使えるツールとライブラリを求めて

Java オープンソース探索

【第 3 回】データベーステスティングフレームワーク DbUnit
有限会社 Nulab(http://www.nulab.co.jp/)
Mobster Project(http://www.mobster.jp/)
Toshitaka Agata
技術評論社 WEB＋DB PRESS Vol.17 掲載記事
ソースコードのダウンロードは http://www.gihyo.co.jp/magazines/wdpress/archive/Vol17 より

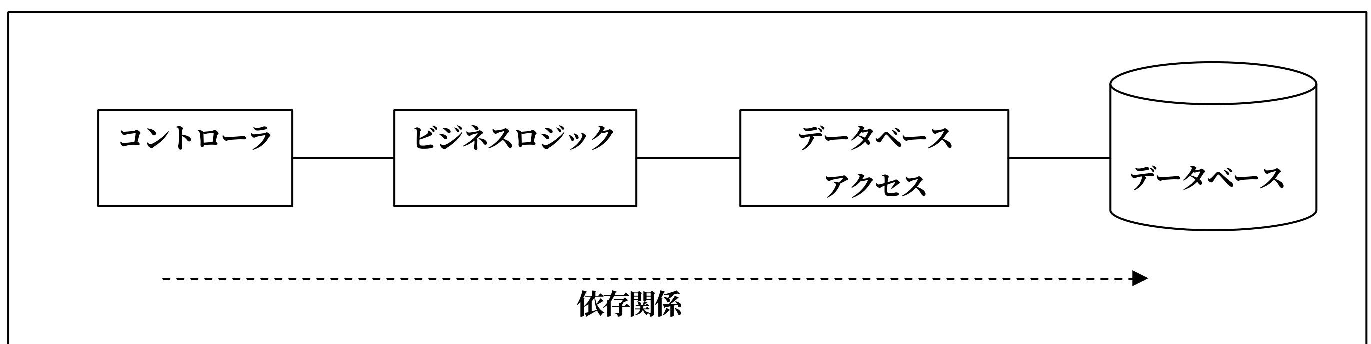
データベース絡みのテストはお好き？

先日、筆者の参加する Java メーリングリスト「Mobster Users」で「データベースが絡むコードのテストは必要か？」という議論が巻き起こりました。「データベース絡むコードのテストは必要なのか？」「どうやってやるの？」という問いかけに対して「データベースのテストは難しい」「データベースのテストはコストがかかる」「納品物ではないので作らない」という意見もでました。

データベースのテストが難しいのは、データベースはデータの状態が変化しやすく、「テストができる状態」を作り出すのが難しいからでしょう。複数人で同じデータベースを使用していると、データが知らないうちに書き換えられて、テストが通らなくなるという点も、テストを難しいものにしています。また、データベース内の大量のデータを相手にテストをおこない、処理が正しいかどうかデータベースに問い合わせてチェックする必要があるため、普通のテストよりコストがかかってしまいそうです。

しかし、筆者は「データベースが絡むコードもテストすべき」と考えています。通常、データベースにアクセスするコードは、ビジネスロジックから切り離して作成します。そして、データベースにアクセスするコードは、ビジネスロジックから呼び出されることがほとんどです。つまりビジネスロジックはデータベースに依存して場合が多いのです。「データベースに絡むコード」をテストしないということは、システムの根幹部分であるビジネスロジックの単体テストができないということになってしまいます。

図：データベースへの依存関係



コラム：モックオブジェクトってどう？

テストに関してモックオブジェクトを利用する方法もよく聞かれます。モックオブジェクトとは、データベースやサーブレットコンテナなど周辺環境を「シュミレート」することで、環境に依存せずにテストをおこなう手法です。モックオブジェクトは単体テストをおこなう上で大変有効な戦略ですが、正しいモックオブジェクトを書くのは難しく訓練を要します。また、業務アプリケーションのように「データベースドリブン」なシステムの場合、データベースにアクセスするコードが多すぎて、全てのモックオブジェクトは作るのは現実的ではありません。つまり、モックオブジェク

トを使わずにデータベースするコードをそのまま使い、ビジネスロジックをテストするのがより現実的な方法だと考えます。

データベーステスト戦略

「言うはやすし、西川きよし、いったいどうやってテストすればいいんだい？」という声が聞こえてきそうですが、ここでデータベース絡みのテストをスムーズにおこなうための戦略をご紹介したいと思います。

