

# コード品質を追求する: FITで解決する

## FITとJUnitを使って要求事項をテストする

Andrew Glover

2006年 2月 28日

JUnitでは、テストはすべて開発者の領域であると考えます。一方、FIT ( Framework for Integrated Tests ) を利用すると、要求を記述するビジネス顧客と、そうした要求を実装する開発者の間で協力してテストを行うことができます。では、FITとJUnitは競争相手なのでしょうか。絶対にそんなことはありません。この記事では、コード品質の完璧主義者であるAndrew Gloverが、FITとJUnitの良いところを組み合わせ、チームワークを改善し、効果的なエンド・ツー・エンド・テストを行うための方法を解説します。

ソフトウェア開発のライフサイクルでは、『誰もが品質に責任を持ちます』。理想的には、開発者が開発サイクルの初期段階でJUnitやTestNGなどのようなテスト・ツールを使って品質を保証し、QAチームがフォローアップとして、開発サイクルの最後の段階でSeleniumなどのツールを使って機能的なシステム・テストを行います。しかし、たとえ素晴らしい品質保証があったとしても、一部のアプリケーションは、顧客のもとに届くと低品質となってしまうものです。なぜなのでしょう。実は彼らは、『意図した通りのことをしていない』のです。

アプリケーションに関する要求を書く顧客あるいはビジネス部門と、そうした要求を実装する部門とのコミュニケーション・エラーは、頻繁に摩擦の原因となり、時には開発プロジェクトそのものの完全な失敗につながる場合すらあります。幸い、要求を書く人と実装者との間の『初期段階での』コミュニケーションを実現する方法があるのです。

### FITをダウンロードしてください

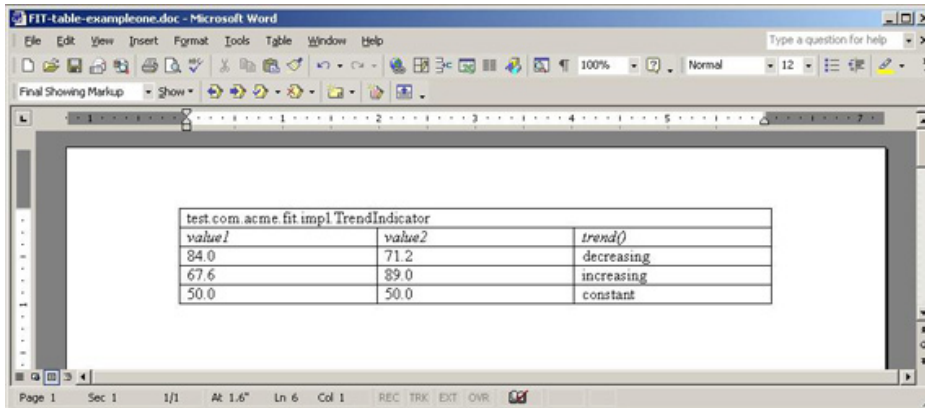
FIT、つまりFramework for Integrated Testsは元々、wikiの発明者としてよく知られているWard Cunninghamによって作成されました。CunninghamのWebサイトを訪れてFITについて学び、[無料のダウンロード](#)を入手してください。

## FITによるソリューション

『FIT ( Framework for Integrated Tests )』は、要求を書く人と要求を実行可能コードに変える人とのコミュニケーションを実現するための、テスト・プラットフォームです。FITでは、要求はテーブル形式でのモデルとして表現され、このモデルが、開発者の書くテスト用データ・モデルの役割を果たします。このテーブルそのものは、テストに対する入力と、想定される出力を表現するものとして動作します。

表1は、FITを使って作られた構造化モデルを示しています。最初の行はテスト名であり、次の行の3つのカラムは、入力に関するヘッダー（value1とvalue2）と、想定される結果（trend()）です。

図1. FITを使って作られた構造化モデル



The screenshot shows a Microsoft Word window with a table containing test data. The table has three columns: 'value1', 'value2', and 'trend()'. The first row is a header, and the following three rows contain numerical values and their corresponding trend results.

value1	value2	trend()
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

このテーブルの良いところは、プログラムの書き方をまったく知らない人でも書けることです。FITは、顧客やビジネス・チームと、彼らの考えを実装する開発者とが、開発サイクルの初期段階から協力できるように作られています。アプリケーションの要求を単純なテーブル形式のモデルで示すことによって、コードと要求が同じページにあるかどうかを、誰もが明確に見ることができるのです。

リスト1は、図1のデータ・モデルと関連するFITコードです。この詳細について気にする必要はありません。いかにコードが単純か、そして検証ロジック（アサーションなど）を含んでいないことだけに注目してください。皆さんの中には、表1と同じ変数名やメソッド名があることに気が付く人もいるかも知れません。これについては、後で詳しく説明します。

## リスト1. FITモデルを元に書かれたコード

```
package test.com.acme.fit.impl;  
  
import com.acme.sedlp.trend.Trender;  
import fit.ColumnFixture;  
  
public class TrendIndicator extends ColumnFixture {  
  
    public double value1;  
    public double value2;  
  
    public String trend(){  
        return Trender.determineTrend(value1, value2).getName();  
    }  
}
```

リスト1のコードは、開発者が表1を調べ、適当なコードをプラグインして書いたものです。ここまでをまとめると、FITフレームワークは、表1のデータを読み取り、対応するコードを呼び、そして結果を判断するのです。

