EasyMock を使用してテストを容易にする

オープンソースのモック・オブジェクト・フレームワークを利用してインターフェース、クラス、例外を模倣する

Elliotte Rusty Harold
Software Engineer
Cafe au Lait

2009年 4月 28日

この記事では著者の Elliotte Rusty Harold が、モック・オブジェクト、具体的には EasyMock フレームワークを使って、いくつかの困難なユニット・テストを容易に行う方法を説明します。このオープンソース・ライブラリーを利用すると時間を節約することができ、またモック・オブジェクトのコードが簡潔で読みやすいものになります。

テスト駆動開発はソフトウェア開発の重要な要素です。コードがテストされていなければ、そのコードには欠陥があるはずです。コードはすべてテストする必要があり、理想的にはモデル・コードを作成する前にテストを作成する必要があります。コードには、簡単にテストができるものとそうでないものがあります。例えばお金の計算をする単純なクラスを作成している場合には、$2.80 に $1.23 を加えると $4.03 になり、$3.03 や $4.029999998 にはならないことをテストするのは容易です。また、実際には $7.465 という金額はありえないことをテストするのも、それほど難しくはありません。しかし $7.50 を €5.88 に変換するメソッドをテストする場合で、1 秒ごとに情報が更新されるライブ・データベースに接続して為替レートの情報を得る必要がある場合には、どのようにテストすればよいのでしょう。`amount.toEuros()` による正しい結果はプログラムを実行するたびに変わってしまいます。

その答えが「モック・オブジェクト」です。最新の為替レート情報を提供する実際のサーバーに接続する代わりに、常に同じ為替レートを返すモック・サーバーにテストを接続するのです。すると、テストの結果を予測することができます。結局のところ、`toEuros()` メソッドのロジックをテストすることが目標なのであり、サーバーが適切な値を送信しているかどうかのテストは目標ではありません。(それはサーバーを構築した開発者達に任せればよいのです。) こうした種類のモック・オブジェクトはフェイクと呼ばれることがあります。

モック・オブジェクトはエラー条件をテストするためにも有効です。例えば、`toEuros()` メソッドが最新の為替レートを取得しようとする際にネットワークがダウンしている場合にはどうなるでしょう。コンピューターからイーサネット・ケーブルを引き抜いてからテストを実行することもできますが、ネットワークの障害をシミュレートするモック・オブジェクトを作成した方が、はるかに手間がかかりません。

モック・オブジェクトをクラスの動作の監視にも使うことができます。モック・コードの中にアサーションを配置すると、テスト対象のコードがそのコードと連携動作するコードに対して適切なタイミングで適切な引数を渡しているかどうかを検証することができます。モックを使用すると、クラスの中で private に設定されている部分を調べたりテストしたりすることができ、本来不必要な public メソッドをとおしてそれらの部分を公開する必要がありません。

モック・オブジェクトが有効な場合をもう 1 つだけ紹介しておくと、モック・オブジェクトを利用することで、大量にある依存関係をテストから取り除くことができます。モック・オブジェクトによって、依存関係に影響されない単体テストを行えるようになります。モック・オブジェクトを使ったテストで失敗する場合には、テスト対象のメソッドの依存関係で失敗している場合よりも、そのメソッドの中で失敗していることが多いものです。そのため、問題を切り分けることができ、またデバッグが単純になります。

EasyMock は Java プログラミング言語用のオープンソースのモック・オブジェクト・ライブラリーであり、上記のようなテスト用のモック・オブジェクトを迅速かつ容易に作成する上で役立ちます。EasyMock を利用すると、動的プロキシーの魔法のおかげで、任意のインターフェースの基本的な実装をたった 1 行のコードで作成することができます。また EasyMock のクラス拡張機能を追加すると、クラスのモックを作成することもできます。これらのモックは、メソッド・シグニチャーのための単純なダミー引数から、延々としたメソッド呼び出しのシーケンスの検証用に複数回呼び出されるスパイに至るまで、さまざまな用途向けに柔軟に構成することができます

EasyMock の紹介

まず、EasyMock がどのように動作するかを示す具体的な例から始めましょう。リスト 1 は架空の `ExchangeRate` インターフェースです。他のインターフェースと同じように、このインターフェースはインスタンスが何をするのかを表現していますが、どのように行うかを規定していません。例えばこのインターフェースは、為替レートを Yahoo!ファイナンスから取得するのか、政府から取得するのか、あるいは他のどこかから取得するのかを規定していません。

