Getting Started with the Senacor DDT framework

SENACOR TECHNOLOGIES

December 19, 2007

Contents

1 Introduction

This is a short step by step tutorial that will teach you the basics of writing data driven tests with the Senacor DDT framework. All the examples included here may be downloaded from our subversion repository, so feel free to try them out and experiment further.

1.1 Setting up your Java Environment

Your setup is easy. You'll need to be working with JDK version 1.4 (and up) with everything listed below in your class path.

- `ddt-complete-`*current_version*`.jar` (if you downloaded the distribution version as a zip binary, this is one of the jars it contains. You may also use `ant` to build all DDT-jars from the sources. They will be created in the `build/jar` directory.)
- a bunch of third-party libraries from the `lib` directory: junit.jar, parallel-junit.jar, jxl-2.6.1.jar, commons-beanutils-core.jar, commons-logging.jar (- You could also just take everything you find under `lib`)
- the directory where the Excel documents live:
  `doc/tutorial/example_src/com/senacor/ddt/tutorial`

1.2 Choosing the Data Format

The examples in this tutorial are set up for the use with MS Excel. Not because it is on our favorite product list but it is widely available in the enterprise world and well accepted. DDT supports other data formats but there is a reason why we think xls-support is the most important.

Just imagine you were able to show the quality of your piece of software by showing your customer a bunch of xls-files containing the data your tests will work with. They'll be happy campers because they feel home and cozy scrolling through even the biggest spread sheets. Although we feel much more comfortable with any plain text based format, our experience in enterprise projects leads us to the fact that customer acceptance is more important. They somehow *trust* Excel so they will trust your tests even more when they are based on xls files. Just go with the flow.

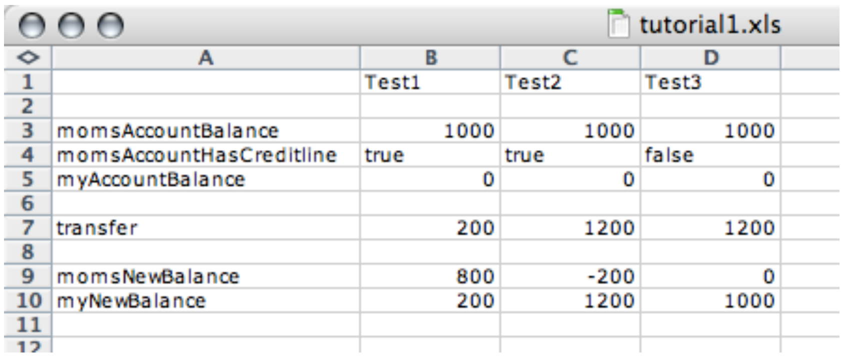
1.3 The Mission Critical Account Class

Most tutorials start off with a "hello world"–example but it's tough to come up with a meaningful test for that. So we chose the next boring example which came to our minds: A banking account class. It's perfectly fine for our little demonstration and the scenery isn't too unrealistic after all: Financial institutes are somewhat picky when it comes to software quality, so let's assume our customer is a bank and we want to make sure there are no bugs in the code we deliver.

So much for the big picture. Here is our bank account class called *Account*. It couldn't be much simpler: An account has a balance and offers methods for making deposits and withdrawals. There is a small piece of logic for the situation when you wish to withdraw more money than you have. In that case it depends on whether you have a credit line or not: If not, only the remaining amount can be withdrawn. This is where a good unit test comes in handy.

```
package com.senacor.ddt.tutorial;

import java.math.BigDecimal;

/**
 * A beginner's bank account class. It doesn't do much.
 * There is a tiny amount of business logic within the
 * withdraw-method we consider worth testing.
 */
public class Account {
  private BigDecimal balance = new BigDecimal("0");

   /**
    * does this account have an unlimited credit line?
    */
  private boolean isCredit;

   /**
    * wihtdraw some money.
    * Only if this account has a credit line,
    * the withdrawn amount may be greater than the balance.
    *
    * @param amount amount to be withdrawn (if possible)
    * @return the actually withdrawn amount
    */
  public BigDecimal withdraw(BigDecimal amount) {
    BigDecimal actualAmount = amount;
    if (!isCredit && balance.compareTo(amount) == -1) {
      actualAmount = balance;
    }
    balance = balance.subtract(actualAmount);
    return actualAmount;
  }

  /**
   * guess what
   */
  public void deposit(BigDecimal amount) {
    balance = balance.add(amount);
  }

  // Getter and setter methods
  // -- what would life be without them?

  public BigDecimal getBalance() {
    return balance;
  }

  public void setBalance(BigDecimal balance) {
    this.balance = balance;
  }

  public boolean isCredit() {
    return isCredit;
  }

  public void setCredit(boolean credit) {
    isCredit = credit;
  }
}
```

