疑似オブジェクトによる単体テスト

コラボレーターを疑似オブジェクトで置き換えることによって単体テストを改善する

Alexander Chaffee (alex@jguru.com)
Consultant
Purple Technology

2002年 11月 01日

William Pietri (william@scissor.com)
Consultant
Scissor

疑似オブジェクトは、オブジェクトの単体テストを作成するのに便利な方法で、仲介者 (mediator) として機能します。テスト対象オブジェクトは、実ドメイン・オブジェクトを呼び出す代わりに、疑似ドメイン・オブジェクトを呼び出します。この疑似ドメイン・オブジェクトが行う処理は、テスト対象オブジェクトにより、設計からみて正しいメソッドが、予期されたパラメーターと共に、正しい順番で呼び出されたか示すことだけです。ただし、テスト対象オブジェクトが疑似ドメイン・オブジェクトを生成しなければならない場合に問題があります。このテスト対象オブジェクトは、実ドメイン・オブジェクトの代わりに疑似ドメイン・オブジェクトを生成しなければならないことを、どのように認識するのでしょうか。今回の記事では、ソフトウェア・コンサルタントであるAlexander Day Chaffee氏とWilliam Pietri氏が、ファクトリー・メソッドの設計パターンに基づいて疑似オブジェクトを生成するリファクタリング技術について説明します。

単体テストは、ソフトウェア開発の「ベスト・プラクティス」として広く受け入れられるようになりました。オブジェクトを作成する場合は、そのオブジェクトの動作を試すメソッドを含み、さまざまなパラメーターと共にさまざまなパブリック・メソッドを呼び出し、返された値が適切であるかを確認する、自動化されたテスト・クラスも提供する必要があります。

シンプルなデータやサービス・オブジェクトを処理する場合には、単体テストの作成は簡単です。しかし多くのオブジェクトは、他の複数のオブジェクトやインフラストラクチャーのレイヤーに依存しています。このようなオブジェクトのテストは、しばしば高価であったり、非現実的であったり、これらのコラボレーターをインスタンス化するには非効率であったりします。

たとえば、データベースを使用するオブジェクトの単体テストを実行する場合、データベースのローカル・コピーをインストールして構成し、導入し、テストを実行し、再びそのローカル・

データベースを削除するのは手間のかかる作業になる可能性があります。このジレンマを解決するには、疑似オブジェクトが役立ちます。疑似オブジェクトは、本物の実オブジェクトのインターフェースと同様に振る舞いますが、いかにも本物のオブジェクトが機能したかのような振る舞いをテスト対象オブジェクトに見せて、テスト対象オブジェクトの振る舞いを追跡するための最低限のコードのみを持ちます。たとえば、特定の単体テストにおけるデータベース接続では、常に同じ固定の結果を返す一方で、クエリーを記録することもあります。テストするクラスが予想どおりに振る舞う限り、テスト対象オブジェクトはその違いに気付かず、したがってこの単体テストでは、適切なクエリーが出力されたかどうかを確認できます。

中間に存在する疑似オブジェクト

疑似オブジェクトを使ったテストの一般的なコーディング・スタイルは次のとおりです。

- 疑似オブジェクトのインスタンスの作成
- 疑似オブジェクトの状態と期待値の設定
- 疑似オブジェクトをパラメーターとしてのドメイン・コードの呼び出し
- 疑似オブジェクト間の整合性の検証

このパターンは多くの場合非常に効果的ですが、疑似オブジェクトをテスト中のオブジェクトに渡すことができない場合があります。そのようなオブジェクトは、代わりに、コラボレーターを作成、検索、あるいは取得するよう設計されています。

たとえば、テスト対象オブジェクトが、Enterprise JavaBean (EJB) コンポーネントまたはリモート・オブジェクトへの参照の取得を必要とする場合があります。あるいは、ファイルを削除してしまう`File`オブジェクトのような、単体テストでは望ましくないと思われる副作用のあるオブジェクトを使用することもあります。

一般的に、こうした状況は、テストをより容易にするために、オブジェクトのリファクタリングを行うよい機会であると考えられています。たとえば、コラボレーター・オブジェクトが渡されるよう、メソッド・シグニチャーを変更することができます。

Nicholas Lesiecki氏の記事「AspectJおよび疑似オブジェクトによる柔軟なテスト」で、氏は、リファクタリングは必ずしも望ましいものではなく、また、それによって必ずしもコードが、読みやすく、理解しやすいものになるとは限らないことを指摘しています。多くの場合、コラボレーターをパラメーターとするようメソッド・シグニチャーを変更すると、メソッドの呼び出し元のコードは、テストされない、混乱を招く、分かりにくいものとなるでしょう。