リスト 1. `ExchangeRate`

```
import java.io.IOException;

public interface ExchangeRate {

    double getRate(String inputCurrency, String outputCurrency) throws IOException;

}
```

リスト 2 は架空の `Currency` クラスのスケルトンです。実はこのクラスは非常に複雑であり、バグを含んでいる可能性が十分にあります。(気になると思うので言いますが、実は非常にたくさんのバグがあります。)

リスト 2. `Currency` クラス

```
import java.io.IOException;

public class Currency {

    private String units;
    private long amount;
    private int cents;
```

```
    public Currency(double amount, String code) {
        this.units = code;
        setAmount(amount);
    }

    private void setAmount(double amount) {
        this.amount = new Double(amount).longValue();
        this.cents = (int) ((amount * 100.0) % 100);
    }

    public Currency toEuros(ExchangeRate converter) {
        if ("EUR".equals(units)) return this;
        else {
            double input = amount + cents/100.0;
            double rate;
            try {
                rate = converter.getRate(units, "EUR");
                double output = input * rate;
                return new Currency(output, "EUR");
            } catch (IOException ex) {
                return null;
            }
        }
    }

    public boolean equals(Object o) {
        if (o instanceof Currency) {
            Currency other = (Currency) o;
            return this.units.equals(other.units)
                    && this.amount == other.amount
                    && this.cents == other.cents;
        }
        return false;
    }

    public String toString() {
        return amount + "." + Math.abs(cents) + " " + units;
    }

}
```

一見しただけでは、`Currency` クラスの設計で重要な点が明確にはわからないかもしれません。為替レートはこのクラスの外部から渡され、このクラスの内部で作成されるわけではありません。これは重要な点であり、このことから為替レートのモックを作成できることがわかります。為替レートのモックを作成することで、為替レート用の実際のサーバーと通信せずにテストを実行できるようになるだけでなく、クライアント・アプリケーションは為替レートのデータ用に別のソースを提供することができるようになります。

リスト 3 は為替レートが 1.5 の場合に $2.50 が €3.75 に交換されることを検証する JUnit テストを示しています。常に 1.5 という値を提供する `ExchangeRate` オブジェクトを作成するために EasyMock が使われています。

リスト 3. `CurrencyTest` クラス

```
import junit.framework.TestCase;
import org.easymock.EasyMock;
import java.io.IOException;

public class CurrencyTest extends TestCase {

    public void testToEuros() throws IOException {
        Currency testObject = new Currency(2.50, "USD");
        Currency expected = new Currency(3.75, "EUR");
        ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
        EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
        EasyMock.replay(mock);
        Currency actual = testObject.toEuros(mock);
        assertEquals(expected, actual);
    }

}
```

実を言うとリスト 3 は、私が初めて実行したときには (テストが毎度そうであるように) 失敗しました。ただし私はそのバグを修正してから、サンプル·コードとしてここに載せました。私達がテスト駆動開発を行う理由はここにあります。

このテストを実行すると、成功します。何が起きたのでしょう。このテストを 1 行ずつ見てみましょう。まず、テスト・オブジェクトと、想定される結果が作成されています。

```
Currency testObject = new Currency(2.50, "USD");
Currency expected = new Currency(3.75, "EUR");
```

ここまでは何も新しいものはありません。

次に、`ExchangeRate` インターフェースの `Class` オブジェクトを静的な `EasyMock.createMock()` メソッドに渡し、このインターフェースのモック・バージョンを作成します。

```
ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
```

ここが何と言っても奇妙な部分です。どこを見ても、`ExchangeRate` インターフェースを実装するクラスを作成していないことに注目してください。しかも、`EasyMock.createMock()` メソッドが `ExchangeRate` のインスタンスを返すように型付けしているところがどこにもありません。`ExchangeRate` は、私がこの記事のために作成したものにすぎず、`EasyMock.createMock()` メソッドはこの型を認識する由もなかったはずです。また、たとえ魔法のように返される `ExchangeRate`によってその型を認識できたとしても、別のインターフェースのインスタンスのモックを作成する場合にはどうすればよいのでしょう。

