象となるテスト対象が大きすぎる可能性があります。可読性を高めるためには、テストメソッドが増えたとしても、1つのテストメソッドで1つのことを検証するとよいでしょう。

メソッドと副作用

わかりにくいテストコードとなってしまう最大の要因は、テスト対象がテストしにくい設計となっていることです。テストの可読性やメンテナンス性を高めるために最も効果的なことは、ユニットテストを行いやすいクラス設計とすることです。次のような性質を持っているメソッドは、テストが行いやすいメソッドです。

- メソッドが戻り値を持つ
- メソッドの呼び出しの結果、副作用がない^{注1}
- 同じ状態、同じパラメータで実行すれば、必ず同じ結果を返す

注1 オブジェクトの内部状態(インスタンス変数など)が変更されないこと。

リスト3.3 期待値と実測値

```
@Test
public void userNameが空のときにエラーメッセージが取得できる()
throws Exception {
    UserForm sut = new UserForm("", "1234");
    String expected = " UserIDは必須項目です。"; ①
    String actual = sut.getErrorMessage();
    assertThat(actual, is(expected));
}
```

このような戻り値を検証でき、副作用もランダム性もないメソッドは「関数的」とも呼ばれます。

テストコードでは、メソッドに対してパラメータを与え、その結果が期待される状態かどうかを検証します。副作用のないメソッドであれば、テストコードもシンプルに記述できます。

たとえば、文字列が空文字またはnullであるかを判定するメソッドは、文字列パラメータを受け取り、判定した結果を返します。この結果は同じ文字列を何度判定しても不变であり、オブジェクトへの副作用もありません（リスト3.4）。

一方、副作用のあるメソッドのユニットテストは難しくなります。たとえば、ArrayListのaddメソッドは副作用があり、戻り値を持たないメソッドの典型的な例です。このメソッドをテストするには、sizeメソッドやgetメソッドを使い、オブジェクトの状態を間接的に確認するしかありません（リスト3.5）。

また、テスト対象オブジェクトが乱数や現在時刻などで実測値が変化する場合は、さらにテストが難しくなります。そのようなクラスやメソッドは、モックオブジェクトなどを使うことでテスト可能にします^{注2}。

注2 ➡ テストダブル(p.172)

リスト3.4 副作用のないメソッド

```
public static boolean isEmptyOrNull(String value) {
    return value == null || value.isEmpty();
}
```

リスト3.5 ArrayListのaddメソッドのテスト

```
@Test
public void addで要素を追加するとサイズが1となりgetで取得できる()
throws Exception {
    ArrayList<String> sut = new ArrayList<String>();
    sut.add("Hello");
    assertThat(sut.size(), is(1));
    assertThat(sut.get(0), is("Hello"));
}
```

4フェーズテスト

ユニットテストに限らず、ソフトウェアのテストは次の4つのフェーズで実行されます。

- ・事前準備
- ・実行
- ・検証
- ・後処理

事前準備(*set up*)では、テスト対象オブジェクト(SUT)の初期化、必要な入力値、期待される結果などの準備を行います。

実行(*exercise*)では、テスト対象オブジェクトに対し、テストする操作を1つだけ行います。

検証(*verify*)では、テストの結果として得られた実測値が期待値と等価であるかを比較検証します。

最後に、後処理(*tear down*)として、次のテストの実行に影響がないよう後に始末をします。

4フェーズテストを意識して書かれたテストコードは、テストを読む人にとって非常に読みやすく理解しやすいコードとなります。特に事前準備とテストの実行が不明瞭なテストコードは、非常に読みにくいものとなります。このため、コメントとして各フェーズの区切りを記述しておくことは良い習慣です(リスト3.6)。

もし、4フェーズテストのコメントがないと、どの部分がSUTの初期化で、どの部分が実際に行うテストなのかが不明瞭です。なお、ユニットテストで後処理が必要となることはほとんどありません^{注3}。このため、後処理のコメントは省略しています。

テストフィクスチャ

テストフィクスチャ(*test fixtures*)とは、テストの実行時に必要とされるす

^{注3} JUnitではテストメソッドの実行ごとにテストクラスのインスタンスが生成され、終了時にインスタンス変数はすべて破棄されるため。

リスト3.6 4フェーズをコメントしたテストコード

```

@Test
public void 要素が2つ追加された状態で要素をremoveするとsizeが1となる()
throws Exception {
    // SetUp
    ArrayList<String> sut = new ArrayList<String>();
    sut.add("Hello");
    sut.add("World");
    // Exercise
    sut.remove(0);
    // Verify
    assertThat(sut.size(), is(1));
    assertThat(sut.get(0), is("World"));
}

```

べてのデータや状態のことです。主にテストを行うための入力値とテストの期待値で構成されます。

単純な整数や生成が容易なオブジェクトであれば、テストコードの中にフィクスチャの初期化コードを含めますが、データベースの初期化や生成が複雑なデータの場合は工夫が必要です。テストフィクスチャのセットアップのためのコードが長くなると、そのテストが本来何をやりたいのかが不鮮明となってしまうからです。

また、テストコードは対象となるクラスやメソッドのサンプルコードとしての価値もあります。したがって、テストコードの読み手にとって最も重要な個所は、テストの実行と検証です。

このため、テストフィクスチャの構築のためのコードはコンパクトにまとめるほうが読みやすくなります。テストフィクスチャは外部定義ファイルに記述する、フィクスチャの初期化メソッドを外部クラスに定義してテストケースから呼び出す、といった工夫をするとよいでしょう^{注4}。

3.4 比較検証を行うアサーション

テストは、実行して得られた結果が期待される結果と一致するかどうか

注4 ➡ テストフィクスチャ(p.106)

の検証を行うプロセスです。ユニットテストでは、テスト対象となるオブジェクトのメソッドを実行し、戻り値が期待される値となっているか、オブジェクトの状態が期待される状態となっているかを比較検証します。ユニットテストでは、このような比較検証のしくみをアサーション(*assertion*、表明)と呼びます。

アサーションとは、ユニットテストにおいて「テスト結果である実測値(*actual*)が期待値(*expected*)と一致する」という宣言です。これをJUnitが提供するAPIを使って記述していきます。

ユニットテストが実行されると、アサーションにより実測値と期待値が一致するかを判定します。そして、一致しない場合にはテスティングフレームワークに通知され、テストが失敗となります。テストコードでは、実行して得られた結果が期待される結果と一致するかどうかを検証するために、いくつかのアサーションを宣言します。

JUnitでは、主にorg.junit.Assertクラスの`assertThat`メソッドとMatcher APIを利用してアサーションを記述します。

JUnitのアサーション

`assertThat`メソッドは2つの引数を持ち、1つ目の引数に実測値を、2つ目の引数に期待値を含むMatcherオブジェクトを指定します。通常はstaticインポート^{注5}して利用するので、アサーションはリスト3.7のようなコードとなります。

注5 static変数やstaticメソッドをインポートし、クラス名を使わずに参照するJavaの機能。

リスト3.7 JUnitのアサーション

```
// static インポート
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;

// テストコード
Object expected = ...;
Object actual = sut.doSomething();
assertThat(actual, is(expected)); ①
```

assertThat メソッドの2つ目の引数に表れる is は、Matcher オブジェクトを生成するファクトリメソッドです。org.hamcrest.CoreMatchers クラスに定義されており、assertThat メソッドと同様に static インポートして利用します。

is メソッドで生成される Matcher オブジェクトは、オブジェクトの同値性、すなわち equals メソッドによる比較で、実測値と期待値が等しいことを検証します。ほかにもさまざまな Matcher オブジェクトがあり、プロジェクト固有の Matcher を定義することもできます。詳細は第4章で解説します。

このように、static インポートを使ってアサーションのコードを記述する目的は、テストコードを自然言語(英語)に近い構文とするためです。実際、リスト 3.7 ① は「assert that actual is expected」(実測値が期待値であると表明する)と読むことができます。

なお、int 型などのプリミティブ型を使っている場合でも、同じようにアサーションのコードを書くことができます(リスト 3.8)。プリミティブ型を指定した場合は、Java のオートボクシング変換が行われ、Integer オブジェクトの equals メソッドによる比較が行われます。

3.5 JUnitが提供するアノテーション

アノテーションとは、Java 5 で導入された言語仕様のひとつで、クラス、メソッド、変数などに補助的な情報を宣言するための機能です。JUnit では、メソッドに Test アノテーションを付与することで、フレームワークが実行するテストメソッドを認識するために利用されています。また、Test アノテーション以外にもさまざまなアノテーションが提供されており、それらを活用することで JUnit の機能を宣言的に利用できます。

本節では、JUnit の提供する代表的なアノテーションを紹介します。

リスト 3.8 int 型のアサーション

```
int expected = 108;  
int actual = sut.doSomething();  
assertThat(actual, is(expected));
```

@Test —— テストメソッドを宣言する

JUnitではテストケースを、リスト3.9のようにorg.junit.Testアノテーションを付与したメソッドとして定義します。

なお、メソッドは戻り値がvoidであり、引数を持たないpublicメソッドである必要があります。また、expectedとtimeoutの2つの属性を持つことができます。

● expected

expected属性は例外の送出を検証するテストで、送出が期待される例外クラスを指定するために利用します⁶。たとえば、あるテストでIllegalArgumentExceptionが送出されることを検証するならば、リスト3.10のようにexpected属性を指定します。

注6 ➔例外の送出を検証するテスト(p.57)

リスト3.9 Testアノテーションの利用

```
@Test
public void addに3と4を与えると7を返す() throws Exception {
    // SetUp - 初期化
    Calculator sut = new Calculator();
    sut.init();
    // Exercise - テストの実行
    int actual = sut.add(3, 4);
    // Verify - 検証
    assertThat(actual, is(7));
    // TearDown - 後処理
    sut.shutdown();
}
```

リスト3.10 Testアノテーションのexpected属性

```
@Test(expected = IllegalArgumentException.class)
public void 例外テスト() throws Exception {
    sut.doSomething();
}
```

● timeout

timeout属性は、ユニットテストのタイムアウト値を設定するために利用されます。テストメソッドの実行に、timeout属性に設定した時間(単位はミリ秒)以上の時間がかかった場合、テストが自動的に失敗するようになります(リスト3.11)。

timeout属性を指定しなかった場合は、テストはタイムアウトしないため、無限ループやデッドロックが発生した場合にテストの実行が永遠に停止しません。

timeout属性は実行時間が長くなるテストがテスト全体の実行を止めてしまわないように保険として使うべき機能であり、パフォーマンステストのために利用する機能ではありません。

なお、Timeoutルールを使うことで、テストクラスに定義されたテストすべてにタイムアウトを設定することもできます^{注7}。

@Ignore —— テストの実行から除外する

原則として、ユニットテストは常にすべてを実行し、成功するようにすべきです。しかしながら、都合により一時的にテストの実行を抑制したい場合があります。このような場合には、テストメソッドにorg.junit.Ignoreアノテーションを付与することで、該当するテストメソッドの実行をスキップできます(リスト3.12)。

Ignoreアノテーションはテストクラスにも付与できます。テストクラスに付与した場合はテストクラス内のすべてのテストメソッドがスキップされます。

注7 ➡ Timeout —— テストのタイムアウトを設定する(p.151)

リスト3.11 Testアノテーションのtimeout属性

```
@Test(timeout = 100L)
public void timeoutTest() throws Exception {
    Timeout sut = new Timeout();
    int actual = sut.doLongTask();
    assertThat(actual, is(100));
}
```

リスト3.12 Ignoreアノテーションによるテストのスキップ

```
@Ignore("未実装")
@Test
public void divideで4と2を与えると2を返す() throws Exception {
    // Exercise - テストの実行
    int actual = sut.divide(3, 4);
    // Verify - 検証
    assertThat(actual, is(7));
}
```

@Before——テストの実行前に処理を行う

JUnitによるユニットテストでは、テスト対象のクラスごとにテストクラスを作成し、テストクラスに複数のテストメソッドを定義していきます。すると、自然と初期化処理に重複したコードが生まれます。重複した初期化処理はBeforeアノテーションを付与した初期化処理メソッドに定義します。

共通した初期化処理はメソッドに抽出し、org.junit.Beforeアノテーションを付与します。このとき、メソッドは戻り値がvoidの引数を持たないpublicメソッドである必要があります。慣習としてメソッド名はsetUpとします^{注8}(リスト3.13)。

このように初期化に必要な処理を宣言的に分離して定義できます。setUpメソッドで例外が発生した場合は、そのテストは失敗となります。

ただし、抽出できる初期化処理は、同一クラス内での共通処理に限られます。複数のテストクラスに共通した処理を抽出するために、テストクラスのスーパークラスを作成することができますが、継承を多用すべきではありません。JUnit 4.7で導入された「ルール」を使用します。

^{注8} JUnit 3では、初期処理メソッドとしてTestCaseクラスのsetUpメソッドをオーバーライドしていました。

リスト3.13 Beforeアノテーションによる初期化処理

```
@Before
public void setUp() throws Exception {
    sut = new Calculator();
    sut.init();
}
```

ルールはテストクラスでの共通処理を独立したクラスに定義できる機能です。ルールを使うことで、より再利用性と拡張性の高い共通処理をテストケースに与えることもできます。詳細は第9章で解説します。

@After —— テストの実行後に処理を行う

`org.junit.After` アノテーションは、テストケースに共通する後処理をメソッドとして抽出する機能を提供します。`Before` と同様にメソッドは戻り値が`void`の引数を持たない`public` メソッドである必要があります。慣習としてメソッド名は`tearDown` とします(リスト 3.14)。

`After` で定義された終了処理は、テストの成功／失敗にかかわらず必ず実行されます。したがって、リソースの解放など「必ず行わなければならない後処理」が必要な場合に利用します。

@BeforeClass —— テストの実行前に一度だけ処理を行う

`org.junit.BeforeClass` アノテーションは、`Before` アノテーションと同様に共通の初期化処理を行うためのアノテーションです。`BeforeClass` アノテーションでは、`Before` と異なり、テストクラスがクラスローダに読み込まれたあと、そのテストクラスに含まれるいずれか最初のテストが開始される前に1回だけ実行されます。メソッドは、戻り値が`void`の引数を持たない`static`な`public` メソッドである必要があります(リスト 3.15)。

リスト3.14 Afterアノテーションによる後処理

```
@After
public void tearDown() throws Exception {
    sut.shutdown();
}
```

リスト3.15 BeforeClassアノテーションによる初期化処理

```
@BeforeClass
public static void setUpClass() throws Exception {
    // JDBC Driverの読み込み
    Class.forName("com.mysql.jdbc.Driver");
}
```

なお、BeforeClass アノテーションを使うシーンは多くありません。なぜならば、ユニットテストにおいて、テストメソッドは基本的に独立しているべきだからです。後述の AfterClass アノテーションも含め、テストクラス単位でのリソースの作成／解放を行うよりも、テスト単位で行なうことが推奨されます。

したがって、一部のライブラリの制約によりテストクラス単位で行なわなければならぬような状況や、データの初期化のコストが大きくテストメソッドごとに実行できないような状況でのみ利用すべきです。

Column

JUnit 3スタイルのテスト

Java 5 で導入されたアノテーションを使ったテストケースの記述スタイルは JUnit 4 で導入されました。それ以前はアノテーションではなく、命名規約でテストメソッドを識別する記述スタイルをとっていました(リスト 3.A)。

JUnit 3 のスタイルのテストクラスは、TestCase クラスのサブクラスとしなければなりません。テストメソッドは「test」で始まる必要があります。setUp や tearDown は TestCase に定義されたメソッドをオーバーライドして実装します。

なお、JUnit は後方互換性を保ちながらバージョンアップを行っているため、JUnit 3 スタイルで記述したテストコードも JUnit 4 で実行することができます。

また、Matcher API や assertThat メソッドは、JUnit 4.4 で導入されました。それ以前の JUnit 4 や JUnit 3 では、assertEquals メソッドや assertTrue メソッドなどを利用してアサーションを記述します。

リスト 3.A JUnit 3スタイルのテスト

```
public class JUnit3StyleTest extends junit.framework.TestCase {

    protected void setUp() throws Exception {
    }

    protected void tearDown() throws Exception {
    }

    public void test加算のテスト() throws Exception {
        assertEquals(7, 3 + 4);
    }
}
```

@AfterClass —— テストの実行後に一度だけ処理を行う

`org.junit.AfterClass` アノテーションは、テストクラスのテストメソッドがすべて実行されたあとに1回だけ実行される後処理メソッドを定義します（リスト3.16）。

`BeforeClass` と同様、やむを得ずテストクラス単位でリソースなどを管理しなければならないとき以外は使用しないほうがよいでしょう。

3.6 JUnitのテストパターン

ユニットテストは、比較的にパターン化しやすい技術です。「このような場合は、このように書けばテストできる」というパターンを構築することで、ユニットテストをより効率良く書くことができます。

本節では、次に示す3つの基本パターンを紹介します。

- ・標準的な振る舞いを検証するテスト
- ・例外の送出を検証するテスト
- ・コンストラクタを検証するテスト

標準的な振る舞いを検証するテスト

ほとんどのユニットテストはこのパターンとなり、初期化／実行／検証／後処理の4つのフェーズで構成されます（4フェーズテスト）。すなわち、テスト対象オブジェクトや入力／検証に必要なデータを初期化し、テスト対象となるメソッドを1つだけ実行したあと、期待される結果となるかを検証します（リスト3.17）。

必要に応じて後処理を行う必要がありますが、基本的にはテストごとに

リスト3.16 AfterClassアノテーションによる後処理

```
@AfterClass
public static void tearDownClass() throws Exception {
    GlobalResources.release();
}
```

テストクラスのインスタンスが破棄されるため、ガベージコレクションに任せるべきです。

標準的な振る舞いを検証するテストでは、検証時に予期された値と実際の値が異なる場合にテスト失敗となります。また、テストの実行中に何らかの例外が発生した場合もテスト失敗となります。

例外の送出を検証するテスト

例外の送出を検証するテストは、メソッドの実行時に例外が送出されることを検証するテストです。JUnitの例外テストでは、送出される例外の型をTestアノテーションのexpected属性に指定します(リスト3.18)。expected属性が指定されているテストでは、設定された例外クラスまたはそのサブクラスの例外が送出されたとき、テストが成功となります。

ただし、JUnitの例外の検証では、例外の型以外はチェックできません。したがって、例外に含まれるメッセージの検証や、想定しない個所で同じ

リスト3.17 標準的な振る舞いを検証するテスト

```
@Test
public void addに3と4を与えると7を返す() throws Exception {
    // SetUp - 初期化
    Calculator sut = new Calculator();
    sut.init();
    // Exercise - テストの実行
    int actual = sut.add(3, 4);
    // Verify - 検証
    assertThat(actual, is(7));
    // TearDown - 後処理
    sut.shutdown();
}
```

リスト3.18 例外の送出を検証するテスト

```
@Test(expected = IllegalArgumentException.class)
public void 例外テスト() throws Exception {
    sut.doSomething();
}
```

型の例外(たとえば、初期化処理の中でIllegalArgumentExceptionが発生したなど)については検証できません。とはいってもほとんどの場合ではデフォルトの検証方法で十分です。

より詳細に例外を検証したい場合は、ルールのひとつであるorg.junit.rules.ExpectedExceptionクラスを使うことで実現できます^{注9}。

コンストラクタを検証するテスト

ほとんどのテストでは、テスト対象となるのはメソッドです。したがって、初期化処理でテスト対象クラスのインスタンスを生成し、テスト対象となるメソッドを呼び出してテストを行います。

特殊なパターンとなるのがインスタンスの生成のテスト、すなわちコンストラクタのテストです。インスタンスが生成されたときに、初期状態として妥当な値が設定されているかを検証します。

Javaではインスタンスの生成はnew演算子を使って行い、コンストラクタに定義された処理が実行されます。コンストラクタをテストする場合は、インスタンスの生成が実行フェーズに相当します(リスト3.19)。

インスタンスはテスト対象オブジェクト(SUT)でもあり、実測値でもあります。リスト3.19のように変数名をinstanceとすると理解しやすいでしょう。

注9 ➔ ExpectedException——詳細な例外を扱う(p.150)

リスト3.19 コンストラクタを検証するテスト

```
@Test
public void インスタンス化テスト() {
    // Exercise
    Person instance = new Person("Duke");
    // Verify
    assertThat(instance.getName(), is("Duke"));
    assertThat(instance.getAge(), is(-1));
    assertThat(instance.getEmail(), is(nullValue()));
}
```

JUnitの機能と拡張

Part 2

第4章
アーティファクト —— テスト比較演算子と`assertEquals()`

第5章
データベース操作 —— テスト実行方法の制御

第6章
データ構造操作 —— テスト実行順序と`Object.equals()`

第7章
データ操作手順 —— テスト実行順序と`Object.equals()`

第8章
データ操作手順 —— テスト実行順序と`Object.equals()`

第9章
データ操作手順 —— テスト実行順序と`Object.equals()`

第10章
データ操作手順 —— テスト実行順序と`Object.equals()`

第4章

アサーション 値を比較検証するしくみ

JUnitでは、AssertクラスのassertThatメソッドとMatcher APIを使ってアサーション(値の比較検証)を行います。特に、Matcher APIは実測値が期待される結果となっているかを検証するための重要なAPIです。Matcher APIを使いこなすことによって、テストの表現力と可読性が高まり、質の高いユニットテストを書くことができるようになります。

この章では、アサーションのしくみと、JUnitを使いこなすために覚えておきたいカスタムMatcherの作り方について解説します。

4.1 Assertによる値の比較検証

`org.junit.Assert` クラスは、JUnitで値の比較検証を行うための基本クラスです。Assertクラスには、アサーションを行うためのアサーションメソッドが多く定義されています。

アサーションメソッドは、staticインポートを利用することを想定したstaticメソッドです。これは、後述のMatcher APIと組み合わせることで、自然言語(英語)に近い表記を実現するためです。たとえば、「 $3 + 4$ 」の演算結果が「7」となることを検証するアサーションのコードを、staticインポートを使わずに記述すると次のようになります。

```
Assert.assertThat(3 + 4, CoreMatchers.is(7));
```

一方、staticインポートを利用すると、次のように記述できます。

```
assertThat(3 + 4, is(7));
```

このコードは「assert that ‘ $3 + 4$ ’ is 7」と読むことができます。このように、テストコードの可読性を高くするためにも、アサーションメソッドはstaticインポートして利用してください。

assertThat——汎用的な値の比較検証

Assert クラスの assertThat メソッドは、値の比較検証を行う汎用的なメソッドです。JUnitによるユニットテストでは、値の比較検証のほとんどを assertThat メソッドで行うことができます。

assertThat メソッドは、型パラメータを持つメソッドで、2つの引数を持ちます。1つ目の引数には実測値を指定し、2つ目の引数には期待値との比較を行う Matcher オブジェクトを指定します(リスト 4.1)。

リスト 4.1 では、変数 `actual` を実測値として1つ目の引数に指定しています。2つ目の引数に記述されている `is("Hello World")` は、CoreMatchers に定義されている static メソッド「`is`」を呼び出しています。`is` メソッドは、期待値の文字列を引数として Matcher オブジェクトを返します。この Matcher オブジェクトは、オブジェクト同士を `equals` メソッドで比較する `org.hamcrest.core.Is` クラスのインスタンスです。

assertThat メソッドは、Matcher オブジェクトによる比較検証で値が一致しない場合、`AssertionError` を送出します。`AssertionError` はテストが失敗したことをフレームワークに通知する例外クラスです。一方、比較検証で値が一致した場合、`assertThat` メソッドは何も行いません。このとき、フレームワークではそのテストが成功したと見なします。

このように、JUnitのアサーションでは、比較のためのルールと比較検証を行うしくみを分離しています。比較のためのルールは、比較したいオブジェクトやプロジェクトごとに異なります。そこで、比較のためのルール

リスト 4.1 assertThatによる比較検証

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

public class AssertionTest {
    @Test
    public void assertion() {
        String actual = "Hello" + " " + "World";
        assertThat(actual, is("Hello World"));
    }
}
```

に適したMatcherオブジェクトを選択できる設計になっています。また、特定の比較のためのルールがJUnitで提供されていない場合は、カスタムMatcherを作成することもできます。一方、比較検証を行うしくみを変える必要はないため、ほとんどの場合でassertThatメソッドが使えます。

fail——テストを失敗させる

Assertクラスのfailメソッドは、無条件にそのテストを失敗させます。引数として文字列をとった場合は、その文字列がテスト失敗時のエラーメッセージとなります。

failメソッドは、何らかの理由でテストを無条件に失敗として扱いたい場合に利用します。たとえば、「テストメソッドを作成したがテストコードを記述していない」といった場合に利用します(リスト4.2)。こうすることで、テストが実装されていないことを明らかにできます。

failメソッドが呼び出されると、ただちにAssertionErrorが送出され、テストは失敗となります。

また、一部のテストケースでは、あるブロックが実行されないことをテストしたい場合があります。言い換えれば、あるブロックが実行された場合にテストを失敗したい場合です。このような場合にもfailメソッドを利用できます。

リスト4.3では、テスト対象オブジェクトに渡すRunnableオブジェクトのrunメソッドがタイムアウトの状態のときに実行されないことをテストしています。しかしながら、このように一見して何をテストしているのかが不明瞭なテストコードは、可能な限り避けるべきでしょう。

リスト4.2 failメソッドによるテストの失敗

```
@Test
public void なにか難しいけど重要なテストケース() {
    fail("TODO テストコードを実装する");
}
```

リスト4.3 特定のステップが呼び出されると失敗となるテスト

```

@Test(expected = IllegalStateException.class)
public void timeoutがtrueのときにロジックが実行されないこと() {
    // SetUp
    Runnable logic = new Runnable() {
        public void run() {
            fail("run が呼ばれてしまった");
        }
    };
    sut.timeOut = true;
    // Exercise
    sut.invoke(logic);
}

```

そのほかのアサーションメソッド

Assertクラスには、assertEqualsメソッドやassertTrueメソッドといった多くのアサーションメソッドが定義されています。しかしながら、これらのアサーションメソッドはJUnitの古いバージョンとの互換性のために残っているものです。JUnit 4で利用するアサーションメソッドは、本書で解説したassertThatメソッドとfailメソッドの2つと考えてよいでしょう。

4.2 Matcher APIによるアサーションの特徴

先ほど解説したように、JUnitのアサーションはAssertクラスのassertThatを利用して行います。このとき、2つ目の引数には値の比較検証を行うMatcherオブジェクトを指定します。このMatcherオブジェクトはMatcher APIにより提供されます。

Matcher APIは、「何らかの値に対して等価であるかを柔軟に比較検証する」ためのAPIです。もとは、Hamcrest^{注1}という独立したJUnitの拡張ライブラリに含まれていましたが、JUnit 4.4で本家に組み込まれました。

なお、JUnitにMatcher APIが導入される前までは、次のように2つの値

注1 <http://code.google.com/p/hamcrest/>

なお、「Hamcrest」は「matchers」のアナグラム(文字の順番を入れ替えて別の言葉を作る文字遊び)です。

を比較し一致するかを検証するアサーションでした。

```
assertEquals(expected, actual);
```

値を比較する処理は、`assertEquals`メソッドに実装されており、プリミティブ型であれば`==`オペレータによる比較、オブジェクトであれば`equals`メソッドによる比較が行われます。

ほとんどの場合では、このしくみで問題ありません。しかしながら、ユニットテストにおける値の比較検証では、一致している条件が自明でない場合もあります。たとえば、「特定のオブジェクトの比較を行いたいが特定のフィールドは比較から除外したい」といった状況です。また、`assertEquals`メソッドでオブジェクト同士を比較したとき、一致するか一致しないかはわかりますが、どのフィールドが一致していないかなどの詳細な情報を知ることはできません。さらに、比較するオブジェクトに`equals`メソッドがオーバーライドされていない場合は、想定しない結果となります。

Matcher APIはこれらの問題をJavaの枠組みの中で、可能な限り解決します。Matcher APIは、従来のアサートメソッドよりも可読性が高く、柔軟な比較を行えます。また、テスト失敗時のメッセージを詳細に output できます。そして、独自の比較処理を行うクラスを簡単に作成できます。

この節では、これらの特徴を掘り下げて解説します。

可読性の高い記述

Matcher APIを利用することで、より自然言語(英語)に近いテストコードとなり、可読性が向上します。

Matcher APIを利用したJUnit 4スタイルのアサーションは、次のようになります。

```
int expected = 7;
int actual = add(3, 4);
assertThat(actual, is(expected));
```

一方、JUnit 3スタイルのアサーションは次のようになります。

```
int expected = 7;
int actual = add(3, 4);
assertEquals(expected, actual);
```

一見、大きな違いがないようにも思えます。しかしながら、JUnit 4のスタイルでは Matcher API を活用することで、検証コードが自然言語の表記に近く、「assert that actual is expected」(実測値が期待値であると表明する)と読むことができます。テストコードが自然言語の表記に近いことで、テストケース(仕様)をテストコードへ変換しやすくなります。また、テストコードを読むときに仕様を自然な形で読むことができます。

なお、`assertEquals` メソッドでは、1つ目の引数が期待値(`expected`)であり、2つ目の引数が実測値(`actual`)です。`assertThat` メソッドと逆である点に注意してください。

柔軟な比較

ユニットテストのほとんどの場合で、実測値と期待値の比較検証は等価性、すなわち`equals` メソッドで比較します。しかしながら、単純な等価性のチェックでは厳密過ぎる場合や、独自ロジックで比較検証を行いたい場合もあります。このような場合、Matcher API を利用することで、可読性を損ねず、再利用可能で柔軟な比較検証を行うことができます。

たとえば、テスト対象メソッドの戻り値が String の List であり、「リストの任意の位置に文字列 "Hello" が含まれていること」を検証したいとします。Matcher API を使わない場合、次のように記述するでしょう。

```
List<String> actual = sut.getList();
assertTrue(actual.contains("Hello"));
```

このテストコードは正しいのですが、「リストの任意の位置に文字列 "Hello" が含まれていること」を「リストの`contains` メソッドに文字列 "Hello" を指定すると `true` となること」に置き換えていません。

一方、Matcher API を使うと、次のように記述できます。

```
List<String> actual = sut.getList();
assertThat(actual, hasItems("Hello"));
```

hasItems メソッドは is メソッドと同様に Matcher オブジェクトを返すファクトリメソッドです。ここでは、新しい語彙として「hasItems」を使うことで、検証したかった「リストの任意の位置に文字列 "Hello" が含まれていること」が直感的に伝わる記述となります。

このように Matcher API は、一貫した書式で柔軟な比較検証を行いうインターフェースを提供します。

● カスタム Matcher による、より柔軟な比較

Matcher API を使ってカスタム Matcher を作成すれば、より柔軟な比較検証を行うことができます。比較検証はカスタム Matcher クラスで実装するため、比較するクラスの equals メソッドをオーバーライドする必要もありません。

たとえば、データベースから取得したオブジェクトの場合、比較時に作成日時や IDなどを無視したくなるケースがあります。カスタム Matcher を作成すれば、そういった特殊な比較検証ロジックを簡単に実装できます^{注2}。

詳細な情報の提供

Matcher API 最後の特徴は、テストに失敗した場合に多くの情報を提供できることです。Matcher オブジェクトはオブジェクト同士の比較検証を行います。そして、比較結果が偽である場合に詳細情報をフレームワークに通知します。

assertEquals メソッドでは、比較検証した結果、一致しないことはフレームワークに通知されます。しかしながら、どのフィールドが異なるなどの詳細な情報までは通知されません。このため、テストが失敗した場合の問題の特定が遅れ、修正までの時間が多くかかります。

一方、カスタム Matcher を作成すれば、どのフィールドが異なるかなど

注2 ➔ カスタム Matcher の作成(p.71)

を詳細に通知できます。ユニットテストが失敗したとき、すばやくエラーメッセージから失敗の原因がわかれれば、開発がスムーズに進むでしょう^{注3}。

4.3 Matcher APIの使用

Matcher APIを使用する場合は、org.hamcrest.CoreMatchers クラスや org.junit.matchers.JUnitMatchers クラスなどの、Matcher のファクトリクラスを使います。これらのクラスに定義されたメソッドは static なファクトリメソッドであり、Matcher オブジェクトを生成します。

Matcher オブジェクトを生成するファクトリメソッドは、テストコードに組み込んで利用できると便利です。このため、static インポートは個々のメソッドを宣言するのではなく、ワイルドカードを利用するといでしまう。ただし、Eclipse で static インポートを行った場合にワイルドカードを利用するには設定の変更が必要です。

また、これらのMatcher を返すファクトリメソッドを Eclipse のお気に入り(Favorites)に登録すれば、コンテンツアシストの候補として呼び出せます。詳細は付録B「Eclipse の便利機能と設定」を参照してください。

CoreMatchers が提供する Matcher

この節では、org.hamcrest.CoreMatchers クラスに定義されている Matcher のファクトリメソッドから、よく使うメソッドを紹介します。

● is

CoreMatchers クラスの is メソッドは、最も自然な比較を行う Matcher を返します。すなわち、オブジェクトクラスの equals メソッドによる比較です。ユニットテストで比較検証を行うとき、多くのケースでこの is メソッドを使います。

```
assertThat(actual, is(expected));
```

^{注3} ⇒ カスタム Matcher の作成(p.71)

ただし、isメソッドは型パラメータを持つメソッドであるため、次のようにnullを指定するとコンパイルエラーとなります。

```
assertThat(actual, is(null));
```

実測値がnullであることを検証する場合は、後述のnullValueメソッドを利用します。

isメソッドには、Matcher型の引数を持つisメソッドがオーバーロードされています。このため、Matcherオブジェクトのファクトリメソッドを組み合わせることで、柔軟な表現ができます。次のコードは、否定の効果のあるMatcherオブジェクトを返すnotメソッドを組み合わせた一例です。

```
assertThat(actual, is(not(expected)));
```

このようにファクトリメソッドの名前を工夫することで、自然言語(英語)に近い表記ができます。

なお、独自に定義したクラスに関してisメソッドで比較検証を行うときは注意してください。そのクラスにequalsメソッドがオーバーライドされていない場合は、デフォルトのequalsメソッドの実装で比較が行われます。その実装はインスタンスの同一性であるため、ほとんどの場合では期待しない結果を返すことになります。

● nullValue

CoreMatchersクラスのnullValueメソッドは、nullであることを検証するMatcherを返します。これはJavaのジェネリクス構文の制約などで、「is(null)」と記述できないためです。

nullValueメソッドは単体でも利用できますが、通常は次のようにisメソッドと組み合わせて利用します。

```
assertThat(actual, is(nullValue()));
```

● not

CoreMatchersクラスのnotメソッドは、ほかのMatcherの評価値を反転させるMatcherを返します。notメソッドを使うことで、次のように否定条

件の検証時に自然言語(英語)に近い表記とすることができます。not メソッドは、nullValue メソッド同様に、is メソッドと組み合わせて利用します。

```
assertThat(actual, is(not(0)));
```

● notNullValue

CoreMatchers クラスの notNullValue メソッドは、null でないことを比較検証する Matcher を返します。

```
assertThat(actual, is(notNullValue()));
```

次のように not メソッドと nullValue メソッドを組み合わせて記述した場合と同等です。

```
assertThat(actual, is(not(nullValue())));
```

● sameInstance

CoreMatchers クラスの sameInstance メソッドは、実測値と期待値が同一のインスタンスであるかを比較する Matcher を返します。つまり、== 演算子を用いた比較を行い、オブジェクトの同一性を検証します。

```
assertThat(actual, is(sameInstance(expected)));
```

なお、プリミティブ型で sameInstance メソッドを利用する場合、ボクシング変換^{注4}に注意する必要があります。なぜならば、ボクシング変換されたプリミティブラッパー型のオブジェクトに対して、インスタンスの同一性を比較するからです。

たとえば、次のコードは多くの環境で検証が失敗します。

```
assertThat(128, is(sameInstance(128)));
```

理由は、128 の Integer オブジェクトが期待値と実測値のそれぞれで生成されるからです。多くの環境では、比較する値が 128 未満であれば検証は

^{注4} プリミティブ型を対応するオブジェクト型に自動変換するしくみ。

成功します。これは、そのJVMが128未満のint値に対して同一のオブジェクトを再利用するような実装だからです。このように、一見して理解できないコードは避けてください。

● `instanceOf`

`CoreMatchers` クラスの `instanceOf` メソッドは、実測値が期待するクラスのインスタンスと互換性のある型であるかを比較判定する `Matcher` を返します。つまり、`instanceof` 演算子を用いた比較を行い、オブジェクトの型を検証します。

次のコードでは、実測値が `Serializable` インタフェースを実装しているかを検証しています。

```
assertThat(actual, is(instanceOf(Serializable.class)));
```

なお、実測値が `null` の場合は必ず失敗します。

JUnitMatchers が提供する Matcher

続けて、`org.junit.matchers.JUnitMatchers` クラスに定義されている `Matcher` のファクトリメソッドから、よく使うメソッドを紹介します。

● `hasItem`

`JUnitMatchers` クラスの `hasItem` メソッドは、リストや配列など反復可能なオブジェクト(実測値)^{注5}に期待する値が含まれているかを判定する `Matcher` を返します。順序やほかの要素は無視し、実測値のリストの中に指定した要素が含まれていることを検証します。

```
List<String> actual = sut.getList();
assertThat(actual, hasItem("World"));
```

● `hasItems`

`JUnitMatchers` クラスの `hasItems` メソッドは、リストや配列など反復可能

注5 正確には繰り返し処理などを行うための `Iterable` インタフェースを実装したクラスのインスタンスとなります。

なオブジェクト(実測値)に期待する値が、複数含まれているかを判定する Matcher を返します。hasItem メソッドと異なる点は可変長引数をとることです。hasItems メソッドはすべての値が実測値に含まれているかを検証します。

```
List<String> actual = sut.getList();
assertThat(actual, hasItems("Hello", "World"));
```

リストの順序は問わずに必要な要素がすべて含まれているかどうかを検証したい場合に有効な Matcher です。

そのほかHamcrestが提供するMatcher

JUnit の拡張ライブラリである hamcrest-library では、コレクション、数値、テキストといった汎用的に使える Matcher を提供しています^{注6}。これらの Matcher は将来的に JUnit に組み込まれる可能性もありますが、JUnit 4.10 の時点では拡張ライブラリとして位置づけられています。表4.1(次ページ)に hamcrest-library で提供されている Matcher の一部を紹介します。

なお、Hamcrest 以外にも汎用的な Matcher を提供するライブラリはあるので、欲しい Matcher がない場合は Web で探してみるとよいでしょう。

4.4 カスタムMatcherの作成

プロジェクト固有の検証やより柔軟な検証を行いたい、詳細なエラーメッセージをレポートしたいといった場合は、カスタム Matcher を作ると便利です。カスタム Matcher は独立したクラスとして再利用できるため、プロジェクトの初期に作成すれば、効率良く開発が進められるでしょう。

本節では、日付の比較検証を行うカスタム Matcher を例に、カスタム Matcher の作り方を解説します。

^{注6} JUnit のバージョンによってはクラス名の競合問題が発生します(JUnit 4.10 では解決済み)。

表4.1 hamcrest-libraryで提供されているMatcher(抜粋)

クラス名	メソッド名	検証内容
org.hamcrest.collection. IsEmptyCollection	empty	空のCollectionであること
org.hamcrest.collection. IsCollectionWithSize	hasSize	Collection が期待するサイズであること
org.hamcrest.collection. IsMapContaining	hasEntry	Map が指定した Key と Value を持つこと
org.hamcrest.collection. IsMapContaining	hasKey	Map が指定した Key を持つこと
org.hamcrest.collection. IsMapContaining	hasValue	Map が指定した Value を持つこと
org.hamcrest.number. OrderingComparison	combratesEqualTo	比較して同じ数値であること
org.hamcrest.number. OrderingComparison	greaterThan	比較して期待値より大きいこと (算術記号の「>」に相当)
org.hamcrest.number. OrderingComparison	greaterThanOrEqualTo	比較して期待値より大きいか同じ 数値であること(算術記号の「≥」 に相当)
org.hamcrest.number. OrderingComparison	lessThan	比較して期待値より小さい数値で あること(算術記号の「<」に相当)
org.hamcrest.number. OrderingComparison	lessThanOrEqualTo	比較して期待値より小さいか同じ 数値であること(算術記号の「≤」 に相当)
org.hamcrest.number. IsCloseTo	closeTo	比較して期待値の範囲の数値で あること
org.hamcrest.text. IsEmptyString	isEmptyString	文字列が空文字列であること
org.hamcrest.text. IsEmptyString	isNullOrEmptyString	文字列が空文字列または null で あること
org.hamcrest.text. IsEqualIgnoringCase	equalIgnoringCase	大文字小文字を無視して文字列が 一致すること
org.hamcrest.beans. SamePropertyValuesAs	samePropertyValuesAs	Bean のプロパティがそれぞれ一 致すること

日付の比較検証を行うカスタムMatcherの要件

Java で日付や時刻を扱う場合、標準的には `java.util.Date` クラスを使用します。ところが、JUnit で Date 型のオブジェクトのアサーションを行うのは手間がかかります。特定日付の Date オブジェクトは作りにくく、さらにミリ秒までを保持するため、年月日だけを比較検証したい場合でも簡単にできません。そこで、次のようなカスタム Matcher を作成します。

- ・年／月／日をそれぞれint型で指定して日付のみを比較できる
- ・時／分／秒／ミリ秒は比較検証時に無視する
- ・比較に失敗した場合はyyyy/mm/dd表記で確認できる

カスタムMatcherの作成手順

カスタムMatcherは、org.hamcrest.Matcherインターフェースを実装したクラスとして作成します。ただし、Matcherインターフェースを直接インプリメンテーションすることは推奨されておらず、org.hamcrest.BaseMatcherクラスのサブクラスとします。

BaseMatcherクラスのサブクラスでは、matchesメソッドとdescribeToメソッドを実装します。matchesメソッドでは、値の比較検証を行います。describeToでは、比較が失敗した場合にフレームワークに通知する情報を作成します。

● IsDateクラスを作成する

まずはカスタムMatcherの実装クラスとして、BaseMatcherクラスを継承したIsDateクラスを作成します(リスト4.4)。BaseMatcherクラスは実測値の型を型パラメータとして持つジェネリクスクラスです。比較対象はDate型なので、型パラメータもDate型です。

リスト4.4 IsDateの作成

```
public class IsDate extends BaseMatcher<Date> {
    @Override
    public boolean matches(Object actual) {
        return false;
    }

    @Override
    public void describeTo(Description desc) {
    }
}
```

● ファクトリメソッドを作成する

次にカスタム Matcherを利用しやすくするように、staticなファクトリメソッドを作成します。IsDateクラスは、staticインポートを利用して、次のように記述できると便利です。

```
assertThat(actual, is(dateOf(2012, 1, 12)));
```

そこで、リスト4.5のようにIsDateクラスにファクトリメソッドを追加します。

プロジェクトで複数のカスタム Matcherを作るならば、ファクトリメソッドをまとめて定義したクラスを作成するとよいでしょう。

● コンストラクタを定義する

カスタム Matcherではファクトリメソッドで期待値または期待値を構成するために必要なパラメータを受け取ります。これらのパラメータはMatcherオブジェクトで保持しなければなりません。

dateOfメソッドでは年月日をそれぞれint型の引数で受け取ります。この情報をIsDateで保持できるようにコンストラクタを定義し、ファクトリメソッドを修正します(リスト4.6)。

● matchesメソッドを実装する

次に比較検証を行うmatchesメソッドを実装します。matchesメソッドは実測値を引数にとり、比較した結果が一致する場合はtrueを、一致しない場合はfalseを返します(リスト4.7)。

IsDateクラスのmatchesメソッドでは、実測値の年月日をそれぞれ期待値としてフィールドに保持していた値と比較し、すべてのフィールドが一

リスト4.5 IsDateクラスのファクトリメソッド

```
public class IsDate extends BaseMatcher<Date> {
    public static Matcher<Date> dateOf(int yyyy, int mm, int dd) {
        return new IsDate();
    }
    // 省略
}
```

4
5
6
7
8
9
10**リスト4.6** 期待値の保持

```
public class IsDate extends BaseMatcher<Date> {
    private final int yyyy;
    private final int mm;
    private final int dd;

    IsDate(int yyyy, int mm, int dd) {
        this.yyyy = yyyy;
        this.mm = mm;
        this.dd = dd;
    }

    @Override
    public boolean matches(Object actual) {
        return false;
    }

    @Override
    public void describeTo(Description desc) {
    }

    public static Matcher<Date> dateOf(int yyyy, int mm, int dd) {
        return new IsDate(yyyy, mm, dd);
    }
}
```

リスト4.7 matchesメソッドの実装

```
Object actual;

@Override
public boolean matches(Object actual) {
    this.actual = actual; ①
    if (!(actual instanceof Date)) return false;
    Calendar cal = Calendar.getInstance();
    cal.setTime((Date) actual);
    if (yyyy != cal.get(Calendar.YEAR)) return false;
    if (mm != cal.get(Calendar.MONTH) + 1) return false;
    if (dd != cal.get(Calendar.DATE)) return false;
    return true;
}
```

致する場合に true を返します。値が一致しない場合には false を返します。そのとき、後述の describeTo メソッドが呼び出されます。このため実測値や報告するための情報をフィールドに待避しておく必要があります(リスト 4.7①)。

● describeTo メソッドを実装する

最後に、matches メソッドが false を返した場合に呼ばれる describeTo メソッドを実装します(リスト 4.8)。describeTo メソッドは、フレームワークに比較検証が失敗した理由を通知するためのメソッドです。

なお、describeTo メソッドは引数として、Description オブジェクトを受け取ります。このオブジェクトは、検証失敗の情報を保持するためのオブジェクトです。このオブジェクトにテスト失敗時のメッセージに含める情報を追加します。

● テスト失敗メッセージの確認

テスト失敗時のメッセージを確認するため、次のコードを JUnit で実行してください。

```
assertThat(new Date(), is(dateOf(2011, 2, 10)));
```

テストは失敗し、次のようなメッセージが表示されます。

```
java.lang.AssertionError:  
Expected: is "2011/02/10" but actual is "2012/03/08"  
      got: <Thu Mar 08 23:02:49 JST 2012>
```

リスト 4.8 検証失敗の詳細情報を記録する

```
@Override  
public void describeTo(Description desc) {  
    desc.appendValue(String.format("%d/%02d/%02d", yyyy, mm, dd));  
    if (actual != null) {  
        desc.appendText(" but actual is ");  
        desc.appendValue(  
            new SimpleDateFormat("yyyy/MM/dd").format((Date) actual));  
    }  
}
```

このメッセージのフォーマットは、`assertThat` メソッドにハードコーディングされているため変更ができませんが、次のようにになっています。

```
java.lang.AssertionError:  
  Expected: is <><Descriptionオブジェクトの文字列表記>>  
  got: <><実測値のtoString結果>>
```

`describeTo` メソッドでは、`<><Descriptionオブジェクトの文字列表記>>` を構成するために必要な情報を `Description` オブジェクトに与えます。

`Description` オブジェクトは、`appendValue` メソッドや `appendText` メソッドといった `StringBuilder` クラスと似たメソッドを持ちます。`appendValue` メソッドは、オブジェクトを引数としてとり、文字列表記を追加するメソッドです。値であることを表すために出力時にはダブルクォーテーションで囲まれた文字列となる点が、`appendText` メソッドと異なります。`appendText` メソッドは単純に文字列を追加します。

なお、`<><実測値のtoString結果>>` については出力フォーマットをカスタマイズできません。

ソースを見ただけではどんな情報が出力されるかイメージがつきにくいため、実行結果を見ながら実装を修正していくといでしよう。

カスタムMatcherのメリット

このようにカスタム `Matcher` を利用すると、柔軟な比較検証ルールを簡単に実装できます。たとえば、データベースから取得したオブジェクトでは、自動生成された ID や最終更新日時などを比較対象から除外したい場合があります。そのような場合には、比較対象のフィールドを制限するカスタム `Matcher` を作り、比較に失敗したフィールドを情報として報告すれば便利です。

また、テスト失敗時に詳細な情報を出力できるため、デバッグ時間が大幅に短縮できます。再利用もしやすいため、さまざまなプロジェクトで利用できます。

IsDateによる比較の実行結果

それでは、リスト4.9を用いてテスト失敗時にどのように詳細な情報が表示されるかを確認してみましょう。

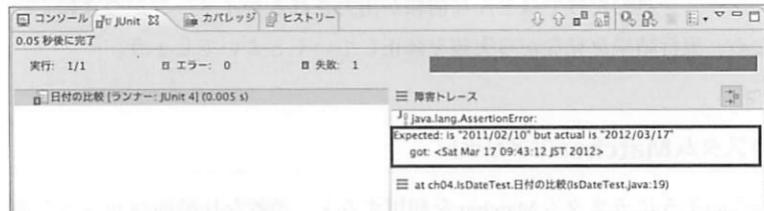
このテストを実行すると、図4.1のようなメッセージが表示されます。このように、yyyy/mm/ddフォーマットで実測値と期待値が表示されるため、どの部分が一致していないかがわかりやすくなります。

最後に今回作成したカスタムMatcherのソースコード全体を示します(リスト4.10)。

リスト4.9 IsDateのテストサンプル

```
@Test
public void 日付の比較() throws Exception {
    Date date = new Date();
    assertThat(date, is(dateOf(2011, 2, 10)));
}
```

図4.1 IsDateによる比較の実行結果



リスト4.10 IsDateクラス

```

public class IsDate extends BaseMatcher<Date> {

    private final int yyyy;
    private final int mm;
    private final int dd;
    Object actual;

    IsDate(int yyyy, int mm, int dd) {
        this.yyyy = yyyy;
        this.mm = mm;
        this.dd = dd;
    }

    @Override
    public boolean matches(Object actual) {
        this.actual = actual;
        if (!(actual instanceof Date)) return false;
        Calendar cal = Calendar.getInstance();
        cal.setTime((Date) actual);
        if (yyyy != cal.get(Calendar.YEAR)) return false;
        if (mm != cal.get(Calendar.MONTH) + 1) return false;
        if (dd != cal.get(Calendar.DATE)) return false;
        return true;
    }

    @Override
    public void describeTo(Description desc) {
        desc.appendValue(String.format("%d/%02d/%02d", yyyy, mm, dd));
        if (actual != null) {
            desc.appendText(" but actual is ");
            desc.appendValue(
                new SimpleDateFormat("yyyy/MM/dd").format((Date) actual));
        }
    }

    public static Matcher<Date> dateOf(int yyyy, int mm, int dd) {
        return new IsDate(yyyy, mm, dd);
    }
}

```

第5章

テストランナー テスト実行方法の制御

JUnitによるユニットテストは、テストクラスにテストメソッドとして定義されます。Eclipseを利用していれば、テストの実行方法を意識しなくともテストを実行できます。しかしながら、テストの実行方法をカスタマイズする場合は、JUnitのテストを実行するしくみについて知る必要があります。

この章では、JavaのプログラムとしてJUnitのテストを実行する方法と、テストを実行するしくみであるテストランナーについて解説します。

5.1 コマンドラインからのJUnitの実行

Javaの開発ではEclipseなどのIDEを利用することが多いため、JUnitのテストをコマンドラインから実行する機会はほとんどありません。JUnitによるユニットテストの流れを理解するためにも、コマンドラインからJUnitのテストを実行する方法をはじめに確認しておきます。

JUnitによるユニットテストをコマンドラインから実行するには、`org.junit.runner.JUnitCore`クラスを使用します。このとき、実行するテストクラスの完全修飾名を起動引数に与えます。たとえば、テストクラス `SomeTest` をコマンドラインから実行するには次のコマンドを入力します^{注1}。

```
$ java org.junit.runner.JUnitCore test.SomeTest
```

次のように複数のテストクラスを指定することもできます。

```
$ java org.junit.runner.JUnitCore test.SomeTest test.OtherTest
```

^{注1} ここでは省略していますが、「`-cp ./lib/junit-4.10.jar`」のように、クラスパスにJUnitのJARファイルを追加する必要があります。

テストクラスからテストを実行する

リスト5.1のようにテストクラスにエントリーポイント(mainメソッド)を定義することで、次のようにテストを実行できます。

```
$ java ch05.CalcTest
```

mainメソッドでは、JUnitCoreクラスのmainメソッドを呼び出します。

JUnitCoreクラスのしくみ

JUnitCoreクラスのmainメソッドが実行されると、JUnitCoreクラスは起動引数で指定したテストクラスからTestアノテーションのついたメソッド、すなわちテストケースを収集します。続けて、各テストケースが実行され、テスト結果がコンソールに出力されます。

通常の開発ではIDE(Eclipse)やビルドツールを利用してテストを実行するため、JUnitCoreクラスを利用するシーンはほとんどありません。しかし、JUnitによるユニットテストが、次の流れで行われていることは、覚えておきましょう。

- ① テストケースの収集
- ② テストの実行
- ③ テスト結果の出力(レポート)

リスト5.1 実行可能なテストクラス

```
public class CalcTest {
    @Test
    public void addに3と4を与えると7を返す() {
        Calc sut = new Calc();
        assertThat(sut.add(3, 4), is(7));
    }

    // エントリーポイント
    public static void main(String[] args) {
        org.junit.runner.JUnitCore.main(CalcTest.class.getName());
    }
}
```

5.2 テストランナーとは？

JUnitCore クラスでは、テストクラスからテストケースを収集します。しかししながら、ソフトウェア開発におけるユニットテストでは、テストクラス単位でテストを実行するだけでは不十分です。プロジェクトの全テストケースの実行、特定のテストケースに絞った実行、特定のテストケースを除外しての実行などが必要です。

JUnit では、このようなテストの実行に関するカスタマイズ要求を満たすために、テストランナーというしくみを提供しています。テストランナーは、主にテストクラスに定義されたテストケースの実行に関する制御を行います。JUnit では標準でいくつかのテストランナーを提供しています。例えば、Suite テストランナーを利用すれば、複数のテストクラスに定義されたテストメソッドをまとめて実行することができます^{注2}。

テストランナーの設定

テストランナーはテストクラスごとに設定します。テストランナーを指定するには、テストクラスに org.junit.runner.RunWith アノテーションを付与し、値としてテストランナーとなるクラスを指定します。

たとえば、テストランナークラスのひとつである org.junit.runners.JUnit4 クラスを指定すると、リスト 5.2 のようになります。JUnit4 クラスは JUnit の標準的なテストランナークラスで、テストクラスから Test アノテーションの付与されたすべてのメソッドをテストケースとして収集してテストを実行します。テストクラスで RunWith テストランナーの付与を省略した場合は、テストランナーとして JUnit4 クラスが利用されます。

^{注2} JUnit 3 では同様の機能を TestSuite クラスを継承したテストクラスを作ることで実現していましたが、JUnit 4 ではテストランナーを使うように変更されました。

リスト 5.2 RunWith アノテーションによるテストランナーの指定

```
@RunWith(JUnit4.class)
public class SomeTest {
}
```

テストランナーは、テストクラスのテストケースを「どのように実行するか」を制御するしくみです。個々のテストケースの拡張を行うには第9章で解説する「ルール」を使用します。

5.3 JUnitが提供するテストランナー

JUnitが標準で提供するテストランナーは、新しいテストケースの記述方法をプログラマに提供します。テストの特徴を捉え、適切なテストランナーとテストケースの記述方法を選択することで、テストコードの表現力とメンテナンス性が大幅に高まるでしょう。

本節ではテストランナーの種類と、どのような目的で利用するかについて、簡単に紹介します。それぞれのテストランナーを使った詳細なテスト手法については、対応する章を参照してください。

Column

ユニットテストの実行結果をカスタマイズする

ユニットテストの実行結果は、プログラマが目視で確認して活用するだけでなく、レポートとして出力したり、別のツールで取り込んで分析することで有効に活用できます。そのためには、EclipseなどのIDEへの出力、独自のGUIへの出力、各種ツールへ連携するためのXMLやJSON(*JavaScript Object Notation*)形式での出力など、テストの実行結果を目的に応じたフォーマットで出力することが必要です。

しかしながら、JUnitではテストクラスの収集とテスト結果の出力に関する機能はありません。これはテスト結果の出力フォーマットに対して、ツールごとに求められる要件が異なるからです。EclipseなどのIDEではテスト結果がリアルタイムにGUIに反映されることが求められます。一方、Mavenなどのビルド支援ツールではXMLやJSONなど、さらにはほかのツールに連携しやすいことが求められます。このような理由から、ユニットテストの実行結果の出力はツールごとに独自実装しています。

本書では扱いませんが、独自の実行結果を出力したい場合は、カスタムテストランナーを作成してください。カスタムテストランナーを作成するには、org.junit.runner.Runnerインターフェースを実装します。

JUnit4 —— テストクラスの全テストケースを実行する

`org.junit.runners.JUnit4` クラスは、JUnitの標準的なテストを実行するテストランナーです。明示的にテストランナーを指定しなかった場合は、このJUnit4テストランナーが使用されます。

JUnit4テストランナーでは、テストクラスから次の条件を満たすメソッドをテストケースとして収集し、実行します。

- `public` メソッドである
- `org.junit.Test` アノテーションが付与されている
- 戻り値が `void` で引数を持たない

Suite —— 複数のテストクラスをまとめて実行する

`org.junit.runners.Suite` クラスは、複数のテストクラスを束ねてテストを実行するテストランナーです。

デフォルトのJUnit4テストランナーでは1つのテストクラスが対象です。しかしながら、テストクラスが増えていくと、パッケージ内あるいはプロジェクトに含まれるすべてのテストケースのテストを行いたくなります。そのような場合にはSuiteクラスを用います。

● テストスイートクラスでテストクラスを束ねる

JUnitでいくつかのテストクラスを束ねるには、テストスイートクラスを作成します。このとき、テストスイートクラスには、`RunWith`アノテーションで Suite テストランナーを指定します。テストスイートクラスは、複数のテストクラスの集合であるため、慣例として「`~Tests`」のように、クラス名を複数形とします。特に「`AllTests`」は、よく使われるテストスイートクラス名です。

たとえば、`FooTest` と `BarTest` の2つのテストクラスを束ねたテストスイートクラスは、リスト 5.3 のように記述します。テストスイートクラスではメソッドなどの実装は不要です。クラスに2つのアノテーションを宣言してください。1つ目は、`RunWith`アノテーションによる Suite テストラン

ナーの指定です。2つ目は、org.junit.runners.Suite.SuiteClasses アノテーションによるテストスイートに含めるテストクラスの指定です。

● テストスイートクラスの利用

テストスイートクラスは、テストクラスです。つまり、通常のテストクラスを指定する場合と同様に、コマンドラインから実行できます。

```
$ java org.junit.runner.JUnitCore test.AllTests
```

また、テストスイートクラスでは、ほかのテストスイートクラスを指定することもできます。パッケージごとにパッケージ内の全テストクラスを含む AllTests を作成し、それらのテストスイートクラスをすべて含むテストスイートクラスを作成することで、プロジェクト内の全テストケースを実行するテストスイートを作成できます(リスト 5.4)。

ただし、プロジェクト全体のテストを行う場合は、テストクラスを作成するごとにテストスイートクラスに追加する必要があります。テストスイートにテストクラスが自動的に追加されるわけではないので注意が必要です。

● テストスイートとビルド支援ツール

テストスイートはプロジェクト全体のテストを実行するために使用できますが、手動でテストクラスを追加しなければなりません。また、プロジ

リスト 5.3 テストスイートクラス

```
@RunWith(Suite.class)
@SuiteClasses({ FooTest.class, BarTest.class })
public class AllTests { }
```

リスト 5.4 プロジェクト全体のテストを行うテストスイート

```
/** * プロジェクト全体のテストを行うテストスイート */
@RunWith(Suite.class)
@SuiteClasses({aaa.AllTests.class, bbb.AllTests.class, ccc.AllTests.class})
public class AllTests { }
```

エクトの構成管理上、コンパイルの実行や依存ライブラリの管理なども必要であるため、Mavenなどのビルドツールを利用する事が現実的です。

MavenなどのビルドツールではJUnitによるユニットテストの実行をサポートしています。プロジェクト全体のテストも含め、柔軟なテストケースの収集と実行、そして詳細なレポート作成も行うことができます^{注3}。

なお、ほとんどのビルドツールでは、テストクラスを「Test」で終わるという命名規則で識別します。テストクラスは「～Test」という名前で統一してください。

Enclosed——構造化したテストクラスを実行する

`org.junit.experimental.runners.Enclosed` クラスは、構造化したテストクラスのテストを実行するテストランナーです。

標準的なテストランナーの場合、テストケースはテストクラスの `public` メソッドとして定義します。Enclosed クラスをテストランナーとして指定した場合、ネストしたクラス(*nested class*)を定義し、テストケースは各ネストしたクラスに定義します。ネストしたテストクラスはいくつでも定義できるため、テストクラスのテストケースが増えてきた場合に、同じ特徴を持つテストケースをまとめ、構造化できます。

リスト 5.5 は、`ItemStock` クラスのテストクラスです。テストケースを「空の場合」と「商品 A を 1 件含む場合」の 2 つのコンテキスト(文脈)に分け、それぞれネストしたテストクラスを定義しています。

各ネストしたテストクラスは標準的なテストクラスとして作成します。したがって、共通の初期化処理は `setUp` メソッドに抽出できます。同じ初期化処理を行うテストケースをまとめる事ができるため、可読性の高いテストコードを作成できるようになります。Enclosed クラスによるテストの構造化は、テストケースが増えてきた場合に大きな効果をもたらします。詳細は第 6 章で解説します。

注3 ⇒ Mavenによるビルドプロセスの自動化(p.277)

リスト5.5 Enclosedで構造化されたTestClass

```

@RunWith(Enclosed.class)
public class ItemStockTest {

    public static class 空の場合 {
        ItemStock sut;

        @Before
        public void setUp() throws Exception {
            sut = new ItemStock();
        }

        @Test
        public void size_Aが0を返す() throws Exception {
            assertThat(sut.size("A"), is(0));
        }

        @Test
        public void contains_Aはfalseを返す() throws Exception {
            assertThat(sut.contains("A"), is(false));
        }
    }

    public static class 商品Aを1件含む場合 {
        ItemStock sut;

        @Before
        public void setUp() throws Exception {
            sut = new ItemStock();
            sut.add("A", 1);
        }

        @Test
        public void sizeが1を返す() throws Exception {
            assertThat(sut.size("A"), is(1));
        }

        @Test
        public void contains_Aはtrueを返す() throws Exception {
            assertThat(sut.contains("A"), is(true));
        }
    }
}

```

4
5
6
7
8
9
10

Theories —— パラメータ化したテストケースを実行する

`org.junit.experimental.theories.Theories` クラスは、パラメータ化テストをサポートするテストランナーです。パラメータ化テストとは、テストケースとテストデータを分離し、同じテストメソッドを複数のパラメータで再利用して使うテクニックです。

JUnitでは、テストランナーとして `Theories` クラスを指定することで、パラメータを持つテストメソッドを定義できるようになります。このとき、`org.junit.Test` アノテーションの代わりに `org.junit.experimental.theories.Theory` アノテーションを使用します。テストメソッドに渡されるパラメータは、`org.junit.experimental.theories.DataPoints` アノテーションの付与された static フィールドなどで定義します。

リスト 5.6 は、パラメータとして int 型の配列を持つテストメソッドを定義しています。テストメソッドでは、配列の 0 番目と 1 番目をテスト対象メソッドの入力値とし、2 番目の値を期待値として検証を行っています。

このように、複数の異なる入力値に対する検証を行いたい場合に効率良く実装できます。入力値や期待値を外部ファイルとして定義することもできます。パラメータ化テストは第 8 章で詳細に扱います。

リスト 5.6 Theories によるパラメータ化テスト

```
@RunWith(Theories.class)
public class CalcTheoriesTest {
    @DataPoints
    public static int[][] VALUES = {
        {0, 0, 0},
        {0, 1, 1},
        {1, 0, 1},
        {3, 4, 7},
    };

    @Theory
    public void add(int[] values) throws Exception {
        Calc sut = new Calc();
        assertThat(sut.add(values[0], values[1]), is(values[2]));
    }
}
```

Categories——特定カテゴリのテストクラスをまとめて実行する

`org.junit.experimental.categories.Categories` クラスは、テストケースをカテゴリ化し、実行するテストケースをフィルタリングするためのテストランナーです。

可能であれば、ユニットテストは常にすべてのテストケースを実行すべきです。しかしながら、テスト数が増えてくると実行時間がかかりすぎて現実的ではありません。このため、各テストケースに目印(カテゴリ)を設定し、テストランナーでどのテストケースを実行するかを制御します。

● カテゴリクラスの作成

目印となるカテゴリクラスは、どのようなクラス(またはインターフェース)でもかまいません。リスト 5.7 は、遅いテストに付与するための `SlowTests` インタフェースです。

● カテゴリクラスの指定

このようにして作成したクラスを利用して、リスト 5.8 のように、`org.junit.experimental.categories.Category` アノテーションを宣言します。これで、テストケースがカテゴリ化されました。

なお、`Category` アノテーションはテストクラスに設定することもできます。その場合、テストクラスに定義されたすべてのテストメソッドがそのカテゴリに属します。

● カテゴリ化テストの実行

カテゴリ化テストを実行するには、`Suite` テストランナーを使う場合と同様に `Suite.SuiteClasses` アノテーションで対象となるテストクラスを指定します。

リスト 5.9 では、`ExcludeCategory` アノテーションで実行から除外するカ

リスト 5.7 目印となるカテゴリクラス

```
public interface SlowTests {  
}
```

テゴリの宣言を行っています。このテストスイートを実行すると、SlowAndFastTest クラスに定義されたテストケースから SlowTests と宣言されていないテストだけが実行されます。カテゴリ化テストの詳細に関しては第10章を参照してください。

なお、テストスイートと同様に、ビルドツールの機能を利用してカテゴリ化テストを実行できます。その場合、Category アノテーションによるカテゴリ化を行えば、実行するテストケースの収集はビルドツールによって行われます。したがって、テストスイートクラスは作成する必要はありません。

リスト5.8 テストメソッドのカテゴリ化

```
public class SlowAndFastTest {
    @Test
    public void fastTest_01() throws Exception {
    }

    @Test
    @Category(SlowTests.class)
    public void slowTest_01() throws Exception {
        fail();
    }

    @Test
    @Category(SlowTests.class)
    public void slowTest_02() throws Exception {
        fail();
    }
}
```

リスト5.9 カテゴリ化テストを実行するテストスイートクラス

```
@RunWith(Categories.class)
@ExcludeCategory(SlowTests.class)
@SuiteClasses(SlowAndFastTest.class)
public class CategoriesTests {
}
```

Column

JUnitのorg.junit.experimentalパッケージ

JUnitのorg.junit.experimentalパッケージは、実験的(experimental)な機能のためのパッケージです。Enclosedクラスをはじめ、TheoriesクラスやCategoriesクラスもこの実験的パッケージに含まれます。このため、今後のバージョンアップで、仕様の変更やパッケージの移動が行われる可能性があります。

どの機能も非常に実用的な機能であるため、今後のリリースで消失する可能性は低いと思われますが、メジャーバージョンアップ時など、該当するクラスが見つからない場合はリリースノートやJavadocを確認するようにしましょう。

4
5
6
7
8
9
10

第6章

テストのコンテキスト テストケースの構造化

テストコードは、プロダクションコードと同様に、可読性が高くなるようにメンテナンスすべき対象です。しかしながら、テストコードをプロダクションコードと同じ手法で整理すると、逆に可読性が低くなる場合があります。

この章では、共通したテストのコンテキストを持つテストケースをグループ化し、可読性の高いテストコードを記述する方法を解説します。

6.1 テストのコンテキストとは？

コンテキスト(*context*)とは、一般に「文脈」と訳される単語です。Java EE のServletContextクラスがよく知られるように、内部状態や前提条件などを表すもの^{注1}として使われます。

テストのコンテキストは、テストに関連する内部状態や前提条件を表すものです。テストの事前条件や状態、テストの入力データ、期待値となるデータなどをすべて含みます。

6.2 テストケースの整理

テストクラスに定義されたテストケースが増加していくと、テストコード自体が長く読みにくいものとなります。また、テストコードは似たようなコードが多いため、重複が多くなりがちです。さらに、これらの重複を取り除きすぎると各テストケースの意図が不明瞭になってしまいます。このため、各テストケースの独立性を保つつつ、テストコードを読みやすく整理する必要があります。

注1 ServletContextは、サーブレットの対応バージョンや初期化パラメータを保持します。

4
5
6
7
8
9
10

テストクラスの構造化

一般的に、クラスベースのプログラミング言語のユニットテストでは、テスト対象となるクラスごとに対応するテストクラスを作成します。Javaでは、慣習としてテストクラス名は「テスト対象クラス + Test」です。たとえば、テスト対象クラスが ItemStock クラスであれば、テストクラスは ItemStockTest クラスとなります。この命名規約に従うことで、テストクラスとテスト対象クラスの関係が明確になります。また、Eclipse で QuickJUnit プラグインを導入すれば、キーボードショートカット [Ctrl]+[9] でテストクラスとテスト対象クラスを相互に移動できます。

ところが、このような命名規約でテストクラスを作成し、テストケースを追加していくと、テストクラスが非常に大きくなってしまいます。テストコードをどの程度書けば十分かは、ユニットテストの方針やテスト対象クラスの性質にもよりますが、プロダクションコードの数倍から数十倍程度にはなることもあります。

また、テストコードでは似たような初期化処理が多くなります。さらに、Before アノテーションで定義する共通の初期化処理は1つが望ましいため^{注2}、きれいな形にまとめることが困難です。このような状態が続くと、テストコードの可読性が低下し、誰も読めないテストコードとなります。

多くのテストケースが定義されたテストコードの可読性が低い理由は、すべてのテストメソッドを同列に定義しているからです。これは1つのフォルダに大量のファイルがある状況と同じです。多くのファイルを整理するには、目的や内容ごとにサブフォルダを作成します。これはテストコードでも同様です。

テストケースが増えてきたならば、何らかの基準でグループ化し、テストクラスを構造化することで可読性を高く保つことができます。

^{注2} Before アノテーションを定義したメソッドを複数定義することはできますが、実行順序が制御できず、すべて実行されるため。

テストケースをグループ化する

テストケースをグループ化する方針としては、大きく次の2つがあります。

- ・テストケースで検証する操作(メソッド)単位でグループ化する
- ・テストケースを共通の初期化処理を含むものでグループ化する

たとえば、商品のストックを管理する ItemStock インタフェース(リスト 6.1)に対するテストコードを整理することを考えます。

このインターフェースに対するテストケースを、次のように抽出したとします。

リスト6.1 ItemStockインターフェース

```
public interface ItemStock {
    /**
     * 商品と数量を指定して追加する
     * @param item 商品名
     * @param num 追加する数量
     */
    void add(String item, int num);

    /**
     * 商品を指定して、商品の在庫数を返す
     * @param item 商品名
     * @return 在庫数、登録されていない場合は0
     */
    int size(String item);

    /**
     * 商品の在庫が残っている場合にtrueを返す
     * @param item 商品名
     * @return 在庫が1以上の場合にtrue
     */
    boolean contains(String item);
}
```

- ItemStock が初期状態のとき、size に商品 A を指定すると 0 を返す
- ItemStock が初期状態のとき、contains に商品 A を指定すると false を返す
- ItemStock が初期状態のとき、add で商品 A を 1 追加すると商品 A の size が 1 を返す
- ItemStock に商品 A が 2 個含まれるとき、size に商品 A を指定すると 2 を返す
- ItemStock に商品 A が 2 個含まれるとき、contains に商品 A を指定すると true を返す
- ItemStock に商品 A が 2 個含まれるとき、add で商品 A を 3 追加すると商品 A の size が 5 を返す

これを検証する操作(メソッド)でグループ化すると表6.1となります。一方、共通の初期化処理でグループ化すると表6.2となります。どちらの方針であっても、テストはグループ化されます。しかしながら、各グループのテストケースを別個のテストクラスに定義することでテストケースの構

表6.1 検証する操作(メソッド)によるグループ化

検証するメソッド	初期化処理	アサーション
size	ItemStock を作成する	size に商品 A を指定すると 0 を返す
	ItemStock を作成し、商品 A を 2 個追加する	size に商品 A を指定すると 2 を返す
contains	ItemStock を作成する	contains に商品 A を指定すると false を返す
	ItemStock を作成し、商品 A を 2 個追加する	contains に商品 A を指定すると true を返す
add	ItemStock を作成する	add で商品 A を 1 追加すると商品 A の size が 1 を返す
	ItemStock を作成し、商品 A を 2 個追加する	add で商品 A を 3 追加すると商品 A の size が 5 を返す

表6.2 共通の初期化処理によるグループ化

初期化処理	検証するメソッド	アサーション
ItemStock を作成する	size	size に商品 A を指定すると 0 を返す
	contains	contains に商品 A を指定すると false を返す
	add	add で商品 A を 1 追加すると商品 A の size が 1 を返す
ItemStock を作成し、商品 A を 2 個追加する	size	size に商品 A を指定すると 2 を返す
	contains	contains に商品 A を指定すると true を返す
	add	add で商品 A を 3 追加すると商品 A の size が 5 を返す

造化を行うならば、大きな違いがあります。

検証する操作(メソッド)でのグループ化では、テストケースごとにテストの前提条件が異なるため、初期化処理を共通化できません。一方、共通の初期化処理でのグループ化では、初期化処理を共通化できます。

テストの前提条件を初期化メソッドに抽出すると、テストメソッドの見通しが良くなります。また、各テストケースで前提条件が共通しているならば、効率良くテストコードの重複を減らすことができます。したがって、初期化処理を共通化できるように、テストケースは共通の初期化処理でグループ化するべきです。

しかしながら、テストケースを複数のテストクラスに分割して整理しようとすると、テストクラスを複数定義する必要があります。これは、テストクラスの命名規則の観点で望ましくありません。この問題を解決するためのしくみが、次項で解説する Enclosed テストランナーを利用したテストクラスの構造化です。

Enclosedによるテストクラスの構造化

`org.junit.experimental.runners.Enclosed` クラスは、ネストしたクラスをテストクラスとして扱うことのできるテストランナーです。Enclosed テストランナーを利用すれば、初期化処理が共通しているテストケースをネストしたクラスにまとめ、テストクラスを構造化できます。

テストクラスの構造化を行うためには、外側のテストクラスにテストランナーとして、`Enclosed` クラスを設定します。このとき、外側のクラス名は標準的なテストクラスの命名規約に従い「テスト対象クラス名 + Test」とします。ネストしたクラスの名前は、グループ化するテストケースの前提条件を表す名前とします。たとえば、ネストしたクラスの名前を「初期状態の場合」とすれば、読みやすく理解しやすいテストコードとなります。

リスト 6.2 は `ItemStock` のテストケースをネストしたクラスで構造化したテストクラスです。テストケースは共通の初期化処理で分類することにより、「空の場合」と「商品 A を 1 件含む場合」の 2 つのテストクラスにグループ化されました。それぞれのテストクラスは、共通の初期化処理やテストデータ、すなわち共通のテストのコンテキストで構造化されています。

リスト6.2 構造化されたItemStockのテストクラス

