# DbUnitとAnthillによるテスト環境の制御

## DbUnit がどのようにしてテストと開発作業を簡略化するかを説明します

Philippe Girolami
Senior Software Developer
Digiplug

2004年 4月 13日

Extreme Programming 方法論の導入は、主流のJava開発実践においてテストと同時進行させる開発そして連続的な統合をもたらしました。このような手法をサーバー側のjava開発に導入する行為は、ツールが揃っていなければ最悪の事態を招きかねません。連続的な統合にいかにして対処するか、そして（テスト前にデータベースの状態を設定してテスト環境を一貫して制御する為に）JUnitと同時進行でいかにしてDbUnitを使用するかを、ソフトウェア開発者であるPhillippe Girolami氏がこの記事で説明します。

### DbUnitをプロジェクトに導入する

DbUnitの導入は簡単です。プロジェクトファイルのダウンロード情報ついては、参考文献を参照して下さい。3つのJARファイル全てをプロジェクトのテスト用ビルドターゲットに導入出来ます。

複数のスキーマと関わる環境にて作業される場合には、`DbUnit.qualified.table.names`のプロパティーを`true`に設定するのが望ましいでしょう。この傾向は（ユーザー一人ひとりがそれぞれのスキーマを所有する）Oracleを使用する開発部隊にて頻繁に見受けられます。これにより、テーブル名の頭にスキーマ名を付け加える手間を介さずに、テストデータをチームで共有する事を可能にします。

テストはソフトウェアの開発で欠かす事の出来ない最も重要な作業の一つです。テストが可能な限り頻繁に自動的に実行される連続的な統合、そしてテストが最優先される環境を推奨する事により、Extreme Programming(XP)はその論理を貫いています。しかしながら、XPと関与しないほとんどの開発現場では、非レグレッション・テスト（レグレッション・テストでは無いテスト）、ブラック・ボックス・テスト、機能テストやその他の名で通る別のテスト方法が採用されています。数多くの開発プロジェクトがリレーショナル・データベースを使用してデータを保管しますので、それぞれのテスト計画においてテストがもたらすデータへの影響を考慮しなくてはなりません。テストがデータに悪影響を及ぼし整合性のない状態にした場合、その後のテストは全て失敗に終わります。この問題に対処する方法の一つとして、テストを実行する前にデータベース状態を整合性のある状態に設定する方法が挙げられます。この記事では、JUnitと共にDbUnitを使用し我々のチームがどのようにしてそれを達成したか、そしてどのようにしてAnthillを駆使してテスト報告作成を自動化させたかを説明します。一見これは費用の掛かる設定に見えますが、実際にはそのような事はなく、有効な手段として証明済みです。

## データベースの内容を表記する

DbUnitとは、テスト実行の合間にデータベースを任意の状態に設定する、Junitの拡張機能です。一つのテストがデータベースを破壊し、その後のテストの失敗を招いたり間違ったテスト結果を出させる等の問題を回避するのに役立ちます。DbUnitはテーブルの内容を読み込み、（ リスト1にて示されているように ）`FaltXmlDataSet`を使用してXMLに保管します。

### リスト1. FlatXmlDataSet の例

```
<dataset>
  <OPERATOR
      ID='APC (Washington/Baltimore)'
      CODE='ABC5APC'
      ENCODED_STRING='aabbcc'/>
  <OPERATOR
      ID='ASA Ritabell'
      CODE='ABC6ASA R'
      ENCODED_STRING='bbccdd'/>
  <OPERATOR
      ID='Advanced Info. Service PLC'
      CODE='ABC1Adva'
      ENCODED_STRING='ccddee'/>
  <OPE_OPERATOR
      ID='Aerial Communications Inc.'
      CODE='ABC2Aeri'
      ENCODED_STRING='ddeeff'/>
</dataset>
```