## FITとJUnit

FITの素晴らしいところは、顧客、あるいは組織の中でビジネス側にある人達が、初期の段階で（つまり開発中に）テスト・プロセスに参加できることです。JUnitはコーディング中にユニット・テストできることが強みですが、FITは上位レベルのテスト・ツールとして、提案された要求に対する実装が有効かどうかを検証できるのです。

例えばJUnitが力を発揮するのは、2つのMoneyオブジェクトの和が、それら2つの値の和と同じことを検証するような場合です。一方FITが力を発揮するのは、注文の合計額が、ライン・アイテム価格の合計から何らかの値引きを差し引いた額であることを検証するような場合です。両者の違いはわずかですが、非常に重要です。JUnitでは特定なオブジェクト（あるいは要求の実装）を相手にしますが、FITでは、上位レベルの『ビジネス・プロセス』を相手にするのです。

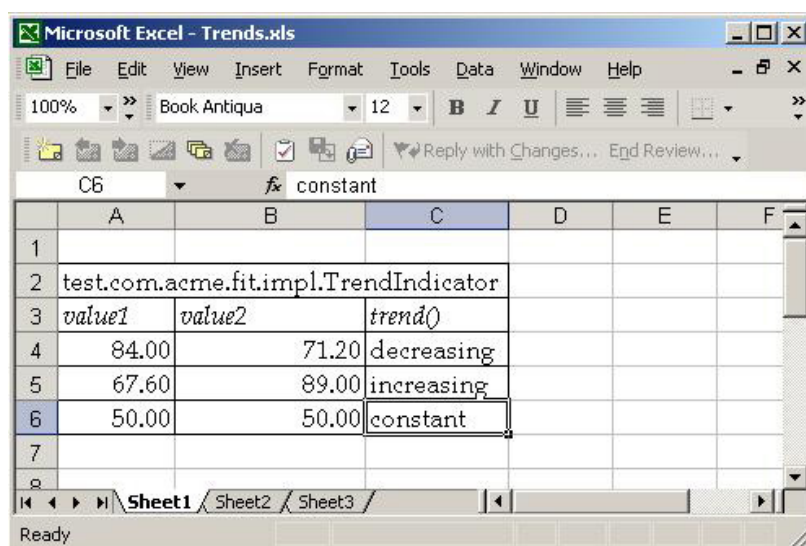
これは大きな違いです。というのは、要求を書く人達は、Moneyオブジェクトなどまったく気にとめないからです。実際彼らは、そんなものが存在することすら知りません。ところがそういう彼らも、ライン・アイテムが注文に追加された場合の合計注文額が、ライン・アイテムの和に値引きを適用したものであることには大きな関心を持つのです。

つまりFITとJUnitは競合するどころか、コード品質を保証するための素晴らしい組み合わせなのです。これについては、この先の[ケース・スタディー](#)を読むと分かるでしょう。

## FITのテーブルはテストに最適

テーブルはFITの心臓部です。テーブルには（様々なビジネス・シナリオに合わせて）幾つか異なるタイプがあり、FITを使う人は、様々なフォーマットを使ってテーブルを作成することができます。HTMLを使ってテーブルを書くこともできれば、図2のようにMicrosoft Excelを使って書くこともできます。

図2. Microsoft Excelを使って書かれたテーブル

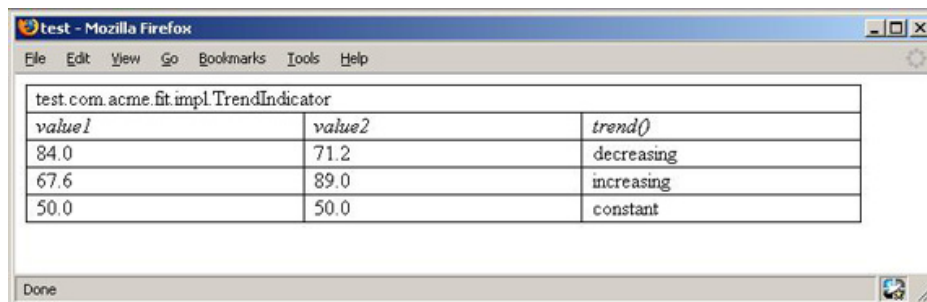


The screenshot shows a Microsoft Excel window titled 'Trends.xls'. The table contains the following data:

	A	B	C	D	E	F
1						
2	test.com.acme.fit.impl.TrendIndicator					
3	value1	value2	trend()			
4	84.00	71.20	decreasing			
5	67.60	89.00	increasing			
6	50.00	50.00	constant			
7						

また、Microsoft Wordなどのツールを使ってテーブルを作成してから、HTMLフォーマットで保存することもできます。これを図3に示します。

図3. Microsoft Wordを使って書かれたテーブル



test.com.acme.fit.impl.TrendIndicator		
value1	value2	trend()
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

テーブルのデータを実行するために開発者が書くコードは『フィクスチャー』と呼ばれます。フィクスチャー・タイプを作成するためには、対応するFITフィクスチャーを拡張し意図したテーブルへとマップする必要があります。先に触れた通り、マップ対象となるビジネス・シナリオによって、テーブルのタイプは異なります。