```

@RunWith(Enclosed.class)
public class ItemStockTest {

    public static class 空の場合 {
        ItemStock sut;

        @Before
        public void setUp() throws Exception {
            sut = new ItemStockImpl();
        }

        @Test
        public void size_Aが0を返す() throws Exception {
            assertThat(sut.size("A"), is(0));
        }

        @Test
        public void contains_Aはfalseを返す() throws Exception {
            assertThat(sut.contains("A"), is(false));
        }
    }

    public static class 商品Aを1件含む場合 {
        ItemStock sut;

        @Before
        public void setUp() throws Exception {
            sut = new ItemStockImpl();
            sut.add("A", 1);
        }

        @Test
        public void sizeが1を返す() throws Exception {
            assertThat(sut.size("A"), is(1));
        }

        @Test
        public void contains_Aはtrueを返す() throws Exception {
            assertThat(sut.contains("A"), is(true));
        }
    }
}

```

4
5
6
7
8
9
10

また、Eclipseのクイックアウトライン機能(エディタにフォーカスがある状態で $[\text{Ctrl}]+[\text{O}]$)を使えば、テストコンテキストで構造化されたテストクラスは図6.1のように表示されます。コンテキストごとに、テスト対象クラスの仕様がわかりやすく、ドキュメントとしての価値も高いことが確認できるでしょう。

なお、ネストしたクラスに、さらにEnclosedテストランナーを指定することもできます。しかしながら、Enclosedテストランナーで3階層以上に構造化することは避けるべきです。3階層以上に構造化が必要であったり、コンテキストごとのテストケースが多すぎるのであれば、テスト対象クラスの責務が大きすぎないか再検討してください。

6.3 コンテキストのパターン

テストのコンテキストは比較的パターン化しやすいため、パターンを覚えておくとテストコードを整理するのも簡単です。本節では、そのようなテストの分類パターンについて解説します。

図6.1 クイックアウトライン

```

ItemStockTest.java
1 package ch06;
2
3 import static org.hamcrest.CoreMatchers.*;
4
5 @RunWith(Enclosed.class)
6 public class ItemStockTest {
7     public static class ItemStock {
8         public void setUp() {
9             sut = new ItemStock();
10        }
11    }
12
13    @Before
14    public void setUp() {
15        sut = new ItemStock();
16    }
17
18    @Test
19    public void size() {
20        assertEquals(0, sut.size());
21    }
22
23    @Test
24    public void contains_A() {
25        sut.add("A");
26    }
27}

```

The screenshot shows the Eclipse IDE's Quick Outline view for the `ItemStockTest.java` file. The outline lists the following members:

- `ch06`
- `ItemStockTest`
 - `C# 空の場合`
 - `sut : ItemStock`
 - `setUp() : void`
 - `size_Aが0を返す() : void`
 - `contains_Aはfalseを返す() : void`
 - `商品Aを1件含む場合`
 - `sut : ItemStock`
 - `setUp() : void`
 - `sizeが1を返す() : void`
 - `contains_Aはtrueを返す() : void`

A tooltip at the bottom right of the outline view says: "選択されたメンバーを表示にするには 'NO' を押します".

4
5
6
7
8
9
10

共通のデータに着目する

入力値や期待値などのテストで、使用するデータに着目し、共通のテストケースをグループ化するパターンです。

最もわかりやすい例は、データベース接続を行うクラスのユニットテストです。データベース接続を行うユニットテストでは、テストの実行前にデータベースを特定の状態に設定する必要があります。たとえば、「あるテーブルが空の場合」や「10件のデータがある場合」などです。その際、テストケースごとにデータベースを初期化するコードを記述するのではなく、重複となります。また、テスト対象クラスには取得／更新／削除などいくつかのメソッドがあることが多いため、同じデータベースの状態に対し、各メソッドのテストを行うほうがわかりやすくなります。

リスト6.3は、ユーザ情報をデータベースから取得するための UserDao クラスです。このクラスのテストクラスをデータベースの値に着目して構造化するとリスト6.4のようになります。

このテストクラスでは、users テーブルのレコードが0件である場合と100件である場合の2つの状態について、それぞれgetList メソッドのテストを行っています。同じ getList メソッドであっても、それぞれの状態で返す値が異なるでしょう。このことは getList メソッド以外のメソッドについても同様です。したがって、テストをデータベースの状態ごとに整理しておくことで、テストケースの追加も状態の追加も行いやすくなります。

リスト6.3 UserDaoクラス(抜粋)

```
public class UserDao {
    public List<User> getList() {
        // 省略
    }
}
```

リスト6.4 UserDaoクラスのテストクラス

```

@RunWith(Enclosed.class)
public class UserDaoTest {

    public static class テーブルが空の場合 {

        UserDao sut;

        @Before
        public void setUp() throws Exception {
            DbUtils.drop("users");
            sut = new UserDao();
        }

        @Test
        public void getListで0件取得できる() throws Exception {
            List<User> actual = sut.getList();
            assertThat(actual, is(notNullValue()));
            assertThat(actual.size(), is(0));
        }
    }

    public static class テーブルにサンプルデータが100件含まれる場合 {

        UserDao sut;

        @Before
        public void setUp() throws Exception {
            DbUtils.drop("users");
            DbUtils.insert("users", getClass().getResource("users.yaml"));
            sut = new UserDao();
        }

        @Test
        public void getListで100件取得できる() throws Exception {
            List<User> actual = sut.getList();
            assertThat(actual, is(notNullValue()));
            assertThat(actual.size(), is(100));
        }
    }
}

```

共通の状態に着目する

テスト対象クラスが状態を持つ場合、事前処理として行われるテスト対象オブジェクトへの操作が重要です。テスト対象オブジェクトは、事前処理の操作の結果、特定の状態となります。このようにテスト対象オブジェクトの状態に着目し、テストケースをグループ化するパターンも有効です。

たとえば、ArrayListなどのコレクションクラスのユニットテストでは、コレクションが空の状態、コレクションに要素が1件ある状態、2件ある状態などで、それぞれメソッドの振る舞いを検証します。コレクションを初期化したあと、addメソッドを1回呼び出せば、リストに要素が1件ある状態となります。しかしながら、このような操作がテストメソッド内で定義されていると、テスト対象の操作があいまいになる危険性があります(リスト6.5)。

リスト6.5 構造化されていないArrayListのテストクラス

```
public class ArrayListFlatTest {

    private ArrayList<String> sut;

    @Before
    public void setUp() throws Exception {
        sut = new ArrayList<String>();
    }

    @Test
    public void listに1件追加してある場合_sizeは1を返す()
        throws Exception {
        sut.add("A");
        int actual = sut.size();
        assertThat(actual, is(1));
    }

    @Test
    public void listに2件追加してある場合_sizeは2を返す()
        throws Exception {
        sut.add("A");
        sut.add("B");
        int actual = sut.size();
        assertThat(actual, is(2));
    }
}
```

一方、リスト6.6はテスト対象オブジェクトの状態に着目し、それぞれの状態でテストクラスを定義したテストコードです。このように、テスト

リスト6.6 構造化されたArrayListのテストクラス

```
@RunWith(Enclosed.class)
public class ArrayListEnclosedTest {

    public static class list1に1件追加してある場合 {
        private ArrayList<String> sut;

        @Before
        public void setUp() throws Exception {
            sut = new ArrayList<String>();
            sut.add("A");
        }

        @Test
        public void sizeは1を返す() throws Exception {
            int actual = sut.size();
            assertThat(actual, is(1));
        }
    }

    public static class list1に2件追加してある場合 {
        private ArrayList<String> sut;

        @Before
        public void setUp() throws Exception {
            sut = new ArrayList<String>();
            sut.add("A");
            sut.add("B");
        }

        @Test
        public void sizeは2を返す() throws Exception {
            int actual = sut.size();
            assertThat(actual, is(2));
        }
    }
}
```

の事前条件を満たすためのコードをsetUpメソッドに移動すれば、テストメソッドが読みやすくなります。

なお、この例では構造化しなかった場合(リスト6.5)よりも、した場合(リスト6.6)のほうがテストコードが長くなってしましましたが、テストケースが増えてくれれば、構造化されたテストクラスのほうが短く読みやすいコードとなるでしょう。

コンストラクタのテストを分ける

インスタンス化テスト、すなわちオブジェクトの生成に関するテストは、ほかのテストとは性質が異なります。なぜならば、テスト対象オブジェクトの生成自体がテスト内容だからです。

インスタンス化テストはほかのテストケースとは分け、「初期状態を検証する特別なコンテキスト」と考えるとよいでしょう。リスト6.7ではUserクラスのインスタンスをデフォルトのコンストラクタで生成した時に、初期値が期待する値となっているかをテストしています。

リスト6.7 構造化されたインスタンス化テスト

```
@RunWith(Erowned.class)
public class UserTest {
    public static class インスタンス化テスト {
        @Test
        public void デフォルトコンストラクタ() throws Exception {
            User instance = new User();
            assertThat(instance.getName(), is("nobody"));
            assertThat(instance.isAdmin(), is(false));
        }
    }
}
```

6.4 テストクラスを横断する共通処理

Enclosed テストランナーによるテストクラスの構造化を行うことで、テストクラス内の共通処理はきれいに整理できます。しかしながら、異なるテストクラス間で共通した処理は整理できません。このようなときは、共通処理を定義した基底クラスを作成してテストクラスに継承させることが多いでしょう。

継承はなるべく避けるべきです。なぜならば、安易な継承はメンテナンス性を下げるだけでなく、無意識に巨大な機能を持った基底クラスを生み出しやすいからです。なるべくならば、共通処理をユーティリティクラスなどに抽出し、テストクラスからはユーティリティクラスを利用するほうがよいでしょう。

リスト 6.8 は、テスト用のインメモリデータベースの起動と停止など、やや複雑な共通処理を持つテストクラスです。このコードでは、テストの本質とは関係のない多くの処理が混入しています。これでは、可読性を損なうだけではなく、InMemoryDB クラスの使い方などが変更された場合に、それを利用しているテストクラス全体に影響を与えてしまいます。

このように、テストクラスからテストとは直接関連しない共通処理を抽出する場合は、JUnit で共通処理を抽出する機能であるルールを使うと便利です。ルールを利用すると、リスト 6.9 のように記述できます。テストクラスでは、テスト対象クラスに関するコードを中心に記述し、それ以外のコードは極力減らすことができます。

ルールについては第 9 章で詳細に解説します。

4
5
6
7
8
9
10**リスト6.8** 複雑な共通処理が記述されたテストクラス

```
public class UserDaoTest {
    private UserDao sut;
    private InMemoryDB db;

    @Before
    public void setUp() throws Exception {
        db = new InMemoryDB();
        db.start();
        sut = new UserDao();
    }

    @After
    public void tearDown() throws Exception {
        db.shutdownNow();
    }

    @Test
    public void getListは0件を返す() throws Exception {
        // 省略
    }
}
```

リスト6.9 ルールを使った共通処理の抽出

```
public class UserDaoTest {
    private UserDao sut;

    @Rule
    public InMemoryDBRule db = new InMemoryDBRule();

    @Before
    public void setUp() throws Exception {
        sut = new UserDao();
    }

    @Test
    public void getListは0件を返す() throws Exception {
        // 省略
    }
}
```

第7章

テストフィクスチャ テストデータや前提条件のセットアップ^{注1}

ユニットテストは、「事前準備」「実行」「検証」「後処理」の4つのフェーズで行われます(4フェーズテスト)。特に「事前準備」は、テストに必要なデータ、環境、オブジェクトの状態などを準備する重要なフェーズです。また、「後処理」では、次のテストに影響がないように、そのテストで作成したデータ、環境、オブジェクトの状態などを適切に解放する必要があります。

この章では、これらの事前準備や後処理で扱うデータやテスト実行環境を扱うパターンについて解説します。

7.1 テストフィクスチャとは?

ソフトウェアテストは、テスト対象となるシステムやオブジェクトだけでは成立しません。入力するデータ、テスト実行のための予備操作、データベースなどの外部リソース、検証に必要なデータなどが必要不可欠です。

これらのテストで扱うデータやテスト実行環境、オブジェクトの状態などはテストフィクスチャ(*test fixtures*)、もしくは単純にフィクスチャ(*fixture*)と呼ばれます^{注1}。

ユニットテストのフィクスチャ

ユニットテストのフィクスチャには次のような要素が含まれます。

- テスト対象オブジェクト
- テストの実行に必要なオブジェクト(入力値)
- テストの検証に必要なオブジェクト(期待値)
- テストの実行までに必要なテストオブジェクトの操作

^{注1} 狹義には、テストデータのみを指して「テストフィクスチャ」と呼ぶ場合もあります。

- ・ファイルなどの外部リソース
- ・データベースやソケットサーバなどの外部システム
- ・依存クラスや依存外部システムのモックオブジェクト

ユニットテストの事前準備フェーズでは、これらのフィクスチャを適切にセットアップする必要があります。もし、意図するフィクスチャがセットアップできなければ、不完全なテストケースです。

ユニットテストのテストケースでは、原則として、1回だけテスト対象メソッドの操作を行うため、実行フェーズのコードは数行です。一方、フィクスチャのセットアップのコードは長く、複雑になります。このため、テストコードの可読性を高めるには、フィクスチャのセットアップをどのように工夫するかが重要です。特に、多くの入力値を持ち外部リソースにも依存するテスト対象クラスでは、注意してフィクスチャのセットアップを行う必要があります。

フレッシュフィクスチャ

ユニットテストでは、「フィクスチャはテストケースごとに独立し、テストの実行ごとに初期化され、終了時に解放する」が基本的な戦略です。この戦略は、フレッシュフィクスチャ(*fresh fixture*)と呼ばれます。

フレッシュフィクスチャは、「テストケースは互いに独立しており、依存し合ってはならない」というユニットテストの原則と相性の良い戦略です。もし、フィクスチャがいくつかのテストケースで共有されていたならば、テストケースの実行順序などによってフィクスチャの状態が影響を受けるかもしれません。すると、テストの実行順序によってテスト結果が変わる可能性が生まれます。また、ユニットテストを並列に実行した場合、共有されたフィクスチャが同時に複数のテストから利用されるため、問題はより深刻になります。

JUnitでは、テストの実行時にテストケースごとにテストクラスのインスタンスを生成し、テストメソッドを実行します。このため、テストクラスのインスタンス変数やローカル変数に保持されたテストデータは、各テストの終了時に、ガベージコレクタによって破棄されます。つまり、各テス

トケースでフィクスチャが共有されることはありません。

一方、シングルトンとして設計されたクラスやデータベースなどの外部リソースでは、データを共有せざるを得ません。テストを並列に実行しない、初期化と後始末を丁寧に行うなど、特別な注意が必要です。

フィクスチャとスローテスト問題

フレッシュフィクスチャは基本的なフィクスチャのセットアップパターンです。しかしながら、データベースを扱うテストなどでは、フィクスチャのセットアップに長い時間がかかることが問題となります。このような問題は、**スローテスト問題(slow tests)**と呼ばれます。

原因は、フィクスチャのセットアップごとに対象となるテーブルの全レコードを削除し、必要なレコードを追加するからです。このため、テストケースが増えてくると、全テストの実行に半日以上かかる場合もあります。

スローテスト問題を解決するには、テストの並列実行や共有フィクスチャ、カテゴリ化テストなどが有効です。

テストの並列実行では、マルチスレッドや複数のテスト環境で並列にテストを実行し、効率化を図ります。

後述する共有フィクスチャでは、同一のフィクスチャを共有して使うことでセットアップコストを抑えます。

カテゴリ化テストでは、テストをいくつかのグループに分け、実行するテストをフィルタリングします。カテゴリ化テストについては第10章で解説します。

なお、テストケースが少ない場合や、データベースや外部サービスへの通信を含むテストケースが少ない場合には、スローテスト問題が発生することはほとんどありません。また、スローテスト問題はユニットテストの実行時間が何十分もかかるようになってから対応るべきです。原則として、ユニットテストはお互いに独立し、すべてのテストを実行してください。

共有フィクスチャ

共有フィクスチャ(*shared fixture*)はスローテスト問題を解決する手段のひ

ひとつです。共有フィクスチャでは、各テストケースで使用するフィクスチャを共有し、再利用することで、セットアップコストを抑えます。

しかしながら、共有フィクスチャには多くの問題があります。最大の問題は、テストケースごとの独立性が弱くなることです。テストケースの実行順序やほかのテストケースの実行結果によって、共有化されたフィクスチャが影響を受け、別のテストケースに影響を与えるかもしれません。

また、テストコードのメンテナンス性も低下します。あるフィクスチャをどのテストコードが参照しているのかがわからなくなり、グローバル変数と同様の問題を引き起こします。さらに、共有フィクスチャでは適切に後処理を行う必要があります。

したがって、可能な限りフィクスチャは共有しないほうがよいでしょう。

なお、共有フィクスチャを不变オブジェクトとすれば、多くの問題は解決します。状態が変わらない不变オブジェクトであれば、テストの実行順序や結果、さらには並列実行の場合でも問題となりません。

7.2 フィクスチャのセットアップパターン

フィクスチャのセットアップはテストコードを記述する中で最も難しい部分です。第1章のチュートリアルで扱ったCalculatorクラスのテストであれば、必要なデータも少なく、テストメソッドの中で記述しても問題ありません(リスト7.1)。

しかしながら、リスト7.2のように複雑なテストデータを扱う場合、フィクスチャのセットアップがテストコードを占める割合が大きくなります。現実のアプリケーションではもっと複雑なテストデータを扱いますので、同じようにセットアップを行っていては、テストコードはますます読めなくなるでしょう。

そこで、この節ではフィクスチャのセットアップのパターンを紹介します。テストケースやフィクスチャに適するパターンを選択し、テストコードの可読性を高めましょう。

インラインセットアップ

インラインセットアップ(*inline setup*)は、最も基本的なフィクスチャのセットアップパターンです。インラインセットアップでは、テストメソッドごとにフィクスチャのセットアップを行います(リスト7.3)。

インラインセットアップの特徴は、テストメソッド内でテストコードが完結し、テストコードの見通しが良くなることです。しかしながら、リスト7.2のようにフィクスチャのセットアップが複雑で長い場合、テストの実行や検証が相対的に短くなるのが欠点です。

一般的に、長すぎるメソッドはコードの可読性を悪くします。絶対的な基準はありませんが、筆者の主觀では20行を超えるメソッドは長いと感じます。非常に長いフィクスチャのセットアップが記述されたテストコード

リスト7.1 単純なフィクスチャのセットアップ

```
@Test
public void multiplyで3と4の乗算結果が取得できること() throws Exception {
    Calculator calc = new Calculator();
    int expected = 12;
    int actual = calc.multiply(3, 4);
    assertThat(actual, is(expected));
}
```

リスト7.2 複雑なフィクスチャのセットアップ

```
@Test
public void getTotalPrice() throws Exception {
    // SetUp
    Book book = new Book();
    book.setTitle("Refactoring");
    book.setPrice(4500);
    Author author = new Author();
    author.setFirstName("Martin");
    author.setLastName("Fowler");
    book.setAuthor(author);
    BookStore sut = new BookStore();
    sut.addToCart(book, 1);
    // Exercise & Verify
    assertThat(sut.getTotalPrice(), is(4500));
}
```

は可読性が悪く、どこで操作を行い、どこで検証しているのかが理解できません。フィクスチャのセットアップが複雑であり、そのコードが長くなるようであれば、ほかのパターンの利用を検討してください。

とはいっても、オンラインセットアップはシンプルで理解しやすいフィクスチャのセットアップパターンです。1つ目のテストメソッドを書く場合や、単純なフィクスチャの場合は、積極的に利用すべきです。

暗黙的セットアップ

JUnitでは、テストクラスで共通した初期化処理を、Beforeアノテーションが付与されたメソッドに抽出できます。このように抽出されたメソッド

リスト7.3 フィクスチャのオンラインセットアップ

```
public class StringUtilTest {

    @Test
    public void isEmptyOrNullは空文字列でtrueを返す() throws Exception {
        // SetUp
        String input = "";
        boolean expected = true;
        // Exercise
        boolean actual = StringUtil.isEmptyOrNull(input);
        // Verify
        assertThat(actual, is(expected));
    }

    @Test
    public void isEmptyOrNullはAAAでfalseを返す() throws Exception {
        // SetUp
        String input = "AAA";
        boolean expected = false;
        // Exercise
        boolean actual = StringUtil.isEmptyOrNull(input);
        // Verify
        assertThat(actual, is(expected));
    }
}
```

はセットアップメソッドと呼ばれ、各テストメソッドを実行する前に暗黙的に実行されます。このようなセットアップメソッドでフィックスチャのセットアップを行うパターンを暗黙的セットアップ(*implicit setup*)と呼びます。

暗黙的セットアップは、フィックスチャのセットアップコードをセットアップメソッドに抽出するパターンです。暗黙的セットアップを使うことにより、テストメソッドのコードはテストの実行と検証が中心となり、何をテストしようとしているかが明確になります。

暗黙的セットアップは、Enclosed テストランナーを利用したユニットテストで高い効果を期待できます。なぜならば、Enclosed テストランナーを利用する場合、共通の初期化処理を持つテストケースでグループ化するからです。各ネストしたクラスで、フィックスチャのセットアップのコードがセットアップメソッドに抽出され、見通しの良いテストコードとなります(リスト7.4)。

ただし、テストケースごとに入力値や期待値が異なる場合もあります。そのような場合は、テストの前提条件を満たすために必要なテスト対象オブジェクトの初期化や事前操作などに関してはセットアップメソッドで行いますが、それ以外のテストケースごとに固有のフィックスチャに関してはテストメソッド内でセットアップします。

リスト7.4 ネストしたクラスごとに行われる暗黙的セットアップ

```
@RunWith(Enclosed.class)
public class LruCacheTest {

    public static class AとBを追加している場合 {
        LruCache<String> sut;

        @Before
        public void setUp() throws Exception {
            sut = new LruCache<String>();
            sut.put("A", "valueA");
            sut.put("B", "valueB");
        }

        @Test
        public void sizeは2() throws Exception {
            assertThat(sut.size(), is(2));
        }
    }
}
```

```

    }

    @Test
    public void get_AでvalueAを返しkeysはBA() throws Exception {
        assertThat(sut.get("A"), is("valueA"));
        assertThat(sut.keys.size(), is(2));
        assertThat(sut.keys.get(0), is("B"));
        assertThat(sut.keys.get(1), is("A"));
    }

    @Test
    public void get_BでvalueBを返しkeysはAB() throws Exception {
        assertThat(sut.get("B"), is("valueB"));
        assertThat(sut.keys.size(), is(2));
        assertThat(sut.keys.get(0), is("A"));
        assertThat(sut.keys.get(1), is("B"));
    }

    @Test
    public void get_Cでnullを返しkeysはAB() throws Exception {
        assertThat(sut.get("C"), is(nullValue()));
        assertThat(sut.keys.size(), is(2));
        assertThat(sut.keys.get(0), is("A"));
        assertThat(sut.keys.get(1), is("B"));
    }

    @Test
    public void put_Cでsizeは3_keysはABCとなる() throws Exception {
        // Set up
        String key = "C";
        String item = "valueC";
        // Exercise
        sut.put(key, item);
        // Verify
        assertThat(sut.size(), is(3));
        assertThat(sut.keys.size(), is(3));
        assertThat(sut.keys.get(0), is("A"));
        assertThat(sut.keys.get(1), is("B"));
        assertThat(sut.keys.get(2), is("C"));
    }
}
}

```

4
5
6
7
8
9
10

生成メソッドでのセットアップ

先ほど解説した暗黙的セットアップは、セットアップメソッドに共通した初期化処理を抽出するパターンですが、テストクラスに共通した処理にしか適用できないという制約があります。つまり、複数のテストクラスでセットアップメソッドで共通の初期化処理を行いたい場合に適用できません。このような場合、共通した初期化処理を独立したクラスのメソッドに抽出する手法が有効です。このように、フィックスチャの生成メソッドを抽出し、フィックスチャのセットアップを行うパターンを、**生成メソッド**(*creation method*)と呼びます。

生成メソッドでのセットアップでは、共通したコードをシンプルにメソッドとして抽出します(リスト7.5、リスト7.6)。このとき、テストクラスのprivateメソッドとして抽出することができますが、独立したクラスにstaticメソッドとして定義するとよいでしょう。なぜならば、staticインポートを使うことができるため、テストコードの可読性を高めることができます。また、メソッド名を日本語で定義し、どんなフィックスチャをセットアップしているのかが一目で理解できるようにするとよいでしょう。

一般的に、長いメソッドは可読性が悪いため、いくつかのメソッドに分割することで整理します。プロダクションコードでは、さらに適切なクラスに分割したり、デザインパターンを適用したりすることで、コードを整理します。しかし、テストコードでは拡張性はあまり考慮せず、テストコードとして読みやすいかどうかを重視します。

外部リソースからのセットアップ

フィックスチャのセットアップを生成メソッドに抽出すればテストコードの可読性は高くなります。しかしながら、複雑なオブジェクトの生成を行うコードが消えるわけではありません。Javaではフィックスチャのセットアップのようなデータを生成するコードは読みやすくありません。これは、Javaの言語仕様では宣言的なコードを記述しにくいためです。

setterメソッドやputメソッドなどの手続き的なメソッド呼び出しではなく、そのデータがどんな値を持っているかを宣言的に記述できればコード

4
5
6
7
8
9
10**リスト7.5** 生成メソッドによるフィクスチャのセットアップ

```
public class BookStoreTest {
    @Test
    public void getTotalPrice() throws Exception {
        // SetUp
        BookStore sut = new BookStore();
        Book book = Bookオブジェクトの作成_MartinFowlerのRefactoring();
        sut.addToCart(book, 1);
        // Exercise & Verify
        assertThat(sut.getTotalPrice(), is(4500));
    }

    @Test
    public void get_0() throws Exception {
        // SetUp
        BookStore sut = new BookStore();
        Book book = Bookオブジェクトの作成_MartinFowlerのRefactoring();
        sut.addToCart(book, 1);
        // Exercise & Verify
        assertThat(sut.get(0), is(sameInstance(book)));
    }
}
```

リスト7.6 Bookオブジェクトの生成メソッド

```
public class BookStoreTestHelper {
    public static Book Bookオブジェクトの作成_MartinFowlerのRefactoring() {
        Book book = new Book();
        book.setTitle("Refactoring");
        book.setPrice(4500);
        Author author = new Author();
        author.setFirstName("Martin");
        author.setLastName("Fowler");
        book.setAuthor(author);
        return book;
    }
}
```

は読みやすくなります。

このため、外部に定義したリソースファイルにテストデータを記述し、生成メソッドなどで読み込む手法が有効です。リソースファイルには、YAML、XML、JSON、CSV、Excelなどのフォーマットが利用できます。絶対的に使いやすいフォーマットはないため、フィックスチャの特徴によって使い分けるとよいでしょう。

なお、XMLは複雑なテストデータも表現することができますが、記述量が多く専用のエディタを使わなければメンテナンスしにくいため、あまりお勧めできません。利用する場合は外部ツールなどを使いフィックスチャを管理し、リソースファイルは自動生成するとよいでしょう。

JSONは複数行テキストの扱いに難がありますが、Webサービスなどでは出力形式としてよく使われます。

フィックスチャがリスト構造で大量に必要ならばCSVやExcelが便利です。筆者のお勧めは、シンプルでXMLと同程度の表現力があるYAMLです。

● YAMLを使ったセットアップ

リスト7.7は、リスト7.6と同等のデータをYAMLを使って定義したリソースファイルです^{注2}。YAMLのフォーマットを知らなくとも、なんとなくBookオブジェクトがどのようなデータ構造になっているかが理解できるでしょう。このようにYAMLでは、オブジェクトを構造的に宣言できるため、複雑なオブジェクトや多くのプロパティを持つオブジェクトを可読性の高い状態で記述できます。

YAMLファイルは、外部ライブラリを利用して読み込めば簡単にオブジェクトに変換できます。リスト7.8では、YAMLを読み書きするライブラリSnakeYaml^{注3}を使ってテストデータを読み込んでいます。

なお、外部リソースからのセットアップでは、テストケースとテストデータがそれぞれ独立したファイルに定義されるため、相互参照を行いにくいことが難点です。

注2 後述するリスト7.8と同じパッケージに配置してください。

注3 <http://code.google.com/p/snakeyaml/>

4
5
6
7
8
9
10**リスト7.7** YAMLによるリソースファイル

```
!!ch07.Book
title: Refactoring
price: 4500
author: !!ch07.Author
  firstName: Martin
  lastName: Fowler
```

リスト7.8 SnakeYamlによるYAMLファイルの読み込み

```
public class BookStoreYamlTest {

    @Test
    public void getTotalPrice() throws Exception {
        // SetUp
        BookStore sut = new BookStore();
        Book book = (Book) new Yaml()
            .load(getClass().getResourceAsStream("book_fixtures.yaml"));
        sut.addToCart(book, 1);
        // Exercise & Verify
        assertThat(sut.getTotalPrice(), is(4500));
    }

    @Test
    public void get_0() throws Exception {
        // SetUp
        BookStore sut = new BookStore();
        Book book = (Book) new Yaml()
            .load(getClass().getResourceAsStream("book_fixtures.yaml"));
        sut.addToCart(book, 1);
        // Exercise & Verify
        assertThat(sut.get(0), is(sameInstance(book)));
    }
}
```

Column

Javaと宣言的記法

YAMLのように宣言的な記述でオブジェクトの構造を表現できればフィックスチャのセットアップコードは読みやすいものとなります。Javaは手続き的な記述の文法であるため、正攻法では可読性の高い宣言的なコードを書くことができません。しかし、トリッキーな方法ながら、宣言的なコードを書くこともできます(リスト7.A)。

この記述を行うためにはいくつか制約があります。まず、BookクラスやAuthorクラスは匿名インナークラスによるサブクラスとして定義されているため、finalクラスでは利用できません。また、厳密にはBookクラスではなくBookクラスのサブクラスとなっているため、クラスローダ関連の問題で悩まされるかもしれません。

各フィールドの設定は、匿名インナークラスにイニシャライザを利用して宣言しています。このとき、BookクラスやAuthorクラスのフィールドにアクセスする必要があるため、privateフィールドやfinalフィールドは利用できません。サブクラスからのアクセスとなるため、デフォルト(パッケージプライベート)以上の可視性が必要となります。

この記法はトリッキーながら、Javaの言語仕様のみで記述していることが最大のメリットです。すなわち、コード補完、コンパイラによる検証、Eclipseによるナビゲーションなどのすべての恩恵を受けられます。

もし、外部リソースを使わずに宣言的なフィックスチャのセットアップを行いた

リスト7.A Javaによる宣言的なセットアップ

```
public static Book Bookオブジェクトの作成_MartinFowlerのRefactoring() {
    return new Book() {
        {
            title = "Refactoring";
            price = 4500;
            author = new Author() {
                {
                    firstName = "Martin";
                    lastName = "Fowler";
                }
            };
        }
    };
}
```

いならば、Groovy^{注a}の導入をお勧めします。Groovyを使えば、Javaのクラスに對して宣言的なコードを書くことができるようになります(リスト7.B)。

GroovyをJavaのテスト用のライブラリとして利用しても、プロダクションコードにまったく影響を与えません。ユニットテストのフィクスチャセットアップに限定して利用するならば、学習コストもほとんどかかりません。可読性の高いテストコードを書くためにGroovyの導入を検討してみてください。

注a <http://groovy.codehaus.org/>

解説書としては、関谷和愛、上原潤二、須江信洋、中野靖治著『プログラミングGroovy』技術評論社(2011年)などがあります。

リスト7.B Groovyによる宣言的なセットアップ

```
class BookStoreGroovyTestHelper {
    static Book Bookオブジェクトの作成_MartinFowlerのRefactoring() {
        new Book(
            title: "Refactoring",
            price: 4500,
            author: new Author(
                firstName: "Martin",
                lastName: "Fowler",
            ),
        )
    }
}
```

第8章

パラメータ化テスト テストケースとテストデータの分離

十分なユニットテストを行うには、1種類の入力値についてテストするだけでは不十分です。テスト対象メソッドによっては、多くの入力値について似たようなテストケースを作成しなければなりません。すると、テストケースが非常に多くなるにもかかわらず、テストケースごとの差異は入力値と期待値のみとなります。このため、入力値と期待値をテストケースとは独立して定義し、それらをテストメソッドのパラメータとして利用できるのであれば、ユニットテストの見通しが良くなり、入力値と期待値を追加することも容易となります。

このようなユニットテストの手法はパラメータ化テストと呼ばれます。JUnitではTheoriesテストランナーを使用することで実現できます。

8.1 テストデータの選択

テストデータをどのように選択するかは、ソフトウェアテストにおいて重要な要素のひとつです。なぜならば、ソフトウェアテストにおいて入力可能な値をすべて検証することは現実として不可能であり、入力可能な値から効率良く選択しなければならないためです。

ユニットテストでは、主にメソッドの振る舞いをテストします。したがって、入力値であるメソッドの引数の組み合わせをどう選択するかが重要です。メソッドがboolean型の引数を1つ持つ場合、とり得る入力値はtrueとfalseのどちらかしかありません。しかし、int型の引数であれば実質的に無限の整数値がとり得る入力値となりますし、引数が2つ以上あればそれらの組み合わせがとり得る入力値となります。

しかしながら、10個のテストデータに対するテストよりも、適切な方法で選択した2個のテストデータに対するテストのほうがよいことは珍しいことではありません。テストデータの選択方法はテストの品質に大きく影

響を与えるのです。また、開発リソースも有限であるため、可能な限り少ないテストデータで効率良くテストすることが求められます。

どのくらいのテストデータが必要か？

本節では「2.2 テスト技法」(p.27)で紹介したテスト技法を使い、ユニットテストにおけるテストデータの選択方法を解説します。

● 同値クラスによるテストデータの選択

同値クラスによるテストは、最も基本的で効果的なテスト技法です。同値クラスとは、テスト対象メソッドで同じ結果を返す入力値の集合です。

たとえば、年齢が18歳以上でなければ登録できない会員制サイトがあつたとします。このサイトを構成するモジュールのひとつに年齢の制限を確認するメソッドがあり、このメソッドのユニットテストを行うとします。このメソッドは、int型引数(年齢)をとり、18歳以上ならばtrueを、18歳未満ならばfalseを返す仕様とします(リスト8.1)。

ここでテストデータとしてすべての年齢を使用することは現実的ではありません。いくつかの値を選択してテストデータとして使用します。それは「10と20」かもしれません。あるいは「15と22」かもしれません。ポイントは、「18以上の値から1つ、18未満の値から1つ」を選択することです。

この例では、メソッドの結果がtrueまたはfalseの2パターンであるため、2つの同値クラスが存在します(表8.1)。

リスト8.1 登録ができるかを確認するメソッド

```
/*
 * 年齢を指定して、会員制サイトへ登録できるかを返す
 * @param age 年齢
 * @return 18歳以上ならばtrueを、18歳未満ならばfalse
 */
public static boolean canRegister(int age) {
    return 18 <= age;
}
```

●組み合わせによるテストデータの選択

次に組み合わせが発生するケースを考えます。同じように会員制サイトを構成するモジュールのひとつに特別割引を受けられる優待会員であるかを確認するメソッドがあるとします(リスト8.2)。優待会員の条件は「20歳以上であり、メールマガジンに登録してあり、かつ前月の利用回数が1回以上ある」ことです。

このメソッドは出力パターンだけに着目した同値クラスによるテストデータの選択では不十分です。なぜなら、同値クラス分析を行ったならば、優待会員であるか否かの2つのグループに分類され、それぞれから1つの組み合わせを選択するからです。

このように入力値が複数の値の組み合わせになる場合、組み合わせを考慮し、テストデータを選択します。ただし、闇雲に選択していくと膨大なテストケースが必要となるため、効率良くテストデータを絞り込みます。

それぞれの条件を真偽値で組み合わせた場合、表8.2のように8パターン

表8.1 同値クラス分析

結果	入力値の例	備考
true	20	18以上の値
false	10	18未満の値

リスト8.2 優待会員であるかを判定するメソッド

```
/**
 * 優待会員かどうかを返す
 * @param age 年齢
 * @param isRegisterMailMagazine メールマガジンに登録している場合にtrue
 * @param usePastMonth 前月の利用回数
 * @return 20歳以上であり、メールマガジンに登録してあり、
 *         かつ前月の利用回数が1回以上ならtrue
 */
public static boolean isSpecialMember(
    int age, boolean isRegisterMailMagazine, int usePastMonth) {
    if (age < 20) return false;
    if (!isRegisterMailMagazine) return false;
    if (usePastMonth < 1) return false;
    return true;
}
```

4
5
6
7
8
9
10

の組み合わせがあります。

期待値(結果)がfalseとなるパターンは7つありますが、それぞれ条件が異なります。ここで、「プログラムに混入する不具合のほとんどは単項目に関連する不具合である」という性質を踏まえて、テストの品質を落とさずテストデータを減らします。つまり、「優待会員の条件をすべて満たしている、あるいは1つ満たしていない」という基準で、テストデータを選択します(表8.3)。

このように単項目の条件に着目してテストデータを選択する方法は自然に行われています。それは、プログラマは内部実装に詳しいため、リスト8.2のように、各項目に対する条件分岐を実装するからです。なお、このようなデータの選択は、テストによるプロダクションコードの実行網羅率(カバレッジ)を測定する場合にも考慮されます^{注1}。

最後に、各テストデータのグループで適当な値を選択し、テストデータを構成すると表8.4のようになります。

注1 ⇒コードカバレッジ(p.256)

表8.2 入力値の組み合わせと期待値の関係

年齢	メルマガ登録	前月の利用回数	期待値
20歳以上	YES	1回以上	true
20歳以上	NO	1回以上	false
20歳以上	YES	0回	false
20歳以上	NO	0回	false
20歳未満	YES	1回以上	false
20歳未満	NO	1回以上	false
20歳未満	YES	0回	false
20歳未満	NO	0回	false

表8.3 単項目に着目した入力値の組み合わせと期待値の関係

年齢	メルマガ登録	前月の利用回数	期待値
20歳以上	YES	1回以上	true
20歳未満	YES	1回以上	false
20歳以上	NO	1回以上	false
20歳以上	YES	0回	false

どのテストデータを選択するか？

テストでは、すべての入力値に対してテストを行うことができません。このため、同値クラス分析を行い、出力値に着目して入力値を絞り込みます。同値クラス分析ではテストデータが不十分であれば、入力値の組み合わせを考慮し、テストデータを増やします。このように、テストデータを作成する場合には、どのような性質を持ったテストデータのグループを作るかを考えます。

そして、各グループからどの値を選択するかも重要です。効果的にテストを行うには、境界値の考え方を利用して値の選択を行うとよいでしょう。これは、プログラムの不具合は、境界値附近で発生しやすいという経験的／統計的事実を利用しています。

特に、「以下」や「未満」といった境界を含むか含まないかのコードはケアレスミスを誘発しやすい条件です。したがって、なるべくテストデータとして境界値を選択することで、テストの精度は高まります。**表8.5**は、表8.4のテストデータを境界値を使って再定義したテストデータです。

このように、境界値である20の近傍に着目し、年齢は19と20を選択することにより効果的なテストを行うことができます。

表8.4 具体的なテストデータ

年齢	メルマガ登録	前月の利用回数	期待値
20	true	1	true
19	true	3	false
25	false	2	false
31	true	0	false

表8.5 境界値を適用したテストデータ

年齢	メルマガ登録	前月の利用回数	期待値
20	true	1	true
19	true	1	false
20	false	1	false
20	true	0	false

Column

妥当な値の範囲を制限する

会員制サイトの例では、年齢をint型の引数とし、同値クラスに対するテストテストの値として「10と20」などを選択しました。ここで「-1と3000」を選択してもテスト結果は変わりませんが、年齢というパラメータでそのような値は不正な値とも言えます。このような場合、パラメータが妥当である範囲を制限することでテストも効率的になります。

ひとつの実装方法としては、メソッドの仕様に「年齢が1以上150未満でない場合は例外を送出する」と加えることです。そうすれば、入力値の有効範囲が1から149までとなり、すべての値のテストも実施可能な範囲になります。このテストではすべての年齢についてテストすることは意味がないかもしれません、状況によっては有効な手段です。

もちろん、テストデータとして-1や3000を追加し、正しく例外が送出されるかを検証しなければなりません。

この実装とテストが必要かどうかは、そのソフトウェアがどの程度の品質を必要とするかに依存します。会員制サイトで年齢を間違えて3000と入力して登録したとしても、ビジネス上に大きな問題はないかと思います。しかし、証券取引システムで売買数をマイナスで入力できたり、億単位の発注数の入力を受け付けてしまったならば、取り返しのつかない問題となるでしょう。

また、ほかの設計方針として、年齢をオブジェクトとして扱い、制約を加える方法があります(リスト8.A)。このAgeクラスを使ってリストを書き換えるリスト8.Bのようになります。

このように値をドメインに特化したオブジェクトとすることでソフトウェアはより堅牢になります。しかしながら、シンプルさとのトレードオフとなります。

リスト8.A 値の制約がある年齢クラス

```
public class Age {
    public final int value;

    public Age(int value) {
        if (value < 0 || 150 <= value)
            throw new IllegalArgumentException();
        this.value = value;
    }
}
```

リスト8.8 Ageオブジェクトを利用したメソッド

```
/*
 * 年齢を指定して、会員制サイトへ登録できるかを返す
 * @param age 年齢
 * @return 18歳以上ならばtrueを、18歳未満ならばfalse
 */
public static boolean canRegister(Age age) {
    return 18 <= age.value;
}
```

8.2 入力値と期待値のパラメータ化

テスト対象メソッドが複数のパラメータを引数として持ち、その組み合わせによって異なる結果を返す場合、検証には多くのテストデータが必要です。しかしながら、テストデータごとにテストメソッドを定義すると、テストメソッドの数が膨大になります。また、各テストケース間の違いはテストの入力値と期待値であり、テストコードは冗長となるでしょう。

リスト8.3 はじんけん判定メソッドのテストコードの一部です。じんけんには、グー、チョキ、パーの3つの手(リスト8.4)があり、それぞれの手の組み合わせが全テストケースです。もし、全パターンをテストするならばテストケースは9パターンとなります。

このように、テストデータの組み合わせによりテストコードが冗長となる問題を解決するには、テストデータとテストメソッドを分割するパラメータ化テスト(*parameterized test*)が有効です。JUnitでは、バージョン4.4で導入されたTheoriesテストランナーを使うことで、引数を持つテストメソッドを定義し、テストデータをフィールドなどで宣言できます。

Theories——パラメータ化テストのテストランナー

org.junit.experimental.theories.Theoriesクラスはテストランナーのひとつで、パラメータ化されたユニットテストをサポートします。パラメータ化テストを行うには、リスト8.5のように、テストクラスにRunWithアノテ

4
5
6
7
8
9
10**リスト8.3** じゃんけんのテストコード

```
@Test
public void グーとチョキなら勝利() throws Exception {
    assertThat(sut.judge(GU, TYOKI), is(WIN));
}

@Test
public void グーとパーなら敗北() throws Exception {
    assertThat(sut.judge(GU, PA), is(LOSE));
}

// 残りの7パターンは省略
```

リスト8.4 じゃんけんクラス

```
public class Janken {
    public enum Hand {
        GU, TYOKI, PA
    }

    public enum Result {
        WIN, LOSE, DRAW
    }

    public Result judge(Hand h1, Hand h2) {
        if (h1 == h2) return Result.DRAW;
        switch (h1) {
            case GU:
                return h2 == Hand.TYOKI ? Result.WIN : Result.LOSE;
            case PA:
                return h2 == Hand.GU ? Result.WIN : Result.LOSE;
            default:
                return h2 == Hand.PA ? Result.WIN : Result.LOSE;
        }
    }
}
```

リスト8.5 Theories テストランナーの指定

```
@RunWith(Theories.class)
public class ParameterizedTest {
```

ーションでTheoriesテストランナーを指定します。

なお、TheoriesテストランナーはEnclosedテストランナー^{注2}と併用して使うこともできます(リスト8.6)。

@Theory—— テストメソッドに指定するアノテーション

org.junit.experimental.theories.Theoryアノテーションはパラメータ化テストでテストメソッドに付与するアノテーションです。パラメータ化テストでは、Testアノテーションの代わりにTheoryアノテーションを使用します(リスト8.7)。

Theoryアノテーションを付与したテストメソッドには、任意の引数を宣言できます。複数の引数を宣言した場合は、各引数の組み合わせによるテストとなります。

注2 ➔ Enclosed—— 構造化したテストクラスを実行する(p.86)

リスト8.6 TheoriesテストランナーとEnclosedテストランナーの併用

```
@RunWith(Enclosed.class)
public class ParameterizedTest {

    @RunWith(Theories.class)
    public static class XXXの場合 {
        }
}
```

リスト8.7 Theoryアノテーションを付与したテストメソッド

```
@RunWith(Theories.class)
public class ParameterizedTest {
    @Theory
    public void testCase(int x) throws Exception {
    }
}
```

@DataPoint——パラメータを定義するアノテーション

Theories テストランナーを使ったパラメータ化テストでは、org.junit.experimental.theories.DataPoint アノテーションを使いパラメータを定義します。パラメータは staticかつ publicなフィールドまたはメソッドで定義します（リスト 8.8）。

リスト 8.8 @DataPointによるパラメータの指定

```
@RunWith(Theories.class)
public class ParameterizedTest {
    @DataPoint
    public static int INT_PARAM_1 = 3;
    @DataPoint
    public static int INT_PARAM_2 = 4;

    public ParameterizedTest() {
        System.out.println("初期化");
    }

    @Theory
    public void 引数を持つテストメソッド(int param) throws Exception {
        System.out.println("引数を持つテストメソッド(" + param + ")");
    }
}
```

Column

Parameterized テストランナー

JUnit でパラメータ化テストを行うには、Theories テストランナーのほかに Parameterized テストランナーを使う方法もあります。

Theories テストランナーではテストメソッドの引数でパラメータを受け取りますが、Parameterized テストランナーでは、テストクラスのコンストラクタでパラメータを受け取ります。このため、Parameterized テストランナーを使う場合、パラメータをテストクラスのフィールドに保持する必要があります。

歴史的には、Parameterized テストランナーのほうが先に実装された機能です。しかし、ほかの xUnit 系フレームワークではテストメソッドでパラメータを受け取るようなしくみが一般的であるため、今後は Theories テストランナーを使ったほうがよいでしょう。

リスト 8.8 を実行すると、次のように出力されます。

初期化

引数を持つテストメソッド(3)

初期化

引数を持つテストメソッド(4)

パラメータとして定義されている 3 と 4 がテストメソッドに渡されていることと、呼び出されるパラメータごとにテストクラスのオブジェクトが作成されていることが確認できます。

● 複数のテストメソッドがある場合

リスト 8.9 のようにテストケース内に複数のテストメソッドがある場合は、パラメータとして定義されたフィールドの型情報からどのテストメソッドのパラメータとなるかが決定します。リスト 8.9 では、int 型のパラメ

リスト 8.9 複数のテストメソッドが定義されたパラメータ化テスト(わかりにくい)

```
@RunWith(Theories.class)
public class ParameterizedTypeTest {
    @DataPoint
    public static int INT_PARAM_1 = 3;
    @DataPoint
    public static int INT_PARAM_2 = 4;

    @DataPoint
    public static String STRING_PARAM_1 = "Hello";
    @DataPoint
    public static String STRING_PARAM_2 = "World";

    @Theory
    public void 引数がint型のテストメソッド(int param) throws Exception {
        System.out.println("引数がint型のテストメソッド(" + param + ")");
    }

    @Theory
    public void 引数がString型のテストメソッド(String param)
        throws Exception {
        System.out.println("引数がString型のテストメソッド(" + param + ")");
    }
}
```

ータと String 型のパラメータの 2 種類を定義し、それぞれの型のテストメソッドを定義しています。

このテストを実行すると、次のように出力されます。

```
引数がint型のテストメソッド(3)
引数がint型のテストメソッド(4)
引数がString型のテストメソッド(Hello)
引数がString型のテストメソッド(World)
```

しかしながら、この実行結果は直感的に理解できる形ではありません。もし、異なる種類のパラメータを扱うパラメータ化テストを行うならば、リスト 8.10 のように Enclosed テストランナーを併用し、テストクラスを分割して整理するほうがよいでしょう。

リスト 8.10 テストメソッドごとにテストクラスを分割して定義したパラメータ化テスト

```
@RunWith(Enclosed.class)
public class EnclosedParameterizedTypeTest {
    @RunWith(Theories.class)
    public static class intのパラメータ化テスト {
        @DataPoint
        public static int INT_PARAM_1 = 3;
        @DataPoint
        public static int INT_PARAM_2 = 4;

        @Theory
        public void 引数がint型のテストメソッド(int param)
            throws Exception {
            System.out.println(
                "引数がint型のテストメソッド(" + param + ")");
        }
    }

    @RunWith(Theories.class)
    public static class Stringのパラメータ化テスト {
        @DataPoint
        public static String STRING_PARAM_1 = "Hello";
        @DataPoint
        public static String STRING_PARAM_2 = "World";

        @Theory
```

```

public void 引数がString型のテストメソッド(String param)
        throws Exception {
    System.out.println(
        "引数がString型のテストメソッド(" + param + ")");
}
}
}

```

● 複数の引数が定義されている場合

リスト8.11のように、テストメソッドに複数の引数が定義されている場合は、定義されたパラメータのすべての組み合わせがテストとして実行されます。実行結果は、次のように出力されます。

```

テストメソッド(3, Hello)
テストメソッド(3, World)
テストメソッド(4, Hello)
テストメソッド(4, World)

```

● 同じ型の引数が複数定義されている場合

リスト8.12のように、同じ型の引数が複数定義されている場合は、重複も含めてすべての組み合わせで実行されます。

```

テストメソッド(3, 3)
テストメソッド(3, 4)
テストメソッド(4, 3)
テストメソッド(4, 4)

```

このように、テストクラスには複数の引数の異なるテストケースや複数の引数を持つテストケースを作成することができます。ただし、組み合わせテストは複雑になるので注意して利用してください。代わりに、次に紹介するフィクスチャオブジェクトを利用するとよいでしょう。

● パラメータをフィクスチャオブジェクトにまとめる

JUnitで可読性の高いパラメータ化テストを行うポイントは、引数を1つとすることです。リスト8.13のようにパラメータ群を1つのオブジェクトにまとめると可読性の高いコードとなります。

4
5
6
7
8
9
10**リスト8.11** 複数の引数を定義したパラメータ化テスト(型が異なる場合)

```
@RunWith(Theories.class)
public class ParameterizedMultiParamsTest {
    @DataPoint
    public static int INT_PARAM_1 = 3;
    @DataPoint
    public static int INT_PARAM_2 = 4;
    @DataPoint
    public static String STRING_PARAM_1 = "Hello";
    @DataPoint
    public static String STRING_PARAM_2 = "World";

    @Theory
    public void テストメソッド(int paramInt, String strParam)
        throws Exception {
        System.out.println(
            "テストメソッド(" + paramInt + ", " + strParam + ")");
    }
}
```

リスト8.12 複数の引数を定義したパラメータ化テスト(型が同じ場合)

```
@RunWith(Theories.class)
public class ParameterizedMultiSameTypeParamsTest {
    @DataPoint
    public static int INT_PARAM_1 = 3;
    @DataPoint
    public static int INT_PARAM_2 = 4;

    @Theory
    public void テストメソッド(int x, int y) throws Exception {
        System.out.println("テストメソッド(" + x + ", " + y + ")");
    }
}
```

リスト8.13 フィクスチャオブジェクトによるパラメータ化テスト

```
@RunWith(Theories.class)
public class CalculatorDataPointTest {
    @DataPoint
    public static Fixture PARAM_1 = new Fixture(3, 4, 7);
    @DataPoint
    public static Fixture PARAM_2 = new Fixture(0, 5, 5);
    @DataPoint
    public static Fixture PARAM_3 = new Fixture(-3, 1, -2);
```

```

@Theory
public void add(Fixture p) throws Exception {
    Calculator sut = new Calculator();
    assertThat(sut.add(p.x, p.y), is(p.expected));
}

static class Fixture {
    int x;
    int y;
    int expected;

    Fixture(int x, int y, int expected) {
        this.x = x;
        this.y = y;
        this.expected = expected;
    }
}
}

```

@DataPoints —— 複数のパラメータを定義するアノテーション

org.junit.experimental.theories.DataPoints アノテーションは、DataPoint アノテーションと同様にパラメータ化テストのパラメータの定義を行うアノテーションです。DataPoint では1つのフィールド(またはメソッド)に対して1つのパラメータしか定義できませんでしたが、DataPoints を使うことで1つのフィールド(またはメソッド)に複数のパラメータを定義できます。

DataPoints を使ってフィールドやメソッドで複数のパラメータを定義する場合は、リスト 8.14 のように型を配列として定義します。

また、DataPoints も DataPoint と同様に static メソッドでパラメータを定義できるため、パラメータを外部リソースとして定義し、テスト時に読み込むこともできます(リスト 8.15、リスト 8.16)。外部リソースとしてパラメータを定義すると、テストデータのバリエーションを増やすためにテストコードを修正する必要もテストメソッドを追加する必要もありません^{注3}。

注3 ➔ 外部リソースからのセットアップ(p.114)

リスト8.14 DataPointsアノテーションの利用

```

@RunWith(Theories.class)
public class CalculatorDataPointsTest {
    @DataPoints
    public static Fixture[] PARAMs = {
        new Fixture(3, 4, 7),
        new Fixture(0, 5, 5),
        new Fixture(-3, 1, -2),
    };

    @Theory
    public void add(Fixture p) throws Exception {
        Calculator sut = new Calculator();
        assertThat(sut.add(p.x, p.y), is(p.expected));
    }

    static class Fixture {
        int x;
        int y;
        int expected;

        Fixture(int x, int y, int expected) {
            this.x = x;
            this.y = y;
            this.expected = expected;
        }
    }
}

```

4
5
6
7
8
9
10**リスト8.15** 外部リソースを使ったパラメータ化テスト

```

@RunWith(Theories.class)
public class CalculatorDataPointsYamlTest {
    @DataPoints
    public static Fixture[] getParams() {
        InputStream in = CalculatorDataPointsYamlTest.class
                        .getResourceAsStream("params.yaml");
        return ((List<Fixture>) new Yaml()
                        .load(in)).toArray(new Fixture[0]);
    }

    @Theory

```

```

public void add(Fixture p) throws Exception {
    Calculator sut = new Calculator();
    assertThat(sut.add(p.x, p.y), is(p.expected));
}

public static class Fixture {
    public int x;
    public int y;
    public int expected;
}
}

```

リスト8.16 YAMLによるパラメータの定義

```

!!seq [
  !!ch08.CalculatorDataPointsYamlTest$Fixture
  { x: 3, y: 4, expected: 7 },
  !!ch08.CalculatorDataPointsYamlTest$Fixture
  { x: 0, y: 5, expected: 5 },
  !!ch08.CalculatorDataPointsYamlTest$Fixture
  { x: -3, y: 1, expected: -2 },
]

```

8.3 組み合わせテスト

Theoriesを使ったパラメータ化テストでテストメソッドに複数の引数を指定した場合、パラメータの全組み合わせでテストメソッドが実行されます。この機能を利用して、多くのパラメータの組み合わせテストを行うことができます。

リスト8.17では、会員サイトへのエントリが可能かどうかを判定するメソッドに対して、年齢と性別の2つの引数を持つテストメソッドを定義しています。このテストを実行すると、次のようにDataPointsで定義された年齢と性別の全組み合わせ、すなわち「15歳男性、20歳男性、25歳男性、30歳男性、15歳女性、20歳女性、25歳女性、30歳女性」の8パターンがすべて出力されます。

```
canEntry(15, MALE)
canEntry(15, FEMALE)
canEntry(20, MALE)
canEntry(20, FEMALE)
canEntry(25, MALE)
canEntry(25, FEMALE)
canEntry(30, MALE)
canEntry(30, FEMALE)
```

しかしながら、これだけではテスト結果が同一のパラメータしか定義できないため、あまり使い道がありません。そこで後述のassumeThatメソッドと組み合わせて利用します。

Assumeによるパラメータのフィルタリング

パラメータ化テストによる組み合わせテストは強力ですが、すべての組み合わせについて実行します。そうすると、テストによっては期待値が一定になりません。これは、条件を満たすパラメータの組み合わせ、または条件を満たさないパラメータの組み合わせでテストを実行しなければならないからです。そこで、org.junit.Assumeクラスを利用して、パラメータをフィルタリングします。

AssumeクラスはAssertクラスとよく似たクラスであり、値の検証メソッド

リスト8.17 2つのパラメータの組み合わせテスト

```
@RunWith(Theories.class)
public class MemberCombinedTest {

    @DataPoints
    public static int[] AGES = { 15, 20, 25, 30 };

    @DataPoints
    public static Gender[] GENDERS = Gender.values();

    @Theory
    public void canEntry(int age, Gender gender) throws Exception {
        System.out.println("canEntry(" + age + ", " + gender + ")");
    }
}
```

ドを提供します。assumeThat メソッドは assertThat メソッドに対応し、引数に実測値と Matcher オブジェクトを受け取り、値の比較検証を行います。同様に assumeTrue メソッドは boolean 値が true であることの検証を行います。しかしながら、assumeThat メソッドや assumeTrue メソッドは検証結果が偽である場合にテストは失敗となりません。そのテストを無視してテストの実行を継続させます。

リスト 8.18 は assumeThat を使った簡単な例ですが、このテストを実行してもテスト失敗になりません。assumeThat メソッドや assumeTrue メソッドでは、assertThat メソッドや assertTrue メソッドと同様に値の検証を行いますが、AssertionException の代わりに AssumptionViolatedException を送出します。この例外はテストランナーの中で特別扱いされ、テスト結果は成功となります。

すなわち、assumeThat メソッドや assumeTrue メソッドを使うと、パラメータ化テストで特定の条件を満たすパラメータをフィルタリングできます。リスト 8.19 では、それぞれのテストケースの条件(①25歳以下の女性、②25歳以下の女性でない)でテストで利用するパラメータを制限しています。これらの条件の記述は宣言的に行われているため、テストコードの可読性も損ないません。

また、Assume クラスを使うことで特定の環境に依存するテストの実行もフィルタリングできます。リスト 8.20 は、Windows 環境のみで実行するテストケースです。

8.4 パラメータ化テストの問題

パラメータ化テストはテストデータを独立して定義したり、組み合わせテストを可能としたりと強力な機能ですが、いくつかの問題もあります。

データの網羅性

1つ目の問題は、データの取り方や網羅性に関して保証されるわけではないということです。この章のはじめに解説した会員サイトへのエントリ可能判定メソッドのテストをパラメータ化テストで行ったとしても、テス

リスト8.18 `assume`の利用

```
@Test
public void assume() throws Exception {
    assumeThat(1, is(0));
    fail("この行は実行されない");
}
```

リスト8.19 `assume`によるパラメータのフィルタリング

```
@RunWith(Theories.class)
public class MemberTest {

    @DataPoints
    public static int[] AGES = { 15, 20, 25, 30 };

    @DataPoints
    public static Gender[] GENDERS = Gender.values();

    @Theory
    public void canEntryは25歳以下の女性の場合にtrueを返す(
        int age, Gender gender) throws Exception {
        assumeTrue(age <= 25 && gender == Gender.FEMALE); ①
        assertThat(Member.canEntry(age, gender), is(true));
    }

    @Theory
    public void canEntryは25歳以下の女性でない場合にfalseを返す(
        int age, Gender gender) throws Exception {
        assumeTrue(25 < age || gender != Gender.FEMALE); ②
        assertThat(Member.canEntry(age, gender), is(false));
    }
}
```

リスト8.20 Windows環境のみで実行するテストケース

```
public class WindowsOnlyTest {

    @Test
    public void windows環境では改行はrn() throws Exception {
        assumeTrue(System.getProperty("os.name").contains("Windows"));
        assertThat(System.getProperty("line.separator"), is("\r\n"));
    }
}
```

トデータに25歳以下の女性が含まれていることは保証されません。また、`Assume`ですべてのパラメータが無視されてしまってもテストは成功となってしまいます。

この問題はフレームワークで検知できる類の問題ではありません。テストデータの妥当性やテストの網羅性については、テスト技法を学び、レビューなどで検証する必要があります。

パラメータに関する情報の欠落

2つ目の問題は、テストが失敗した場合に「どんなパラメータで失敗したか?」という情報が欠落してしまうことです^{注4}。

通常のテストケースでは、テストメソッド名がテストケースの概要を示しますが、パラメータ化テストではテストメソッドにパラメータの情報を含めることができません。つまり、テスト失敗時には「"canEntryは25歳以下の女性の場合にtrueを返す"が失敗した」という情報しか報告されません。具体的にどんな値を入力値としたときに、期待される結果とならなかったかがわからなければ、問題の分析に時間がかかるてしまうでしょう。

この問題を解決するには、`assertThat`メソッドの第1引数に失敗時のメッセージを指定します。リスト8.21のように失敗時のメッセージにはそのときに受け取ったパラメータ情報を含めてください。ユニットテストでは、失敗したときにそのフィードバックを受け、すばやく対処可能であることが重要です。

リスト8.21 パラメータの情報を指定するアサーション

```
@Theory
public void canEntryは25歳以下の女性でない場合にfalseを返す(
    int age, Gender gender) throws Exception {
    assumeTrue(25 < age || gender != Gender.FEMALE);
    String msg = "When age=" + age + ", gender=" + gender;
    assertThat(msg, Member.canEntry(age, gender), is(false));
}
```

^{注4} JUnit 4.10では、何番目のパラメータであるかという情報は出力されます。

第9章

ルール

テストクラスを拡張するしくみ

JUnitでは、テストクラスにTestアノテーションを付与したメソッドを作成することでテストケースを定義します。テストクラスで共通した処理はセットアップメソッドに抽出できますが、テストクラス間で共通した処理はサブクラスなどに抽出せざるを得ませんでした。また、テストクラスにはテストのメタ情報にアクセスする手段がありません。このため、テストの実行方法をカスタマイズするにはテストランナーを拡張せざるを得ませんでした。このような背景から、JUnit 4.7で追加されたしくみがルールです。

この章では、ルールの概要とJUnitで提供しているルールについて解説します。また、カスタムルールの作り方も解説します。

9.1 ルールとは？

JUnitでは、何をテストするのか(テストケース)と、どう実行するのか(テストランナー)が分離されています。このため、独自のテストランナーを作ることで、ユニットテストの実行方法をカスタマイズすることができます。しかしながら、ユニットテストの実行方法を少しだけ拡張したい状況で独自のテストランナーを作るのは手間がかかります。このため、もっと簡単に、プラグインのようにユニットテストを拡張できる機能として、ルール(Rule)がJUnit 4.7で追加されました。

ルールは再利用しやすく宣言的に記述できるJUnitの拡張フレームワークです。ルールを使うことで、共通で使う処理を独立したクラスに定義できます。また、テスト実行時のメタデータにアクセスできるため、テスト実行時の拡張も容易に行うことができます。

ルールの宣言

ルールでは、ユニットテストの拡張機能を独立したクラスとして作成し、その拡張機能を必要とするテストクラスにアノテーションを使って宣言的に記述します。

ルールは、org.junit.Rule アノテーションを付与した public フィールドに定義します（リスト 9.1 ①）。ルールとして定義するクラスは、org.junit.rules.TestRule インタフェースを実装したクラスでなければなりません。また、フィールドの宣言と一緒にインスタンスを作成するか、コンストラクタでインスタンスを生成する必要があります。

たとえば、あるテストクラスでは「1つのテストケースで 100ms 以上の時間がかかる場合は失敗としたい」ならば、リスト 9.1 のように Timeout クラスをルールとして宣言します。

ルールのしくみ

ルールで定義された拡張処理は、セットアップメソッドなどと同様に、テストメソッドの実行ごとに行われます。org.junit.ClassRule アノテーションでルールを宣言すれば、BeforeClass アノテーションと同様に、テストクラスごとに拡張処理を行うことができます。

なお、JUnit では、Timeout クラスなど、汎用的に利用できるルールをいくつか提供しています。また、プロジェクトごとに必要なユニットテストの共通処理は、カスタムルールとして簡単に作成できます。カスタムルール

リスト 9.1 Rule アノテーションによるルールの宣言

```
public class TimeoutExampleTest {
    @Rule
    public Timeout timeout = new Timeout(100); ①

    @Test
    public void 長い時間がかかるかもしれないテスト() throws Exception {
        doLongTask();
    }
}
```

ルを定義すれば、共通処理の再利用性が高まり、テストコードのメンテナブル性も高まります。詳細はこの章の後半で解説します。

複数のルールの宣言

ルールでは、リスト9.2のように複数のルールを宣言することもできます。ただし、ルールの実行順序は制御できません。制御したい場合は、後述のRuleChainを使用してください。

9.2 JUnitが提供するルール

JUnitでは、表9.1に示したルールを提供しています。単体で利用できるルールと、カスタムルールを作るときに基底クラスとして利用するルール

リスト9.2 複数のルールの宣言

```
public class RulesExampleTest {
    @Rule
    public TestName testName = new TestName();
    @Rule
    public ExpectedException expectedException = ExpectedException.none();

    @Test
    public void test() throws Exception {
    }
}
```

表9.1 JUnitが提供するルール

ルール	概要
TemporaryFolder	テンポラリフォルダの作成と解放を行うルール
ExternalResource	外部リソースを扱うカスタムルールを定義するための基底クラス
Verifier	テスト後の事後条件を検証するルール
ErrorCollector	テスト時の例外の扱いをカスタマイズするルール
ExpectedException	詳細な例外に関する検証を行うルール
Timeout	テスト時のタイムアウトを制御するルール
TestWatcher	テストの実行時の記録を行うルール
TestName	実行中のテストメソッドのメソッド名を参照できるルール

があります。また、自身でカスタムルールを作成する場合、これらのソースコードは非常に参考になります。

TemporaryFolder——一時ファイルを扱う

`org.junit.rules.TemporaryFolder` クラスは、テストの実行ごとにテンポラリフォルダを作成し、テストの実行後に作成したテンポラリフォルダを削除するルールです。また、`TemporaryFolder` クラスのメソッドを使うことで、テンポラリフォルダでのファイルやフォルダの作成が簡単に行えます。

このルールは、ファイルシステムを扱うメソッドのユニットテストで便利なルールです。ファイルシステムに関連するユニットテストでは、テストの独立性を高めるため、テストケースごとに独立したフォルダを利用したり、使用したフォルダをテストの後処理でクリアしたりします。それらの処理は初期化処理や後処理で記述できますが、このルールを宣言すればそれらの処理を省略できます。テストの初期化処理の前に、そのテスト専用のテンポラリフォルダが作成され、テスト終了時に破棄されます。

リスト 9.3 の例では、`TemporaryFolder` ルールを宣言し、テンポラリフォルダを使ったテストを行っています。

リスト 9.3 TemporaryFolder の使用

```
public class TemporaryFolderExampleTest {
    @Rule
    public TemporaryFolder tempFolder = new TemporaryFolder();

    @Test
    public void mkDefaultFilesで2つのファイルが作成される()
        throws Exception {
        File folder = tempFolder.getRoot();
        TemporaryFolderExample.mkDefaultFiles(folder);
        String[] actualFiles = folder.list();
        Arrays.sort(actualFiles);
        assertThat(actualFiles.length, is(2));
        assertThat(actualFiles[0], is("UnitTest"));
        assertThat(actualFiles[1], is("readme.txt"));
    }
}
```

● TemporaryFolder の拡張

テンポラリフォルダの中に特定のサブフォルダを作成したい場合は、リスト9.4のようにTemporaryFolderのサブクラスを作るといでしよう。

TemporaryFolderクラスのbeforeメソッドは、テンポラリフォルダの作成を行なうメソッドで、テストの実行前に呼び出されます。したがって、テンポラリフォルダにsrcフォルダとbinフォルダを追加するには、リスト9.4のようにbeforeメソッドをオーバーライドします。このとき、スーパークラスのbeforeメソッドを呼び出すことを忘れる、テンポラリフォルダが作成されないので注意してください(リスト9.4❸)。

同様に後処理をカスタマイズしたい場合は、テンポラリフォルダを削除するafterメソッドをオーバーライドします。この場合もbeforeメソッド同様、スーパークラスのafterメソッドを呼び出すことを忘れないでください。

なお、作成したサブフォルダは、ルールのフィールドに保持することで便利に利用できます(リスト9.4❶❷)。

ExternalResource——外部リソースを扱う

org.junit.rules.ExternalResourceクラスは、テストの実行前に必要なリソースを準備し、テストの実行後にリソースを解放するルールです。ExternalResourceクラスは抽象クラスであり、具体的に管理するリソース

リスト9.4 特定のサブフォルダを作成するルール

```
public class SpecificTemporaryFolder extends TemporaryFolder {
    public File srcFolder;          ①
    public File binFolder;          ②

    @Override
    protected void before() throws Throwable {
        super.before();            ❸
        srcFolder = newFolder("src");
        binFolder = newFolder("bin");
    }
}
```

はサブクラスで定義します。データベース、ソケット、組込みサーバなどを扱うテストを行うときに便利なルールです。

リスト9.5は、組込みサーバの起動と停止を行いうルールです。リスト9.5ではネストしたクラスとしてルールを定義していますが、独立したクラスとして定義すれば複数のテストクラスから再利用できます。

ExternalResourceクラスでは、beforeメソッドとafterメソッドの2つのメソッドをオーバーライドします。beforeメソッドには外部リソースの初期化処理を、afterメソッドには外部リソースの解放処理を定義してください。

初期化処理と解放処理の実行コストが大きく、クラス単位で実行すればよいならば、後述のClassRuleアノテーションを使うこともできます。

なお、前述したTemporaryFolderクラスは、このExternalResourceクラスのサブクラスです。

Verifier——事後検証を行う

org.junit.rules.Verifierクラスは、テストの最終局面で共通した検証を行いたい場合に使用するルールです。つまり、テストの事後条件を検証します。ExternalResourceクラスと同様にVerifierクラスは抽象クラスです。要件に

リスト9.5 ExternalResourceの利用例

```
@Rule
public ServerResource resource = new ServerResource();

static class ServerResource extends ExternalResource {
    Server server;

    @Override
    protected void before() throws Throwable {
        server = new Server(8080);
        server.start();
    }

    @Override
    protected void after() {
        server.shutdown();
    }
}
```

4
5
6
7
8
9
10

応じて verify メソッドをオーバーライドして使用します。

事後条件の検証は後処理メソッドで行うこともできますが、事後条件の検証が複雑であったり、複数のテストクラスで横断的に行いたい場合は、ルールとして独立したクラスに抽出すると便利です。

リスト 9.6 では、Verifier ルールを利用してテスト対象オブジェクトに対する事後条件を検証しています。この検証は、After アノテーションの付与された後処理のさらにあとで実行されます。つまり、リスト 9.6 では次の順番で実行されます。

リスト 9.6 Verifier による事後検証

```
public class VerifierExampleTest {
    @Rule
    public Verifier verifier = new Verifier() {
        protected void verify() throws Throwable {
            assertThat("value should be 0.", sut.value, is(0));
        }
    };
    VerifierExample sut;

    @Before
    public void setUp() throws Exception {
        sut = new VerifierExample();
    }

    @After
    public void tearDown() throws Exception {
        // do nothing
    }

    @Test
    public void getValueで計算結果を取得する() throws Exception {
        sut.set(200);
        sut.add(100);
        sut.minus(50);
        int actual = sut.getValue();
        assertThat(actual, is(250));
    }
}
```

- ❶ `setUp` メソッド(リスト 9.6 ❷)
- ❷ テストメソッド(同❸)
- ❸ `tearDown` メソッド(同❹)
- ❹ Verifier クラスの `verify` メソッド(同❺)

リスト 9.7 のように、テスト対象でないオブジェクトに対し、事後条件の検証を行ってもかまいません。

リスト 9.7 ErrorLog のサイズの検証を行う Verifier

```
public class VerifierExternalObjectExampleTest {  
    ErrorLog log = new ErrorLog();  
    @Rule  
    public ErrorLogVerifier errorLogVerifier = new ErrorLogVerifier(log);  
  
    @Test  
    public void testCase() throws Exception {  
    }  
}  
  
class ErrorLogVerifier extends Verifier {  
  
    final ErrorLog log;  
  