このデータセットは、OPE_OPERATORと名付けられたデータベース・テーブルの3列を示し、それはTable 1の最後の3行で説明されています。

### リスト1のデータのテーブル定義

| OPE_OPERATOR | | |
|---|---|---|
| **ID** | `INT` | |
| **CODE** | `VARCHAR` | |
| **ENCODED_STRING** | `VARCHAR` | |

それぞれのXMLエンティティーはテーブル内のデータベースを認識し、それぞれの属性値は列の値を表示します。

## テーブルの内容を照会する

DbUnitにより、JDBC クエリーの実行そしてJDBC クエリーからの値の取得が簡単に実行出来ます。JDBCだけでは無くDbUnit JDBC ラッパーを使用する理由は複数挙げられます。

- `Dataset`をSQLクエリーから作成し、DbUnitのアサーション・メソッド（ 下記にて参照 ）を使用出来ます。
- `Dataset`をSQLクエリーから作成し、`FlatXmlDataset`にて保存出来ます。後の機会にデータベースに再ロードする事も可能です。
- 繰り返し（ イテレーション ）を介さずに、列の内容をどの行からも簡単に回収出来ます。

リスト2に含まれるコードは、クエリーの結果を含むITableを作成します。列の番号が１である事を確認した後、（ 0から数えて ）最初の列でFK_OTHER_IDの行が番号『1234』を含んでいる事を確認します。

### リスト2. DbUnitのクエリー機能

```
String query = "SELECT * FROM MEDIA WHERE ID= "+id;
ITable databaseData =
dbConnection.createQueryTable("EXPECTED_DATA",query);

assertEquals(1, databaseData.getRowCount());

BigDecimal foreignKey = (BigDecimal) databaseData.getValue(0,
"FK_OTHER_ID");
assertEquals(new BigDecimal(1234)), foreignKey);
```

## アサーション・メソッドを使用してデータベースの内容をチェックします

リスト3にて示されるようなアサーションがDbUnitに含まれており、2セットのデータ又は（ 同一テーブルの ）2種類の表記を比較する場合に使えます。一般的には、複数のクエリーを同時に実行するよりも、テーブルの正確な内容をそれぞれのテスト実行後にチェックしたい場合に使用します。

### リスト3. DbUnitの付加的なアサーション・メソッド

```
public static void assertEquals(ITable expected, ITable actual);
public static void assertEquals(IDataSet expected, IDataSet actual);
```

## データ作成

テストデータを無から作成するか別の本番データベースのコピーから引き出すかの選択は、データベースのサイズ、スキーマの安定性、そして開発進行状況に依存します。既存のデータベースからコンテンツを引き出す例をリスト4にて示します。（ getConnection()のメソッドはリスト6にて示されてます。 ）

### リスト4. 既存のデータベースからFlatXmlDataSetsを作成します

```
public void extractTables(String targetDirectory,String[] tableNames)
    throws Exception {
    IDatabaseConnection connection = getConnection();

    for (int i = 0; i < tableNames.length; i++) {
        String tableName = tableNames[i];

        IDataSet partialDataSet = connection.createDataSet
                                  (new String[] { tableName });
        FlatXmlDataSet.write
            (partialDataSet, new FileOutputStream
              (targetDirectory + "/" + tableName + ".xml"));
    }
}
```

本番データベース全体をエクスポートする場合、余分な列を排除する（ 又は、この場合には、接続から直接データ・セットを作成する代わりにクエリーを使用する ）必要が生じる可能性があ

ります。テーブルのサイズが巨大な場合、抽出されたデータそのものが問題をはらみがちです。（我々の場合、テーブルによっては、クエリーを駆使してテーブルの一部のみを抽出すれば済みました。）主に全ての外部キーの誘導そしてデータの一貫性の維持に伴う困難な作業に関わるので、列をテーブルから排除する作業も問題をはらみます。

## テストデータの追加

テストデータの追加は、時には面倒な作業に成り得ます。我々の経験によれば、初期の段階でデータを正確に追加する作業で苦労した後、峠を越えて作業が簡単になっただけでは無く、我々のデータベースの構造に対する理解も劇的な飛躍を遂げました。