What would be a good test? Let's think about a test scenario first before we get to the implementation. Using a spread sheet application we like (or Excel), we specify three different test cases which we think will cover all interesting aspects. Utilizing two different accounts, we transfer money from one account to another and write down the expected values to be verified against. Take a look at the test sheet's screenshot in figure 1.



tutorial1.xls

	A	B	C	D
1		Test1	Test2	Test3
2				
3	momsAccountBalance	1000	1000	1000
4	momsAccountHasCreditline	true	true	false
5	myAccountBalance	0	0	0
6				
7	transfer	200	1200	1200
8				
9	momsNewBalance	800	-200	0
10	myNewBalance	200	1200	1000
11				
12				

Figure 1: test cases for a money transfer

2 A First Shot

Now that we know what the test cases will look like we need to read the data from the `xls` file into the test. The test class does not have to extend any special DDT base class, `AccountTest` is just a regular subclass of `junit.framework.TestCase`. It implements the `DataDrivenTestCase` instead which forces us to implement a getter and setter method for an instance of `TestCaseData`. All our data will be injected by the DDT framework into that instance as soon as we run the test.

DDT needs to know where to read the test data from. That aspect is handled by creating an `ExcelObjectMatrixFactory` which will be used by the `JUnitTestSuiteBuilder` to create a suite for you. For each column of the specified sheet of the given `xls`-file the test method will be run.

Please note that DDT expects the `xls`-file to be accessible from your class path.

```
package com.senacor.ddt.tutorial;

import junit.framework.TestCase;
import junit.framework.Test;
import com.senacor.ddt.test.DataDrivenTestCase;
import com.senacor.ddt.test.TestCaseData;
import com.senacor.ddt.test.junit.JUnitTestSuiteBuilder;
import com.senacor.ddt.objectmatrix.excel.ExcelObjectMatrixFactory;

import java.math.BigDecimal;

/**
 * The simplest DDT-test we could come up with..
 */
public class AccountTest extends TestCase
    implements DataDrivenTestCase {
  TestCaseData testCaseData;

  public void testTransfer() {
    BigDecimal momsAccountBalance =
        testCaseData.getBigDecimal("momsAccountBalance");
    Boolean momsAccountHasCreditLine =
        testCaseData.getBoolean("momsAccountHasCreditline");
    BigDecimal myAccountBalance =
        testCaseData.getBigDecimal("myAccountBalance");
    BigDecimal transferAmount =
        testCaseData.getBigDecimal("transfer");

    Account myAccount = new Account();
    myAccount.setBalance(myAccountBalance);

    Account momsAccount = new Account();
    momsAccount.setBalance(momsAccountBalance);
    momsAccount.setCredit(momsAccountHasCreditLine.booleanValue());

    myAccount.deposit(momsAccount.withdraw(transferAmount));

    BigDecimal momsNewBalance =
        testCaseData.getBigDecimal("momsNewBalance");
    BigDecimal myNewBalance =
        testCaseData.getBigDecimal("myNewBalance");

    assertEquals(momsNewBalance, momsAccount.getBalance());
    assertEquals(myNewBalance, myAccount.getBalance());
    assertEquals(myAccountBalance.add(momsAccountBalance),
        myNewBalance.add(momsNewBalance));
  }

  public static Test suite() {
    ExcelObjectMatrixFactory objectMatrixFactory =
        new ExcelObjectMatrixFactory(
            "tutorial1.xls",
            new String[]{"TransferMoneyTests"});

    JUnitTestSuiteBuilder jUnitTestSuiteBuilder =
        new JUnitTestSuiteBuilder(objectMatrixFactory, AccountTest.class);
    return jUnitTestSuiteBuilder.buildSuite();
  }


  public void setTestCaseData(TestCaseData tcd) {
    this.testCaseData = tcd;
  }

  public TestCaseData getTestCaseData() {
    return testCaseData;
  }
}
```