    ErrorLogVerifier(ErrorLog log) {  
        this.log = log;  
    }  
  
    @Override  
    protected void verify() throws Throwable {  
        assertThat(log.size(), is(not(0)));  
    }  
}
```

ErrorCollector——エラーを収集する

`org.junit.rules.ErrorCollector` クラスは、アサーションの失敗やエラーが発生した場合でも、テストを継続して実行することのできるしくみを提供します。テストが失敗したという情報は、`ErrorCollector` オブジェクトに蓄積され、テストを最後まで実行したあとに評価されます。

リスト9.8 はコンストラクタのテストです。オブジェクトに定義された各フィールドの初期値を検証するために `ErrorCollector` を利用しています。

検証を `ErrorCollector` クラスの `checkThat` メソッドを利用することで、検証中に期待しない結果となった場合でもテストは失敗とならず、最後まで検証が行われます。これにより、テストが失敗しても、期待する結果となるないすべての項目に関する情報を知ることができます。

なお、`ErrorCollector` クラスは、`Verifier` クラスのサブクラスです。

リスト9.8 ErrorCollectorを使ったインスタンス化テスト

```
@RunWith(Enclosed.class)
public class ShopInfoTest {
    public static class インスタンス化テスト {
        @Rule
        public ErrorCollector ec = new ErrorCollector();
        @Test
        public void デフォルトコンストラクタ() throws Exception {
            // Exercise
            ShopInfo instance = new ShopInfo();
            // Verify
            ec.checkThat(instance, is(notNullValue()));
            ec.checkThat(instance.id, is(nullValue()));
            ec.checkThat(instance.name, is(""));;
            ec.checkThat(instance.address, is(""));;
            ec.checkThat(instance.url, is(nullValue()));
        }
    }
}
```

ExpectedException —— 詳細な例外を扱う

org.junit.rules.ExpectedException クラスは、送出された例外を詳細に検証するためのルールです。

JUnitで例外の送出を検証する場合、リスト 9.9 のように Test アノテーションの expected 属性に例外クラスを指定します。この記法は JUnit の標準的な記法ですが、例外メッセージなど詳細な検証ができません。例外メッセージなどを検証する場合、リスト 9.10 のように例外オブジェクトの参照を取得し、検証用のコードを書く必要があります。

このテストコードは期待どおりのテストを実施できますが、検証用のコードが catch 節にあったり、例外が送出されなかった場合の fail メソッドが必要であったりと、非常に読みにくいコードとなっています。

一方、ExpectedException クラスを利用すれば、簡単に詳細な検証を行うことができます（リスト 9.11）。例外のメッセージまで含めてユニットテストする機会はありませんが、ExpectedException クラスは例外オブジェクトに対して詳細な検証が必要な場合に有効なルールです。

リスト 9.9 例外の発生を検証する標準的なテスト

```
@Test(expected = IllegalArgumentException.class)
public void 例外の発生を検証する標準的なテスト() throws Exception {
    throw new IllegalArgumentException();
}
```

リスト 9.10 例外の発生とメッセージを検証する標準的なテスト

```
@Test
public void 例外の発生とメッセージを検証する標準的なテスト()
throws Exception {
    try {
        throwNewIllegalArgumentException();
        fail("例外が発生しない");
    } catch (IllegalArgumentException e) {
        assertThat(e.getMessage(), is("argument is null."));
    }
}

private void throwNewIllegalArgumentException() { }
```

Timeout——テストのタイムアウトを設定する

org.junit.rules.Timeout クラスは、テストケースのタイムアウトを設定するルールです。テストの実行時間が指定された値を超えると、そのテストは自動的に失敗となります。

タイムアウトは JUnit の標準的な機能としても提供されています。リスト 9.12 のように、Test アノテーションの timeout にタイムアウト時間(単位はミリ秒)を設定できます。

一方、リスト 9.13 は Timeout クラスを利用したコードです。どちらの記法であってもタイムアウトが設定できることは変わりませんが、Timeout クラスを利用した場合、テストクラス内のすべてのテストメソッドにタイムアウトが設定されます。

リスト 9.11 例外の発生とメッセージを検証するテスト

```
public class ExpectedExceptionExampleTest {
    @Rule
    public ExpectedException expectedException = ExpectedException.none();

    @Test
    public void 例外の発生とメッセージを検証するテスト() throws Exception {
        expectedException.expect(IllegalArgumentException.class);
        expectedException.expectMessage("argument is null.");
        throw new IllegalArgumentException();
    }
}
```

リスト 9.12 アノテーションを使ったタイムアウト

```
public class TimeoutTest {
    @Test(timeout = 100L)
    public void アノテーションを使ったタイムアウト() throws Exception {
        doLongTask();
    }
}
```

リスト9.13 Timeoutルールを使ったタイムアウトの設定

```
public class TimeoutExampleTest {
    @Rule
    public Timeout timeout = new Timeout(100);

    @Test
    public void 長い時間がかかるかもしれないテスト() throws Exception {
        doLongTask();
    }
}
```

TestWatcher——テストの実行を監視する

`org.junit.rules.TestWatcher` クラスはテストの実行を監視するルールです。テスト実行時のさまざまなタイミングで追加の処理を実行できます。`TestWatcher` は抽象クラスであり、サブクラスで必要なメソッドをオーバーライドして利用します。

リスト9.14はテストの各フェーズでログを出力する `TestWatcher` です。ユニットテストのログなどを記録する必要がある場合や、ユニットテストの失敗時に通知が欲しいような場合に有効なルールです。

TestName——テスト名を扱う

`org.junit.rules.TestName` クラスは、テストメソッド内で実行中のテストメソッド名を取得するためのルールです。リスト9.15では、`TestName` オブジェクトからテストメソッド名を取得しています。

JUnitによるユニットテストでは、テストメソッド名をテストケースの概要とします。このため、テストメソッド名は有益な情報を多く含んでおり、テストメソッド内で利用したいシーンは少なくありません。テストメソッド名と外部リソースファイルの名前を関連付けることもあるでしょう。

しかしながら、Javaで実行中のメソッド名を取得するのは簡単ではありません^{注1}。`TestName` クラスを利用することで、簡単に実行中のテストメソッド名を取得できます。

注1 StackTraceオブジェクトを使うことで、`TestName` を使わずにメソッド名を取得することは可能です。

4
5
6
7
8
9
10**リスト9.14** ログを出力するTestWatcher

```

public class TestWatcherExampleTest {
    @Rule
    public TestWatcher testWatcher = new TestWatcher() {

        @Override
        protected void starting(Description description) {
            Logger.getAnonymousLogger()
                .info("start: " + description.getMethodName());
        }

        @Override
        protected void succeeded(Description description) {
            Logger.getAnonymousLogger()
                .info("succeeded: " + description.getMethodName());
        }

        @Override
        protected void failed(Throwable e, Description description) {
            Logger.getAnonymousLogger()
                .log(Level.WARNING,
                    "failed: " + description.getMethodName(), e);
        }

        @Override
        protected void finished(Description description) {
            Logger.getAnonymousLogger()
                .info("finished: " + description.getMethodName());
        }
    };

    @Test
    public void 成功するテスト() throws Exception {
    }

    @Test
    public void 失敗するテスト() throws Exception {
        fail("NG");
    }
}

```

リスト9.15 テスト名の取得

```
public class TestNameExampleTest {
    @Rule
    public TestName testName = new TestName();

    @Test
    public void テスト名() throws Exception {
        fail(testName.getMethodName() + " is unimplements yet.");
    }
}
```

9.3 カスタムルールの作成

前節では JUnit が提供するルールを解説しましたが、ルールの最大の特徴は簡単にカスタムルールを作成できることです。つまり、カスタムルールを作成すれば簡単にユニットテストの実行を拡張することができます。

カスタムルールを作るには、`org.junit.rules.TestRule` インタフェースを実装したクラスを作成します。あるいは、`ExternalResource` クラスなどの既存のルールのサブクラスを作成します。

TestRuleインターフェース

`TestRule` インタフェースには次のメソッドが定義されています。

`Statement apply(Statement base, Description description);`

`org.junit.runners.model.Statement` オブジェクトは、テストの実行を制御するオブジェクトです。`Statement` クラスに定義された `evaluate` メソッドが呼び出されるとテストが実行されます。

`apply` メソッドに引数として渡される `Statement` オブジェクトは、JUnit の通常のテスト実行を行う `Statement` オブジェクトです。`evaluate` メソッドを呼び出すと、次のようにテストが実行されます。

①テストクラスのインスタンスを生成する

②Before アノテーションの付与されたメソッドを実行する(事前処理)

③ テストメソッドを実行する**④** After アノテーションの付与されたメソッドを実行する(事後処理)

`org.junit.runner.Description` オブジェクトは、テストケースのメタ情報を保持するオブジェクトであり、そこからテストクラスの名前やテストメソッドの名前、付与されたアノテーションなどを取得できます。

`apply` メソッドの戻り値は `Statement` 型です。ルールの一般的な実装では、引数として渡された `Statement` オブジェクトのプロキシオブジェクト^{注2}を作成して返します。つまり、オリジナルの `evaluate` メソッドを実行する前後に独自の処理を行う `Statement` オブジェクトを生成します。

事前条件をチェックするカスタムルール

リスト 9.16 の `PreCondition` クラスは、テストの事前条件をチェックするカスタムルールです。

`PreCondition` クラスの実装は、`Verifier` クラスの実装を参考にしました。`apply` メソッドでは、引数として渡される `Statement` オブジェクトのプロキシオブジェクトを返します。

プロキシオブジェクトの `evaluate` メソッドでは、オリジナルの `Statement` オブジェクトに定義された `evaluate` メソッドを呼び出す前に、事前条件のチェックを行う `verify` メソッドを呼び出します。`verify` メソッドは `PreCondition` クラスのサブクラスでオーバーライドします。`verify` メソッドで例外が送出された場合、テストは実行されず、テストは失敗となります。

このように、カスタムルールを作成すれば、テストの実行前後に独自の処理を行うことができます。初期化処理メソッドや後処理メソッドで処理を行う場合と比較し、よりテストクラスから独立した形で拡張機能が提供できます。また、テストに関する多くのメタ情報を使うこともできます。

注2 オリジナル処理の前後で追加の処理を行う代理(プロキシ)のオブジェクト。Javaでは、同一のインターフェースを実装してオリジナルのオブジェクトをラップする形で実装されることが多いです。

リスト9.16 事前条件をチェックするカスタムルール

```
public abstract class PreCondition implements TestRule {
    @Override
    public Statement apply(final Statement base,
                          Description description) {
        return new Statement() {

            @Override
            public void evaluate() throws Throwable {
                verify();
                base.evaluate();
            }
        };
    }

    protected abstract void verify() throws Throwable;
}
```

OSに依存したテストを行うカスタムルール

リスト9.17のOSDependentクラスは、テストが実行される環境(OS)によって実行するテストを制御するカスタムルールです。

このルールを宣言したテストクラスで、かつ、テストメソッドにRunOnアノテーション(リスト9.18)が付与されている場合、実行環境が指定されたOSであるかをチェックし、一致しなければテストを実行しません。

先ほどのPreConditionクラスと同様に、OSDependentクラスのapplyメソッドではStatementオブジェクトのプロキシオブジェクトを返します(リスト9.19)。

プロキシオブジェクトのevaluateメソッドでは、Descriptionオブジェクトを利用し、テストメソッドに定義されたアノテーションを取得しています。もし、実行環境とアノテーションで指定されたOSが一致しない場合、テストは実行されません。

リスト9.17 OSに依存したテスト

```

public class OSDependentTest {
    @Rule
    public OSDependent osDependent = new OSDependent();

    @Test
    @RunOn(OS.WINDOWS)
    public void onlyWindows() throws Exception {
        System.out.println("test: onlyWindows");
        assertThat(File.separator, is("\\"));
    }

    @Test
    @RunOn(OS.MAC)
    public void onlyMac() throws Exception {
        System.out.println("test: onlyMac");
        assertThat(File.separator, is("/"));
    }
}

```

4
5
6
7
8
9
10**リスト9.18** RunOnアノテーション

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface RunOn {

    public enum OS {
        WINDOWS, MAC, LINUX
    }

    public OS value();
}

```

リスト9.19 OSに依存したテストを行うカスタムルール

```

public class OSDependent implements TestRule {
    @Override
    public Statement apply(final Statement base, final Description desc) {
        return new Statement() {

            @Override
            public void evaluate() throws Throwable {
                RunOn env = desc.getAnnotation(RunOn.class);

```

9.4 RuleChainによるルールの連鎖

ルールを利用して外部リソースの制御などを行っているとき、特定の順序で制御させたいケースがあります。たとえば、アプリケーションサーバとデータベースサーバを利用するユニットテストがあり、アプリケーションサーバより前にデータベースサーバを起動しておかなければならぬ、といったケースです。

それぞれの機能はカスタムルールとして簡単に定義できます。ところが、それらのカスタムルールをテストクラスにそれぞれ定義するだけでは、どちらのルールか先に処理されるかを制御できません^{注3}。

ルールの処理順序を制御する必要がある場合、org.junit.rules.RuleChain クラスを利用します。RuleChain クラスは、複数のルールを連鎖させるこ

注3 JUnit の実行時にどのルールが先に認識されるかは、Java のバージョンや JVM の実装などに依存するため。

とのできるルールです^{注4}。

リスト9.20では、RuleChainクラスを利用してアプリケーションサーバ(AppServerルール)とデータベース・サーバ(DBServerルール)の実行順序を制御しています。outerRuleメソッドで先に初期化し、あとから終了処理を行う外側のルールを指定し、aroundメソッドあとに初期化して先に終了処理を行う内側のルールを指定します。

注4 いわゆる、Chain of Responsibilityパターンです。

リスト9.20 RuleChainによるルールの連鎖

```
public class RuleChainExampleTest {
    @Rule
    public RuleChain ruleChain = RuleChain
        .outerRule(new DBServer())
        .around(new AppServer());
    @Test
    public void テスト() throws Exception {
    }
}

class AppServer extends ExternalResource {
    @Override
    protected void before() throws Throwable {
        System.out.println("start AP");
    }
    @Override
    protected void after() {
        System.out.println("shutdown AP.");
    }
}

class DBServer extends ExternalResource {
    @Override
    protected void before() throws Throwable {
        System.out.println("start DB");
    }
    @Override
    protected void after() {
        System.out.println("shutdown DB.");
    }
}
```

9.5 ClassRuleによるテストクラス単位でのルールの適用

ユニットテストでは、原則として、テストケースごとに事前処理や事後処理を行います。しかしながら、データベースサーバの起動／停止など、テストケースごとに処理を行うのではコストが大きすぎる場合もあります。このような場合、初期化処理や事後処理では、Before/After アノテーションの代わりに BeforeClass/AfterClass アノテーションを利用しました^{注5}。ルールでも同様に、Rule アノテーションの代わりに、org.junit.ClassRule アノテーションを使うことで、テストクラスごとにルールを適用できます。

ClassRule アノテーションを使用してルールを定義する方法は、Rule アノテーションを利用する場合とほとんど変わりません。TestRule インタフェースを実装したオブジェクトをテストクラスの public フィールドとして定義します。ただし、ClassRule の場合は static フィールドでなければなりません。

リスト 9.21 では、すべてのテストケースの実行前にサーバに接続し、全テストの終了後にサーバから切断しています。

なお、ClassRule を利用した場合、apply メソッドに渡される Statement オブジェクトはそのテストクラスを実行するテストランナーオブジェクトです。したがって、evaluate メソッドを呼び出すことにより、テストクラスに定義されたテストが順次実行されます。また、Description オブジェクトはテストクラスのメタ情報です。

注5 ➔ JUnit が提供するアノテーション (p.50)

リスト9.21 ClassRuleアノテーションによるサーバの接続と切断

```

public class ClassRuleExampleTest {

    @ClassRule
    public static ExternalServer externalServer = new ExternalServer();

    @Test
    public void test01() throws Exception {
        System.out.println("test01");
    }

    @Test
    public void test02() throws Exception {
        System.out.println("test02");
    }

    static class ExternalServer extends ExternalResource {

        @Override
        protected void before() throws Throwable {
            System.out.println("connect");
        }

        @Override
        protected void after() {
            System.out.println("disconnect");
        }
    }
}

```

Column**JUnitのバージョンとルール**

ルールは比較的新しいJUnitの機能です。そのため、使用するJUnitのバージョンによっては一部の機能について仕様が異なります。

特にJUnit 4.9からMethodRuleインターフェースが非推奨となり、TestRuleインターフェースを使用するように変更されているので注意してください。ほかにもTestWatcherクラスが非推奨となり、代わりにTestWatcherクラスが追加されています^{注a}。

また、ClassRuleアノテーションは2011年8月にリリースされたJUnit 4.9で、RuleChainクラスは2011年9月にリリースされたJUnit 4.10で追加された機能です。

注a 機能的には変更はありません。

第10章

カテゴリ化テスト テストケースのグループ化

JUnitでは、テストクラスまたはテストスイートを指定し、ユニットテストを実行します。このとき、最初に行われるるのはテストケースの収集です。JUnitは、テストクラスやテストスイートに含まれるすべてのテストメソッドを、テストケースとして収集します。

このとき、JUnit 4.8で導入されたカテゴリを利用すると、アノテーションでタグ付けされたテストケースから特定のタグを含む(または含まない)テストケースのみを実行することができます。

この章では、テストケースが増えてきた場合にテスト実行の時間が無視できなくなるスローテスト問題と、その解決方法として有効なカテゴリ化テストについて解説します。

10.1 スローテスト問題

ユニットテストを実践していくと、ユニットテスト特有の問題が発生することがあります。その中で厄介な問題のひとつがスローテスト問題です。スローテスト問題が発生すると、それまで効果的に機能していたユニットテストが機能しなくなる可能性もあります。

スローテスト問題とは?

通常のユニットテストは一瞬で終わります。開発環境などにもよりますが、1つのテストケースの実行に0.1秒以上かかるようであれば、それは遅いテストです。

開発が進み、ユニットテストの遅いテストが増えてくると、次第にテスト実行時間は無視できなくなるほど長くなります。ユニットテストでは、すばやいフィードバックを得るために、短いサイクルでテストを実行します。

このため、テストの実行に長い時間がかかるることは大きな問題です。このような問題を、**スローテスト問題(Slow Tests)**と呼びます。

なお、どの程度の時間でスローテストであるかの明確な定義はありません。筆者の感覚では、全ユニットテストの実行に10分以上かかるならば、「スローテスト問題が発生している」と考えます。

対策

スローテスト問題の対策には、主に次の4つの方法があります。

- テストの実行時間を短くする
- テストの実行環境を強化する
- テストを並列で実行する
- 実行するテストを絞り込む

● テストの実行時間を短くする

パフォーマンスチューニングやテスト戦略を変更し、各テストの実行時間を短くすることで、スローテスト問題の対策を行います。具体的にはテストデータを共有化したり^{注1}、リアルオブジェクトからモックオブジェクトに置換したりすることで^{注2}、テストの実行速度を高速化します。

この方法はスローテスト問題を根本的に解決します。しかしながら、テストコードを大きく変更するため、リスクも高く時間もかかります。したがって、ほとんどの場合は最終手段とするべきです。

また、あらかじめスローテストが発生することを抑制するためにパフォーマンスを考慮したテストコードを書くこともできます。しかしながら、パフォーマンスを重視したテストコードは可読性が低くなることに注意してください。特にテストデータの共通化は、ユニットテスト自体に不具合を生み出す原因となります。

^{注1} ➔共有フィックスチャ(p.108)

^{注2} ➔テストダブル(p.172)

● テストの実行環境を強化する

テストを実行するマシンのスペックを上げることで、テストの実行速度を高め、スローテスト問題の対策を行います。もし、ユニットテスト用のマシンが数世代前のスペックであるならば、最新のマシンに変更するだけでテストの実行速度が劇的に変化する可能性があります。

この方法はマシンを調達さえできればよいので、リスクがほとんどありません。ソフトウェア開発で最もコストが高いのは人件費です。たかだか10万円程度のテスト用マシンを導入することで開発が円滑に進むのであれば、非常に低コストで効果的な方法です。

● テストを並列で実行する

テストを複数のマシンで並列で実行することで、スローテスト問題の対策を行います。1台で1時間かかるテストでも、6台で効果的に分割して実行できれば、10分程度で実行できる可能性があります。近年はクラウド環境も充実しているため、ユニットテストのために仮想環境を借りることもひとつの選択肢です。

ただし、テストを並列に実行するためには、テストケースがお互いに独立し、テストの実行順番に依存しない設計になっていることが必要です。ユニットテストの原則に従っていればそれほど困難な制約ではありませんが、あとからこの条件を満たそうとすることは困難です。

テストを分割して並列に実行するには、なんらかのツールを導入したり、作成したりする必要があります。また、効果的に分割できなければ、思ったほど実行時間は短縮できません。

● 実行するテストを絞り込む

実行するテスト数を減らすことで、スローテスト問題の対策を行います。

テストの目的に応じて実行するテストケースを絞り込んで実行します。たとえば、あまり重要でないテストや時間のかかるテストは1日1回など定期的に、それ以外のテストは當時実行するなどの方針をとります。テストの目的や頻度に合わせて実施するテストを適切に選択することで、ユニットテストのすばやいフィードバックを損なわずに、テストの実行時間を短縮できるでしょう。

JUnitでは、後述のカテゴリ機能を利用することで、実行するテストを制御できます。

10.2 カテゴリ化テスト

前述したように、スローテスト問題の対策方法は1つではありません。しかしながら、JUnit 4.8で追加されたカテゴリを利用し、テストを絞り込む方法を覚えておくと便利です。

カテゴリ化テストとは？

JUnitのカテゴリは、アノテーションでタグ付けされたテストケースから特定のタグを含む(または含まない)テストケースのみを実行するしくみです。このため、目的に応じたテストを任意のタイミングで実行することができます。

このようにテストケースにどんなテストであるかを表す属性(カテゴリ)が付与し、特定のカテゴリに含まれるテストを実行したり、実行から除外するといった絞り込みをしてテストを実行したりすることを、**カテゴリ化テスト**(*Categorized Test*)と呼びます。

カテゴリ化テストを行うことで、スローテスト問題が発生したとき、目的に応じて、実行するテストを絞り込むことができます。また、テストケースのタグ付けは、アノテーションを付与するだけであるため、カテゴリ化テストの実行とは無関係に、低コストで行えます。

カテゴリ化テストの実行

カテゴリ化テストは、次の手順で実行します。

- ① カテゴリの種類を表すカテゴリクラスを作成する
- ② テストケースまたはテストクラスにカテゴリを設定する
- ③ カテゴリ化テストを実行するためのテストスイートクラスを作成する

Mavenなどテストを実行するビルドツールがJUnitのカテゴリ化テストに

対応しているならば、カテゴリ化テスト用のテストスイートクラスを作成する必要はありません。

● カテゴリクラスを作成する

最初にカテゴリとして使用するクラスを作成します。このカテゴリクラスは、単にカテゴリを特定するマーカークラス(マーカーインターフェース)^{注3}です。

たとえば、遅いテストケースを表す SlowTests クラス(リスト 10.1)と早いテストを表す FastTests クラス(リスト 10.2)を定義します。クラス名はテストクラスの命名規則と競合しないように、「～Tests」や「～TestCategory」などといった名前にしてください。

なお、カテゴリクラスはインターフェースでもかまいません。また、カテゴリクラスはいくつ定義してもかまいません。

● テストケースにカテゴリを指定する

次に org.junit.experimental.categories.Category アノテーションを使用してテストメソッドにカテゴリを指定します(リスト 10.3)。Category アノテーションの value 属性に、先ほど定義したカテゴリクラスを指定してください。

また、リスト 10.4 のように Category アノテーションはテストクラスに定義することもできます。カテゴリをテストクラスに指定した場合、そのテストクラスに含まれるすべてのテストメソッドが指定したカテゴリに属します。

なお、次のように記述することで、Category アノテーションには複数のカテゴリを指定することもできます。

```
@Category({FooTests.class, BarTests.class})
```

● テストスイートクラスを作成する

JUnit で複数のテストクラスに含まれるテストケースをまとめて実行するには、テストスイートクラスを作成する必要があります。カテゴリ化テス

^{注3} java.io.Serializable インタフェースのようにメソッドが定義されていない、何らかの目印(マーク)とするために利用されるインターフェースのこと。

リスト10.1 SlowTestsクラス

```
public class SlowTests {  
}
```

リスト10.2 FastTestsクラス

```
public class FastTests {  
}
```

リスト10.3 テストメソッドのカテゴリ指定

```
public class FooTest {  
    @Category(FastTests.class)  
    @Test  
    public void fastTest() throws Exception {  
        System.out.println("FooTest#fastTest");  
    }  
  
    @Category(SlowTests.class)  
    @Test  
    public void slowTest() throws Exception {  
        System.out.println("FooTest#slowTest");  
    }  
}
```

リスト10.4 テストクラスのカテゴリ指定

```
@Category(FastTests.class)  
public class BarTest {  
    @Test  
    public void test01() throws Exception {  
        System.out.println("BarTest#test01");  
    }  
  
    @Test  
    public void test02() throws Exception {  
        System.out.println("BarTest#test02");  
    }  
}
```

トを行う場合も同様に、テストスイートクラスを作成しなければなりません。

カテゴリ化テストでは、Suite テストランナーの代わりに org.junit.experimental.categories.Categories クラスをテストランナーとして指定します(リスト 10.5)。対象となるテストクラスの指定は、テストスイートの作成と同様に、SuiteClasses アノテーションを利用します^{注4}。実行するカテゴリの指定は、IncludeCategory アノテーションと ExcludeCategory アノテーションを利用します。

リスト 10.5 では、FastTests をテスト対象とし、SlowTests をテスト対象から除外しています。

● Eclipse からカテゴリ化テストを実行する

Eclipse からカテゴリ化テストを実行するためには、テストスイートクラスを指定してテストを実行します。パッケージエクスプローラなどで、テストスイートクラスを選択し、コンテキストメニューから[実行]-[JUnit テスト]を実行してください。

Eclipse 3.7 では、テストスイートクラスもまたテストクラスとして認識されます。このため、プロジェクトを指定してテストを実行した場合、タグ付けされたテストケースとテストスイートの両方でテストが実行されてしまします。これを回避するには、テストスイートクラスを異なるソースフォルダに配置し、ソースフォルダを指定してテストを実行してください。

注4 ➔ Suite —— 複数のテストをまとめて実行する(p.84)

リスト 10.5 テストスイートクラスの作成

```
@RunWith(Categories.class)
@IncludeCategory(FastTests.class)
@ExcludeCategory(SlowTests.class)
@SuiteClasses({ FooTest.class, BarTest.class })
public class CategorizedTest { }
```

10.3 カテゴリ化テストのパターン

スローテスト問題が発生し、カテゴリ化テストが必要となるパターンはそれほど多くありません。逆に、開発がこれらのパターンに該当するならば、開発の初期段階で、カテゴリクラスを作成し、テストケースのカテゴリ化を行っておくとよいでしょう。カテゴリ化が完了していれば、スローテスト問題が発生してもすぐにカテゴリ化テストを始められます。

ここではスローテスト問題が発生するパターンをいくつか紹介します。

データベーステスト

スローテスト問題が最もよく発生するのは、データベース接続を行うクラスのユニットテストです。データベースの初期化やコネクションの作成／解放などは多くのリソースを使います。1つのテストを実行する時間が0.5秒だったとしても、テストケースが数百件となれば、すべてのテストを実行するために数十分かかるようになります。

データベース関連のテストはいずれスローテスト問題を引き起こすと考えてよいでしょう。このため、データベーステスト用のカテゴリクラスを作成し、データベース関連のテストクラスをカテゴリ化しておくことを推奨します。

通信処理を伴うテスト

外部Webサービスを利用するテストや、テスト用のアプリケーションサーバと通信が必要なテストもスローテスト問題を引き起こしやすいテストです。ネットワークを使用するテストの実行速度はネットワークの状態にも依存しやすく、遅くなりがちです。また、常に外部サービスを利用できるとも限らないため、テスト結果が不安定になる可能性もあります。さらに、外部Webサービスをテストで何度も繰り返して実行すると相手のサービスに不必要的負荷を与えててしまいます。

したがって、テストダブル^{注5}を利用してテストするか、実行する頻度を制御しておく必要があります。このような通信処理を伴うテストはカテゴリ化しておくと便利です。

GUIを伴うテスト

ブラウザやSwingなどのGUIの操作を含んだテストもスローテストを起こしやすいテストです。GUIが含まれると、アプリケーションの起動や画面のレンダリングなどをテスト中に行う必要があるからです。また、テストの実行中にマシンを操作できない場合もあります。したがって、普段のテストの実行からは除外できるように、カテゴリ化しておくとよいでしょう。

10.4 ビルドツールによるカテゴリ化テスト

Mavenなどのビルドツールを利用してユニットテストを行うならば、もっと便利にカテゴリ化テストを行うことができます。

ビルドツールでは、独自にテストクラスとテストケースを収集するため、テストスイートを作成する必要がありません。通常はプロジェクトの特定のソースフォルダに含まれるクラスから、テストクラスを命名規則(通常は「～Test」)で抽出します。そして、デフォルトではすべてのテストが実行されます。

Mavenなどカテゴリ化テストをサポートしているビルドツールを使うならば、設定ファイルなどに対象カテゴリを指定することで、カテゴリ化テストを実行できます^{注6}。

注5 ➔ テストダブルとは？(p.181)

注6 ➔ Mavenによるカテゴリ化テスト(p.292)

Part 3

江口小手の活用実験

- 第11章 手本教習 —— 手本教習法、手本／手本教習法
- 第12章 手本一観察 —— 手本による外見による見聞調査
- 第13章 Android手本 —— UI設計効率化手本
- 第14章 ハーフハンドル —— ハーフハンドルの測定

第11章

テストダブル

テスタビリティと、モック／スタブによるテスト

理想的なユニットテストでは、依存するすべてのクラスや外部システムを使用して実行します。しかしながら、依存する本物のオブジェクトを常に使用できるとは限りません。また、ユニットテストで扱いにくいオブジェクトもあります。

このような場合、テストしやすいようにリファクタリングすることや、本物のオブジェクトを代役となるオブジェクトに置き換えて、テストを行うことができます。

この章では、テストにおけるリファクタリングや、モックやスタブとして知られるテストダブルを活用する方法について解説します。

11.1 テスタビリティを高めるリファクタリング

ユニットテストは、テスト対象となるクラスやメソッドの振る舞いを検証するプロセスです。一般的にテスト対象が複雑であればユニットテストも難しくなります。たとえば、複雑な状態を持ったクラス、多くの引数を持ったメソッド、多くのことをやりすぎているメソッドなどです。

しかしながら、テスト対象がそれほど複雑でなくとも、ユニットテストが難しい状況もあります。

本節では、そのようなテストのしやすさ(テスタビリティ)と、テストしやすいようにソースコードを改善するリファクタリングについて解説します。

テスタビリティとは？

テストのしやすさは、テスタビリティ (*testability*) と呼ばれます。

たとえば、リスト11.1のdoSomethingメソッドは、何らかの処理を行い、テスト対象クラスのdateフィールドにシステム時間を設定するメソッドで

す。リスト11.1の❶では、java.util.Dateクラスのデフォルトコンストラクタを使い、システム時刻のDateオブジェクトを生成しています。ここで、doSomethingメソッドを実行するとdateフィールドにシステム時間が設定されることを検証したいとします。

リスト11.2は、リスト11.1のdoSomethingメソッドに対するテストコードの一例です。このテストは成功する場合もありますが、失敗する場合もあります。JavaのDateオブジェクトの精度はミリ秒であるため、テストの実行が十分に早く、new Date()が実行されるタイミングでシステム時間のミリ秒が変わっていなかったならば、テストが成功します。

しかしながら、このような不安定な結果をもたらすユニットテストは、避けなければなりません。このままではテストしにくいため、テストしやすい設計にする必要があります。質の高いユニットテストを書くには、テストドリブンの高いクラス設計が必要です^{注1}。

注1 テストドリブンを高くする最良の手段のひとつは、第16章で紹介するテスト駆動開発です。

リスト11.1 システム時間に依存する処理を含むメソッド

```
public class DateDependencyExample {  
  
    Date date = new Date();  
  
    public void doSomething() {  
        this.date = new Date();      ❶  
        // 何らかの処理（省略）  
    }  
}
```

リスト11.2 システム時間に依存するメソッドのテスト

```
public class DateDependencyExampleTest {  
  
    @Test  
    public void doSomethingでdateに現在時刻が設定される()  
        throws Exception {  
            DateDependencyExample sut = new DateDependencyExample();  
            sut.doSomething();  
            // このアサーションは実行タイミングによって成功にも失敗にもなる  
            assertThat(sut.date, is(new Date()));  
        }  
    }  
}
```

リファクタリングとは？

リファクタリング (*refactoring*) とは、プログラムの外部的な振る舞いを変えずに内部構造を変更し、ソースコードを整理するテクニックです。

たとえば、あるメソッドのソースコードが長すぎるのであれば、「メソッドの抽出」を行い、いくつかのメソッドに分割して読みやすくします。あるクラスの責務が多すぎるのであれば、「クラスの抽出」を行い、適切な責務を持った新しいクラスを作成します。

どちらの場合でも、元となるメソッドの振る舞いには影響を与えずに、プログラムを再設計します。このようなリファクタリングの技術は、ソースコードの可読性を高めたり、より洗練された設計にしたりする目的で行われます。

本書では、よく使われるリファクタリングを紹介します。リファクタリングについてより深く学ぶには、書籍『リファクタリング』^{注2}を参照してください。

注2 Martin Fowler著／児玉公信、友野晶夫、平澤章、梅澤真史訳『リファクタリング——プログラミングの体質改善テクニック』ピアソン・エデュケーション(2000年)

Column

ユニットテストを前提としたリファクタリング

本来、リファクタリングを行う前にユニットテストを作成します。プログラムの外部的な振る舞いが変わらないという保証を先に作り、それから内部構造を変更するのです。もし、ユニットテストがなければ、コードの変更を行った結果、意図せずに外部的な振る舞いが変更されてしまう可能性があります。特に大規模なリファクタリングを行う場合にはユニットテストが必要不可欠です。

本章では、テストコードを書く前にプロダクションコードを修正しています。これは本当のリファクタリングではなく、リファクタリングのテクニックを利用したプロダクションコードの改善です。

一方、第16章で紹介するテスト駆動開発では、プロダクションコードよりもテストコードを先に記述します(テストファースト)。テスト駆動開発を導入すれば、常にテストコードが先にある状態となるため、リファクタリングを行い、プロダクションコードを積極的に改善できるようになります。

メソッドの抽出

リスト11.1では、Dateオブジェクトの生成が問題でした。そこで、リファクタリングを行い、テストしやすく修正します。システム時間を取得する部分を1つの独立した処理と見なし、その部分をテストコードから制御できるようにします。

はじめに、テストが難しい処理をメソッドとして抽出し、抽出したメソッドをオーバーライドしてテストする方法を紹介します。

● 处理をメソッドに抽出する

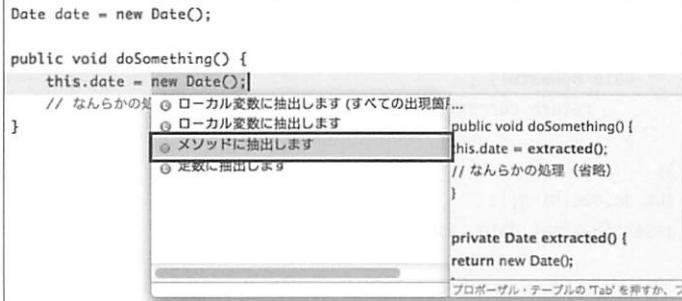
`new Date()`をメソッドとして抽出します。Eclipseを使っているのであれば、「`new Date()`」の部分を選択状態とし、ショートカット`Ctrl+1`で表示されるクイックフィックスの一覧から「メソッドに抽出します」を選択すれば簡単にメソッドとして抽出できます(図11.1)。

抽出したメソッドの名前を`newDate`メソッドと変更すると、リスト11.3のようになります。`newDate`メソッドは、テストコードでオーバーライドできるように、可視性をデフォルト(パッケージプライベート)としています。

● 抽出メソッドをテストでオーバーライドする

テストコードではテスト対象のサブクラスを定義し、テストを実行します。リスト11.4では、テスト対象クラスのサブクラスを匿名クラスとして定義し、`newDate`メソッドをオーバーライドしています。オーバーライド

図11.1 クイックフィックスによるメソッドの抽出



```

Date date = new Date();

public void doSomething() {
    this.date = new Date();
    // なんらかの処理
}

```

The 'Quick Fix' dialog is open over the line `this.date = new Date();`. The 'Method' option is selected, and a tooltip shows the extracted code:

```

public void doSomething() {
    this.date = new Date();
    // なんらかの処理
}

private Date extracted0 {
    return new Date();
}

```

されたnewDateメソッドでは、Dateクラスのコンストラクタを利用しません。テストコード内で定義された、予測可能なDateオブジェクト(current)を返します(リスト11.4①)。

なお、厳密に言えばテスト対象オブジェクトは、テスト対象クラスのサブクラスです。しかしながら、影響範囲はnewDateメソッドに限定されて

リスト11.3 システム時刻に依存する処理をnewDateメソッドに抽出

```
public class MethodExtractExample {

    Date date = newDate();

    public void doSomething() {
        this.date = newDate();
        // 何らかの処理（省略）
    }

    Date newDate() {
        return new Date();
    }
}
```

リスト11.4 newDateメソッドをオーバーライドしたテスト

```
public class MethodExtractExampleTest {

    @Test
    public void doSomethingを実行するとdateに現在時刻が設定される()
        throws Exception {
        final Date current = new Date();
        MethodExtractExample sut = new MethodExtractExample() {
            @Override
            Date newDate() {
                return current; ①
            }
        };
        sut.doSomething();
        assertThat(sut.date, is(sameInstance(current)));
    }
}
```

いるので、成功するか失敗するかわからない不安定なテストを実行したり、テストを実行しなかったりするよりはよいでしょう。

このようにテスト時に扱いが難しい処理がテスト対象メソッドに含まれている場合、その部分をメソッドとして抽出し、テストコードでオーバーライドすると簡単にテストできるようになります。

委譲オブジェクトの抽出

処理を置き換えるための2つ目の方法は、処理をオブジェクトとして抽出して委譲する方法です。ある特定の処理を抽出して独立したクラスにすることは良いクラス設計です。ただし、この方法は「システム時間の生成」といった簡単な処理では大袈裟な方法かもしれません。

● 処理を委譲オブジェクトに抽出する

はじめに、委譲したい処理を `newDate` メソッドに抽出します(リスト 11.3)。次に、この処理(メソッド)をオブジェクトに変換し、テストコードで置き換えられるようにします。

Java では、オブジェクトとして扱うためにはクラスを定義しなければな

Column

メソッドとオブジェクト

Java のメソッドはクラスの定義の一部であり、オブジェクトではありません。一方、JavaScript や Groovy などの言語では、メソッドをオブジェクトとして扱うことができます。

メソッドをオブジェクトとして扱える言語では、ユニットテスト時にテスト対象オブジェクトのメソッドを別のメソッドで置き換えることは簡単です。置き換えるメソッドオブジェクトを作成し、フィールドと同じようにテスト対象オブジェクトに設定するだけです。

しかしながら、Java ではメソッドをオブジェクトとして扱うことができません。このため、処理を行うクラスを定義し、テスト対象オブジェクトでは外部から差し替えることができるようオブジェクトとして保持します。

なお、2013年夏にリリース予定のJDK 8で導入されるラムダ式を利用すれば、单一の処理を行う委譲オブジェクトは簡潔に記述できるようになります。

りません。リスト11.5、11.6のようにDateFactoryクラスを作成し、抽出した処理(メソッド)を移動します。DateFactoryオブジェクトはフィールドに保持されているので、テストコードでDateFactoryクラスのサブクラスを作成し、差し替えることができます(リスト11.7)。

●委譲オブジェクトをテストで置き換える

さらに、DateFactory クラスからインターフェースを抽出し、ポリモフィズム^{注3}を利用した設計にすることもできます。

はじめに DateFactory インタフェースを定義し、デフォルトの実装を用意します(リスト 11.8、11.9)。DateFactory インタフェースとデフォルト実装を使いリスト 11.1 を書き換えると、リスト 11.10 となります。テストコードでは、リスト 11.7 と同様に、テスト対象となるオブジェクトの DateFactory フィールドを差し替えてテストします。

A horizontal row of twelve small circles, each containing a dot.

このように、処理を委譲オブジェクトへ抽出することで、テストビリティの高い設計に改善できます。クラスに抽出することで、そのクラスは独立した責務を持つため、個別にテストを行うこともできます。また、インターフェースを利用した設計とするならば、実装クラスが提供されていなくとも仮の実装クラスでユニットテストを書くこともできます。

ただし、インターフェースとして抽出することは、過剰設計となる可能性があるので注意してください。過剰設計もまたテストabilityを低下させます。テストabilityを高くする目的であれば、多くの場合で、オーバーライド可能なメソッドとして抽出したり、処理を委譲オブジェクトに抽出したりするだけで、十分な効果があるでしょう。

注3 オブジェクト指向の特徴のひとつで、多型性とも呼ばれます。Javaでは、同一インターフェースを異なるクラスがインプリメントすることで、同じ型の同じ名前のメソッドに、それらの異なるクラスごとに異なる振る舞いを実装できることを言います。

リスト11.5 newDateを定義した新しいクラス

```
public class DateFactory {
    Date newDate() {
        return new Date();
    }
}
```

リスト11.6 DateFactoryクラスの利用

```
public class DelegateObjectExample {

    DateFactory dateFactory = new DateFactory();
    Date date = new Date();

    public void doSomething() {
        this.date = dateFactory.newDate();
        // 何らかの処理（省略）
    }
}
```

リスト11.7 インタフェースを使って処理を差し替えたユニットテスト

```
public class DelegateObjectExampleTest {
    @Test
    public void doSomethingを実行するとdateに現在時刻が設定される()
        throws Exception {
        final Date current = new Date();
        DelegateObjectExample sut = new DelegateObjectExample();
        sut.dateFactory = new DateFactory() {
            @Override
            public Date newDate() {
                return current;
            }
        };
        sut.doSomething();
        assertThat(sut.date, is(sameInstance(current)));
    }
}
```

リスト11.8 DateFactoryクラスのインターフェース化

```
public interface DateFactory {
    Date newDate();
}
```

リスト11.9 DateFactoryクラスのデフォルト実装

```
public class DateFactoryImpl implements DateFactory {
    @Override
    public Date newDate() {
        return new Date();
    }
}
```

リスト11.10 インタフェースを使ってシステム時間の処理を委譲

```
public class DelegateObjectExample {

    DateFactory dateFactory = new DateFactoryImpl();
    Date date = new Date();

    public void doSomething() {
        this.date = dateFactory.newDate();
        // 何らかの処理（省略）
    }
}
```

Column**DIとユニットテスト**

ユニットテストで依存するオブジェクトをインターフェースなどを利用して差し替える手法は、DI(*Dependency Injection*、依存性の注入)と相性の良い手法です。

DIは「ソフトウェアはコンポーネントの組み合わせで構成される」という考え方に基づいています。各コンポーネント間の依存関係はインターフェースを介して最小限とし、コンポーネント間の依存関係はDIコンテナと呼ばれるオブジェクトが解決するというのがDIの基本です。コンポーネント間の依存関係を小さくし、分離することで、各コンポーネントが再利用しやすく独立性を高くできます。

DIは、Spring FrameworkやSeasar2などのフレームワークで使われています。その主な目的は、コンポーネントの依存関係を最小化することです。その結果、ソフトウェアの設計が適切にレイヤ分割されます。そして、依存オブジェクトを仮実装したりテストダブルで置き換えることが簡単になるため、ユニットテストも行いやすくなります。

11.2 テストダブルとは？

テスト対象となるクラスやメソッドが、ほかのクラスに依存していないケースはほとんどありません。依存しているクラスもまた、ほかのクラスに依存しています。したがって、ユニットテストでは、テスト対象クラスに加え、依存しているクラスも含めてテストしていると言えます。これにはメリットもあればデメリットもあります。

メリットは、ソフトウェアの実行時に近い状態でテストが実行されることです。ソフトウェアは多数のクラスが依存し合って動作するため、クラス単体の動作を保証するよりも、ソフトウェア全体の動作を保証するほうが価値が高くなります。

一方、テスト対象クラス以外の問題を原因として、ユニットテストが失敗する可能性があることは大きなデメリットです。テストが失敗したときには、最初にテスト対象クラスに問題があると考えるため、原因の分析に余計な時間がかかります。また、依存する機能が外部サービスや乱数、システム時間など、期待する結果を予測できない場合には検証の精度を落とさざるを得ません。

テスト対象クラスが依存するクラスをどう扱うかは、簡単に結論を出しがれません。しかしながら、Dateクラスのデフォルトコンストラクタのように、依存する機能がユニットテストで扱いづらい場合や、ユニットテストの独立性を高めたい場合には、依存するオブジェクトを「代役」で置き換える方法が有効です。この代役となるオブジェクトはスタブやモックとして知られ、総称してテストダブル(*test double*)^{注4}と呼ばれます。

しかしながら、JUnitではモックやスタブの機能を提供していません。そのため、通常は外部のライブラリを利用します。本書では、外部ライブラリのひとつである Mockito の使い方をあとで紹介します。

スタブ——依存オブジェクトに予測可能な振る舞いをさせる

スタブ(stub)とは、依存するクラスやモジュールの代用として使用する

^{注4} 「double」は「(演劇や映画の)代役、影武者」などを意味します。

仮のクラスやモジュールです。ユニットテストにおいてスタブを使う目的は、依存オブジェクトに予測可能な振る舞いをさせることです。主に次のような場合に使用されます。

- ・依存オブジェクトが予測できない振る舞いをする
- ・依存オブジェクトのクラスがまだ存在しない
- ・依存オブジェクトの実行コストが高く、簡単に利用できない
- ・依存オブジェクトが実行環境に強く依存している

● 固定値を返すスタブ

スタブの例として、ランダムな整数を返す機能を考えます。`java.util.Random` クラスの `nextInt` メソッドはランダムな整数を生成する Java の標準的なメソッドです。しかしながら、この `nextInt` メソッドはユニットテストでは扱いにくいメソッドです。そこで、同様の機能を持つインターフェースを定義し(リスト 11.11)、`Random` クラスを利用するデフォルト実装を定義します(リスト 11.12)。

リスト 11.13 は、固定値を返す `RandomNumberGenerator` インタフェースのスタブ実装です。このようにインターフェースを定義することで、ユニットテストではスタブ実装を作成して利用できます。スタブ実装であれば不確実性を排除できるため、正しくユニットテストを行うことができるでしょう。

テスト対象クラスでは、リスト 11.9 と同じように乱数生成オブジェクトのインスタンスをフィールドとして定義します(リスト 11.14)。

リスト 11.15 では、固定値を返す匿名クラスから生成したスタブオブジェクトを作成し、テスト対象クラスの乱数生成オブジェクトを差し替えていきます。

なお、スタブ実装はテストに応じて自由に作ることができます。各テストケースで無名クラスのオブジェクトを作成することも、汎用的に作成して再利用できるクラスを作成することもできます。たとえば、リスト 11.16 のようなスタブ実装を用意すれば、再利用できます。ただし、汎用的すぎるクラスはテストコードの可読性を落とす原因となるので注意してください。

ほかにも、乱数生成オブジェクトを返すメソッドを定義しテストクラスでオーバーライドする方法、Setterメソッドを用意する方法、ファクトリクラスを作成する方法、DIコンテナなどのフレームワークを使用する方法

リスト11.11 亂数生成のインターフェース

```
public interface RandomNumberGenerator {
    int nextInt();
}
```

リスト11.12 亂数生成のデフォルト実装

```
public class RandomNumberGeneratorImpl implements RandomNumberGenerator {
    private final Random rand = new Random();

    @Override
    public int nextInt() {
        return rand.nextInt();
    }
}
```

リスト11.13 固定値を返すスタブ実装

```
public class RandomNumberGeneratorFixedResultStub
    implements RandomNumberGenerator {
    @Override
    public int nextInt() {
        return 1;
    }
}
```

リスト11.14 亂数生成オブジェクトをフィールドに持つクラス

```
public class Randoms {
    RandomNumberGenerator generator = new RandomNumberGeneratorImpl();

    public <T> T choice(List<T> options) {
        if (options.size() == 0) return null;
        int idx = generator.nextInt() % options.size();
        return options.get(idx);
    }
}
```

11
12
13
14

などがあります。筆者はなるべくコードはシンプルに記述量を少なくするほうが好みなので、リスト 11.14 のようにパッケージプライベートのフィールドとして定義する方法をよく使います。

リスト 11.15 亂数生成オブジェクトのスタブを利用したテスト

```
public class RandomsTest {

    @Test
    public void choiceでAを返す() throws Exception {
        List<String> options = new ArrayList<String>();
        options.add("A");
        options.add("B");
        Randoms sut = new Randoms();
        // スタブの設定
        sut.generator = new RandomNumberGenerator() {
            @Override
            public int nextInt() {
                return 0;
            }
        };
        assertThat(sut.choice(options), is("A"));
    }

    @Test
    public void choiceでBを返す() throws Exception {
        List<String> options = new ArrayList<String>();
        options.add("A");
        options.add("B");
        Randoms sut = new Randoms();
        // スタブの設定
        sut.generator = new RandomNumberGenerator() {
            @Override
            public int nextInt() {
                return 1;
            }
        };
        assertThat(sut.choice(options), is("B"));
    }
}
```

●例外を送出するスタブ

オブジェクトによっては、例外が発生する条件が非常に複雑であったり、外的要因に依存したりします。このため、例外ハンドリングに関するユニットテストは簡単にはできません。ですが、スタブオブジェクトで無条件に例外を送出するように実装すれば、簡単に例外ハンドリングのテストを行うことができます。

たとえば、リスト11.17は、RDBからユーザ情報を取得するインターフェースです。実装では、さまざまな理由からユーザオブジェクトを取得できないことが想定されます。UserDaoインターフェースを使う処理では、UserNotFoundExceptionが送出される可能性を考慮し、例外ハンドリングが必要です。

UserDaoインターフェースを利用するクラスで、例外ハンドリングのテストを行いたいならば、リスト11.18のように無条件で例外を送出するスタブオブジェクトを作成し、デフォルトのオブジェクトと置き換えます。

リスト11.16 再利用可能なスタブ実装

```
public class RandomNumberGeneratorStub implements RandomNumberGenerator {
    private final int num;

    public RandomNumberGeneratorStub(int num) {
        this.num = num;
    }

    @Override
    public int nextInt() {
        return num;
    }
}
```

リスト11.17 ユーザ情報を取得するインターフェース

```
public interface UserDao {
    User find(String userId) throws UserNotFoundException;
}
```

リスト11.18 UserDaoの例外を送出するスタブ

```
public class UserDaoStub implements UserDao {

    @Override
    public User find(String userId) throws UserNotFoundException {
        throw new UserNotFoundException("connection error");
    }

}
```

モック——依存オブジェクトの呼び出しを検証する

モック(*mock*)は、スタブと同様に、テスト対象が依存するクラスやモジュールの代用として使用するクラスやモジュールです。モックは、依存クラスやモジュールが正しく利用されているかを検証する目的で使われます。

ユニットテストでモックを使う場合、テストコードで依存クラスのメソッドが正しく利用されているかを検証できるようにモックオブジェクトを用意します。

たとえば、リスト11.19のようにモックを定義すれば、期待するメソッドが呼び出されたかを検証できます。リスト11.19では、RandomNumberGeneratorクラスのサブクラスを定義し、検証対象となるnextIntメソッドをオーバーライドしています。オーバーライドされたメソッドでは、呼び出されたかどうかを保持する変数isCallNextIntMethodを利用し、実行された場合にtrueを設定しています。テストケースの最後では、nextIntメソッドが呼び出されたことを検証しています。

● モックとスタブの違い

このようにモックとスタブはよく似たテクニックです。スタブは仮の実装で予測可能な値を返すことで、予測が難しいテストケースを検証可能なテストケースにします。一方、モックは目的のメソッドがテストの実行中に呼び出されたかを検証することが主な目的です。

テストケースによっては、両方の目的で使用されることもあるため、モックなのかスタブなのかがいまいな場合もあります。また、モックは基本的にテストでしか利用しませんが、スタブはテストコード以外でも仮実

リスト11.19 RandomNumberGeneratorクラスのnextIntが呼び出されたかを検証するテスト

```

public class RandomsMockTest {

    @Test
    public void choiceでAを返す() throws Exception {
        List<String> options = new ArrayList<String>();
        options.add("A");
        options.add("B");
        Randoms sut = new Randoms();
        // モックの設定
        final AtomicBoolean isCallNextIntMethod = new AtomicBoolean(false);
        sut.generator = new RandomNumberGenerator() {
            @Override
            public int nextInt() {
                isCallNextIntMethod.set(true);
                return 0;
            }
        };
        assertThat(sut.choice(options), is("A"));
        assertThat(isCallNextIntMethod.get(), is(true));
    }
}

```

装として利用されることがあります。

なお、モックオブジェクトの作成は手作業で行うと非常に面倒であるため、通常は後述の Mockito などのライブラリを利用します。

スパイ——依存オブジェクトの呼び出しを監視する

基本的なユニットテストでは、入力(メソッドへの引数)に対する出力(戻り値)が期待した値となることを検証します。このため、テスト対象オブジェクトが状態を持たず、戻り値を返すメソッドは、最もテストしやすいメソッドです。

一方、テスト対象メソッドに戻り値がない場合や、テスト対象オブジェクトの状態が変わるのはテストが難しくなります。そして、メソッドを実行したときに依存オブジェクトの状態が変わる場合、検証はさらに困難

です。

依存オブジェクトへの副作用があるテストの典型的な例は、標準出力やロガーへ書き込まれた内容の検証です。

リスト 11.20 の doSomething メソッドでは、ロガーオブジェクトの info メソッドを呼び出しています。このメソッドで適切にロガーオブジェクトが呼び出されているかを検証することは簡単ではありません。検証を行うひとつ的方法は、**リスト 11.21** のようにロガーオブジェクトにどのような操作が行われたかを記録できる代役のオブジェクトを作成することです。リスト 11.21 では、本物のロガーオブジェクトに書き込みを行いつつ、自分自身の StringBuffer フィールドに出力された文字列を記録しています。この

Column

状態に着目するテストと相互作用に着目するテスト

ユニットテストは、状態に着目するテストと相互作用に着目するテストに分類できます。スタブはオブジェクトの状態に着目するテストで利用し、モックはオブジェクトの相互作用に着目するテストで利用するため、両者の違いはテストの観点の違いとも言えるでしょう。

オブジェクトの状態に着目したテストは、テスト実行後のテスト対象オブジェクトの状態やメソッドの戻り値の状態を検証するテストです。極端に言えば、テストの前提条件を満たし、特定のメソッドを実行したとき、予想された結果となることがすべてであり、内部実装がどうなっていようと関係ありません。テスト対象オブジェクトに対する入出力のテストでしかないため、依存オブジェクトが特定の結果を返す、例外を送出するなどといった予測可能な振る舞いを行うことでテストが成立します。

一方、オブジェクトの相互作用に着目したテストでは、依存オブジェクトが予測可能な振る舞いを行うだけでなく、テスト対象オブジェクトからどのようなメソッドがどんな順番で呼び出されたかも検証します。このため、相互作用に着目したテストは、予測可能な振る舞いを行う依存オブジェクトであってもモックに差し替えて、各メソッドがどのように呼び出されたかを検証する必要があります。

なお、相互作用に着目したテストは、内部実装に依存したテストです。内部実装に依存したテストは、^{もろい} テスト (fragile test)^{注a} となるため、必要以上に行わないよう注意してください。筆者は、なるべく状態に着目したテストを行うように心がけています。

注a 変更に弱く壊れやすいテスト。

SpyLoggerクラスを利用すれば、doSomethingメソッドのテストは、リスト11.22のように記述できます。

このように、どのような操作が行われたかを記録／追跡できるように、オリジナルのオブジェクトをラップしたオブジェクト^{注5}はスパイ(spy)と呼ばれます。

注5 いわゆるプロキシオブジェクトパターンです。

リスト11.20 ログ出力を行うテスト対象クラス

```
public class SpyExample {
    Logger logger = Logger.getLogger(SpyExample.class.getName());

    public void doSomething() {
        logger.info("doSomething");
    }
}
```

11
12
13
14

リスト11.21 ロガーのスパイ

```
public class SpyLogger extends Logger {
    final Logger base;
    final StringBuffer log = new StringBuffer();

    public SpyLogger(Logger base) {
        super(base.getName(), base.getResourceBundleName());
        this.base = base;
    }

    // infoのみ対応
    @Override
    public void info(String msg) {
        log.append(msg);
        base.info(msg);
    }
}
```

リスト11.22 ロガーオブジェクトにログを記録していることを検証するテスト

```

@Test
public void SpyLoggerを利用したテスト() {
    SpyExample sut = new SpyExample();
    SpyLogger spy = new SpyLogger(sut.logger);
    sut.logger = spy;
    sut.doSomething();
    assertThat(spy.log.toString(), is("doSomething"));
}

```

11.3 Mockitoによるモックオブジェクト

JUnitではスタブやモックといったテストダブルを扱う機能は提供していません。このため、スタブやモックを使ってテストケースを作成するときは、外部ライブラリを利用します。

前節で書いたように、単純なスタブならば独自に実装することも簡単です。しかし、モックオブジェクトを作成して各メソッドの呼び出し回数の検証などを行うのは大きな手間なので、ライブラリを導入します。

テストダブルを扱うライブラリは一般的にモック用ライブラリとして公開されています。なぜならば、モック機能はスタブ機能のほぼ上位互換となるためです。目的や使い方によってそのオブジェクトがモックなのかスタブなのかという違いはありますが、APIとしては同じであるライブラリがほとんどです。なお、スパイはライブラリによっては対応していません。

Mockitoとは？

Javaのモック用ライブラリとしては、Mockito^{注6}、EasyMock^{注7}、jMock^{注8}などが多く知られています。本書ではMockitoを紹介します。

Mockitoを利用してすることで、簡単にスタブオブジェクトを作成し、スタブオブジェクトのメソッドが呼び出されたときに期待される振る舞いをさ

^{注6} <http://code.google.com/p/mockito/>

^{注7} <http://easymock.org/>

^{注8} <http://www.jmock.org/>

せることができます。また、モックオブジェクトによるメソッド呼び出しの検証や、実オブジェクトをラップしたスパイオブジェクトによる監視も行うことができます。

Mockitoを利用する準備

Eclipseのプロジェクトで Mockito を使用する場合は、プロジェクトサイト(URLは脚注6)から JAR ファイルをダウンロードして、プロジェクトにコピーします(図11.2)。次に JAR ファイルのコンテキストメニューを開き、「[ビルド・パス] - 「ビルド・パスに追加」を選択し、JAR ファイルをビルドパスに追加します。

プロジェクトのビルドツールとして Maven を使っている場合は、依存ライブラリに追加してください。^{注9}

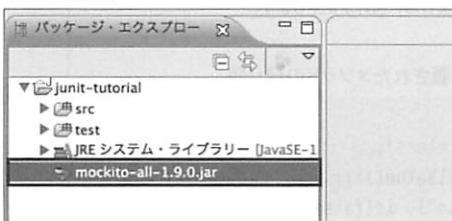
11
12
13
14

モックオブジェクトの作成

Mockito を利用してモックオブジェクトを作成するには、org.mockito. Mockito クラスの mock メソッドを使用します。mock メソッドの引数には、作成するモックオブジェクトの型オブジェクトを指定します。たとえば、List<String>型のモックオブジェクトを作成するには、次のように記述してください。

注9 ➔ 依存ライブラリの管理(p.283)

図11.2 JARファイルの追加



```
// モックオブジェクトの作成
List<String> mock = org.mockito.Mockito.mock(List.class);
```

mock メソッドはジェネリクスメソッドであり、戻り値は引数で指定した型と同じ型です^{注10}。 Mockito では、通常のクラスでもインターフェースでも、同じようにモックオブジェクトを作ることができます。ただし、final クラスは指定できません^{注11}。

スタブメソッドの戻り値

mock メソッドで生成されたモックオブジェクトでは、すべてのメソッドが null(プリミティブ型の場合はデフォルト値)を返すスタブメソッドとして設定されます(リスト 11.23)。

Mockito ではモックオブジェクトとスタブオブジェクトに API として明確な区別はありません。メソッド呼び出しの検証を行うならばモックオブジェクトであり、スタブメソッドのみを利用するならばスタブオブジェクトです。どちらも mock メソッドで作成します。

スタブメソッドの定義

モックオブジェクト(スタブオブジェクト)に定義されたスタブメソッドの戻り値を再定義するには、Mockito クラスの when メソッドとその戻り値の型である OngoingStubbing クラスの thenReturn メソッドを使用します。

たとえば、get メソッドに 0 を指定して呼び出すと文字列 Hello を返したい

注10 Java のジェネリクスの制約から、String の List(List<String>) を型パラメータに渡すことはできません。

注11 コンパイルエラーにはなりませんが、実行時に例外が発生します。

リスト 11.23 モックオブジェクトに定義されたメソッドの戻り値

```
// モックオブジェクトの作成
List<String> mock = mock(List.class);
assertThat(mock.get(0), is(nullValue()));
assertThat(mock.contains("Hello"), is(false));
```

ならば、リスト11.24のようにスタブオブジェクトを設定します。whenメソッドの引数の部分では対象となるメソッド(get)を実行し、そのメソッドの戻り値を引数とします。whenメソッドはOngoingStubbingオブジェクトを返すので、thenReturnメソッドを連鎖して呼び出しています。このとき、thenReturnメソッドには、スタブメソッドの戻り値(Hello)を指定します。

このコードは自然言語(英語)として「when mock.get(0) then return "Hello"」(getメソッドが引数0で呼び出されたときの戻り値をHelloとする)と読めるように設計されています。この処理が実行されるときには裏側で複雑な処理が行われていますが、Mockitoの利用者は自然な記述方法でスタブを定義できます。

●例外を送出するスタブメソッド

スタブメソッドに例外を送出させたい場合は、thenReturnの代わりにthenThrowメソッドを使用します(リスト11.25)。なお、メソッドが送出できないチェック例外は設定できません。

●void型を返すスタブメソッド

通常は戻り値がvoid型のスタブメソッドを再定義することはありません。なぜならば、戻り値がvoid型のメソッドは戻り値が一定であり、オブジェクトの状態を変化させるメソッドだからです。

とはいっても、戻り値がvoid型のメソッドを呼び出したときに例外を送出させたい場合もあります。ところが、Java言語の制約からMockitoクラスの

リスト11.24 スタブメソッドの定義

```
List<String> stub = mock(List.class); // スタブオブジェクトの作成
when(stub.get(0)).thenReturn("Hello"); // スタブメソッドの定義
assertThat(stub.get(0), is("Hello")); // 検証
```

リスト11.25 例外を送出するスタブメソッド

```
List<String> stub = mock(List.class);
when(stub.get(0)).thenReturn("Hello");
when(stub.get(1)).thenReturn("World");
when(stub.get(2)).thenThrow(new IndexOutOfBoundsException());
stub.get(2); // 例外が送出される
```

when メソッドは利用できません。そこで、リスト 11.26 のように doThrow メソッドを使います。リスト 11.26 では、List オブジェクトの clear メソッドが呼び出されたときに例外を送出させています。

●任意の引数に対するスタブメソッド

Mockito でモックのスタブメソッドを定義するときは、そのメソッドが呼び出されるときの引数を同時に指定します。すなわち、指定したメソッドが指定した引数で呼び出されたときに指定した結果を返します。

しかしながら、任意の引数に対して一定の戻り値を返したい場合や、引数として渡されるオブジェクトが予測できない場合もあります。このような場合は、リスト 11.27 のように anyInt メソッドなどを利用します。

anyInt メソッド以外にも、anyString メソッドや anyBoolean メソッドなどを利用できます。メソッドの引数の型によって使い分けてください。

スタブメソッドの検証

Mockito でスタブオブジェクトを利用するならば、前節までで解説したように、スタブオブジェクトに対し、スタブメソッドを定義すれば十分です。モックオブジェクトのスタブメソッドがどのように呼び出されたかを検証するには、Mockito クラスの verify メソッドを利用します。

リスト 11.28 では、モックオブジェクトの clear メソッドが 1 回、引数に Hello を指定した add メソッドが 2 回呼ばれ、引数に World を指定した add

リスト 11.26 void 型を返すスタブメソッド

```
List<String> stub = mock(List.class);
doThrow(new RuntimeException()).when(stub).clear();
stub.clear(); // 例外が送出される
```

リスト 11.27 任意の整数に対するスタブメソッド

```
List<String> stub = mock(List.class);
when(stub.get(anyInt())).thenReturn("Hello");
assertThat(stub.get(0), is("Hello"));
assertThat(stub.get(1), is("Hello"));
assertThat(stub.get(999), is("Hello"));
```

メソッドが呼ばれなかったことを検証しています。

`verify` メソッドの第1引数にはモックオブジェクトを指定します。

第2引数には検証するメソッドの期待される呼び出し回数を指定します。ここで `times` メソッドを使用すれば、実行回数を指定した検証を行なうことができます。また、`never` メソッドを使用すれば、そのメソッドが実行されていないことを検証できます。指定を省略した場合は「`times(1)`」を指定した場合と同等です。このほか、`atLeastOnce` メソッド(最低でも1回)、`atLeast` メソッド(最低でも `n` 回)、`atMost` メソッド(最大で `n` 回)などが提供されて

リスト11.28 スタブメソッドの検証

```
List<String> mock = mock(List.class);
mock.clear();
mock.add("Hello");
mock.add("Hello");
verify(mock).clear();
verify(mock, times(2)).add("Hello");
verify(mock, never()).add("World");
```

Column

Mockitoの後置記法と前置記法

Mockitoでは、モックオブジェクトにスタブメソッドを定義するときに、後置記法と前置記法の記法を使うことができます。

後置記法では、次のように `when` メソッドを利用し、モックオブジェクトでスタブメソッドを呼び出し、`thenXxx` メソッドで戻り値を定義します。

```
when(mock.get(0)).thenReturn("Hello");
when(mock.get(1)).thenThrow(new IndexOutOfBoundsException());
```

前置記法では、次のように、はじめに `doXxx` メソッドで記録する戻り値を定義します。あとから、`when` メソッドにモックオブジェクトを指定し、スタブメソッドを実行します。

```
doReturn("Hello").when(mock).get(0);
doThrow(new IndexOutOfBoundsException()).when(mock).get(1);
```

どちらも同じモックオブジェクトを定義していますが、戻り値が `void` のメソッドでは、後者の記法でしか定義できません。

います。

最後に、verifyメソッドの戻り値に対し、検証対象のメソッドを引数を指定して呼び出します。

このようにモックオブジェクトにverifyメソッドを使用することで、スタブメソッドがどのように呼び出されたかについて、厳密に検証できます。しかしながら、モックオブジェクトの呼び出しを検証することは、テスト対象オブジェクトの内部実装に強く依存することに注意してください。

部分的なモックオブジェクト

Mockitoのモックオブジェクトは、指定した型を模したダミーオブジェクトです。このため、すべてのスタブメソッドはデフォルトの戻り値としてnullやプリミティブ型の基本値を返します。しかしながら、一部のメソッドのみをスタブメソッドに置き換えることがあります。mockitoでは、実オブジェクトを利用し、部分的なモックオブジェクトを作成することができます。

部分的なモックオブジェクトを作成するには、リスト11.29のようにspyメソッドを利用します。この方法で作成されたオブジェクトは、スタブメソッドが定義されていない限り、リアルオブジェクトの対応するメソッドが呼び出します。したがって、getメソッドは「Hello」を返し、sizeメソッドは「100」を返します。

なお、部分的なモックオブジェクトを作成するときに利用するメソッドの名前が「spy」である理由は、歴史的な経緯と後述のスパイオブジェクトが部分的なモックオブジェクトとして実現しているためです。

リスト11.29 部分的なモックオブジェクト

```
List list = new ArrayList();
List spy = spy(list);
when(spy.size()).thenReturn(100);
spy.add("Hello");

assertThat(spy.get(0), is("Hello"));
assertThat(spy.size(), is(100));
```

スパイオブジェクト

Mockitoでは、実オブジェクトを監視するスパイオブジェクトを作成することができます。スパイオブジェクトは、テスト対象オブジェクトがレガシーコードに依存している場合に有効です。ただし、スタブオブジェクトやモックオブジェクトと同様に、finalクラスのスパイオブジェクトは作成できません。

スパイオブジェクトを利用するには、リスト11.30のように、Mockitoクラスのspyメソッドを利用してスパイオブジェクトを生成します。スパイオブジェクトには、モックオブジェクトと同様に、特定のメソッドの戻り値を再定義できます。スタブメソッドが定義されていない場合は、実オブジェクトの対応するメソッドが実行されます。また、verifyメソッドを利用し、メソッドの呼び出し回数などを検証することができます。

このように、スパイオブジェクトは実オブジェクトの一部に対しスタブメソッドを定義できます。なお、「when～thenReturn」の形式はスパイオブジェクトでは利用できません。

● メソッド実行時のコールバック

スタブメソッドを定義するとき、Answerクラスを使いメソッド実行時のコールバックを登録できます。リスト11.31、11.32では、ロガーを扱うメソッドの実行時に、ロガーオブジェクトのメソッドを監視しています。

Answerクラスのanswerメソッドで引数として渡されるInvocationOnMockオブジェクトは、実オブジェクトのメソッド実行を表すオブジェ

リスト11.30 スパイオブジェクトの作成

```
List<String> list = new java.util.LinkedList<String>();
List<String> spy = spy(list);
doReturn("Mockito").when(spy).get(1);
spy.add("Hello");
spy.add("World");
verify(spy).add("Hello");
verify(spy).add("World");
assertThat(spy.get(0), is("Hello"));
assertThat(spy.get(1), is("Mockito"));
```

クトです。実オブジェクトでの本来の振る舞いをさせるには、callRealMethod メソッドを呼び出します。このコードでは、ロガーに出力する前にその文字列を監視し、検証用に用意した infoLog オブジェクトに記録しています。

なお、Answer クラスによるコールバックは、モックオブジェクトに対しても利用できます。

リスト11.31 ロガーを扱うクラス

```
public class SpyExample {

    Logger logger = Logger.getLogger(SpyExample.class.getName());

    public void doSomething() {
        logger.info("doSomething");
    }
}
```

リスト11.32 ロガーのスパイオブジェクトの利用

```
@Test
public void Mockitoのspyを使ったテスト() throws Exception {
    // SetUp
    SpyExample sut = new SpyExample();
    Logger spy = spy(sut.logger);
    final StringBuilder infoLog = new StringBuilder();
    doAnswer(new Answer<Void>() {

        @Override
        public Void answer(InvocationOnMock invocation) throws Throwable {
            infoLog.append(invocation.getArguments()[0]);
            invocation.callRealMethod();
            return null;
        }
    }).when(spy).info(anyString());
    sut.logger = spy;
    // Exercise
    sut.doSomething();
    // Verify
    assertThat(infoLog.toString(), is("doSomething"));
}
```

第12章

データベースのテスト テストコードで外部システムを制御する

ソフトウェア開発、特にWebアプリケーション開発において、データベースを利用しない開発はほとんどありません。データベースは、開発対象のシステムにとって外部システムのひとつであり、JDBCやSQLなどを介して利用します。このような外部システムとのインターフェースが含まれる場合、ユニットテストは複雑となります。

この章では、軽量データベース「H2 Database」と、ユニットテストをサポートする「DbUnit」を使い、データベースを扱うユニットテストのポイントを解説します。

11
12
13
14

12.1 データベースに依存するユニットテスト

データベースは複雑な状態を持つ外部システムです。このため、データベースに依存するクラスは単純なクラスのようにテストできません。はじめに、データベースを扱うユニットテストの特徴とテストの方針について解説します。

データベースを扱うソフトウェアの設計

データベースを扱うクラスのユニットテストを行ううえで最も重要なことは、クラス設計です。データベースに直接アクセスするクラスを明確に分離した設計にしなければなりません。

もし、データベースにアクセスするコードがプログラムに点在していたならば、それらのクラスはすべてデータベースに依存するクラスになってしまいます。

● Web三層構造アーキテクチャ

Web三層構造アーキテクチャは、データベースにアクセスするクラスを明確に分離するための代表的な設計です。Web三層構造アーキテクチャでは、データベースに直接アクセスするパーシステンス層、ビジネスロジックを実装するサービス層、入力値の検証や画面への表示を行うプレゼンテーション層の3層に分けてアプリケーションを設計します。したがって、データベースを直接扱うのは、パーシステンス層だけになります。

しかしながら、サービス層やプレゼンテーション層ではパーシステンス層のモジュールを利用するため、間接的にはデータベースへ依存しています。ただし、サービス層やプレゼンテーション層のユニットテストでデータベースへの依存を完全になくすために、パーシステンス層のオブジェクトをスタブオブジェクトに置き換えることができます。

● パーシステンス層のスタブによる置き換え

スタブオブジェクトを用いてパーシステンス層ごとデータベースへの依存をなくすことにはメリットもありますが、デメリットもあります。

スタブオブジェクトで置き換えるメリットは、データベースに依存しないため実行速度が速いこと、パーシステンス層のモジュールが未完成な状態でもインターフェースさえ定義されていればサービス層の実装を進められることなどです。このため、スローテスト問題を回避することにもつながり、大規模開発における開発範囲の分担のためにも活用できます。