我々はEnterprise JavaBeans(EJB)の技術でデータベースを隠しましたが、直接体験から得たこの知識はそれでも役に立ちました。開発者はデータベースをより深く理解し、その内容をより速くチェック出来た為、ディバッグ作業はより簡単になりました。それにより、我々がコードとデータベースをリファクタリングする時に大いに役立ちました。

## 基底クラス作成時にDbUnitとJUnitを使用

動作を特殊化する`TestCase`基底クラスを展開する事は、JUnitを使用する開発者に推奨される良き習慣と言えます。DbUnitは`DatabaseTestCase`で独自の特殊化を提供し、ユーザーはそれをニーズ（必要性）に合わせて特殊化出来ます。

まず最初に、リスト5に示される通りに、ユーティリティー・メソッドの追加先として、`ProjectDatabaseTestCase`と呼ばれる基本テスト・ケースを作成します。そして、`setUp()`と`teardown()`を再定義して、DbUnitを介するデータベースへの接続を成立させそして破壊するようにします。

### リスト5. 基底クラスの定義そしてデータベース設定の基本的なメソッド

```
public class ProjectDatabaseTestCase extends DatabaseTestCase
{
     /** Use this connection to perform database setup */
     protected IDatabaseConnection connection;

     public DatabaseTestCase (String s)
     {
          super(s);
     }

     protected void setUp() throws Exception
     {
          super.setUp();
          connection = getDbUnitConnection();
     }

     protected void tearDown() throws Exception
     {
          connection.close();
          super.tearDown();
     }
}
```

前述のメソッドにクラス内で使用された様々なメソッドが、リスト6にて表記されています。

## リスト6. 様々なユーティリティー・メソッド

```
/**
 * This method returns a DbUnit database connection
 * based on the schema name
 */
private IDatabaseConnection getDbUnitConnection() throws Exception
{
   IDatabaseConnection connection = new DatabaseConnection (getJDBCConnection(), getSchemaName());
   return connection;
}

private IDataSet getFlatXmlDataSet(String tableName) throws Exception
{
   URL url = DatabaseTestCase.class.getResource( "/"+ tableName + ".xml");
   if (url == null)
      throw new Exception("could not find file for " + tableName);

   File file = new File(url.getPath());
   return new FlatXmlDataSet(file);
}

/** Implement yourself */
private Connection getJDBCConnection() throws Exception
{
   /* Get your JDBC connection through a data source of JDBC itself */
}
* Implement yourself */
private Connection getSchemaName() throws Exception
{
}
```

上記のコードに関する考察

- JDBC接続をどのような手段（データソース(`DataSource`)がシリアライズ可能（`Serializable`）な場合には、アプリケーション・サーバーのJNDI ツリーを介して、又はJDBCを直接使用する。）で確立したいかによってメソッドの実装は左右されますので、`getJDBCConnection()`メソッドは表記されていません。
- `getDbUnitConnection()` メソッドはデータベースへのDbUnit接続を返します。DbUnitの`DatabaseConnection`のコンストラクターはスキーマ名を採用出来ます。そうする事により、全てのテーブル名の頭にスキーマ名を追加する必要はありません。
- `getFlatXmlDataSet`メソッドは、クラスパスに位置するXMLファイルのコンテンツに含まれるDbUnitデータセットを作成します。

最後に、テスト予定のテーブルにデータを実際に挿入します。DbUnitは様々なデータベース制御を可能にし、我々はそのうちの2種類を使用しました。

- `DELETE_ALL`は、テーブル内の全ての列を消去します。
- `CLEAN_INSERT`は、テーブル内の全ての列を消去し、供給されたデータ・セットからの列を挿入します。

下記に示された`ProjectDatabaseTestCase`内の4つのメソッドのみを必要とします。

- **`insertFileIntoDb()`**: ファイルをデータベースに挿入します。
- **`emptyTable()`**: データベーステーブルを空にします。
- **`insertAllFilesIntoDb()`**: 全てのプロジェクト用ファイルを挿入します。