私は初めてこれを見たとき、ほとんど椅子から転げ落ちそうになりました。私にはこのコードがコンパイルできるとは思えなかったのですが、実際にコンパイルできたのです。ここには奥の深い暗黒の魔法があり、それは Java 5 の Generics と、はるか昔の Java 1.3 で導入された動的プロキシー (「参考文献」を参照) の組み合わせで実現されています。幸いなことに、この魔法の仕組みを知らなくても、この魔法を使うことはできるのです (そして、こうした手法を発明したプログラマー達の賢明さに驚嘆することになるのです)。

次のステップも同じくらい驚きに満ちています。何を想定しているかをモックに伝えるために、このメソッドを `EasyMock.expect()` メソッドの引数として呼び出しています。次に `andReturn()` を呼び出し、このメソッドを呼び出す結果として何を出力する必要があるかを指定します。

```
EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
```

EasyMock は後ほど何を再現すればよいかがわかるように、この呼び出しを記録します。

モックを使用する前に `EasyMock.replay()` を呼び出すことを忘れると、`IllegalStateException` が発生し、あまり役に立たないエラー・メッセージ `missing behavior definition for the preceding method call` が表示されます。

次に、`EasyMock.replay()` メソッドを呼び出し、記録したデータを再現するようにモックの準備を整えます。

```
EasyMock.replay(mock);
```

ここは少しばかり混乱する設計の一部です。`EasyMock.replay()` は実際にはモックを再現するわけではありません。実際にはモックをリセットし、次回 EasyMock.replay() のメソッドが呼び出されたときに再現を開始するのです。

これでモックの準備ができたので、このモックをテスト対象のメソッドに引数として渡します。

クラスのモックを作成する

実装の観点からは、クラスのモックを作成することは簡単ではありません。クラスに対して動的プロキシーを作成することはできません。標準的な EasyMock フレームワークはクラスのモックをサポートしていません。しかし EasyMock クラス拡張機能はバイトコード操作を使ってそれと同じことをクラスのモックを作成します。コードの中のパターンはほとんど完全に同じです。単純に `org.easymock.EasyMock` の代わりに `org.easymock.classextension.EasyMock` をインポートするだけです。クラスのモックを作成できると、他のものをそのままにした状態で、クラスの中の一部のメソッドをモックで置き換えることができます。

```
Currency actual = testObject.toEuros(mock);
```

最後に、答えが想定どおりであることを確認します。

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

これで終わりです。インターフェースがテスト用に何らかの結果を返す必要がある場合には、簡単なモックを作成するだけでよいのです。信じられないほど簡単です。`ExchangeRate` インターフェースは、規模が小さく単純なので、その気になれば手動でモック・クラスを作成することもできました。しかしインターフェースが大規模で複雑になればなるほど、それぞれの単体テスト用に、個別にモックを作成する作業は一層面倒になります。EasyMock を利用すると `java.sql.ResultSet` や `org.xml.sax.ContentHandler` などの大規模なインターフェースの実装を 1 行のコードで作成することができ、テストの実行に十分なだけの動作をその実装に追加することができます。

例外をテストする

モックの使い方のもっと一般的な例として、例外条件のテストがあります。例えば、要求に応じてネットワークの障害を起こすことは容易にはできませんが、その状況を模倣したモックを作成することができます。

`Currency` クラスは `getRate()` が `IOException` をスローすると `null` を返すことになっています。リスト 4 はこれをテストします。

リスト 4. あるメソッドが適切な例外をスローするかどうかをテストする

```
public void testExchangeRateServerUnavailable() throws IOException {
   Currency testObject = new Currency(2.50, "USD");
   ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
   EasyMock.expect(mock.getRate("USD", "EUR")).andThrow(new IOException());
   EasyMock.replay(mock);
   Currency actual = testObject.toEuros(mock);
   assertNull(actual);
}
```

ここで新しい部分は `andThrow()` メソッドです。ご想像のとおり、このメソッドは呼び出されると、指定された例外をスローするように `getRate()` メソッドを単純に設定します。

スローされる例外は、チェック例外であれ、ランタイム例外であれ、エラー例外であれ、任意の種類を設定することができます (ただしメソッド・シグニチャーがその例外をサポートしている必要があります)。これは、非常に起こりにくい条件 (out-of-memory エラーや class def not found など) をテストする場合や、仮想マシンのバグを示す条件 (例えば UTF-8 文字エンコーディングが利用できない、など) をテストする場合などに特に便利です。