一方、スタブオブジェクトを利用すると、データベースを使っていないと発生しない問題を見逃す可能性があります。

● プロダクション環境とユニットテスト

スタブオブジェクトを使わず、データベースを利用するテストは、プロダクション環境に近い環境で行われると考えられます。このため、データベース固有の問題などを早期に発見できます。同じバージョンのデータベースを使用し、スキーマ定義を統一すれば、よりプロダクション環境に近づけることができます。

しかしながら、データベースを各テスト環境にセットアップしなければなりません。また、データベースの種類によってはライセンスや動作環境

の問題で、各環境へのセットアップが困難な場合もあります^{注1}。そして、データベースへのデータ投入が初期化処理として発生するため、スローテスト問題を招く可能性があります。

したがって、プロジェクトの規模や特徴などを考慮して、実データベースを使ってエンドトゥエンドでテストするか、スタブで置き換えるかを選択する必要があります。しかしながら、ユニットテストを行ううえで、最も重要なことは問題の早期発見とフィードバックです。テストで使用するデータも含め、プロダクションに近い環境を早期に用意する方針を先に検

注1 組込みや無償で利用できるデータベースで代替する場合もあります。

11
12
13
14

Column

代替データベースを利用したテスト

データベースを扱うユニットテストを行う場合、プログラマはそれぞれ専用のデータベースを用意するべきです。なぜならば、1つのデータベースに対し、同時に複数のプログラマがユニットテストを実行すると、テストがお互いに影響するからです。原則としてテストを実行する環境ごとにデータベース、または専用のスキーマを用意してください。

プロダクション環境で有償のデータベースや特定の環境でしか動作しないデータベースを扱う場合、開発環境に同じものをセットアップすることは難しくなります。そのような場合は、無償で利用できる軽量データベースで代用することもできます。

軽量データベースとしては、組込みでも動作するH2 Databaseやファイルベースで扱いが簡単なSQLite^{注a}などがあります。

しかし、プロダクション環境で使うデータベースとは異なるため、SQLの方言やデータの型の差異があることに注意して使わなければなりません。そのような場合は、データベース間の差異をHibernate^{注b}やDoma^{注c}といったO/Rマッピングフレームワークを使うことで吸収できます。

いずれにせよ、プロダクション環境で利用するデータベースと同じものではないため、プロダクション環境で予想外の問題が起きる可能性がある点はリスクとして考慮すべきです。

注a <http://www.sqlite.org/>

注b <http://www.hibernate.org/>

注c <http://doma.seasar.org/>

討するほうがよいでしょう。

データベースの状態とユニットテスト

データベースに依存するクラスはデータベースに対して参照または更新を行います。このとき、常に同じ結果が参照または更新されなければ、ユニットテストが不安定になります。したがって、データベースの状態が正しく初期化されていることが重要です。

● 参照系のテスト

テスト対象クラスが参照系の処理しか行わないならば、それほど難しくありません。参照系処理のユニットテストでは、初期化処理でデータベースを特定の状態に初期化します。すなわち、関連するテーブルのレコードをすべて削除し、前提条件となるレコードを挿入します。テストケースごとにデータベースを初期化することで、テスト対象クラスの参照系の処理は、予測可能な結果を返すことができます。

ただし、データが複数である場合は、データの順番について考慮しなければなりません。ソート順などを指定することで特定の順序であることを期待しテストする場合もあれば、順序に依存せずに集合として期待される結果となることをテストする場合もあります。

● 更新系のテスト

一方、テスト対象クラスが更新系の処理を行う場合は、参照系の処理の場合に比べて複雑です。なぜならば、データベースへの操作を行う前の状態と行ったあとの状態が異なるからです。更新系の処理を行ったあとに、データベースが期待される状態になっているかを検証しなければなりません。

また、データベースの状態が、常に同じ状態になるとは限りません。たとえば、レコードの更新時などに自動的に設定される「最終更新日時」や自動で採番される「ID」など、期待値が予測できないカラムがあります。このため、一部のカラムを除外して検証するなどの工夫も必要です。

12.2 ユニットテストの自動化とH2 Database

データベースを扱うクラスのユニットテストは、データベースサーバが起動していなければ実行できません。このため、ローカル環境やテスト環境でデータベースサーバをあらかじめ起動させておきます。もし、データベースサーバの起動や停止をテストの一部として組み込むことができれば、テスト環境や開発環境にプロジェクトを展開するだけで、ユニットテストを完全に自動化できます。

このようなデータベースサーバのセットアップや起動の自動化は、OracleやPostgreSQLなどの本格的なデータベースサーバで行なうことは難しいでしょう。しかしながら、軽量で組込み用途でも利用できるH2 Database^{注2}を利用すれば、簡単に実現できます^{注3}。

たとえば、開発環境やテスト環境ではH2 Databaseを利用し、ステージング環境^{注4}やプロダクション環境ではPostgreSQLを利用するといった開発もできます。

11
12
13
14

H2 Databaseサーバの起動／停止を行なうルール

データベースサーバのセットアップや起動の自動化は、第9章で紹介したルールを使うことで簡単に実現できます。

リスト12.1は、H2 Databaseサーバを起動／停止するルールです。H2DatabaseServerクラスは、ExternalResourceクラスのサブクラスとし、beforeメソッドでorg.h2.tools.ServerクラスのcreateTcpServerメソッドを使い起動処理を、afterメソッドで同shutdownメソッドを使い停止処理をそれぞれオーバーライドしています。

このルールを次のように宣言すれば、テストの実行前にデータベースサーバが起動し、実行後にデータベースサーバが停止します。

```
@Rule
public H2DatabaseServer server = new H2DatabaseServer("h2", "db", "ut");
```

注2 <http://www.h2database.com/>

注3 H2 Databaseの導入手順については付録Cを参照してください。

注4 プロダクション環境(本番環境)とほとんど同じ構成でアプリケーションを実行する環境。

もし、テストケースごとに起動と停止を行いたくないのであれば、次のように ClassRule アノテーションを使い、テストクラスごとに起動と停止を行うこともできます。

```
@ClassRule
public static H2DatabaseServer SERVER
    = new H2DatabaseServer("h2", "db", "ut");
```

このようにデータベースの起動や終了もテストケース内で自動化すれば、テストの可搬性が高まります。特に複数の実行環境でユニットテストを行いたい場合は有効な方法です。

12.3 DbUnitによるデータベースのテスト

リスト 12.2 は、users テーブルからユーザ名の一覧を取得するメソッドと、ユーザを追加するメソッドを持つ Dao クラスです。本節では、このクラスのユニットテストを作成しながら、データベースのテストをサポートする DbUnit を紹介します。

なお、ここで解説する UserDao のプロダクションコードは JDBC と SQL を直接利用した必要最低限の実装です。

DbUnitとは？

DbUnit^{注5}は、データベースに依存するクラスのユニットテストを行うための JUnit の拡張ライブラリです。DbUnit では、主にデータベースの状態をセットアップするための機能と、データベースの状態を検証するための機能を提供しています。

DbUnitのJUnit 4対応

DbUnit はプロダクトとして歴史も長く、安定して利用できますが、JUnit 3

^{注5} <http://www.dbunit.org/>

リスト12.1 H2 Databaseサーバを起動／停止するルール

```

public class H2DatabaseServer extends ExternalResource {

    private final String baseDir;
    private final String dbName;
    private final String schemaName;
    private Server server = null;

    public H2DatabaseServer(
        String baseDir, String dbName, String schemaName) {
        this.baseDir = baseDir;
        this.dbName = dbName;
        this.schemaName = schemaName;
    }

    @Override
    protected void before() throws Throwable {
        // DBサーバの起動
        server = Server.createTcpServer("-baseDir", baseDir);
        server.start();
        // スキーマの設定
        Properties props = new Properties();
        props.setProperty("user", "sa");
        props.setProperty("password", "");
        String url = "jdbc:h2:" + server.getURL() + "/" + dbName;
        Connection conn = org.h2.Driver.load().connect(url, props);
        try {
            conn.createStatement()
                .execute("CREATE SCHEMA IF NOT EXISTS " + schemaName);
        } finally {
            JdbcUtils.closeSilently(conn);
        }
    }

    @Override
    protected void after() {
        // DBサーバの停止
        server.shutdown();
    }
}

```

11
12
13
14

のころに開発されたライブラリです。本書で扱っているJUnit 4のフォーマットに混ぜて使うこともできますが、異なるフォーマットが混在してしまい、混乱を招くでしょう。そこで、本書ではDbUnitをJUnit 4のルール^{注6}として利用するためのラッパークラスを作成します。

● DbUnitのルールクラスの作成

リスト12.3は、DbUnitが提供するorg.dbunit.AbstractDatabaseTesterクラスを拡張し、JUnitのルールとして利用できるようにしたクラスです。

コンストラクタ(リスト12.3①)では、JDBCドライバのクラス名(driverClass)、データベースの接続先(connectionUrl)、ユーザ名とパスワード(username、password)などを引数として受け取ります。また、JDBC

注6 ⇒ルール(p.141)

リスト12.2 ユーザテーブルにアクセスするためのDAOクラス

```
public class UserDao {

    public List<String> getList() throws SQLException {
        ResultSet rs = createStatement()
            .executeQuery("SELECT name FROM users");
        LinkedList<String> result = new LinkedList<String>();
        while (rs.next()) {
            result.add(rs.getString(1));
        }
        return result;
    }

    public void insert(String username) throws SQLException {
        String sql = "INSERT INTO users(name) VALUES('" + username + "')";
        createStatement().executeUpdate(sql);
    }

    private Statement createStatement() throws SQLException {
        String url = "jdbc:h2:tcp://localhost/db;SCHEMA=ut";
        Connection connection = DriverManager.getConnection(url, "sa", "");
        return connection.createStatement();
    }
}
```

ドライバのロードを行っています。

DbUnitTester クラスはルールとして利用できるように TestRule インタフェースを実装し、apply メソッドで拡張した Statement オブジェクトを返します(同❷)。拡張した Statement オブジェクトでは、いくつかの前処理と後処理を行います。

before メソッド(同❸)は、DbUnitTester のサブクラスでオーバーライド可能なメソッドです。before メソッドをオーバーライドすることで、各テストクラスで固有の前処理を実行できます。

setDataSet メソッド(同❹)は AbstractDatabaseTester クラスに定義されたメソッドです。このメソッドは、データベースのテストを行うために必要なデータセット(テーブルごとのレコード一覧)を設定します。

onSetup メソッド(同❺)も AbstractDatabaseTester クラスに定義されたメソッドです。setDataSet メソッドで設定したデータセットを使いデータベースの状態を更新します。このとき、デフォルトの動作はクリーンインサートです。すなわち、対象となるテーブルのレコードをすべて削除し、データセットで定義されたレコードを追加します。

これらの前処理を実行したあと、base オブジェクトの evaluate メソッドが呼び出され、テストが実行されます(同❻)。

テストが実行されたあと、finally で後処理が行われます。after メソッド(同❼)は、before メソッドと同様に、DbUnitTester のサブクラスでオーバーライド可能な後処理を実行します。onTearDown メソッド(同❽)は、onSetup メソッドと同様に、後処理を行う AbstractDatabaseTester クラスに定義されたメソッドです。

リスト12.3 DbUnitを利用するためのRuleクラス

```
public abstract class DbUnitTest
    extends AbstractDatabaseTester implements TestRule {
    private final String connectionUrl;
    private final String username;
    private final String password;

    public DbUnitTest(String driverClass, String connectionUrl) {
        this(driverClass, connectionUrl, null, null);
```

11
12
13
14

```

}

public DbUnitTester(
    String driverClass, String connectionUrl,
    String username, String password) {
    this(driverClass, connectionUrl, username, password, null);
}

public DbUnitTester(
    String driverClass, String connectionUrl,
    String username, String password, String schema) { ❶
    super(schema);
    this.connectionUrl = connectionUrl;
    this.username = username;
    this.password = password;
    assertNotNullNorEmpty("driverClass", driverClass);
    try {
        // JDBC ドライバのロード
        Class.forName(driverClass);
    } catch (ClassNotFoundException e) {
        throw new AssertionException(e);
    }
}

@Override
public IDatabaseConnection getConnection() throws Exception {
    Connection conn = null;
    if (username == null & password == null) {
        conn = DriverManager.getConnection(connectionUrl);
    } else {
        conn = DriverManager
            .getConnection(connectionUrl, username, password);
    }
    DatabaseConnection dbConnection = new DatabaseConnection(
        conn, getSchema());
    DatabaseConfig config = dbConnection.getConfig();
    config.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
        new H2DataTypeFactory());
    return dbConnection;
}
}

```

```

protected void executeQuery(String sql) throws Exception {
    Connection conn = getConnection().getConnection();
    conn.createStatement().execute(sql);
    conn.commit();
    conn.close();
}

protected void before() throws Exception {
}

protected void after() throws Exception {
}

abstract protected IDataSet createDataSet() throws Exception;

@Override
public Statement apply(final Statement base, Description description) { ②

    @Override
    public void evaluate() throws Throwable {
        before(); ③
        setDataSet(createDataSet()); ④
        onSetup(); ⑤
        try {
            base.evaluate(); ⑥
        } finally {
            try {
                after(); ⑦
            } finally {
                onTearDown(); ⑧
            }
        }
    }
};

}

```

11
12
13
14

● DbUnitのルールクラスの利用

DbUnitを利用するときにはいくつかの前処理と後処理が必要です。JUnit 3のスタイルでは、リスト 12.4 のように、それらの処理をテストクラスに共通のスーパークラスを強いることで実現してきました。

JUnit 4では、それらの処理はルールを使うことで簡単に拡張でき、テストクラスが宣言的でシンプルなテストコードとなります。

DbUnitTester をテストクラスで使うには、リスト 12.5 のように宣言します。

DbUnitTester オブジェクトは匿名クラスとして定義し、before メソッドと createDataSet メソッドをオーバーライドしています。before メソッドではスキーマと users テーブルの初期化を行い、createDataSet メソッドでは XML で定義された users テーブルのデータセットを読み込んで返しています（データセットに関しては次の節で解説します）。

このようにルールを利用すれば、DbUnit を JUnit の拡張として宣言的に使うことができます。

リスト 12.4 DbUnit の利用 (JUnit 3スタイル)

```
public class UserDaoTest extends DBTestCase {
    public UserDaoTest(String name)
        super(name);
    System.setProperty(DBUNIT_DRIVER_CLASS, "org.h2.Driver");
    System.setProperty(DBUNIT_CONNECTION_URL,
                      "jdbc:h2:tcp://localhost/db;SCHEMA=ut");
    System.setProperty(DBUNIT_USERNAME, "sa");
    System.setProperty(DBUNIT_PASSWORD, "");
    System.setProperty(DBUNIT_SCHEMA, "ut");
}
@Before
public void setUp() throws Exception {
    super.setUp();
}
// その他の処理は省略
}
```

リスト12.5 DbUnitルールの利用

```

@Rule
public DbUnitTester tester = new DbUnitTester(
    "org.h2.Driver", "jdbc:h2:tcp://localhost/db;SCHEMA=ut",
    "sa", "", "ut") {

@Override
protected void before() throws Exception {
    executeQuery("DROP TABLE IF EXISTS users");
    executeQuery(
        "CREATE TABLE users(id INT AUTO_INCREMENT, name VARCHAR(255))");
}

@Override
protected org.dbunit.dataset.IDataSet createDataSet()
    throws Exception {
    return new FlatXmlDataSetBuilder()
        .build(getClass().getResourceAsStream("fixtures.xml"));
}
};
```

11
12
13
14**データベースのセットアップ**

データベースを扱うユニットテストでは、データベースのセットアップが最も重要です。データベースの状態が正しくセットアップされていなければ、ユニットテストが正しく実行されないだけではなく、プロダクションコードに不具合があるにもかかわらずテストが成功してしまう可能性もあるでしょう。

●DbUnitのデータセット

DbUnitではデータセットというしくみを使い、データベースのセットアップを行います。データセットとは、データベース上のデータ、すなわちテーブルと含まれるレコードをJavaのオブジェクトとして抽象化したものです。データセットは、org.dbunit.dataset.IDataSetインターフェースやorg.dbunit.dataset.ITableインターフェースで構成されます。

DbUnitを利用する場合、初期化処理でデータセットをクリーンインサートし、後処理ではデータベースにアクセスしません。これは、シンプルさ

を保ちつつ、パフォーマンスを考慮した設計です。

はじめにデータセットに含まれるテーブルのレコードがすべて削除されます。これにより予期せぬデータが残っている可能性がなくなります。後処理でレコードの削除を行っても同様の効果を得ることができます、ユニットテストの開始時に行うほうが確実です。また、後処理でデータベースをリセットしないため、ユニットテストが失敗したときにデータベースの状態を確認できます。

初期化処理の最後では、データセットに含まれるレコードがインサートされます。

● データセットの外部定義

データセットはJavaのオブジェクトです。テストコードの中で構築することができますが、冗長で読みにくく、メンテナンスしにくいコードとなりがちです。このため、通常はデータセットを外部ファイルに定義して読み込みます。

前節で作成したDbUnitTesterでは、次のようにcreateDataSetメソッドでfixtures.xmlに定義されたデータセットを読み込みました。

```
protected org.dbunit.dataset.IDataSet createDataSet() throws Exception {
    return new FlatXmlDataSetBuilder()
        .build(getClass().getResourceAsStream("fixtures.xml"));
}
```

fixtures.xmlは、DbUnitが提供するFlat XML フォーマットのファイルです(リスト12.6)。要素名がテーブル名となり、属性と値でカラム名と値を定義します。

DbUnitでは、Flat XMLのほかにも XML、CSV、Excelなどのフォーマットに対応しています。各フォーマットには一長一短がありますが、データベースは複数のテーブルと行と列から構成されるため、Excelがもっとも適したフォーマットと言えます。しかしながら、テキストエディタなどで確認しにくいことや、バージョン管理システムで管理しにくいことが難点です。

もし、YAMLやJSONなどテキスト形式で構造化可能な別のフォーマットを好むならば、IDataSetの実装クラスとローダークラスを作成して利用することもできます。

データベースのアサーション

データベースの更新を扱うユニットテストでは、データベースの状態を検証することが必要です。この検証を行うためには、期待されるデータベースのデータと実データベースに保持されているデータを取得し、比較検証しなければなりません。

DbUnitでデータを比較検証する場合、データベースのセットアップで利用したデータセットを利用します。期待されるデータセットは、初期化処理と同様に、外部ファイルに定義して読み込みます。実データベースに保持されているデータは、IDatabaseConnectionインターフェースに定義されているcreateDataSetメソッドやcreateTableメソッドで取得できます。

それぞれの方法で取得したデータセットは、org.dbunit.AssertionクラスのassertEqualsメソッドで比較検証します。

リスト12.7は、usersテーブルのアサーションを行うサンプルです。

● JUnit 4スタイルでのアサーション

リスト12.7で使用しているorg.dbunit.AssertionクラスのassertEqualsメソッドは、JUnit 3スタイルであるため、第1引数が期待値であり、第2引数が実測値です。

リスト12.6 ユーザ情報の定義されたXMLファイル

```
<dataset>
    <users id="1" name="Ichiro" />
    <users id="2" name="Jiro" />
</dataset>
```

リスト12.7 usersテーブルのアサーション

```
// 期待されるusersテーブルのデータセット
InputStream expectedIn = getClass().getResourceAsStream("expected.xml");
ITable expected = new FlatXmlDataSetBuilder()
    .build(expectedIn).getTable("users");
// 実データベースが保持するusersテーブルのデータセット
IDataSet actual = tester.getConnection().getTable("users");
// アサーション
org.dbunit.Assertion.assertEquals(expected, actual);
```

JUnit 4 のアサーションと同様に、`assertThat` メソッドを利用するには、リスト 12.8 のようなカスタム Matcher クラスを用意するとよいでしょう。ITableMatcher クラスでは、BaseMatcher クラスのサブクラスである TypeSafeMatcher クラスを利用しました。また、内部実装では、org.dbunit.Assertion クラスの `assertEquals` メソッドを使用しています。

`tableOf` メソッドを static インポートして使用すれば、`assertThat` を使ってリスト 12.9 のように記述できます。

コンテキストによるテストケースの整理

データベースを扱うユニットテストでは、同じメソッドに対してデータベースの状態を変えてテストを実行します。たとえば、一覧を取得するメソッドであれば、テーブルにレコードが 0 件の場合と 2 件の場合といった状況でテストを行うでしょう。そこで、第 6 章で紹介した Enclosed テストランナーを使うと効果的です^{注7}。

リスト 12.10 は、Enclosed テストランナーを使った UserDao のテストクラスです。データベースに依存するテストでは、コンテキストごとにテストケースをまとめると、見通しの良いテストクラスが作成できます。

DbUnit のそのほかの機能

本書では紹介しきれませんが、DbUnit にはさまざまな機能があります。たとえば、アサーションで扱いにくい最終更新日やオートインクリメントされるカラムを比較時に除外する機能や、データベースから XML ファイルなどを出力する機能です。

詳細は、公式サイトのドキュメントなどを参照してください。

注7 ➔ Enclosed によるテストクラスの構造化(p.96)

リスト12.8 ITableを比較検証するMatcher

```

public class ITableMatcher extends TypeSafeMatcher<ITable> {

    public static Matcher<ITable> tableOf(ITable expected) {
        return new ITableMatcher(expected);
    }

    private final ITable expected;
    String assertionMsg = null;

    ITableMatcher(ITable expected) {
        this.expected = expected;
    }

    @Override
    public boolean matchesSafely(ITable actual) {
        try {
            org.dbunit.Assertion.assertEquals(expected, actual);
        } catch (DatabaseUnitException e) {
            throw new DatabaseUnitRuntimeException(e);
        } catch (AssertionError e) {
            assertionMsg = e.getMessage();
            return false;
        }
        return true;
    }

    @Override
    public void describeTo(Description desc) {
        desc.appendValue(expected);
        if (assertionMsg == null) return;
        desc.appendText("\n >>>").appendText(assertionMsg);
    }
}

```

11
12
13
14**リスト12.9** assertThatによるITableの比較検証

```

ITable actual = tester.getConnection().createDataSet().getTable("users");
InputStream expectedIn = getClass().getResourceAsStream("expected.xml");
ITable expected = new FlatXmlDataSetBuilder()
    .build(expectedIn).getTable("users");
assertThat(actual, is(tableOf(expected)));

```

リスト12.10 コンテキストごとに整理した UserDaoTest

```

@RunWith(Enclosed.class)
public class UserDaoTest {

    public static class usersに2件のレコードがある場合 {
        @ClassRule
        public static H2DatabaseServer server = new H2UtDatabaseServer();
        @Rule
        public DbUnitTester tester
            = new UserDaoDbUnitTester("fixtures.xml");
        UserDao sut;

        @Before
        public void setUp() throws Exception {
            this.sut = new UserDao();
        }

        @Test
        public void getListで2件取得できる事() throws Exception {
            // Exercise
            List<String> actual = sut.getList();
            // Verify
            assertThat(actual, is(notNullValue()));
            assertThat(actual.size(), is(2));
            assertThat(actual.get(0), is("Ichiro"));
            assertThat(actual.get(1), is("Jiro"));
        }

        @Test
        public void insertで1件追加できる() throws Exception {
            // Exercise
            sut.insert("Sabrou");
            // Verify
            ITable actual = tester.getConnection()
                .createDataSet().getTable("users");
            InputStream expectedIn = getClass()
                .getResourceAsStream("expected.xml");
            ITable expected = new FlatXmlDataSetBuilder()
                .build(expectedIn).getTable("users");
            assertThat(actual, is(tableOf(expected)));
        }
    }
}

```

```

}

public static class usersに0件のレコードがある場合 {
    @ClassRule
    public static H2DatabaseServer server = new H2UtDatabaseServer();
    @Rule
    public DbUnitTester tester
        = new UserDaoDbUnitTester("zero_fixtures.xml");

    UserDao sut;

    @Before
    public void setUp() throws Exception {
        this.sut = new UserDao();
    }

    @Test
    public void getListで0件取得できる事() throws Exception {
        // Exercise
        List<String> actual = sut.getList();
        // Verify
        assertThat(actual, is(notNullValue()));
        assertThat(actual.size(), is(0));
    }

    @Test
    public void insertで1件追加できる() throws Exception {
        // Exercise
        sut.insert("Sirou");
        // Verify
        ITable actual = tester.getConnection()
            .createDataSet().getTable("users");
        InputStream expectedIn = getClass()
            .getResourceAsStream("zero_expected.xml");
        ITable expected = new FlatXmlDataSetBuilder()
            .build(expectedIn).getTable("users");
        assertThat(actual, is(tableOf(expected)));
    }
}

static class H2UtDatabaseServer extends H2DatabaseServer {
    public H2UtDatabaseServer() {
}

```

11
12
13
14

```
        super("h2", "db", "ut");
    }
}

static class UserDaoDbUnitTeser extends DbUnitTester {
    private final String fixture;
    public UserDaoDbUnitTeser(String fixture) {
        super("org.h2.Driver", "jdbc:h2:tcp://localhost/db;SCHEMA=ut",
              "sa", "", "ut");
        this.fixture = fixture;
    }

    @Override
    protected void before() throws Exception {
        executeQuery("DROP TABLE IF EXISTS users");
        executeQuery(
            "CREATE TABLE users(id INT AUTO_INCREMENT, name VARCHAR(64))"
        );
    }

    @Override
    protected org.dbunit.dataset.IDataSet createDataSet()
        throws Exception {
        return new FlatXmlDataSetBuilder()
            .build(getClass().getResourceAsStream(fixture));
    }
}
```

第13章

Androidのテスト UIとロジックを分けてテストする

近年、Androidアプリケーションの開発に対するニーズが高まっています。多くのAndroidアプリケーションは、一定の品質を求められるB to C (*Business to Consumer / Customer*)を主なターゲットとしています。しかしながら、AndroidアプリケーションのようなGUIアプリケーションのテストには多くの課題があります。その主な要因は、多くのクラスが複雑な依存関係を持った実装になっていることです。

これらの課題を解決するには、テストしやすいように設計することが重要です。クラスの責務を明確にし、複雑な依存関係を減らし、テストabilitiを高めます。

本章ではAndroidアプリケーションの開発でユニットテストを行うために必要な概念や、具体的なテスト方法について解説します。

13.1 GUIアプリケーションの設計

多くのアプリケーションは、ユーザの入力に対して、何らかの出力を返すといった相互作用で利用されます。このように、アプリケーションがユーザと対話するしくみはユーザインターフェースと呼ばれ、大きくCLI (*Command Line Interface*)^{注1}とGUI (*Graphical User Interface*)とに分類されます。

CLIは、コンソールからコマンド入力を行うシンプルなユーザインターフェースです。しかしながら、近年では一般ユーザにとってCLIは馴染みのないインターフェースです。ほとんどのパーソナルコンピュータやスマートフォンデバイスのアプリケーションでは、アイコンやメニューを選択式に操作するGUIが提供されています。

注1 CUI (*Character User Interface*)とも呼ばれます。

GUIアプリケーション開発のポイント

GUIアプリケーション開発では、重要なポイントが2つあります。

1つ目のポイントは、デザイン(見た目)やユーザビリティ(使いやすさ)です。同じ機能を持つアプリケーションであれば、見た目がよく、使いやすいアプリケーションの満足度が高いことは言うまでもありません。特にユーザビリティは重要です。マニュアルを参照しなくとも直感的に操作できるGUIが最良と言われています。ただし、デザインやユーザビリティを高めるためには高度な専門スキルを必要とします。

2つ目のポイントは、変更に対して強い設計とすることです。GUIアプリケーションでは、見た目や操作性に対して、常に変更要求があります。ユーザのフィードバックや新機能の追加の影響から、GUIは常に改善を求められます。

設計はユニットテストと強い関係があります。変更に強い設計は、クラスの責務が明確で疎結合なクラス設計であり、自然とテストアビリティが高まります。また、変更を繰り返して何度もリリースすることになるため、十分にユニットテストが行われていることは安心につながります。

MVCパターン

MVCパターンはGUIアプリケーションを、モデル(*model*)、ビュー(*view*)、コントローラ(*controller*)に分解して設計するアーキテクチャです。MVCパターンを適用することで、各クラスの責務が明確になり、変更に強いGUIアプリケーションを設計できます。

● モデル

MVCパターンにおけるモデルとは、表示に関連しないすべての部分です。モデルは、アプリケーションのデータと処理で構成されます。アプリケーションでは、ユーザ情報や画像情報などのデータ(情報)を扱い、画面を介してユーザに表示されます。

しかしながら、画面でどのように見えるかはデータの本質ではありません。MVCパターンでは、データそのものはモデルとして、表示(ビュー)と



分けて設計します。また、データそのものだけではなく、表示に関連しない処理はすべてモデルとして扱います。

ピュー

MVCパターンにおけるビューとは、表示に関連する部分です。ビューは、画面を構成するすべてのコンポーネントで構成され、どのようにデータ(モデル)を表示するかについて責任を持ちます。

GUIアプリケーションでは、テキストエリアやリストなどのコンポーネントを介してユーザに情報を表示します。また、ユーザは、ボタンやテキストボックスなどのコンポーネントを介してアプリケーションに処理を実行させます。すなわち、ビューはアプリケーションとユーザとのインターフェースを担当します。ただし、ボタンが押されたときに処理を行う部分は後述のコントローラです。

●ヨントヨーラ

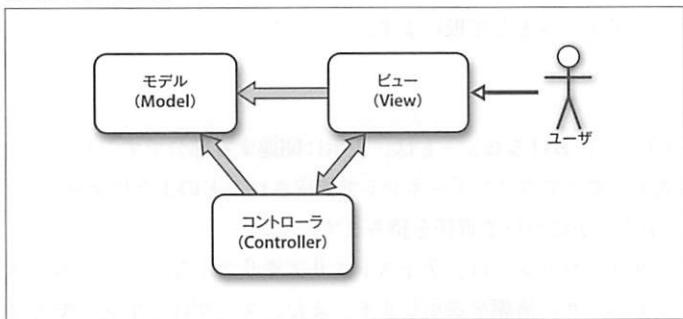
コントローラは、モデルとビューとの接着剤です。モデルだけではユーザーの操作を受け、処理を行うことはできません。また、ビューだけではユーザーに情報を見せることしかできません。

ユーザは、ビューを操作することでアプリケーションに入力を行います。アプリケーションでは、その入力に対応する処理を行い、結果をビューに反映します。コントローラは、これらの入力イベントを受け付け、モデルに処理をさせ、ビューに反映させる責任を持ちます。

3 3 3 3 3 3 3 3 3 3 3 3 3

MVCパターンでアプリケーションを設計するときには、モデルが独立していることが特に重要です。モデルはビューやコントローラを参照しないように設計します。一方、ビューとコントローラは密接な関係となります。ビューは、モデルとコントローラの両方を参照します。コントローラは、モデルとビューの両方を参照します(図13.1)。

図13.1 MVCパターン



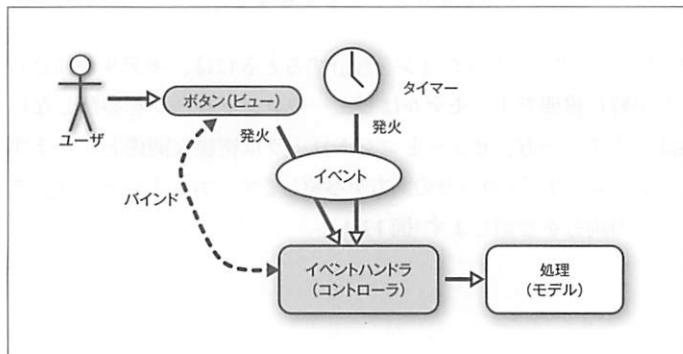
イベント駆動

GUI アプリケーションでは、ボタンやテキストエリアなどのコンポーネントにユーザが何らかの操作を行うと、その操作に対応する処理が実行されます。また、時計のアラームのように特定の時間になると実行される処理や、メッセージが到着すると実行される処理などのように、特定の条件を満たしたときに対応する処理が実行される場合もあります。

このように、ユーザの操作やそのほかの条件を契機として処理が実行されるしくみはイベント駆動(event driven)と呼ばれます(図13.2)。

GUI アプリケーションはイベント駆動のアプリケーションです。イベントが発生するまでは待機し、特定のイベント(ボタンがクリックされる、な

図13.2 イベント駆動の概念



ど)が発生したならば、対応する処理が実行されます。プログラムでは、ボタンなどのコンポーネントと対応する処理を関連付けるコードを記述します。このような関連付けは、バインドすると呼ばれます。

● イベントハンドラ／イベントリスナー

ボタンなどにバインドされるクラスやメソッドは、イベントハンドラ(*event handler*)やイベントリスナー(*event listener*)と呼ばれます。

Javaのプログラムでは、イベントハンドラをオブジェクトとしてコンポーネントオブジェクトに設定する形で記述します。この設定は、GUIの初期化のときに行われます。

イベントハンドラを実行するのは、GUIの基盤となるフレームワークです。Androidなどのフレームワークは、クリックなどの外部イベントを監視し、イベントを認識したならば、設定されているイベントハンドラを実行します。このイベントハンドラの実行は、発火する(*fire*)と呼ばれます。

イベントハンドラが発火したあとは、対応する処理が行われ、結果に応じてGUIが更新されます。なお、イベントの処理中も、フレームワークは常にイベントの監視を行っています。

MVCモデルと対応させると、ボタンなどのコンポーネントがビュー、処理がモデル、イベントハンドラがコントローラです。

11
12
13
14

GUIアプリケーションのスレッドモデル

GUIアプリケーションでは、GUIの描画処理とイベントの処理をシングルスレッド^{注2}で行います。これは、GUIアプリケーションを設計するうえで重要なことです。なお、このスレッドはGUIスレッドと呼ばれます。

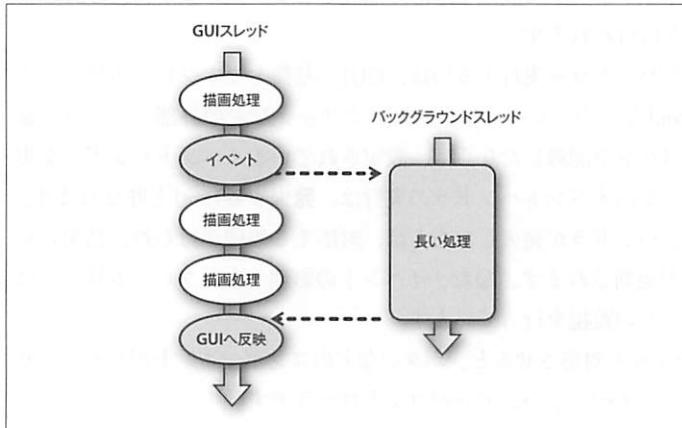
GUIの描画処理とイベントの処理がシングルスレッドで行われるため、イベントの処理に長い時間がかかると、GUIの再描画を行うことができません。アプリケーションは固まったような状態になります。このため、イベントによって実行される処理は、制御可能な限り早く呼び出し元に返す必要があります。

^{注2} スレッドとはプログラムの実行単位のことと、1つのスレッドで処理される場合はシングルスレッド、複数のスレッドが並列に処理される場合はマルチスレッドと呼びます。

しかしながら、WebサーバにHTTPリクエストを行うことで情報を取得するなど、アプリケーションでは長い処理を行う必要もあります。そのような場合は、バックグラウンドスレッド(ワーカスレッド)を作って実行させます(図13.3)。

バックグラウンドスレッドで処理が完了したならば、結果をGUIに反映しなければなりません。このとき、バックグラウンドスレッドからGUIの更新などを行うと、シングルスレッドモデルに違反することになります。シ

図13.3 GUIスレッドとバックグラウンドスレッド



Column

なぜGUIはシングルスレッドモデルなのか?

現在、GUIフレームワークでは、ほぼ例外なくシングルスレッドモデルが採用されています。マルチコアで並列処理が当たり前の現在でも、GUIフレームワークでマルチスレッドは採用されていません。これは、GUIフレームワークはシビアなタイミングで動かざるを得ないため、マルチスレッドモデルで実装することが困難だからです。

歴史的にはマルチスレッドで処理しようとしたGUIフレームワークも存在しました。しかしながら、並列処理を行ううえで、データの揮発性、同期、ロックなどの課題を解決できず、実用レベルに至っていません。また、ユーザの画面操作速度に対し、CPUは十分に高速となったため、シングルスレッドでシンプルに処理しても十分なパフォーマンスが得られるようになっています。

これらの理由から、Androidでもシングルスレッドモデルが採用されています。

ングルスレッドモデルに違反した場合、予期せぬ不具合が発生する可能性があります。必ず問題が発生するわけではありませんが、再現性の低い不具合ほど厄介な問題もないでしょう。

バックグラウンドスレッドでの処理のあとにGUIを更新したい場合は、ユーティリティクラスなどを利用してGUIスレッドで行います。

13.2 MVC/パターンによるAndroidアプリケーションの作成

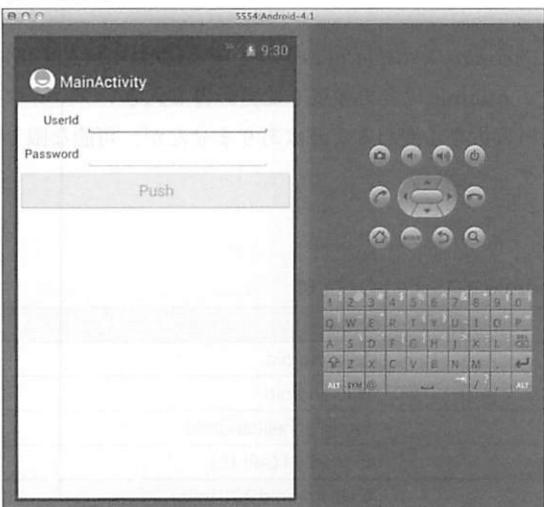
Androidアプリケーションのユニットテストを行うには、テストしやすい設計になっていることが重要です。ここでは、前節で解説したMVCパターンを適用したAndroidアプリケーションとそのテストを作成します。

HelloAndroidの概要

ここでは、UserIdとパスワードの入力フォームと押しボタンがあるだけのシンプルなアプリケーション、「HelloAndroid」を作成します(図13.4)。

HelloAndroidアプリケーションの仕様は次のとおりです。

図13.4 HelloAndroidアプリケーション



11
12
13
14

- ・ユーザIdとパスワードは4文字以上
- ・ユーザIdとパスワードの入力が4文字未満の場合、[Push]ボタンは押すことができない
- ・ユーザIdとパスワードを入力し、[Push]ボタンを押すとユーザ認証が行われる
- ・ユーザ認証が成功すると「ようこそ、XXXさん」というメッセージが表示される
- ・ユーザ認証が失敗すると「ユーザIdまたはパスワードが間違っています」とメッセージが表示される

ユーザ認証に関する仕様はまだ確定していませんが、何らかのWebサービスを使う予定です。

Androidのプロジェクトの作成

はじめに、Eclipseを起動し、Androidのプロジェクトを作成します。Androidの開発環境のセットアップを行っていない方は、付録Dを参照してセットアップしてください。

Androidプロジェクトは、メニューから[ファイル]-[新規]-[プロジェクト...]を選択し、「新規プロジェクト」ダイアログから[Android]-[Android Application Project]を選択して作成します。プロジェクトの設定値は、表13.1を参考に設定してください。その他の設定は特に変更する必要はありません。

なお、Android SDK(*Software Development Kit*)のバージョンは4.1(API 16)で動作確認しています。Androidは変更頻度も変更影響も大きいプロダクトです。設計方針やAPIは大きく変わることはありませんが、可能な限り同じバージョンで試してください。

表13.1 作成するAndroidプロジェクトの設定値

項目名	設定値
Application Name	HelloAndroid
Project Name	hello-android
Package Name	examples.helloandroid
Build SDK	Android 4.1(API 16)
Minimum Required SDK	API 8: Android 2.2(Froyo)

エミュレータの起動

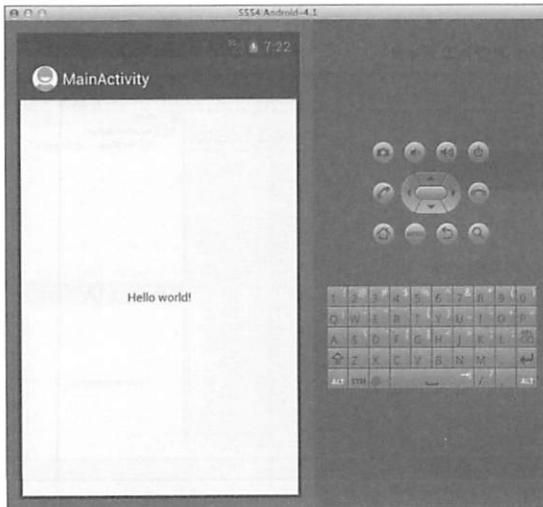
Android SDKでは、Android端末を持っていなくても、エミュレータを使った動作確認を行うことができます。エミュレータを起動するには、AndroidプロジェクトをEclipseから起動するだけです。ただし、あらかじめAVD(*Android Virtual Device*)をセットアップしておく必要があります。AVDのセットアップに関しては、付録Dを参照してください。

プロジェクトにエラーがないことを確認し、プロジェクトのコンテキストメニューなどから[実行]-[Androidアプリケーション]を選択します。最初の起動ではエミュレータの起動には長い時間がかかります^{注3}。一度起動すれば、エミュレータを終了させる必要はありません。

エミュレータが起動するとAndroidのロック解除画面が表示されます。鍵のマークを鍵穴にドラッグしてロックを解除すれば、「HelloAndroid」アプリケーションが起動します(図13.5)。ロックを解除してもアプリケーションが表示されない場合は、メニューを開き、「HelloAndroid」アプリケーションを起動してください。

注3 マシンスペックによっては5分以上かかることがあります。

図13.5 エミュレータで起動したHelloAndroidアプリケーション



なお、ソースコードやレイアウトを修正したあとに、アプリケーションをエミュレータに反映させるとときは、エミュレータを起動したままで再度[実行]-[Androidアプリケーション]を実行します。

レイアウトを作成する

Androidのプロジェクトでは、基本的に画面のレイアウトをXMLで記述します^{注4}。このレイアウトXMLは、EclipseのAndroidプラグインが提供するレイアウトエディタで作成します。

HelloAndroidプロジェクトにある、res/layout/activity_main.xml^{注5}をダブルクリックして開くと、レイアウトエディタの[Graphical Layout]タブが選択された状態で、図13.6のように表示されます。

ここで[activity_main.xml]タブを開くとXMLのソースコードが表示されます。このソースをリスト13.1のように書き換えてください。^{注6} [Graphical Layout]タブに切り替えると、図13.4と同様のレイアウトが表示されます^{注7}。

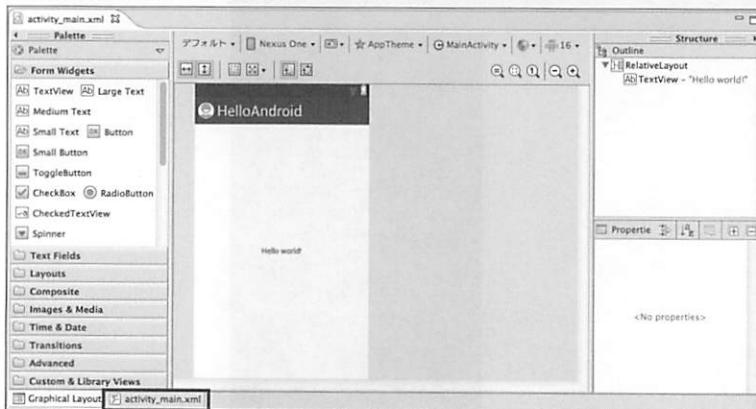
注4 Swingのようプログラムでコンポーネントを配置することもできます。

注5 古いバージョンの場合は、「activity_main.xml」を「main.xml」に置き換えて読み進めてください。

注6 長文ですのでサンプルコードをダウンロードしてコピーすることをお勧めします。

注7 レイアウトエディタの使い方やレイアウトXMLの詳細について詳しく知りたい方は、Androidアプリケーション開発の入門書などをあたってください。

図13.6 Androidのレイアウトエディタ



なお、文字列の国際化関連で警告が表示されますが、本書では無視しても問題ありません。

以上でアプリケーションの画面レイアウトが作成できました。プロジェクトのショートカットメニューから[実行]→[Androidアプリケーション]でアプリケーションを再度実行して、エミュレータで確認してください。

リスト13.1 HelloAndroidのレイアウトXML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <TableRow
            android:id="@+id/tableRowUserId"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >

            <TextView
                android:id="@+id/userIdTextView"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="right"
                android:paddingLeft="@dimen/padding_medium"
                android:paddingRight="@dimen/padding_small"
                android:text="UserId" />

            <EditText
                android:id="@+id/userIdEditText"
                android:layout_height="wrap_content"
                android:ems="10"
                android:inputType="textNoSuggestions" >

                <requestFocus />
            </EditText>
        
```

11
12
13
14

```
</TableRow>

<TableRow
    android:id="@+id/tableRowPassword"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >

    <TextView
        android:id="@+id/passwordTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="right"
        android:paddingLeft="@dimen/padding_medium"
        android:paddingRight="@dimen/padding_small"
        android:text="Password" />

    <EditText
        android:id="@+id/passwordEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPassword" />

</TableRow>
</TableLayout>
<Button
    android:id="@+id/pushButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:enabled="false"
    android:text="Push" />
<TextView
    android:id="@+id/statusTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

アクティビティを実装する

Android アプリケーションでは、アクティビティと呼ばれるベースコン

ポーネントと、その上に配置されるボタンやテキストなどのコンポーネントでビューが構成されます。そして、それらのコンポーネントに、イベントハンドラであるリスナオブジェクトやウォッチャーオブジェクトをコントローラとして設定します。

アクティビティを定義した examples.helloandroid.MainActivity クラスは、初期状態でリスト 13.2 のようになっています。Android のアクティビティでは、onCreate メソッドでイベントなどのセットアップを行います。

● コンポーネントの参照

ボタンやテキストフィールドなどのコンポーネントは、レイアウト用の XML で定義されています。それらの参照は、次のように Activity クラスの findViewById メソッドを使い取得します。

```
Button pushButton = (Button) findViewById(R.id.pushButton);
```

● イベントハンドラの追加

ボタンやテキストフィールドを取得したならば、各コンポーネントにイベントハンドラを設定します。ボタンのクリックイベントであれば View.

リスト 13.2 初期の MainActivity クラス

```
public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

OnClickListener インタフェースの実装クラス、テキストの入力イベントであればTextWatcher インタフェースの実装クラスがイベントハンドラです。

リスト 13.3 は、すべてのイベントハンドラを登録した MainActivity クラスです。EditTextWatcher クラスと PushButtonListener クラスは、どちらも HelloAndroidActivity クラスのインナークラスとして定義されているため、MainActivity クラスのメソッドにアクセスできます。

これでビューとコントローラが分離できました。アプリケーションを再度実行して、アプリケーションの動作を確認してください。

リスト 13.3 イベントハンドラを登録した MainActivity クラス

```
public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getUserIdEditText().addTextChangedListener(new EditTextWatcher());
        getPasswordEditText().addTextChangedListener(new EditTextWatcher());
        getPushButton().setOnClickListener(new PushButtonListener());
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    EditText getUserIdEditText() {
        return (EditText) findViewById(R.id.userIdEditText);
    }

    EditText getPasswordEditText() {
        return (EditText) findViewById(R.id.passwordEditText);
    }

    Button getPushButton() {
        return (Button) findViewById(R.id.pushButton);
    }
}
```

```

class EditTextWatcher implements TextWatcher {

    @Override
    public void onTextChanged(
        CharSequence text, int arg1, int arg2, int arg3) {
        if (getUserIdEditText().length() < 4
            || getPasswordEditText().length() < 4) {
            getPushButton().setEnabled(false);
        } else {
            getPushButton().setEnabled(true);
        }
    }

    @Override
    public void beforeTextChanged(
        CharSequence text, int arg1, int arg2, int arg3) {
    }

    @Override
    public void afterTextChanged(Editable editable) {
    }
}

class PushButtonListener implements View.OnClickListener {

    @Override
    public void onClick(View v) {
        // TODO 認証処理を実装する
    }
}

```

モデル用プロジェクトの作成

MVCパターンでAndroidアプリケーションを開発する場合、モデルは独立したプロジェクトとして作成すると便利です。モデルプロジェクトを作成し、Androidプロジェクトから外部ライブラリとして参照することで、モデルの独立性を高め、ライブラリの再利用も可能です。本節では、モデル

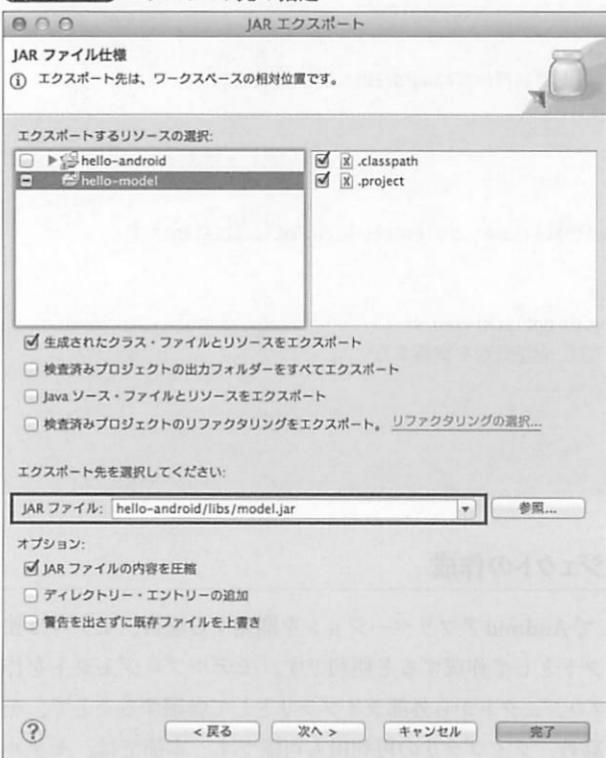
11
12
13
14

プロジェクトを作成し、JARファイルをAndroidプロジェクトから参照する手順を解説します。

はじめに、「hello-model」という名前で新規にJavaプロジェクトを作成し、JARファイルをエクスポートします。モデルクラスの作成はまだ(次項で行います)ですが、JARファイルは作成できるので問題ありません。hello-modelプロジェクトを右クリックし、コンテキストメニューから[エクスポート]を選択し、現れたダイアログから[Java]-[JARファイル]を選択して[次へ]をクリックします。

エクスポート先は、hello-androidプロジェクトのlibsディレクトリ、JARファイルの名前はmodel.jarとしてください(図13.7)。これは、AndroidプロジェクトではlibsディレクトリにあるJARファイルを、自動的に依存ラ

図13.7 エクスポート先の指定



イブラリとして認識するためです。

[完了]ボタンをクリックすればエクスポート作業は完了です。hello-androidプロジェクトの「Android Dependencies」を開くと、図13.8のように「model.jar」が追加されていることがわかります。

なお、この手順でJARファイルを作成する場合は、依存するプロジェクトに修正を加えるごとにJARファイルをエクスポートする必要がありますので忘れないようにしてください。^{注8}

モデルの定義

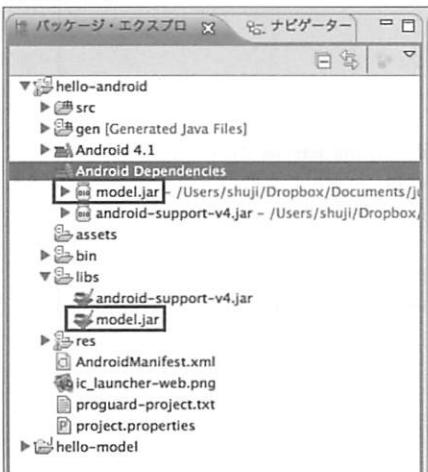
続けてモデルを定義します。

HelloAndroidアプリケーションのモデルは、ユーザ認証を行う処理です。このモデルのインターフェースを作成し、APIを定義します。hello-modelプロジェクト下に、新規にAuthServiceインターフェース(リスト13.4)、およびAuthUserクラス(リスト13.5)を作成してください。

AuthServiceインターフェースのloginメソッドは、認証用のユーザオブジ

注8 このような手順はMavenなどのビルドツールを利用して自動化すると便利です。

図13.8 hello-androidプロジェクトの依存ライブラリの確認



エクト(AuthUser クラス)を引数とし、認証が成功したならばtrueを返すシンプルな API です。

AuthUser クラスは必須ではありませんが、メソッドのパラメータが多い場合やパラメータの仕様が変更されることが予想される場合は作成しておくと便利です。AuthUser クラスは認証に関連するデータをまとめて定義しているため、関連するデータが増えた場合でも AuthUser クラスを変更するだけで、AuthService インタフェースを変更する必要がありません。また、AuthUser クラスでは事前に各パラメータの条件をチェックできるため、安全なオブジェクトをメソッドに渡すこともできます。

最後に JAR ファイルを忘れずにエクスポートします。これで、hello-android プロジェクトから、AuthUser クラスと AuthService インタフェースを参照できるようになりました。

リスト13.4 AuthServiceインターフェース

```
public interface AuthService {
    boolean login(AuthUser authUser) throws IOException;
}
```

リスト13.5 AuthUserクラス

```
public class AuthUser {
    public final String userId;
    public final String password;

    public AuthUser(String userId, String password) {
        if (userId == null || userId.length() < 4) {
            throw new IllegalArgumentException("userId: " + userId);
        }
        if (password == null || password.length() < 4) {
            throw new IllegalArgumentException("password: " + password);
        }
        this.userId = userId;
        this.password = password;
    }
}
```

モデルのスタブ実装

ユーザ認証のインターフェースクラスは定義しましたが、HelloAndroidアプリケーションにおけるユーザ認証の仕様は検討中のため、実装クラスを作成できません。このような状況はアプリケーション開発で、しばしば発生します。

このような場合、仮の実装を作成することで、アプリケーション開発を継続できます。ここでは、特定のユーザIDとパスワードであれば認証が成功するスタブ実装をhello-androidプロジェクトに作成します。AuthServiceStubクラス(リスト13.6)を新規クラスとして作成してください。

スタブ実装は、修正したときに即時に反映できるように、Androidプロジェクト側に作成します。本物の実装クラスはモデルプロジェクト側に作成してください。

Column

独立したモデルプロジェクトのメリット

モデルを独立したJavaプロジェクトとして作成するとさまざまなメリットがあります。

第1に、モデルプロジェクトからAndroidプロジェクトを参照できないため、モデルに含まれるクラスはビューとコントローラから完全に独立します。また、一般的に、GUIに依存するプロジェクトは、そのGUIフレームワークに強く依存します。したがって、GUIに依存しない部分を別プロジェクトとして作成することで、モデルの独立性を高めることができます。

次に、モデルプロジェクトはAndroid SDKに依存しないため、Google App Engine for Java(GAE/J)などのJavaを使ったほかのプロジェクトと、ライブラリとして共有できます。Androidアプリケーションとサーバアプリケーションで共通のクラスを再利用できることは大きなメリットです。

また、Android SDKでは内包するJUnitのAPIのバージョンがJUnit 3系と古く、JUnit 3スタイルでユニットテストを書かざるを得ません。しかしながら、モデルを独立したプロジェクトとすることで、JUnit 4のスタイルでモデルのテストコードを書くこともできます。

HelloAndroidアプリケーションは小さいアプリケーションであるため、効果を実感しにくいですが、アプリケーションの規模が大きいほど、モデルを独立したプロジェクトとして作成する効果も大きくなります。ビルド手順が複雑になることは難点ですが、ビルドはMavenなどのビルドツールで自動化できます。

リスト13.6 ユーザ認証のスタブ実装

```
public class AuthServiceStub implements AuthService {
    @Override
    public boolean login(AuthUser authUser) {
        return (authUser.userId.equals("duke") && authUser.password.equals("3micro"));
    }
}
```

スタブ実装によるイベントの実装

最後の仕上げとして、hello-androidプロジェクトのMainActivityクラスのログインボタンにバインドされたイベントハンドラを実装します(リスト13.7)。

AuthServiceオブジェクトは、MainActivityクラスのフィールドです。パッケージプライベートにしておくことで、ユニットテスト時にAuthServiceをスタブオブジェクトに差し替えることができます。setterメソッドを定義するなどの方法でもよいでしょう。

onClickメソッドでは、各コンポーネントから入力されたテキストを取得し、認証処理を行っています。この処理は長い時間がかかる可能性があるため、AsyncTaskクラスを利用し、バックグラウンドスレッドで実行しなければなりません。

以上で、HelloAndroidアプリケーションは完成しました。エミュレータで実行し、期待どおりに動作するかを確認してください。次節からは、テストを書いていきましょう。

13.3 モデルのテスト

GUIアプリケーションをMVCパターンで設計していれば、モデルのテストは難しくありません。JUnitとこれまでに紹介したユニットテストを支援するライブラリを使ってテストコードを書くことができます。

HelloAndroidアプリケーションでは、モデルを独立したJavaプロジェクトとして作成しています。はじめに、プロジェクトにJUnit4をライブラリ

として追加してください。^{注9}

また、テスト用のソースフォルダ「test」を作成してください。

^{注9} これ以降の操作については、第1章「1.3 JUnitテストを始めよう」(p.4)と共通する部分が多くあります。手順がわからない場合にはそちらを参照してください。

リスト13.7 MVCパターンを適用したMainActivityクラス(抜粋)

```
// TODO 本実装に差し替える
AuthService authService = new AuthServiceStub();
class PushButtonListener implements View.OnClickListener {
    @Override
    public void onClick(View v) {
        String userId = getUserIdEditText().getText().toString();
        String password = getPasswordEditText().getText().toString();
        final AuthUser authUser = new AuthUser(userId, password);
        final TextView status
            = ((TextView) findViewById(R.id.statusTextView));
        new AsyncTask<Object, Object, String>() {
            @Override
            protected String doInBackground(Object... params) {
                try {
                    return authService.login(authUser) ?
                        "ようこそ、" + authUser.userId + "さん" :
                        "ユーザIDとパスワードを正しく入力してください";
                } catch (Exception e) {
                    return "システムエラー";
                }
            }

            @Override
            protected void onPostExecute(String result) {
                status.setText(result);
            }
        }.execute();
    }
}
```

11
12
13
14

認証ユーザクラスのテスト

AuthUser クラスは、login メソッドの引数となるパラメータオブジェクトです。このクラスの責務は、認証に必要なユーザ ID とパスワードを1つの情報としてまとめ、それらの値の妥当性チェックを行うことです。

AuthUserTest(リスト 13.8)では、AuthUser クラスのコンストラクタのテストとして、各パラメータが正しく設定されることと、パラメータが不正な場合に例外が送出されることをテストしています。

このテストが成功することで、AuthUser オブジェクトのユーザ ID やパスワードには、null や4文字以下の文字列が設定されないことが保証されます。login メソッドでは安心して AuthUser オブジェクトを使うことができるようになります^{注10}。

ユーザ認証クラスのテスト

HelloAndroid アプリケーションではユーザ認証に外部サービスを利用します。外部サービスに依存するクラスのユニットテストを行う場合、可能

注10 オブジェクト自体が null であるかはチェックする必要があります。

リスト 13.8 AuthUser クラスのテスト(抜粋)

```
public class AuthUserTest {
    @Test
    public void userIdとpasswordが4文字以上のとき_正しく設定されていること() throws Exception {
        String userId = "userId";
        String password = "password";
        AuthUser instance = new AuthUser(userId, password);
        assertThat(instance.userId, is(userId));
        assertThat(instance.password, is(password));
    }

    @Test(expected = IllegalArgumentException.class)
    public void userIdがnullの場合に例外が発生すること() {
        new AuthUser(null, "password");
    }
}
```

な限り外部サービスに依存する部分を独立させるとテストしやすくなります。その部分をユニットテスト時にスタブやモックで差し替えることができれば、外部環境に依存しないユニットテストを行うことができます。

たとえば、ユーザ認証の仕様が「特定のURLへHTTPリクエストを行い、レスポンスのJSONにsuccessが含まれている場合に認証成功」であるとします。このような場合、レスポンスをJSONとして返す部分(リスト13.9)を独立させるため、インターフェースを定義します。このインターフェースを利用すれば、ユーザ認証クラスはリスト13.10のように書くことができます。

そして、このユーザ認証クラス(AuthServiceImplクラス)は、スタブオブジェクトを利用してすることでユニットテストを書くことができます。リスト13.11では、Mockitoをライブラリとして利用し、JsonHttpClientオブジェ

リスト13.9 外部サービスを定義したインターフェース

```
public interface JsonHttpClient {
    String sendRequest(String url, Map<String, String> params)
        throws IOException;
}
```

リスト13.10 ユーザ認証の実装クラス

```
public class AuthServiceImpl implements AuthService {
    private static final String URL = "http://localhost/api/login";
    // TODO 仕様確定後に実装
    JsonHttpClient httpClient = null;

    @Override
    public boolean login(AuthUser authUser) throws IOException {
        if (authUser == null) throw new IllegalArgumentException();
        String json = httpClient.sendRequest(URL, toParams(authUser));
        return json.toLowerCase().contains("success");
    }

    private Map<String, String> toParams(AuthUser authUser) {
        Map<String, String> params = new HashMap<String, String>();
        params.put("userId", authUser.userId);
        params.put("password", authUser.password);
        return params;
    }
}
```

11
12
13
14

クトのスタブを作成しています^{注11}。

注11 → Mockitoによるモックオブジェクト(p.190)

リスト13.11 AuthServiceImplのユニットテスト

```
public class AuthServiceImplTest {

    AuthServiceImpl sut;
    JsonHttpClient httpClient;
    AuthUser authUser;
    Map<String, String> params;

    @Before
    public void setup() throws Exception {
        sut = new AuthServiceImpl();
        httpClient = mock(JsonHttpClient.class);
        sut.httpClient = httpClient;
        authUser = new AuthUser("u000001", "123456");
        params = new HashMap<String, String>();
        params.put("userId", "u000001");
        params.put("password", "123456");
    }

    @Test(expected = IllegalArgumentException.class)
    public void authUserがnullのとき_例外() throws Exception {
        sut.login(null);
    }

    @Test
    public void httpClientがsucessを含むJSONを返すときにtrueを返すこと()
        throws Exception {
        when(httpClient.sendRequest("http://localhost/api/login", params))
            .thenReturn("{ result: 'success' }");
        assertThat(sut.login(authUser), is(true));
    }

    @Test
    public void httpClientがfailを含むJSONを返すときにfalseを返すこと()
        throws Exception {
        when(httpClient.sendRequest("http://localhost/api/login", params))
            .thenReturn("{ result: 'fail' }");
        assertThat(sut.login(authUser), is(false));
    }
}
```

```

    @Test(expected = IOException.class)
    public void httpClientがIOExceptionを送出するときそのまま送出する()
        throws Exception {
        when(httpClient.sendRequest("http://localhost/api/login", params))
            .thenThrow(new IOException());
        sut.login(authUser);
    }
}

```

13.4 ビューとコントローラのテスト

モデルのテストと比較して、ビューとコントローラ、すなわちGUIのテストは単純ではありません。なぜならば、GUIはユーザーの操作とさまざまなコンポーネントの相互作用で実行されるためです。また、テストの粒度が小さすぎると、膨大なテストを書かなければなりません。その結果、アプリケーションに対する変更の影響も非常に大きくなり、テストコードをメンテナンスしづらいものとします。さらに、GUIの場合、モデルのテストに比べ、テストを作成するコストに対してその効果が相対的に小さくなります。

しかしながら、適切にテスト設計を行い、自動化されたテストを行うことができれば、Androidアプリケーション開発では強力な武器となります。なぜならば、Androidはバージョン更新の頻度が高く、さまざまな端末で実行されることが求められるため、手動テストでは限界があるからです。

本節ではビューとコントローラのテスト方針と、Android SDK(*Android Software Development Kit*)の提供するテスティングフレームワークを紹介します。

Androidアプリケーションの機能テスト

AndroidアプリケーションなどGUIのテストを行う場合、ユニットテストと同じ粒度でテストすることは避けるべきです。なぜならば、GUIのコンポーネントやイベントは、ほかの多くのコンポーネントやイベントに依存しているため、モックやスタブを多用したテストとなりやすいからです。そのような内部実装に強く依存したテストは、画面レイアウトを少し変えただけで影響が出る「脆いテスト」となります。

このため、GUIのテストでは、ユーザがGUIを操作した状況を再現し、期待される結果が得られるかをテストする方針が有用です。このようなテストはユニットテストに対し、**機能テスト**(*functional testing*)と呼ばれます。

機能テストはブラックボックステストです。ユーザ視点でGUIを操作し、ユーザ視点で期待される動作を検証します。したがって、内部実装にはほとんど依存しません。

JUnitはユニットテスト用のフレームワークですが、使い方次第で機能テストに利用できます。実際に標準で提供されるAndroidの機能テストは、JUnitを利用しています。しかしながら、機能テストに適したフレームワークを使うほうが、より効率良くテストを行うことができます。興味があればサードパーティー製のテスティングフレームワークを調べてみてください。

Android SDKが提供するテスティングフレームワーク

GUIアプリケーションのテストでは、テスト実行時に実機やエミュレータを実行し、ユーザの操作を自動化する方法が定石です。実機などを使わずにプログラム内で画面操作をエミュレートする方法もありますが、実機などを利用するほうが直感的にテスト設計が行え、より実際の環境に近い状況でテストできます。

Android SDKでは、エミュレータ上で機能テストを実行するためのフレームワークを提供しています。エミュレータ上で実行したAndroidアプリケーションでユーザ操作を自動化し、GUIコンポーネントの状態などを検証できます。

Android SDKではさまざまな状況でAndroidアプリケーションのテストができるよう多くのクラスを提供しています。代表的なクラスには次のものがあります。

- `ActivityInstrumentationTestCase2` クラス
- `ActivityUnitTestCase` クラス
- `ProviderTestCase` クラス
- `ServiceTestCase` クラス
- `ApplicationTestCase` クラス

それぞれのクラスはAndroidのフレームワークに対応しているため、使いこなすにはAndroidのフレームワークを学ぶ必要があります^{注12}。ここではGUIの機能テストの導入として、ActivityInstrumentationTestCase2クラスを使ったアクティビティのテスト方法を紹介します。

ただし、Android SDKが提供するAndroidのテスティングフレームワークは、JUnit 3.8をベースにしています。したがって、本書で解説してきたassertThat構文やさまざまな拡張機能は利用できません。また、独自のモックAPIやカテゴリ化テストのためのアノテーションなどを利用できます。

まだ機能的に不足している部分はあるものの、Androidアプリケーションのテストを行うための基盤は整備されつつあります。また、サードパーティーからは不足している機能を補完するライブラリも提供されています。

アクティビティのテスト

Androidアプリケーションのアクティビティとは、画面の基本的な構成単位です。通常は、いくつかのアクティビティを画面遷移させてAndroidアプリケーションを構成します。Androidアプリケーションの機能テストでは、アクティビティが基本単位です。

ここでは、HelloAndroidアプリケーションの機能テストとして、アクティビティに対するテストを行います。

● テスト用プロジェクトの作成

Androidアプリケーション開発では、メインとなるプロジェクトとテスト用のプロジェクトを分けて管理します。

メニューから[ファイル]-[新規]-[プロジェクト...]を選択し、新規プロジェクト作成ウィザードに[Android]-[Androidテスト・プロジェクト]を選択してテスト用のプロジェクトを作成します。プロジェクト名は「hello-android-test」とし、テスト対象プロジェクトは本章で作成した「hello-android」プロジェクトを選択してください。

なお、Androidアプリケーションのテストプロジェクトでは、デフォル

^{注12} 詳細はドキュメント(英文。<http://developer.android.com/tools/testing/>)を参照してください。

トでパッケージ名が「テスト対象プロジェクトのルートパッケージ.test」となっています。このままでもテストの実行に支障はありませんが、テスト対象プロジェクトと同じパッケージとしたほうがテストしやすいため、パッケージ名をテスト対象プロジェクトのルートパッケージと同じ名前に変更してください。^{注13}

注13 パッケージ名を選択して[F2]を入力し、末尾の「.test」を削除。

Column

JUnit 3.8におけるテストを書くルール

JUnit 3.8では、Java 1.4以下も対象としていたため、アノテーションベースのテストコードは書くことができません。本書では詳細は解説しませんが、Androidの機能テストを行うには、以下の点に注意すれば十分です。

- テストクラスはTestCaseクラスのサブクラスとする
- テストメソッドはtestで始める
- 共通の初期化処理はTestCaseクラスのsetUpメソッドをオーバーライドする
- 共通の後処理はTestCaseクラスのtearDownメソッドをオーバーライドする
- assertThatメソッドの代わりに、assertTrueメソッドやassertEqualsメソッドを使う

テストクラスはTestCaseクラスのサブクラスとしてください。Androidアプリケーションのテストでは、TestCaseクラスのサブクラスであるActivityInstrumentationTestCase2クラスなどのサブクラスとなります。これらのクラスにはテストを行うための初期化処理や、便利なメソッドが定義されています。

テストメソッドは、メソッド名をtestで開始してください。JUnit 3.8ではテストメソッドであるかの識別にメソッド名の命名規則を使用します。

共通の前処理や後処理は、TestCaseのメソッドをオーバーライドします。このとき、スーパークラスのメソッドを実行することを絶対に忘れないでください。すなわち、setUpメソッドであれば、最初の行にsuper.setUp();を記述します。これを忘れるるとスーパークラスで実行されるはずの前処理が行われず、テストが正しく実行されません。このミスは非常によく発生します。

アサーションにはassertTrueメソッドやassertEqualsメソッドを使用します。このとき、assertThatメソッドとは、引数の順番が逆になるので注意してください。すなわち、assertEqualsメソッドでは、第1引数が期待値で、第2引数が実測値です。逆に指定した場合でも比較検証自体は行われますが、失敗した場合のエラーメッセージが逆になり混乱します。

● MainActivityTest の作成

ActivityInstrumentationTestCase2 クラスは、アクティビティの機能テストを行うための基底クラスです。Android アプリケーションをエミュレータ上で実行し、テストコードで定義したユーザ操作を実行します。そして、各 GUI コンポーネントの状態などを検証します。

それでは、MainActivity クラスのテストクラスを作成します。新規に Java クラスを作成し、図 13.9 のように、名前に「MainActivityTest」を、スーパークラスに「android.test.ActivityInstrumentationTestCase2<MainActivity>」を指定します。また、「スーパークラスからのコンストラクター」にチェックを入れてください。

11
12
13
14

図13.9 MainActivityTestクラスの作成



スーパークラスを指定する際は、[参照...]ボタンをクリックし、クラスの選択ダイアログを利用すると便利です。このとき、図13.10のようにフィルタに ActivityInstrumentationTestCase2 クラスの単語区切り文字である「AITC」を入力すると簡単に目的のクラスを選択できます。

このダイアログから選択した状態では、型パラメータが<T>となっていますので、手動で<MainActivity>に書き換えてください。なお、「ActivityInstrumentationTestCase2<MainActivity>」は「MainActivity の ActivityInstrumentationTestCase2 クラス」と呼びます。リスト13.12は、作成された初期の MainActivityTest クラスです。以上でアクティビティのテストを行う準備が整いました。

● アクティビティテストの初期化

アクティビティのテストでは、setUp メソッドで、初期化されたアクティビティ上のコンポーネントをテストクラスのフィールドに設定します（リスト13.13）。このとき、super.setUp() の実行を忘れる、アクティビティが初期化されないので注意してください。



リスト13.12 MainActivityTestクラス

```
public class MainActivityTest extends
    ActivityInstrumentationTestCase2<MainActivity> {

    public MainActivityTest(Class<MainActivity> activityClass) {
        super(activityClass);
        // TODO 自動生成されたコンストラクタスタブ
    }

}
```

リスト13.13 MainActivityTestクラスの初期化

```
public class MainActivityTest
    extends ActivityInstrumentationTestCase2<MainActivity> {

    MainActivity activity;
    EditText userIdEditText;
    EditText passwdEditText;
    Button pushButton;
    TextView statusTextView;

    public MainActivityTest() {
        super(MainActivity.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        activity = getActivity();
        userIdEditText = (EditText) activity
            .findViewById(R.id.userIdEditText);
        passwdEditText = (EditText) activity
            .findViewById(R.id.passwordEditText);
        pushButton = (Button) activity.findViewById(R.id.pushButton);
        statusTextView = (TextView) activity
            .findViewById(R.id.statusTextView);
    }

}
```

11
12
13
14

● 初期状態のテスト

コンポーネントの初期状態を検証するテストケースは、慣習として「testPreConditions」という名前で作成します。日本語で「test 初期状態」という名前でもかまいませんが、メソッド名は必ず「test」で開始しなければなりません^{注14}。

リスト 13.14 では、2つのEditTextに文字列が入力されていないこと、プッシュボタンが無効になっていること、ステータスを表示するTextViewの文字列が空文字列であることを検証しています(assertThat メソッドとは引数の順番が逆なので注意してください)。

● アクティビティテストの実行

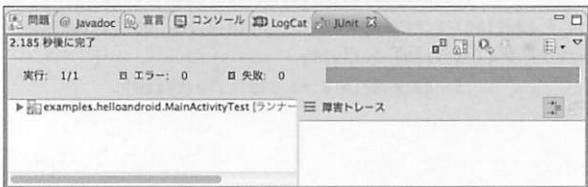
アクティビティのテストを実行するには、テストプロジェクトのコンテキストメニューを開き、[実行] - [Android JUnit Test]を選択します。エミュレータ上でアプリケーションが起動し、JUnitによるテストが実行され、実行結果が表示されます(図 13.11)。

注 14 JUnit 3におけるテストメソッドの規約です。

リスト 13.14 MainActivityTest の初期状態テスト