- **`emptyAllTables()`**: プロジェクト内の全てのテーブルを空にします。

リスト7にてこれらのメソッドが使用されています。

### リスト7. データベース設定のための基本的なテストにて使用されるメソッド

```
/** A method to insert all tables into the database.
 *  Specify all tables to be inserted
 */
 protected void insertAllFilesIntoDb()  throws Exception
 {
    insertFileIntoDb("PRODUCT");
    (...)
    insertFileIntoDb("ACCOUNT");
 }

 /**
  * This method inserts the contents of a FlatXmlDataSet file
  * into the connection
  */
  protected void insertFileIntoDb(String tableName) throws Exception
  {
     DatabaseOperation.CLEAN_INSERT.execute(connection,getFlatXmlDataSet(tableName));
  }

  /** Empty a table */
  protected void emptyTable(String tableName) throws Exception
  {
     IDataSet dataSet = new DefaultDataSet(new DefaultTable(tableName));
     DatabaseOperation.DELETE_ALL.execute(connection, dataSet);
  }

  /** Empty all the tables from the database    */
  protected void emptyAllTables() throws Exception
  {
     emptyTable("ACCOUNT");
     (...)
     emptyTable("PRODUCT");
  }
```

## 全てを同時に使用・実行します

基底クラスが設定されれば、リスト8にて示される通り、DbUnitを使用してデータベースを滞り無く設定し、メソッドを実行し、そして返される数値をチェックするのは簡単です。

### リスト8. 実際のテスト・ケース

```
public void setUp() throws Exception
{
   super.setUp();
   emptyAllTables();
   service = Service.getInstance();
}

public void testFindProductByPrimaryKey() throws Exception
{
   insertFileIntoDb("PRODUCT");
   ProductDTO productDTO = service.findProductByPrimaryKey(new Integer(12));
   assertNotNull(productDTO);
   assertEquals("product Name", productDTO.getName());
}

public void testCreateAProduct() throws Exception
```

```
{
   service.createProduct("newly created product name");

   String query = "SELECT * FROM PRODUCT";
   ITable databaseData = dbConnection.createQueryTable("EXPECTED_DATA",query);
   assertEquals(1, databaseData.getRowCount());
   String productName = (String) databaseData.getValue(0, "NAME");
   assertEquals("newly created product name", productName);
}
```

このテストでは、データベースを空にし、一つのテーブルの内容を挿入し、その主キーによりプロダクトを探す為のファインダ・メソッドが正常に機能する事を（正常な属性の要素を返す事をチェックする事により）確認します。そしてオブジェクト作成の機能をテストし、DbUnitのクエリー機能を使用してデータベースの内容を検証します。

テスト終了時では無くテストのセットアップの段階でデータベースの整理が行われる事が、ここで述べられるべき重要な課題です。それぞれのテストがテスト終了時でのデータベース整理に頼らなくてはならないのは好ましくは無いと言えます。

### データ挿入時の留意点

データベースの統合性から生じる制約は、データの挿入と消去を特定の順序で行なう事を強要します。`insertAllFiles()`そして`emptyAllTables()`のメソッドを作成する場合、順序は関係無い訳では無く、実際には統合性の制約により強要されます。

別の予測される落とし穴としましては、列によっては挿入されていないように見受けられる場合もあります。この場合のほとんどは、`FlatXmlDataSet` 内の第1列目には列が一つ足りないからです。DbUnit がその行を他の列でも認識しない事もあり得ます。例えば、リスト9にて定義されるデータ・セットを挿入すれば、唯一のnon-null 列が主キー`PK_ACC_ID`である二つの列を含むテーブル`ACC_ACCOUNT`のような結果を招きます。

### リスト9. ACCOUNTの第2列のNAMEが存在しません

```
<dataset>
   <ACCOUNT ID='1' />
   <ACCOUNT ID='2' NAME='first name' />
</dataset>
```

最初の列の記述がテーブル内の全ての行を含む事を心掛けましょう。NULL値を挿入する必要がある場合、リスト10にて示される通りにその列を第2列にして下さい。

### リスト10. ACCOUNTの第2列のNAMEが存在します

```
<dataset>
   <ACCOUNT ID='2' NAME='first name' />
   <ACCOUNT ID='1' />
</dataset>
```