・戦略その１：テストの実行前にテストデータをデータベースに反映すべし

単体テストをおこなうときに、データベースの状態は「テストができる状態」でないといけません。単体テストとテストデータをペアにして、テストを作成しましょう。テストが実行される前に、テストデータをデータベースに反映させ、「テストができる状態」を作る必要があります。これはツールやスクリプトを使うことで実現できます。今回ご紹介する「Dbunit」は「テストができる状態」にすることを目的としたフレームワークです。

・戦略その２：開発者ごとに、専用のデータベースを持つべし

テスト中にデータベースのデータを変更されてしまっては、正しいテストは行えません。これを回避するためにも、開発者ごとにデータベースユーザを作成して、各自のデータベースを用意しましょう。これを「ユーザデータベース」と呼びます。マスタとなる、共有のデータベースは「マスタデータベース」です。「ユーザデータベース」はCVSにおけるローカルコピーと同じです。自由にテストしたり、新しいコードを試したりするワークスペースとして利用することができます。また他人のデータベースに影響を与えることも、影響を受けることもありません。「マスターデータベース」と「ユーザデータベース」という考え方はマーチンファウラーの「データベースの進化的設計」に詳しく書かれていますので参考にしてみてください。（http://objectclub.esm.co.jp/eXtremeProgramming/evodb-jp.html）

■今回のツール

さて、今回ご紹介するツールはDbunitです。名前からご想像できるように、DbunitはJUnitを拡張したデータベース絡みのコードをテストするためのテスティングフレームワークです。Dbunitを使ったテストケースでは、XML形式で用意したテストデータを、テストの実行前にデータベースに反映することができます。データベースの状態が「テストができる状態」になれば、難しかったデータベース絡みのテストも容易におこなえるというわけです。

ご紹介ファイル #03

名前:

Dbunit(1.5.5)

URL :

http://dbunit.sourceforge.net/

機能・特徴:

・JUnitを拡張して、データベース絡みのテストを容易におこなうためのフレームワーク。
・データベースとXMLファイルを相互変換が可能。
・Antから利用することもできる。

適用例:

- データベースが絡むコードのテストをおこないたい。
- テスト前にデータベースを一定の状態に保ちたい
- テスト用のデータを XML ファイルに保持して、いつでも DB に反映できるようにしたい。

ライセンス形態:

GNU Lesser Public License (LGPL)

Dbunitを簡単に使ってみる

まずは、データベーステストの前に、Dbunit の持つエクスポート・インポート機能を試してみたいと思います。以下の手順でサンプルの実行環境を準備してください。

・サンプルソースのダウンロード

紙面の都合で全てのソースを載せることはできませんので、サンプルソースを下記の URL からダウンロードして、解凍してください。以降、解凍したディレクトリを「%SAMPLE_HOME%」と呼びます。サンプルプログラムのディレクトリ構成は次のようになっています。

図：サンプルのディレクトリ構成

```
[%SAMPLE_HOME%]
  +[classes]  クラス出力先
  +[data]     HSQLDB のデータ格納ディレクトリ
  +[lib]      ライブラリ
  +[src]      ソース
  +[sql]      SQL
  +build.xml  Ant スクリプト
```

・サンプルの動作環境

Windows2000,J2SDK1.4.2 で動作確認をおこなっています。また Jakarta Antが必要ですので、Jakarta のサイトからダウンロードして、Ant をインストールしてください。

・HSQLDB について

今回はデータベースとして HSQLDB を使用します。HSQLDB は Java で作られたシンプルなデータベースです。サンプルにはテーブルを作成して、テストデータ挿入済みの HSQLDB を含んでいます。HSQLDB 起動用の Ant スクリプトを用意していますので、次のコマンドで HSQLDB を起動して下さい。以降は HSQLDB が起動済みのものとして解説します。

HSQLDB の起動

```
%SAMPLE_HOME%> ant hsqldb
      (中略)
```

```
    [junit] Use SHUTDOWN to close normally. Use [Ctrl]+[C] to abort abruptly
    [junit] Mon Sep 15 18:30:53 JST 2003 Listening for connections ...
```