## フィクスチャーで修正する

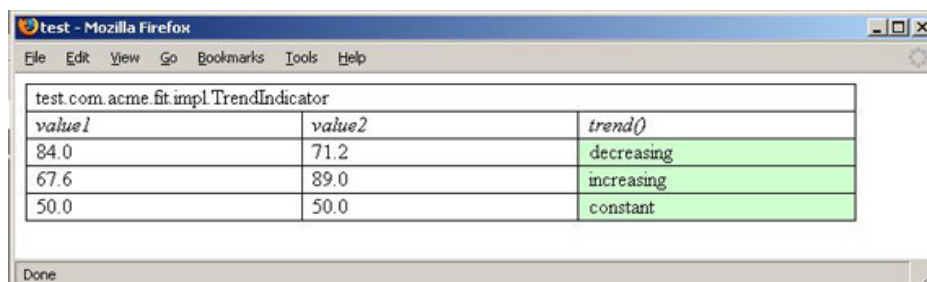
テーブルとフィクスチャーの最も単純な組み合わせ（FITでは最も一般的に利用されます）は、単純なカラム・テーブルです。このテーブルでは、カラムは、必要なプロセスの入力と出力にマップされます。これに対応するフィクスチャー・タイプはColumnFixtureです。

リスト1を再度見ると、TrendIndicatorクラスがColumnFixtureを拡張していること、また図3と対応していることに気がつくでしょう。図3で、最初の行の名前が、完全修飾されたクラス名（test.com.acme.fit.impl.TrendIndicator）に一致していることに注意してください。次の行には3つのカラムがあります。最初の2つのセルの値は、TrendIndicator クラスのpublicインスタンス・メンバー（value1とvalue2）に一致し、最後のセルの値は、TrendIndicatorの中に1つだけあるメソッド（trend）に一致します。

今度はリスト1のtrendメソッドを見ると、String値を返しています。ここまでに想像できるかも知れませんが、テーブルの中に残された各行に対して、FITは値を置き換え、結果を比較しています。この場合には3つの「データ」行があるので、FITはTrendIndicatorフィクスチャーを3回実行します。1回目では、value1は84.0に、value2は71.2に設定されます。次にFITはtrendメソッドを呼び、このメソッドから得られる値を、テーブルの中で見つかる値（「decreasing」）と比較します。

このように、FITはフィクスチャー・コードを使ってTrenderクラスをテストします（TrenderクラスのdetermineTrendメソッドは、FITがtrendメソッドを実行する毎に実行されます）。コードのテストが完了すると、FITは図4に示すようなレポートを生成します。

図4. FITがtrendテストの結果をレポートする



test.com.acme.fit.impl.TrendIndicator		
value1	value2	trend()
84.0	71.2	decreasing
67.6	89.0	increasing
50.0	50.0	constant

trendカラムのセルが緑色になっているのは、テストをパスしたことを示しています（つまりFITは、value1を84.0に、value2を71.2に設定し、trendが呼び出されると「decreasing」という値を受け取っています）。

## FITの実行を見る

FITはAntタスクを使ってコマンドラインから呼び出すことができ、またMavenを使って呼び出すことができるため、FITテストをビルド・プロセスの中に容易にプラグインすることができます。FITテストは（JUnitと同じように）自動化されているため、連続統合システムの中などで、一定間隔で実行することもできます。

最も単純なコマンドライン・ランナーは、リスト2に示すような、FITのFolderRunnerです。これには2つのパラメーター（FITテーブルの位置と、結果を書くべき場所）があります。クラスパスをコンフィギュレーションすることを忘れないでください。

## リスト2. コマンドライン用のFIT

```
%>java fit.runner.FolderRunner ./test/fit ./target/
```

またFITは、リスト3のようにプラグインを追加すれば、Mavenでも非常にうまく動作します。単純にこのプラグインをダウンロードし、fit:fitコマンドを実行するだけです。（Mavenプラグインに関しては[参考文献](#)を見てください。）

### リスト3. MavenでFITを使う

```
C:\dev\proj\edoa>maven fit:fit

|_ \/_|_|_ _Apache_ _
| | \ / | / _ \ \ v / - ) ' \ ~ intelligent projects ~
|_| | | \_ , - | \ / \_|_|_| v. 1.0.2

build:start:

java:prepare-filesystem:

java:compile:
    [echo] Compiling to C:\dev\proj\edoa\target\classes

java:jar-resources:

test:prepare-filesystem:

test:test-resources:

test:compile:

fit:fit:
    [java] 2 right, 0 wrong, 0 ignored, 0 exceptions
BUILD SUCCESSFUL
Total time: 4 seconds
Finished at: Thu Feb 02 17:19:30 EST 2006
```

## FITを試す: ケース・スタディー

これでFITの基礎は学んだので、今度は実際に試してみましょう。まだFITをダウンロードしていなければ、すぐにダウンロードしてください。先に触れた通り、このケース・スタディーを見れば