問題の核心は、テスト対象オブジェクトがこれらのオブジェクトを「内部で」取得していることです。何らかの解決策をこの生成コードのすべてのオカレンスに適用しなければなりません。この問題を解決するために、Lesiecki氏は、検索アスペクトまたは生成アスペクトを使用しています。この解決策では、検索を実行するコードが、代わりに疑似オブジェクトを返すコードと自動的に置き換えられます。

AspectJは選択できない場合があるので、この記事では代わりのアプローチを紹介します。このアプローチは基本的にリファクタリングであるため、Martin Fowler氏の独創性に富んだ著書である「Refactoring: Improving the Design of Existing Code」 (参考文献を参照) で氏によって確立された

表示規則に従います (ここで紹介するコードは、Javaプログラミング用の最もよく知られた単体テスト・フレームワークであるJUnitに基づいていますが、JUnit専用というわけではありません)。

リファクタリング: ファクトリー・メソッドの抽出およびオーバーライド

リファクタリングとは、プログラムの機能を変えずにソース・コードを変える手順のことで、より理解しやすく、より効率的に、そしてより簡単にテストできるようにコードの設計を変更します。このセクションでは、ファクトリー・メソッドの抽出およびオーバーライドのリファクタリングについて段階的に説明します。

問題:テスト対象オブジェクトでコラボレーター・オブジェクトが生成されます。このコラボレーターは、疑似オブジェクトと置き換える必要があります。

リファクタリング前のコード

```
class Application {
...
  public void run() {
    View v = new View();
    v.display();
...
```

解決策:生成コードをファクトリー・メソッドに抽出し、テスト・サブクラス内でこのファクトリー・メソッドをオーバーライドし、オーバーライドされたメソッドが代わりに疑似オブジェクトを返すようにします。最後に、実用的であれば、元のオブジェクトのファクトリー・メソッドに対し、正しい型のオブジェクトを返すよう要求する単体テストを追加します。

リファクタリング後のコード

```
class Application {
...
  public void run() {
    View v = createView();
    v.display();
...
  protected View createView() {
    return new View();
  }
...
}
```

このリファクタリングによって、リスト1に示す単体テスト・コードが有効になります。

リスト1. 単体テスト・コード

```
class ApplicationTest extends TestCase {
  MockView mockView = new MockView();
  public void testApplication {
    Application a = new Application() {
      protected View createView() {
        return mockView;
      }
    };
    a.run();
    mockView.validate();
```

```
  }
  private class MockView extends View
  {
    boolean isDisplayed = false;
    public void display() {
      isDisplayed = true;
    }
    public void validate() {
      assertTrue(isDisplayed);
    }
  }
}
```

役割

この設計では、システムのオブジェクトによって実行される次の役割について紹介しています。

- **ターゲット・オブジェクト**: テスト対象オブジェクト
- **コラボレーター・オブジェクト**: ターゲットによって生成または取得されるオブジェクト
- **疑似オブジェクト**: 疑似オブジェクトのパターンに従うコラボレーターのサブクラス (または実装)
- **特化オブジェクト**: コラボレーターの代わりに疑似を返すよう生成メソッドをオーバーライドする、ターゲットのサブクラス

リファクタリング技巧

リファクタリングは、複数の小さな技術的ステップで構成されます。これらをまとめてリファクタリング技巧と呼びます。料理本のレシピに従うのと同様に、この技巧に従えば、大きなトラブルもなくリファクタリングを学ぶことができます。

1. コラボレーターを生成または取得するコードのすべての出現を特定します。
2. この生成コードに抽出メソッド・リファクタリングを適用し、ファクトリー・メソッドを作成します (Fowler氏の著書の110ページに記述。詳細については、参考文献のセクションを参照)。
3. ファクトリー・メソッドがターゲット・オブジェクトとそのサブクラスからアクセスできるようにします (Java言語では、`protected` キーワードを使用します)。
4. テスト・コードで、コラボレーターと同じインターフェースを実装する疑似オブジェクトを作成します。
5. テスト・コードで、ターゲットを拡張する (特化する) 特化オブジェクトを作成します。
6. 特化オブジェクトで、テストに対応した疑似オブジェクトを戻すために、作成メソッドをオーバーライドします。
7. オプション: 元のターゲット・オブジェクトのファクトリー・メソッドが、今までどおり、疑似ではない正しいオブジェクトを戻すように、単体テストを作成します。