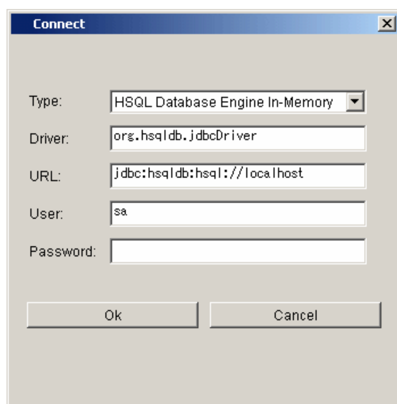
・HSQL DatabaseManager の起動

HSQL DatabaseManager は HSQLDB 用の SQL クライアントで、HSQLDB に含まれています。では、HSQLDB とは別の DOS プロンプトを立ち上げて、次のコマンドで DatabaseManager を起動してみましょう。

HSQL DatabaseManager の起動

```
%SAMPLE_HOME%> ant manager
```

図：HSQLDB の起動画面

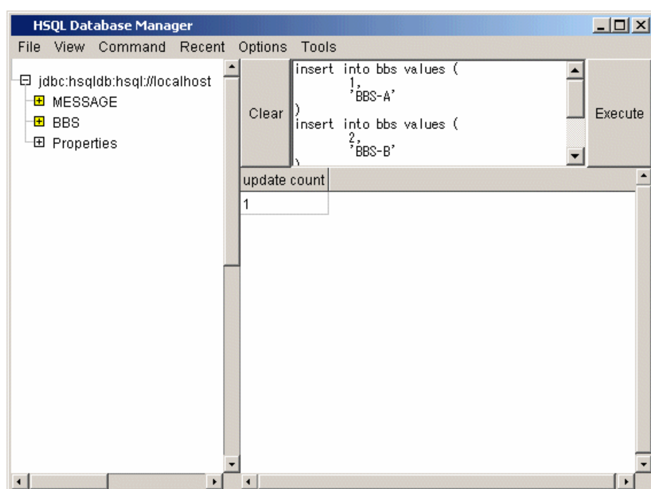


HSQL DatabaseManager が起動したら、以下の設定でデータベースに接続します。

```
Type:HSQL Database Engine In-Memory
Driver: org.hsqldb.jdbcDriver
URL:jdbc:hsqldb:hsql://localhost
User:sa
Password:(無し)
```

接続に成功するとメイン画面が表示されます。画面右のテキストエリアで SQL の実行ができますので、テーブル内のデータを確認してみてください。

図：HSQL DatabaseManager のメイン画面



データベースのデータをエクスポートしてみよう。

　では、データベースのデータを XML ファイルにのエクスポートしてみます。Export.java はデータベースのデータを XML ファイルにエクスポートするサンプルです。次のコマンドで Export.java の main メソッドを実行します。

```
%SAMPLE_HOME%> ant export
```

・データベースコネクションの取得

　Export.java の①の部分で java.sql.Connection を Dbunit 内部でのコネクションクラスである org.dbunit.database.DatabaseConnection のコンストラクタに渡してラップしています。

・DB から IDataSet オブジェクトを生成

　Export.java の②の部分で DatabaseConnection にテーブル名の配列を渡して、IDataSet オブジェクトを取得しています。サンプルでは、Message テーブルのデータを持つ、IDataSet オブジェクトを取得しています。

　また、④のようにテーブル名を指定しない場合、全てのテーブルデータを持つ、データセットオブジェクトが返されます。

・IDataSet インタフェイス

　Dbunit 内部ではデータベースや XML から取得したデータはインタフェイス IDataSet に変換されて利用されます。インタフェイスを使うことで、データの取得元を意識することなく同一に扱うことができます。また、Dbunit で使用される主な API の一覧を載せていますので、参考にしてください。（図：Dbunit の主な API の一覧）

・データセットを XML ファイルに出力

　Export.java の③⑤の部分でデータセットオブジェクトを XML ファイルに出力しています。出力されたファイル名はそれぞれ message.xml と all.xml です。XML は要素名にテーブル名、属性名にカラム名が使用されます。