```
public void testPreConditions() throws Exception {
    assertEquals(userIdEditText.getText().toString(), "");
    assertEquals(passwdEditText.getText().toString(), "");
    assertFalse(pushButton.isEnabled());
    assertEquals(statusTextView.getText().toString(), "");
}
```

図 13.11 Android JUnit Test の実行結果



シナリオテスト

機能テストでテストケースを設計する場合、ユーザ視点でアプリケーションをどのように利用するかを検討します。つまり、アプリケーションのユースケースシナリオ(利用例)をもとにテストケースを定義します^{注15}。このようなテストはシナリオテスト(*scenario testing*)や例によるテスト(*example testing*)と呼ばれます。

ユースケースシナリオのテストを自動化することは一般的に難しいものです。しかしながら、Android SDKの提供するテスティングフレームワークを利用すれば比較的簡単に実装できます。また、ユースケースをもとにしたテストは、顧客やエンドユーザも理解しやすいテストです。

● ハッピーパスのテスト

シナリオテストでは、最初にハッピーパスやサクセスストーリーと呼ばれるシナリオをテストします。ハッピーパスとは、例外的な状況が発生せずに、そのシナリオのゴールに到達するシナリオです。ハッピーパスのシナリオは、後述する拡張シナリオのテストに比べて重要なシナリオです。なぜならば、アプリケーションに対するユーザの要求を満たすシナリオだからです。

たとえば、HelloAndroidアプリケーションであれば、認証が成功するユーザIDとパスワードの組み合わせで入力が行われ、プッシュボタンを押し、ようこそメッセージが表示されるシナリオです。このシナリオをテストコードに変換すると、リスト13.15のようになります。

コンポーネントの初期状態を検証する `testPreConditions` メソッド(リスト13.14)と比較すると、`clickView` メソッドと `sendKeys` メソッドが登場しています^{注16}。これらのメソッドは、Android端末の操作をエミュレートするメソッドで、その名前のとおり、ビューコンポーネントのクリック(タッチ)とキー入力を行います^{注17}。

^{注15} 設計時にユースケースシナリオを定義していればプログラムに変換するだけで済みます(ユースケース駆動開発)。

^{注16} `clickView` メソッドは `android.test.TouchUtils` クラスの static メソッド、`sendKeys` メソッドはテストクラスのスーパークラスに定義されたメソッドです。

^{注17} 以前のバージョンではこの操作をGUIスレッド実装するなど面倒な記述が必要でした。

リスト13.15 ハッピーパスのテスト

```

public void testHappyPath() throws Exception {
    // 1. userIdEditTextを選択し、「duke」と入力する
    clickView(this, userIdEditText);
    sendKeys("D U K E");
    assertEquals("duke", userIdEditText.getText().toString());
    // 2. passwdEditTextを選択し、「3micro」と入力する
    clickView(this, passwdEditText);
    sendKeys("3 M I C R O");
    assertEquals("3micro", passwdEditText.getText().toString());
    // 3. プッシュボタンをクリックする
    clickView(this, pushButton);
    // 4. ステータスバーに
    // 「ようこそ、dukeさん」と表示されていることを確認する
    assertEquals("ようこそ、dukeさん",
                statusTextView.getText().toString());
}

```

そして、初期状態の検証と同様に、それぞれの操作を行ったときに、コンポーネントが期待される動作を行っているかを検証しています。

テストケースを定義したならば、再度テストを実行してください。エミュレータ上でテキスト入力とボタンのクリックが自動的に実行されることを確認できます。

● 拡張シナリオのテスト

最も重要なシナリオはハッピーパスですが、ユーザ操作で想定可能なシナリオをどれだけ考慮しているかは、アプリケーションの品質に大きく影響します。

たとえば、ユーザIDとパスワードは4文字以上で入力されていても、認証に失敗するシナリオがあります(リスト13.16)。このようなシナリオは、ユースケースでは**拡張シナリオ**(*extensions scenario*)と呼ばれます。

拡張シナリオのテストは、可能な範囲でユーザの操作を網羅するべきです。しかしながら、完璧に網羅することは不可能です。プロジェクトの予算や特性などを考慮し、どの程度まで行うかを決める必要があります。

● エラーシナリオのテスト

拡張シナリオは、ユーザが操作可能な範囲でのハッピーパスでないシナリオです。しかしながら、アプリケーションの実行中にはユーザ操作からは想定できないシナリオもあります。

たとえば、認証モデルに認証を行ったところ、例外(IOException)が発生するようなケースです(リスト13.17)。このようなシナリオは、エラーシナリオと呼ばれます。

● シナリオの優先順位

開発では、はじめにハッピーパスを作成し、ハッピーパスを通るように開発を進めることが重要です。ユーザに価値を届けるためには、ハッピーパスを実行でき、そのユースケースの目的を満たせることが必要です。どれだけ拡張シナリオが考慮されていても、ハッピーパスが実行できなければアプリケーションは何の価値も生み出さないからです。このような考え方方はアジャイル開発では特に強く意識される部分です。

リスト13.16 認証に失敗するシナリオのテスト

```
public void testユーザ認証に失敗する() throws Exception {
    // 1. userIdEditTextを選択し、「duke」と入力する
    clickView(this, userIdEditText);
    sendKeys("D U K E");
    assertEquals("duke", userIdEditText.getText().toString());
    // 2. passwdEditTextを選択し、「0r4cle」と入力する
    clickView(this, passwdEditText);
    sendKeys("0 R 4 C 1 E");
    assertEquals("0r4cle", passwdEditText.getText().toString());
    // 3. ブッシュボタンをクリックする
    clickView(this, pushButton);
    // 4. ステータスバーに
    // 「ユーザIDとパスワードを正しく入力してください」と
    // 表示されていることを確認する
    assertEquals("ユーザIDとパスワードを正しく入力してください",
                statusTextView.getText().toString());
}
```

リスト13.17 認証時に例外が発生するエラーシナリオのテスト

```
public void testユーザ認証で例外が発生する() throws Exception {
    activity.authService = new AuthService() {
        @Override
        public boolean login(AuthUser authUser) throws IOException {
            throw new IOException("Error");
        }
    };
    // 1. userIdEditTextを選択し、「duke」と入力する
    clickView(this, userIdEditText);
    sendKeys("D U K E");
    assertEquals("duke", userIdEditText.getText().toString());
    // 2. passwdEditTextを選択し、「3micro」と入力する
    clickView(this, passwdEditText);
    sendKeys("3 M I C R O");
    assertEquals("3micro", passwdEditText.getText().toString());
    // 3. プッシュボタンをクリックする
    clickView(this, pushButton);
    // 4. ステータスバーに「システムエラー」と表示されていることを確認する
    assertEquals("システムエラー", statusTextView.getText().toString());
}
```

13.5 GUIアプリケーションのテストにおける注意点

GUIアプリケーションのテストは、実装コストもメンテナンスコストも、通常のユニットテストよりも高くなります。また、処理の中心であるモデルと、ビューおよびコントローラを比べた場合、テストの重要性が高いのはモデルです。したがって、プロジェクトの予算や性質によって、GUIのテストをどの程度行うかを判断しなければなりません。プロジェクトによっては、ビューとコントローラのテストは手動で行うほうがよい場合もあるでしょう。

Androidアプリケーションでは、プラットフォームの多様性も更新頻度も高いため、テストが自動化されていることは大きなアドバンテージです。拡張シナリオのテストは省略しても、ハッピーパスのテストがすべて自動化されていれば、新しいAndroid SDKがリリースされたとしても簡単に全機能のテストを行うことができます。もし、ハッピーパスのテストが失敗

したならば、アプリケーションにとって致命的な問題です。すぐに対応するためにも、可能な限りは自動化することをお勧めします。

なお、本書で紹介したAndroid SDKのテスト機能はほんの一端です。アクティビティのテスト以外にも、Androidには特有のテストがいくつかあります。本書では紹介しきれませんので、ドキュメントなどを参照してください。^{注18}

また、Androidのテスティングフレームワークは、非常に高機能でさまざまなテストを可能とします。しかしながら、GUIの操作にはたくさんのバリエーションがあるため、サポートしていない検証パターンも多くあります。そのような場合は、サードパーティ製の拡張ライブラリを導入したり、プロダクションコードを工夫してテストを行ってください。

もちろん、基本的な部分は、おおむねサポートされています。重要な部分のみテストを行い、そうでない部分は省略することも重要です。

11
12
13
14

^{注18} <http://developer.android.com/tools/testing/>

Column

正常系／準正常系／異常系

システム開発を行った経験があれば、システムテストなどのフェーズで「正常系」「準正常系」「異常系」といった用語を聞いたことがあるでしょう。

基本的にはハッピーパスが正常系、拡張シナリオが準正常系、エラーシナリオが異常系に対応しています。しかしながら、テストの粒度や目的についてはまったく異なる場合があります。プロジェクトごとに定義を確認しておくことが重要です。

Column

Webアプリケーションの機能テスト

Webアプリケーションの機能テストを行う場合は、Selenium^{注a}というフレームワークが有名です。Seleniumを使うことで、さまざまなブラウザ上での機能テストを自動的に行うことができます。Seleniumはさまざまな使い方のできるフレームワークなので、Webアプリケーションの開発を行っているのであれば、調べてみる価値があるでしょう。

^{注a} <http://seleniumhq.org/>

第14章

コードカバレッジ テスト網羅率の測定

ユニットテストによりプロダクションコードの振る舞いは検証できますが、実行したユニットテストが十分に行われているかを測定する絶対的な基準はありません。なぜならば、すべての入力パターンに対するテストケースを洗い出すことも、実行することも、事実上は不可能だからです。また、テスト対象のソフトウェアが顧客の要求を満たしているかどうかは、ユニットテストの範囲では検証できません。

しかしながら、何らかの基準を設け、ソフトウェアの性能や品質を評価できれば、効果を実感できるだけでなく、ミスや不具合を検出しやすくなります。本章では、ユニットテストの実行網羅率を測定するコードカバレッジについて解説します。

14.1 コードカバレッジとは？

ユニットテストにおけるコードカバレッジ(*code coverage*。以下、カバレッジと表記します)とは、テストの実行時にプロダクションコードが実行された割合を表したものです^{注1}。

カバレッジの基準

カバレッジは、ユニットテストを実行したときにプロダクションコードの実行割合を測定した値です。カバレッジは、どのような基準で測定するかによって異なる値となります。代表的な基準として、C0、C1、C2の3つの基準があります。

なお、それぞれの基準は、どれが精度が高くどれが精度が低いという性

^{注1} カバレッジはユニットテストに限定されるわけではありませんが、ユニットテストで活用されることが多い指標です。

質のものではありません。あくまでカバレッジを測定し、割合を計算するための計算方法です。

● C0(命令網羅)

C0(命令網羅)は、プログラム中に定義されたすべての「命令」について1回以上実行されたかについて判定する基準です。

Javaでの「命令」は、ソースコードレベルでのステートメントを基準とする方法と、バイトコード上のインストラクション(実行命令)を基準とする方法の2つの方法があります。どちらの場合でもC0カバレッジ(命令網羅率)が100%であることは、ユニットテストでプログラム内の全命令を少なくとも1回は実行しているということです。

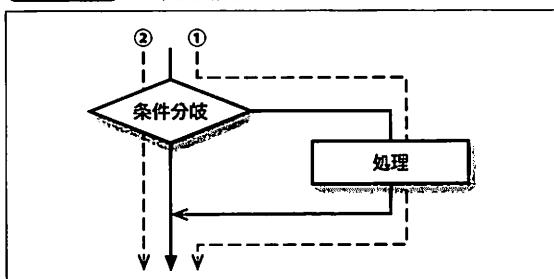
● C1(分岐網羅)

C1(分岐網羅)は、プログラム中の各分岐について1回以上実行されたかについて判定する基準です。

Javaでの分岐とはif文やswitch文などの条件分岐、try-catch構文による例外処理などが相当します。C1カバレッジ(分岐網羅率)が100%であるということは、ユニットテストでプログラムのすべての条件分岐、すなわちすべてのブロックが実行されていることです。

たとえば、図14.1のようなプロダクションコードがあった場合、C1カバレッジでは①と②の2つの経路を通りのテストを実行することで網羅率は100%となります。一方、C0カバレッジでは①の経路だけを実行すれば、網羅率は100%となります。

図14.1 C1(分岐網羅)とC0(命令網羅)



● C2(条件網羅)

C2(条件網羅)は、プログラムの判定条件に着目し、すべての真偽条件の組み合わせが実行されているかどうかを判定する基準です。複数の条件の組み合わせによる条件分岐において、すべての分岐が実行されたかではなく、すべての真偽値の組み合わせが実行されたかを測定します。

なお、C0カバレッジやC1カバレッジと異なり、C2カバレッジを測定できるツールはほとんどありません。

カバレッジ測定の効果

カバレッジは、純粹にプロダクションコードの実行割合を表す指標でしかありません。しかしながら、カバレッジが提供する測定値から、ユニットテストを進めるうえで有用な情報を多く得ることができます。

たとえば、あるパッケージのカバレッジが、ほかの似たパッケージよりも極端に低いのであれば、おそらくはユニットテストが足りないか、ユニットテストの実装に問題がある可能性があります。ある処理のユニットテストを追加したにもかかわらず、カバレッジに変化がないのであれば、何らかの理由で意図した処理が実行されていない可能性があるでしょう。

14.2 カバレッジツールの利用

カバレッジは、カバレッジツールを導入することで手軽に測定できます。Eclipseでもカバレッジ測定用のプラグインがいくつか利用できます。

カバレッジツールを利用すると、実行の割合がパーセントで表示されるだけでなく、ソースコード上で実行された行と、実行されなかった行がマークアップされて表示されます。カバレッジツールをうまく利用することで、プログラミングとユニットテストを効率良く進めることができます。

本節では、カバレッジおよびカバレッジツールの利用方法をいくつか紹介します。

ユニットテストの漏れを監視する

カバレッジは、ユニットテストがプロダクションコードを十分に実行しているかの指標です。カバレッジを定期的に測定し、プロジェクトで定めた基準値を下回ったならば原因を分析してください。もし、テストケースが漏れているのであれば、不足しているテストケースを追加します。

ただし、基準値を下回った場合でも、下回ったこと自体を問題とすべきではありません。カバレッジが低くなった原因を分析したうえで、必要なテストケースを追加します。たとえば、効果的なリファクタリングが行われ、多くの重複コードが減った結果としてカバレッジが低くなったのであれば、問題はありません^{注2}。ですが、一部のプログラマがユニットテストを書かなかったことが原因であれば、改善する必要があります。

テストケースの問題を検知する

ユニットテストでは、プロダクションコードの分岐や処理ロジックを考慮して、テストデータを作成します。

プログラム上の分岐や条件がテストデータの作成基準のすべてではありません。しかしながら、重要な処理や分岐が実行されていなかったならば、テストケースに何らかの不備がある、またはテストケース自体が漏れていると考えられます。逆に十分なテストデータによりテストが実行されているにもかかわらず、実行されていない処理があったならば、その処理は不要である可能性があります。

このように実行されている(または、されていない)コードを確認することで、テストケースの問題に気付くことができます。

コードの実行をトレースする

カバレッジツールは、実行状態を見やすくするため、問題箇所を絞り込むために利用できます。また、テストケースを作成したときに、意図し

^{注2} リファクタリングを行うとコードの全体構造が変わるために、カバレッジは高くなることもあります。

た処理を実行しているかを確認するためにも利用できます。

開発を進めていると、ユニットテストを実行したときに予期しない失敗や成功となることがあります。このような場合、通常はデバッガや標準出力のコードを埋め込むなどの手段で、どのようにコードが実行されたかをトレースして問題を解決することがありますが、ここでデバッガの代わりにカバレッジツールを利用することもできます。

カバレッジツールを利用してテストを実行すれば、実行された部分と実行されなかった部分をグラフィカルに見ることができます。それでも問題が判明しないのであれば、ピンポイントでブレークポイントを設定し、デバッガを実行できます。

カバレッジツールの選択肢

カバレッジを測定するツールには、コマンドラインから実行するツールと、EclipseなどのIDEから実行するツールの2種類があります。また、無償のツールもあれば有償のツールもあります。しかしながら、C0カバレッジやC1カバレッジを測定する用途であれば無償のツールでも十分に実用的です。

本書では、Eclipseのプラグインとして利用できる「EclEmma」を紹介します。

14.3 EclEmmaによるカバレッジ測定

EclEmma^{注3}はEPLライセンスで提供されています。その特徴は、Eclipseのプラグインとして簡単に導入できることや、カバレッジの測定結果をグラフィカルに確認できること、既存のプロジェクトに修正や変更を与えずに利用できることなどです。

また、カバレッジ測定エンジンである「JaCoCo」(Java Code Coverage library)^{注4}は独立したモジュールとなっており、コマンドラインからの実行

注3 <http://www.eclemma.org/>

注4 <http://www.eclemma.org/jacoco/trunk/>

だけでなく、AntやMavenなどのビルドツールからも利用できます^{注5}。

EclEmma プラグインのインストール

EclEmma プラグインの導入は簡単です。Eclipse のメニューから [ヘルプ] - [Eclipse マーケットプレース] を選択し、EclEmma で検索してください。あとは検索結果の一覧から「EclEmma Java コード・カバレッジ」を探して、

注5 ➔ Maven によるカバレッジレポート (p.293)

Column

そのほかのカバレッジツール

本書執筆時点では、EclEmma 以外にもカバレッジツールはいくつかの選択肢があります。無償で利用できるツールとしては、「Cobertura」^{注a} が有名です。Cobertura は、カバレッジツールの先駆けであった「JCoverage」^{注b} の後継にあたるツールであり、EclEmma と同様に現在も活発に開発が行われています。

EclEmma と Cobertura を比較すると、EclEmma は Eclipse プラグインとしてインターフェースなどが優れており、Cobertura はコマンドラインやビルドツールからの実行と、HTML などでのレポート作成について優れていると言えます。また、EclEmma はカバレッジ測定エンジンである EMMA の開発が止まっている点が難点でした。このため、特に Maven などのビルドツールを利用しているユーザは Cobertura を好みます。

ところが、EclEmma 2.0.0 からカバレッジ測定エンジンが JaCoCo に代わりました。これにより、Cobertura と同等にコマンドラインからも実行しやすくなっています。このため、現在では EclEmma のほうが総合的に使いやすいと思われます。

有償のツールも選択肢に加えるのであれば、「Sonar」^{注c} の評判が高いです。Sonar はカバレッジの測定だけでなく、コードのメトリクス測定^{注d} なども行える総合的なツールであり、Eclipse を代表とする各種 IDE 用のプラグイン、コマンドラインからの実行、各種ビルドツールや継続的インテグレーションツールへの対応など、有償ならではの幅広いサポートが行われています。なお、オープンソースプロジェクトなどの個人利用であれば無償で利用できますので、興味があれば試してみるとよいでしょう。

注a <http://cobertura.sourceforge.net/>

注b <http://www.jcoverage.com/>

注c <http://www.sonarsource.org/>

注d ソフトウェア、主にソースコードをもとに、さまざまな観点で定量的な評価を行うこと。

11
12
13
14

[インストール]ボタンをクリックし、Eclipseを再起動すればインストールは完了です。

インストールが成功すれば、メニューバーにカバレッジ測定用のアイコンが追加されます。

EclEmmaプラグインの実行

EclEmmaプラグインを利用すると、JUnitテストを実行する場合とほぼ同じ操作でカバレッジを測定できます。カバレッジを測定するには、JUnitテストを実行するときと同様に、プロジェクトエクスプローラなどで実行したいテストクラス(またはパッケージやプロジェクト)のコンテキストメニューを開き、[カバレッジ]-[JUnitテスト]を選択します。

EclEmmaプラグインによりJUnitのテストが実行されると、カバレッジが測定されます。そして、テストが完了すると、カバレッジビューにカバレッジの測定結果が表示されます(図14.2)。表示されない場合は、Eclipseのメニューから[ウィンドウ]-[ビューの表示]-[カバレッジ]を選択して表示させてください)。

デフォルトの設定では、カバレッジの測定を行った場合にカバレッジビューが表示されるようになっています。この設定を変更したい場合は、Eclipseの設定画面から[Java]-[コード・カバレッジ]を開き、「自動的にカバレッジ・ビューを開く」のチェックボックスで行います。

Column

EclEmmaプラグインとJaCoCoエンジン

EclEmmaプラグインは、もとは「EMMA^{注e}」という名前のカバレッジ測定エンジンを使用していました。

しかしながら、EMMAの開発は事実上停止しています。このため、EclEmmaの開発者たちによって後継となるカバレッジ測定エンジン「JaCoCo」が開発され、2011年12月にリリースされたEclEmma2.0.0から使用されています。

このような背景があるため、現在はEclEmmaという名前のプラグインですが、将来的には名称が変更されるかもしれません。

^{注e} <http://emma.sourceforge.net/>

カバレッジビューとカウンタ

カバレッジビューには、ドリルダウン型で表示されるパッケージ／クラスなどの「要素」に対し、カバレッジのパーセンテージとグリーンとレッドのバーを表示する「カバレッジ」、カバレッジの基準に対する数値データなどが表示されます。

カバレッジの基準としては、細かく測定するほうから順に次の5種類が提供されています。

- 命令カウンタ
- ブランチカウンタ
- 行カウンタ
- メソッドカウンタ
- 型カウンタ

これらの基準は**カウンタ**と呼ばれます。

EclEmmaでは、これらのカウンタのほかに「複雑度」を選択して表示することもできます。これらは、カバレッジビューの右上にある[▽]メニュー(ビューメニュー)から切り替えることができます(図14.3)。

● 命令カウンタ

命令カウンタ(*instruction counters*)はC0カバレッジに相当する最小粒度のカウンタで、Javaのバイトコードレベルでの命令をカウントしてカバレッジを測定します。この指標は最も詳細な情報を提供します。

命令カウンタは、バイトコードにデバッグ情報が含まれなくとも利用で

図14.2 カバレッジビュー



11
12
13
14

きることが特徴です。

● ブランチカウンタ

ブランチカウンタ (*branches counters*) は C1 カバレッジに相当し、すべての if ステートメントと switch ステートメントをカウントしてカバレッジを測定するカウンタです。if 文や switch 文などの分岐をカウントし、実行された分岐の割合をカバレッジとして測定します。ただし、EclEmma では例外処理を分岐と見なしません。通常のカバレッジ測定では、このブランチカウンタをデフォルトのカウンタとしてもよいでしょう。

このカウンタを使用するには、バイトコードにデバッグ情報が含まれている必要があります。

● 行カウンタ

行カウンタ (*line counters*) は、バイトコードに埋め込まれたデバッグ情報のコードライン情報から、実行された行をカウントしてカバレッジを測定するカウンタです。このカウンタは非常にわかりやすいですが、カバレッジが高いこととテストの網羅率の関係が薄い点に注意してください。

ブランチカウンタと比較した場合、行カウンタでは実行ライン数を全体のライン数で割った値がカバレッジです。たとえば、重要なエラー処理が 1 ブロック 1 行で構成され、それ以外のコードが 1 ブロックで 99 行であったとします。このとき、このテストのカバレッジは、行カウンタでは 99% になりますが、ブランチカウンタでは 50% です。

ライン数という誰にでもわかりやすい基準を使ったカウンタですが、扱いには十分注意してください。

図 14.3 カウンタの選択



● メソッドカウンタ

メソッドカウンタ (*method counters*) は、クラスごとに実行されたメソッドをカウントしてカバレッジを測定するカウンタです。メソッドカウンタでは、例外が発生せずに最後まで実行された場合に「実行された」と見なされます。なお、抽象メソッドはカウント対象とはなりませんが、*static* イニシャライザとコンストラクタはカウント対象となります。

メソッドカウンタは、テストが行われていないメソッドを探すときに、有効なカウンタです。

● 型カウンタ

型カウンタ (*type counters*) は、各クラスに定義された任意のメソッドが少なくとも1回ずつ実行されたか否かのカウントでカバレッジを測定するカウンタです。メソッドと同様に *static* イニシャライザとコンストラクタの実行も1回としてカウントします。インターフェースに定義された *static* イニシャライザも同様です。

型カウンタは、テストが行われていないクラスを探すときに、有効なカウンタです。

● 複雑度

複雑度(循環的複雑度。 *cyclomatic complexity*) とは、Thomas McCabe によって提唱されたプログラムの複雑さを測定する基準です。

簡単に言えば、メソッド内のコードの複雑性を表しており、if文やfor文などで分岐が増えるほど高い値となります。ソフトウェアのメトリクス測定では、この値が一定値以上のクラスやメソッドは、可読性が低く不具合が混入する可能性が高いと判定されます。なお、複雑度の値は、そのメソッドのすべての分岐を実行するのに最小でも必要なユニットテストのテストケース数となる性質があります。

EclEmma プラグインでは、複雑度に対し、テストで実行したパス数の割合をカバレッジとして表示します。複雑度を選択することで、複雑すぎるメソッドの存在を発見することもできます。

コードハイライト

EclEmma プラグインのカバレッジビューでクラス名やメソッド名をダブルクリックすると、対象のソースコードがエディタに表示されます。このとき、カバレッジの測定時に実行された行はグリーンに、実行されなかった行はレッドに、一部が実行された行はイエローにハイライト表示されます(図 14.4)。

これにより、どの行が実行され、どの行が実行されなかつたかが直感的に理解できます。テストが意図したコードブロックを通過しているかを確認できるでしょう。また、予期せぬ挙動でテストが失敗したならば、そのときにどのようなステップで実行されたかを簡単に確認できます。

測定結果のクリア

カバレッジによるコードハイライトを消したい場合など、カバレッジの測定結果をクリアするときは、カバレッジビューの[アクティブ・セッションを除去]ボタン()をクリックします。

14.4 カバレッジに関するよくある疑問

カバレッジは非常に有用な指標であり、ユニットテストを効率良く行うために活用できます。しかしながら、誤った形で導入するとソフトウェア

図 14.4 EclEmma によるコードハイライト

```

1  public class Calc {
2
3      static {
4          System.out.println("init!");
5      }
6
7      public Calc() {
8
9      }
10
11     public double divide(int x, int y) {
12         if (y == 0) {
13             throw new IllegalArgumentException();
14         }
15         return x / y;
16     }
17 }
18
19 }
```

開発を混乱に導く危険な存在となります。

そこで本章の最後に、カバレッジを正しく理解し、適切に運用するための、よくある疑問をまとめました。

カバレッジとは何か？

カバレッジは、ユニットテストによってプロダクションコードがどの程度の割合で実行されたかを表す指標です。

したがって、ある重要な処理が丸々抜けていたとしても、そもそも仕様が誤っていたとしても、カバレッジには反映されません。つまり、カバレッジが100%であったとしても、コードの品質や仕様の妥当性について保証するものではありません。

11
12
13
14

カバレッジ測定の目的は何か？

カバレッジは、ユニットテストを効率良く進めるために利用します。

カバレッジを測定することで、ユニットテストが不十分な個所を検出できます。また、実行されていないコードを目視できることにより、意図したとおりにテストコードが実行されているかを確認できます。このような情報は、品質に直接反映されるものではありませんが、プログラマが開発

Column

デフォルトコンストラクタとカバレッジ

カバレッジ測定を行ったあと、コードハイライトでクラスの宣言部分のみがレッドとなることがあります。これは、「コンストラクタが明示的に宣言されていない場合、引数なしのpublicコンストラクタが暗黙的に定義される」というJavaの言語仕様と関連しています。

つまり、カバレッジ測定時にデフォルトのコンストラクタが実行されていないため、デフォルトコンストラクタに関するカウンタが増えず、実行されていないラインとして表示されているのです。したがって、デフォルトのコンストラクタを実行するテストコードを追加すればグリーンにできます。しかしながら、グリーンにする…… すなわちカバレッジを高くすることを目的としないように注意してください。

を進めるうえで有効な情報となります。

くれぐれもカバレッジの値を高くすることを目的にしてはいけません。カバレッジは、プロダクションコードの実行網羅率を表す指標でしかないので、カバレッジが高くとも、品質は保証されません。

何%のカバレッジを目標とするべきか?

この疑問に対する普遍的な答えはありませんが、80%から90%程度に設定するプロジェクトが多いようです。ほかのテスト戦略と同様に、どの程度のテストを行うかの基準としてカバレッジの目標値を設定します。

具体的には、そのプロジェクトのテスト戦略(ユニットテストの方針)で決定します。ミッションクリティカルな業務アプリケーションで、かつ予算も豊潤であれば高い値を設定するでしょう。社内向けの業務アプリケーションであればやや低い値を設定するかもしれません。また、Swingアプリケーションなどユニットテストが行いにくいアプリケーションであれば、目標とするカバレッジも低くせざるを得ません。パッケージやサブシステムによって目標値が異なることもあります。

ユーティリティクラスのコンストラクタをどう扱うか?

採用しているJavaのコーディング標準によっては、staticメソッドのみを提供するユーティリティクラスについて、「コンストラクタをprivateとすることでインスタンス化を意図していないクラスであることを明確にする」といった規約があります(リスト14.1)。

このような規約を採用しているプロジェクトでカバレッジを測定している場合、privateのコンストラクタがテストで実行できず、カバレッジが低くなることがあります。このような場合に、リフレクションAPIを使用するなどの方法で、強引にprivateのコンストラクタを実行することもできます。しかしながら、その労力に対するメリットはありません。

繰り返しになりますが、カバレッジを高くすることを目的としないでください。正当な理由があって実行されないコードがあることは妥当なことです。

再現困難な例外処理をどうするか?

JDBC ドライバの読み込み時にキャッチしなければならない ClassNotFoundException、文字コードを扱うときに発生する可能性のある UnsupportedEncodingException、File I/O の close 処理で発生する可能性のある IOException など、ユニットテストで再現困難なチェック例外は少なくありません。

しかしながら、Java の言語仕様では、このようなチェック例外について try-catch 句で例外処理を行なうか、メソッドの throws 句に記述しなければなりません。このような制約があるため、ユニットテストで検証困難なコードブロックを記述せざるを得ません(リスト 14.2)。

このような場合、Exception や Throwable で catch 節を書いて回避することもできます。その結果、例外処理ブロックが減るためカバレッジは高く

リスト14.1 privateコンストラクタ

```
public class NumberUtils {
    private NumberUtils() {
        // このコードは実行されない（カバレッジも通せない）
    }

    public static String format(int num) {
        return new DecimalFormat("###,###,###,###,##").format(num);
    }
}
```

リスト14.2 再現困難な例外処理

```
public class StringUtils {
    public static String toString(byte[] bytes) {
        try {
            return new String(bytes, "utf-8").intern();
        } catch (UnsupportedEncodingException e) {
            // この例外を発生させるのは難しい
            throw new RuntimeException("発生するはずがない", e);
        }
    }
}
```

なるでしょう。しかしながら、先ほどの例と同様、カバレッジを高くする目的でコードを改悪してはいけません^{注6}。

いつカバレッジを測定するか？

Eclipseなどでカバレッジを測定するならば、いつでも気になったときに行ってください。カバレッジの測定はユニットテストの実行と同じくらい簡単にできます。コードハイライトはデバッガに勝る便利ツールになります。自分の書いたテストケースの実行状況を確認するために積極的にカバレッジを測定すべきです。

もし、継続的テスト(第15章)を行っているのであれば、デイリービルドやコミット／プッシュのタイミングで測定するのが効果的です。Jenkinsなどを利用しているれば、カバレッジの閾値を設定し、閾値を下回ったときに警告メールを送信することも有効です。

継続的テストを行っていないのであれば、週に1回でもよいので定期的に測定したほうがよいでしょう。カバレッジは低くなるのはあっという間ですが、一度低くなったカバレッジを回復させるには手間がかかります。

カバレッジは本当に役に立つのか？

カバレッジは、十分に良いテストが行われているならば、非常に役に立ちます。

十分なテストが実行されていることを前提として、カバレッジの数値に大きな変化があったならば、何らかの問題が発生している可能性があります。また、プロダクションコード全体からテストが薄い部分を分析してくれます。

しかしながら、テストが十分に行われていない場合や、テストの内容がいい加減な場合はカバレッジは何の役にも立ちません。

注6 Java 7で導入された例外の再スロー機能や例外のマルチキャッチ機能を利用してことで、コードを改悪することなく、例外処理を簡潔に記述できます。

改善のための口語

Part 4

- 第15章 程度助詞の「と」——「と」と「も」の意味と用法
- 第16章 「と」の取扱い規範——「と」の文法的規範
- 第17章 指示代詞の「それ」——「それ」と「その」の意味と用法

第15章

継続的テスト

すばやいフィードバックを手に入れる

ソフトウェア開発ではさまざまなテストが行われます。テストの種類によって目的も手法も異なりますが、ソフトウェアを何らかの形で検証し、期待する結果とならなかった場合にはその欠陥を修正します。しかしながら、ある欠陥を修正した結果、別の欠陥が発生することがあります。このような手戻りは「リグレッション」と呼ばれます。開発を進める際には、欠陥の修正による恩恵とリスクのバランスを考慮する必要があります。

このような問題を解決するひとつの方法は、テストを早い段階から、自動的に、繰り返して行うことです。本章では、継続的なテストの実行に必要なツールと、最近の開発では一般的となりつつある継続的インテグレーションについて解説します。

15.1 継続的テスト

アプリケーション開発で十分なユニットテストを行うことは簡単なことではありません。ユニットテストに関する知識や技術を学ぶ必要もあり、慣れるまではテストコードを書くコストは非常に高いものです。また、ユニットテストの導入効果を感じるには、ある程度の経験が必要です。

特に開発チームにユニットテストを行う文化がない場合、効果を実感し、テストを行う文化を根付かせる必要があります。そのためには継続的テストを導入し、ユニットテストの効果を見る形で開発チームにフィードバックすることが効果的です。

開発チームへのすばやいフィードバック

ユニットテストは、早い段階から、自動的に、何度も繰り返して実行することで、最大の恩恵を得ることができます。その恩恵とは「開発チームへ

のすばやいフィードバック」です。

●早期からテストする

従来、テストはソフトウェアが完成してから行うものと考えられてきました。ですが、ソフトウェアを構成する最小部品のクラスなどを対象とするユニットテストは、開発の早い段階から実施できます。また、ユニットテストは、対象となるクラスの内部実装を考慮したテストです。したがって、対象のクラス実装と同時にテストを作成するほうが効率良くテストコードを書くことができます。そして、ユニットテストが実施されているクラスは、安心して利用できます。

●自動的にテストする

ユニットテストは、自動的に実行します。ユニットテストは、GUIのツールやコマンドラインからプログラムとして自動的に実行できなければなりません。もし、ユニットテストの実行がすべて手作業であったならば、テストの実行に多くのリソースを割かなければなりません。

しかしながら、プログラムとして自動的に実行できるならばテストの実行コストをほとんど無視できます。また、手動で行うテストと異なり、プログラムとして実行されるテストは、実行手順を誤ることはありません。

人間が判断する必要のあるユーザビリティテストなどを自動化することはできませんが、ユニットテストは簡単に自動化できます。そして、自動化されることで、実行コストをほとんど無視でき、繰り返し実行できるようになります。

●繰り返しテストする

ユニットテストは、何度も繰り返して実行します。ソフトウェアは多くのクラスがお互いに依存して構築されます。そして、各クラスはいつでも修正できるため、あるクラスを修正したとき、そのクラスに依存しているクラスにも影響が伝搬します。このため、特にリリース後の欠陥修正や機能拡張ではリグレッションのリスクがあります。リグレッションを嫌がり、「動いているコードは原則として修正してはならない」といった制約を課される場合もあります。

これらは、コードを変更したときの影響が予測できないという不安が原因です。ユニットテストを繰り返し実行できれば、この不安は安心となります。なぜならば、リグレッションが発生したならば、すぐに気付き、修正できるからです。

継続的テストとは？

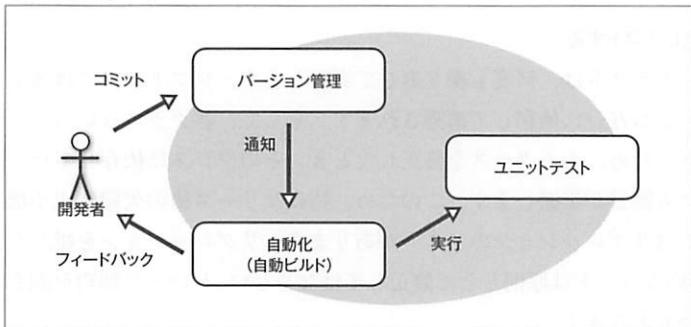
継続的テスト (*continuous testing*) とは、早い段階からテストを繰り返し実行するプラクティスです。継続的テストを実践することで、修正や追加による影響が開発チームにすばやくフィードバックされます。問題が起きたときに修正可能であり、それを通知するしくみがあるため、開発チームは安心してコードを修正できます。

継続的テストは、ソフトウェアの品質を直接高めるわけではありませんが、開発チームは自信を持ってソフトウェアをリリースできるようになります。

継続的テストを行うための3つの要素

ソフトウェア開発に継続的テストを導入するには、3つの要素が必要です。それは、ユニットテスト、自動化、バージョン管理です(図15.1)。これらは、現代ソフトウェア開発の三本柱とも呼ばれます。3本の柱がしっかりとソフトウェア開発を支えるような体制を築くことで、開発チームは

図15.1 現代ソフトウェア開発の三本柱



プロダクションコードを作るだけの作業から解放され、より創造的な仕事に集中できるようになります。

● ユニットテスト

ユニットテストが継続的テストに必要なことは言うまでもないでしょう。JUnitを利用してテストコードを作成することで、プログラムとして実行可能とします。

● 自動化

自動化は継続的テストで最も重要な要素です。コンパイル、ユニットテスト、パッケージング、デプロイといったビルドプロセスを、コマンドラインなどから自動的に実行できるようにします。なぜならば、手動で何度も繰り返して実行する手間とコストは計り知れないからです。また、自動化することにより、ビルド作業で発生する人為的なミスを防ぐことができます。プロジェクトのはじめに少しだけコストを払い、ビルドプロセスを自動化してください。

15

16

17

● バージョン管理

バージョン管理は、継続的テストを行うために必要な最後の要素です。もし、プロダクトがバージョン管理されていなければ、最後にテストが成功したときのプロダクトと、テストが失敗したときのプロダクトの差分を知ることができません。差分を知ることができなければ、テストが失敗した原因を分析するのに多くの時間がかかります。

プロダクトがバージョン管理されていれば、影響を与えた範囲が絞り込まれるため、原因の発見は簡単になるでしょう。また、問題の解決が困難であるならば、プロダクトをその前のバージョンに戻すことで、元の状態に復元できます。

継続的テストの運用

典型的な継続的テストの運用では、バージョン管理システムに変更が行われたタイミングで、ユニットテストを含む自動化されたビルドプロセス

を実行します。このとき、すべてのテストが実行され、すべてのテストが成功し続けることを目標とします。

しかしながら、現実では予期せぬ問題が発生してテストは失敗します。ソフトウェア開発において、予期できない問題が発生することを減らすことはできてもゼロにすることはできません。

繰り返しますが、重要なことは、問題の発生を早い段階で検知し、開発チームにフィードバックすることです。開発チームは、早い段階で問題が検知されるしくみがあることで、プロダクトの開発に集中できます。

継続的テストとリファクタリング

継続的テストを導入すると、リファクタリングをより積極的に行うことができます。継続的テストとリファクタリングにより、期待通りに動作するきれいなソースコードを手に入れることができます。

● ユニットテストとリファクタリング

リファクタリングは、主にソースコードのメンテナンス性を高めるためのテクニックです^{注1}。特に重複したコードを減らし^{注2}、再利用や拡張が行いやすい設計にする目的で行われます。

定期的なリファクタリングは、品質の高いソフトウェアを開発するために必要不可欠です。そして、リファクタリングを行うにはユニットテストが必要となります。ユニットテストなしでリファクタリングを行ってはいけません。なぜならば、ユニットテストがなければ、プロダクションコードの修正により影響があったとしても、気付くことができないからです。

ユニットテストが成功しているならば、リファクタリングの前後で外部的振る舞いが変更されていないことが保証されます。このため、ユニットテストが十分に行われているプロダクトでは、いつでも大胆なリファクタリングを行うことができます。

注1 性能改善などの目的で行われることもあります。

注2 DRY原則(*Don't Repeat Yourself*)として知られています。

● 積極的なリファクタリング

継続的テストを導入すると、プログラマは何時でもリファクタリングできるようになります。バージョン管理システムにプロダクションコードの変更を登録すれば、自動的にすべてのユニットテストが実行されるからです。問題が発生したならばそれを通知するしくみになっているため、変更による影響を恐れずにプロダクションコードを修正できます。致命的な問題が発生したとしても、バージョン管理システムを利用して問題が発生する前の状態に戻すこともできます。

大規模なリファクタリングでも、小規模なリファクタリングであっても変わりません。大きな修正でも小さな修正でも、ソフトウェアに影響を与えるかもしれません。したがって、「もしかしたら影響を与えててしまうのでは」と不安になり、「動いているコードは変更してはならない」というルールを作ることも一理あるのです。しかしながら、継続的テストによるセーフティネットがあれば、安心して、いつでも、積極的にリファクタリングを行うことができます。

15
16
17

15.2 Mavenによるビルドプロセスの自動化

Javaのソフトウェアをリリースするには、ソースコードの作成後に、コンパイル、ユニットテスト、パッケージング、デプロイなどの手順が必要になります。このような手順はソフトウェアのビルド(build)と呼ばれます。Javaのソフトウェア開発では、Ant^{注3}、Maven^{注4}、Gradle^{注5}などのビルドツールがよく利用されていますが、本書ではEclipseで正式にサポートされたMavenを紹介します。

ビルドツールを利用することで、ユニットテストはビルドプロセスに組み込まれます。JUnitテストは、EclipseのGUI上やコマンドラインから実行できますが、ビルドツールを利用して実行することもできます。たとえば、Mavenを使えば、「mvn test」というコマンドでテストを実行できます。

注3 <http://ant.apache.org/>

注4 <http://maven.apache.org/>

注5 <http://www.gradle.org/>

Mavenとは？

MavenはApache Foundationが提供するJava用のビルドツールです。

Mavenが登場するまでは、Javaのビルドツールと言えば、Antが主流でした。現在ではMavenを採用するプロジェクトが多くなっています。理由として、MavenではAntに欠けていたパッケージ管理のシステムが提供され依存ライブラリの管理などが楽に行えること、オープンソースとして作成したプロジェクトなども同じしきみの中で公開できることなどがあります^{注6}。また、プロジェクト構成などのデフォルト設定を提供しているため、初めて触るプロジェクトでも構成を理解しやすいというメリットもあります。

一方で、XMLの設定ファイルの記述が面倒であること、プラグインの設定などに癖があるため習得が難しいことなどがデメリットです。しかしながら、Mavenを習得することでプロジェクトの構成管理は大きく改善されるため、Javaの開発では必須ツールと言えます。

本書では2012年7月時点でのMavenの最新バージョンであるMaven 3.0.4を使っています。なお、本書では主にユニットテストに関連する部分を中心いてMavenの使い方を解説します。さらに詳しく学習したい方は、専門に扱った書籍やWebを検索してください^{注7}。

注6 現在はApache Ivy(<http://ant.apache.org/ivy/>)を使うことでAntでも依存ライブラリを管理できます。

注7 やや古いますが、野瀬直樹、横田健彦著『Apache Maven 2.0入門』技術評論社(2006年)など。

Column

Mavenのバージョン

Mavenはバージョンによって差異が大きいツールです。本書ではMaven 3を扱いますが、執筆時点でWebや書籍で入手できる情報はMaven 2に関するもののがほとんどです。しかしながら、Maven 3は、Maven 2との互換性を持って移行されているため、ほとんどの情報はそのまま利用できるでしょう。

とはいえ、Maven 3では、設定ファイルの記述がより厳密になっているため、そのままでは警告が出る場合もあります。また、プラグインによってはMaven 3に対応していないプラグインやMaven 3でなければ動かないプラグインもあります。

なお、Maven 1は、Maven 2やMaven 3としきみも大きく異なり、互換性もありません。

Mavenの準備

Eclipseでは、バージョン3.7(Indigo)よりデフォルトでMavenのプラグインが提供され、標準で組み込まれるようになりました^{注8}。したがって、EclipseからMavenを実行するならば、インストール作業は不要です。設定ファイルの編集やプロジェクトの依存ライブラリの管理もEclipse上から行えます。なお、Eclipse 3.7に標準で組み込まれているMavenのバージョンは3.0.2です。

しかしながら、Mavenは基本的にコマンドラインから利用するツールです。コマンドラインから利用できるように、公式サイト^{注9}からダウンロードしたアーカイブを適当なディレクトリに解凍し、パスを通してください。

MavenプロジェクトとPOM

MavenもEclipseと同様、ソースコードやリソースをプロジェクト単位で管理します。

Mavenプロジェクトでは、プロジェクトを構成するルートディレクトリにXMLファイルを配置し、プロジェクトの構成を定義します。このXMLファイルは、**POM**(*Project Object Model*)と呼ばれ、通常は「pom.xml」というファイル名とします。

プロジェクト構成を設定したPOMを追加すれば、どんなプロジェクトでもMavenを利用できます。ただし、Mavenプロジェクトの標準的なディレクトリ構成があるため、構成を合わせておくほうが便利です。

^{注8} なあ、Eclipse 4.2ではデフォルトでインストールされなくなったため、マーケットプレイスからインストールする必要があります。

^{注9} <http://maven.apache.org/>

Maven プロジェクトの作成

Maven プロジェクトは手動で作成することもできますが、Eclipse を利用すれば簡単に Maven プロジェクトを作成できます。メニューから[新規]—[プロジェクト...]を選択してプロジェクトの新規作成ウィザードを開き、[Maven]—[Maven プロジェクト]を選択して[次へ]をクリックしてください(図 15.2)。

プロジェクトを作成するときにアーキタイプと呼ばれるプロジェクトのひな型を利用することもできますが、本書では扱いませんので、ここでは「シンプルなプロジェクトの作成」にチェックを入れて[次へ]をクリックしてください(図 15.3)。

次の画面で入力を求められる「グループ Id」「アーティファクト Id」「バージョン」は、そのプロダクトを一意に特定するための識別子です(図 15.4)。ここでは、グループ Id に「junit.examples」、アーティファクト Id に「hellojunit」、バージョンはデフォルトの「0.0.1-SNAPSHOT」を指定してプロジェクトを作成しています。この識別子は、プロダクトを公開したり、ほ

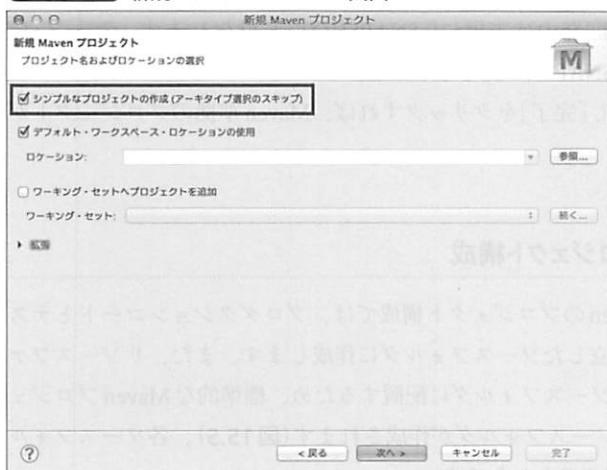
図 15.2 ウィザード選択画面



かのプロジェクトから利用したりする場合に利用します。

グループ Id (*groupId*) は、複数のプロダクトをグループ化した識別子です。Java プログラムのパッケージ名に相当し、同じグループ Id でいくつかのプロダクトを含むことができます。パッケージ名と同様にドット区切りで指定します。たとえば、「jp.deathmarch.hogehogeapps」や「junit.examples」とい

図15.3 新規 Maven プロジェクト画面



15
16
17

図15.4 新規 Maven プロジェクトの Id 設定



った名前になります。

アーティファクト Id(*artifactId*)は、そのプロダクトの識別子(名前)となります。複数のモジュールで構成されるプロジェクトでは、モジュール名となります。たとえば、「helloworld-app」や「issuetracker-client」といった名前になります。

バージョン(*version*)はそのプロダクトのバージョンを示し、一般的には「メジャーバージョン.マイナーバージョン.リビジョン」という形式で定義します。また、開発中は末尾に「-SNAPSHOT」と付与します。たとえば、「1.2.0」や「0.1.0-SNAPSHOT」となります。

これらを入力し[完了]をクリックすれば、Maven 準拠のプロジェクトが作成されます。

Maven のプロジェクト構成

標準的な Maven のプロジェクト構成では、プロダクションコードとテストコードとを独立したソースフォルダに作成します。また、リソースファイルも独立したソースフォルダに配置するため、標準的な Maven プロジェクトでは4つのソースフォルダが作成されます(図 15.5)。各ソースフォルダの用途は表 15.1 のとおりです。

なお、プロジェクトをビルドしたときに作成されるクラスファイルなどは、target フォルダの下に出力されます。Subversion や Git などのソースコード管理システムを使っている場合は、target フォルダを除外フォルダとして設定してください。

図 15.5 標準で作成されるソースフォルダ



表15.1 標準的なMavenプロジェクトのフォルダ構成

フォルダ	用途
src/main/java	プロダクションコード用のソースフォルダ
src/main/resources	成果物に含めるリソースファイル用のソースフォルダ
src/test/java	テストコード用のソースフォルダ
src/test/resources	テストで使用するリソースファイル用のソースフォルダ
target	クラスファイルやJARファイルなどの成果物やレポートの出力先

依存ライブラリの管理

Mavenにはプロジェクトに必要な依存ライブラリを管理する機能があります。依存ライブラリを追加するには、そのライブラリのグループId、アーティファクトId、バージョンを調べ、POMに設定します。これだけで、ネットワーク上にあるリポジトリからJARファイル／ソースコード／ドキュメントをダウンロードし、プロジェクトに追加できます^{注10}。また、追加したライブラリが、別のライブラリに依存する場合、その依存関係から必要なライブラリもすべてプロジェクトに追加されます。

● Mavenのリポジトリ

リポジトリとは、Mavenでライブラリ(Mavenプロジェクトの成果物)を管理するデータベースです。JUnitのような一般的なライブラリは、セントラルリポジトリと呼ばれる標準リポジトリで見つけられます。そのほかにも、サードパーティの提供するリポジトリや、社内用のライブラリなどを格納した限定的なリポジトリを利用することもできます。

Mavenでは、一度ダウンロードされたライブラリは、ローカルマシンのリポジトリに保存されます。このリポジトリはローカルリポジトリと呼ばれ、ビルド時に参照されます。

なお、Mavenではコンパイル機能やテスト機能も、すべてプラグインとして提供されています。プラグインも、初めて使用するタイミングで、リポジトリからダウンロードされます。このため、ローカルリポジトリが空の状態でビルドを行うと、多くのプラグインヒライブラリがダウンロード

注10 Mavenを利用するときは、インターネットに接続している必要があります。

され、時間がかかります。一度ダウンロードしてしまえば、2回目以降はローカルリポジトリを参照するため、ほとんど時間はかかりません。

● 依存ライブラリの追加

それでは、実際に JUnit を依存ライブラリとして追加してみましょう。JUnit を依存ライブラリとして追加するには、POM の dependencies セクションに JUnit を指定した dependency セクションを追加します。この設定は手動で XML ファイルに変更を行うことでもできますが、Eclipse を使うことでライブラリの検索と指定を簡単に行えます。

プロジェクトの pom.xml を「Maven POM エディタ」で開き^{注11} [依存関係] タブを開いて [追加] ボタンをクリックします(図 15.6)。

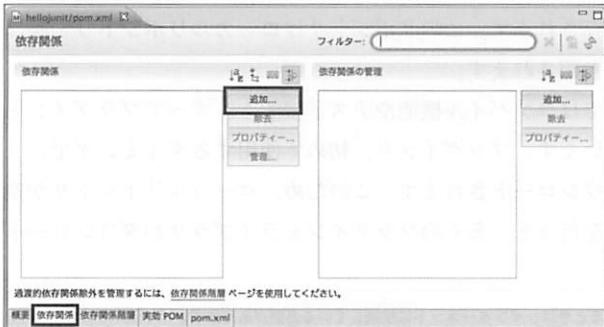
続けて「junit」を検索し、「junit junit」下にある「4.10[jar]」を選択してください。すると、グループ Id に「junit」、アーティファクト Id に「junit」、バージョンに「4.10」が入力されます。また、バージョンの横にあるスコープは「test」を選択してください(図 15.7)。

スコープには依存ライブラリを利用する範囲を指定します。JUnit はテスト時にのみ使用し、実行時には必要としないので、ここでは「test」を選択します(実行時に必要なライブラリの場合は「compile」を選択します)。

準備ができたならば[OK]を押します。すると POM にリスト 15.1 のよう

注11 Android の開発環境を導入している場合、pom.xml を開くと Android Common XML Editor がデフォルトの XML エディタとなることがあります。その場合は、コンテキストメニューから [アプリケーションから開く...] - [Maven POM エディター] を指定してください。

図 15.6 POM エディタ



な dependencies セクションが追加されます。なお、dependencies セクションの設定は手動で XML を編集して行うこともできます。

これで依存ライブラリがプロジェクトに追加されました。ただちに JAR ファイルがダウンロードされ、プロジェクトの「Maven 依存関係」に追加されます。これで、プロジェクトで依存ライブラリを使うことができます。なお、依存ライブラリがダウンロードされない場合は、プロジェクトを右クリックしてコンテキストメニューを開き、[Maven] - [依存関係の更新...] を実行してください。

依存ライブラリのソースコードやドキュメントが必要な場合は、プロジ

図15.7 依存ライブラリの追加



リスト15.1 追加されたPOMのdependenciesセクション

```
<project>
  <!-- 省略 -->
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

エクトのコンテキストメニューから、[Maven]-[ソースのダウンロード]あるいは[Javadocのダウンロード]を実行することでダウンロードできます。

ソースコードのエンコーディング設定

Mavenではソースコードのエンコーディングを、プラットフォームに依存したデフォルトエンコーディングで扱います。たとえば、Windows環境や最近のMac OS X環境ではMS932(いわゆるShift_JIS)です。しかしながら、Javaのプロジェクトでは特別な事情がない限り、UTF-8が無難です。

Mavenプロジェクトのソースコードのエンコーディングは、POMのpropertiesセクションにproject.build.sourceEncodingタグを追加して指定します(リスト15.2)。EclipseでMavenプロジェクトを扱っている場合、POMの設定を変更することでプロジェクトのエンコーディング設定も自動的に反映されます。

Mavenのプラグイン

実は、ダウンロードしたMaven本体はコア機能しか提供しません。多くの機能はプラグインによって提供されます。プラグインは、大きく次の4種類に分類されます。

- ・コンパイルやテストといったコア機能に関連するもの
- ・JARやWARの作成などパッケージングに関連するもの
- ・テスト結果やJavadocなどのレポート生成に関連するもの
- ・それ以外の拡張機能に関連するもの

リスト15.2 POMのエンコーディング設定

```
<project>
  ...
  <properties>
    <project.build.sourceEncoding>utf-8</project.build.sourceEncoding>
  </properties>
  ...
</project>
```

また、公式プラグインのほかにもサードパーティ製のツールを支援するプラグインなど、数多く公開されています。

● プラグインの設定

コンパイルを行うプラグインなど、デフォルトで使用されるプラグインについては、明示的に設定を記述する必要はありません。適切なバージョンのプラグインがデフォルト設定で使用されます。しかしながら、デフォルト設定から変更する場合は、POMのbuildセクションのpluginsまたはpluginManagementセクションに設定を追加する必要があります^{注12}。

リスト15.3では、MavenのcompilerプラグインがJava 6のVMをターゲットとしてコンパイルを行うように設定しています。Maven 3のcompilerプラグインのデフォルト設定では、Java 5(バージョン1.5)のVMをターゲットとしてコンパイルを行います。つまり、デフォルトではjavacコマンドの実行オプションでtargetとsourceに1.5を指定します。これをJava 6(バージョン1.6)をターゲットとするために、compilerプラグインのconfiguration

^{注12} pluginManagementは主に複数のモジュールで構成されるプロジェクトで子プロジェクトに設定を引き継ぐ目的で使われます。

リスト15.3 Java 6をターゲットとしたコンパイル設定

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

セクションにこのように設定を記述します。

なお、Eclipseの設定はプロジェクトを作成したときにはJava 5がターゲットとなっています。Mavenの設定を変更することで自動的にJava 6をターゲットに変更できます。変更が反映されない場合は、プロジェクトのコンテキストメニューから[Maven]-[プロジェクト構成の更新]を実行してプロジェクト設定を更新してください。

Column

プラグインのconfigurationセクション

Mavenではプラグインの設定値をPOMのconfigurationセクションに記述しますが、この設定は慣れるまでわかりにくい部分です。設定方法は、次のように configurationタグの中にプロパティ名で要素を並べ、各プロパティ要素の値としてプロパティ値を記述します。

```
<configuration>
  <property_1>value1</property_1>
  <property_2>value2</property_2>
</configuration>
```

ここで設定可能なプロパティは、プラグインのドキュメントに記述されています。たとえば、compilerプラグインならば、<http://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html>で確認できるでしょう。ドキュメントには各設定のデフォルト値や、さまざまな設定値とその説明が書かれています。これらに目を通すことで、プラグインがどのような機能を提供しているかを知ることができます。

なお、configurationセクションで複数の値を設定する場合は、次のようにプロパティを複数形とし、その中に単数形のプロパティを列挙します。

```
<configuration>
  <includes>
    <include>AAA</include>
    <include>BBB</include>
    <include>CCC</include>
  </includes>
</configuration>
```

ほかにもいろいろな記述方法があります。詳しく知りたい方はドキュメント^{注a}を参照してください。

注a <http://maven.apache.org/guides/mini/guide-configuring-plugins.html>

Mavenによるビルドの実行とフェーズ

Mavenでは、コマンドラインから mvn コマンドを使ってビルドを実行します。

たとえば、Mavenを利用してプロジェクトのテストを実行するには、プロジェクトディレクトリに移動し、次のようにコマンドを実行します。

```
$ mvn test
```

ここで、mvn コマンドの後に指定した test はフェーズと呼ばれます。フェーズとは、Maven で定義されているビルドプロセスのことです。フェーズには、clean フェーズ、compile フェーズ、test フェーズ、package フェーズ、install フェーズ、deploy フェーズなどがあります。

また、それぞれのフェーズは前提となるフェーズを持ちます。たとえば、test フェーズを指定した場合、前提となる compile フェーズが先に実行されます。そして compile フェーズでコンパイルが成功したあと、test フェーズでテストが実行されます。ただし、プロジェクトの成果物をすべて削除する clean フェーズはどのフェーズも前提条件としているため、明示的に指定しなければ実行されません。

また、mvn コマンドに複数のフェーズを指定することもできます。たとえば、clean フェーズでプロジェクトの成果物をすべて削除し、package フェーズまで実行して JAR ファイルを作成するならば、次のようにコマンドを実行します。

```
$ mvn clean package
```

このように、Maven ではコマンドラインからフェーズを指定してビルドを実行します。

● テストをスキップする

Maven のビルドプロセスでは、プロジェクトのパッケージング前に必ずユニットテストを実行します。このため、Maven でビルドされた JAR ファイルなどは、すべてのユニットテストをパスしていることが保証されます。

しかしながら、小さな修正を行ったあとにすばやくパッケージを作成す

る場合など、ユニットテストの実行を省略したい場合もあります。そのようなときは、システムプロパティ「maven.test.skip」を true に設定することで、テストプロセスをスキップできます。次のように mvn コマンドを実行します。

```
$ mvn -Dmaven.test.skip=true package
```

このように、mvn コマンドの実行オプションで「-D プロパティ名=値」と指定することで、システムプロパティの値を Maven の実行時に設定できます。「maven.test.skip」のデフォルト値は false ですが、mvn コマンドの実行オプションでデフォルトのプロパティを上書きしています。

なお、原則的には、ユニットテストの実行は省略するべきではありません。プロダクトが常にすべてのユニットテストを成功させるように維持することが重要です。

● Eclipse から実行する

Eclipse から mvn コマンドを実行する場合は、プロジェクトのコンテキストメニューから[実行]-[Maven ビルド]を選択します。実行するフェーズは、「構成の編集」ダイアログ(図 15.8)の「ゴール」に指定します(ゴールに関しては後述)。

図 15.8 mvn コマンドの設定



なお、Eclipseが利用するデフォルトのMavenは、Eclipseに組み込まれているMavenです。コマンドラインでパスが通っているMavenではないため、バージョンの違いに注意してください。Eclipseで使用するMavenを変更する場合は、Eclipseの設定画面でMavenのパスを指定します^{注13}。

プラグインとゴール

先に述べたようにMavenで実行可能な機能はプラグインで提供されます。各プラグインはさらにいくつかの関連した機能を提供しており、それらはゴールと呼ばれます。

ゴールはMavenが実行できる最小のタスクです。たとえば、compilerプラグイン(正式名はMaven Compiler Plugin)は、プロダクションコードのコンパイルを行うcompileゴールとテストコードのコンパイルを行うtestCompileゴールを持ちます。

● ゴールを実行する

`mvn`コマンドには、フェーズだけではなく、ゴールを指定することもできます。このとき、「プラグイン名:ゴール名」の形で指定します。たとえば、compilerプラグインのtestCompileゴールを実行するには、次のようにコマンドを実行します。

```
$ mvn compiler:testCompile
```

このようにプラグインとゴールを指定して`mvn`コマンドを実行した場合、指定したプラグインのゴールのみが実行されます。

フェーズは、実行するプラグインとゴールをまとめたものです。したがって、フェーズを指定して`mvn`コマンドを実行した場合、各フェーズで定義されているすべてのプラグインのゴールが順番に実行されます。

なお、フェーズとゴールを組み合わせて実行することもできます。次のコマンドでは、プロジェクトの成果物をすべて削除(cleanフェーズ)し、コンパイルとテスト(testフェーズ)、Javadocの出力(javadocプラグインの

注13 [Maven] - [インストール]から利用するMavenの設定を行えます。

javadoc ゴール)を実行しています。

```
$ mvn clean test javadoc:javadoc
```

Eclipse から Maven を実行する場合は、フェーズを指定して実行する場合と同様です。「構成の編集」ダイアログ(図 15.8)の「ゴール」に複数のフェーズとゴールを指定します。

Mavenによるカテゴリ化テスト

テストを実行する surefire プラグインは JUnit のカテゴリ化テストに対応しています。カテゴリ化テストを有効にするには、POM の configuration セクションで「groups/excludeGroups」を設定し、JUnit Provider として「surefire-junit47」を指定します^{注14}。

リスト 15.4 では、category.SlowTests カテゴリに属するテストケースをユニットテストの実行時に除外しています。excludeGroups はリスト要素のプロパティです。実行時に除外したいカテゴリを表すクラスを完全修飾名で指定してください。

続く dependencies セクションには、JUnit Provider の設定を記述します。JUnit Provider は、JUnit のテストケースを実行するプラグインで、surefire プラグインから利用されます。surefire プラグインでは、JUnit のバージョンによって、junit3、junit4、junit47 の 3 種類の JUnit Provider を使い分けます。

surefire プラグインでは、設定値によって利用する JUnit Provider を自動選択します。しかしながら、バージョン 2.11 では groups/excludeGroups に設定を行っただけでは junit4 が選択されてしまい、カテゴリ化テストが有効になりません。明示的に surefire-junit47 プラグインを指定する必要があります。詳細は surefire プラグインのドキュメント^{注15}を参照してください。

なお、Maven からカテゴリ化テストを行うときは、Maven のプラグインがテストケースを収集するため、第 10 章で紹介したテストスイートクラス

^{注14} JUnit Provider は、surefire プラグインで JUnit のテストを実行するエンジンに相当し、いくつかの種類から選択できるようになっています。

^{注15} <http://maven.apache.org/plugins/maven-surefire-plugin/>

を作成する必要はありません^{注16}。

Mavenによるカバレッジレポート

カバレッジ測定プラグインを使用すれば、Mavenによるテストの実行時にカバレッジレポートを作成できます。Mavenのカバレッジ測定プラグインの定番としてはCobertura^{注17}がありますが、Eclipseからの実行についてのサポートが弱いのが難点です。

そこで本書では、Eclipseでも Mavenでも利用可能なカバレッジ測定エン

注16 ➔ テストスイートクラスを作成する(p.166)

注17 <http://cobertura.sourceforge.net/>

リスト15.4 surefireプラグインのカテゴリ化テスト設定

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.11</version>
        <configuration>
          <excludeGroups>
            <excludeGroup>category.SlowTests</excludeGroup>
          </excludeGroups>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>org.apache.maven.surefire</groupId>
            <artifactId>surefire-junit47</artifactId>
            <version>2.11</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

15

16

17

ジン JaCoCo を使用します。JaCoCo は第 14 章で紹介した Eclipse のカバレッジ測定プラグイン EclEmma のカバレッジ測定エンジンです^{注18}。

● JaCoCo プラグインの導入

Maven で JaCoCo を利用するには、POM の plugin セクションに jacoco-maven-plugin を追加します（リスト 15.5）。本書執筆時点での最新バージョンは 0.5.7 です。現状、Maven 用のプラグインは若干の不安定さが残っていますが、Eclipse 用のプラグインは定番でかつ安定もしています。現在も早いペースで開発が進められているため、リリースごとに改善されていくでしょう。最新バージョンのリリース情報などは、公式サイトのドキュメント^{注19}を参照してください。

● JaCoCo によるカバレッジレポート

JaCoCo では、Java のクラスローディングを監視し、ダイレクトにインストゥルメンテーション^{注20}することで、カバレッジデータを記録します。このため、テストの実行前に、監視と記録を行うエージェントプログラムを起動します。そしてエージェントプログラムの起動後に、カバレッジを測

注18 ⇒ EclEmma によるカバレッジ測定（p.260）

注19 <http://www.eclemma.org/jacoco/trunk/doc/>

注20 カバレッジの測定を目的としてバイトコードを追加すること。

リスト 15.5 jacoco-maven-plugin の設定

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.5.7.201204190339</version>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

定するプログラムを実行し、最後に記録されたカバレッジデータをHTMLなどの読みやすい形で出力します。

カバレッジを測定するエージェントプログラムを起動するには、テストの実行前にprepare-agentゴールを実行します。

```
$ mvn clean jacoco:prepare-agent test
```

jacoco-maven-pluginはデフォルトのビルドプロセスに含まれていませんので、このように明示的にプラグインとゴールを指定してください。

テストが実行されると、カバレッジの測定データは、target/jacoco.execに保存されます。この測定データをHTML形式に変換するには、reportゴールを実行します。

```
$ mvn jacoco:report
```

生成されたレポートは、target/site/jacoco以下に保存されます。index.htmlをWebブラウザで開き、カバレッジレポートを確認してください(図15.9)。なお、クラス名をクリックすれば、各クラスの詳細なカバレッジが確認できます。

なお、「mvn jacoco:prepare-agent test jacoco:report」のように、一連のプロセスをまとめて実行することもできます。しかしながら、テストが失敗した場合にはその時点でビルドが中断されるため、後続のレポート作成のゴールは実行されません。テストが失敗した場合もカバレッジレポートを出力したい場合は、独立してreportゴールを実行するようにします。

図15.9 カバレッジレポート

cucumber.porker											Session 0	
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Ctry	Missed	Lines	Missed	Methods	Missed	Classes
⊕ Pat.OnePair	[progress bar]	55%	[progress bar]	50%	5	7	4	13	1	3	0	1
⊕ PokerGame	[progress bar]	76%	[progress bar]	50%	3	7	0	12	0	4	0	1
⊕ CardSuit	[progress bar]	84%	n/a		2	4	0	2	2	4	0	1
⊕ PokerGame.Status	[progress bar]	81%	n/a		2	4	0	2	2	4	0	1
⊕ Card	[progress bar]	84%	[progress bar]	67%	3	8	2	12	0	2	0	1
⊕ Hand	[progress bar]	90%	[progress bar]	75%	1	4	0	7	0	2	0	1
⊕ Pat	[progress bar]	100%	[progress bar]	100%	0	7	0	13	0	3	0	1
⊕ PatNoPair	1	100%	n/a		0	1	0	1	0	1	0	1
Total	63 of 381	83%	11 of 35	69%	16	42	6	62	5	23	0	8

Created with JaCoCo 0.5.7.201204120308

15.3 バージョン管理システムによる継続的テストの運用

継続的テストは適切なバージョン管理システムのもと、なるべく短いサイクルで運用すべきです。なぜならば、プロダクトへの修正量が少ないほうが、与える影響も少ないからです。また、問題が発生し過去のバージョンとの差分を分析して原因を見つける場合にも、変更量が少なければ少ないほど見つけやすくなり、修正コストも小さくなります。そして、問題が解決せずに元のバージョンに戻したほうがよいと判断した場合、手戻りを小さく抑えることができます。

バージョン管理システムとは？

バージョン管理システム（VCS : Version Control System）とは、プロダクトの各種リソースの履歴管理を行うためのデータベース（リポジトリ）です。バージョン管理システムの主な機能は、プロダクトに対して、いつ、誰が、どのような変更をしたかという履歴を管理することです。また、バージョン管理システムを利用することで、プロダクトをバックアップすることができます。

バージョン管理システムの歴史は古く、多くのプロダクトがあります。現在、Javaのソフトウェア開発ではSubversion^{注21}が多く利用されています。ほかにも、分散型バージョン管理システムのGit^{注22}やMercurial^{注23}なども利用されています。

バージョン管理システムへのコミット

バージョン管理システムへのコミット^{注24}は、短いサイクルで行うべきです。ただし、コンパイルが通らないソースコードやテストが失敗するソースコードはコミットすべきではありません。

注21 <http://subversion.apache.org/>

注22 <http://git-scm.com/>

注23 <http://mercurial.selenic.com/>

注24 ファイルの変更をバージョン管理システムへ登録すること。

テストの失敗が日常化してしまうと、本当に問題のあるテストの失敗は大量のテストの失敗に埋もれ、重大な問題を見落とす可能性が高くなります。このため、バージョン管理システムへコミットするときは、すべてのテストが成功するように努めなくてはなりません。

なお、具体的にどの程度のサイクルでコミットを行うかについては、さまざまな考え方があります。プロジェクトの特徴、開発者のスキル、採用しているプログラミング言語によっても変わってきます。また、なるべく短くとることは誰もが合意するところですが、プログラマにとって気分よくプログラミングできるリズムは異なるため、好みの問題でもあります。

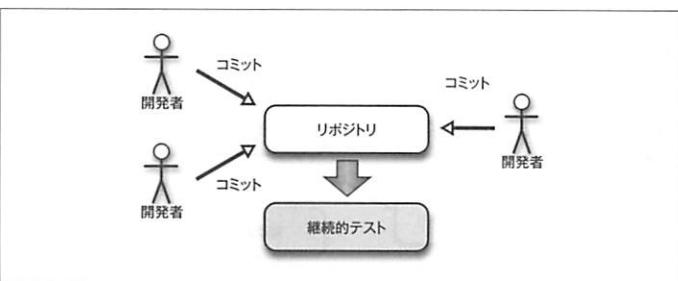
Subversionでのテストサイクル

Subversionなどの中央集権型のバージョン管理システムは、サーバに設置された1つのリポジトリでプロダクトを集中管理します。ある開発者がコミットしたならば、その時点でリポジトリのバージョンが更新されます。

Subversionを利用して継続的テストを運用するならば、コミットごとにテストを実行します(図15.10)。したがって、なるべく小さな単位で、かつテストが成功する単位でコミットします。

たとえば、クラス単位やメソッド単位で作業を行い、コミットするとよいでしょう。なぜならば、Subversionのリポジトリは単一であるため、コミットした内容がほかの開発者にも影響を与えてしまうからです。テストが成功し、かつ小さな単位でコミットを行い、ほかの開発者は細かくアップデートすることで、競合のリスクを最小限に抑えることができます。

図15.10 中央集権型のバージョン管理システム



しかしながら、大規模なリファクタリングを行うときや、複雑な機能を実装しようとするときなど、細かくコミットすることが難しい場合もあります。そのような場合は、失敗するテストにIgnoreアノテーションを付けてコミットするとよいでしょう。ただし、なるべく早く解決してください。

GitやMercurialでのテストサイクル

GitやMercurialなどの分散型バージョン管理システムは、ローカルリポジトリと呼ばれる、ローカルマシンに作成された個人用のリポジトリにコミットを行います。このため、中央集権型のバージョン管理システムに比べ、コミットのサイクルが短くできます。

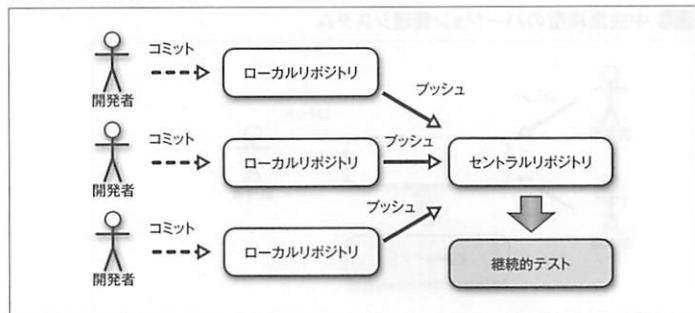
各開発者のリポジトリは、何らかのタイミングでマージされます。一定の作業単位でローカルリポジトリへの変更内容をマージし、1つのリポジトリ(セントラルリポジトリ)に反映(プッシュ)する運用が一般的です。