想定を設定する

EasyMock は定型的な入力に対する応答として定型的な答えを提供するだけではなく、入力が想定どおりかどうかをチェックすることもできます。例えば、リスト 5 に示すバグが `toEuros()` メソッドにあったとします (結果をユーロで返していますが、カナダ・ドル用の為替レートを取得しています)。これでは誰かが大金を得たり大金を失ったりする可能性があります

リスト 5. バグのある `toEuros()` メソッド

```
public Currency toEuros(ExchangeRate converter) {
    if ("EUR".equals(units)) return this;
    else {
        double input = amount + cents/100.0;
        double rate;
        try {
            rate = converter.getRate(units, "CAD");
            double output = input * rate;
            return new Currency(output, "EUR");
        } catch (IOException e) {
            return null;
        }
    }
}
```

しかし、このために追加のテストは必要ありません。リスト 4 の `testToEuros` はこのバグを検出する機能を既に持っています。バグのあるコードを持つこのテストをリスト 4 で実行すると、テストは失敗し、下記のエラー・メッセージが表示されます。

```
"java.lang.AssertionError:
  Unexpected method call getRate("USD", "CAD"):
    getRate("USD", "EUR"): expected: 1, actual: 0".
```

このアサーションは私が作ったものではないことに注意してください。EasyMock は私が渡した引数が想定どおりでないことを認識し、このテスト・ケースを失敗させたのです。

EasyMock はデフォルトで、テスト・ケースが指定のメソッドを指定の引数を使って呼び出すことのみを許可します。ただしこれでは厳密すぎる場合があるため、これを緩和する方法が用意されています。例えば `getRate()` メソッドに対して、単に `USD` と `EUR` のみではなく任意のストリングを渡せるようにしたいとします。その場合には、明示的なストリングではなく `EasyMock.anyObject()` を想定していることを指定するために、次のようにします。

```
EasyMock.expect(mock.getRate(
       (String) EasyMock.anyObject(),
       (String) EasyMock.anyObject())).andReturn(1.5);
```

私は少し細かいので、`EasyMock.notNull()` を指定し、`null` ではないストリングのみを許可するようにします。

```
EasyMock.expect(mock.getRate(
        (String) EasyMock.notNull(),
        (String) EasyMock.notNull())).andReturn(1.5);
```

静的な型チェックを行えば、このメソッドに `String` 以外のものが渡されるのを防ぐことができます。しかしここでは `USD` と `EUR` に加えて `String` を許可しています。正規表現を使うと、`EasyMock.matches()` を利用して、もっと明示的にすることができます。この場合は大文字で 3 文字の ASCII の`String` が必要です。

```
EasyMock.expect(mock.getRate(
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"),
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"))).andReturn(1.5);
```

`EasyMock.matches()` の代わりに `EasyMock.find()` を使うと、大文字で 3 文字のサブストリングを含む任意の `String` を受け付けることができます。

EasyMock には以下のような似たメソッドが基本データ型用に用意されています。

- `EasyMock.anyInt()`
- `EasyMock.anyShort()`
- `EasyMock.anyByte()`
- `EasyMock.anyLong()`
- `EasyMock.anyFloat()`
- `EasyMock.anyDouble()`
- `EasyMock.anyBoolean()`

数値型の場合、`EasyMock.lt(x)` を使って `x` よりも小さな任意の値を受け付けたり、あるいは `EasyMock.gt(x)` を使って `x` よりも大きな任意の値を受け付けたりすることもできます。

長く連続した想定対象をチェックする場合には、1 つのメソッド呼び出しの結果または引数を取得し、それを別のメソッド呼び出しに渡された値と比較します。そして最後に、カスタムの突き合わせを定義し、引数に関して考えられるあらゆる詳細をチェックすることができます。そのためのプロセスは少しばかり複雑になりますが、大部分のテストでは、基本的な突き合わせ (`EasyMock.anyInt()`、`EasyMock.matches()`、`EasyMock.eq()` など) を行えば充分です。

厳密なモックと順序チェック

EasyMock は、想定されたメソッドが適切な引数を使って呼び出されるかどうかを単にチェックするだけではありません。EasyMock は、これらのメソッドのみが呼び出されていること、そしてその呼び出し順序が適切かどうかを検証することもできます。この検証はデフォルトでは行われないので、この検証が行われるようにするには、テスト・メソッドの最後で `EasyMock.verify(mock)` を呼び出します。例えばリスト 6 は、`toEuros()` メソッドが `getRate()` を複数回呼び出すと失敗します。