Export.java

```
package sample;
(import 文は省略)
/**
 * データエクスポートサンプル
 * DB のテーブル -> XML ファイル
 */
public class Export {
    /**
     * コネクションの生成
     */
    public static IDatabaseConnection getConnection() throws Exception {
        String driver = "org.hsqldb.jdbcDriver";
        String url = "jdbc:hsqldb:hsql://localhost";
        String user = "sa";
        String password = "";
```

```
        // コネクションの取得
        Class.forName(driver);
        Connection conn = DriverManager.getConnection(url, user, password);
        IDatabaseConnection dbConn = new DatabaseConnection(conn);                    ・・・①
        return dbConn;
    }
    public static void main(String[] args) throws Exception {
        // コネクション取得
        IDatabaseConnection dbConn = getConnection();

        // 任意のテーブル(message)をエクスポート
        String[] tableNames = { "message" };
        IDataSet userDataSet = dbConn.createDataSet(tableNames);                      ・・・②
        FlatXmlDataSet.write(userDataSet, new FileOutputStream("message.xml"));       ・・・③

        // 全てのテーブルをエクスポート
        IDataSet fullDataSet = dbConn.createDataSet();                                ・・・④
        FlatXmlDataSet.write(fullDataSet, new FileOutputStream("all.xml"));           ・・・⑤
    }
}
```

mesasge.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <MESSAGE MESSAGE_ID='1001' TITLE='タイトル 1001' NAME='Agata' MESSAGE='メッセージ 1001' POSTED_TIMESTAMP='2003-09-14
23:37:06.545'/>
  <MESSAGE MESSAGE_ID='1002' TITLE='タイトル 1002' NAME='Agata' MESSAGE='メッセージ 1002' POSTED_TIMESTAMP='2003-09-14
23:37:06.545'/>
</dataset>
```

all.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <MESSAGE MESSAGE_ID='1001' TITLE='タイトル 1001' NAME='Agata' MESSAGE='メッセージ 1001' POSTED_TIMESTAMP='2003-09-14
23:37:06.545'/>
  <MESSAGE MESSAGE_ID='1002' TITLE='タイトル 1002' NAME='Agata' MESSAGE='メッセージ 1002' POSTED_TIMESTAMP='2003-09-14
23:37:06.545'/>
  <BBS BBS_ID='1' BBS_NAME='BBS-A'/>
  <BBS BBS_ID='2' BBS_NAME='BBS-B'/>
  <BBS BBS_ID='3' BBS_NAME='BBS-C'/>
```

```
</dataset>
```

データベースのインポート

次にインポートを試してみます。Import.java は XML ファイルのデータを DB にインポートするサンプルです。実行は次のコマンドでおこないます。

```
%SAMPLE_HOME%> ant import
```

Import.java の①の部分で XML ファイルをデータセットオブジェクトに変換しています。後は、②の部分で DatabaseOperation.CLEAN_INSERT を使ってデータベースにクリーンインサートの操作を実行するだけです。クリーンインサートではテーブルのデータを全て削除してから、インストール処理がおこなわれます。DatabaseOperation は削除のみをおこなう操作(DELETE)や、挿入のみをおこなう操作 (INSERT) などの操作を、static なフィールドに保持しています。

これで、DB と XML の相互変換ができるようになりました。意外と簡単ですね。データベースの「状態」をいつでも XML ファイルにストレージできれば、単体テストだけでなく、最終的なシステムのテスト環境を作るのにも役に立ちそうです。

Import.java

```
package sample;
(import 文は省略)
/**
 * インポートのサンプル
 * XML -> DB
 */
public class Import extends Export {
    public static void main(String[] args) throws Exception {
        IDatabaseConnection dbConn = getConnection();
        IDataSet dataSet = new FlatXmlDataSet(new FileInputStream("full.xml"));            ・・・①

        // データベースにデータをクリーンインサート
        DatabaseOperation.CLEAN_INSERT.execute(dbConn, dataSet);                            ・・・②
    }
}
```