GitやMercurialなどを利用して継続的テストを運用するならば、セントラルリポジトリへのプッシュごとにテストを実行します(図15.11)。

プッシュを行う作業単位は、運用方針で決める必要があります。通常は、ユーザストーリー、ユースケース、機能といった、プロダクトにとって価値のある単位で行います。したがって、半日から最大で1週間程度のサイクルでプッシュされることになります。これはセントラルリポジトリにあるプロダクトを、常にリリース可能な状態に保つためです。

Subversionなどを利用した場合に比べ、継続的テストのサイクルは長く

図15.11 分散型のバージョン管理システム



なります。このため、ローカルリポジトリへのコミットごとに、ローカル環境で継続的テストを実行してもよいでしょう。

15.4 Jenkinsによる継続的インテグレーション

近年、プログラミング言語やツールといったソフトウェア開発に必要な環境は著しい進化を遂げています。JUnitなどのテスティングフレームワークはテストの自動化を、Mavenなどのビルドツールはコンパイルやテストの自動実行を可能にしています。また、ハードウェアも進化したため、ビルドやテストの実行といったコンピュータのリソースに依存するプロセスのコストも非常に小さくなりました。このため、ソフトウェア開発の作業をできる限り自動化することで、開発者はより考える仕事に注力できます。

ビルドツールとバージョン管理システムとを組み合わせることで、簡単に継続的テストの環境を構築できます。しかしながら、すばやいフィードバックを実現するためには、コンパイルやテストが失敗したときに、開発メンバーに通知するしくみも必要です。また、テスト結果、所要時間、カバレッジなどの情報を蓄積できれば、統計データとしても活用できます。

このような背景から、近年のソフトウェア開発では、「継続的インテグレーション」というプラクティスを採用するプロジェクトが増えてきました。

継続的インテグレーションとは？

継続的インテグレーション(CI : *Continuous Integration*)とは、継続的テストを含むプラクティスであり、「ビルドやテストといったソフトウェアの統合プロセスを可能な限り自動化し、継続的に行う」ためのしくみです^{注25}。

ソフトウェアはたくさんのクラスやモジュールで構成されています。このため、それらを統合(インテグレーション)するフェーズではさまざまな問題が発生します。これはソフトウェアの規模が大きければ大きいほど顕著です。そして、各クラスのユニットテストの問題よりも、いくつかのクラスに依存するテストやモジュールを結合した結果で生じる問題のほうが、

^{注25} さらに一步進め、リリースまで継続的に行う「継続的デリバリ」というプラクティスもあります。

気付きにくく早期に対応が必要です。

継続的インテグレーションでは、定期的にソフトウェアを統合／テストします。このため、繰り返してユニットテストを行うだけでは見落とされる種類の問題も、早い段階で検知できます。

継続的インテグレーションをサポートするツールでは、バージョン管理システムとビルドツールと連携し、プロジェクトの統合(コンパイル、テスト、デプロイなど)を行います。したがって、継続的インテグレーションは、継続的テストを含みます。さらに、継続的インテグレーションのツールは、メールやIRC(*Internet Relay Chat*)などを使って開発者に問題を通知する機能を持っています。また、テスト結果、所要時間、カバレッジなどもツールに蓄積されます。

継続的インテグレーションは、ユニットテストを可視化するプラクティスとも言えます。ユニットテストの件数、失敗したテストケース、実行時間などを、プロジェクトメンバーに見える形で提供します。プログラマ以外のプロジェクトメンバーにとっても状況が把握しやすくなるでしょう。

Jenkinsとは？

Javaの開発における継続的インテグレーションのツールとしては、Jenkins^{注26}がデファクトスタンダードです。Jenkinsは川口耕介氏^{注27}によって生み出されたプロダクトです。JenkinsはJavaで書かれたWebアプリケーションですが、アプリケーションサーバにデプロイしなくとも、単体で実行することもできます。

Jenkinsの導入

Jenkinsをローカル環境でスタンドアローン実行するには、公式サイト(URLは注26を参照)からWARファイルをダウンロードし、次のコマンドを実行します。

注26 <http://jenkins-ci.org/>

注27 <http://d.hatena.ne.jp/kkawa/>

```
$ java -jar jenkins.war
```

サーバが起動したならば、Web ブラウザから「<http://localhost:8080/>」でアクセスできます(図 15.12)。

このように、簡単に試せることも人気の理由です。そのほかの Jenkins の設定については、付録 E を参照してください。

なお、継続的インテグレーションや Jenkins のさまざまな機能や運用のノウハウなどは本書で紹介しきれません。日本語による入門的書籍^{注28} も多く出版されているため、詳細はそれらの書籍を参照してください。

Jenkins のジョブ

Jenkins は、プラグインを導入することでさまざまな言語や環境に対応できます。Java の開発では、Subversion などのバージョン管理システムと、Maven をビルドツールとして採用しているならば、驚くほど簡単に継続的インテグレーションを行えます。

^{注28} 川口耕介監／佐藤聖規監・著／和田貴久、河村雅人、米沢弘樹、山岸啓著『Jenkins 実践入門——ビルド・テスト・デプロイを自動化する技術』技術評論社(2011年)など。

図 15.12 Jenkins



Jenkinsでは、プロジェクトのビルドなどを行う一連のプロセスを、ジョブと呼ばれる単位で管理しています。典型的なジョブでは次のような流れでビルドやテストを実行します。

- ①バージョン管理システムから最新のソースコードを取得する
- ②ソースコードをコンパイルする
- ③ユニットテストを実行する
- ④JARファイルやWARファイルを作成する
- ⑤ステージング環境やプロダクション環境にデプロイする

このうち②から⑤までは、Mavenによって実行されるビルドプロセスです。したがって、Jenkinsに必要な設定はそれほど多くはありません。バージョン管理システムのリポジトリ、実行する Maven のコマンド、ジョブを実行するトリガー、ビルトが失敗した場合の通知方法などです。

ジョブの作成

ジョブを作成するには、Jenkinsのホーム画面から「新規ジョブ作成」のリンクをクリックします。次に、ジョブ名を入力し、プロジェクトの種類を選択します。本書では Maven 3 を利用するため、「Maven 2/3 プロジェクトのビルト」を選択してください(図 15.13)^{注29}。

ジョブの設定

続けてジョブの設定を行います。Maven 2/3 プロジェクトでは、プロジェクトのルートディレクトリに pom.xml があるならば、特別な設定は必要ありません。必要な設定は、バージョン管理システムの種類を選択しリポジトリ URL を設定することと、ビルトのゴールを設定することです。

本書では、バージョン管理システムに Mercurial を利用しているという前提で解説を行います。ほかのバージョン管理システムを利用している場合は、適宜読み替えてください。

^{注29} 「フリースタイルプロジェクト」でも Maven を利用できます。

● バージョン管理システムの設定

「ソースコード管理システム」でMercurialを選択し、「Repository URL」にプロジェクトのルートディレクトリの絶対パスを入力してください(図15.14)。サーバ上にリポジトリがあるのであれば、httpやhttpsなどのプロトコルで指定します。

● ビルドゴールの設定

次に「ビルド」の「ゴールとオプション」で、Mavenで実行するゴールまたはフェーズを指定します(図15.15)。たとえば、コンパイルとテストを行い、JARファイルなどのパッケージを作成するならば、「package」と設定します。

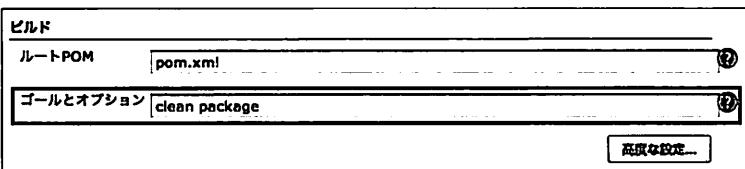
図15.13 ジョブの作成



図15.14 バージョン管理システムの設定

ソースコード管理システム	
<input type="radio"/> CVS	
<input type="radio"/> Git	
<input checked="" type="radio"/> Mercurial	
Mercurial Version	(Default)
Repository URL	/Users/shuji/Projects/practice/junit/hellojunit
Branch	

図15.15 ビルドゴールの設定



なお、Jenkinsで利用するワークスペースはジョブごとに作成され、2回目のジョブの実行で再利用されます。このため、クラスを削除したときなどに、古いバージョンのクラスファイルが残ってしまいビルドが予期せぬ形で失敗する場合があります。このため、「ゴールとオプション」の設定では、「clean package」のように成果物を削除してからビルドを実行することを推奨します。

● ジョブの実行

これだけでJenkinsの設定は完了です。ジョブの「ビルド実行」を選択すれば、バージョン管理システムから最新のソースコードを取得し、Mavenによるビルドが実行されます。

ビルドトリガーの設定

ジョブを作成し、手動で実行できたならば、バージョン管理システムに変更があったタイミングでビルドが行われるように設定します。なぜならば、ソースコードに変更が行われてからテストが実行されるまでの時間が短ければ短いほど、問題があったときに早く気付くことができるからです。

このしくみは、Jenkinsのリモートビルド機能と、バージョン管理システムのフック機能を利用して設定します。

● Jenkinsのリモートビルド機能

Jenkinsのリモートビルド機能は、ジョブのビルド用URLにアクセスすることでビルドを実行する機能です。

リモートビルドを行うには、「`http://<HOST_NAME>/job/<JOB_NAME>/build`」にアクセスします。たとえば、ローカルホストのジョブ「junit-tutorial」

をビルドするならば、「<http://localhost:8080/job/junit-tutorial/build>」へアクセスします。

なお、リモートビルド機能は認証トークンを知らなければ実行できないように設定することもできます。Jenkinsの設定でセキュリティを有効化し、ビルドトリガでジョブごとに認証トークンを設定してください。

● バージョン管理システムのフック機能

バージョン管理システムのフック機能は、バージョン管理システムごとに設定が異なります。

Mercurialでコミット時に実行するならば、次のように .hg/hgrc に記述します。

```
[hooks]
commit = wget -O /dev/null "http://localhost:8080/job/junit-tutorial/build"
```

プッシュ時に実行するならば commit を incoming としてください。なお、スクリプトを直接記述せずに、実行権限のあるスクリプトファイルを指定することもできます。

Gitでコミット時に実行するならば、次のように .git/hooks/post-commit にスクリプトを記述します。

```
#!/bin/sh
wget -O /dev/null "http://localhost:8080/job/junit-tutorial/build"
```

プッシュ時に実行するならば post-commit ファイルではなく、post-update ファイルに記述してください。

フック機能の詳細はそれぞれのバージョン管理システムのマニュアルなどを参照してください。なお、Windows環境の場合は、何らかの手段でコマンドラインから HTTP アクセス可能な環境を用意してください^{注30}。

15
16
17

^{注30} Windows 版の wget (<http://gnuwin32.sourceforge.net/packages/wget.htm>) をインストールするか、Python や Rubyなどのスクリプト言語をインストールするなどの方法があります。

継続的インテグレーションの効果

現在、ソフトウェア開発における継続的インテグレーションは標準的な技術となりつつあります。Javaの開発を行うのであれば、Jenkinsは必須のツールと言えるでしょう。

本章の最後では、継続的インテグレーションを導入することによって、ユニットテストの効果が最大化する理由を紹介します。

● テストの実行コスト削減

開発の初期段階では作成されるユニットテストの総数も多くありません。したがって、各開発者の開発環境ですべてのユニットテストを実行し、成功を確認してからソースコードをコミットする運用としても問題はありません。

しかしながら、開発が進み、ユニットテストの総数が増えてくれば、すべてのユニットテストを実行するために長い時間がかかるようになります。テストの実行が終わるまで待つ時間も、累積していくと大きなコストとなります。

「自動的にテストを実行し、問題があったならば通知してくれる」ならば、それらのコストは限りなくゼロにすることができます。また、開発者はテストの実行を裏方に任せることができるため、プログラミングに集中できます^{注31}。

● すばやいフィードバック

Jenkinsを導入し、継続的インテグレーションの環境を整備すれば、ソースコードのコミットごとにテストが実行されます。開発者はユニットテストの実行という面倒な作業から解放されます。問題があった場合にも、すばやいフィードバックが保証されます。

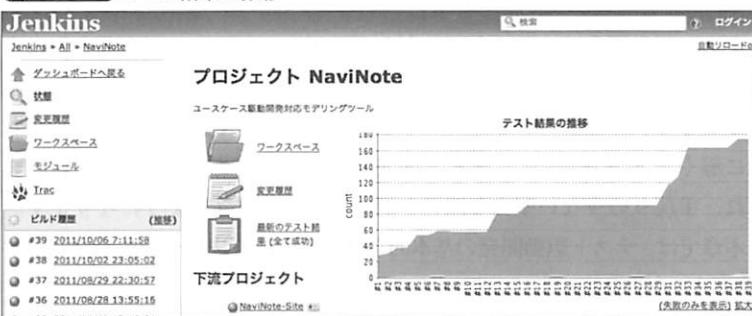
● テスト結果の履歴

Jenkinsにはテスト結果が履歴として蓄積されていきます。コミットごと

^{注31} テストが失敗すると、親しみを込めて「jenkins様がお怒りじゃ……」といった会話が開発チームで交わされ、コミュニケーションが促進される効果もあります。

にテストが実行され、何件のテストを行い何件のテストが成功したかがグラフとして表示されるため、開発者のモチベーションも自然と高くなります(図15.16)。

図15.16 テスト結果の推移



Column

失敗するテストの扱い

Jenkinsを導入するかしないかにかかわらず、ユニットテストが失敗する状況は可能な限り避けるべきです。一時的にテストが失敗することはやむを得ませんが、何日も失敗したテストがある状況は好ましくありません。

「割窓理論」^{注b}で言われているように、ユニットテストの成功を維持していれば、自然と成功を維持するような力が働きます。しかしながら、ユニットテストが1件でも失敗している状況が続くと、次第に失敗するテストが増えても誰も気にしなくなります。また、小さなバグをすぐに修正するのであれば労力も少なくて済みますが、長い間放置されたバグを修正するには大きな労力が必要となります。したがって、常にテストが成功することを維持し、壊れたならば可能な限り早く修復するようしてください。

なお、どうしてもテストが通らない場合は、Ignoreアノテーションを使いテストをスキップするほうがよいでしょう。もちろん、スキップしたテストも可能な限り早く解決するべきです。

^{注b} アメリカの犯罪学者George L. Kellingが考案した理論で、「建物の窓が破壊されたまま放置すると、誰も注意を払っていないという象徴となり、建物全体が破壊されていく」という考え方です。逆に建物の窓が破壊されてもすぐに修繕することで建物を健全な状態に保つことができます。ソフトウェア開発では、ソースコードを一部でも汚いままで放置しておくと、全体がメンテナンスできなくなるほど破壊させていくことに対してよく用いられます。

第16章

テスト駆動開発

テストファーストで設計する

テスト駆動開発は、テストコードをプロダクションコードよりも先に書くを中心としたソフトウェアの開発技法です。テスト駆動開発を実践することで、プロダクションコードをテストビリティの高いシンプルな設計に導くことができます。また、開発プロセスにユニットテストが組み込まれ、手戻りの少ない堅実な開発を進めるができるようになります。

本章では、テスト駆動開発の基本的な概念と実践方法について紹介します。

16.1 テスト駆動開発とは？

テスト駆動開発(TDD：*Test Driven Development*)は、ユニットテストをプロダクションコードよりも先に記述することを原則とした開発手法です。この原則はテストファースト(*test first*)と呼ばれ、XP^{注1}で紹介されたプラクティスのひとつです。その中心となる技術はユニットテストであり、ユニットテストをより効果的に開発に取り込む開発手法と考えることができます。

テストファーストを行うことは、テスト対象のクラスやメソッドを実装するよりも先に、どのようなテストが成功すれば目的のものが得られるか、というゴールを先に明確にするということです。ゴールが先に、そして明確に決めることができれば、最短距離で目的を達成できます。

また、テスト駆動開発は短いサイクルを繰り返して開発を進める手法です。1回のサイクルでは、ゴールをなるべく近い場所に設定します。短い距離をゴールまで進むサイクルをリズムよく繰り返し、最終的なゴールを目指します^{注2}。

注1 eXtreme Programming(エクストリームプログラミング)。Kent Beckらによって提唱されているソフトウェア開発手法。テストファーストのほかにも、リファクタリングや継続的インテグレーションなどのいくつかのプラクティスを含みます。

注2 テスト駆動開発の詳細については次の書籍を参照ください。Kent Beck著／長瀬嘉秀、テクノロジックアート訳『テスト駆動開発入門』ピアソンエデュケーション(2003年)。

16.2 テスト駆動開発のサイクル

テスト駆動開発は次のサイクルで行います(図16.1)。

- ①設計する
- ②テストコードを書く(レッド：テスト失敗)
- ③プロダクションコードを書く(グリーン：テスト成功)
- ④リファクタリングする(グリーン：テスト成功)

このサイクルは、レッド—グリーン—リファクタリングと呼ばれます。

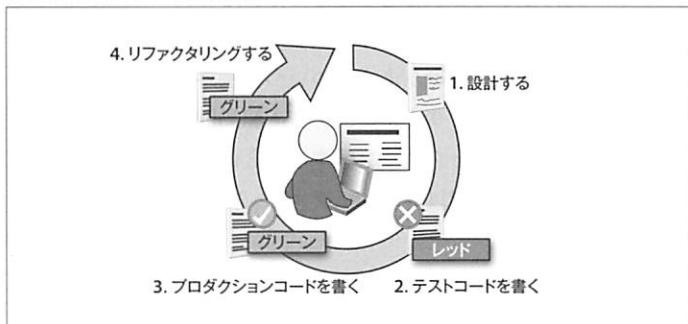
設計する

テスト駆動開発は、最初にテストコードを書く、すなわちテストファーストが起点です。しかし、何もないところからいきなりテストを書くことはできませんので、何をテストしたらよいかがわかるまで設計を行います。

設計は、主にインターフェースについて行います。ここでいうインターフェースとは、Java言語仕様におけるインターフェースクラスではなく、クラスやメソッドの外部的な仕様です。すなわち、どのような入力(引数)を持ち、どのような副作用があり、どのような出力(戻り値)を返すかを設計します。このとき、具体的な例をリストアップするとスムーズにテストを書くことができます。

ここで重要なのは「テストすべきことがわかるまで設計する」ということ

図16.1 テスト駆動開発



です。実装の詳細や細かい仕様を決めるわけではありません。なぜならば、実装の詳細や細かい仕様は作ってみなければわからないことが多く、少なくとも最初から完璧に洗い出すことは不可能だからです。解決しなければならない問題が大きすぎるならば細かく分割し、解決できる大きさにします。

テストコードを書く

何をテストすべきかがわかったならば、動作するドキュメントとしてテストコードを作成します。ただし、1回のサイクルで作成するテストケースは原則として1つです。

本書では、テストコードの書き方についてさまざまなテクニックやノウハウを紹介してきました。それらを利用して、プロダクションコードがあるという前提で、テストしたいこと、すなわち何を満たせばよいかを考えてテストコードに落とし込みます。

プロダクションコードを書く

テストコードが作成できたならば、プロダクションコードを作成します。まずはコンパイルエラーを解決しましょう。コンパイルエラーが解決したならば、テストコードを実行できるようになります。この時点では、プロダクションコードに手を入れていないため、テストはレッドとなるはずです。もし、予想に反してテストが成功してしまったならば、テストコードに問題があるかもしれません。

テストコードが実行できるようになったならば、そのテストがグリーンになるようにプロダクションコードを作成します。このとき、テストをグリーンにするために最短距離を走ります。ついつい例外処理などの気になっていた処理をここで実装したくなりますが、テスト駆動開発ではそのような実装は我慢し、最速でテストをグリーンとすることに集中します。

例外処理などの実装は、次回以降のサイクルで、ユニットテストを作成したあとに実装します^{注3}。

注3 メモ用紙やテキストファイルなどに次に行うテストとして書いておくとよいでしょう。

リファクタリングを行う

最後にリファクタリングを行います。

ユニットテストがグリーンとなったので、必要なプロダクションコードはすでに実装できています。ここでプロダクションコードに手を入れるのは無駄にも思えますが、テスト駆動開発ではリファクタリングすることが重要です。

「動いているコードに手を入れてはならない」と教えられた人もいると思いますが、安心してください。テストがグリーンを維持するのであれば、プロダクションコードにどれだけ手を入れても、動作は保証されているのです。将来のため、プロダクションコードを常にきれいなものとします。

次のサイクルを始める

これで、テスト駆動開発の1サイクルは終了です。テスト駆動開発では、このサイクルをリズムよく、早く回すことが重要です。

このサイクルは可能な限り短く行うことが大切です。なぜならば、1つのサイクルが短ければ短いほど、ミスを犯した場合の手戻りが少なくなるからです。短ければ短いほど良いですが、目安としては10分から15分を上限とするとよいでしょう(ただし、慣れるまでは時間がかかります)。

なお、これらのサイクルをどれだけ繰り返せばよいか、という問い合わせに正解はありません。テスト対象に対し不安がなくなるまで行うべきですが、コスト的な問題もあるため、どこかで止める決断をする必要があります。

16.3 Calculatorクラスのテスト駆動開発

テスト駆動開発は、文章で説明されてもなかなか伝わらない開発手法です。そこで、本書で何度も登場しているCalculatorクラスをテスト駆動開発で実装する手順を解説します。テスト駆動開発について初めて触れる方は、実際にソースコードを書きながら本節を読み進めることをお勧めします。

テストファースト

テストファーストは、テスト駆動開発の根幹となるプラクティスです。プロダクションコードよりも先にテストコードを書くため、ほとんどの場合はコンパイルエラーとなるでしょう。慣れるまでは違和感がありますが、テストファーストでテストケースを考えることに価値があります。

ここでは、「計算を行うクラスで加算をするメソッドの作成」を最初のゴールとします。

● テストクラスの作成

はじめにテスト対象となるクラスの名前を決めます。クラス名を Calculator と定義したならば、対応するテストクラスは CalculatorTest となります。ここで Calculator クラスよりも先に CalculatorTest クラスを作成します。

```
public class CalculatorTest {  
}
```

● 設計とテストケースの作成

次にテスト対象のメソッドの名前を考えます。加算メソッドですから、add という名前とします。

続けて add メソッドの仕様、すなわち戻り値や引数、例外などを検討し、テストクラスにテストケースを追加します。このとき、どんな入力値と期待値があるのかを具体的に考え、テストメソッド名に反映してください。

```
public class CalculatorTest {  
    @Test  
    public void addは3と4で7を返す() throws Exception {  
    }  
}
```

このような設計は、テスト対象がシンプルであれば、テストコードを書きながら行うことができます。

テスト対象が複雑であれば、最初にホワイトボードなどを使いクラス設計を行うとよいでしょう。クラス間の依存関係やインターフェースを先に設計し、詳細な仕様は各クラスのテストコード作成時に考えます。

● テストケースの実装

設計ができたならば、テストコードを書いていきます。テストコードを書く段階では、プロダクションコードが存在しないため、Eclipseのコード補完機能を頼ることができません。しかし、ここは我慢して、どんなインターフェースでメソッドを定義すれば使いやすいか、ということに集中します。そして、実際に使ったならばどうだろうと考えながらテストコードを書きます。

```
public class CalculatorTest {
    @Test
    public void addは3と4で7を返す() throws Exception {
        Calculator sut = new Calculator();
        assertThat(sut.add(3, 4), is(7));
    }
}
```

ここで、作成しようとしているメソッドの仕様が不明確であったり、使いにくいく感じたりしたならば、メソッドのシグニチャを再検討するチャンスです。実装者が使いにくいと思うメソッドは、利用者にとっても使いにくいことは間違ありません^{注4}。プロダクションコードはまだ作成していないため、作りなおすという抵抗もありません。

addメソッドでは、int型の引数を2つ持ち、int型で戻り値を返す仕様としました。たとえば、3と4をメソッドに与えると合計値である7を返します。これでCalculatorのインターフェースが明確になりました。

テストファーストに慣れるまでは非常に難しいと感じます。はじめは何をしたらよいかわかりません。また、コード補完が機能しないため、イライラすることもあります。しかしながら、テスト駆動開発を学び、多くの経験を蓄積すると、設計力が高まり、テストファーストも自然に感じられるようになります。

^{注4} 製品をリリースする前に自分たちで使え、という意味で「ドックフードを食え」(Eat Your Own Dog Food.)と呼ばれます。

コンパイルエラーの解決

テストコードを記述したならば、可能な限り早くグリーンにすることを目指します。そのためには、はじめにテストコードを実行できるようにコンパイルエラーを解決しなければなりません。

Javaでは強力な型システムとコンパイラの文法チェックにより、コンパイルが通ることで多くの問題を解決できます。さらに、Eclipseのクイックフィックス機能を使えば、IDEの誘導に従ってコマンドを選択していくだけで多くの問題は解決します。

Column

テスタビリティとクラス設計

ソフトウェア開発でユニットテストを導入する場合、特にテスト駆動開発でテストファーストを実践する場合には、テスタビリティ^{注a}を意識したクラス設計を行うことが重要です。テスタビリティを意識した設計を行うことで、プロダクションコードもテストコードもより読みやすく、メンテナンス性の高いコードとなります。

- いくつもの機能を実現する1つのメソッドをテストするには、数多くのテストケースが必要となり複雑になる。一方、それぞれの機能を個別のメソッドとしたならば、テストのコンテキストが絞られ、個々のテストは書きやすくなる
- 多くの状態を持ち、各メソッドが副作用を持つクラスはテストが複雑になる。一方、状態を持たないクラスの副作用を持たないメソッドのテストはシンプルである
- 多くの責務を持ち、依存するクラスの状態によって振る舞いが変わらるようなクラスはテストが複雑になる。一方、適切な責務を持ち依存関係が疎のクラスであればテストが書きやすくなる
- 継承を多用したクラスはモックなどの代役オブジェクトに置き換えるにくく、テストが難しくなる。一方、委譲により責務を分離した設計であればモックなどのテストダブルに置き換えやすく、テストが簡単になる

このようにテスタビリティを高くすることは、一般的なオブジェクト指向プログラミングにおける推奨されるクラス設計を促進します。したがって、先にテストコードを書き、テストがやりにくいと感じたならばクラス設計に問題があると気付くことができます。

テストファーストを実践していないとも、クラス設計で悩んだときは、「どうやってユニットテストするか?」を意識すると、良い設計を行うヒントとなります。

注a ➔テスタビリティを高めるリファクタリング(p.172)

● クイックフィックスの活用

さっそく、コンパイルエラーだらけの CalculatorTest からコンパイルエラーを解決していきましょう。まず、ソースコードの先頭にカーソルがある状態で、**[Ctrl]+[.]**(ピリオド) を押してください。一番近いエラーまでカーソルが移動します。続けて、**[Ctrl]+[1]** を押します。すると、いくつかのコマンドがポップアップメニューで表示されます(図 16.2)。

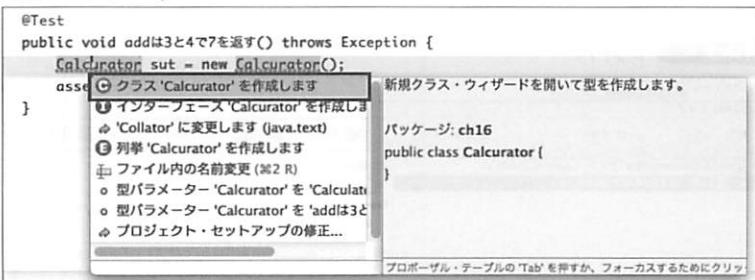
これはクイックフィックスと呼ばれる Eclipse の機能で、プログラマが次に行うべきことを、Eclipse が提案してくれます。図 16.2 では、クラスが見つからないことでコンパイルエラーとなっているため、クラスを新規に作成するか、ほかの似たクラスで置き換えるかを提案しています。なお、エラーがない場合でもクイックフィックスは有効ですが、コンパイルエラーに対するクイックフィックスは特に精度が高くなります。

エラーへのジャンプとクイックフィックスを繰り返していくば、次のように Calculator クラスが作成され、コンパイルエラーは解消されます。

```
public class Calculator {
    public Integer add(int i, int j) {
        // TODO 自動生成されたメソッド・スタブ
        return null;
    }
}
```

しかし、このままでは設計意図に反する部分があるため、次のようにメソッドの戻り値型と戻り値を修正します。

図 16.2 Eclipse のクイックフィックス



```
public int add(int i, int j) {
    // TODO 自動生成されたメソッド・スタブ
    return 0;
}
```

この段階で一度テストを実行してください。テストが失敗し、レッドバーが表示されます(図16.3)。

このように、コンパイラとIDEの強力なサポートを得られることはJavaによるテスト駆動開発の大きな特徴です。Javaは記述の冗長性や強い型付けから苦手とするプログラマも多いプログラミング言語ですが、味方にてしまえばこれほど心強いものはありません。あなたのタイプミスやケアレスミスを簡単に検知してくれるでしょう。

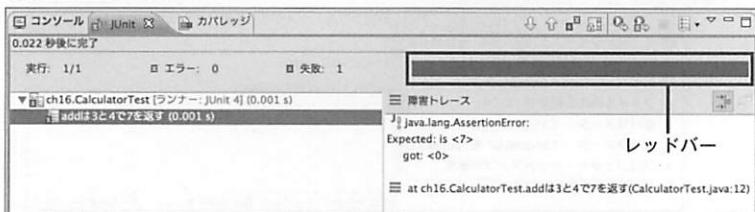
仮実装

コンパイルエラーは解決しましたが、テストはレッドです。レッドバーはプログラマにとって不安な状態のため、最速でグリーンバーにする必要があります。なぜわざわざ「最速で」なのかというと、一度にいろいろなことを考えるよりもシンプルで確実だからです。「まずはグリーンバーにしてから難しいことを考える」のです。

先ほどの図16.3のエラーメッセージからわかるのは、「7を期待しているところで0が取得された」ということです。実装を確認すると、Eclipseによって自動的に作成されたメソッドが仮の値として0を返していることがわかります。

ここで「 $i+j$ 」と変更したい気持ちを抑え、0を7に変更します。また、TODOコメントも削除します。

図16.3 レッドバー



```
public class Calculator {
    public int add(int i, int j) {
        return 7;
    }
}
```

修正を行ったならば、すぐにテストを実行します。期待どおりにテストは成功し、グリーンバーとなります(図16.4)。

このような暫定的に定数を返す実装は**仮実装(fake it)**と呼ばれます。

● 仮実装の目的

仮実装は茶番に思えますが、テスト駆動開発の基本的なテクニックです。

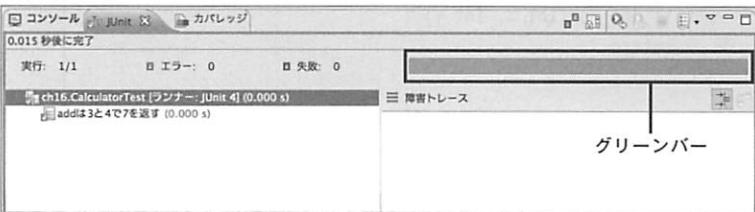
仮実装を行う目的は、最速でテストをグリーンバーとすることです。なぜならば、プログラマが不安と感じるレッドバーの状態が長く続くと、テスト駆動開発のリズムが乱れるからです。

よくある話ですが、簡単と思って実装してみたところ、期待どおりに動作せず、テストがうまくグリーンバーとならないことがあります。たいていはちょっとした勘違いやコーディングミスが原因ですが、一度そのような状況に陥ると、原因が見つからずに焦り、さらに原因が見つからないという悪循環に陥ります^{注5}。このような焦りや不安からリズムが崩れるよりも、確実にひとつずつ、リズムよく進めるための一手法が仮実装です。

Calculatorクラスにおけるメソッドの仕様は単純であるため、実装時に考慮することは多くありません。しかしながら、実際の開発ではパラメータ妥当性や例外処理など、実装時に多くのことを考慮しがちです。このた

^{注5} このような泥沼状態のときは、翌日の朝にチェックしてみると一瞬で解決することが多いです。

図16.4 グリーンバー



め、最速でグリーンバーにするという目的を忘れてしまいます。はじめに定数を返すだけの最もシンプルな実装をすると決めれば、多くのことを考慮するという誘惑から逃れられます。

なお、仮実装は常に行わなければならないテクニックではありません。今回の例のようにシンプルで、自信を持って実装できるような場合ならば、はじめから本実装をしてもかまいません。しかしながら、本実装してみて思ったようにうまくいかず、自信が揺らいだならば、すぐさま仮実装を行うとよいでしょう。

テスト駆動開発では、リズムが重要です。仮実装は、不安を感じるときに、ギアを落として確実にクリアするためのテクニックです。

リファクタリング

仮実装を行うことでテストがグリーンとなり、プロダクションコードは期待する振る舞いをするようになりました。しかしながら、テストがグリーンとしただけでは、テスト駆動開発として十分ではありません。リファクタリングにより、クラスやメソッドの外部的な振る舞いを変えずに、内部構造の変更を行います。

Calculatorクラスのaddメソッドにおける外部的振る舞いとは、「addメソッドが3と4に対して7を返す」ことです。テスト駆動開発では、その振る舞いはテストで保証されるため、テストがグリーンとなることを維持すれば、どれだけプロダクションコードを修正しても安心です。

Calculatorクラスのプロダクションコードは単純であるため、リファクタリングできる部分はほとんどありません。ここでは、数学的な慣習に従い、変数名(i, j)を(x, y)に変更しました。

```
public class Calculator {
    public int add(int x, int y) {
        return 7;
    }
}
```

リファクタリングを行ったならば、すぐにテストを実行します。グリーンを維持しているならば、「外部的な振る舞いが変わっていない」ことが保

証されます。不幸にもレッドとなり、外部的振る舞いを変えてしまったならば、ソースコードをもとの状態に戻すか、問題となる個所を修正します。

●リファクタリングの目的

テスト駆動開発では、最初にテストケースを作成し(テストファースト)、テストを成功させるためにプロダクションコードを最も単純な方法で実装(仮実装など)します。似たようなメソッドを作成するのであれば、元のプロダクションコードをコピー&ペーストしてもかまいません。重要なことは、最も単純な方法で実装し、少しでも早くテストをグリーンとすることだからです。

しかしながら、仮実装やコピー&ペーストしたコードは重複が多く、きれいなコードではありません。そのままの状態で放置してしまうと不具合の温床となります。そこで、グリーンにしたあとに、重複を排除するなどの変更をプロダクションコードに行います。

このような汚いコードや重複の多いコードは**技術的負債**と呼ばれます。技術的負債は、時間が経つにつれ、利息を増やしながら開発を圧迫します。このため、早い段階で減らしておくことが重要です。

テスト駆動開発では、リファクタリングが開発サイクルに組み込まれています。最初にテストコードを書き、そのテストをグリーンにすることに集中し、最後にコードをきれいにするのです。つまり、実装時に技術的負債を作ったとしても、リファクタリングですぐに返済します。これにより、常にプロダクションコードを動作するきれいなコードに導きます。

三角測量

テスト対象メソッドに対するテスト駆動開発の1サイクルが完了した時点では実装が不完全です。このため、すぐに次のサイクルを開始します。

2回目のサイクルでは、同じテスト対象メソッドに対して追加のテストケースを作成します。その際、1回目で作成したテストケースとは異なる値でテストケースを作成してください。つまり、仮実装で成功するテストケースに加え、異なる値によるテストケースを作成します。

このように、テスト対象メソッドに対して2つの観測地点を用いること

で信頼できるプロダクションコードを導くテクニックは、**三角測量 (triangulate)**と呼ばれます。三角測量は仮実装と対になって行うテスト駆動開発の重要なテクニックです。

● 計測点の追加

2つ目のテストケースでは、1つ目のテストケースとは可能な限り直交するような値を選択します^{注6}。たとえば、加算メソッドのテストの入力値として1つ目のテストケースで3と4を選択した場合、2つ目のテストケースとして5と1の加算を選択します。

```
@Test
public void addは5と1で6を返す() throws Exception {
    Calculator sut = new Calculator();
    assertThat(sut.add(5, 1), is(6));
}
```

4と3や2と5では仮実装のままでもグリーンとなるため、2つ目のテストケースの値として不適切な値の組み合わせです。1つ目のテストケースの入力値に含まれる3と4、期待値である7は避けるほうが効果的です。

テストを書いたならば、すぐにテストを実行し、レッドバーを確認します。これは些細なことと感じるかもしれませんが、大切な手順です。テスト駆動開発を続けていると、予期せずにテストがグリーンとなることがあります。その場合の原因は、テストコードに問題があることがほとんどです。レッドとなることが期待されているときに、確実にレッドとなるかを確認していくことで、大きな安心が生まれます。

なお、テスト対象によっては2つだけではなく、より多くの測量点が必要です。たとえば、リストなどの集合に対するテストでは、0個、1個、2個の3つの測量点を使います。0個は初期状態という特殊な状況です。最も単純な状況は1個です。そして、最も単純で複数ある状況は2個です。この3つの状況でテストすれば、感覚的にも安心できるでしょう。

逆に、3個以上の測量点についてのテストを実施したとしても、不具合を検知する可能性はありません。すなわち、テストケースの追

注6 この例だと1つ目に使った3と4、および合計が7となる数字の組み合わせは避けます。

加コストに対する不具合の検知率が低くなります。

ソフトウェア開発では、完璧なテストを行うことも完璧なソフトウェアを作ることも不可能です。したがって、十分に良いソフトウェアを作ることを目標に、プログラマが安心して開発を進めることのできる、可能な限り少ないテストケースを選択することが重要です。どのような値をテストデータとして選択すると効果的かは、テスト技法の書籍^{注7}などで詳しく解説されていますので、ユニットテストと併せて学習するとよいでしょう。

● プロダクションコードの修正

最後に2つのテストがどちらもグリーンとなるようにプロダクションコードを修正します。

```
public class Calculator {  
    public int add(int x, int y) {  
        return x + y;  
    }  
}
```

今回はリファクタリング可能な部分はなさそうです。最後にもう一度テストを実行し、すべてのテストがグリーンとなることを確認します。

このように、テスト駆動開発は、テスト対象となるクラスやメソッドの仕様を、ユニットテストとして先に書き、少しづつ反復的にプロダクションコードを実装する開発手法です。

テスト駆動開発を導入することで、結果的に品質は高くなります。しかし、それはユニットテストが十分に行われているからではなく、実装前に十分な設計を行っていることと、リファクタリングを行っているためです。

テスト駆動開発で作成するテストは、品質を高めるためのテストではなく、設計を駆動し、リファクタリングを行うためのテストです。

^{注7} Lee Copeland著／宗雅彦訳『はじめて学ぶソフトウェアテストの技法』日経BP社(2005年)、秋山浩一著『ソフトウェアテスト技法ドリル——テスト設計の考え方と実際』日科技連出版社(2010年)など。

16.4 テスト駆動開発の目的

テスト駆動開発を導入することにより、ソフトウェア開発のリズムは大きく変化します。しかしながら、その効果は品質に直結するわけではありません。

テスト駆動開発を行う目的は、プログラマの不安を軽減することです。本章の最後に、なぜプログラマの不安がなくなるかについてまとめます。

ステップバイステップ

テスト駆動開発の基本は「小さく、ひとつずつ」です。テスト駆動開発は、1つのテストケースを書きグリーンにするというサイクルを、リズムよく繰り返す開発手法です。このとき、自分の実力を過信して、たくさんのことと一緒にやろうとしてはいけません。

人間は完璧ではありません。まとめてたくさんのことを行おうとすれば、必ずミスを犯します。そして、そのようなミスがバグとなり、ソフトウェアに欠陥を引き起こします。

テスト駆動開発を導入し、1回のサイクルでは簡単なことを1つずつ行うのであれば、ミスを大幅に減らすことができます。また、扱う問題を小さな問題とするため難易度も下がり、思考が停止する時間を減らすことができます。

テスト駆動開発は、小さなものをひとつずつ安心して構築するための開発手法です。

自分が最初のユーザ

テスト駆動開発を導入すると、テスト対象となるクラスやメソッドをプログラマが最初に利用します。しかも、プログラマはテストコードを先に書くため、プロダクションコードを書く前に、対象となるクラスやメソッドを利用します。その時点でテスト対象となるクラスやメソッドが使いにくいのであれば、修正することができます。

このため、ユーザビリティに関して確認されたクラスやメソッドが提供

されるようになります。

動作するきれいなコード

従来の開発手法では、はじめに完璧できれいな設計を行い、その設計に従ってプロダクションコードをコーディングするという方法がとられてきました。しかしながら、実際にプログラミングしてみなければ、最適な設計や重複個所の抽出は困難です。このため、完璧な設計を行おうとして分析麻痺に陥ったり、設計どおりの実装が困難であると実装時に判明し、大幅な手戻りを招いたりといった問題が起きます。

完璧な設計を行うことはできません。できたとしても非常にコストがかかることになるでしょう。

一方、テスト駆動開発では、はじめにテストケースを作成し、動作するプロダクションコードとすることを最優先とします。動作させることができた時点ではきれいなコードではありませんが、リファクタリングを行うことで、きれいなコードとなります。

テスト駆動開発では、完璧な設計をしてから実装するのではなく、動くプロダクションコードを実装したあとで最適な設計に修正します。

すばやいフィードバック

継続的テストや継続的インテグレーションを導入しているのであれば、ユニットテストにより「すばやいフィードバック」を最大限に得ることができます。これはテスト駆動開発を導入している場合には、特に大きな効果となります。

なぜならば、プロダクションコードを実装した時点で、必ず対応するテストコードが先に作成されているからです。バージョン管理システムにコミットが行われたならば、全ユニットテストが実行され、まったく予期しない個所で副作用を誘発したとしても、即時にフィードバックを得ることができます。

テスト駆動開発と継続的テストを導入しているのであれば、すばやいフィードバックが保証されるため、プログラマは安心して開発を進めることができます。

ができます。

メンテナンスされたドキュメント

テスト駆動開発を導入しているならば、テストケースを先に書き、テストが成功する状態を保ちます。これらのテストケースはテスト対象メソッドの具体的な仕様とユースケース(利用例)です。すなわち、テストコードはメンテナンスされ動作するドキュメントでありサンプルコードです。

仕様書やドキュメントは軽視されるべきではありませんが、どうしてもメンテナンスがなおざりになる部分です。しかし、ユニットテストのコードはメンテナンスしなければユニットテストの失敗となります。このため、テスト駆動開発を導入し、テストの成功を維持することで、メンテナンスされたドキュメントを得ることができます。

第17章

振舞駆動開発

ストーリーをテスト可能にする

ユニットテストは設計を洗練させ、リファクタリングを支援し、開発者に安心を与えます。その結果としてソフトウェアの品質は高くなりますが、ソフトウェアが顧客の要求を満たしているかどうかは保証できません。ユニットテストではテスト対象の粒度が小さすぎるためです。

そこで本章では、受け入れテストとソフトウェアの期待される外部的振る舞いを定義するところから開発を始める振舞駆動開発について紹介します。

17.1 受け入れテストとは?

本書で扱ってきたユニットテストの対象はクラスやメソッドであり、これらはソフトウェアを構成する最小単位の部品です。しかしながら、どれだけユニットテストを実施したとしても、関連するクラスやメソッドが正しく結合されて実行できなければ、そのソフトウェアに価値はありません。ソフトウェアの構成要素を結合し、エンドトゥエンドで検証する必要があります。

従来の開発プロセスでは、ソフトウェアを統合した最終的なテストを開発の終盤で行います。このようなテストは、システムテスト(*system testing*)^{注1}と呼ばれます。

システムテストは、設計書や要件定義書に定義された仕様が正しくソフトウェアに実装されているかを検証するテストです。このため、要求分析や設計で、顧客の要求が適切に漏れなく反映されている場合には、システムテストの成功によって顧客の要求を満たしていることを保証したと言えます。しかしながら、自然言語で記述された設計書や要件定義書によって、完璧な要求分析や設計を事前にを行うことは困難です。

^{注1} 開発ベンダーによって呼び方は異なる場合も多いですが、ここではシステム全体を統合して行うテストの意味で用いています。

適切に顧客の要求を抽出する手法として、ユーザストーリー^{注2}やユースケースシナリオをもとにソフトウェアの外部的振る舞いを定義する方法があります。それらの外部的振る舞いをテストとして実行することにより、顧客の要求を満たしていることを保証できます。このようなテストは、ユーザ受け入れテスト(UAT : *User Acceptance Testing*)、または単に受け入れテストと呼ばれます。

受け入れテストのテストシナリオは、仕様書などに比べて顧客が理解しやすいため、本当に欲しいものがソフトウェアで実現されているかどうかを判断しやすくなります。また、早い段階でテストシナリオを顧客と合意することもできます。それにより、開発の手戻りも小さく済みます。さらには、顧客がテストシナリオを自ら書くことができれば、自分たちの要求を開発者に伝えやすくなります。

受け入れテストの自動化

受け入れテストは、すべてのモジュールやサブシステムを統合した状態で行います。受け入れテストの多くは手動で行われていますが、自動化できれば何度もテストを実行でき、リグレッションが防止できます。また、早いフィードバックがあるため、問題も早い段階で解決できます。自動化により実行コストが下がるため、これまでより短いサイクルでプロダクトをリリースできます。

自動化された受け入れテストを行うには、ユーザの理解できる言葉でテストシナリオを記述する必要があります。しかしながら、JUnitのようなJavaのプログラムでは、プログラマしか読むことができません。また、受け入れテストのテストシナリオは、プロダクションコードがない状態で書ける必要があります。このような受け入れテストを行うには、ユニットテストとは異なる特徴を持ったテスティングフレームワークが必要です。

本章では、JUnitの拡張であり、受け入れテストを支援するテスティングフレームワークである「cucumber-junit」を用いた振舞駆動開発を紹介します。cucumber-junitを利用すれば、開発の早い段階で受け入れテストを定

注2 ソフトウェアのフィーチャ(機能)を定義したもの。アジャイル開発で利用されます。

義し、自動化することができます。

17.2 振舞駆動開発とは？

振舞駆動開発(BDD: *Behaviour Driven Development*)とは、ソフトウェアの相互作用に着目し、ソフトウェアがどのように振る舞う(動作する)かを定義することを起点とした開発手法です。

ソフトウェアの振る舞いを記述するシナリオ

振舞駆動開発では、ソフトウェアがどのように振る舞うかをシナリオ(*scenario*)として記述します。シナリオとは、ユーザがソフトウェアを利用するときのユースケースや具体的な例です。ユーザが、どのような前提条件で、どのようなデータをソフトウェアに入力したならば、どのようになるべきか、を定義します。

たとえば、計算機システムであれば、次のようなシナリオが定義できます。

初期状態から、3と4を加算したならば、7を出力すべき

このようなシナリオはスペック(*spec*)または仕様とも呼ばれます。

振舞駆動開発では、このようなシナリオをソフトウェアのプロダクションコードを書く前に定義します。限りなく自然言語に近い記述ですが、フレームワークを用いて実行することができます。このシナリオが期待どおりに動作するように、ソフトウェアを開発していきます。

●「～するべき」とアサーション

振舞駆動開発はテスト駆動開発によく似ています。どちらもソフトウェアが期待した動作を検証することには変わりありません。そして、その検証を行うためのシナリオやユニットテストを先に定義することも共通しています。

振舞駆動開発におけるシナリオは、テスト駆動開発におけるテストケースに相当します。しかしながら、振舞駆動開発のシナリオでは「～するべき」(*should*)という言い回しが好まれます。これは、ユニットテストにおけ

るアサーションとは異なるニュアンスです。

アサーションは、ある程度設計されたプログラムや検証可能なプログラムに対する仕様の宣言です。一方、振舞駆動開発で使われる「～するべき」という表現は、純粹にプログラムに対する要求を表します。極端な例を挙げれば、計算機システムのシナリオは、実装がJavaであるという制約すらありません。

振舞駆動開発と受け入れテスト

振舞駆動開発では、はじめにどのような仕様が期待されるかを定義し、シナリオに記述します。たとえば、本書で後ほど取り上げる Cucumber の書式だと、プレーンテキストで次のように記述します。

```
# language: ja
フィーチャ: 計算機
  シナリオ: 加算ができる
    前提 計算機が初期状態
    もし 3と4を加算した
    ならば 7を出力すべき
```

振舞駆動開発のシナリオは、ユーザ視点でソフトウェアの期待される振る舞いを定義したものです。そのシナリオは、顧客に近い視点でソフトウェアへの要求を記述できるため、主に受け入れテストのために利用されます^{注3}。受け入れテストを起点に開発を進めるため、受け入れテスト駆動開発(ATDD: Acceptance Test Driven Development)とも呼ばれます。

また、シナリオは、Cucumberなどのフレームワークを利用することで実行できるため、受け入れテストを自動化する方法として有用です。

テスト駆動開発との違い

振舞駆動開発はテスト駆動開発はよく似た開発手法ですが、大きな違いがあります。

^{注3} ユニットテストレベルで振舞駆動開発を適用することもできます。

● テストコードと自然言語によるシナリオ

テスト駆動開発でテストファーストを行う場合は、次のようなテストコードをJavaプログラムとして作成します。

```
@Test
public void addメソッドに3と4を与えると7を返す() throws Exception {
    Calculator sut = new Calculator();
    int actual = sut.add(3, 4);
    assertThat(actual, is(7));
}
```

振舞駆動開発のシナリオとテスト駆動開発のテストコードは、どちらも検証していることは変わりません。しかしながら、シナリオは自然言語で記述されているため、Javaの言語仕様を知らない人であっても、内容を確認できます。

一方、テストコードはJavaのプログラムとして記述されているため、Javaの言語仕様とJUnitの使い方を知らなければ読むことはできません。

Column

ユースケースシナリオと振舞駆動開発のシナリオ

ユースケースシナリオと振舞駆動開発のシナリオは、多くの似た特徴を持ちます。

ユースケースシナリオとは、システムの利用シーンをユーザの操作とシステムの振る舞いとの相互作用で記述したものです。通常はユーザが期待する結果を得られる1つのシナリオ^{注a}と、バリデーションエラーなどによって期待する結果の得られない複数のシナリオ^{注b}で構成されます。

振舞駆動開発のシナリオとの違いは、具体的な入力データや期待結果が含まれるか否かです。ユースケースシナリオを活用した開発では、システムテストや受け入れテストのとき、ユースケースシナリオをもとに、具体的な入力データや期待結果を準備し、テストシナリオを作成します。

したがって、外部設計の段階で十分なユースケースシナリオを作成しているのであれば、振舞駆動開発の採用は簡単です。テストフェーズで行っていた具体的な入力データや期待結果の準備を、開発の早い段階で行うだけです。

注a 「ハッピーパス」や「正常系」と呼ばれます。

注b 「準正常系」や「代替シナリオ」と呼ばれます。

● API設計のタイミング

シナリオとテストコードを比較すると大きな違いがあります。それは、テストコードでは、CalculatorクラスのAPI設計が完了している点です。振舞駆動開発を行った場合でもAPI設計は必要となりますし、シナリオを記述する段階ではAPI設計についてほとんど考慮する必要はありません。

この違いは、テスト駆動開発におけるテストファーストがAPI設計を含んでいることを意味します。テストコードを記述しながらAPI設計を行うことは、テスト駆動開発の大きな特徴です。このため、「テスト駆動開発の本質は設計にある」とも言われます。テストという名前が付いたプロセスの本質が設計であることは、違和感だけでなく、設計と同時にテストコードを書く難しさでもあるのです。

一方の振舞駆動開発では、仕様の定義とAPI設計は異なる段階で行うため、シナリオの作成に集中できます。

● 検証する対象の粒度

テスト駆動開発は、クラスやメソッドといった小さな単位のプログラムを検証することに適した開発手法です。クラス設計を行いながらテストコードを書くことで、テンポよく開発を進めることができます。

一方、振舞駆動開発は、ソフトウェア全体といった大きな単位のプログラムを検証することに適した開発手法です。内部のクラス設計に束縛されず、プログラムの振る舞いに集中してシナリオを記述できます。

振舞駆動開発の語彙

本書で紹介するCucumberなどの振舞駆動開発を支援するフレームワークでは、ユニットテストとは異なる語彙を用います。それらは、ある意味ではユニットテストと変わりませんが、ある意味ではユニットテストとはまったく異なります。このことはたびたび議論になりますが、考え方の違いであり、機能的な違いではありません。

振舞駆動開発では、「Given/前提」「When/もし」「Then/ならば」が重要な語彙となります。

● Given/前提

「Given/前提」は、シナリオにおいて前提条件となる記述です。ユーザがシナリオの中心となるアクションを行う前提条件を定義します。「Given/前提」は、ユニットテストでは「初期化処理」(set up)、ユースケースシナリオでは「事前条件」に相当します。

● When/もし

「When/もし」は、シナリオにおいて中心となるユーザの操作を記述します。ユニットテストでは「実行」(exercise)に相当します。シナリオでは複数の操作が連続して行われることがあります。

ユニットテストでは、原則として1つのテストケースで1つのメソッドを実行して期待する動作となるかを検証します。シナリオでは「Aをして、Bをして、Cをした場合にDとなる」というように、いくつかの操作を連続して実行します。これは、ユーザにとって価値があるのは最後まで操作を実行した場合であるからです。個々の操作が独立して検証できたとしても、シ

15
16
17

Column

振舞駆動開発の生まれた背景

振舞駆動開発とテスト駆動開発は、「プログラムに期待する結果を先に記述することを起点に開発を行う」という点でとても似ています。

テスト駆動開発では、はじめにテストコードを書きます(テストファースト)。次に、そのテストの成功をゴールとしてプロダクションコードを書きます。そして、テストが成功したあとにリファクタリングを行います。このサイクルが完了したならば、次のテストを書いて新しいサイクルを始めます。

一方、振舞駆動開発では、はじめにシナリオを書きます(スペックファースト)。次に、そのシナリオが実行できるようにプロダクションコードを書きます。そして、シナリオが実行できたらリファクタリングを行います。このサイクルが完了したならば、次のシナリオを書いて新しいサイクルを始めます。

このように、振舞駆動開発の「シナリオ」や「振る舞い」とは、テスト駆動開発の「テスト」を単に置き換えたものです。これは「テスト」という単語の持つ「検証」という意味が強すぎるためです。そこで、「テスト」という単語を「振る舞い」に置き換え、自然な形で「先に対象が満たさなければならない仕様を記述する」ことができるよう再定義した、という背景があります。

ナリオを最初から最後まで実行できなければ価値はありません。

ユースケースシナリオの場合も同様で、ユーザがあるユースケースで実行する一連の操作をシナリオとして記述します。

● Then/ならば

「Then/ならば」は、シナリオの実行結果として、期待されることを記述するために使います。ユニットテストでは「検証」(verify)に相当します。ユニットテストと同様に複数の検証を行なうこともあります。

17.3 cucumber-junitによる振舞駆動開発

Cucumber^{注4}は、振舞駆動開発を行うためのフレームワークです。Cucumberを使うことで、ソフトウェアの振る舞いをプレーンテキストに記述して実行できます。

Cucumberは、もともとはRubyのテスティングフレームワーク「RSpec」^{注5}を拡張するフレームワークとして開発されました。Cucumberでは、Gherkin^{注6}と呼ばれるシナリオを記述するプレーンテキストのフォーマットと、そのシナリオを実行するためのフレームワークとが独立して設計されています。このため、シナリオをGherkinフォーマットで用意すれば、各言語のCucumberで利用することができます。

cucumber-junit^{注7}は、そんなCucumberをJavaで利用するためのライブラリです。

本節では、ポーカーアプリケーションをサンプルに、cucumber-junitを使った振舞駆動開発を紹介します。

注4 <http://cukes.info/>

注5 <http://rspec.info/>

注6 Cucumber(キューケンバー)もGherkin(ガーキン)もキュウリの意味。

注7 CucumberをJVM上で実行するライブラリ群「Cucumber-JVM」の一部。
<https://github.com/cucumber/cucumber-jvm>

cucumber-junitのしくみ

cucumber-junitでは、シナリオをフィーチャファイルと呼ばれるテキストファイルに定義し、JUnitのテストスイートとして実行します。

このとき、シナリオのステップである「Given/前提」「When/もし」「Then/ならば」が1つのテストケースです。このため、シナリオの数と実行結果のテスト数は異なります。

一方、各ステップが個別のテスト結果としてレポートされるため、シナリオの詳細な実行結果を確認できます。

cucumber-junitの準備

cucumber-junitはJUnitの拡張ライブラリとして提供されています。

まず、新規にpokerプロジェクトを用意します。MavenプロジェクトでもEclipseのプロジェクトでもかまいません^{注8}。

Mavenプロジェクトでcucumber-junitを利用する場合は、リスト17.1を参考に、依存ライブラリに追加してください。依存するライブラリも含めてプロジェクトに追加されます^{注9}。

注8 サンプルコードはMavenプロジェクトです。

注9 ➔依存ライブラリの追加(p.284)

リスト17.1 cucumber-junitの設定

```
<dependency>
  <groupId>info.cukes</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>1.0.8</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>info.cukes</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>1.0.8</version>
  <scope>test</scope>
</dependency>
```

通常の Eclipse プロジェクトで cucumber-junit を利用する場合は、JUnit に加えて次の 5 つの JAR ファイルをダウンロードしてプロジェクトにコピーし、ビルドパスに追加してください^{注10}。

- cucumber-core
- cucumber-java
- cucumber-junit
- cucumber-html
- gherkin

JAR ファイルは、Maven リポジトリからダウンロードします。詳細は、Cucumber-JVM の GitHub^{注11} を参照してください。

ポーカーアプリケーションの概要

ポーカーでは、ジョーカーを除いた 52 枚のトランプを使い、5 枚のカードをプレイヤーに配ります。プレイヤーは、最大で 5 枚のカードをチェンジ(山札と交換)できます。そして、5 枚のカードで作られる役の強さを競います。

本書では、プレイヤーを 1 人とし、役を判定するところまでを作成します。

フィーチャファイルの作成

はじめに、シナリオを定義するフィーチャファイルをプロジェクトのルートパッケージに作成します。テスト用のソースフォルダ(Maven プロジェクトの場合は src/test/resources)に、「cucumber.poker」パッケージを作成し、「poker_game.feature」ファイルを作成してください(図 17.1)。

フィーチャファイルの 1 行目には使用する自然言語を、2 行目にはフィーチャの定義を行います。

注10 本書では、執筆時点での最新版となる「cucumber-core-1.0.8」「cucumber-java-1.0.8」「cucumber-junit-1.0.8」「cucumber-html-0.2.1」「gherkin-2.10.0」を使用しています。

注11 <https://github.com/cucumber/cucumber-jvm>

```
# language: ja
フィーチャ: ポーカーゲーム
```

なお、デフォルトの言語設定は英語です。ここではシナリオやキーワードを日本語で記述したいので、日本語(ja)に設定しています。

言語設定が英語(en)の場合は、「フィーチャ」ではなく「Feature」というように語彙が英語となります。シナリオを日本語で記述するのであれば、日本語の設定とするほうがよいでしょう。

シナリオの作成

最初は単純なシナリオから始めましょう。「初期に配られた5枚の手札をチェンジしないとき、役ができなかった」というシナリオを定義します。

```
# language: ja
フィーチャ: ポーカーゲーム
  シナリオ: ノーチェンジ/ノーペア (役なし)
    前提 手札にS1,H4,D6,D8,C3が配られた
    もし チェンジしない
    ならば ノーペアであるべき
```

シナリオでは「S1」や「H4」といった記号でトランプのカードを表すことにしました。S1はスペードの1、H4はハートの4、同様にD6はダイアの6、

図17.1 poker_game.featureの作成



15
16
17

C3はクローバーの3となります。

なお、インデントが半角スペース2つでなければ正しく動作しないので注意してください。

● ランダム性とテスト

ポーカーではランダムにカードが配られるが、ランダム性の高いシナリオは検証することが困難です。ディーラーがランダムにカードを配るシナリオはここでは割愛します。

シナリオの実行

次に、このフィーチャファイルをJUnitで実行するためのテストクラスを作成します。テストクラスの名前に制約はありませんが、ほかのテストクラスと同様にTestで終わるようにしてください。

ここではポーカーゲームの受け入れテストクラスということで、PokerGameAcceptanceTestクラスとしました(リスト17.2)。RunWithアノテーションでは、Cucumberテストランナークラスを指定します。また、Cucumber.Optionsアノテーションで、実行時のオプションを指定します。これで、シナリオを実行するための準備は整いました^{注12}。このテストクラスを通常のJUnitテストと同様に実行してください(図17.2)。結果がグリーンとなります。しかしながら、テスト自体はスキップ(ignore)され

注12 formatではきれい(pretty)なフォーマットを、monochromeではモノクロ出力を設定しています。

リスト17.2 PokerGameのテストクラス

```
package cucumber.poker;

import org.junit.runner.RunWith;
import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)
@Cucumber.Options(format = { "pretty" }, monochrome = true)
public class PokerGameAcceptanceTest { }
```

ており、成功しているわけではありません。

JUnitではスキップされたテストはグリーンとなる仕様です。しかしながら、Cucumberでのシナリオを実行したときにスキップされたテストケースは、直感的にはレッドです。執筆時点のcucumber-junitのバージョン(1.0.8)ではJUnitの仕様に準拠しているため、グリーンです。しかしながら、修正してほしいとの声は多いため、今後のバージョンで変更されるかもしれません。

ステップ定義の作成

Cucumberでは、シナリオの実行をステップ単位で行います。このステップ単位の実行を行うために、プログラミング言語ごとにプロダクションコードを実行するコードスニペットを記述します。このコードスニペットをステップ定義ファイルに記述していきます。

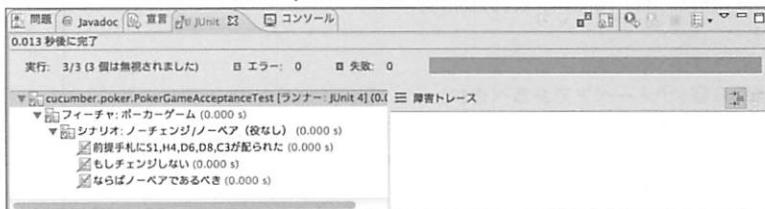
cucumber-junitでは、ステップ定義ファイルはJavaのクラスとして定義します。

● コードスニペットの利用

先ほど実行したシナリオの実行結果をコンソールウィンドウで確認してください。リスト17.3のようにステップ定義に必要なコードスニペットが出力されています。

このように、cucumber-junitではシナリオの実行時、対応するステップ定義をアノテーションと正規表現のマッチングで見つけます。キャプチャされたパラメータはメソッドの引数として利用できます。

図17.2 PokerGameAcceptanceTestの実行結果



● ステップ定義ファイルの作成

それでは、ステップ定義ファイルとして、PokerGameStepDefs クラスを cucumber.poker パッケージに作成します。

PokerGameStepDefs クラス内に標準出力されたコードスニペットをコピー&ペーストで貼り付けたならば、**[Ctrl]+[Shift]+[O]** でインポートを再編成してください。ただし、コードスニペットではカードのスト (絵柄) が固定値となっているため、「前提」の部分をリスト 17.4 のように修正します。

引数が多くきれいなコードではありませんが、メソッド内でパース処理を行うよりはよいでしょう。

ここでシナリオを実行すれば、テストは失敗となります(リスト 17.5)。これは「前提」で PendingException が送出されているためです。

このように、Cucumber でシナリオを定義する場合、プロダクションコードの実装や設計を行なう必要はありません。これは、テスト駆動開発でテストファーストを実践するときと比べて、最も異なる特徴です。

リスト 17.3 PokerGameAcceptanceTest の実行結果(標準出力)

You can implement missing steps with the snippets below:

```
@前提("^手札にS(¥¥d+),H(¥¥d+),D(¥¥d+),C(¥¥d+)が配られた$")
public void 手札にS_H_D_C_が配られた(
    int arg1, int arg2, int arg3, int arg4, int arg5)
    throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@もし("^チェンジしない$")
public void チェンジしない() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@ならば("^ノーペアであるべき$")
public void ノーペアであるべき() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}
```

ステップ定義の実装

続けて各ステップを実行できるように実装します。

なお、振舞駆動開発のシナリオのテスト対象は複数のクラスが相互作用するアプリケーションとなります。このため、一気にステップ定義を実装することは困難です。プロダクションコードの設計を行いながら、ステップ定義を行わなければなりません。

● ポーカーアプリケーションの設計

ステップ定義を実装するとき、まだプロダクションコードがありません。どのようなクラスが必要で、どのように利用されるかをコードとして記述

リスト17.4 修正後のステップ定義(一部)

```
@前提("^手札に([SHDC])(¥d+),([SHDC])(¥d+),([SHDC])(¥d+)
      ,([SHDC])(¥d+),([SHDC])(¥d+)が配られた$")
public void 手札にカードが配られた(
    char suit1, int no1,
    char suit2, int no2,
    char suit3, int no3,
    char suit4, int no4,
    char suit5, int no5) {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}
```

15
16
17

リスト17.5 ステップ定義後の実行結果

```
# language: ja
フィーチャ: ポーカーゲーム

シナリオ: ノーチェンジ/ノーペア # cucumber/poker/poker_game.feature:3
  前提手札にS1,H4,D6,D8,C3が配られた # PokerGameStepDefs
    .手札にカードが配られた(char,int,char,int,char,int,char,int,
      cucumber.runtime.PendingException: TODO: implement me
      at cucumber.poker.PokerGameStepDefs
        .手札にカードが配られた(PokerGameStepDefs.java:18)
      at *.前提手札にS1,H4,D6,D8,C3が配られた(
        cucumber/poker/poker_game.feature:4)
```

します。

リスト17.6では、アプリケーションのインターフェースとなるクラスとしてPokerGameクラスを定義し、setUpメソッドでカードを5枚用意しました。カードはCardクラスで表現され、staticなgetメソッドで取得しています。PokerGameクラスには、チェンジしない場合のnoChangeメソッドが定義されます。役の判定は、PokerGameクラスのpatメソッドで行われ、役を表すPatクラスが返されます。

このようなソフトウェア全体の設計は、クラス図やシーケンス図などを利用して、ホワイトボードなどで整理します。

大まかにクラス設計を行ったならば、各クラスの詳細な設計と実装を行います。ここでテスト駆動開発を適用し、テストコードを書きながらクラスの設計と実装を進めるとよいでしょう。

● プロダクションコード

プロダクションコードはリスト17.7～17.10のように設計し、実装しました。

Cardクラス(リスト17.7)はスートと数値を保持する不变クラスとして設計しました。不正なカードを作成できないように、インスタンスはファクトリメソッドを利用して取得します。カードは有限かつ不变ですので、はじめにすべてのカードのインスタンスを生成しておいてもよいでしょう。

Handsクラス(リスト17.8)はカードを保持するコンテナクラスです。現時点では特別な実装はありませんが、チェンジの実装などが追加される予定です。

役の判定は、Patクラス(リスト17.9)で行う設計としました。この役クラスは未完成であり、現時点で必要なノーペアのみを定義しています。次のシナリオとしてワンペアのシナリオを受け入れテストとして作成したあとに実装を追加していきます。

PokerGameクラス(リスト17.10)は、このソフトウェアの中心となるクラスです。このクラスは、ユーザがこのソフトウェアを操作するときのインターフェースとなります。また、ゲームの状態を管理し、不正な操作が行われないように監視しています。状態を管理するクラスはバグを埋め込みやすく、テストも行いにくいため、状態の管理はPockerGameクラスのみで

行い、具体的な処理は Hands クラスや Pat クラスに委譲する設計になっています。

プロダクションコードを追加したならば、シナリオを実行してグリーンとなることを確認してください。

Column

PokerGame クラスの設計

本書で解説しているポーカーゲームのクラス設計は、仕様が大きく変更される可能性はなく、仕様もそれほど複雑ではないため、過剰設計かもしれません。もっとシンプルに、1つか2つのクラスで設計することもできるでしょう。

しかしながら、一般的には、オブジェクト指向設計の単一責務の原則(SRP : *The Single Responsibility Principle*)に従って設計されることが良いとされています。つまり、クラスはただ1つの責務(役割)を持つべきであり、そのクラスが変更される理由は2つ以上あってはならないということです。

ポーカーゲームにおけるCard クラスは、カードの種類や数値が変更されない限りは変更されません。Hands クラスは、ポーカーのルールが変更され、手札の数が変わらない限りは変更されません。Pat クラスは、新しい役が追加されたり判定方法が変わったりしない限りは変更されません。

クラスの設計がシンプルであるならば、それぞれのクラスのユニットテストは簡単になります。

15
16
17

リスト17.6 ステップ定義の実装

```
package cucumber.poker;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;
import cucumber.annotation.ja.ならば;
import cucumber.annotation.ja.もし;
import cucumber.annotation.ja.前提;

public class PokerGameStepDefs {

    PokerGame sut;

    @前提("^手札に([SHDC])(¥¥d+),([SHDC])(¥¥d+),([SHDC])(¥¥d+)
           ,([SHDC])(¥d+),([SHDC])(¥d+)が配られた$")
    public void 手札にカードが配られた()
    {
        sut = new PokerGame();
        sut.hand();
    }
}
```

```

    char suit1, int no1,
    char suit2, int no2,
    char suit3, int no3,
    char suit4, int no4,
    char suit5, int no5) {
sut = new PokerGame();
sut.setUp(Card.get(suit1, no1),
          Card.get(suit2, no2),
          Card.get(suit3, no3),
          Card.get(suit4, no4),
          Card.get(suit5, no5));
}

@もし("^チェンジしない$")
public void チェンジしない() {
    sut.noChange();
}

@ならば("^ノーペアであるべき$")
public void ノーペアであるべき() {
    Pat result = sut.pat();
    assertThat(result, is(Pat.NO_PAIR));
}
}

```

リスト17.7 Cardクラス

```

package cucumber.poker;

public class Card {
    public enum Suit {
        DIAMONDS, SPADES, HEARTS, CLUBS;
    }

    public final int no;
    public final Suit suit;

    Card(Suit suit, int no) {
        this.suit = suit;
        this.no = no;
    }
}

```

```

public static Card get(char suit, int no) {
    if (no < 1 || 13 < no)
        throw new IllegalArgumentException();
    switch (suit) {
        case 'D':
            return new Card(Suit.DIAMONDS, no);
        case 'S':
            return new Card(Suit.SPADES, no);
        case 'H':
            return new Card(Suit.HEARTS, no);
        case 'C':
            return new Card(Suit.CLUBS, no);
        default:
            throw new IllegalArgumentException();
    }
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + no;
    result = prime * result + ((suit == null) ? 0 : suit.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Card other = (Card) obj;
    if (no != other.no) return false;
    if (suit != other.suit) return false;
    return true;
}
}

```

15

16

17

リスト17.8 Handsクラス

```
package cucumber.poker;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class Hands implements Iterable<Card> {

    private static final int SIZE = 5;
    final Set<Card> cards = new HashSet<Card>(SIZE);

    public Hands(Card... cards) {
        if (cards.length != SIZE) throw new IllegalArgumentException();
        for (Card card : cards) {
            this.cards.add(card);
        }
    }

    @Override
    public Iterator<Card> iterator() {
        return cards.iterator();
    }
}
```

リスト17.9 Patクラス

```
package cucumber.poker;

import java.util.HashMap;
import java.util.Map.Entry;

public abstract class Pat {
    public static final Pat NO_PAIR = new NoPair();

    public static Pat make(Hands hands) {
        // TODO 他の役の実装
        return NO_PAIR;
    }

    public static class NoPair extends Pat {
        // hashCode, equalsメソッドは省略
    }
}
```

リスト17.10 PokerGameクラス

```

package cucumber.poker;

public class PokerGame {
    private enum Status {
        INIT, READY, CHANGED
    }
    private Status status = Status.INIT;
    private Hands hands = null;

    public void setUp(Card... cards) {
        if (status != Status.INIT) throw new IllegalStateException();
        hands = new Hands(cards);
        this.status = Status.READY;
    }

    public void noChange() {
        if (status != Status.READY) throw new IllegalStateException();
        // do nothing
        this.status = Status.CHANGED;
    }

    public Pat pat() {
        if (status != Status.CHANGED) throw new IllegalStateException();
        return Pat.make(hands);
    }
}

```

15

16

17

ワンペアシナリオの作成

1つ目のシナリオが実行できたら、新しいシナリオをフィーチャファイルに追加します。ここでは、リスト17.11のようにワンペアのシナリオを追加しました。

続けてステップ定義をPokerGameStepDefsクラスに追加します(リスト17.12)。ノーペアの場合と異なる点は、どの数字のワンペアかを引数としてキャプチャしている点です。「前提」と「もし」のステップ定義は、ノーペアの場合と変わらないため、今回は定義する必要ありません。

最後に Pat クラスを修正します(リスト 17.13)。ここではワンペアを判定するため最も単純なロジックとしています。すなわち、すべてのカードに対し、各数字が何枚ずつあるかを計算し、2枚のカードがあった場合にワンペアとしています。

なお、この実装ではツーペア(ワンペアが2種類)やフルハウス(ワンペアとスリーカード)の場合もワンペアとして判定されますが、この段階では気にしません。振舞駆動開発やテスト駆動開発では、それらの実装はツーペアやフルハウスを扱うシナリオやテストを追加したときに考慮します。

このように、振舞駆動開発では、1つのシナリオを定義し動くように実装するサイクルを繰り返しながらソフトウェアを開発します。また、ここでは触れませんでしたが、シナリオが実行できることを確認したならば、積極的にリファクタリングを行います。

リスト 17.11 ワンペアのシナリオを追加したフィーチャファイル

```
# language: ja
フィーチャ: ポーカーゲーム
  シナリオ: ノーチェンジ/ノーペア (役なし)
    前提 手札にS1,H4,D6,D8,C3が配られた
    もし チェンジしない
    ならば ノーペアであるべき
```

```
シナリオ: ノーチェンジ/ワンペア
  前提 手札にS1,H1,D6,D8,C3が配られた
  もし チェンジしない
  ならば 1のワンペアであるべき
```