例: ATM

銀行の自動預金支払機 (ATM) 用のテストを作成するとしましょう。その1つの例をリスト2に示します。

リスト2. 疑似オブジェクト導入前の初期の単体テスト

```
 public void testCheckingWithdrawal() {
    float startingBalance = balanceForTestCheckingAccount();
    AtmGui atm = new AtmGui();
    insertCardAndInputPin(atm);
    atm.pressButton("Withdraw");
    atm.pressButton("Checking");
    atm.pressButtons("1", "0", "0", "0", "0");
    assertContains("$100.00", atm.getDisplayContents());
    atm.pressButton("Continue");
    assertEquals(startingBalance - 100,
balanceForTestCheckingAccount());
  }
```

さらに、`AtmGui` クラス内部のマッチ・コードは、リスト3のようになります。

リスト3. リファクタリング前の生成コード

```
     private Status doWithdrawal(Account account, float amount) {
Transaction transaction = new Transaction();
    transaction.setSourceAccount(account);
    transaction.setDestAccount(myCashAccount());
    transaction.setAmount(amount);
    transaction.process();
    if (transaction.successful()) {
      dispense(amount);
    }
    return transaction.getStatus();
  }
```

このアプローチはうまくいきますが、残念なことに副作用があります。それは、テスト開始時より当座預金残高が少なくなり、他のテストがより難しくなるというものです。これを解決する方法がいくつかありますが、それらはすべてテストをより複雑なものにします。さらに悪いことに、このアプローチでは、預金管理システムまで3回往復する必要があります。

この問題を解決する最初のステップとして、リスト4のように`AtmGui` のリファクタリングを行い、本物のトランザクションの代わりに疑似トランザクションを使用します (何を変更しているかは**太字**のソース・コードを比較してください)。

リスト4. AtmGuiのリファクタリング

```
  private Status doWithdrawal(Account account, float amount) {
Transaction transaction = createTransaction();
    transaction.setSourceAccount(account);
    transaction.setDestAccount(myCashAccount());
    transaction.setAmount(amount);
    transaction.process();
    if (transaction.successful()) {
      dispense(amount);
    }
    return transaction.getStatus();
  }

  protected Transaction createTransaction() {
    return new Transaction();
  }
```

テスト・クラス内部に戻り、リスト5に示すように、`MockTransaction` クラスをメンバー・クラスとして定義します。

リスト5. MockTransactionクラスをメンバー・クラスとして定義

```
            private MockTransaction extends Transaction {
    private boolean processCalled = false;
    // override process method so that no real work is done
    public void process() {
      processCalled = true;
      setStatus(Status.SUCCESS);
    }
    public void validate() {
      assertTrue(processCalled);
    }
  }
```

最後に、リスト6に示すように、テスト対象オブジェクトが、本物のクラスではなく`MockTransaction`クラスを使用するようテストを作り直すことができます。

リスト6. MockTransactionkクラスの使用

```
  MockTransaction mockTransaction;
  public void testCheckingWithdrawal() {

    mockTransaction = new MockTransaction();
    AtmGui atm = new AtmGui() {
        protected Transaction createTransaction() {
          return mockTransaction;
        }
    };
    insertCardAndInputPin(atm);
    atm.pressButton("Withdraw");
    atm.pressButton("Checking");
    atm.pressButtons("1", "0", "0", "0", "0");
    assertContains("$100.00", atm.getDisplayContents());
    atm.pressButton("Continue");

    assertEquals(100.00, mockTransaction.getAmount());
    assertEquals(TEST_CHECKING_ACCOUNT,
mockTransaction.getSourceAccount());
    assertEquals(TEST_CASH_ACCOUNT,
mockTransaction.getDestAccount());
    mockTransaction.validate();
}
```

この解決策では、テストが多少長くなりますが、ATMのインターフェースを超えたシステム全体の振る舞いではなく、テスト対象クラスの直接の振る舞いにのみ関与しています。つまり、テスト口座の最終残高が正確であることはもはや確認しません。その機能は、`AtmGui`オブジェクトではなく、`Transaction`オブジェクト用の単体テストで確認します。

注:発明者によると、疑似オブジェクトは、その`validate()`メソッド内部で自身の検証のすべてを実行することになっています。この例では、わかりやすくするため、その検証の一部をテスト・メソッド内部に残しました。疑似オブジェクトの使用に慣れるに従って、疑似にどの程度の検証を分担させるかという感覚が養われるようになるでしょう。

内部クラスの魔術

リスト6では、`AtmGui`の匿名内部サブクラス (anonymous inner subclass) を使用して、`createTransaction`メソッドをオーバーライドしました。1つのシンプルなメソッドをオーバーライドするだけでよかったので、私たちの目的を達成するにはこれが簡潔な方法でした。複