・Dbunit の主な API の一覧

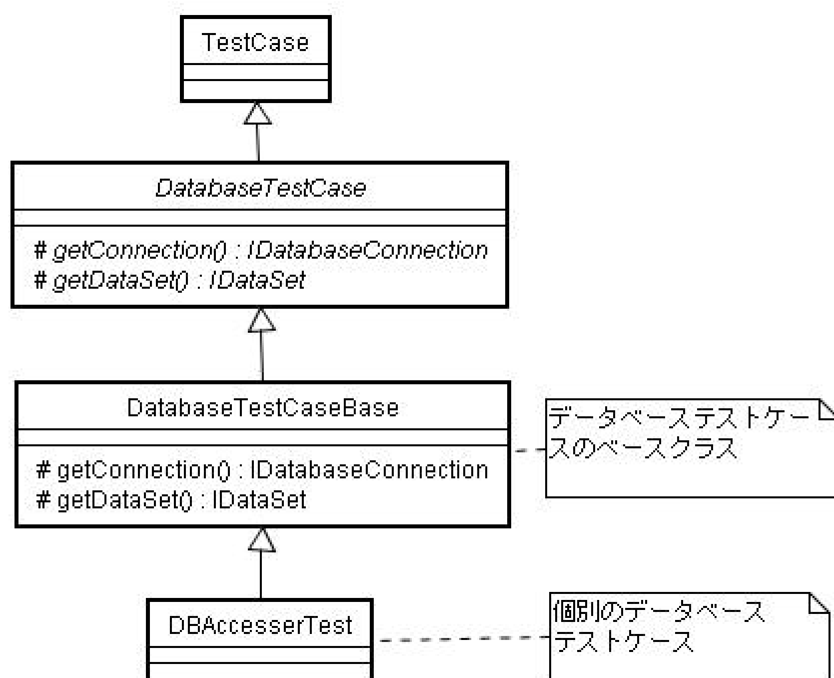
クラス名	メソッド名	説明
IDatabaseConnection	createDataSet	データベースから IDataSet を取得
	createQueryTable	条件を指定して IDataSet を取得
	getRowCount	データ件数を取得
IdataSet	getTable()	テーブル名を指定して、ITable を取得
	getTableMetaData	テーブル名を指定して、ITableMetaData を取得
	getTableNames	テーブル名の一覧を取得
	getTables	ITable の配列を取得

Itable	getRowCount	データ件数を取得
	getTableMetaData	ITableMetaData を取得
	getValue	列、行を指定してデータ値を取得
ITableMetaData	getColumns	カラム名の一覧を取得
	getPrimaryKeys	プライマリーキーの一覧を取得
	getTableNames	テーブル名を取得
Assertion	assertEquals	IDataSet,ITable 同士の比較ができる

JUnit と組み合わせて使う

いよいよここからが本番です。Dbunit で単体テストを作る場合、DatabaseTestCase を継承して作成します。DatabaseTestCase は JUnit の TestCase を継承して拡張した抽象クラスです。下位クラスで、データベースコネクションを生成する getConnection メソッドと、テストデータ IDataSet を生成する getDataSet メソッドを実装する必要があります。テストケースの継承関係は次のようになります。(図：サンプルにおけるクラス図)

図：サンプルにおけるクラス図



・データベーステスト用のベースクラスを作る

DatabaseTestCaseBase.java は DatabaseTestCase を継承した、各テストクラスのベースとなるクラスです。個別のデータベーステストケースはこのベースクラスを継承して作ることにします。DatabaseTestCaseBase.java の①の部分でファイル「all.xml」から IDataSet オブジェクトを生成し、テストデータとして使用します。テストの実行前にこのテスト用データが自動的にデータベースに反映されることになります。

・個別のデータベーステストクラス

個別のテストケースは通常の JUnit で作成するテストケースとほとんど同じです（ソース：DBAccesserTest.java)。違う点は、DatabaseTestCase を継承している点と、テストの実行前にテスト用データがデータベースに反映済みであるという点です。個別のテストの流れをまとめると次のようになります。