リスト 6. `getRate()` が 1 度だけ呼び出されていることをチェックする

```
public void testToEuros() throws IOException {
    Currency expected = new Currency(3.75, "EUR");
    ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
    EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
    EasyMock.replay(mock);
    Currency actual = testObject.toEuros(mock);
    assertEquals(expected, actual);
    EasyMock.verify(mock);
}
```

`EasyMock.verify()` がそうしたチェックをどの程度行うかは、EasyMock.verify() が以下のどのモードで動作するかに依存します。

- **通常モード — `EasyMock.createMock()`**: 想定されるすべてのメソッドが指定の引数を使って呼び出される必要があります。ただしそれらのメソッドの呼び出し順序はチェックされません。想定以外のメソッドへの呼び出しが行われると、テストは失敗します。
- **厳密モード — `EasyMock.createStrictMock()`**: 想定されるすべてのメソッドが指定の引数を使って指定の順序で呼び出される必要があります。想定以外のメソッドへの呼び出しが行われると、テストは失敗します。
- **緩和モード — `EasyMock.createNiceMock()`**: 想定されるすべてのメソッドが指定の引数を使って呼び出される必要がありますが、呼び出し順序はチェックされません。想定以外のメソッドへの呼び出しが行われてもテストは失敗しません。緩和モードのモックは、明示的にはモックを作成していないメソッドに対して妥当なデフォルトを提供します。例えば数値を返すメソッドは `0` を返します。ブール値を返すメソッドは `false` を返します。オブジェクトを返すメソッドは `null` を返します。

大規模なインターフェースや大規模なテストの場合には、メソッドが呼び出される順序と回数をチェックできると一層便利です。例えば `org.xml.sax.ContentHandler` インターフェースを考えてみてください。XML パーサーをテストする場合、文書を供給し、パーサーが `ContentHandler` の中

の適切なメソッドを適切な順序で呼び出しているかどうかを検証する必要があります。例えば単純な XML 文書を考えてみてください (リスト 7)。

リスト 7. 単純な XML 文書

```
<root>
  Hello World!
</root>
```

SAX 仕様によると、パーサーがこの文書を構文解析する際には、パーサーは以下のメソッドを以下の順序で呼び出す必要があります。

1. `setDocumentLocator()`
2. `startDocument()`
3. `startElement()`
4. `characters()`
5. `endElement()`
6. `endDocument()`

しかしさらに興味深いことに、`setDocumentLocator()` への呼び出しはオプションなのです。つまりパーサーは `characters()` を複数回呼び出すことができます。パーサーは、1 回の呼び出しの中で可能な限りテキストを連続させて渡す必要はなく、実際、大部分のパーサーはそうした動作をしません。これを従来の方法でテストしようとすると、たとえリスト 7 のように単純な文書の場合でさえ困難です。しかし EasyMock を使うと、このテストはリスト 8 のように単純になります。

リスト 8. XML パーサーをテストする

```
import java.io.*;
import org.easymock.EasyMock;
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
import junit.framework.TestCase;

public class XMLParserTest extends TestCase {

    private  XMLReader parser;

    protected void setUp() throws Exception {
        parser = XMLReaderFactory.createXMLReader();
    }

    public void testSimpleDoc() throws IOException, SAXException {
        String doc = "<root>\n  Hello World!\n</root>";
        ContentHandler mock = EasyMock.createStrictMock(ContentHandler.class);

        mock.setDocumentLocator((Locator) EasyMock.anyObject());
        EasyMock.expectLastCall().times(0, 1);
        mock.startDocument();
        mock.startElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"),
                (Attributes) EasyMock.anyObject());
        mock.characters((char[]) EasyMock.anyObject(),
                EasyMock.anyInt(), EasyMock.anyInt());
        EasyMock.expectLastCall().atLeastOnce();
        mock.endElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"));
        mock.endDocument();
        EasyMock.replay(mock);

        parser.setContentHandler(mock);
```

```
        InputStream in = new ByteArrayInputStream(doc.getBytes("UTF-8"));
        parser.parse(new InputSource(in));

        EasyMock.verify(mock);
    }
}
```

このテストではいくつかの新しい手法を使っています。まず、このテストは厳密モードのモックを使っています。このため、呼び出す順序が重要です。例えば、パーサーは `startDocument()` を呼び出す前に `endDocument()` を呼び出してはいけません。