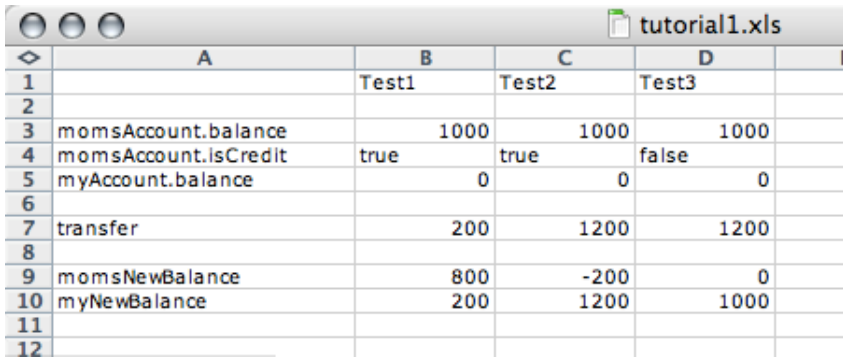
Once you start that test it will run all test methods once for each column of the spread sheet. Well, all columns but the first: It is used to name the lines of the matrix you refer to. Take a look at the usages of `TestCaseData`'s `getBigDecimal`-method to retrieve a single value by the name that is written in the first column: `testCaseData.getBigDecimal("momsNewBalance")` will return an instance of `BigDecimal` for the value 200 in the first, -200 in the second and 0 in the third test case.

`TestCaseData` offers a bunch of getters for all common types. This is a good moment to go check its JavaDoc documentation to find out what is there for you to use.

3 Getting better

In the last section you learned how to extract values from the sheet by using `TestCaseData`'s getter methods for each value. It is certainly nice to have DDT convert the text into the right types by using those convenient getter methods but it is still quite cumbersome to fill all of an object's properties that way. In our example, mom's account needs a balance to be set and the boolean value for the credit line. Now imagine that for an object with twenty attributes. It would clutter up your test source code.

DDT supports filling whole object graphs by letting you write paths instead of just names in the first column of the sheet. Technically we are saving only one line of code by using `TestCaseData`'s `fillBean` method to populate all properties at once instead of setting each value separately. For real world classes with lots of attributes and deep object graphs this is huge advantage, though. And there is another difference compared to the former version of the test class: You no longer have to worry about choosing the correct getter method to correspond the attribute's type. It is all handled by the DDT framework!



	A	B	C	D
1		Test1	Test2	Test3
2				
3	momsAccount.balance	1000	1000	1000
4	momsAccount.isCredit	true	true	false
5	myAccount.balance	0	0	0
6				
7	transfer	200	1200	1200
8				
9	momsNewBalance	800	-200	0
10	myNewBalance	200	1200	1000
11				
12				