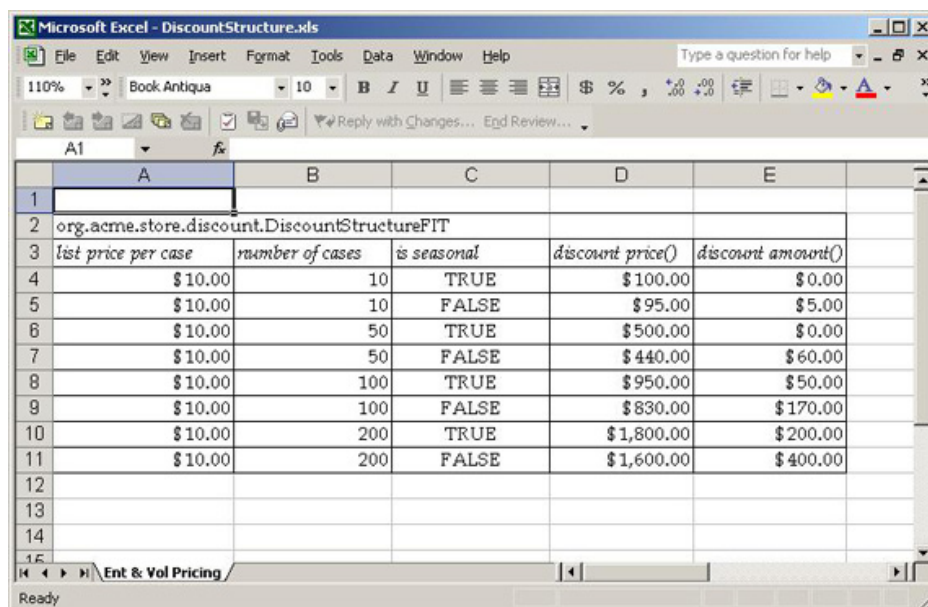
ば、多重階層の品質保証のためにFITとJUnitを組み合わせてテストすることが非常に容易なことを理解できるでしょう。

皆さんが、あるビール会社の注文処理システムを構築するように依頼されたと考えてみてください。このビール会社は様々なタイプの飲料を販売していますが、大きく分けると、季節商品（seasonal）と通年商品（year-round）、という2つのカテゴリーに分類することができます。このビール会社は卸売りとして操業しているため、すべての飲料はケース単位で販売されます。小売店への販売奨励策として、複数ケース買いに対する値引きサービスがあります。値引きの仕組みは、ケースの数と、季節商品か通年商品かによって変化します。

こうした要求の管理は、なかなか面倒です。例えば、ある小売店が季節商品を50ケース買う場合には値引きは適用されません。しかし、この50ケースが季節商品『ではない』場合には、12%の値引きが適用されます。ある小売店が季節商品を100ケース買う場合には、値引きは適用されますが、値引率は5%だけです。季節商品ではない飲料を100ケース買う場合には、17%の値引きです。200ケース買う場合にも、似たルールがあります。

開発者にとっては、こうした一連の要求は、非常にややこしく思えます。しかしFITテーブルを使うと、ビール・ビジネスのアナリストは、非常に容易に要求を記述できるのです。これを図5に示します。

図5. ビジネス要求が一目瞭然です



	A	B	C	D	E
1					
2	org.acme.store.discount.DiscountStructureFIT				
3	list price per case	number of cases	is seasonal	discount price()	discount amount()
4	\$10.00	10	TRUE	\$100.00	\$0.00
5	\$10.00	10	FALSE	\$95.00	\$5.00
6	\$10.00	50	TRUE	\$500.00	\$0.00
7	\$10.00	50	FALSE	\$440.00	\$60.00
8	\$10.00	100	TRUE	\$950.00	\$50.00
9	\$10.00	100	FALSE	\$830.00	\$170.00
10	\$10.00	200	TRUE	\$1,800.00	\$200.00
11	\$10.00	200	FALSE	\$1,600.00	\$400.00
12					
13					
14					

## テーブルの意味体系

このテーブルは、ビジネスの観点からは意味があり、要求をうまく表現しています。しかし開発者にとって価値あるものとするためには、もう少しテーブルの持つ意味合いを理解する必要があります。何よりもまず、このテーブルの最初の行はテーブルの名前を記述しています。この名前は、対応するクラス（org.acme.store.discount.DiscountStructureFIT）とうまく一致しています。適切な名前付けをするためには、テーブルを作る人と開発者との間に、少しばかり協力作業が必要です。少なくとも開発者は、完全修飾されたテーブル名を規定する必要があります（つまり、FITは対応するクラスを動的にロードするため、パッケージ名を含める必要があります）。

テーブルの名前が『FIT』で終わっていることに注意してください。恐らく皆さんは、『Test』にしたいと思うかも知れませんが、そうしてしまうと、自動化された環境でFITテストとJUnitテストを実行した場合にJUnitと衝突が起きる可能性があります。JUnitクラスは通常、名前付けのパターンによって発見されるため、FITテーブルの名前を『Test』で始めたり終わったりすることは避けた方が無難です。

次の行には5つのカラムが含まれています。各セルの中にある文字列は、意図的に斜字体で表現されていますが、これはFITの要求です。先に学んだ通り、セルの名前は、インスタンス・メンバーとフィクスチャーのメソッドに一致します。もっと正確に言うと、FITは、値が括弧で終わるセルはすべてメソッド、括弧で終わらない値はすべてインスタンス・メンバーと認識するのです。