リスト 17.12 ステップ定義の追加

```
@ならば("^(¥¥d+)のワンペアであるべき$")
public void ワンペアであるべき(int no) {
  Pat expected = new Pat.OnePair(no);
  assertThat(sut.pat(), is(expected));
}
```

リスト17.13 ワンペアに対応したPatクラス

```

package cucumber.poker;

import java.util.HashMap;
import java.util.Map.Entry;

public abstract class Pat {
    public static final Pat NO_PAIR = new NoPair();

    public static Pat make(Hands hands) {
        HashMap<Integer, Integer> nums = new HashMap<Integer, Integer>(5);
        for (Card card : hands) {
            Integer count = nums.get(card.no);
            if (count == null) count = 0;
            count++;
            nums.put(card.no, count);
        }
        for (Entry<Integer, Integer> entry : nums.entrySet()) {
            if (entry.getValue() == 2) return new OnePair(entry.getKey());
        }
        // TODO 他の役の実装
        return NO_PAIR;
    }

    public static class NoPair extends Pat {
        // hashCode, equalsメソッドは省略
    }

    public static class OnePair extends Pat {
        public final int no;

        OnePair(int no) {
            this.no = no;
        }
        // hashCode, equalsメソッドは省略
    }
}

```

15

16

17

シナリオテンプレートの利用

ポーカーゲームのシナリオでは、手札のパターンを1つだけ用意して開発を進めていました。しかしながら、ノーペアやワンペアのパターンは1つだけではなく、いくつかのパターンで動作を確認しなければ不安が残ります。

シナリオをコピーしてパターンを増やすこともできますが、第8章で紹介したパラメータ化テストができると便利です。Cucumberではシナリオをテンプレート化することでパラメータ化テストを実現できます(リスト17.14)。リスト17.14①②のように、`<手札>`をプレースホルダとして利用し、③で「例」としてプレースホルダで置換するパラメータを与えてください。

このように、シナリオをテンプレート化すれば、さまざまな「例」を効率良く追加できます。

アウトサイドインの開発

振舞駆動開発もテスト駆動開発も、1つのシナリオやユニットテストを定義してからプロダクションコードを作成する開発手法です。しかしながら、受け入れテストとなるシナリオを定義して振舞駆動開発を行う場合、ブ

リスト17.14 シナリオテンプレートの利用

```
# language: ja
フィーチャ: ポーカーゲーム
  シナリオテンプレート: ノーチェンジの場合
    前提 手札に<手札>が配られた      ①
    もし チェンジしない
      ならば <役>であるべき        ②
```

例: ③

手札	役
S1,H4,D6,D8,C3	ノーペア
S1,S2,S3,S4,C9	ノーペア
H1,H4,D2,D8,D9	ノーペア
H1,H4,D1,D8,D9	1のワンペア
H3,H9,D1,D8,D9	9のワンペア

ロダクションコードが単一のクラスとして構成されることはほとんどありません。このため、テスト駆動開発に比べ、振舞駆動開発のサイクルは長くなってしまいます。

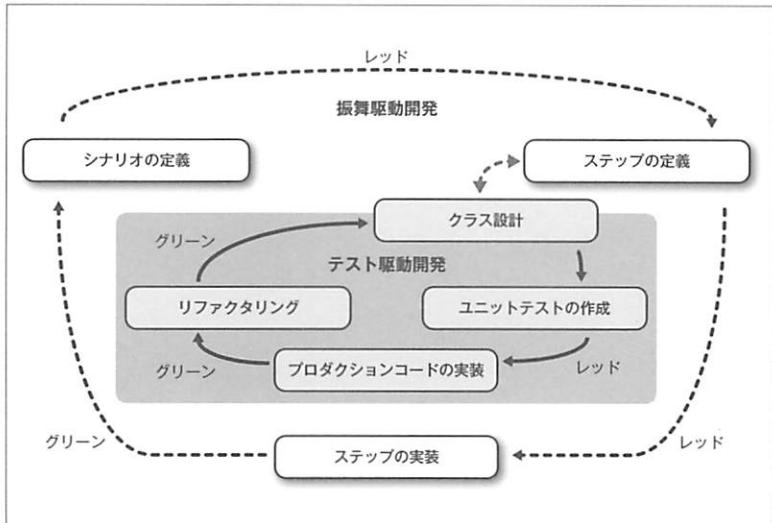
ソフトウェア開発で、開発のサイクルが長いことは大きな足かせです。開発のサイクルが短いほど、さまざまなフィードバックを早い段階で得られます。そして、そのフィードバックをもとにより開発を改善していくことができます。対象となる問題が小さければ悩む時間も少なく、リズムよく開発を進めることができます。一方で、「木を見て森を見ず」とならないよう、ソフトウェア全体をとらえた視点も必要です。

そこで、はじめにユーザ視点でのソフトウェアの外部的な振る舞いを振舞駆動開発で定義し、ソフトウェアを構成する個々のクラスをテスト駆動開発で設計し、シナリオを動作させるという開発を行うと効果的です。つまり、外側のサイクルとして振舞駆動開発を用い、内側のサイクルとしてテスト駆動開発を用います(図17.3)。大きなサイクルと小さなサイクルを組み合わせることで、開発のリズムを損なわず、かつ全体視点でソフトウェアをとらえることができます。

ポーカーアプリケーションの例であれば、はじめに振舞駆動開発の最初

15
16
17

図17.3 アウトサイドインの開発スタイル



のステップとしてシナリオを作成し、ステップ定義のひな型を作成するまでを行います。ここでソフトウェアの大まかなクラス設計を行い、Cardクラス、Handsクラス、Patクラス、PokerGameクラスを抽出します。

次に各クラスに対してテスト駆動開発を行い、テストコードとプロダクションコードを作成します。最後に振舞駆動開発のサイクルに戻り、ステップ定義を実装し、シナリオが動くことを確認します。

このように、振舞駆動開発とテスト駆動開発を組み合わせ、アウトサイドイン(*outside-In*)で開発すると、お互いのメリットを最大限に活かすことができます^{注13}。

注13 同語はRubyとなります。次の書籍でCucumberとRSpecを利用したアウトサイドインによる開発が詳しく説明されています。David Chelimsky、Dave Astels、Zach Dennis著／株式会社クイープ監訳／角谷信太郎、豊田祐司監『The RSpec Book』翔泳社(2012年)。

于で小笠山の活用
第20章

芦井一司による小笠山
第19章

八一の小笠山
第18章

演習問題

Part 5

Part 5 では、実践的なユニットテストの演習問題を紹介します。これまで学んできたJUnitの機能やテスト技法を使って、演習問題を解いてみましょう。難易度は「☆☆☆」から「★★★」までの4段階で、次のような位置づけです。

- ☆☆☆ …… 入門レベル
- ★☆☆ …… 基礎レベル
- ★★☆ …… 実践レベル
- ★★★ …… 応用レベル

各問題には、問題を解くためのヒントと解答例を用意しています。それらは以下の構成になっています。

● ヒント

設計では、テスト対象クラスの設計に関して記述しています。プロダクションコードがイメージできない場合に参考にしてください。

テストケースでは、プロダクションコードに対し、どのようなテストを記述するかのサンプルを記述しています。具体的にどのようなテストを書けばよいかイメージできない場合に参考にしてください。

● 解答

テストコードでは、テストコードのサンプルを記述しています。なお、あくまで解答例であり、絶対的な正解ではありません。

プロダクションコードも同様にサンプルのひとつです。また、テストを満たすための最小限の実装としているため、原則として例外処理などは省略しています。

解説では、テストコードのサンプルをもとにポイントを解説しています。



まずは問題を読み、自分自身でテストコードやプロダクションコードを書いてみましょう。イメージが湧かない場合は、順番に読み進め、実際にコードを書いてみてください。たとえ丸写しであっても、多くのテストコードを書くことが経験を高める最良の方法です。

第18章

ベーシックなテスト

本章は、JUnitを使ったユニットテストの基本的な演習問題です。簡単なテストコードが多くなりますが、テストコードを簡単に書けることは良い設計であるともいえます。

18.1 状態を持たないメソッドのテスト

難易度:☆☆☆

文字列をスネークケース(すべて小文字で単語区切りがアンダースコア)に変換するユーティリティメソッドのテストを作成せよ。

ヒント

設計

- StringUtils クラスを定義する
- StringUtils クラスに toSnakeCase メソッドを定義する
- toSnakeCase メソッドは、publicかつstatic メソッドとする
- toSnakeCase メソッドは、String型の引数を1つ持ち、戻り値を String型とする

テストケース

- 「aaa」を入力すると、「aaa」が取得できる
- 「HelloWorld」を入力すると、「hello_world」が取得できる
- 「practiceJunit」を入力すると、「practice_junit」が取得できる

解答

テストコード

リスト18.1 状態を持たないメソッドのテスト

```
public class StringUtilsTest {
    @Test
    public void toSnakeCaseはスネークケースを返す_aaaの場合() {
        assertThat(StringUtils.toSnakeCase("aaa"), is("aaa"));
    }
}
```

 18
19
20

```

    }

    @Test
    public void toSnakeCaseはスネークケースを返す_HelloWorldの場合() {
        assertThat(StringUtils.toSnakeCase("HelloWorld"),
                   is("hello_world"));
    }

    @Test
    public void toSnakeCaseはスネークケースを返す_practiceJunitの場合() {
        assertThat(StringUtils.toSnakeCase("practiceJunit"),
                   is("practice_junit"));
    }
}

```

プロダクションコード

リスト18.2 StringUtilsクラス

```

public class StringUtils {
    public static String toSnakeCase(String text) {
        if (text == null) throw new NullPointerException("text == null.");
        String snake = text;
        Pattern p = Pattern.compile("[A-Z]");
        for (;;) {
            Matcher m = p.matcher(snake);
            if (!m.find()) break;
            snake = m.replaceFirst("_" + m.group(1).toLowerCase());
        }
        return snake.replaceFirst("^_", "");
    }
}

```

解説

状態を持たず、同じ入力ならば常に同じ結果を返すメソッドはユーティリティメソッドとも呼ばれます。ユーティリティメソッドは、最もユニットテストを行いやすいクラス設計のひとつです。

ユーティリティメソッドにはいくつかの引数が定義され、一定の戻り値を返します。したがって、テストケースの入力値、出力値(実測値)、期待値が明確です。

この例では、テストメソッドごとにテストケースを定義しました(リスト18.1)。テストケースを可能な限り独立したテストメソッドに定義することは良い習慣です。

しかしながら、テストケースごとにテストメソッドを分けた場合、重複の多い、大量のテストメソッドを持つテストコードとなりがちです。そのような場合は、パラメータ化テスト(第8章)を使い、テストコードを整理することを検討してください。

18.2 例外を送出するメソッドのテスト

難易度:☆☆☆

Calculator クラスに整数の割り算を行う divide メソッドを作成し、0 で割る場合に例外が送出されることを検証するテストを作成せよ。なお、整数の割り算は小数点以下を切り捨ててよい。

ヒント

設計

- Calculator クラスに、divide メソッドを定義する
- divide メソッドは、int 型の引数を 2 つ持ち、戻り値を int 型とする
- 戻り値は第 1 引数を第 2 引数で割った値とする
- 第 2 引数が 0 である場合、IllegalArgumentException を送出する

テストケース

- 第 2 引数に 0 を指定して divide を呼び出すと、IllegalArgumentException が発生する

解答

テストコード

リスト18.3 例外を送出するメソッドのテスト

```
public class CalculatorTest {
    @Test(expected = IllegalArgumentException.class)
    public void divideの第2引数に0を指定すると例外が発生する()
        throws Exception {
        Calculator sut = new Calculator();
        sut.divide(1, 0);
```

 18
 19
 20

```

    }
}

```

プロダクションコード

リスト18.4 Calculatorクラス

```

public class Calculator {
    public int divide(int x, int y) {
        if (y == 0) throw new IllegalArgumentException("y is zero");
        return x / y;
    }
}

```

解説

例外の送出を検証するテストケースでは、Testアノテーションのexpected属性を使用します。標準的なテストメソッドでは結果の検証にアサーションを記述しますが、例外の送出を検証するテストケースでは検証部分をアノテーションで宣言的に記述します。expected属性をもとにフレームワークにより検証されます。

戻り値があるメソッドの例外送出をテストする場合、リスト18.3のようにメソッドの実行だけを記述するほうがよいでしょう。リスト18.5のように変数に代入してしまうと、変数の利用目的がわかりにくくなります。

なお、例外メッセージ、エラーコード、ネストされた例外などのより詳細な検証を行う場合は、ExpectedExceptionルールを利用します^{注1}。

リスト18.5 例外が送出されるため変数への代入が行われないテストコード

```

@Test(expected = IllegalArgumentException.class)
public void divideの第2引数に0を指定すると例外が発生する()
    throws Exception {
    Calculator sut = new Calculator();
    // 例外がthrowされるため代入されることはない
    int actual = sut.divide(1, 0);
}

```

注1 ➡ ExpectedException——詳細な例外を扱う(p.150)

18.3 副作用を持つメソッドのテスト

難易度:★☆☆

Counter クラスの increment メソッドは、呼び出すごとに 1 ずつ加算された値を返す。このメソッドのテストを作成せよ。

なお、初回の increment メソッドの呼び出し時には 1 を返すこと。

ヒント

設計

- Counter クラスに increment メソッドを定義する
- Counter クラスに現在値を保持する初期値 0 の int 型フィールドを定義する
- increment メソッドは引数を持たず、戻り値を int 型とする

テストケース

- 初期状態で、increment メソッドを実行すると 1 が取得できる
- increment メソッドを 1 回実行した状態で、increment メソッドを実行すると 2 が取得できる
- increment メソッドを 50 回実行した状態で、increment メソッドを実行すると 51 が取得できる

解答

テストコード

リスト18.6 副作用を持つメソッドのテスト

```
@RunWith(Enclosed.class)
public class CounterTest {
    public static class 初期状態の場合 {
        Counter sut;
        @Before
        public void setUp() {
            sut = new Counter();
        }
        @Test
        public void incrementで1が取得できる() {
            assertThat(sut.increment(), is(1));
        }
    }
    public static class incrementが1回実行された場合 {
        Counter sut;
```

18
19
20

```

@Before
public void setUp() {
    sut = new Counter();
    sut.increment();
}
@Test
public void incrementで2が取得できる() {
    assertThat(sut.increment(), is(2));
}
}
public static class incrementが50回実行された場合 {
    Counter sut;
    @Before
    public void setUp() {
        sut = new Counter();
        for (int i = 0; i < 50; i++) sut.increment();
    }
    @Test
    public void incrementで51が取得できる() {
        assertThat(sut.increment(), is(51));
    }
}
}

```

プロダクションコード

リスト18.7 Counterクラス

```

public class Counter {
    int count = 0;
    public int increment() {
        return ++count;
    }
}

```

解説

メソッドの実行によりオブジェクトの内部状態などが更新され、そのメソッド自身やほかのメソッドの振る舞いに影響を与えるメソッドは、「副作用を持つメソッド」と呼ばれます。一般的に、副作用を持つメソッドは、副作用を持たないメソッドよりもテストが難しくなります。

オブジェクト指向プログラミングでは、プログラムの基本構成単位であるクラスに属性(フィールド)と振る舞い(メソッド)を持たせ、オブジェクトの相互作用でソフトウェアを構築します。したがって、オブジェクトが状態を持つことは不自然なことではありません。しかしながら、オブジェクトが状態を持つことによりテスタビリティは大きく低下します。オブジェクトの状態によって期待する結果が異なるため、状態が複雑であるほどテストケースは増えますし、初期化処理も複雑になります。

このため、テスタビリティの高い設計とするには、可能な限りオブジェクトに状態を持たせるべきではありません。ただし、現実としては状態をまったく持たないことは困難です。そこで、状態を持つクラスと持たないクラスを明確に分けて設計することが現実的な対策となります。

状態を持つクラスのテストを行う場合はEnclosed テストランナーを利用し、状態ごとにコンテキストを分けるとよいでしょう^{注2}。リスト18.6ではコンテキストごとにテストケースが1つしか定義されていませんが、各コンテキストに新しいテストメソッドを追加するならば、重複する初期化処理を共通化すると見通しの良いテストコードとなります。

もし、状態がより複雑であるならば、状態遷移図などを併用して整理してください。オブジェクトの状態やそれぞれの状態での振る舞いを整理しておくと、テストが作成しやすくなります。

18.4 同値クラスに対するテスト

難易度:★☆☆

整数値を引数として持ち、偶数である場合にtrueを返すisEven メソッドのテストを作成せよ。isEven メソッドはNumberUtils クラスに static メソッドとして定義する。

注2 ➔ テストのコンテキスト(p.92)

ヒント**設計**

- NumberUtils クラスにisEven メソッドを定義する
- isEven メソッドはint型の引数xを持ち、戻り値をboolean型とする
- xが2で割り切れる場合にtrue を返す

テストケース

- 入力値に「10」を与えると、true を返す
- 入力値に「7」を与えると、false を返す

解答**テストコード****リスト18.8 同値クラスによるテスト**

```
public class NumberUtilsTest {
    @Test
    public void isEvenは10でtrueを返す() throws Exception {
        assertThat(NumberUtils.isEven(10), is(true));
    }

    @Test
    public void isEvenは7でfalseを返す() throws Exception {
        assertThat(NumberUtils.isEven(7), is(false));
    }
}
```

プロダクションコード**リスト18.9 NumberUtilsクラス**

```
public class NumberUtils {
    public static boolean isEven(int num) {
        return num % 2 == 0;
    }
}
```

解説

ユニットテストにおいて、テストデータの選択は重要です。効率良く、少ないテストデータを選択できれば、ユニットテストの実装コストだけでな

く、メンテナンスコストも軽減します。

この問題のテスト対象メソッドでは、入力値(整数)は無数に存在しますが、期待される結果はtrueまたはfalseの2種類しかありません。このような場合、期待される結果に着目することで、効率良くテストデータを選択できます。リスト18.8では、期待する結果であるtrueとfalseにそれぞれ対応する入力値として10と7を選択しました。

もちろん、10と7だけではなく、ほかの入力値(たとえば8と13など)についてもテストを行うことができます。しかしながら、これ以上のテストケースを追加したとしても、テスト対象メソッドのバグを検知できる可能性は低いでしょう。なぜならば、10と7の2種類の値を選択してテストした時点で、ほかの値も10か7と同じように処理され、同じ結果を返すからです。このような、同じように処理が行われたり、同じ結果を返したりするデータのグループは同値クラス^{注3}と呼ばれます。

完璧なソフトウェアを作ることは不可能です。同値クラスに対するテストを行うことで、コストと品質とのバランスを保つことができます。

18.5 void型を戻り値とするメソッドのテスト 難易度:★★☆

商品を表すItemクラスが定義されているとき、商品を追加／削除／数量の変更などができるItemStockクラスを作成したい。商品を追加するaddメソッドのテストを作成せよ。

なお、商品は商品名(name)で一意に識別できるとする。

```
Itemクラス
public class Item {
    public final String name;
    public final int price;
    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }
}
```

注3 ➔同値クラスに対するテスト(p.28)

ヒント**設計**

- ItemStock クラスに add メソッドを定義する
- add メソッドは Item 型の引数を 1 つ持ち、戻り値は void とする
- ItemStock クラスは、指定した Item オブジェクトの数量を返す getNum メソッドを持つ
- getNum メソッドは、Item 型の引数を 1 つ持ち、戻り値は int とする
- getNum メソッドは、指定した Item オブジェクトが登録されていない場合は 0 を返す

テストケース

- 初期状態で、getNum で 0 が取得できる
- 初期状態で、add で Item を追加すると getNum で 1 が取得できる
- Item が 1 つ追加されている状態で、getNum で 1 が取得できる
- Item が 1 つ追加されている状態で、add で同じ Item オブジェクトを追加すると getNum で 2 が取得できる
- Item が 1 つ追加されている状態で、add で異なる Item オブジェクトを追加すると getNum で 1 が取得できる

解答**テストコード****リスト18.10 void型を戻り値とするメソッドのテスト**

```
@RunWith(Enclosed.class)
public class ItemStockTest {
    public static class 初期状態の場合 {
        ItemStock sut;
        Item book;

        @Before
        public void setUp() throws Exception {
            book = new Item("book", 3800);
            sut = new ItemStock();
        }
    }

    @Test
}
```

18
19
20

```

public void getNumはbookで0を返す() throws Exception {
    assertThat(sut.getNum(book), is(0));
}

@Test
public void addでbookを追加するとgetNumで1を返す()
    throws Exception {
    sut.add(book);
    assertThat(sut.getNum(book), is(1));
}

}

public static class bookが1回追加されている場合 {
    ItemStock sut;
    Item book;

    @Before
    public void setUp() throws Exception {
        book = new Item("book", 3800);
        sut = new ItemStock();
        sut.add(book);
    }

    @Test
    public void getNumはbookで1を返す() throws Exception {
        assertThat(sut.getNum(book), is(1));
    }

    @Test
    public void addでbookを追加するとgetNumで2を返す()
        throws Exception {
        sut.add(book);
        assertThat(sut.getNum(book), is(2));
    }

    @Test
    public void addでbikeを追加するとgetNumでbookとbikeは1を返す()
        throws Exception {
        Item bike = new Item("bike", 57000);
        sut.add(bike);
        assertThat(sut.getNum(book), is(1));
    }
}

```

```
        assertThat(sut.getNum(bike), is(1));  
    }  
}  
}
```

プロダクションコード

リスト18-11 ItemStockクラス

```
public class ItemStock {  
  
    private final Map<String, Integer> values  
        = new HashMap<String, Integer>();  
  
    public void add(Item item) {  
        Integer num = values.get(item.name);  
        if (num == null) num = 0;  
        num++;  
        values.put(item.name, num);  
    }  
  
    public int getNum(Item item) {  
        Integer num = values.get(item.name);  
        return num != null ? num : 0;  
    }  
}
```

解說

戻り値がvoid型であるメソッドは、戻り値による検証ができません。このため、状態の変化を計測するためのメソッドやフィールドを参照することで検証を行う必要があります。

リスト18.10では、数量を取得できるgetNumメソッドを用いてaddメソッドの検証を行っています。同様にgetNumメソッドもaddメソッドを使わなければ十分な検証を行うことができません。このように、いくつかのメソッドが相互作用するクラスでは、テストも複雑になります。

add メソッドのテストは、ItemStock クラスの values フィールドをパッケージプライベートなどで定義し、values フィールドを直接検証することでも実現できます。しかしながら、この方法ではテストコードがプロダクシ

ヨンコードの実装に強く依存し、リファクタリングを行いにくくなります。

なお、いくつかのメソッドのテストが相互依存している場合、テストの成否がほかのテストの結果に影響し、テスト全体の信頼性を保つことが困難です。この例であれば、getNum メソッドのテストが失敗していると、関連するテストはすべて期待された動きをしていない可能性があります。

18.6 マルチスレッドのテスト

難易度:★★★

処理をバックグラウンドのスレッドで非同期に実行したい。

バックグラウンドスレッドで処理を実行する BackgroundTask クラスを作成し、タスクがバックグラウンドのスレッドで実行されることを検証するテストを作成せよ。

なお、タスクは Runnable オブジェクトに実装する。

ヒント

設計

- BackgroundTask は、コンストラクタで Runnable オブジェクトを引数に持つ
- BackgroundTask クラスに、invoke メソッドを定義する
- invoke メソッドは、引数を持たず、戻り値を void 型とする
- invoke メソッドでは、バックグラウンドスレッドでタスクを実行し、すぐに制御を呼び出し元に返す

テストケース

- invoke メソッドにより Runnable オブジェクトの run メソッドが別スレッドで実行される

解答

テストコード

リスト18.12 マルチスレッドのテスト

```
public class BackgroundTaskTest {
    @Rule
    public Timeout timeout = new Timeout(1000);
```

18
19
20

```

    @Test
    public void invokeで別スレッドで実行される() throws Exception {
        // SetUp
        final AtomicReference<String> backgroundThreadName
            = new AtomicReference<String>();
        final CountDownLatch latch = new CountDownLatch(1);
        Runnable task = new Runnable() {
            @Override
            public void run() {
                backgroundThreadName.set(Thread.currentThread()
                    .getName());
                latch.countDown();
            }
        };
        BackgroundTask sut = new BackgroundTask(task);
        // Exercise
        sut.invoke();
        latch.await();
        // Verify
        assertThat(backgroundThreadName.get(),
            is(not(Thread.currentThread().getName())));
    }
}

```

プロダクションコード

リスト18.13 BackgroundTaskクラス

```

public class BackgroundTask {

    private final Runnable task;

    public BackgroundTask(Runnable task) {
        this.task = task;
    }

    public void invoke() {
        Executors.newSingleThreadExecutor().execute(task);
    }
}

```

解説

スレッドを扱う処理のユニットテストは簡単ではありません。「どのように実行されるか」と「何を実行するか」を明確に分離した設計とし、それをテストすることが基本となります。この問題では、「何を実行するのか」はRunnableオブジェクトの実装クラスで行い、「どのように実行されるか」を担当するBackgroundTaskについてテストしています。

invokeメソッドを実行すると、すぐに制御が返されます。このため、タスクの実行が終わるまで、何らかの方法でテストメソッドを実行しているスレッドを待機しなければ、検証を行うことができません。また、タスクの中にアサーションのコードを記述したとしても、AssertionErrorはバックグラウンドスレッドで送出されるため、テスティングフレームワークに通知されません。

リスト18.12では、Runnableオブジェクトの実行完了を待機するために、Java Concurrentフレームワークに含まれるCountDownLatchクラスを利用しています。CountDownLatchクラスは、コンストラクタで指定した整数値をカウンタとして保持します。そして、awaitメソッドを呼び出すことによって、カウントがゼロになるまで待機します。このCountDownLatchを利用すると、別スレッドでcountDownメソッドが実行され、カウントがゼロになるまで、テストを実行しているスレッドを待機させることができます^{注4}。

また、リスト18.12では確実に処理が別スレッドで実行されたかを検証するためにスレッドの名前を取得しています。ここで利用しているAtomicReferenceクラスはスレッドセーフにオブジェクトの参照を保持できるコンテナです。Runnableオブジェクトの処理が実行されたあとに、テストメソッドを実行したスレッドと異なるスレッドで実行されたかどうかを検証しています。

なお、マルチスレッドを扱うテストケースでは、Timeoutルール^{注5}を付与しておくことをお勧めします。なぜならば、何らかのバグで処理がロックしてしまった場合に、テストが成功にも失敗にもならず、待ち続けてしまうからです。

注4 このサンプルでは、厳密には処理が終了するまでは待機していませんが、テストの要件は満たせています。

注5 ➔Timeout——テストのタイムアウトを設定する(p.151)

第19章

アサーションとフィクスチャ

本章は、実践的なユニットテストの要となるアサーションとフィクスチャに関する演習問題です。

テストケースが増えてきたり、テストの前提条件が複雑になってきたりすると、テストコードには重複が増え、変更に弱くなってしまいます。これらの問題を解決するには、カスタム Matcher^{注1}や Enclosed テストランナー^{注2}が有効です。テストコードを整理し、きれいなテストコードを書くことを心がけてください。

19.1 リストのアサーション

難易度:★☆☆

1から指定された値までの値を文字列に変換し、リストとして返すクラスのテストを作成せよ。

ただし、値が3の倍数の場合は「Fizz」を、値が5の倍数の場合は「Buzz」を、値が3の倍数かつ5の倍数の場合は「FizzBuzz」を値の代わりにリストに格納すること^{注3}。たとえば、4を指定した場合は「1, 2, Fizz, 4」を返す。

ヒント

設計

- FizzBuzz クラスに、createFizzBuzzList メソッドを定義する
- createFizzBuzzList メソッドは static メソッドとする
- createFizzBuzzList メソッドは、int 型の引数を 1 つ持ち、戻り値を java.util.List<String> 型とする
- 引数がマイナスの場合などの例外処理は省略する

注1 ➔ カスタム Matcher の作成(p.71)

注2 ➔ テストケースの整理(p.92)

注3 「FizzBuzz問題」と呼ばれ、プログラミングの入門的な演習問題として知られています。

テストケース

- 16を指定してcreateFizzBuzzListメソッドを実行すると、「1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16」が取得できる

解答

テストコード

リスト19.1 リストを1つずつ検証するテスト

```
public class FizzBuzzTest {

    @Test
    public void createFizzBuzzListで16まで取得できる() {
        List<String> actual = FizzBuzz.createFizzBuzzList(16);
        assertThat(actual, is(notNullValue()));
        assertThat(actual.size(), is(16));
        assertThat(actual.get(0), is("1"));
        assertThat(actual.get(1), is("2"));
        assertThat(actual.get(2), is("Fizz"));
        assertThat(actual.get(3), is("4"));
        assertThat(actual.get(4), is("Buzz"));
        assertThat(actual.get(5), is("Fizz"));
        assertThat(actual.get(6), is("7"));
        assertThat(actual.get(7), is("8"));
        assertThat(actual.get(8), is("Fizz"));
        assertThat(actual.get(9), is("Buzz"));
        assertThat(actual.get(10), is("11"));
        assertThat(actual.get(11), is("Fizz"));
        assertThat(actual.get(12), is("13"));
        assertThat(actual.get(13), is("14"));
        assertThat(actual.get(14), is("FizzBuzz"));
        assertThat(actual.get(15), is("16"));
    }
}
```

18
19
20

プロダクションコード

リスト19.2 FizzBuzzクラス

```
public class FizzBuzz {
```

```

public static List<String> createFizzBuzzList(int size) {
    ArrayList<String> list = new ArrayList<String>(size);
    for (int i = 1; i <= size; i++) {
        if (i % 15 == 0) {
            list.add("FizzBuzz");
        } else if (i % 3 == 0) {
            list.add("Fizz");
        } else if (i % 5 == 0) {
            list.add("Buzz");
        } else {
            list.add(Integer.toString(i));
        }
    }
    return list;
}
}

```

解説

戻り値がコレクションで、順序も含めすべての要素を検証する場合、リスト19.1のようにサイズと各要素のアサーションを行うのがシンプルな方法です。それぞれの要素を個別に検証すれば、テストが失敗したときに、何番目の要素で検証が失敗したかを簡単に知ることができます。しかしながら、検証用のコードが長く、冗長になってしまいます。

一方、リスト19.3のように期待値となるリストを初期化処理で作成し、アサーションを行う方法もあります。検証用のコードは短くなりましたが、テストが失敗した場合に、何番目の要素で検証が失敗したかがわかりにくくなります。

このような場合、両方のメリットを持つカスタムMatcherを作成するとよいでしょう。リスト19.4は、リストの要素を1件ずつ検証し、検証に失敗した場合に詳細な情報を出力するカスタムMatcherです。リスト19.4では、コンストラクタとカスタムMatcherのファクトリメソッドに可変長引数を利用しています。

ListMatcherを利用すればテストコードを直感的な表現で記述できるため、テストコードはコンパクトになります、読みやすくなります(リスト19.5)。また、テストが失敗した場合にも、何番目の要素が一致しなかったかはJUnitの実行結果で次のように確認できます。

```
java.lang.AssertionError:  
Expected: is "5", but "Buzz" at index of 4  
got: <[1, 2, Fizz, 4, Buzz, Fizz]>
```

ユニットテストの実践的な運用では、テストコードの可読性とテスト失敗時の詳細な情報は最も重要です。アサーションでより詳細な情報を取得し、テストコードを読みやすくコンパクトにするために、カスタムMatcherを作成してください。

リスト19.3 リストを作成して検証する

```
@Test  
public void createFizzBuzzListで16まで取得できる() {  
    List<String> expected = new ArrayList<String>(16);  
    expected.add("1");  
    expected.add("2");  
    ...  
    List<String> actual = FizzBuzz.createFizzBuzzList(16);  
    assertThat(actual, is(expected));  
}
```

リスト19.4 リストのカスタムMatcher

```
public class Lists {  
  
    public static <T> Matcher<List<?>> list(T... items) {  
        return new ListMatcher(items);  
    }  
  
    static class ListMatcher extends BaseMatcher<List<?>> {  
  
        final Object[] items;  
        List<?> actual;  
        boolean matches = false;  
        int idx = 0;  
  
        ListMatcher(Object[] items) {  
            this.items = items;  
        }  
    }  
}
```

```

@Override
public boolean matches(Object actual) {
    if (!(actual instanceof List)) {
        return false;
    }
    List<?> list = (List<?>) actual;
    this.actual = list;
    if (list.size() != items.length) {
        return false;
    }
    for (Object obj : list) {
        Object other = items[idx];
        if (obj != null && !obj.equals(other)) {
            return false;
        } else if (obj == null && other != null) {
            return false;
        }
        idx++;
    }
    return true;
}

@Override
public void describeTo(Description desc) {
    if (actual == null) {
        desc.appendValue(items);
    } else {
        desc.appendValue(items[idx])
            .appendText(", but ")
            .appendValue(actual.get(idx))
            .appendText(" at index of " + idx);
    }
}
}

```

リスト19.5 カスタムMatcherを使ったテストコード

```

List<String> actual = FizzBuzz.createFizzBuzzList(6);
assertThat(actual, is(list("1", "2", "Fizz", "4", "Buzz", "Fizz")));

```

19.2 JavaBeansのアサーション

難易度:★★☆

例に示すように、テキストファイルに「名前,名字,メールアドレス」がカンマ区切りで定義されるとする。

```
Ichiro,Tanaka,ichiro@example.com
```

このとき、このテキストファイルから、Employeeオブジェクトのリストを読み込むメソッドのテストを作成せよ。

Employeeクラス

```
public class Employee {
    private String firstName;
    private String lastName;
    private String email;
    // Getter/Setter省略
}
```

ヒント

設計

- Employeeクラスにloadメソッドを定義する
- loadメソッドはstaticメソッドとする
- loadメソッドは、InputStream型の引数を1つ持ち、戻り値をList<Employee>型とする

テストケース

- テキストファイルを指定してメソッドを実行すると、名前、名字、メールアドレスが反映されたEmployeeオブジェクトのリストを取得できる

解答

テストコード

リスト19.6 Employeeオブジェクトの読み込みテスト

```
public class EmployeeTest {
    @Test
    public void loadでEmployeeの一覧を取得できる() throws Exception {
        // SetUp
        InputStream input = getClass().getResourceAsStream("Employee.txt");
```

18

19

20

```

// Exercise
List<Employee> actual = Employee.load(input);
// Verify
assertThat(actual, is(notNullValue()));
assertThat(actual.size(), is(1));
Employee actualEmployee = actual.get(0);
assertThat(actualEmployee.getFirstName(), is("Ichiro"));
assertThat(actualEmployee.getLastName(), is("Tanaka"));
assertThat(actualEmployee.getEmail(), is("ichiro@example.com"));
}
}

```

プロダクションコード

リスト19.7 Employeeクラスのloadメソッド(抜粋)

```

public static List<Employee> load(InputStream input) {
    LinkedList<Employee> list = new LinkedList<Employee>();
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new InputStreamReader(input));
        for (;;) {
            String line = reader.readLine();
            if (line == null) break;
            String[] values = line.split(",");
            Employee employee = new Employee();
            employee.setFirstName(values[0]);
            employee.setLastName(values[1]);
            employee.setEmail(values[2]);
            list.add(employee);
        }
        return list;
    } catch (IOException e) {
        throw new RuntimeException(e);
    } finally {
        if (reader != null) {
            try { reader.close(); }
            catch (IOException e) {/* do nothing */}
        }
    }
}

```

解説

一般的に、Javaではオブジェクトの同値比較に`equals`メソッドを利用します。JUnitでアサーションを行う場合でも同様で、`is`メソッドを使ったアサーションでは同値比較に`equals`メソッドを利用します。したがって、ほとんどの場合では`is`メソッド^{注4}を利用し、`equals`メソッドによるアサーションを行います。

ところが、`equals`メソッドによる比較では、多くのフィールドを持つJavaBeansオブジェクトの検証に失敗した場合、どのフィールドが一致しなかったかはわかりません。このため、`toString`メソッドなどでフィールドの値を出力し、文字列として確認を行うことが多いでしょう。しかし、これは非常に面倒で非生産的な作業です。

このような問題を解決するためには、リスト19.6のようにフィールドごとのアサーションが有効です。テストに失敗した場合に、どのフィールドが一致していないのかが明確になります^{注5}。しかしながら、フィールドごとにアサーションを行った場合、フィールドを追加した場合に検証用のコードを追加し忘れるリスクがあります。また、いくつかのテストケースで同様のアサーションを行うならば、テストコードが重複し、可読性とメンテナビリティを低下させます。このため、リスト19.8のように複雑なオブジェクトを比較検証し、一致しなかった場合に詳細な情報を出力するカスタムMatcherを作成してください。

カスタムMatcherを作ることで、泥臭い処理をテストコードから取り除き、アサーション用のコードを再利用することができます。

18
19
20

注4 ➡`is`(p.67)

注5 ➡問題の局所化—— テスト失敗時に問題を特定しやすいこと(p.36)

リスト19.8 EmployeeクラスのカスタムMatcher

```

public class EmployeeMatcherTest {
    @Test
    public void loadでEmployeeの一覧を取得できる() throws Exception {
        // SetUp
        InputStream input = getClass().getResourceAsStream("Employee.txt");
        Employee expected = new Employee();
        expected.setFirstName("Ichiro");
        expected.setLastName("Tanaka");
        expected.setEmail("ichiro@example.com");
        // Exercise
        List<Employee> actual = Employee.load(input);
        // Verify
        assertThat(actual, is(notNullValue()));
        assertThat(actual.size(), is(1));
        assertThat(actual.get(0), is(employee(expected)));
    }
}

class EmployeeMatcher extends BaseMatcher<Employee> {

    private final Employee expected;
    private String field;
    private Object expectedValue;
    private Object actualValue;

    public static Matcher<Employee> employee(Employee expected) {
        return new EmployeeMatcher(expected);
    }

    EmployeeMatcher(Employee expected) {
        this.expected = expected;
    }

    @Override
    public boolean matches(Object actual) {
        if (expected == null) return (actual == null);
        if (!(actual instanceof Employee)) return false;
        Employee other = (Employee) actual;
        if (notEquals(expected.getFirstName(), other.getFirstName())) {
            field = "firstName";
            expectedValue = expected.getFirstName();
        }
    }
}

```

```

        actualValue = other.getFirstName();
        return false;
    }
    if (notEquals(expected.getLastName(), other.getLastName())) {
        field = "lastName";
        expectedValue = expected.getLastName();
        actualValue = other.getLastName();
        return false;
    }
    if (notEquals(expected.getEmail(), other.getEmail())) {
        field = "email";
        expectedValue = expected.getEmail();
        actualValue = other.getEmail();
        return false;
    }
    return true;
}

private boolean notEquals(Object obj, Object other) {
    if (obj == null) return other != null;
    return !obj.equals(other);
}

@Override
public void describeTo(Description desc) {
    if (expected == null || field == null) {
        desc.appendValue(expected);
    } else {
        desc.appendText(field + " is ").appendValue(expectedValue)
            .appendText(", but ").appendValue(actualValue);
    }
}
}

```

18

19

20

19.3 複数行テキストのアサーション

難易度:★★☆

引数で指定した複数の文字列を、改行で連結して返すメソッドのテストを作成せよ。

なお、メソッドの引数は可変長引数とし、改行コードはシステムの標準改行コードとする。また、テストが失敗したとき、アサーションに失敗した個所を識別しやすいように、カスタム Matcher を作成すること。

ヒント

設計

- MultiLineString クラスに、join メソッドを定義する
- join メソッドは static メソッドとする
- join メソッドは、String 型の引数を可変長引数として持ち、戻り値を String 型とする
- 連結するときの改行コードはシステム環境変数から取得する

テストケース

- Hello と World の 2 つの文字列を join メソッドに与えると、改行で連結された文字列が取得できる

解答

テストコード

リスト19.9 複数行テキストのアサーション

```
public class MultiLineStringTest {
    @Test
    public void joinで文字列が連結される() throws Exception {
        String ls = System.getProperty("line.separator");
        String expected = "Hello" + ls + "World" + ls;
        assertThat(MultiLineString.join("Hello", "World"),
                   is(text(expected)));
    }
}

class MultiLineStringMatcher extends BaseMatcher<String> {
```

```

public static Matcher<String> text(String text) {
    return new MultiLineStringMatcher(text,
        System.getProperty("line.separator"));
}

private final String expected;
private Object actual;
private final List<String> expectedLines = new ArrayList<String>();
private final List<String> actualLines = new ArrayList<String>();
private final Pattern pattern;

public MultiLineStringMatcher(String expected, String ls) {
    this.expected = expected;
    if (ls.equals("\r")) {
        this.pattern = Pattern.compile(".+(\\r|$)");
    } else if (ls.equals("\n")) {
        this.pattern = Pattern.compile(".+(\\n|$)");
    } else {
        this.pattern = Pattern.compile(".+(\\r\\n|$)");
    }
    if (expected != null) {
        java.util.regex.Matcher m = pattern.matcher(expected);
        while (m.find()) {
            expectedLines.add(expected.substring(m.start(), m.end()));
        }
    }
}

@Override
public boolean matches(Object actual) {
    this.actual = actual;
    if (expected == null) return (actual == null);
    if (!(actual instanceof String)) {
        return false;
    }
    if (expected.equals(actual)) return true;
    String actualString = (String) actual;
    java.util.regex.Matcher m = pattern.matcher(actualString);
    while (m.find()) {
        actualLines.add(actualString.substring(m.start(), m.end()));
    }
    return expectedLines.equals(actualLines);
}

```

18
19
20

```

    }

    @Override
    public void describeTo(Description desc) {
        if (expected == null || actual == null) {
            desc.appendValue(expected);
        } else {
            int lines = Math.min(expectedLines.size(), actualLines.size());
            for (int idx = 0; idx < lines; idx++) {
                String expectedLine = expectedLines.get(idx);
                String actualLine = actualLines.get(idx);
                if (!expectedLine.equals(actualLine)) {
                    desc.appendValue(expectedLine);
                    desc.appendText(", but actual is ");
                    desc.appendValue(actualLine);
                    desc.appendText(", line " + (idx + 1) + "¥n");
                    desc.appendValue(expected);
                }
            }
            desc.appendText("expected text is "
                + expectedLines.size() + " lines, ");
            desc.appendText("but actual text is "
                + actualLines.size() + " lines¥n");
            desc.appendValue(expected);
        }
    }
}

```

プロダクションコード

リスト19.10 MultiLineStringクラス

```

public class MultiLineString {
    public static String join(String... lines) {
        if (lines == null) return null;
        String lineSeparator = System.getProperty("line.separator");
        StringBuilder text = new StringBuilder();
        for (String line : lines) {
            text.append(line != null ? line : "").append(lineSeparator);
        }
        return text.toString();
    }
}

```

解説

XML や JSON など複数行で構成される文字列のアサーションは、失敗したときの分析が厄介です。なぜならば、文字列が非常に長くなり、改行コードも含むため、単純な文字列として比較するには複雑すぎるからです。

リスト 19.9 は、アサーションのエラーメッセージに一致しなかった行番号とその文字列を出力するカスタム Matcher クラスです。このカスタム Matcher を利用することで、複数行で構成される文字列も次のようにわかりやすくエラーメッセージが表示されるようになります。

```
java.lang.AssertionError:  
expected: is "ddd\n", but actual is "DDD\n", line 4  
aa\nbbb\nccc\nnnn\n"  
got: "aaa\nbbb\nccc\nnnn\n"
```

複雑なカスタム Matcher ですが、再利用性も高く、テスト失敗時の検証で大きな効果を得ることができます。

19.4 境界値のテスト

難易度:★★☆

浮動小数点数の範囲を表す Range クラスがある。この Range クラスに、浮動小数点数を指定して範囲内であるかどうかを判定する contains メソッドを定義し、そのテストを作成せよ。

なお、Range クラスの contains メソッドの判定では、範囲の両端は範囲内であるとする。

Rangeクラス	18 19 20
----------	----------------

```
public class Range {  
    public final double min;  
    public final double max;  
    public Range(double min, double max) {  
        this.min = min;  
        this.max = max;  
    }  
}
```

ヒント**設計**

- Range クラスに、contains メソッドを定義する
- contains メソッドは、引数に double 型の引数を 1 つ持ち、戻り値を boolean 型とする

テストケース

- min=0.0、max=10.5 の Range のとき
 - contains メソッドに -0.1 を与えると、false を返す
 - contains メソッドに 0.0 を与えると、true を返す
 - contains メソッドに 10.5 を与えると、true を返す
 - contains メソッドに 10.6 を与えると、false を返す
- min=-5.1,max=5.1 の Range のとき
 - contains メソッドに -5.2 を与えると、false を返す
 - contains メソッドに -5.1 を与えると、true を返す
 - contains メソッドに 5.1 を与えると、true を返す
 - contains メソッドに 5.2 を与えると、false を返す

解答**テストコード****リスト19.11** 境界値のテスト

```
public class RangeTest {
    @Test
    public void containsのテスト() {
        assertThat(new Range(0d, 10.5).contains(-0.1), is(false));
        assertThat(new Range(0d, 10.5).contains(0.0), is(true));
        assertThat(new Range(0d, 10.5).contains(10.5), is(true));
        assertThat(new Range(0d, 10.5).contains(10.6), is(false));
        assertThat(new Range(-5.1, 5.1).contains(-5.2), is(false));
        assertThat(new Range(-5.1, 5.1).contains(-5.1), is(true));
        assertThat(new Range(-5.1, 5.1).contains(5.1), is(true));
        assertThat(new Range(-5.1, 5.1).contains(5.2), is(false));
    }
}
```

プロダクションコード

リスト19.12 Rangeクラスのcontainsメソッド

```
public boolean contains(double value) {
    return min <= value && value <= max;
}
```

解説

このように範囲などを判定するメソッドのテストでは、境界値に対するテスト^{注6}が有効です。境界値に対するテストは、プログラムのバグが判定の境界値に偏在する性質を利用し、境界値をテストデータとして選択するテスト技法です。

リスト19.11ではRangeオブジェクトを2種類用意し、それぞれの範囲の両端を基準に4つずつのテストデータを選択しています。

Rangeオブジェクトでは、条件やテストデータがそれほど多くないため、リスト19.11のように1つのテストメソッドに連続してテストコードを記述しても可読性は低くなりません。しかしながら、条件が複雑になり、テストデータも多くなった場合には、パラメータ化テストを検討してください。

リスト19.13は、パラメータ化テストとEnclosedによるコンテキストごとにテストケースを整理したテストコードです。Enclosedテストランナーはネストさせることができますため、パラメータとそのテストの期待値ごとにコンテキストを分けることもできます。このテストコードは複雑に感じますが、テストケースが構造化されているため、テストデータの追加や変更は行いやすいでしょう。

注6 ➔境界値に対するテスト(p.28)

リスト19.13 境界値テストの構造化とパラメータ化(抜粋)

```

@RunWith(Enclosed.class)
public class RangeTest {
    @RunWith(Enclosed.class)
    public static class Rangeが0から10_5で {
        @RunWith(Theories.class)
        public static class かつ範囲外の場合 {
            Range sut;
            @Before
            public void setUp() {
                sut = new Range(0d, 10.5);
            }
            @DataPoints
            public static double[] VALUES = { -0.1, 10.6 };
            @Theory
            public void containsはfalseを返す(double value)
                throws Exception {
                assertThat("value=" + value,
                           sut.contains(value), is(false));
            }
        }

        @RunWith(Theories.class)
        public static class かつ範囲内の場合 {
            Range sut;
            @Before
            public void setUp() {
                sut = new Range(0d, 10.5);
            }
            @DataPoints
            public static double[] VALUES = { 0.0, 10.5 };
            @Theory
            public void containsはtrueを返す(double value)
                throws Exception {
                assertThat("value=" + value,
                           sut.contains(value), is(true));
            }
        }
    }
}

```

19.5 フィクスチャを用いたパラメータ化テスト

難易度:★★☆

消費税を表す ConsumptionTax クラスを作成し、金額を引数に与えると消費税込みの金額を返すメソッドのテストを作成せよ。なお、消費税率はコンストラクタで指定し、税額は小数点以下を切り捨てとする。

ヒント

設計

- ConsumptionTax クラスはコンストラクタに int 型引数を持ち、税率としてフィールドに保持する
- ConsumptionTax クラスに、apply メソッドを定義する
- apply メソッドは、int 型引数を 1 つ持ち、戻り値を int 型とする

テストケース

- 表 19.1 のパラメータに関してそれぞれテストを実施する

表 19.1 テストパラメータ

税率	入力値(価格)	期待値
5	100	105
5	3000	3150
10	50	55
5	50	52
3	50	51

解答

テストコード

リスト 19.14 フィクスチャを用いたパラメータ化テスト

```
@RunWith(Theories.class)
public class ConsumptionTaxTest {
    @DataPoints
    public static Fixture[] FIXTURES = new Fixture[] {
        new Fixture(5, 100, 105),
        new Fixture(5, 3000, 3150),
        new Fixture(10, 50, 55),
        new Fixture(5, 50, 52),
        new Fixture(3, 50, 51),
    }
}
```

18
19
20

```

};

@Theory
public void applyで消費税が加算された価格が取得できる(Fixture fixture)
throws Exception {
    ConsumptionTax sut = new ConsumptionTax(fixture.taxRate);
    String desc = "when rate=" + fixture.taxRate
        + ", price=" + fixture.price;
    assertThat(desc, sut.apply(fixture.price), is(fixture.expected));
}
}

class Fixture {
    final int taxRate;
    final int price;
    final int expected;
    Fixture(int taxRate, int price, int expected) {
        this.taxRate = taxRate;
        this.price = price;
        this.expected = expected;
    }
}
}

```

プロダクションコード

リスト19.15 ConsumptionTaxクラス

```

public class ConsumptionTax {

    private final int rate;

    public ConsumptionTax(int rate) {
        this.rate = rate;
    }

    public int apply(int price) {
        return price + (price * this.rate / 100);
    }
}

```

解説

多くの入力値と期待値のセットに対してユニットテストを行いたい場合、それらの値を1つのフィクスチャとしてパラメータ化テストを行うと効果

的です。

リスト 19.14 では、税率、入力値(金額)、期待値をフィールドとして持つFixtureクラスを定義しています。Fixtureクラスは、パラメータ化テストのパラメータとしてテストクラスに定義されています。コンストラクタを工夫することで、テストパラメータの表と同じように読むことができます。

テストを実行すれば、各Fixtureオブジェクトがテストメソッドに渡されます。このようにテストコードを記述すれば、簡単にテストパターンを追加できます。

なお、入力値と期待値が多い場合や複雑なオブジェクトを生成する必要がある場合は、YAMLなどのフォーマットを利用した外部ファイルに定義するとよいでしょう^{注7}。

19.6 組み合わせテスト

難易度:★★★

あるWebアプリケーションフレームワークのサポート状況は、表19.2となっている。このとき、各環境を引数として指定し、サポートされる組み合わせならばtrueを返すメソッドと、そのテストを作成せよ。

注7 ➔外部リソースからのセットアップ(p.114)

18
19
20

表19.2 フレームワークのサポート状況

アプリケーションサーバ	データベース	サポート
GlassFish	Oracle	○
GlassFish	DB2	○
GlassFish	PostgreSQL	○
GlassFish	MySQL	○
JBoss	Oracle	×
JBoss	DB2	○
JBoss	PostgreSQL	○
JBoss	MySQL	×
Tomcat	Oracle	×
Tomcat	DB2	×
Tomcat	PostgreSQL	×
Tomcat	MySQL	○

ヒント**設計**

- Frameworks クラスに、isSupport メソッドを定義する
- isSupport メソッドは、static メソッドとする
- isSupport メソッドは、ApplicationServer型の引数、Database型の引数を持ち、戻り値を boolean型とする
- ApplicationServer型、Database型はそれぞれ enum として定義する

テストケース

- isSupport は、表19.2で○の組み合わせの場合に true を返す
- isSupport は、表19.2で×の組み合わせの場合に false を返す

解答**テストコード****リスト19.16 void型を戻り値とするメソッドのテスト**

```
@RunWith(Theories.class)
public class FrameworksTest {

    @DataPoints
    public static ApplicationServer[] APP_SERVER_PARAMS
        = ApplicationServer.values();

    @DataPoints
    public static Database[] DATABASE_PARAMS = Database.values();
    static Map<String, Boolean> SUPPORTS = new HashMap<String, Boolean>();

    @BeforeClass
    public static void setUpClass() throws Exception {
        InputStream in = FrameworksTest.class
            .getResourceAsStream("support.txt");
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new InputStreamReader(in));
            for (;;) {
                String line = reader.readLine();
                if (line == null) break;
                String[] params = line.split("¥|");
                SUPPORTS.put(params[0] + "-" + params[1],
```

```

        Boolean.valueOf(params[2]));
    }
} finally {
    if (reader != null) reader.close();
}
}

@Theory
public void isSupportはtrueを返す(
    ApplicationServer appServer, Database db) throws Exception {
    assumeTrue(isSupport(appServer, db));
    String desc = ", AppServer:" + appServer + ", DB:" + db;
    assertThat(desc, Frameworks.isSupport(appServer, db), is(true));
}

@Theory
public void isSupportはfalseを返す(
    ApplicationServer appServer, Database db) throws Exception {
    assumeTrue(!isSupport(appServer, db));
    String desc = ", AppServer:" + appServer + ", DB:" + db;
    assertThat(desc, Frameworks.isSupport(appServer, db), is(false));
}

private boolean isSupport(ApplicationServer appServer, Database db) {
    return SUPPORTS.get(appServer.toString() + "-" + db.toString());
}
}

```

リスト19.17 サポート状況定義ファイル

```

GlassFish|Oracle|true
GlassFish|DB2|true
GlassFish|PostgreSQL|true
GlassFish|MySQL|true
JBoss|Oracle|false
JBoss|DB2|true
JBoss|PostgreSQL|true
JBoss|MySQL|false
Tomcat|Oracle|false
Tomcat|DB2|false
Tomcat|PostgreSQL|false
Tomcat|MySQL|true

```

18
19
20

プロダクションコード

リスト19.18 Frameworksクラス

```
public class Frameworks {
    public static boolean isSupport(
        ApplicationServer appServer, Database db) {
        switch (appServer) {
            case GlassFish:
                return true;
            case Tomcat:
                return db == Database.MySQL;
            case JBoss:
                return db == Database.DB2 || db == Database.PostgreSQL;
            default:
                return false;
        }
    }
}
```

リスト19.19 ApplicationServerクラス

```
public enum ApplicationServer {
    GlassFish, Tomcat, JBoss
}
```

リスト19.20 Databaseクラス

```
public enum Database {
    Oracle, DB2, PostgreSQL, MySQL
}
```

解説

このような組み合わせが多数となる場合、パラメータ化テストを行うと、テストコードの可読性を損うことなくテストの網羅性を高めることができます。また、組み合わせ表などはコードではわかりにくいため、外部ファイルに定義するとメンテナンスしやすくなります。

リスト19.16では、アプリケーションサーバとデータベースをそれぞれ列挙型としてDataPointsに定義しています。列挙型^{注8}の定義値をすべて指定するには、列挙型の全定義値を返すvaluesメソッドが便利です。DataPoints

注8 Javaにおける型がある安全な定数を表現するための言語仕様。

アノテーションに、それぞれの列挙型の定義値が定義されているため、2つのテストメソッドはすべての列挙型の定義値の組み合わせを引数として実行されます。

それぞれのテストメソッドでは、パラメータの組み合わせを `assumeTrue` メソッドでフィルタリングします。このとき、各テストで有効なパラメータかどうかを判定するために、外部ファイル(リスト 19.17)に定義したサポート状況を参照し、`boolean`型を返すメソッドを組み込みました。

外部定義ファイルは、表 19.2 とほぼ同等のテキストファイルであるため、読みやすく、サポート状況の仕様に変更があった場合でも修正が簡単です。また、修正時にはテストコードは修正する必要がありません。

また、テストメソッドではどのパラメータによるテストケースであるかを明確にするため、パラメータの情報をアサーションのエラーメッセージに追加しています。

18
19
20

第20章

テストダブルの活用

本章は、スタブやモックといったテストダブル^{注1}に関する演習問題です。ランダム性の高い機能や外部システムに依存する機能など、期待する振る舞いを制御できないクラスを含んだユニットテストは扱いが難しくなります。このため、そのような個所をスタブやモックに置き換えることができる、テスタビリティの高い設計が重要です。

20.1 システム時間に依存するテスト

難易度:★★☆

システム時間を基準とし、月末までの日数を取得するメソッドとテストを作成せよ。なお、月の末日である場合は0を返すこととする。

ヒント

設計

- MonthlyCalendar クラスに getRemainingDays メソッドを作成する
- getRemainingDays メソッドは、引数を持たず、int 型の戻り値を持つ

テストケース

- 現在時刻が 2012/1/31 の場合、0 を返す
- 現在時刻が 2012/1/30 の場合、1 を返す
- 現在時刻が 2012/2/1 の場合、29 を返す

解答

テストコード

リスト 20.1 システム時間に依存するテスト

```
public class MonthlyCalendarTest {
```

```
    @Test
```

注1 ⇒ テストダブル(p.172)

```

public void 現在時刻が20120131の場合getRemainingDaysは0を返す() {
    MonthlyCalendar sut
        = new MonthlyCalendar(newCalendar(2012, 1, 31));
    assertThat(sut.getRemainingDays(), is(0));
}

@Test
public void 現在時刻が20120130の場合getRemainingDaysは1を返す() {
    MonthlyCalendar sut
        = new MonthlyCalendar(newCalendar(2012, 1, 30));
    assertThat(sut.getRemainingDays(), is(1));
}

@Test
public void 現在時刻が20120201の場合getRemainingDaysは28を返す() {
    MonthlyCalendar sut
        = new MonthlyCalendar(newCalendar(2012, 2, 1));
    assertThat(sut.getRemainingDays(), is(28));
}

static Calendar newCalendar(int yyyy, int mm, int dd) {
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR, yyyy);
    cal.set(Calendar.MONTH, mm - 1);
    cal.set(Calendar.DATE, dd);
    cal.set(Calendar.HOUR, 0);
    cal.set(Calendar.MINUTE, 0);
    cal.set(Calendar.SECOND, 0);
    cal.set(Calendar.MILLISECOND, 0);
    return cal;
}
}

```

プロダクションコード

リスト20.2 MonthlyCalendarクラス

```

public class MonthlyCalendar {

    private final Calendar cal;

    public MonthlyCalendar() {

```

18
19
20

```

        this(Calendar.getInstance());
    }

MonthlyCalendar(Calendar cal) { ❶
    this.cal = cal;
}

public int getRemainingDays() {
    return cal.getActualMaximum(Calendar.DATE)
        - cal.get(Calendar.DATE);
}
}

```

解説

システム時間などを扱うクラスでは、テストを実行するタイミングによって結果が変わってしまいます。このような場合、システム時間などをテストコードから設定できる設計とし、期待される結果が一定となるように制御します。

リスト 20.2 ❶では、Calendar オブジェクトを引数に持つコンストラクタ MonthlyCalendar(Calendar cal) を定義しています。このコンストラクタを利用することで、任意の日付に関してテストが実行できます(リスト 20.1)。なお、このコンストラクタはユニットテスト用であるため、可視性をパッケージプライベートとしました。このように、任意の時間を指定して処理を実行できるようにすることで、テストアビリティが向上します。

別 の方法としては、テストで制御できない処理をメソッドに抽出してオーバーライドすることもできます(リスト 20.3)。また、処理を別のクラスに委譲すれば、スタブを利用してテストすることもできます(リスト 20.4)。

getRemainingDays メソッドのようなユーティリティメソッドは、一般的には static メソッドとします。しかしながら、static メソッドではここで紹介したテクニックを使うことができません。テストアビリティを高く設計するためには、static メソッドを避けることも必要です。

リスト20.3 Calendarオブジェクトの生成をメソッドに抽出

```
public int getRemainingDays() {
    Calendar cal = getCalendar();
    return cal.getActualMaximum(Calendar.DATE) - cal.get(Calendar.DATE);
}

Calendar getCalendar() {
    return Calendar.getInstance();
}
```

リスト20.4 スタブで置き換える可能なSystemCalendar

```
public static interface SystemCalendar {
    Calendar getInstance();
}

SystemCalendar sysCal = new SystemCalendar() {
    @Override
    public Calendar getInstance() {
        return Calendar.getInstance();
    }
};

public int getRemainingDays() {
    Calendar cal = sysCal.getInstance();
    return cal.getActualMaximum(Calendar.DATE) - cal.get(Calendar.DATE);
}
```

18
19
20**20.2 例外ハンドリングのテスト**

難易度:★★☆

ログファイルを読み込み、解析を行う LogAnalyzer クラスは、ログの読み込み部分を LogLoader クラスに分離した設計となっている。

LogLoader クラスは、ログファイル名を文字列で引数に取り、読み込み結果を Map で返す load メソッドを持つ。LogAnalyzer は、LogLoader を利用してログを読み込み、解析処理を行う。

LogAnalyzer クラスの analyze メソッドの実行時、LogLoader クラスの load メソッドが IOException を送出したならば AnalyzeException が再送出することをテストせよ。なお、原因となった例外に IOException が含まれていること。

LogAnalyzerクラス

```

public class LogAnalyzer {

    LogLoader logLoader = new LogLoader();

    public Object analyze(String file) {
        try {
            Map<String, String> rawData = logLoader.load(file);
            return doAnalyze(rawData);
        } catch (IOException e) {
            throw new AnalyzeException(e);
        }
    }

    private Object doAnalyze(Map<String, String> rawData) {
        // これは仮実装です
        return new Object();
    }

    static class AnalyzeException extends RuntimeException {
        public AnalyzeException(Throwable cause) {
            super(cause);
        }
    }
}

```

LogLoaderクラス

```

public class LogLoader {
    public Map<String, String> load(String fileName) throws IOException {
        // これは仮実装です
        return new HashMap<String, String>();
    }
}

```

ヒント**設計**

- LogAnalyzer クラス、LogLoader クラスは問題で示したクラス設計とする

テストケース

- LogLoader の load メソッドが呼び出されたときに IOException を送出されただならば、AnalyzeException が送出される
- 上記のとき、エラーメッセージに原因となった例外のメッセージが含まれる
- 上記のとき、原因となった例外が AnalyzeException に含まれる

解答

テストコード

リスト20.5 例外ハンドリングのテスト

```

public class LogAnalyzerTest {

    @Rule
    public ExpectedException ex = ExpectedException.none();

    @Test
    public void LogLoaderが例外を送出するときAnalyzeExceptionが再送出される()
        throws Exception {
        // SetUp
        LogAnalyzer sut = new LogAnalyzer();
        final IOException errorCause = new IOException("error by stub");
        LogLoader mockLoader = mock(LogLoader.class);
        sut.logLoader = mockLoader;
        when(mockLoader.load("test")).thenThrow(errorCause);
        ex.expect(LogAnalyzer.AnalyzeException.class);
        ex.expectMessage("error by stub");
        ex.expect(new BaseMatcher<Object>() {
            Throwable cause;
            @Override
            public boolean matches(Object item) {
                Throwable t = (Throwable) item;
                cause = t.getCause();
                return cause == errorCause;
            }
            @Override
            public void describeTo(Description description) {
                description.appendValue(cause);
            }
        });
    }
}

```

18
19
20

```
// Exercise
sut.analyze("test");
}
}
```

解説

例外ハンドリングのユニットテストを行う場合、意図的に例外を送出できる設計にしておくと便利です。例外を意図的に送出させるためには、例外を送出する個所をメソッドとして抽出してオーバーライドするか、例外を送出するテストダブルで置き換えます。

リスト 20.5 では、Mockitoを利用して LogLoader のスタブオブジェクトを作成しています。スタブオブジェクトは load メソッドが呼び出されると IOException を送出するように記録され、LogAnalyzer オブジェクトに設定します。

また、リスト 20.5 では例外のアサーションに、ExpectedException ルール^{注2}を使用しています。ExpectedException ルールを利用することで、例外メッセージや原因となった例外の検証なども行うことができます。ただし、ExpectedException クラスには原因となった例外クラスの検証メソッドはないため、汎用的に利用できる expect メソッドを利用してカスタム Matcher を指定して検証しています。

20.3 外部システムに依存するテスト

難易度:★★☆

NetworkResources クラスは、ネットワーク上のリソースファイルを読み込み、文字列として返す load メソッドを持つ。ただし、ネットワークに接続し、InputStream オブジェクトを生成する処理は NetworkLoader クラスに定義する。

NetworkResources クラスの load メソッドのテストを作成せよ。

注2 ➔ ExpectedException —— 詳細な例外を扱う (p.150)

```
NetworkResourcesクラス
public class NetworkResources {

    NetworkLoader loader = new NetworkLoader();

    public String load() throws IOException {
        InputStreamReader reader = null;
        try {
            reader = new InputStreamReader(loader.getInput());
            StringBuilder str = new StringBuilder();
            char[] buf = new char[512];
            for (;;) {
                int len = reader.read(buf);
                if (len == -1) break;
                str.append(new String(buf, 0, len));
            }
            return str.toString();
        } finally {
            if (reader != null) reader.close();
        }
    }
}
```

```
NetworkLoaderクラス
public class NetworkLoader {
    public InputStream getInput() {
        // TODO 未実装
        return null;
    }
}
```

ヒント**設計**

- NetworkResources クラス、NetworkLoader クラスは問題で示したクラス設計とする

テストケース

- NetworkLoader が「Hello World」を返す InputStream を返すとき、NetworkResources の load メソッドは「Hello World」を返す

解答

テストコード

リスト20.6 外部システムに依存するテスト

```

public class NetworkResourcesTest {
    @Test
    public void loadでネットワークから取得した文字列を返す()
        throws Exception {
        // SetUp
        String expected = "Hello World";
        NetworkLoader mockLoader = mock(NetworkLoader.class);
        ByteArrayInputStream input
            = new ByteArrayInputStream(expected.getBytes());
        when(mockLoader.getInput()).thenReturn(input);
        NetworkResources sut = new NetworkResources();
        sut.loader = mockLoader;
        // Exercise
        String actual = sut.load();
        // Assertion
        assertThat(actual, is(expected));
    }
}

```

①
②

解説

ネットワークアクセスなど外部システムに依存する機能は期待する結果を制御することが困難です。そのような場合、NetworkResources クラスから外部システムに依存する機能を NetworkLoader クラスに抽出すれば、テストダブルで置き換えられるようになり、さまざまな状況下のユニットテストを行うことができます。

このとき、外部システムに依存する部分を可能な限り最小範囲で抽出すれば、テストダブルを利用することによる影響も小さく抑えることができます。

リスト 20.6 ①では、「Hello World」を返す InputStream オブジェクトを作成しています。そして、Mockito を利用して NetworkLoader クラスの load メソッドがその InputStream オブジェクトを返すように記録し、Network Resources オブジェクトに設定しています(**②**)。

なお、java.io.ByteArrayInputStream クラスと java.io.ByteArrayOutputStream クラスは、入出力系のテストを行う際に便利なクラスなので覚えておくといいでしょう。

20.4 インタフェースとスタブによるテスト 難易度:★★☆

あるアプリケーションのアカウント認証クラスを作成したい。アカウント情報はデータベースに保存されており、AccountDao インタフェースで Account クラスとして取得できる設計とした。

AccountDao インタフェースを利用する認証クラスを作成し、認証メソッドのテストを作成せよ。ただし、AccountDao の実装クラスはまだ実装されていないため、スタブを使用すること。

```
AccountDaoインターフェース
public interface AccountDao {
    /**
     * userIdを指定し、アカウント情報を取得する
     * @param userId システムで一意であるユーザID
     * @return 指定されたユーザIDのアカウント情報、存在しない場合はnull
    */
    Account findOrNull(String userId);
}

Accountクラス
public class Account {
    private String name;
    private String password;
    // Getter/Setterなどは省略
}
```

18
19
20

ヒント

設計

- Authentication クラスに、authenticate メソッドを定義する
- authenticate メソッドは、String 型の引数を 2 つ持ち、戻り値を Account 型とする

- Authentication クラスは、フィールドとして AccountDao オブジェクトを持つ
- authenticate メソッドでは、AccountDao オブジェクトから Account オブジェクトを取得する
- authenticate メソッドでは、Account オブジェクトが取得でき、かつパスワードが一致したならば、Account オブジェクトを返す

テストケース

- AccountDao が null を返す場合、authenticate メソッドは null を返す
- AccountDao が Account オブジェクトを返し、かつパスワードが一致する場合、authenticate メソッドは Account オブジェクトを返す
- AccountDao が Account オブジェクトを返し、かつパスワードが一致しない場合、authenticate メソッドは null を返す

解答

テストコード

リスト 20.7 インタフェースとスタブによるテスト

```
@RunWith(Enclosed.class)
public class AuthenticationTest {
    public static class アカウントが存在しない場合 {
        Authentication sut;
        @Before
        public void setUp() throws Exception {
            sut = new Authentication();
            sut.dao = mock(AccountDao.class);
            when(sut.dao.findOrNone("user001")).thenReturn(null);
        }
        @Test
        public void authenticateはnullを返す() throws Exception {
            assertThat(sut.authenticate("user001", "pw123"),
                       is(nullValue()));
        }
    }
    public static class アカウントが存在しパスワードが一致する場合 {
        Authentication sut;
        Account account;
        @Before
```

```

public void setUp() throws Exception {
    sut = new Authentication();
    sut.dao = mock(AccountDao.class);
    account = new Account("user001", "pw123");
    when(sut.dao.findOrNone("user001")).thenReturn(account);
}
@Test
public void authenticateはaccountを返す() throws Exception {
    assertThat(sut.authenticate("user001", "pw123"),
               is(sameInstance(account)));
}
}

public static class アカウントが存在するがパスワードが一致しない場合 {
    Authentication sut;
    Account account;
    @Before
    public void setUp() throws Exception {
        sut = new Authentication();
        sut.dao = mock(AccountDao.class);
        account = new Account("user001", "PW999");
        when(sut.dao.findOrNone("user001")).thenReturn(account);
    }
    @Test
    public void authenticateはnullを返す() throws Exception {
        assertThat(sut.authenticate("user001", "pw123"),
                   is(nullValue()));
    }
}
}

```

プロダクションコード

リスト20.8 認証クラス

```

public class Authentication {

    AccountDao dao = null;

    public Account authenticate(String userId, String password) {
        assert dao != null;
        Account account = dao.findOrNone(userId);
    }
}

```

18
19
20

```

    if (account == null) return null;
    return account.getPassword().equals(password) ? account : null;
}
}

```

解説

Web三層構造アーキテクチャでは、DAOなどのインターフェースを利用してデータベースにアクセスするパーシステンス層(永続化層)を分離した設計を行います^{注3}。

このような構造を採用した場合、サービス層のユニットテストでは、次の2つのユニットテストの戦略があります。

- ・データベースを使ってパーシステンス層まで通してテストする戦略
- ・依存するパーシステンス層をスタブで置き換えてサービス層を独立してテストする戦略

どちらの戦略を選択するかは、プロジェクトの規模や方針に依存します。スローテスト問題や開発体制上の問題がないのであれば、データベースやリアルオブジェクトを利用すべきです。なぜならば、よりプロダクション環境に近い環境でのユニットテストとなるためです。逆に、SQLExceptionの発生時に正しく例外処理を行っているかなど、外部システムに依存する検証はテストダブルを用いるほうが便利です。

リスト 20.7 では、AccountDaoオブジェクトをスタブオブジェクトで置き換えてテストしています。テストケースとしては3パターン行えば十分に網羅できるため、それぞれについてテストのコンテキストを定義しました。

注3 ⇒データベースを扱うソフトウェアの設計(p.199)

20.5 サーブレットのテスト

難易度:★★★

リクエストパラメータ「name」の値を取得し、「Hello <nameの値>」という文字列を出力する HelloServlet クラスと、そのテストを作成せよ。なお、リクエストは getのみ対応し、レスポンスの content-type に「text/plain; utf-8」を設定すること。

ヒント

設計

- javax.servlet.http.HttpServlet クラスのサブクラスとして、HelloServlet クラスを定義する
- doGet メソッドをオーバーライドし、HttpRequest オブジェクトから name パラメータを取得し、HttpResponse オブジェクトに「Hello <nameの値>」を出力する
- content-type に「text/plain; utf-8」を設定する

テストケース

- name パラメータに「JUnit」を設定した HttpRequest オブジェクトを引数に与えると、HttpResponse オブジェクトに「Hello JUnit」を書き出す
- HttpResponse オブジェクトの content-type に「text/plain; utf-8」が設定される
- HttpResponse オブジェクトの flushBuffer メソッドが呼ばれている

解答

テストコード

リスト20.9 サーブレットのテスト

```
public class HelloServletTest {
    @Test
    public void doGetでリクエストパラメータを含むテキストを出力する()
        throws Exception {
        // SetUp
        HelloServlet sut = new HelloServlet();
        HttpServletRequest request = mock(HttpServletRequest.class);
        when(request.getParameter("name")).thenReturn("JUnit");
        ServletOutputStream output = mock(ServletOutputStream.class);
```

18
19
20

```

    HttpServletResponse response = mock(HttpServletRequest.class);
    when(response.getOutputStream()).thenReturn(output);
    // Exercise
    sut.doGet(request, response);
    // Verify
    verify(output).println("Hello JUnit");
    verify(response).setContentType("text/plain; charset=UTF-8");
    verify(response).flushBuffer();
}
}

```

プロダクションコード

リスト20.10 HelloServletクラス

```

public class HelloServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String name = req.getParameter("name");
        resp.getOutputStream().println("Hello " + name);
        resp.setContentType("text/plain; charset=UTF-8");
        resp.flushBuffer();
    }
}

```

解説

HttpRequest インタフェースや HttpResponse インタフェースは、多くのメソッドが定義されているインターフェースです。このため、スタブクラスを作成するには多くのコードを書かなければなりません。そこで、モック用のライブラリを利用します。

リスト 20.9 では、Mockito を利用し HttpRequest と HttpResponse のモックオブジェクトを作成しています。出力される文字列の検証には、HttpResponse オブジェクトの ServletOutputStream オブジェクトが利用されるため、HttpResponse のモックオブジェクトから ServletOutputStream のモックオブジェクトを返すように設定しています。

ただし、リスト 20.9 のような `ServletOutputStream` オブジェクトの利用といった内部実装へ強く依存したモックの利用は、変更に対して脆くなるため可能な限り避けるべきです。

なお、`ServletOutputStream` オブジェクトにモックを使わずに、リスト 20.11 のようなスタブクラスを使うこともできます。`StringServletOutputStream` クラスを利用すれば、リスト 20.12 のように、モックへの依存度が小さくなります。

リスト 20.11 テスト用の `ServletOutputStream`

```
public class StringServletOutputStream extends ServletOutputStream {
    final StringBuilder out = new StringBuilder();

    @Override
    public void write(int b) throws IOException {
        out.append((char) b);
    }

    public String getOutput() {
        return out.toString();
    }
}
```

18
19
20

リスト 20.12 出力にモックを利用しないサーブレットのテスト

```
@Test
public void doGetでリクエストパラメータを含むテキストを出力する2()
throws Exception {
    // SetUp
    HelloServlet sut = new HelloServlet();
    HttpServletRequest request = mock(HttpServletRequest.class);
    when(request.getParameter("name")).thenReturn("JUnit");
    StringServletOutputStream output = new StringServletOutputStream();
    HttpServletResponse response = mock(HttpServletResponse.class);
    when(response.getOutputStream()).thenReturn(output);
    // Exercise
    sut.doGet(request, response);
    // Verify
    assertThat(output.getOutput(), is("Hello JUnit¥r¥n"));
}
```

```

    verify(response).setContentType("text/plain; charset=UTF-8");
    verify(response).flushBuffer();
}

```

20.6 Hello Worldのテスト

難易度:★★★

標準出力に「Hello World」を出力するメソッドとテストを作成せよ。

ヒント

設計

- Hello World クラスに、say メソッドを定義する
- say メソッドは、引数を持たず、戻り値は void 型とする

テストケース

- 標準出力(System.out)の println メソッドが「Hello World」を引数として呼び出されていること

解答

テストコード

リスト20.13 Hello Worldのテスト

```

public class HelloWorldTest {
    @Rule
    public SysoutSpy sysoutSpy = new SysoutSpy();
    @Test
    public void outputにHelloWorldがOutputされる() throws Exception {
        HelloWorld sut = new HelloWorld();
        sut.say();
        verify(sysoutSpy).println("Hello World");
    }

    class SysoutSpy implements TestRule {
        PrintStream spy;
        @Override
        public Statement apply(
            final Statement base, Description description) {
            return new Statement() {

```

```
    @Override
    public void evaluate() throws Throwable {
        PrintStream out = System.out;
        spy = Mockito.spy(out);
        try {
            System.setOut(spy);
            base.evaluate();
        } finally {
            System.setOut(out);
        }
    };
}
```

プロダクションコード

リスト20.14 HelloWorldクラス

```
public class HelloWorld {  
    public void say() {  
        System.out.println("Hello World");  
    }  
}
```

18
19
20

解說

「Hello World」は、プログラミングを学ぶときに書く定番のプログラムです。標準出力に文字列を表示するだけの簡単なプログラムですが、ユニットテストの対象としては簡単ではありません。なぜならば、Hello Worldはテスト対象となるメソッドが戻り値を返さないからです。また、テスト対象クラスが状態を持つわけでもないため、アサーションも困難です。

Hello Worldでは、標準出力(System.out)に文字列を出力します。このとき、ターミナルなどの外部システムに影響を与えますが、プログラムには何も情報を返しません。しかしながら、出力処理をスタブなどで置き換えてしまうと、標準出力に出力されなくなります。このような場合、テスト

ダブルのひとつであるスパイ^{注4}が有効です。

スパイは、対象のオブジェクト(ここではPrintWriter)をラップし、各メソッドの実行を監視するオブジェクトです。Mockitoのスパイを利用するとい、スパイオブジェクトにどんなメソッドが実行されたかを検証できます。

リスト20.13では、sayメソッドの実行後、PrintWriterに対しprintlnメソッドが実行されているかを検証しています。また、ルールを利用して、標準出力をスパイに差し替える処理をテストメソッドから分離しました。このため、テストコードにテストダブルに関連する処理が混在せず、可読性の高いコードとなっています。

なお、モックと同様、スパイによるテストは内部実装に依存します。たとえば、プロダクションコードが次のような実装に変更されるとテストが失敗します。

```
public void say() {
    System.out.print("Hello");
    System.out.println(" World");
}
```

モックやスパイは非常に強力なテクニックですが、テストコードがテスト対象に強く依存するようになるため、テストが脆くなるというデメリットを持ちます。既存の実装を変更できないなどの理由がない限りは、設計を工夫して、利用せずに済むようにすることを優先させてください。

^{注4} ⇒スパイ——依存オブジェクトの呼び出しを監視する(p.187)

付録

本書利用環境のまとめ

付録F

Jenkinsの設定

付録E

Android開発環境のセットアップ

付録D

H2 Databaseの便利機能の設定

付録C

Eclipseの便利機能の設定

付録B

開発環境のセットアップ

付録A

付録A

開発環境のセットアップ

ここでは、本書のサンプルコードを実行するために必要な開発環境のセットアップについて簡単に解説します。

A.1 JDKのセットアップ

JDK(*Java Development Kit*)はコンパイラやドキュメントを含むJavaの開発環境です。Javaのプログラムを実行するにはJRE(*Java Runtime Environment*)というJavaの実行環境も必要ですが、JDKにはJREが同梱されているため、JDKをインストールすればJREのインストールは不要です。

Javaのバージョン

本書のサンプルコードはJava 6で動作確認を行っています。なお、Java 5についても言語仕様的にはJava 6と大きく変わらないため、実行可能でしょう。

● Java 1.4以前について

本書で扱っているJUnitの機能の多くはJava 5で導入されたアノテーションを利用しています。このため、Java 1.4以前ではサンプルコードは実行できません。

● Java 7について

2012年現在、Javaの最新バージョンとしてJava 7がリリースされています。しかしながら、開発現場ではしばらくの間はJava 6が主流であること、Java 7に対応した入門書などが少ないと、Mac OS Xでは10.7(Lion)以降でなければJava 7がサポートされていないことなどの理由から、本書ではJava 6をデフォルトの動作環境としました。

A
B
C
D
E
F

なお、Javaは後方互換性の高い言語仕様です。Java 7を利用した場合でも、サンプルコードは実行できます。ただし、EclipseなどのIDEでは言語仕様の変更に伴う警告メッセージが表示されることがあります。

JDKのインストール

Windows環境とMac OS X環境でのJDKのインストール方法を解説します。

● Windows環境の場合

Windows環境でJDKをインストールするにはJDKのインストーラをダウンロードして実行します。

Oracle社のWebサイト^{注1}にアクセスして、「ダウンロード」メニューをクリックしてください(図A.1)。

次に、表示されたページからJavaのカテゴリを探し、「Java SE」をクリックします。続けて、誘導に従ってUSサイトへのリンクをクリックします^{注2}(図A.2)。

ここでは、アクセスした時点で最新バージョンのJDKがダウンロードできます。本書では、Java 6をインストールするため、「Java SE 6 Update XX」を探します^{注3}。

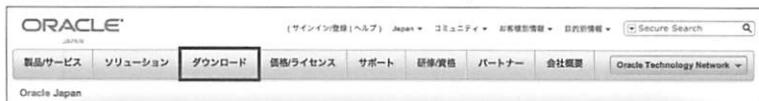
Java SE 6を見つけたならば、JDKのダウンロードリンクをクリックします。すると、ライセンスへの同意と利用するプラットフォームの選択が求められます(図A.3)。

注1 <http://www.oracle.com/jp/>

注2 本書の執筆時点では、日本語サイトからJDKのダウンロードはできません。

注3 もし、このページにJava SE 6が見つからない場合は、「Previous Releases」のリンクをクリックして探してください。

図A.1 Oracle社のWebサイト(日本語ページ)



付録

最初に「Accept License Agreement」のラジオボタンをチェックし、32ビットのWindowsであれば「Windows x86」を、64ビットのWindowsであれば「Windows x64」をダウンロードします。

ダウンロードが完了したらダブルクリックしてインストーラを起動し、指示に従ってインストールしてください。

図A.2 Java SEのダウンロードページ(USサイト)

The screenshot shows the Oracle Java SE Downloads page. At the top, there's a navigation bar with links for Products and Services, Solutions, Downloads, Store, Support, Training, Partners, and About. A search bar is also at the top right. The main content area has tabs for Overview, Downloads, Documentation, Community, Technologies, and Training. Under the Downloads tab, there are four buttons: Latest Release, Next Release (Early Access), Embedded Use, and Previous Releases. Below these buttons are two download options: 'Java Platform (JDK) 7u7' and 'JavaFX 2.2'. To the right, there's a sidebar titled 'Java SDKs and Tools' which includes links for Java SE, Java EE and Glassfish, Java ME, Java FX, Java Card, NetBeans IDE, and Java Resources. The Java Resources section is currently expanded, showing links for New to Java?, APIs, Code Samples & Apps, Developer Training, Documentation, Java.com, and Java.net.