Figure 2: test cases for a money transfer

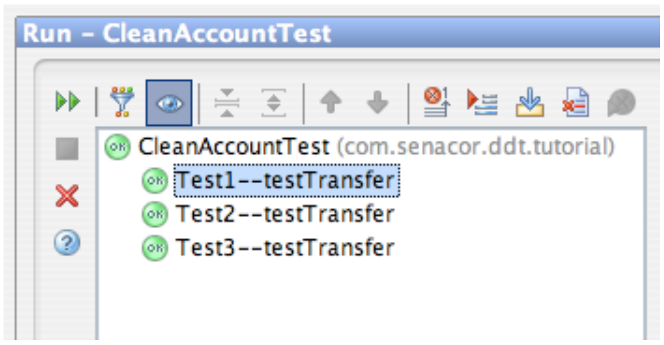
```java
package com.senacor.ddt.tutorial;

import junit.framework.TestCase;
import junit.framework.Test;
import com.senacor.ddt.test.DataDrivenTestCase;
import com.senacor.ddt.test.TestCaseData;
import com.senacor.ddt.test.junit.JUnitTestSuiteBuilder;
import com.senacor.ddt.objectmatrix.excel.ExcelObjectMatrixFactory;

import java.math.BigDecimal;
```

```
/**
 * The simplest DDT-test we could come up with..
 * ... but this time we are reading objects instead of single values
 */
public class BetterAccountTest extends TestCase implements DataDrivenTestCase {
  TestCaseData testCaseData;

  public void testTransfer() {
    Account momsAccount = new Account();
    testCaseData.fillBean(momsAccount, "momsAccount");

    Account myAccount = new Account();
    testCaseData.fillBean(myAccount, "myAccount");

    BigDecimal transferAmount = testCaseData.getBigDecimal("transfer");

    myAccount.deposit(momsAccount.withdraw(transferAmount));

    BigDecimal momsNewBalance = testCaseData.getBigDecimal("momsNewBalance");
    BigDecimal myNewBalance = testCaseData.getBigDecimal("myNewBalance");

    assertEquals(momsNewBalance, momsAccount.getBalance());
    assertEquals(myNewBalance, myAccount.getBalance());
    assertEquals(myAccount.getBalance().add(momsAccount.getBalance()),
        myNewBalance.add(momsNewBalance));
  }

  public static Test suite() {
    ExcelObjectMatrixFactory objectMatrixFactory = new ExcelObjectMatrixFactory(
        "tutorial2.xls",
        new String[]{"balanceTests"}
    );

    JUnitTestSuiteBuilder jUnitTestSuiteBuilder =
        new JUnitTestSuiteBuilder(objectMatrixFactory, BetterAccountTest.class);
    return jUnitTestSuiteBuilder.buildSuite();
  }


  public void setTestCaseData(TestCaseData tcd) {
    this.testCaseData = tcd;
  }

  public TestCaseData getTestCaseData() {
    return testCaseData;
  }
}
```

4 An Almost Perfect Test Class

Now let's clean things up. In section 2 we filled the objects we need in our test from an Excel sheet but things still look kind of mixed up. To fix that we'll move all test case initializing code into the `setUp`-method so common objects may be used by additional test methods and leave `TestCaseData`-handling code in the test method only if it is specific to that test method. Take a look at the `testTransfer` method now: All that is left is pure test logic.

There is another improvement to our example test class. The name of the test cases are read from the test case sheet in the overridden `getName`-method so they can be displayed by the Junit runner as shown in figure 3.



CleanAccountTest (com.senacor.ddt.tutorial)
Test1--testTransfer
Test2--testTransfer
Test3--testTransfer