## 特別なインテリジェンス

FITは、対応するフィクスチャー・クラスとセル値との一致検出に、インテリジェントな構文解析を使用します。図5から分かる通り、2番目の行のセル値は、「number of cases」（ケース数）など）単純な英語で書かれています。FITはこうした文字列を、ラクダケース（camel casing）を使って連結しようとしています。例えば「number of cases」は「numberOfCases」になります。そうするとFITはこれを、対応するフィクスチャー・クラスの中で見つけようとしています。この原則は、メソッドに対する動作でも同じです。図5を見ると分かる通り、「discount price()」は「discountPrice()」になっています。

またFITは、ある特定なセル値の『型』に関しても、インテリジェントな推測を行います。例えば図5の残り8行では、各カラムには対応する型があります。この型は、FITによって正確に推測されるか、あるいは幾らかカスタム・プログラミングが必要になります。この場合では、図5には3つの別々な型があります。「number of cases」（ケース数）に関連付けられたカラムはintと一致比較され、「is seasonal」（季節商品か否か）と関連付けられたカラム値はbooleanと一致比較されます。

残りの3つのカラム、「list price per case」（ケース当たりのリスト価格）、「discount price()」（値引き価格）「discount amount()」（値引き額）は明らかに通貨の額を表しています。これらにはカスタム型が必要であり、ここではその型をMoneyと呼んでいます。見て分かる通り、このアプリケーションはオブジェクトが金額を表すことを要求しています。ですからちょっとした意味体系に従うだけで、FITフィクスチャーの中で、このオブジェクトを利用できるのです。

## FITの意味体系のまとめ

表1は、名前の付いたセルと、それに対応するフィクスチャーのインスタンス・メンバーとの関係を要約したものです。

表1. セルとフィクスチャーとの関係: インスタンス・メンバー

テーブル・セルの値	対応するフィクスチャー・インスタンス・メンバー	型
list price per case	listPricePerCase	Money
number of cases	numberOfCases	int
is seasonal	isSeasonal	boolean

表2は、FIT名の付いたセルと、それに対応するフィクスチャーのメソッドとの関係を要約したものです。

表2. セルとフィクスチャーとの関係: メソッド

テーブル・セルの値	対応するフィクスチャーのインスタンス・メンバー	戻り型
discount price()	discountPrice	Money
discount amount()	discountAmount	Money

## いよいよビルドです！

このビール会社用の注文処理システムには、3つの主なオブジェクトがあります。つまり値引率を取得するためのビジネス・ルールを具体化するPricingEngine、注文を表現するWholeSaleOrder、そして金額を表現するMoney型の3つです。

### Money用に1つ・・・

コード化すべき最初のクラスはMoneyクラスです。このクラスには、値の加算、乗算、減算用のメソッドがあります。JUnitを使うと、新しく作成されたクラスをテストすることができます。これをリスト4に示します。

### リスト4. JUnitのMoneyTestクラス

```
package org.acme.store;

import junit.framework.TestCase;

public class MoneyTest extends TestCase {

    public void testToString() throws Exception{
        Money money = new Money(10.00);
        Money total = money.mpy(10);
        assertEquals("$100.00", total.toString());
    }

    public void testEquals() throws Exception{
        Money money = Money.parse("$10.00");
        Money control = new Money(10.00);
        assertEquals(control, money);
    }

    public void testMultiply() throws Exception{
        Money money = new Money(10.00);
        Money total = money.mpy(10);

        Money discountAmount = total.mpy(0.05);
```



```
    assertEquals("$5.00", discountAmount.toString());
}

public void testSubtract() throws Exception{
    Money money = new Money(10.00);
    Money total = money.mpy(10);

    Money discountAmount = total.mpy(0.05);
    Money discountedPrice = total.sub(discountAmount);
    assertEquals("$95.00", discountedPrice.toString());
}
}
```

## WholeSaleOrderクラス

次に、WholeSaleOrder型を定義します。この新しいオブジェクトは、このアプリケーションの中心です。WholeSaleOrder型は、ケース数（number of cases）、ケースあたりの価格（price per case）、製品タイプ（季節商品か通年商品か）などによってコンフィギュレーションされた後、PricingEngineに渡されます。PricingEngineは対応する値引きを判断し、それに従ってWholeSaleOrderインスタンスの中で値引きをコンフィギュレーションします。

リスト5はWholesaleOrderクラスを定義しています。

### リスト5. WholesaleOrderクラス

```
package org.acme.store.discount.engine;

import org.acme.store.Money;

public class WholesaleOrder {

    private int numberOfCases;
    private ProductType productType;
    private Money pricePerCase;
    private double discount;

    public double getDiscount() {
        return discount;
    }

    public void setDiscount(double discount) {
        this.discount = discount;
    }

    public Money getCalculatedPrice() {
        Money totalPrice = this.pricePerCase.mpy(this.numberOfCases);
        Money tmpPrice = totalPrice.mpy(this.discount);
        return totalPrice.sub(tmpPrice);
    }

    public Money getDiscountedDifference() {
        Money totalPrice = this.pricePerCase.mpy(this.numberOfCases);
        return totalPrice.sub(this.getCalculatedPrice());
    }

    public int getNumberOfCases() {
        return numberOfCases;
    }

    public void setNumberOfCases(int numberOfCases) {
        this.numberOfCases = numberOfCases;
    }
}
```

```
public void setProductType(ProductType productType) {
    this.productType = productType;
}

public String getProductType() {
    return productType.getName();
}

public void setPricePerCase(Money pricePerCase) {
    this.pricePerCase = pricePerCase;
}

public Money getPricePerCase() {
    return pricePerCase;
}
}
```