1.　テストの事前状態を構築

　XML ファイルからデータベースにテストデータを流し込みます。スーパークラスの DatabaseTestCase が自動的にこの処理をおこないます。

2.テストの実行

　DBAccesserTest.java の①や②のようなデータベースにアクセスするコードを書きます。

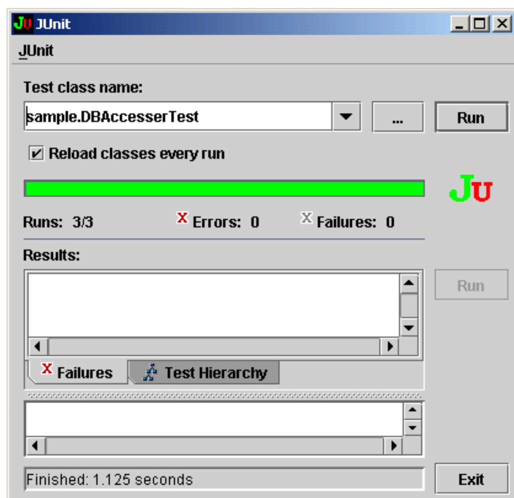
3.テストの事後状態と期待値を比較

　DBAccesserTest.java の③や④のようにデータベースアクセス後の戻り値、およびデータベースの状態が期待どうりかどうかを確認します。また Assertion クラスを使うと、ITable,IDataSet 同士を直接比較できて便利です。⑤のように結果の期待値を XML ファイルで用意しておき(delete_expected.xml)、データベースから取得した ITable,IDataSet と比較する方法がおすすめです。

次のコマンドでテストを実行してみましょう。JUnit が起動して、緑のバーが伸び、データベースアクセスコードが正しいことを確認することができるはずです。

```
%SAMPLE_HOME%>ant test
```

DBAccesserTest の実行結果



DatabaseTestCaseBase.java

```
package sample;
(import 文は省略)
/**
 * データベースアクセスクラスのベースとなるクラス
 */
public class DatabaseTestCaseBase extends DatabaseTestCase {
    ・・・(中略)
    /** データベースコネクションを作成 */
    protected IDatabaseConnection getConnection() throws Exception { ・・・(中略) }
```

```
    /** データベースの初期化用データを取得 */
    protected IDataSet getDataSet() throws Exception {
        IDataSet dataSet = new FlatXmlDataSet(new FileInputStream("all.xml"));          ・・・①
        return dataSet;
    }

    /** データベース内の最新のテーブルデータを取得するユーティリティメソッド */
    protected ITable getTable(String tableName) throws Exception {  ・・・(中略) }

    /** 指定されたテーブルのデータ件数を取得するユーティリティメソッド */
    protected int getTableRowCount(String tableName) throws Exception { ・・・(中略) }
}
```

DBAccesserTest.java

```
package sample;
(import 文は省略)
/**
 * メッセージデータアクセスクラスのテスト
 */
public class DBAccesserTest extends DatabaseTestCaseBase {
    /** データベースアクセスクラス */
    private DBAccesser accesser = null;
    /**
     * コンストラクタ
     */
    public DBAccesserTest(String name) {
        super(name);
    }
    /**
     * テストの初期化
     * 各テスト実行前に呼び出されます。
     */
    protected void setUp() throws Exception {
        super.setUp();
        // データベースアクセスクラスを用意
        Connection conn = getConnection().getConnection();
        accesser = new DBAccesser(conn);
    }
    /**
     * 1件分のメッセージデータの取得テスト
```

```
 */
public void testGetMessage() throws Exception {

    // データの取得、
    // メッセージ ID を指定してデータを正しく取得できるか？
    Message msg = accesser.getMessage(1001);                    ・・・①
    assertEquals(1001, msg.getMessageId());
    assertEquals("タイトル 1001", msg.getTitle());
    assertEquals("Agata", msg.getName());
    assertEquals("メッセージ 1001", msg.getMessage());

    // 存在しないメッセージ ID の場合 null が返される
    msg = accesser.getMessage(0);
    assertNull(msg);

}
/**
 * 1件分のメッセージデータの挿入テスト
 */
public void testInsertMessage() throws Exception {

    // データ件数チェック、データは 2 件のはず
    assertEquals(getTableRowCount("message"), 2);

    // message テーブルに 1 件分のデータ挿入
    Connection conn = getConnection().getConnection();
    DBAccesser accesser = new DBAccesser(conn);
    Message inMsg = new Message();
    inMsg.setMessageId(1003);
    inMsg.setTitle("タイトル 1003");
    inMsg.setName("Hashimoto");
    inMsg.setMessage("メッセージ 1003");
    accesser.insertMessage(inMsg);                              ・・・②

    // データ挿入後は 3 件のデータ
    assertEquals(getTableRowCount("message"), 3);               ・・・③

    // 挿入データの確認チェック
    Message msg = accesser.getMessage(1003);
    assertEquals(inMsg.getMessageId(), msg.getMessageId());     ④
```

```
        assertEquals(inMsg.getTitle()    , msg.getTitle());
        assertEquals(inMsg.getName()     , msg.getName());
        assertEquals(inMsg.getMessage()  , msg.getMessage());
    }
    /**
     * 1件分のメッセージデータの削除テスト
     */
    public void testDeleteMessage() throws Exception {

        // データ件数チェック、データは2件のはず
        assertEquals(getTableRowCount("message"), 2);

        // メッセージ削除
        accesser.deleteMessage(1001);

        // 削除データは取得できないはず
        Message msg = accesser.getMessage(1001);
        assertNull(msg);

        //期待値のXMLファイルとDBを比較
        IDataSet dataSet = new FlatXmlDataSet(new FileInputStream("delete_expected.xml"));
        ITable expectedTable = dataSet.getTable("message");
        Assertion.assertEquals(expectedTable, getTable("message"));                ・・・⑤
    }
}
```