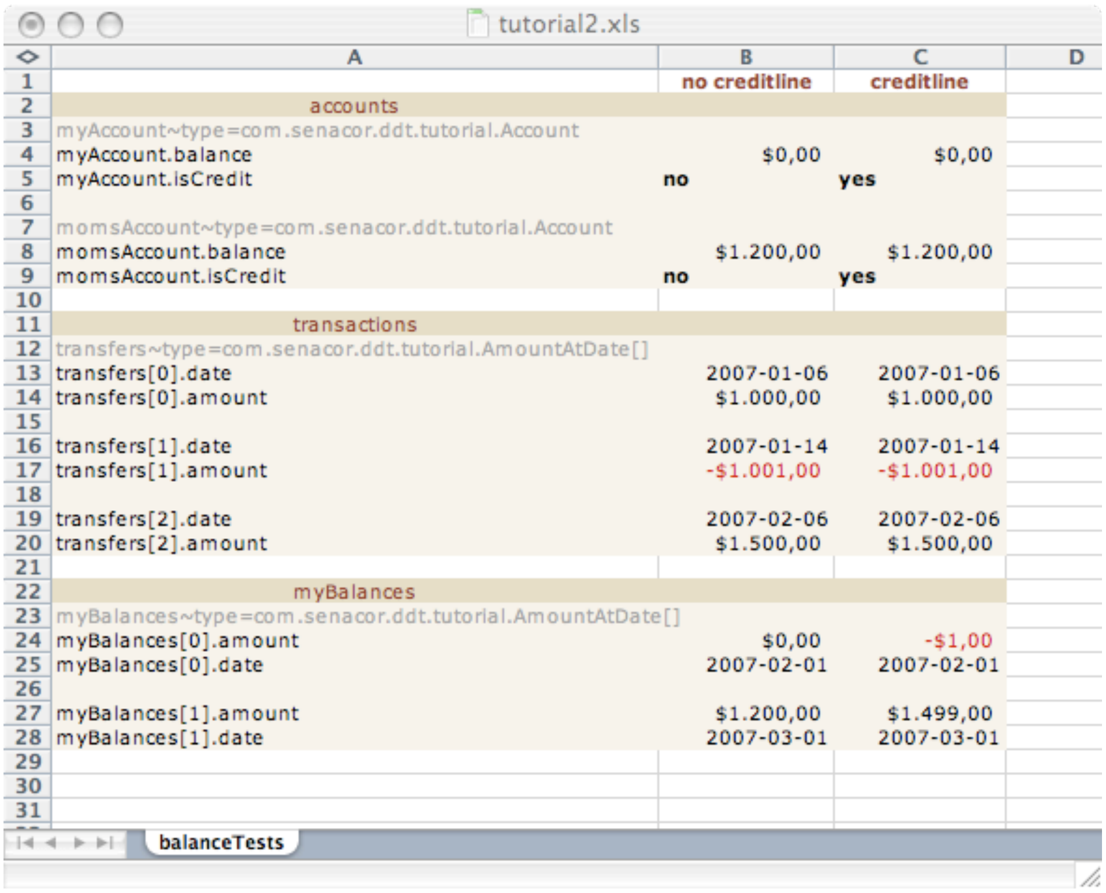
Figure 3: test case names and test method names getting displayed

```
package com.senacor.ddt.tutorial;

import junit.framework.TestCase;
import junit.framework.Test;
import com.senacor.ddt.test.DataDrivenTestCase;
import com.senacor.ddt.test.TestCaseData;
import com.senacor.ddt.test.junit.JUnitTestSuiteBuilder;
import com.senacor.ddt.objectmatrix.excel.ExcelObjectMatrixFactory;

import java.math.BigDecimal;

/**
 * The simplest DDT-test we could come up with..
 * ... in a cleaned up version: using setUp and naming test cases
 */
public class CleanAccountTest extends TestCase implements DataDrivenTestCase {
  TestCaseData testCaseData;
  Account momsAccount;
  Account myAccount;
  BigDecimal momsNewBalance;
  BigDecimal myNewBalance;

  protected void setUp() throws Exception {
    momsAccount = new Account();
    testCaseData.fillBean(momsAccount, "momsAccount");
    myAccount = new Account();
    testCaseData.fillBean(myAccount, "myAccount");

    momsNewBalance = testCaseData.getBigDecimal("momsNewBalance");
    myNewBalance = testCaseData.getBigDecimal("myNewBalance");
  }

  public String getName() {
    return testCaseData.getTestCaseName() + "——" + super.getName();
  }

  public void testTransfer() {
    BigDecimal transferAmount = testCaseData.getBigDecimal("transfer");

    myAccount.deposit(momsAccount.withdraw(transferAmount));

    assertEquals(momsNewBalance, momsAccount.getBalance());
    assertEquals(myNewBalance, myAccount.getBalance());
    assertEquals(myAccount.getBalance().add(momsAccount.getBalance()),
        myNewBalance.add(momsNewBalance));
  }

  public static Test suite() {
    ExcelObjectMatrixFactory objectMatrixFactory = new ExcelObjectMatrixFactory(
        "tutorial1.xls",
        new String[]{"TransferMoneyTests2"}
    );

    JUnitTestSuiteBuilder jUnitTestSuiteBuilder =
        new JUnitTestSuiteBuilder(objectMatrixFactory, CleanAccountTest.class);
    return jUnitTestSuiteBuilder.buildSuite();
  }


  public void setTestCaseData(TestCaseData tcd) {
    this.testCaseData = tcd;
  }

  public TestCaseData getTestCaseData() {
    return testCaseData;
  }
}
```