リスト5を見ると分かる通り、いったんWholeSaleOrderインスタンスの中で値引きが設定されると、getCalculatedPriceメソッドとgetDiscountedDifferenceメソッドを呼ぶことによって、それぞれ値引きされた価格と値引き幅を取得することができます。

これらのメソッドは（JUnitで）テストした方が良いでしょう。

MoneyクラスとWholesaleOrderクラスが定義できたら、JUnitテストを書いて、getCalculatedPriceメソッドとgetDiscountedDifferenceメソッドの機能を検証しましょう。このテストをリスト6に示します。

## リスト6. JUnitのWholesaleOrderTestクラス

```
package org.acme.store.discount.engine.junit;

import junit.framework.TestCase;
import org.acme.store.Money;
import org.acme.store.discount.engine.WholesaleOrder;

public class WholesaleOrderTest extends TestCase {

    /**
     * Test method for 'WholesaleOrder.getCalculatedPrice()'
     */
    public void testGetCalculatedPrice() {
        WholesaleOrder order = new WholesaleOrder();
        order.setDiscount(0.05);
        order.setNumberOfCases(10);
        order.setPricePerCase(new Money(10.00));

        assertEquals("$95.00", order.getCalculatedPrice().toString());
    }

    /**
     * Test method for 'WholesaleOrder.getDiscountedDifference()'
     */
    public void testGetDiscountedDifference() {
        WholesaleOrder order = new WholesaleOrder();
        order.setDiscount(0.05);
        order.setNumberOfCases(10);
        order.setPricePerCase(new Money(10.00));

        assertEquals("$5.00", order.getDiscountedDifference().toString());
    }
}
```

## PricingEngineクラス

PricingEngineクラスは『ビジネス・ルール・エンジン』を利用しています。この場合のビジネス・ルール・エンジンはDroolsです（[囲み記事](#)、[「Droolsについて」](#)を見てください）。PricingEngineは非常に単純で、1つのpublicメソッド（applyDiscount）しか持っていない。単純にWholeSaleOrderインスタンスを渡すと、エンジンはDroolsに対して、値引きを適用するように依頼します。これをリスト7に示します。

### リスト7. PricingEngineクラス

```
package org.acme.store.discount.engine;

import org.drools.RuleBase;
import org.drools.WorkingMemory;
import org.drools.io.RuleBaseLoader;

public class PricingEngine {

    private static final String RULES="BusinessRules.drl";
    private static RuleBase businessRules;

    private static void loadRules() throws Exception{
        if (businessRules==null){
            businessRules = RuleBaseLoader.
                loadFromUrl(PricingEngine.class.getResource(RULES));
        }
    }

    public static void applyDiscount(WholesaleOrder order) throws Exception{
        loadRules();
        WorkingMemory workingMemory = businessRules.newWorkingMemory( );
        workingMemory.assertObject(order);
        workingMemory.fireAllRules();
    }
}
```

### Droolsについて

Droolsは、Java™言語用に調整されたルール・エンジン実装です。Droolsにはプラグ可能な言語実装が用意されており、現在ではJavaやPython、Groovyを使ってルールを書くことができます。Droolsについて詳しくは、あるいはDroolsをダウンロードするためには、Droolsのホームページを見てください。

## Droolsによるルール

値引きを計算するためのビジネス・ルールは、Drools専用のXMLファイルの中で定義する必要があります。例えばリスト8のコード断片は、注文に対して5%の値引きを適用するルールです（ケースの数が9より大きく、50よりも小さく、季節商品ではない場合）。

## リスト8. BusinessRules.drlファイルからのサンプル・ルール

```
<rule-set name="BusinessRulesSample"
  xmlns="http://drools.org/rules"
  xmlns:java="http://drools.org/semantics/java"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/rules rules.xsd
    http://drools.org/semantics/java java.xsd">
<rule name="1st Tier Discount">
  <parameter identifier="order">
    <class>WholesaleOrder</class>
  </parameter>

  <java:condition>order.getNumberOfCases() > 9 </java:condition>
  <java:condition>order.getNumberOfCases() < 50 </java:condition>
  <java:condition>order.getProductType() == "year-round"</java:condition>

  <java:consequence>
    order.setDiscount(0.05);
  </java:consequence>
</rule>
</rule-set>
```

## タグ・チームのテスト

PricingEngineが用意でき、アプリケーション・ルールが定義できたら、皆さんは恐らく、すべてが正しく動作するかどうかを検証したくなるでしょう。そうすると問題は、「JUnitを使うのかFITを使うのか」ということになるかも知れません。いっそのこと、両方使ったらどうでしょう。すべての組み合わせをJUnitでテストすることは可能ですが、それには大量のコードが必要です。まずJUnitを使って幾つかの値を素早くテストし、順調に動作していることを検証してから、FITの力を利用して必要な組み合わせを実行した方が賢明です。リスト9以下では、その結果がどうなるかを示しています。