図A.3 ライセンスの同意とプラットフォームの選択

This screenshot shows a modal dialog for accepting the Oracle Binary Code License Agreement. It contains two radio buttons: 'Accept License Agreement' (which is selected) and 'Decline License Agreement'. Below the buttons is a table with columns for Product / File Description, File Size, and Download. The table lists various Java SE versions for different platforms:

Product / File Description	File Size	Download
Linux x86	65.42 MB	jdk-6u35-linux-i586-rpm.bin
Linux x86	68.43 MB	jdk-6u35-linux-i586.bin
Linux x64	65.65 MB	jdk-6u35-linux-x64-rpm.bin
Linux x64	68.7 MB	jdk-6u35-linux-x64.bin
Solaris x86	68.34 MB	jdk-6u35-solaris-i586.sh
Solaris x64	119.88 MB	jdk-6u35-solaris-i586.tar.Z
Solaris x64	8.44 MB	jdk-6u35-solaris-x64.sh
Solaris x64	12.18 MB	jdk-6u35-solaris-x64.tar.Z
Solaris SPARC	73.33 MB	jdk-6u35-solaris-sparc.sh
Solaris SPARC	124.6 MB	jdk-6u35-solaris-sparc.tar.Z
Solaris SPARC 64-bit	12.12 MB	jdk-6u35-solaris-sparcv9.sh
Solaris SPARC 64-bit	15.41 MB	jdk-6u35-solaris-sparcv9.tar.Z
Windows x86	69.71 MB	jdk-6u35-windows-i586.exe
Windows x64	59.71 MB	jdk-6u35-windows-x64.exe

● Mac OS X 環境の場合

Mac OS X では、歴史的経緯から Java 実行環境の開発が Apple 社で行われてきました。このため、Mac OS X 環境でバージョン 6 の JDK をインストールするには Apple 社のサポートサイトからインストーラをダウンロードします。

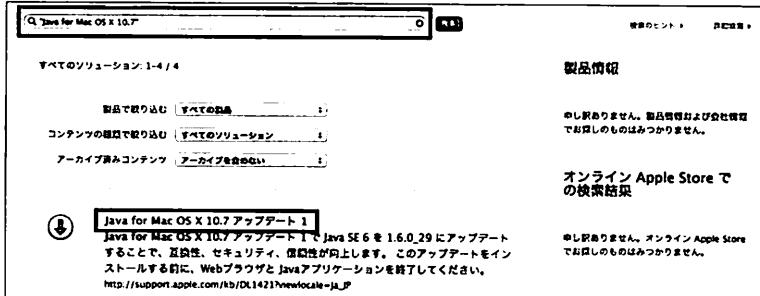
Apple 社のサポートサイト^{注4}にアクセスし、「"Java for Mac OS X 10.X"」で検索してください(ダブルクォーテーション込み)。たとえば、Mac OS X 10.7(Lion)を利用しているならば、「"Java for Mac OS X 10.7"」で検索します。すると「Java for Mac OS X 10.7 アップデート X」が見つかるのでこれをダウンロードしてください(図 A.4)。

ダウンロードが完了したら、ダブルクリックでインストーラを起動し、指示に従いながらインストールしてください。Mac OS X の Java は、/System/Library/Frameworks/JavaVM.framework/ にインストールされます。

なお、Java 7 のインストールについては、Windows 環境の場合と同様に Oracle 社の Web サイトからインストーラをダウンロードして実行します。ただし、Mac OS X 10.7(Lion) 以降が必要です。

注4 <http://www.apple.com/jp/support/>

図 A.4 Java for Mac OS X のダウンロード



A
B
C
D
E
F

A.2 Eclipseのセットアップ

本書では、IDE(*Integrated Development Environment*：統合開発環境)のデファクトスタンダードであるEclipseを利用します。ここでは、Eclipseのインストールと日本語化の手順について紹介します。

Eclipseのバージョン

本書執筆時点でのEclipseの最新バージョンは4.2ですが、本書ではバージョン3.7.2を用いて開発環境の構築を行います。

Eclipseは年に1回のアップデートを行っており、2012年にメジャーバージョンが3.X系から4.X系に切り替わりました。近年はバージョンアップ時にプラグインが追従できず利用できないというケースは減りましたが、まだ一部のプラグインが最新バージョンに対応していないこともあります。今回は安定しているバージョンを利用しました。

Eclipseのインストール

Eclipseをダウンロードするには、eclipse.org^{注5}にアクセスして「Downloads」をクリックしてください。^{注6}

ダウンロードページでは、最新版のリストが表示されるため、画面右側のリンクから「Older Versions」をクリックします。次に「Eclipse Indigo SR2 Packages (v 3.7.2)」をクリックしてパッケージを表示します。

Eclipseには開発するソフトウェアの種類ごとにパッケージが用意されています。本書で扱う範囲の開発であれば「Eclipse IDE for Java Developers」で十分ですので、利用するプラットフォームに合わせたリンクをクリックしてダウンロードしてください(図A.5)^{注7}。

注5 <http://www.eclipse.org/>

注6 本書では「Pleiades All in One」は用いません。理由は後述します。

注7 「Eclipse Classic」は、EclipseのプラットフォームのみでJavaの開発環境を含みません。「Java EE Developers」はJava Enterprise Edition向けの開発環境であり、Webアプリケーション開発用途でなければ不要です。

● インストール先について

ファイルをダウンロードし、適当なフォルダに展開してください。Eclipseは展開したフォルダ内で完結していますので、フォルダごと移動するのであればどこに配置してもかまいません(アンインストールする場合も、フォルダを削除するだけです)。なお、Windows環境の場合、不要なトラブルを避けるため、デスクトップや、フルパスに日本語や空白スペースが含まれる場所には配置しないでください。

Eclipseをダウンロードして展開したならば、パッケージに含まれる `eclipse.exe`(Macの場合は `Eclipse`)から起動できます。

Eclipseの日本語化

現在、公式サイトで配布している Eclipse は国際化対応されていません。また、公式の言語パックも配布されていないため、メニュー やメッセージなどを日本語化するにはサードパーティー製の日本語化プラグインを利用します。

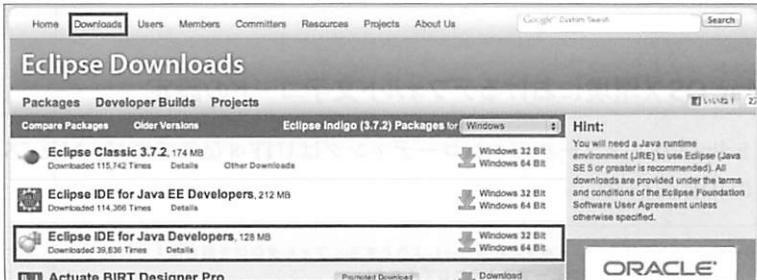
● Pleiades のインストール

Eclipseを日本語化するには、MergeDoc Project^{注8}で配布している Pleiades Eclipse プラグインをインストールします。

まず、プロジェクトの Web サイトの「Pleiades ダウンロード」から「最新版」

注8 <http://mergedoc.sourceforge.jp/>

図A.5 Eclipseのダウンロード



A
B
C
D
E
F

のアーカイブをダウンロードします。そしてダウンロードしたアーカイブを解凍し、plugins フォルダの中身と features フォルダの中身を Eclipse のインストール先の同名フォルダの中にコピーします^{注9}。

次に、Windows 環境の場合は、Eclipse のインストールフォルダにある eclipse.ini ファイルをテキストエディタで開き、最終行に次の行を追加します。

```
-javaagent:plugins/jp.sourceforge.mergedoc.pleiades/pleiades.jar
```

Mac OS X 環境の場合は、Eclipse のアイコン (Eclipse.app) を右クリックし、コンテキストメニューから [パッケージの内容を表示] を選択し、Contents/MacOS/eclipse.ini ファイルをテキストエディタで開き、最終行に次の行を追加します。

```
-javaagent:../../../../plugins/jp.sourceforge.mergedoc.pleiades/pleiades.jar
```

● Pleiades All in One について

MergeDoc Project で配布されている「Eclipse 3.X Pleiades All in One」は、日本語化された Eclipse にさまざまなプラグインをパッケージングしたバージョンで、初めて Eclipse を利用しようという場合には手軽に利用できます。

ただし、Windows 環境向けのパッケージしか提供されていないことと、ベースとなる Eclipse が「Eclipse IDE for Java Developer」ではなく、「Eclipse IDE for Java EE Developer」のため、本書で扱う範囲の開発では不要な機能も多く、高いマシンスペックも要求します。

このため、本書では、Pleiades All in One ではなく、Eclipse + Pleiades プラグインを使用しました。

Mac OS X 環境におけるデフォルト文字コードの設定

Eclipse のコンソールのエンコーディングは UTF-8 なのですが⁹、Mac OS X の JDK 6 はデフォルトエンコーディングが MS932 (Shift_JIS) です。このた

^{注9} Mac OS X の場合、フォルダごとコピーしようとするとフォルダ自体を置き換えてしまうので、必ずファイルを選択してコピーするようにしてください。

め、日本語のクラス名などを利用するときに不都合が生じることがあります。そこで、Eclipseの起動設定を修正し、デフォルトエンコーディングをUTF-8にすることをお勧めします。

デフォルトエンコーディングを変更するには、eclipse.iniファイルを開き、次の行を追加します。

```
-Dfile.encoding=UTF-8
```

Column

Eclipseは英語版のまま使おう

本書では、Eclipseを日本語化して解説していますが、筆者の本音としては英語版のまま使用することをお勧めします。単純に処理が重くなることも理由のひとつですが、それ以上に環境設定やショートカットが扱い難くなることが理由です。

Eclipseでは多くの項目を環境設定で設定できます。このとき、フィルタ機能を使うことで簡単に設定項目を探すことができます。しかしながら、フィルタ機能は日本語に対応していません。たとえば、staticインポートの設定を変更したいなら、「static」とフィルタに入力することで簡単に絞り込みを行えます。しかしながら、「静的」と入力しても該当する設定項目を見つけることができません。

また、ショートカットのほとんどは英語表記でのコマンドから先頭文字が選択されています。たとえば、よく利用する [Ctrl]+[Shift]+[O] の [O] は「Organize Imports」の [O] です。日本語訳された「インポートの編成」だと、なぜ [O] なのか覚えにくいですが、英語のままだと違和感なく覚えられるでしょう。

また、すべてのメッセージが翻訳されるわけではありません。未対応のプラグインでは、翻訳されていることで意味がわからなくなる場合もあります。バージョンアップなどで一部のメッセージや項目のみが英語となる場合もあります。

実際に英語版を使ってみると、それほど困ることはありません。むしろ、環境設定などで目的の設定項目を探しやすかったり、英語のドキュメントを読むときに用語の変換を行わなくともよいなど、多くのメリットがあります。

エンジニアであれば、英語ドキュメントは避けて通れない道です。幸いにも Eclipseで使用されている英単語は簡単なものばかりです。Eclipseを日本語化して操作に慣れたならば、英語版での利用をお勧めします。

A
B
C
D
E
F

付録B

Eclipseの便利機能と設定

Eclipseは、Java開発におけるIDEとしてのデファクトスタンダードです。Javaの開発効率は、Eclipseをどれだけ使いこなせるかに依存するといっても過言ではありません。これは、テストコードを書く場合であっても同様です。ユニットテストに便利な機能を覚え、Eclipseを使いやすく設定すれば、リズムよくテストコードを書くことができるでしょう。

そこで、ここではユニットテストに便利なEclipseの便利機能や設定を紹介します。

B.1 EclipseのJUnitサポート

EclipseのJavaプロジェクトでJUnitなどの外部ライブラリを利用する場合、通常はJARファイルをダウンロードし、クラスパスに追加します。ですが、Eclipseには共有ライブラリとしてJUnitが組み込まれているため、簡単な設定でJUnitを外部ライブラリとして利用することもできます。

ただし、通常の開発ではMavenなどを使い外部ライブラリを管理してください^{注1}。なお、Eclipseの共有ライブラリに含まれるJUnitのバージョンは、Eclipseのバージョンに依存します^{注2}。

また、EclipseではJUnitのテストプログラムを簡単に実行できます。プロジェクトを選択してJUnitテストを実行すれば、プロジェクトに含まれるすべてのJUnitテストケースを収集してテストを実行できます。テストクラスを選択してJUnitテストを実行すれば、選択したテストクラスが実行できます。アウトラインなどからテストメソッドを選択してJUnitテストを実行すれば、選択したテストメソッドのみが実行できます。

注1 ➔ Mavenによるビルドプロセスの自動化(p.277)

注2 Eclipse 3.7.2では、JUnit 4.8.2とJUnit 3.8.2が共有ライブラリとして利用できます。

B.2 テキストファイルのエンコーディング設定

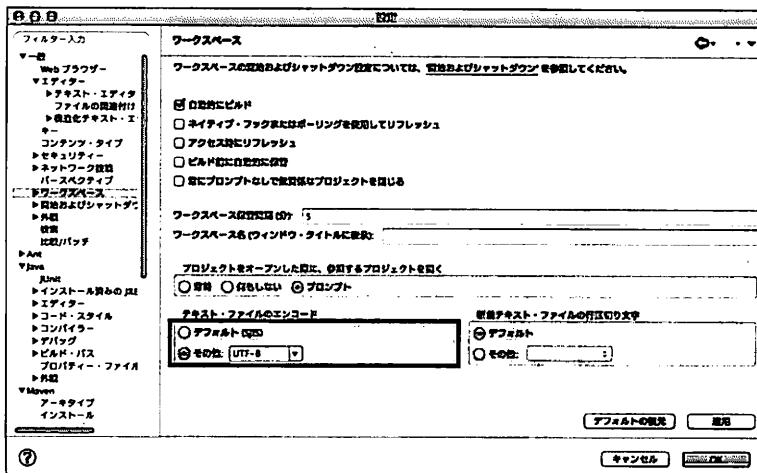
Eclipseではプラットフォームのデフォルトエンコーディングが、ソースコードファイルのデフォルトエンコーディングとなります。このため、Windowsや最近のMac OS XではデフォルトがMS932(Shift_JIS)です。しかしながら、特別な事情がない限り、ソースコードやリソースファイルのエンコーディングはUTF-8とすることが無難です。

テキストファイルのデフォルトエンコード設定を変更するには、Eclipseの設定画面から[一般]-[ワークスペース]を開き、「テキスト・ファイルのエンコード」の「その他」で「UTF-8」を選択します(図B.1)。

B.3 staticインポートのワイルドカード

Eclipseでは、インポートの編成(**Ctrl**+**Shift**+**Q**)で使用されていないクラスのimport文を整理できます。このとき、Eclipseのデフォルト設定では、import文でワイルドカードを使用しません。この設定は同名クラスのコンフリクトを避けることが目的であり、一般的なコーディング標準として知られています。

図B.1 テキストファイルのエンコーディング設定



A
B
C
D
E
F

ところが、JUnitを利用してテストコードを書く場合、AssertionクラスやMatcherクラスでstaticインポート^{注3}を多用します。このとき、ワイルドカードを使用しない設定でコンフリクトを避けるメリットよりも、ワイルドカードを使ってさまざまなstaticメソッドにアクセスできるメリットのほうが大きく感じます。このため、staticインポートに限り、import文のワイルドカードを許可する設定に変更することをお勧めします。

設定方法は、設定画面から[Java]-[コード・スタイル]-[インポートの編成]を開き、「.*に必要な静的インポート数」の値を1に変更します(図B.2)。これにより、static import文ではワイルドカードが優先されて使用されます。

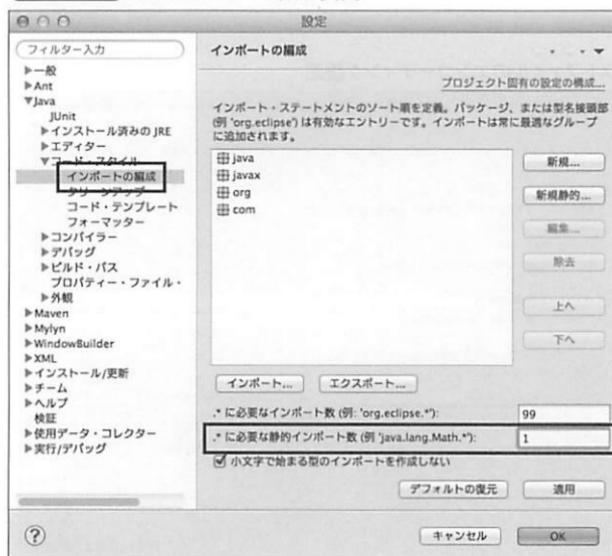
B.4 Quick JUnit

Quick JUnit^{注4}は、JUnitの起動をより簡単にし、テストコードとプロダクションコードを切り替えることができるEclipseのプラグインです。

注3 日本語化されたEclipseなどでは「静的インポート」と訳されています。

注4 <http://quick-junit.sourceforge.jp/>

図B.2 staticインポート数の変更



Quick JUnit プラグインをインストールすれば、テストクラスとテスト対象クラスとの切り替えがショートカットでできるようになります。また、テストコードからショートカットでユニットテストを実行できるようになります。

このシンプルで強力な2つのショートカットにより、ユニットテストのコーディングとテストの実行をリズムよく進められるようになります。地味な機能と感じるかもしれませんのが、ユニットテストを書いていく中でリズムよくコードを書いていくことは重要です。想像以上の効果があるため、Eclipseで開発をするのであれば、必ずインストールするべきプラグインです。

Quick JUnit プラグインをインストールする

Quick JUnit をインストールするには、Eclipse のメニューから [ヘルプ] - [Eclipse マーケットプレース] を選択し、「Quick JUnit」で検索します(図B.3)。Quick JUnit が見つかったならば、[インストール] ボタンを押してインストールしてください。

図B.3 QuickJUnitのインストール



A
B
C
D
E
F

テスティングペアを開く

Quick JUnitにより、プロダクションコードとテストコードをショートカットで切り替えることができます。

エディタでプロダクションコードにフォーカスがあたっている状態で、ショートカット **[Ctrl]+[G]** を入力すると、対応するテストコードに切り替えることができます。もし、対応するテストコードが存在しないならば、テストクラスの作成ダイアログが表示されます(図B.4)。

逆に、エディタでテストコードにフォーカスがあたっている状態で、ショートカット **[Ctrl]+[G]** を実行すれば、対応するプロダクションコードに切り替えることができます。

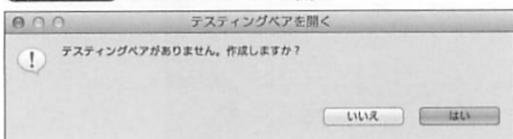
ただし、プロダクションコードとテストコードは、一定の命名規約に従って作成されなければなりません。プロダクションコードのクラス名に対して、テストコードのクラス名は必ず「<プロダクションコードのクラス名>Test」としてください。

テストを実行する

Eclipseではショートカット **[Ctrl]+[Alt]+[X]-[T]** でユニットテストを実行できます。テストコードにフォーカスがあたっている状態でテストを実行すれば、そのテストクラスに定義されたテストがすべて実行されます。しかしながら、テストメソッド単位でテストを実行するためには、アウトラインビューなどからテストメソッドを選択しなければなりません。

Quick JUnitがインストールすれば、ショートカット **[Ctrl]+[0]** でユニットテストを実行できます。このとき、テストメソッドの中にカーソルがあるならば、そのテストケースのみが実行されます。テストメソッドの外にカ

図B.4 テスティングペアを開く



ソルがある場合は、テストクラスに定義されたすべてのテストケースが実行されます。

テストコードを書き、すぐに実行できるため、テストメソッドを指定して実行するショートカットは非常に便利です。

B.5 コンテンツアシストとお気に入り

「コンテンツアシスト」と呼ばれるコード補完機能(ショートカット **[Ctrl]+[Space]**)は、Eclipse を代表する機能です。

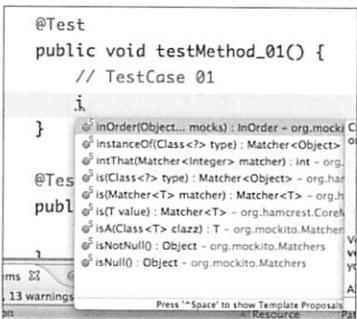
コンテンツアシストでは、コードの文脈などから適切な候補をプログラマに提示します(図B.5)。プログラマは、Eclipse が提示した候補から適切な項目を選択するだけで、面倒なコードの入力をすばやく正確に行えます。これは、Java の開発では欠かせない機能です。

JUnit を使ってユニットテストを書くときには、アサーション用のメソッドがコンテンツアシストの候補として表示されると便利です。このため、コンテンツアシストの「お気に入り」の設定を行います。

お気に入りの設定は、設定画面から [Java]-[エディター]-[コンテンツ・アシスト]-[お気に入り]で行います(図B.6)。ここで追加したいクラスを、[新規タイプ]から追加してください。

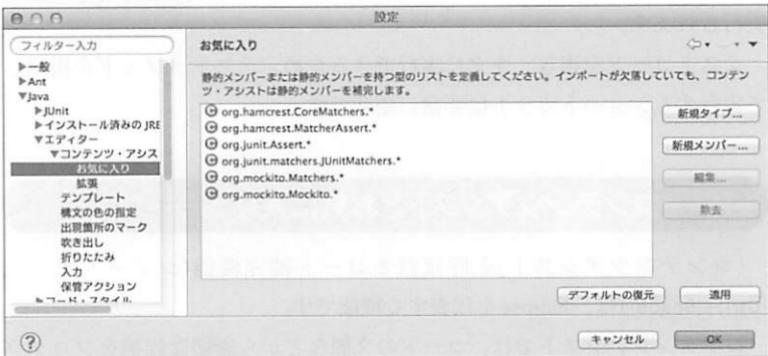
なお、Quick JUnit をインストールすれば、次のクラスが自動的に追加されます。

図B.5 コンテンツアシストによるstaticメソッドの表示



A
B
C
D
E
F

図B.6 お気に入り



- org.hamcrest.CoreMatchers
- org.hamcrest.MatcherAssert
- org.junit.Assert
- org.junit.matchers.JUnitMatchers
- org.mockito.Matchers
- org.mockito.Mockito

必要に応じて追加や削除を行って、Eclipseをカスタマイズしてください。プロジェクト固有のMatcherを定義した場合にも、お気に入りに追加しておくと便利です。

B.6 テンプレート

Eclipseでは、よく利用するコードスニペットをテンプレートとして登録できます。登録したテンプレートは、テンプレート名を入力し、コンテンツアシスト(**Ctrl**+**Space**)から呼び出すことができます。

たとえば、「System.out.println();」は「sysout」というテンプレート名で登録されているので、「sys」と入力して**Ctrl**+**Space**を押すと、コンテンツアシストの候補としてコードスニペットが表示されます^{注5}。

ユニットテストでは、テストメソッドの宣言やアサーションなど、似た

^{注5} テンプレート名は途中まで入力すれば、候補として選択できます。

A
B
C
D
E
F

コードを多く記述します。このため、よく使うコードスニペットをテンプレートとしてEclipseに登録すれば、タイピング時間の節約になり、リズムよくコーディングできるでしょう。

Eclipseのテンプレートの設定は、設定画面から[Java]-[エディター]-[テンプレート]で行います(図B.7)。

テンプレートを追加するには[新規]ボタンを押して新規テンプレートダイアログを表示し、名前、説明、パターンを入力します(図B.8)。

図B.7 テンプレートの設定



図B.8 新規テンプレート



テンプレートのパターンは、Javaのソースコードに近いフォーマットで記述します。さらに、特定のクラスをインポートさせたり、テンプレートを挿入するときにカーソルの位置を制御したりと柔軟な設定ができます。

また、Quick JUnitをインストールすると、ユニットテストで便利なテンプレートが追加されます。

なお、Eclipseの標準テンプレートとして、JUnit 3系のスタイルのテンプレートも設定されています。これらはJava 1.4以下の環境で開発をしないのであれば無効にしてよいでしょう。無効とすれば、コンテンツアシストで候補として表示されなくなります。

以下、筆者が利用しているテンプレートを紹介します。なお、このテンプレートは本書のサンプルコードと合わせてダウンロードできます。ご自由にインポートしてお使いください。

test—— テストメソッド

```
@${testType:newType(org.junit.Test)}
public void ${testCase}() throws Exception {
    // SetUp
    // Exercise
    // Verify
}
```

「test」は使用頻度の高いテンプレートとなるため、使いやすいうようにカスタマイズしたいテンプレートです。このテンプレートは、Eclipseで標準設定されているテンプレートですが、4フェーズテスト^{注6}を反映し、コメントを追加しています^{注7}。

また、デフォルトではfailメソッドが挿入され、初期状態でテストが失敗するようになっていますが、筆者はテストメソッドを作成したらすぐにテストコードを書くため、省略しています。

注6 ➔4フェーズテスト(p.47)

注7 TearDown(後処理)は使用頻度が低く、行わないほうがよいテストであるため省略しています。

A
B
C
D
E
F

at —— アサーション

```
assertThat(${actual:localVar(java.lang.Object)},
           is(${expected:localVar(java.lang.Object)}));
${junit:importStatic('org.junit.Assert.assertThat')}
${hamcrest:importStatic('org.hamcrest.CoreMatchers.is')} ]
```

実際は1行

「at」は assertThat の略で、最も使用頻度の高いテンプレートです。「at」と入力し、コンテンツアシストを使えば、「assertThat(actual, is(expected));」というアサーションの基本的な型を挿入できます。また、このテンプレートを利用すると、static インポートに org.junit.Assert クラスと org.hamcrest.CoreMatchers クラスが追加されます。

このテンプレートを有効に活用するために、テストメソッドでの期待値と実測値の変数名を「actual」と「expected」に統一します。変数名を統一すれば、変数名をテストメソッドごとに考慮する必要がありません。また、テストコードの可読性も高くなります。そして、テンプレートを利用した場合に、テストメソッドごとに変数名を変更する必要もありません。

一般的には、変数名は意味を表す具体的な名前を付けることが推奨されますが、テストメソッドの中では期待値であるか実測値であるかという意味を優先しましょう^{注8}。

setUp —— 初期化メソッド

```
@${testType:newType(org.junit.Before)}
public void setUp() throws Exception {
    ${cursor}
}
```

「setUp」は、テストクラスで共通の初期化処理を行うメソッドのテンプレートです。共通の初期化処理は、テストコードから重複を減らす最も効果的な方法であるため、使用頻度の高いテンプレートです。

注8 ➔ 実測値と期待値(p.44)

tearDown —— 後処理メソッド

```
@${testType:newType(org.junit.After)}
public void tearDown() throws Exception {
    ${cursor}
}
```

「tearDown」は、テストクラスで共通の後処理を行うメソッドのテンプレートです。共通の後処理は、共通の初期化処理に比べ、あまり使用されません。

setUpClass —— クラスの初期化メソッド

```
@${testType:newType(org.junit.BeforeClass)}
public static void setUpClass() throws Exception {
    ${cursor}
}
```

「setUpClass」は、setUpと同様にテストクラスで共通の初期化処理を行うメソッドのテンプレートです。ただし、テストクラスに定義されたすべてのテストが実行される前に1回だけ実行される初期化処理のテンプレートです。このテンプレートはあまり使用されません。

tearDownClass —— クラスの後処理メソッド

```
@${testType:newType(org.junit.AfterClass)}
public static void tearDownClass() throws Exception {
    ${cursor}
}
```

「tearDownClass」は、tearDownと同様にテストクラスで共通の後処理を行うメソッドのテンプレートです。ただし、テストクラスに定義されたす

べてのテストが実行されたあとに1回だけ実行される共通の後処理のテンプレートです。このテンプレートもほとんど使用されません。

when —— ネストしたテストクラス

```
public static class ${TestContext} {
    ${cursor}
}
```

「when」は、Enclosed テストランナーを利用したテストにおけるネストしたテストクラスのテンプレートです。Enclosed テストランナーを使うことで、テストケースを共通の初期化処理ごとに束ねたネストクラスを作成できます。これにより、テストクラスがコンテキストで構造化され、見通しの良いテストコードとなります^{注9}。

ネストしたクラスの名前は、そのテストの条件や状況であるため、テンプレートの名前を「when」としています。テストクラス名は「～の場合」という名前にするとわかりやすいでしょう。

instantiation —— インスタンス化テスト

```
public static class インスタンス化テスト {
    @${testType:newType(org.junit.Test)}
    public void ${デフォルトコンストラクタ}() throws Exception {
        // Exercise
        ${t:newType(java.lang.Object)} ${instance:localVar(java.lang.
        Object)} = new ${t}(); 実際は1行
        // Verify
        assertThat(instance, is(notNullValue())); 実際は1行
        ${junit:importStatic('org.junit.Assert.assertThat')}
        ${hamcrest:importStatic('org.hamcrest.CoreMatchers.is')}_
    }
}
```

^{注9} ➡ テストのコンテキスト(p.92)

A
B
C
D
E
F

「instantiation」は、インスタンスの生成と初期値の検証を行うテストケースのテンプレートです。このテンプレートも、Enclosed テストランナーを利用したコンテキストごとのテストを行うときに利用します。

B.7 次の注釈

ショートカット **[Ctrl]+[.]**(ピリオド)で実行できる「次の注釈」は、ソースコードにエラーや警告があるときに、カーソルを最も近いエラーや警告まで移動する機能です。

これだけでは強力な機能と感じられませんが、後述のクリックフィックスと組み合わせることでコーディングの効率が劇的に変化します。エラーや警告に移動し、クリックフィックスで修正するという流れでコーディングを行うと、Javaの静的な性質とコンパイラのパワーを十分に活かすことができます。

B.8 クリックフィックス

ショートカット **[Ctrl]+[1]**で実行できる「クリックフィックス」は、ソースコードの警告やエラーを半自動的に修正する機能です。

警告やエラーのある個所にカーソルがある状態で、クリックフィックスを起動すると、Eclipseはコンパイルエラーや警告の内容から、修正候補をいくつか提示します。これは、Javaの強い型付けと冗長な構文を味方にすることができる強力な機能です。

たとえば、あるクラスが名前を解決できずにエラーとなっているとします。この状態でクリックフィックスを実行すると、型名から問題を類推し、新しいクラスの作成や似た名前のクラスへの変更、該当クラスのインポートなどを、修正候補として表示してくれます(図B.9)。修正候補の中から適切な修正項目を選択すれば、半自動的に問題は修正されるでしょう。

なお、クリックフィックスは、エラーや警告がない状態でも実行できます。たとえば、変数にカーソルがある状態でクリックフィックスを実行すれば、その変数の名前変更や変数のフィールド変換などが修正候補として表示されます。

A
B
C
D
E
F

クリックフィックスは、前述の「次の注釈」機能と組み合わせることで、テストコードやプロダクションコードの実装を効率良く進めることができます。特に、テストコードを先に書くテストファースト^{注10}を行うならば、次の注釈とクリックフィックスを繰り返すことでプロダクションコードをすばやく実装できます。

B.9 クリックアウトライン

テストコードは、プロダクションコードに比べて多くのメソッドで構成され、コード行数も多くなりがちです。また、コンテキストごとにネストクラスを作成して構造化した場合^{注11}、テストクラス全体を把握できると便利です。

クラスのアウトライン表示はテストクラスの構造化と相性が良い機能です。Eclipseでは、「アウトラインビュー」^{注12}を利用することで、クラスのアウトラインを表示できます(図B.10)。アウトラインビューにはクラスに定義されたメソッドの一覧が表示され、目的のメソッドへすばやく移動することができます。

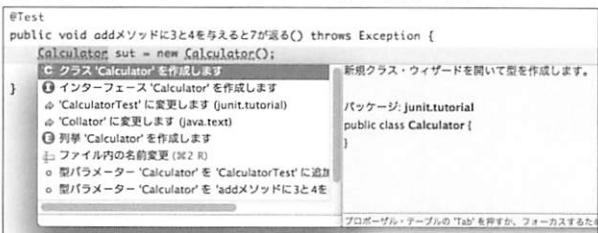
このアウトラインを簡単に表示する機能が、ショートカット **Ctrl+O** で表示できる「クリックアウトライン」です。ソースコードにカーソルがある状態でクリックアウトラインを実行すると、アウトラインがポップアップ

注10 ➔ テスト駆動開発とは？(p.308)

注11 ➔ テストのコンテキスト(p.92)

注12 アウトラインビューを表示するには、Eclipseのメニューから[ウィンドウ]—[ビューの表示]—[アウトライン]を選択します。

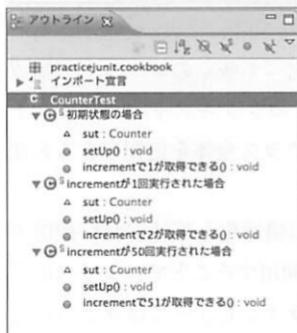
図B.9 クリックフィックスによる修正候補の表示



ウィンドウに表示されます(図B.11)。

クリックアウトラインでは、アウトラインビューと同様にクラスの構造が一目で確認できます。また、目的のメソッドを選択すれば、そのメソッドに移動できます。フィルタ条件を入力することで目的のメソッドをすばやく見つけることもできます。

図B.10 アウトラインビュー



図B.11 クイックアウトライン



付録C

H2 Databaseのセットアップと使い方

本書の第12章では、データベースのユニットテストを説明するためにDbUnitとH2 Databaseを紹介しました。ここでは、H2 Databaseのセットアップと基本的な使い方について紹介します。

C.1 H2 Databaseの特徴

H2 Databaseは、軽量で組込み用途でも簡単に利用できるオープンソースのデータベースです。

H2 DatabaseはJavaで記述されているため、サーバとして別プロセスで実行するだけでなく、Javaプログラム内でインメモリデータベース^{注1}として利用することもできます。このため、デスクトップアプリケーションや、データ量が少ないアプリケーションなどで利用するならば最適なデータベースのひとつです。

また、各種データベースのSQL方言と互換性を持つという興味深い特徴を持っています。特にPostgreSQL^{注2}に関してはドライバレベルでの互換性^{注3}を持っているため、ユニットテスト専用のデータベースとして利用することもできます。

C.2 H2 Databaseのセットアップ

H2 Databaseは単一のJARファイルで提供されています。プロジェクトのWebサイト^{注4}よりJARファイルをダウンロードすればjavaコマンドで実行できます。

注1 外部プログラムとしてではなく、同一のプロセスで実行するデータベース。

注2 <http://www.postgresql.org/>

注3 プログラムからPostgreSQL用のJDBCドライバを使ってH2 Databaseを利用できます。

注4 <http://www.h2database.com/>

A
B
C
D
E
F

Eclipseのプロジェクトに追加する場合は、プロジェクトにコピーしたJARファイルをビルドパスに追加するか、Mavenの依存ライブラリとして設定します。

C.3 H2 Databaseの起動と停止

H2 Databaseは、サーバモードと組み込みモードの2つのモードで起動できます。どちらのモードでも、データベースのデータは単一のファイルに保存されます。

サーバモード

サーバモードは独立したプロセスとしてH2 Databaseを動作させるモードです。一般的なデータベースと同様に、コンソールなどからデータベースサーバを起動します。そして、アプリケーションからは、JDBCを経由してデータベースサーバへ接続します。

● コンソールからの起動と停止

コンソールからH2 Databaseを起動するには、次のようにjavaコマンドで起動します。

```
$ java -jar h2-*.jar
```

サーバを停止するには、**[Ctrl]+[D]**を入力するなどしてコマンドを停止させてください。

● Eclipseからの起動と停止

EclipseからサーバモードでH2 Databaseを起動するには、ビルドパスにJARファイルを追加し、Eclipseのメニューから[実行]-[実行]-[Java アプリケーション]を選択します。しばらくすると、「Java アプリケーションを選択」ダイアログが表示されるので、フィルタに「Server」などの文字列を入力して絞り込み、「org.h2.tools.Server」クラスを選択して[OK]をクリックしてください(図C.1)。

2回目以降の起動は、メニューの[実行]–[ヒストリーの実行]からでも実行できます。

サーバを停止する場合は、Eclipseからプログラムを停止してください。

組込みモード

H2 Database の組込みモードは、Java アプリケーションの内部で H2 Database を動作させるモードです。組込みモードで H2 Database を起動するには、次のようにプログラム内でユーティリティクラスを利用します。

```
String[] args = ...  
Server server = org.h2.tools.Server.createTcpServer(args);
```

サーバを停止するには、Server クラスの shutdown メソッドを利用します。なお、組込みモードかつサーバモードで実行することもできます(ミックスモード)。

図 C.1 EclipseからのH2 Databaseの起動



C.4 H2 Databaseの起動オプション

H2 Database をデフォルトのままで実行すると、データベースサーバが 9092 ポートで起動します。さらに、SQL を実行しデータベースを操作できる管理コンソールが 8082 ポートで、PostgreSQL 互換のデータベースサーバが 5435 ポートで起動します。

起動オプションを指定することで、これらの挙動はカスタマイズできます。コンソールからの起動で起動オプションを設定する場合は、起動コマンドにオプションを追加します。Eclipse からの起動で起動オプションを設定したい場合は、メニューから [実行] - [実行構成] を選択し、「実行構成」ダイアログから設定します。どちらの場合でも、設定できるオプションは同じです。

表 C.1 は、H2 Database の実行時に指定できる主なオプションです。デフォルトでは、起動オプションとして「-tcp -web -browser -pg」が指定されて、H2 Database は起動します。

C.5 H2 Console

H2 Database ではスキーマやテーブルの一覧の確認や、SQL の実行結果を確認するために利用できる管理ツールが組み込まれています。この管理ツールは、Web アプリケーションとして提供されており、サーバの起動時に -web オプションを指定するか、Server クラスの createWebServer メソッドを

表 C.1 H2 Database の起動オプション(抜粋)

オプション	内容
-baseDir <PATH_TO_DIR>	データベースファイルが保存されるディレクトリ
-tcp	TCP サーバの起動(コマンドラインのみ)
-tcpPort <port_no>	TCP サーバのポート(デフォルト:9092)
-web	Web サーバ(管理コンソール)の起動(コマンドラインのみ)
-webPort <port_no>	Web サーバのポート(デフォルト:8082)
-browser	管理コンソールをブラウザで表示(コマンドラインのみ)
-pg	PostgreSQL サーバの起動(コマンドラインのみ)
-pgPort <port_no>	PostgreSQL サーバのポート(デフォルト:5435)

実行することで利用できます(図C.2)。

「JDBC URL」にはH2 Databaseのデータファイルへのパスを指定します。「jdbc:h2:~/test」といったフォーマットで指定してください。また、インメモリモードで利用する場合は、JDBC URLに「jdbc:h2:mem:test」といったフォーマットで指定します。

図C.2 H2 Console



A
B
C
D
E
F

付録D

Android開発環境のセットアップ

ここでは、第13章で紹介したAndroidアプリケーション開発のための、開発環境の構築方法を紹介します。

なお、本書の執筆時点では、Androidの最新バージョンは4.1(API Level 16)です。Androidは非常にバージョンアップの早いプロダクトです。バージョン間でさまざまな差異があるため、本書の内容を確認する場合は、なるべく同じバージョンを利用して下さい。

D.1 ADTのインストール

Androidアプリケーションを開発するには、EclipseにAndroid開発用のプラグイン(ADT：*Android Development Tools*)をインストールします。

まずEclipseを起動し、メニューから[ヘルプ]→[新規ソフトウェアのインストール]を選択すると、「インストール」ダイアログが表示されます(図D.1)。

「作業対象」欄の右端にある[追加]ボタンをクリックし、「サイトの追加」ダイアログが表示されたら「ロケーション」にプラグインの配布URL「<http://dl-ssl.google.com/android/eclipse/>」を入力します(図D.2)^{注1}。

なお、「名前」は適当に入力してかまいません。

[OK]をクリックし、プラグインの情報が表示されたならば、「開発ツール」を選択してインストールします(図D.3)。

なお、ADTをインストールすることで、Android SDKも併せてインストールされます。

注1 公式サイトのドキュメント(<http://developer.android.com/sdk/eclipse-adt.html>)ではhttpsとなっていますが、うまくダウンロードできないことが多いのでここではhttpを利用しています。

図D.1 新規ソフトウェアのインストール



図D.2 ロケーションの追加



図D.3 ADTのインストール

A
B
C
D
E
F

D.2 AVDの設定

ADTには、エミュレータを含んだAndroid SDKが含まれているため、Android端末を持っていなくとも開発を行うことができます。ただし、Eclipseからエミュレータを使用するためには、AVD(*Android Virtual Device*)の設定が必要です。

設定するには、Eclipseのメニューから[ウィンドウ]-[AVDマネージャ]を選択して、「Android仮想デバイス・マネージャー」を起動します(図D.4)。

Android仮想デバイス・マネージャーが起動したならば、[新規]ボタンをクリックしてAVDを追加します。AVDを追加するときには、エミュレータで扱うAPIのバージョンをターゲットで選択します。図D.5では、Android 4.1を選択し、名前に「Android-4.1」と入力しました。もし、開発対象のAPIが選択肢に表示されない場合は、追加でダウンロードしてください。

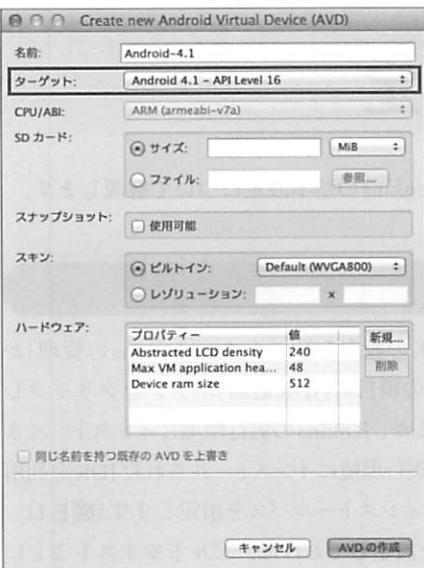
最後に[AVDの作成]をクリックすれば設定は完了です。なお、ここでAVDがうまく作成できない場合は、[ウィンドウ]-[Android SDKマネージャー]を選択し、「パッケージ」一覧から「Android 4.1(API 16)」下にある「SDK Platform」などがインストールされていることを確認してください。

AVDを作成したならば、[開始]ボタンでエミュレータを起動できます(図D.6)

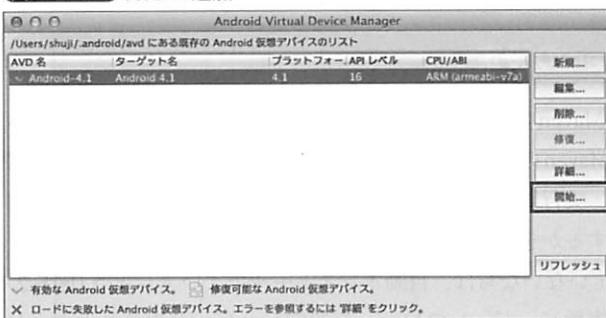
図D.4 Android仮想デバイス・マネージャー



図D.5 AVDの追加



図D.6 AVDの追加

A
B
C
D
E
F

付録E

Jenkinsの設定

ここでは、第15章で紹介したJenkinsの基本設定について解説します。

E.1 JDKの設定

Jenkinsのジョブで利用するJDKの設定を行います。[Jenkinsの管理]から[システムの設定]を開き、JDKの項目の[JDK追加]ボタンをクリックします。JDKは自動インストールするか、Jenkinsの実行環境にインストールされたJDKのパスを指定します^{注1}。実行環境にインストールされたJDKを利用するには、JAVA_HOMEにJDKのインストールパスを指定します(図E.1)。

なお、Jenkinsでは複数のJDKを設定しておけば、ビルドやテストごとに切り替えることもできます。

E.2 Mavenの設定

次に使用するMavenの設定を行います。[Jenkinsの管理]から[システムの設定]を開き、Mavenの項目の[Maven追加]ボタンをクリックします。JDKの設定と同様に、実行環境にインストールされたMavenを利用するか、自動インストールするかを選ぶことができます。ローカル環境にMavenがインストールされていないならば、自動インストールでよいでしょう(図E.2)。

JDKと同様に複数バージョンのMavenを設定し、ビルドやテストごとに切り替えることができます。

E.3 プラグインの設定

Subversionのようなバージョン管理システムと連携するには、使用するバージョン管理システムに対応するプラグインをインストールする必要が

注1 自動インストールにはOracleのアカウント入力が必要です。

あります。ただし、Subversionについてはデフォルトで利用できます。

[Jenkinsの管理]から[プラグインの管理]を開き、[利用可能]のタブを開いてください(図E.3)。Gitを利用する場合は「Git Plugin」を、Mercurialを利用する場合は「Mercurial Plugin」をプラウザで検索してインストールします。

Git プラグインと Mercurial プラグインは、パスに git や hg のコマンドが通っていれば特別な設定は必要ありません。パスを通してない場合や、パスの通っている git や hg とは異なるバージョンの Git や Mercurial を利用したい場合は、[Jenkinsの管理]の[システムの設定]にあるプラグインの設定を変更してください。

図E.1 JDKの設定



図E.2 Mavenの設定



図E.3 利用可能なプラグイン

名前	バージョン
Analysis Collector Plugin	1.26
Static Code Analysis Plug-ins	1.40
CCM Plugin	3.0
Checkstyle Plugin	3.26

A
B
C
D
E

付録F

本書利用環境のバージョン

ここでは、本書で利用している開発環境やライブラリのバージョン情報をまとめます。

F.1 開発環境

本書で利用した開発環境のバージョンと対応状況については、表F.1を参照してください。

なお、Jenkinsはウィークリーリリースが行われているため、バージョンは目安程度に考えてください。

表F.1 開発環境のバージョン

開発環境	バージョン
JDK	6
Eclipse IDE for Java Developers	3.7.2
Android SDK	4.1(API 16)
Maven	3.0.4
Jenkins	1.445

F.2 Eclipseプラグイン

Eclipseプラグインについては、Eclipseが選択した最新バージョンを利用してください。

表F.2 Eclipseプラグインのバージョン

プラグイン	ロケーション
QuickJUnit	http://quick-junit.sourceforge.jp/updates/current/
ADT	http://dl-ssl.google.com/android/eclipse/

F.3 外部ライブラリ

本書で利用した外部ライブラリについて、ライブラリのバージョンに加えて、MavenのPOMに設定するグループId、アーティファクトId、バージョンを記載します。

表F.3 外部ライブラリのバージョン

ライブラリ	グループId	アーティファクトId	バージョン
JUnit	junit	junit	4.10
Mockito	org.mockito	mockito-all	1.9.0
SnakeYAML	org.yaml	snakeyaml	1.10
h2 database	com.h2database	dbunit	1.3.167
DbUnit	org.dbunit	dbunit	2.4.8
cucumber-java	info.cukes	cucumber-java	1.0.8
cucumber-junit	info.cukes	cucumber-junit	1.0.8

F.4 Mavenプラグイン

本書で紹介したMavenプラグインはMaven 3.0.4で動作確認をしています。Maven 2では動作しない可能性がありますのでご注意ください。

表F.4 Mavenプラグインのバージョン

プラグイン名	グループId	アーティファクトId	バージョン
Maven Compiler Plug-in	org.apache.maven.plugins	maven-compiler-plugin	2.3.2
Maven Surefire Plug-in	org.apache.maven.plugins	maven-surefire-plugin	2.11
Surefire JUnit47	org.apache.maven.surefire	surefire-junit47	2.11
JaCoCo Maven Plug-in	org.jacoco	jacoco-maven-plugin	0.5.7.201204190339



API索引

android

Activityクラス	231
ActivityInstrumentationTestCase2クラス	245
sendKeysメソッド	251
TextWatcherインターフェース	232
TouchUtilsクラス	
clickViewメソッド	251
View.OnClickListenerインターフェース	231

cucumber

Cucumberクラス	336
Cucumber.Optionsアノテーション	336
PendingExceptionクラス	338
前提アノテーション	338
ならばアノテーション	338
もしアノテーション	338

dbunit

AbstractDatabaseTesterクラス	206
onSetupメソッド	207
onTearDownメソッド	207
setDataSetメソッド	207
Assertionクラス	213
assertEqualsメソッド	213
DBTestCaseクラス	210
FlatXmlDataSetBuilderクラス	212
IDatabaseConnectionインターフェース	213
createDataSetメソッド	213
createTableメソッド	213
IDataSetインターフェース	211
ITableインターフェース	211

h2database

Serverクラス	203
-----------	-----

junit

Afterアノテーション	54
AfterClassアノテーション	56
Assertクラス	49, 60
assertEqualsメソッド	63
assertThatメソッド	13, 49, 61
assertTrueメソッド	63
failメソッド	62
AssertionErrorクラス	61, 62
AssertionExceptionクラス	138
Assumeクラス	137
assumeThatメソッド	138
assertTrueメソッド	138, 389
AssumptionViolatedExceptionクラス	138
BaseMatcherクラス	73, 214
Beforeアノテーション	53, 357
BeforeClassアノテーション	54, 388
Categoriesクラス	89, 168
Categoryアノテーション	89, 166
ClassRuleアノテーション	142, 160
CoreMatchersクラス	67
instanceOfメソッド	70
isメソッド	14, 50, 61, 67, 375
notメソッド	68
notNullValueメソッド	69
nullValueメソッド	68
sameInstanceメソッド	69
DataPointアノテーション	129, 134
DataPointsアノテーション	
appendTextメソッド	88, 134, 385, 388
Descriptionクラス	76, 155
appendValueメソッド	77
Enclosedクラス	86, 96, 214, 357, 383
ErrorCollectorクラス	149
ExcludeCategoryアノテーション	89, 168

ExpectedException クラス	150, 397
ExternalResource クラス	145, 203
Ignore アノテーション	52
IncludeCategory アノテーション	168
JUnit4 クラス	82, 84
JUnitCore クラス	80
mainメソッド	81
JunitMatchers クラス	70
hasItemメソッド	70
hasItemsメソッド	66, 70
Matcher インタフェース	73
describeToメソッド	73, 76
matchesメソッド	73
MethodRule インタフェース	161
Parameterized クラス	129
Rule アノテーション	142
RuleChain クラス	158
aroundメソッド	159
outerRuleメソッド	159
Runner インタフェース	83
RunWith アノテーション	82
Statement クラス	154, 207
evaluateメソッド	154
Suite クラス	84
Suite.SuiteClasses アノテーション	85, 168
TemporaryFolder クラス	144
Test アノテーション	10, 40, 51
expected属性	21, 51, 57, 356
timeout属性	52
TestCase クラス	55
setUpメソッド	55
tearDownメソッド	55
TestName クラス	152
TestRule インタフェース	
.....	142, 154, 207, 408
applyメソッド	154
TestWatcher クラス	152
Theories クラス	88, 126, 385, 388
Theory アノテーション	88, 128, 386, 389
Timeout クラス	151, 367
TypeSafeMatcher クラス	214
Verifier クラス	146
mockito	
Answer クラス	197
answerメソッド	197
InvocationOnMock クラス	197
callRealMethodメソッド	198
Mockito クラス	191
anyBooleanメソッド	194
anyIntメソッド	194
anyStringメソッド	194
atLeastメソッド	195
atLeastOnceメソッド	195
atMostメソッド	195
doReturnメソッド	195, 197
doThrowメソッド	194
mockメソッド	191, 397, 400, 402, 405
neverメソッド	195
spyメソッド	196, 409
timesメソッド	195
verifyメソッド	194, 406, 408
whenメソッド	192, 397, 400, 402, 405
OngoingStubbing クラス	192
thenReturnメソッド	192, 400, 402, 405
thenThrowメソッド	193, 397
Stubber クラス	
whenメソッド	194
snakeyaml	
Yaml クラス	
loadメソッド	117, 135

索引

記号・数字

～するべき	327
4フェーズテスト	47, 56

アルファベット

actual	45
ADT (Android)	440
Android	219, 440
～のテスティングフレームワーク	244
Android SDK	226
ATDD	328
AVD (Android)	227, 442
BDD	327
C0(命令網羅)	257
C1(分岐網羅)	257
C2(条件網羅)	258
CI	299
CLI	219
configurationセクション	288
Cucumber	332
cucumber-junit	332
CUI	219
DbUnit	204
～のデータセット	211
dependenciesセクション	284
DI	180
DRY原則	276
EclEmma	260, 261
Eclipse	420
～のセットアップ	416
Eclipseプラグイン	446
expected	45
Gherkin	332
Git	298
Given/前提	331
Groovy	119
GUI	219, 224
GUIスレッド	223
H2 Database	203, 435
Hamcrest	6, 63, 71
JaCoCo	260, 294
JDK	412
Jenkins	299, 300, 444
～のジョブ	301
～のリモートビルド機能	304
JUnit	3
～のテストパターン	56
～をコマンドラインから実行する	80
Matcher API	49, 63
～の使用	67
Matcherオブジェクト	61, 63
Maven	277, 278
～によるカテゴリ化テスト	292
～によるカバレッジレポート	293
～によるビルドの実行	289
～のプラグイン	286
～のリポジトリ	283
Mavenプラグイン	447
Mavenプロジェクト	279
～の構成	282
Mercurial	298
Mockito	190
MVCパターン	220
mvnコマンド	289
O/Rマッピングフレームワーク	201
Pleiades	417
Pleiades All in One	418
POM	279, 447
pom.xml	279
POMエディタ	284
Quick JUnit	11, 422
SnakeYaml	116

SRP	341	依存ライブラリ	283
staticインポート	49	～の追加	284
～のワイルドカード	421	イベント駆動	222
Subversion	297	イベントハンドラ	223
SUT	43	イベントリスナー	223
TestNG	32	インスタンス化テスト	103, 431
Then/ならば	332	インラインセットアップ	110
throws句	40	受け入れテスト	325
UAT	326	～の自動化	326
VCS	296	受け入れテスト駆動開発	328
Web三層構造アーキテクチャ	200	エミュレータ(Android)	227
When/もし	331	エラーシナリオ	253
XP	308	お気に入り	425
xUnit	39		
xUnit Test Patterns	33		
YAML	116		
あ行			
アーティファクトId	282, 447	か行	
アウトサイドイン	348	外部リソースからのセットアップ	114
アクティビティ	230, 245	カウンタ	263
アクティビティテスト	245	拡張シナリオ	252
～の実行	250	過剰設計	178, 341
～の初期化	248	カスタムMatcher	37, 71
アサーション	13, 48, 60, 327, 429	カスタムルール	154
JavaBeansの～	373	型カウンタ	263, 265
データベースの～	213	カテゴリ	89, 165
複数行テキストの～	378	カテゴリ化テスト	89, 108, 165
リストの～	368	Eclipseから～を実行する	168
アサーションメソッド	60	Mavenによる～	292
後処理	47	～の実行	165
後処理メソッド	430	～のパターン	169
アノテーション	50	カバレッジ	256
暗黙的のセットアップ	111	～測定の目的	267
委譲オブジェクトの抽出	177	カバレッジツール	258, 261
異常系	255	カバレッジビュー	263
		カバレッジレポート(Maven)	293
		仮実装	316
		閾値的(メソッド)	46
		完璧なテスト	26, 43

技術的負債	319
期待値	14, 44, 126
機能テスト	26, 244, 255
境界値	28, 124, 381
境界値に対するテスト	28
行カウンタ	263, 264
共有フィックスチャ	108
クリックアウトライン	433
クリックフィックス	315, 432
組み合わせテスト	122, 136, 387
クラス	
ネストした～	86
クラス設計	314
テストが行いやすい～	45
グリーン	309
グリーンバー	15
グループId	281, 447
継続的インテグレーション	299, 306
継続的テスト	272, 274
計測点	320
検証	47
現代ソフトウェア開発の三本柱	274
更新系のテスト	202
後置記法(Mockito)	195
コードカバレッジ	256
ゴール	291
コミット	296
コメントは消臭剤	35
コンストラクタを検証するテスト	58
コンテキスト	36, 92
～のパターン	98
コンテンツアシスト	425
コントローラ	221
～のテスト	243
コンポーネント	231

さ行	
サービス層	200
サクセスストーリー	251
三角測量	319
参照系のテスト	202
事後条件の検証	146
システムテスト	325
事前準備	47
実行	47
実行環境	34
実測値	14, 44
自動化	275
自動化されたテスト	32, 33
シナリオ	327, 329
～の作成	335
～の優先順位	253
シナリオテスト	251
シナリオテンプレート	348
十分に良いテスト	26
循環的複雑度	265
準正常系	255
仕様	327
障害トレース	12, 16
条件網羅	258
状態(データベース)	202
状態に着目するテスト	188
初期化メソッド	429
ジョブ	302
シングルスレッド	223
シングルトン	37
シングルトンオブジェクト	38
スコープ	284
スタックトレース	16
スタブ	181, 401
モックと～の違い	186
例外を送出する～	185

スタブ実装.....	237	テスティングフレームワーク.....	39
ステージング環境.....	203	Androidの～.....	244
ステップ定義ファイル.....	337	テスティングペア.....	424
スパイ.....	187	テスト	
スパイオブジェクト.....	197	アクティビティの～.....	245
スペック.....	327	完璧な～.....	26, 43
スレッドモデル.....	223	組み合わせによる～.....	122
スローテスト問題.....	108, 162	更新系の～.....	202
正常系.....	255	コンストラクタを検証する～.....	58
生成メソッド.....	114	コントローラの～.....	243
～でのセットアップ.....	114	参照系の～.....	202
セットアップメソッド.....	112	自動化された～.....	32, 33
宣言的な記述.....	118	十分に良い～.....	26
前置記法(Mockito).....	195	状態に着目する～.....	188
セントラルリポジトリ.....	283	相互作用に着目する～.....	188
相互作用に着目するテスト.....	188	ドキュメントとしての～.....	35
ソースフォルダ(Eclipse).....	9	独立した～.....	37
ソフトウェアテスト.....	23	～のコンテキスト.....	92
た行		ビューの～.....	243
代替データベース.....	201	不安定な～.....	34
タイムアウト.....	52	不明瞭な～.....	36
～を設定する.....	151	モデルの～.....	238
単一責務の原則.....	341	脆い～.....	188
単体テスト.....	3	例外の送出を検証する～.....	20, 57, 150, 355
中央集権型のバージョン管理システム....	297	テスト環境.....	34
次の注釈(Eclipse).....	432	テスト技法.....	27, 121
データセット.....	211	テスト駆動開発.....	19, 308, 348
DbUnitの～.....	211	～のサイクル.....	309
～の外部定義.....	212	テストクラス.....	5, 9
データベース		～の構造化.....	93, 96
代替～.....	201	～を実行する.....	84
～のアサーション.....	213	テストケース.....	11, 24, 35, 42
～の状態.....	202	～の整理.....	92
デグレ(デグレーション).....	32	不完全な～.....	43
テスタビリティ.....	172, 314	テストコード.....	42

テストスイート	24	
テストスイートクラス	84	
テスト対象	43	
テスト対象クラス	5, 8	
テストダブル	181	
テストデータの選択	120	
テストパターン	56	
テストファースト	308, 312	
テストフィクスチャ	47, 106	
テストメソッド	11, 40, 428 ～のthrows句	40
テストランナー	82	
デベロッパーテスト	32	
テンプレート(Eclipse)	41, 426	
テンポラリフォルダ	144	
動作するきれいなコード	323	
同値クラス	28, 121, 359	
同値クラスに対するテスト	28	
ドキュメントとしてのテスト	35	
独立したテスト	37	
な行		
入力値	126	
ネストしたクラス	86, 431	
は行		
バーシステンス図	200	
バージョン	282, 447	
バージョン管理	275	
バージョン管理システム	296 中央集権型の～	297
分散型～	298	
バインドする	223	
発火する	223	
バックグラウンドスレッド	224	
ハッピーパス	251	
パラメータ	88 ～のフィルタリング	137
パラメータ化テスト	88, 120, 126, 385 ～の問題	138
ビジネスロジック	200	
ビュー	221 ～のテスト	243
ビルド	277 ～の実行	289
ビルドトリガー	304	
品質保証	26	
不安定なテスト	34	
フィーチャファイル	333	
フィクスチャ	106, 385 ～のセットアップパターン	109
フェーズ	289	
不完全なテストケース	43	
複雑度	265	
副作用(メソッド)	45, 357	
フック機能	305	
部分的なモックオブジェクト	196	
不明瞭なテスト	36	
プラグイン	291	
ブラックボックステスト	27	
プランチカウンタ	263, 264	
振舞駆動開発	327, 331, 348	
プレゼンテーション層	200	
フレッシュフィクスチャ	107	
プロジェクト(Eclipse)	5	
プロダクション環境	34, 200	
プロダクションコード	42	
分岐網羅	257	
分散型バージョン管理システム	298	
ホワイトボックステスト	27	
本番環境	34	

ま行

マーカーインターフェース	166
マーカークラス	166
マルチスレッド	365
命令カウンタ	263
命令網羅	257
メソッド	
関数的な～	46
テストが行いやすい～	45
～の抽出	175
副作用のある～	46
副作用のない～	46
メソッドカウンタ	263, 265
モック	46, 186
モックオブジェクト	190
～の作成	191
部分的な～	196
モデル	220
～のスタブ実装	237
～の定義	235
～のテスト	238
脆弱なテスト	188
問題の局所化	36

や行

ユーザ受け入れテスト	326
ユーザビリティ	220
ユーザビリティテスト	23, 43
ユースケースシナリオ	251, 329
ユーティリティメソッド	353
ユニットテスト	2, 23, 29, 275
～のパターン	33
～のフレームワーク	32

ら行

リグレッション	32
---------	----

リファクタリング	172, 174, 276, 309, 318
リファクタリング(略名)	35
リポジトリ	283
リモートビルド機能(Jenkins)	304
ルール	141
～の宣言	142
例外の送出を検証するテスト	
	20, 57, 150, 355
例外ハンドリング	395
例外を送出するスタブ	185
例によるテスト	251
レッド	309
レッド一グリーン一リファクタリング	309
レッドバー	12, 16
ローカルリポジトリ	283

わ行

ワーカスレッド	224
割り振り理論	307

著者紹介

渡辺 修司 Watanabe Shuji

ネコを愛でるふつうのプログラマ。札幌在住。地元ではコミュニティ活動を通じてJavaやテスト駆動開発の啓蒙活動を行う。業務では開発プロセスの改善や開発環境整備を行うことが多い。

“JUnitについてまとめた本を探していたと思ったら、
いつのまにか自分で書いていた”

最近の趣味はロードバイク。

●カバー・本文デザイン

西岡 裕二

●レイアウト

安達 恵美子

●本文図版

加藤 久(技術評論社)

●企画

稻尾 尚徳(技術評論社)

●編集アシスタント

佐藤 ひとみ(技術評論社)

●編集

緒方 研一(技術評論社)

ウェブディーピー ブ レ ス プラス WEB+DB PRESS plusシリーズ ウェブディーピー ブ レ ス プラス WEB+DB PRESS plusシリーズ ウェブディーピー ブ レ ス プラス

JUnit実践入門

—— 体系的に学ぶユニットテストの技法 ——

2012年12月25日 初 版 第1刷発行

2013年10月25日 初 版 第3刷発行

著 者 渡辺 修司

発行者 片岡 巍

発行所 株式会社技術評論社

東京都新宿区市谷左内町21-13

電話 03-3513-6150 販売促進部

03-3513-6175 雑誌編集部

印刷／製本 港北出版印刷株式会社

定価はカバーに表示しております。

本書の一部または全部を著作権法の定める範囲を超えて複写、複製、転載、あるいはファイルに落とすことを禁じます。

©2012 渡辺 修司

造本には細心の注意を払っておりますが、万一、乱丁(ページの乱れ)
や落丁(ページの抜け)がございましたら、小社販売促進部までお送
りください。送料小社負担にてお取り替えいたします。

ISBN 978-4-7741-5377-3 C3055

Printed in Japan

本書に関するご質問は紙面記載内容についてのみとさせていただきます。本書の内容以外の
ご質問には一切応じられませんので、あらかじめご了承ください。

なお、お電話でのご質問は受け付けておりま
せん。書面またはFAX、弊社Webサイトのお問
い合わせフォームをご利用ください。

〒162-0846

東京都新宿区市谷左内町21-13

株式会社技術評論社

JUnit実践入門係

FAX 03-3513-6173

URL <http://gihyo.jp/>

(技術評論社Webサイト)

ご質問の際に記載いただいた個人情報は回
答以外の目的に使用することはありません。使
用後は速やかに個人情報を廃棄します。