5 Advanced Stuff

Now that you are familiar with the DDT's basic features we are all set for our show off section. Take a look at the Excel sheet in figure 4. That is what test data should look like, especially when we want to show that to our customer. DDT does not mind you using colors, text styles or data type specific cell formatting for readable calendar dates or currencies in monetary values. Note that we used "yes" and "no" instead of "true" and "false".[1]

This test proves that an account's balance will stay correct over time while some transfers will be processed at specific points of time. In the accounts section of the spread sheet two accounts will be set up. In the center block the transfers are specified and in the section titled "myBalances" the expected balances in certain points of time are stated.



	A	B	C	D
1		no creditline	creditline	
2	accounts			
3	myAccount~type=com.senacor.ddt.tutorial.Account			
4	myAccount.balance	$0,00	$0,00	
5	myAccount.isCredit	no	yes	
6				
7	momsAccount~type=com.senacor.ddt.tutorial.Account			
8	momsAccount.balance	$1.200,00	$1.200,00	
9	momsAccount.isCredit	no	yes	
10				
11	transactions			
12	transfers~type=com.senacor.ddt.tutorial.AmountAtDate[]			
13	transfers[0].date	2007-01-06	2007-01-06	
14	transfers[0].amount	$1.000,00	$1.000,00	
15				
16	transfers[1].date	2007-01-14	2007-01-14	
17	transfers[1].amount	-$1.001,00	-$1.001,00	
18				
19	transfers[2].date	2007-02-06	2007-02-06	
20	transfers[2].amount	$1.500,00	$1.500,00	
21				
22	myBalances			
23	myBalances~type=com.senacor.ddt.tutorial.AmountAtDate[]			
24	myBalances[0].amount	$0,00	-$1,00	
25	myBalances[0].date	2007-02-01	2007-02-01	
26				
27	myBalances[1].amount	$1.200,00	$1.499,00	
28	myBalances[1].date	2007-03-01	2007-03-01	
29				
30				
31				

Figure 4: an advanced sheet using formatting and expandable arrays

First of all, this example introduces a new class named `AmountAtDate` which is used as a container for the combination of a calendar date and an amount. It is common practice to create these kind of container classes to make extracting test case data easy.

```java
package com.senacor.ddt.tutorial;

import java.util.Date;
import java.math.BigDecimal;

/**
 * Container object for a timed money transfer.
 */
public class AmountAtDate implements Comparable {
```

[1]read all about *transformers* in the technical documentation to learn how that works.

```
  /**
   * date of tranfer
   */
  private Date date;

  /**
   * amount of transfer
   */
  private BigDecimal amount;

  public BigDecimal getAmount() {
    return amount;
  }

  public void setAmount(BigDecimal amount) {
    this.amount = amount;
  }

  public Date getDate() {
    return date;
  }

  public void setDate(Date date) {
    this.date = date;
  }

  public String toString() {
    return "transfer $" + amount + " on " + date;
  }

  public int compareTo(Object o) {
    return date.compareTo(((AmountAtDate) o).date);
  }
}
```

The next thing you probably noticed is the use of brackets in the "transactions" and "myBalances" sections. That notation is used to describe elements of an array, just like you would write the same in Java source code. Here comes the real big deal with that feature: Without changing the code of your test class you are able to expand your test case by adding new array elements. Just add some lines and increment the index. DDT will expand the array for you as it reads in the additional line. That means your customer may not only create new test cases by adding new columns to the sheet but also by adding new lines for array elements that will be automatically read when running the test. Let the business folks deliver the contents as you provide the logic of the test class!

Take a look at the test code that imports the data from the sheet: The method `testBalances` reads two arrays from the test sheet but it does not instantiate the arrays by itself. DDT even creates the array for you when you call `TestCaseData`'s `createAndFillBean`-method. But how does it know what type of bean (or array in this case) need to be used? That's what the type annotation[2] in the spread sheet is for. By placing the line

```
transfers~type=com.senacor.ddt.tutorial.AmountAtDate[]
```

you tell DDT that it has to create an array of type `AmountAtDate`, whereas

```
myAccount~type=com.senacor.ddt.tutorial.Account
```

denotes an instance of class `AmountAtDate`.