## リスト9. JUnitを使って、順調に動作しているかを素早く検証する

```
package org.acme.store.discount.engine.junit;

import junit.framework.TestCase;
import org.acme.store.Money;
import org.acme.store.discount.engine.PricingEngine;
import org.acme.store.discount.engine.ProductType;
import org.acme.store.discount.engine.WholesaleOrder;

public class DiscountEngineTest extends TestCase {

  public void testCalculateDiscount() throws Exception{
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(20);
    order.setPricePerCase(new Money(10.00));
    order.setProductType(ProductType.YEAR_ROUND);

    PricingEngine.applyDiscount(order);

    assertEquals(0.05, order.getDiscount(), 0.0);
  }

  public void testCalculateDiscountNone() throws Exception{
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(20);
    order.setPricePerCase(new Money(10.00));
```

```
order.setProductType(ProductType.SEASONAL);

PricingEngine.applyDiscount(order);

assertEquals(0.0, order.getDiscount(), 0.0);
}
}
```

## うまく行かなければ、FITを使ってみましょう

図5のFITテーブルの中には、データ値の行が8つあります。最初の2行については、リスト7のようにJUnitでコード化できたかも知れません。しかし皆さんは、残りの行に関しても本当にテストを書きたいと思うのでしょうか。8行すべてに対してコードを書くには、あるいは、顧客が新しいルールを追加した場合に新しい行を追加するためには、相当な忍耐が必要です。幸い、もっと容易な方法があるのです。いや、テストを無視しろと言っているのではありません。FITがあると言っているのです。

FITは、ビジネス・ルールのテストや、値の組み合わせに関わるテストには理想的なのです。さらに良いことに、誰か別の人が、こうした組み合わせをテーブルの形で定義できるのです。ただし、テーブルに対するFITフィクスチャーを作成する前に、Moneyクラスに特別なメソッドを追加する必要があります。FITテーブルの中では通貨の額（例えば\$100.00など）を表現するため、FITがMoneyのインスタンスを認識するための方法が必要です。これは、2ステップのプロセスで実現することができます。まず、static parseメソッドをカスタム・データ型に追加する必要があります。これをリスト10に示します。

## リスト10. static parseメソッドをMoneyクラスに追加する

```
public static Money parse(String value){
    return new Money(Double.parseDouble(StringUtils.remove(value, '$')));
}
```

MoneyクラスのparseメソッドはString値（つまり、FITがテーブルから取り出すもの）を取り、適切にコンフィギュレーションされたMoneyインスタンスを返します。この場合、\$という文字は削除され、残るStringはdoubleに変換されます。これは、既にMoneyの中で見つかっているコンストラクターに一致します。

幾らかのテスト・ケースをMoneyTestクラスに追加し、新たに追加されたparseメソッドが期待した通り動作することを検証するのを忘れないでください。この新しい2つのテストを、リスト11に示します。

## リスト11. Moneyクラスの中のparseメソッドをテストする

```
public void testParse() throws Exception{
    Money money = Money.parse("$10.00");
    assertEquals("$10.00", money.toString());
}

public void testEquals() throws Exception{
    Money money = Money.parse("$10.00");
    Money control = new Money(10.00);
    assertEquals(control, money);
}
```



## FITフィクスチャーを書く

これで、最初のFITフィクスチャーをコード化する準備ができました。インスタンス・メンバーとメソッドは、既に表1と表2の中に分類されています。ですから、あとはすべてをつなぎ合わせ、カスタム型（Money）を処理するメソッドを、あと1つだけ追加するだけです。フィクスチャーの中で特定な型を処理するためには、別のparseメソッドも追加する必要があります。ただしこのメソッドは、先ほどのメソッドと少しだけ異なるシグニチャーを持っています。このメソッドは、皆さんがFixtureクラスの上にオーバーライドしようとしているインスタンス・メソッドであり、ColumnFixtureの親です。

リスト12で、DiscountStructureFITのparseメソッドが、class型に対して『比較』を行っている様子に注意してください。もし一致があれば、Moneyからのカスタムparseメソッドが呼び出され、一致がなければ、スーパークラス（Fixture）版のparseが呼び出されます。

リスト12のコードの残り部分は、プラグ・アンド・プレイです！ 図5に示すFITテーブルの各データ行に対して、値が設定され、メソッドが呼び出され、そしてFITが結果を検証しています。例えば、初めてFITテストを実行すると、DiscountStructureFITのlistPricePerCaseが\$10.00に設定され、numberOfCasesが10に設定され、isSeasonalが真に設定されます。次にDiscountStructureFITのdiscountPriceが実行され、そこから返る値と\$100.00が比較され、続いてdiscountAmountが実行されます。その戻り値は\$0.00と比較されます。