delete_expected.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <MESSAGE MESSAGE_ID='1002' TITLE='タイトル1002' NAME='Agata' MESSAGE='メッセージ1002'/>
</dataset>
```

その他補足事項

・JUnitではカバーできない部分をDbunitで

今回はデータベースにアクセスするクラスのテストのみをおこないました。データベースにアクセスするクラスのテストも大事なのですが、これだけを見ると「テスト用のMessageオブジェクトを作成してJUnitでビジネスロジックをテストすればいいじゃん」「この程度なら、わざわざやらなくてもいいんじゃない？」と感じられる方もいらっしゃるかもしれません。実際の業務では「データベースにアクセスするクラス」を「ビジネスロジック」からじゃんじゃん呼び出すことも多いので、JUnitよりもDbunitを使って「ビジネスロジックのコード」「データベースにアクセスするコード」を同時にテストするのが、テストのコストと品質とのバランスの上で良い方法だと考えます。この場合もデータベースの状態を「ビジネスロジックのテストができる状態」にすることがポイントになります。テストの種類や粒度に応

じて、テスト用のデータを何パターンか用意する必要があるかもしれません。

・更新系の処理で特に効果的

update, delete, insert など更新系の処理では「データベースの事前状態・事後状態」がはっきりとしています。このような更新系の処理に対するテストはぜひ Dbunit で作りましょう。テストの修正が発生しても「修正前にテストが成功」→「修正」→「修正後にテストが成功」という手順で、「変更が正しくおこなわれた」ことを確認することができます。

・DatabaseTestCase の前処理と後処理の変更

DatabaseTestCase の前処理、後処理を変更するには getSetUpOperation、getTearDownOperation をオーバライドします。例えば、前処理でクリーンインサートをおこないたくない場合は下位クラスで次のように定義します。

```
protected DatabaseOperation getSetUpOperation() throws Exception {
    // return DatabaseOperation.CLEAN_INSERT;  //もともとの前処理はクリーンインサート
    return DatabaseOperation.NONE; // 何もしない
}
```

・Ant タスク

紙面の都合でご紹介できませんが、Dbunit は Ant タスクが用意されていて、Ant から呼び出して使用することができます。詳しくは Dbunit のドキュメントをご覧ください。

おわりに

Dbunit を使うと、あきらめかけていたデータベース絡みのテストが、かなり部分までできるようになります。また単体テストの導入は、品質、開発工数、メンテナンス、拡張性など開発の多くの場面において有効です。テストに対して半信半疑な方もぜひ「だまされたと思って」単体テストやデータベース絡みのテストに挑戦してみてください。最初は難しいかもしれませんが、試行錯誤を繰り返すうちに、きっとテストに対する雲も晴れて光が見えてくると信じています。

■ 今月の道草スポット

・eXtreme programming FAQ　http://objectclub.esm.co.jp/eXtremeProgramming/

日本のエクストリームプログラミングの総本山です。ここからリンクされている文書は、私たち開発者に疑問を投げかけ、また勇気を与えることでしょう。単体テストに半信半疑な方、ソフトウェアプロセスに不満がある方、まずはここを訪れてみてはいかがでしょうか。

・Mobster Project Users http://www.mobster.jp/

筆者が参加する喫煙室系メーリングリストです。話題は Java やデザインパターン、エクストリームプログラミングから、熱帯魚、テクノミュージックまで雑多的です。興味を持たれた方はぜひ気軽に参加してみてください。