数のメソッドをオーバーライドするか、`AtmGui` サブクラスを多くのテスト間で共有するのであれば、完全な (匿名でない) メンバー・クラスを生成する価値があるでしょう。

また、インスタンス変数を使用して、疑似オブジェクトへの参照を保存しました。これは、テスト・メソッドと特殊クラス間でデータを共有するための最も簡単な方法です。私たちのテスト・フレームワークは、マルチスレッド化されておらず、再入可能でもないので、この方法が有効です(マルチスレッド化され再入可能な場合は、`synchronized` ブロックによる保護が必要となります)。

最後に、疑似オブジェクト自体を、テスト・クラスのprivateな内部クラスとして定義しました。疑似オブジェクトを使用するテスト・コードの直後にそれを置くことでより明確になり、また、内部クラスは、その周囲のクラスのインスタンス変数にアクセスできるので、このアプローチは多くの場合便利です。

念には念を

このテストを作成するのに、ファクトリー・メソッドをオーバーライドしたため、元の生成コード (現在は、ベース・クラスのファクトリー・メソッド内部にある) のすべてのテスト対象が失われています。このコードを明示的に対象とするテストを追加することが有益かもしれません。このテストは、ベース・クラスのファクトリー・メソッドを呼び出し、返されるオブジェクトが正しい型であると明示するだけの簡単なものです。例を以下に示します。

```
    AtmGui atm = new AtmGui();
    Transaction t = atm.createTransaction();
    assertTrue(!(t instanceof MockTransaction));
```

`MockTransaction` も`Transaction` であるため、この逆の`assertTrue(t instanceof Transaction)` では十分でないことに注意してください。

ファクトリー・メソッドから抽象ファクトリーへ

この時点でさらに一歩進み、Erich Gamma氏などの 「Design Patterns」 (参考文献を参照) に詳しく説明されているように本格的な抽象ファクトリー・オブジェクトと置き換えたいと思うかもしれません。実際、多くのみなさんは、ファクトリー・メソッドではなくファクトリー・オブジェクトを使って、このアプローチを始めたことでしょう。私たちもそうしましたが、すぐに撤退しました。

3つめのオブジェクト・タイプ (ロール) をシステムに導入すると、いくつかの欠点が発生する可能性があります。

1. 複雑さが増すが、それに見合うだけの機能性の向上がない。
2. ターゲット・オブジェクトへのパブリック・インターフェースを変更しなければならない場合がある。抽象ファクトリー・オブジェクトを渡さなければならない場合、新しいパブリック・コンストラクターまたはミューテーターを追加しなければなりません。
3. 多くの言語に、「ファクトリー」の概念に関係した紛らわしい規則がある。たとえば、Java 言語では、ファクトリーは多くの場合、静的メソッドとして実装されますが、この状況ではそれは適切ではありません。

ここで思い出してください。今回の記事の全体のポイントは、オブジェクトをより簡単にテストできるようにすることです。多くの場合、テストしやすいように設計することで、オブジェクトのAPIがよりわかりやすく、よりモジュール化された状態になることもあります。ただし、これは大変な作業になるかもしれません。テスト主導の設計変更が、元のオブジェクトのパブリック・インターフェースに悪影響を与えてはなりません。

ATMの例では、生成コードに関する限り、`AtmGui` オブジェクトは、常に`Transaction` オブジェクトの1つの (実際の) 型だけを生成します。テスト・コードでは、それが異なる型 (疑似) を生成することが望まれます。しかし、テスト・コードが望むからといって、パブリックAPIにファクトリー・オブジェクトや抽象ファクトリーを処理させるのは誤った設計です。生成コードでそのコラボレーターの多くの型をインスタンス化する必要がない場合、そうした機能を追加すると、結果としてその設計は、不必要に理解が困難なものになるでしょう。

著者について

Alexander Chaffee

Alexander Day Chaffee氏は、Java言語、エクストリーム・プログラミング、およびオープン・ソースのコンサルティングとトレーニングを提供する企業である、Purple Technology の設立者です。氏は、jGuru に関するServlets FAQを管理しています。EarthWebのソフトウェア・エンジニアリングの管理者として、Javaコミュニティーの公式ディレクトリーであるGamelan を共同で作成しました。氏の連絡先は、alex@jguru.comです。

William Pietri

William Pietri氏は、システム・アナリスト兼企業家の息子であり、13歳のときにコンピューターで昼食代を稼ぐことを始めました。それから数年間、氏は、技術サポートからシステム管理、ソフトウェア・エンジニアリング、ユーザー・インターフェースの設計まで、技術界のほぼすべての面に携わってきました。氏は、技術コンサルティング会社であるScissor の設立者です。