第 2 に、ここでのテスト対象のメソッドはすべて `void` を返します。これはつまり、`getRate()` の場合とは異なり、`EasyMock.expect()` の引数としてテスト対象のメソッドを渡すことはできないということです。(EasyMock は多くのことでコンパイラーを騙しますが、`void` が引数の型として適切であるとコンパイラーに認識させるほどには賢くはありません。) 代わりに、単純にモックの void メソッドを呼び出すと、EasyMock がその結果を取り込みます。想定の詳細の一部を変更する必要がある場合には、モック・メソッドを呼び出した直後に `EasyMock.expectLastCall()` を呼び出します。また、想定している引数としておなじみの `String`、`int`、配列などを渡すことはできないことにも注目してください。これらをすべて最初に `EasyMock.eq()` でラップすることで、これらの値が想定の中で取り込まれるようにする必要があります。

リスト 8 は、メソッドが呼び出される想定回数を `EasyMock.expectLastCall()` を使って調整しています。デフォルトで、各メソッドは 1 回だけ呼び出されることが想定されています。しかし、ここでは `.times(0, 1)` を呼び出すことで、`setDocumentLocator()` の呼び出し回数が 0 と 1 の間であると指定しており、このメソッドの呼び出しをオプションにしています。もちろん、これらの引数を変更し、メソッドの想定呼び出し回数を 1 から 10 の間にしたり、3 から 30 の間にしたり、あるいはその他任意の範囲に設定したりすることができます。`characters()` に関しては、これが何回呼び出されるかまったくわかりませんが、少なくとも 1 回は呼び出される必要があるので、`.atLeastOnce()` としています。これが `void` を返すメソッドではない場合には、想定回数として `times(0, 1)` と `atLeastOnce()` を直接適用することもできます。ただしこれらのメソッドは `void` を返すようにモックが作成されているため、`EasyMock.expectLastCall()` を適用する必要があります。

最後に、`characters()` の引数に `EasyMock.anyObject()` と `EasyMock.anyInt()` を使っていることに注目してください。こうすることによって、パーサーに許容された、`ContentHandler` にテキストを渡すためのさまざまな方法に対応しています。

モックと現実

EasyMock に価値はあるのでしょうか。手動で作成するモック・クラスで不可能なことは EasyMock でも不可能であり、また手動でクラスを作成する場合にはプロジェクトの依存関係が 1 段階か 2 段階少なくなります。例えばリスト 3 は匿名内部クラスを使って手動で作成されたモックの例ですが、まだ EasyMock を知らない開発者にとっても非常に簡潔で読みやすいものです。ただしこの例は、この記事のために意図的に簡潔にしているにすぎません。もっと大規模なインターフェース、例えば (25 のメソッドを持つ) `org.w3c.dom.Node` や、(139 のメソッドを持ち、さらにメソッドが増えつつある) `java.sql.ResultSet` などの場合には、EasyMock を使うことで大幅に時間を節約することができ、より簡潔で読みやすいコードを最小限のコストで作成することができます。

ここで一言、注意しておきます。モック・オブジェクトを使いすぎないでください。あまりにも多くのものをモックとして作成したために、コードに深刻な欠陥がある場合でも常にテストをパスしてしまう、ということが起こり得ます。モックを増やせば増やすほど、テスト対象は少なくなります。依存ライブラリーの中や、あるメソッドとそのメソッドが呼び出すメソッドとのやり取りの中には、バグが数多く存在しています。依存関係をモックにすると、本来見つかるはずの大量のバグを隠してしまうかもしれません。どのような状況においても、モックが第 1 の選択肢であってはなりません。実際の依存関係を使用できる場合には、そうしてください。モックは実際のクラスの置き換えになるほどには優れてはいません。しかし何らかの理由から、実際のクラスを使って信頼性の高い自動的なテストが行えない場合には、モックを使ってテストをした方が、何もテストを行わない場合とは比べものにならないほど良い結果を得ることができます。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold はニューオーリンズ出身であり、時たま、おいしいガンボ (gumbo: オクラ入りのスープ) を食べに帰っています。ただし現在はアーヴィン近郊の University Town Center に、妻の Beth と猫の Charm (charmed quarkからとりました) と Marjorie（ 義理の母の名前からとりました ）と一緒に住んでいます。彼の Webサイト Cafe au Lait は、インターネット上で最も人気のある独立系 Java サイトの 1 つです。また、そこから派生した Cafe con Leche は、最も人気のある XML サイトの一つです。彼の最近の著作には『Refactoring HTML』があります。