The ability to create objects not just fill existing objects with data during the spread sheet import is important for two things: If you want to be type safe you need to work with arrays instead of collections but arrays need to be sized correctly when instantiated. Since you don't know how many entries an array will get there is no way of creating the array in advance. DDT resizes arrays as necessary while reading the spread sheet. Furthermore, all entries of an array must not be instances of the same class when using inheritance. You could denote the first entry of transfers to be an instance of `MySpecialAmountAtDate` (that has to extend `AmountAtDate`) by inserting

```
transfers[0]~type=com.senacor.ddt.tutorial.MySpecialAmountAtDate
```

assuming `MySpecialAmountAtDate` extends `AmountAtDate`.

```
package com.senacor.ddt.tutorial;

import com.senacor.ddt.objectmatrix.excel.ExcelObjectMatrixFactory;
import com.senacor.ddt.test.DataDrivenTestCase;
import com.senacor.ddt.test.TestCaseData;
import com.senacor.ddt.test.junit.JUnitTestSuiteBuilder;
import junit.framework.Test;
import junit.framework.TestCase;

import java.util.Arrays;
import java.util.Date;

/**
 * a fully featured DDT test which executes a list of tranfers between two account
 * and tests for the correct balance at certain points of time. All test data objects
 * are created by DDT instead of using constructors here.
 */
public class BalanceTest extends TestCase implements DataDrivenTestCase {
  TestCaseData testCaseData;

  /**
   * execute all transfers before a certain date,
   * then check for expected balances.
   * Loops over all expected balances.
   */
  public void testBalances() throws Exception {
    AmountAtDate[] myBalances =
        (AmountAtDate[]) testCaseData.createAndFillBean("myBalances");
    AmountAtDate[] transfers =
        (AmountAtDate[]) testCaseData.createAndFillBean("transfers");
```

[2] DDT-annotations start with the tilde sign

```
    for (int i = 0; i < myBalances.length; i++) {
      Account myAccount = (Account) testCaseData.createAndFillBean("myAccount");
      Account momsAccount = (Account) testCaseData.createAndFillBean("momsAccount");
      AmountAtDate myBalance = myBalances[i];

      executeAllBefore(transfers, momsAccount, myAccount, myBalance.getDate());
      assertEquals("balance of my account wrong on " + myBalance.getDate(),
          myBalance.getAmount(), myAccount.getBalance());
    }
    assertTrue(myBalances.length > 0);
  }

  private void executeAllBefore(AmountAtDate[] transfers,
                                Account source, Account target,
                                Date beforeDate) {
    Arrays.sort(transfers); // by date

    for (int i = 0; i < transfers.length; i++) {
      AmountAtDate transfer = transfers[i];
      if (transfer.getDate().before(beforeDate)) {
        if (transfer.getAmount().signum() >= 0) {
          target.deposit(source.withdraw(transfer.getAmount()));
        } else {
          source.deposit(target.withdraw(transfer.getAmount().negate()));
        }
      } else {
        break;
      }
    }
  }

  public static Test suite() {
    ExcelObjectMatrixFactory objectMatrixFactory = new ExcelObjectMatrixFactory(
        "tutorial2.xls",
        new String[]{"balanceTests"}
    );

    JUnitTestSuiteBuilder jUnitTestSuiteBuilder =
        new JUnitTestSuiteBuilder(objectMatrixFactory, BalanceTest.class);
    return jUnitTestSuiteBuilder.buildSuite();
  }

  public String getName() {
    return testCaseData.getTestCaseName() + "--" + super.getName();
  }

  public TestCaseData getTestCaseData() {
    return testCaseData;
  }

  public void setTestCaseData(TestCaseData testCaseData) {
    this.testCaseData = testCaseData;
  }
}
```

6 Where to Go from Here

Now that you got things started, you will probably want to have a look at the DDT Reference Manual which offers an architectural overview and plenty of detailed information about how to get the most out DDT.

You might also want to check `http://ddt.senacor.com` for recent news about this project.