## テストデータの整理

XMLデータ・セットをファイルに保管する事をDbUnitは可能にします。一つのファイル内にて全てのデータベースを保管する事をも可能にします。リスト11 はACCOUNTそしてMEDIAのテーブルの内容を示します。

### リスト11. 2つのテーブルに関わるFlatXmlDataSet の使用例

```
<dataset>
   <ACCOUNT NAME='first name' />
   <ACCOUNT NAME='second name' />

   <MEDIA ID='123' />
   <MEDIA ID='234' />
</dataset>
```

### Anthillとの連続的な統合

Anthill はフリーの自動化ビルドツール（ 参考文献にて参照 ）であり、ビルドをスケジュールし結果を公開する事によりXPに精通したプロジェクトチームが連続的な統合を実践するのを手助けします。CVS等のバージョン管理ツール からソースのチェックアウト、ビルドスクリプトの実行、結果の公開、そしてユーザーへの結果報告から、ビルドは形成されます。それはAntと整然と統合し、通常のビルドスクリプトの再利用を可能にします。

どのようにしてテストデータを保管するかが重要です。それぞれのテーブルの内容を別々のファイルに保管するか、それともシステムの主要なエンティティーと関わる全てのテーブルの全ての列を一つのファイルに保管するか？どちらも確実な方法とは言えません。

前者のやり方では、複数のテーブルを通してデータの一貫性を保証するのはより難しいですが、既に存在するデータベースからクエリーを作成するのはより簡単です。後者の方法では、それぞれのテストのテストセットを作成するのは簡単ですが、ほとんどのシステムが統一された主要なエンティティーを中心に設計されている訳では無い事を考慮に入れればあまり実用性は高くはありません。我々は、「1つのテーブルに1つのファイル」の方式を採用しました。

## Anthill内のテスト・スイートを実行し、結果を報告します

統合による問題を和らげるやり方として、連続的な統合はXP熟練者に推奨されています。全てのコードを頻繁に統合する事により、問題が起きた場合には確実に原因を容易に突きとめられます。データのチェックアウト、ビルド、そしてデプロイを伴い、受け入れテストを実行する為、統合は非常に時間を浪費する作業と成り得ます。幸運な事に、AnthillやCruiseControl等のツールを使用する事により、これらのプロセスの大半は自動化出来ます。もしも未だAnt等を使用してビルドプロセスを自動化させていないのであれば、自動化される事をお奨めします。ビルドプロセスが既に自動化されていれば、テスト・セクションをビルドに追加するのが良いでしょう。XPに深く精通しているのであれば、これらを受け入れテストとして見なすべきです。我々と同様の見解を抱くのであれば、（ 単体テスト、受け入れテスト、そしてその他のテスト方法において ）これらが全てのテストです。

我々のビルドプロセスはAntをベースにしており、Anthillにより作成されます。我々の主な課題は、Anthillがテスト失敗を報告しつつもテスト結果を公開するようにする事でした。Anthillの難点は、ビルドスクリプトが失敗すれば公開スクリプトが実行されず、その場合には開発者にテスト報告を提出する手段が無い事です。我々が考えた解決法は、プロパティーが正しい(true)か間違っている(false)かを確認させる事により、公開スクリプトの終了時にAnthillの実行を失敗させる方法でした。

### テストを実行する際の目標

我々がどのようにテストを実行したかを下記にて記述します。我々はバッチ・テスト・メソッドを使いましたが、他のメソッドでも機能します。重要な点は

- JDK 1.3内のXML パーサーを含むクラスパスと正常に機能するように、テストがforkしなくてはなりません。
- エラーや失敗が発生した場合、`testsuite.error` そして`testsuite.failure` のプロパティーはtrueに設定されるべきです。そうでない場合には、放置されます。