## リスト12. FITを使った値引きのテスト

```
package org.acme.store.discount;

import org.acme.store.Money;
import org.acme.store.discount.engine.PricingEngine;
import org.acme.store.discount.engine.ProductType;
import org.acme.store.discount.engine.WholesaleOrder;
import fit.ColumnFixture;

public class DiscountStructureFIT extends ColumnFixture {

    public Money listPricePerCase;
    public int numberOfCases;
    public boolean isSeasonal;

    public Money discountPrice() throws Exception {
        WholesaleOrder order = this.doOrderCalculation();
        return order.getCalculatedPrice();
    }

    public Money discountAmount() throws Exception {
        WholesaleOrder order = this.doOrderCalculation();
        return order.getDiscountedDifference();
    }

    /**
     * required by FIT for specific types
     */
    public Object parse(String value, Class type) throws Exception {
        if (type == Money.class) {
            return Money.parse(value);
        } else {
            return super.parse(value, type);
        }
    }
}
```

```

}

private WholesaleOrder doOrderCalculation() throws Exception {
    WholesaleOrder order = new WholesaleOrder();
    order.setNumberOfCases(numberOfCases);
    order.setPricePerCase(listPricePerCase);

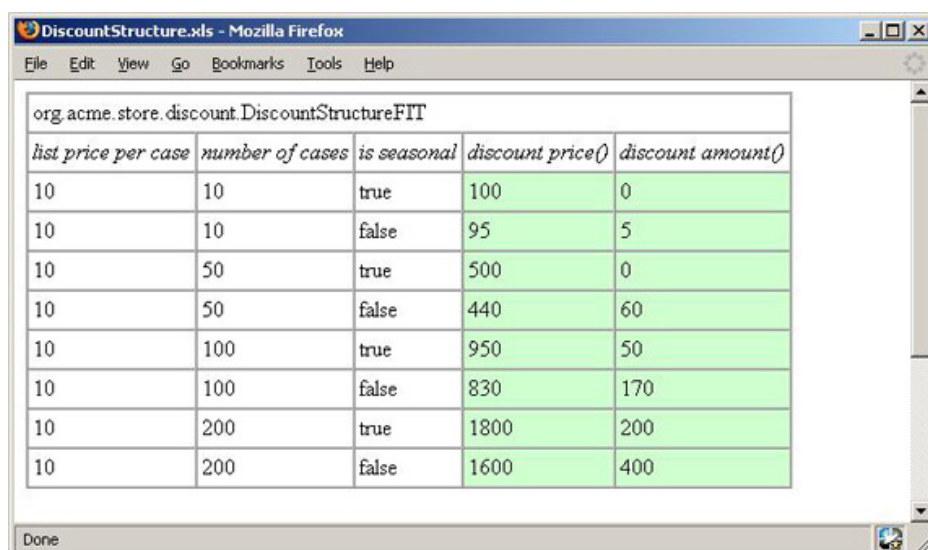
    if (isSeasonal) {
        order.setProductType(ProductType.SEASONAL);
    } else {
        order.setProductType(ProductType.YEAR_ROUND);
    }

    PricingEngine.applyDiscount(order);
    return order;
}
}

```

今度は、[リスト9](#)のJUnitテスト・ケースをリスト12のコードと比較します。リスト12の方が効率的に見えませんか。必要なテストすべてをJUnitでコード化することは確かにできますが、FITを使った方が、ずっと楽になるのです。これで満足できたら（満足できるはずですが）、ビルドを実行し、FITランナーを呼ぶと、図6に示すような素晴らしい結果が得られるわけです。

図6. FITで得られる結果は素晴らしい



org.acme.store.discount.DiscountStructureFIT				
<i>list price per case</i>	<i>number of cases</i>	<i>is seasonal</i>	<i>discount price()</i>	<i>discount amount()</i>
10	10	true	100	0
10	10	false	95	5
10	50	true	500	0
10	50	false	440	60
10	100	true	950	50
10	100	false	830	170
10	200	true	1800	200
10	200	false	1600	400

## まとめ

組織はFITを利用することによって、ビジネス顧客と開発者との間で起こりがちなコミュニケーション不足や誤った理解、誤った解釈などを避けることができます。要求を書く人達をテスト・プロセスの『初期』の段階に参加させることによって、開発の悪夢となる前に問題を捉え、解決することができるようになります。さらにFITは、JUnitのように既に確立された技術と完全な互換性があります。実際、ここで示したように、JUnitとFITは互いにうまく補完し合うのです。問題解決にFITを利用することで、今年を皆さんの『コード品質の追求』における輝かしい年にしてください。

## 関連トピック

- カバレッジ・レポートによって誤った結論を出さないように、「[コード品質を追求する: カバレッジ・レポートに騙されないために](#)」 ( Andrew Glover, developerWorks, 2006年1月 ) を読んでください。
- 「[JUnit 4の現状を紹介する](#)」 ( Elliotte Harold, developerWorks, 2005年9月 ) は、この貴重なテスト・フレームワークに近々予定されている変更について紹介しています。
- 「[Continuously ensuring quality: A case study](#)」 ( The Rational Edge, Laura Rose, 2004年12月 ) を読んで、IBM RationalがRational独自のソフトウェア製品を開発するために使用しているツールや技法について学んでください。
- 「[Ending requirements chaos](#)」 ( The Rational Edge, Susan August, 2002年8月 ) は、アプリケーションに関する要求を書くことや実装することの困難さを要約しています。
- [Java technologyゾーン](#)には、Javaプログラミングのあらゆる面を網羅した記事が他にも豊富に用意されています。
- [FIT \( Framework for Integrated Tests \) を入手](#)して、自分自身で試してみてください。
- [FIT Mavenプラグイン](#)をダウンロードして、MavenでのFITテストを自動化してください。
- JUnitのホームページからJUnitをダウンロードし、またJUnit 4に関する最新ニュースを入手してください。
- Droolsのホームページから、この記事で使用しているDroolsルール・エンジンをダウンロードしてください。

© Copyright IBM Corporation 2006

([www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml))

商標

([www.ibm.com/developerworks/jp/ibm/trademarks/](http://www.ibm.com/developerworks/jp/ibm/trademarks/))