特定のモジュールに関わる全てのテストを実行する例をリスト12にて表示します。

## リスト12. 一つのモジュールにてテストを実行

```
<target name="test-common">
   <mkdir dir = "${project.reports}/common"/>

   <junit fork="true" errorproperty="testsuite.error" failureproperty="testsuite.failure">
      <classpath>
         <pathelement location="${out.classes.dir}"/>
         <fileset dir = "${shared.lib.dir}">
            <patternset refid="necessary.jars"/>
         </fileset>
      </classpath>

      <formatter type="xml"/>
      <batchtest todir="${project.reports}/common">
         <fileset dir="${out.src.dir}">
            <include name="**/Test*.java"/>
         </fileset>
      </batchtest>
    </junit>
</target>
```

## テスト結果を公開スクリプトに提供します

どのようにしてテストを我々のビルドプロセスにて実行したかを、リスト13にて示します。

## リスト13. build.xmlの断片：全てのテストの実行そして結果の設定

```
<target name = "all-tests" depends = "test-module1,test-module2">
   <property name="testsuite.error" value="false"/>
   <property name="testsuite.failure" value="false"/>
   <propertyfile file="${deployDir}/tests.results">
      <entry key="testsuite.error" value="${testsuite.error}"/>
      <entry key="testsuite.failure"
value="${testsuite.failure}"/>
   </propertyfile>
</target>
```

Antにて重要な技は、値が既に設定されていない時のみにプロパティー値を設定する機能を知る事です。テスト・ターゲットを実行する度に、`testsuite.error` そして`testsuite.failure` のプロパティー値がtrueになるのはエラー又は失敗が発生した時のみです。

ここで困難なのは、テストスクリプトの出力を主要なAntスクリプトに報告可能な事です。運の悪い事に、これらはAnthillの処理にて存在する2つのそれぞれ別のAntビルドファイルであり、そのようなパラメーターをAnt内のビルドスクリプト間にて受け渡す事は不可能なため、この問題は単純ではありません。ところが、簡単な解決方法があります。それは、公開スクリプトが認識するテスト結果をファイルにて保存する方法です。

同じAntの技を使い、`testsuite.error` そして`testsuite.failure` のプロパティーが常にテスト・スクリプトの終わりに値を与えられるように`<property>`コマンドを使用する方法、そしてファイルにプロパティー値を保存する方法が、リスト13にて示されます。

### テストが失敗した場合、終了時に公開スクリプトを故意に失敗させます

リスト14を使用し、どのテストが失敗しても公開スクリプトが失敗するようにします。ビルドスクリプト内に保存されたプロパティー値がtrueに設定されているかをチェックするだけの事です。

### リスト14. テストが失敗したりエラーが発生した場合、公開スクリプトを故意に失敗させます

```
<condition property="must.fail">
  <or>
    <istrue value="${testsuite.error}"/>
    <istrue value="${testsuite.failure}"/>
  </or>
</condition>

<fail message="Tests didn't run 100%. Check the log and make
necessary changes!" if="must.fail"/>
```

## まとめ

我々のチームはDbUnitとAnthillを2003年当初に売り出すのに成功しました。それ以来、我々は数千ものテストを書き出し自動化させました（テストの内75%はデータベースの状態設定を伴います。）。我々は毎時間これらのテストを実行し、近い内にはより頻繁に実行する事を計画しております。これらのテストは数多くの予期せぬバグを検出しますので、欠かせる事の出来ないツールとしての地位を確立しています。

# 著者について

Philippe Girolami

Philippe Girolamiはフランスのパリ出身。フランスのEcole Centrale de Lilleにて技術学士号、そして米国テキサス州のThe University of Texas at Austinにて科学修士号を取得。現在、Digiplugの中核コンテンツの開発マネジメント、発信、そしてプロビジョニング・プラットフォーム（サービス提供基盤）関連の活動を統率し、全てのフォーマット、発信経路、そして端末仕様に通用する携帯電話用の精錬されたコンテンツを供給する。Ericsson's I-Lab in Paris の開発者として、Ericssonの 複数端末ポータル・オファリング、キャリアグレードJ2EE サーバー、そしてモバイル・グループ・コミュニケーションの開発プロジェクトに貢献。