

# Data-Driven Testing with FitNesse

*Data-driven testing is testing where data contained in an input test data file control the flow and actions performed by the automated test script.  
[Just Enough Software Test Automation, page 106]*

## Introduction

This article demonstrates how the open source Wiki FitNesse can be used to implement data-driven testing, making tests at once more efficient and easier to maintain. The examples are in Java, but FitNesse supports other languages.

I assume that you have FitNesse up and running (very easy).

## What is Data-Driven Testing?

Test automation needs data. The easiest way to get this data is to execute an existing application and to record this with a **capture and replay tool** (e.g. <http://jakarta.apache.org/jmeter>). This has several problems, though, some of which are:

- the data needs to be reusable (e.g. a database reset needs to be executed)
- when the application changes all tests might need to be rerecorded
- we only test things we already saw to be working
- adding new test cases forces us to execute (and record) them by hand
- we need a working application to even start (no TestFirst here)

Data-driven testing separates the test data from the test itself. This makes both the test and the data more flexible and reusable and certainly much more easy to maintain. In its most primitive form one might just have some data in an Excel file, save it as a CSV file and read that into a normal JUnit test. Many tests need many files, though, and maintenance will become hard again. Tools like FitNesse (and of course WinRunner) make this easier.

The test data is defined in tables and organized in a hierarchical system in FitNesse (you don't have to organize them). Tests can be executed automatically (using Ant) or manually against all or selected tables.

## An Example

This example will demonstrate how TestFirst can improve quality and how easy data-driven testing with FitNesse can make this. I will start with the traditional approach of presenting the specification, the code and then the (data-driven) tests. These tests will show several problems and I will then restart using a TestFirst approach.

### *TestLast Approach*

We are working on a banking application and one module is to calculate compound interest. The specification states:

*The Compound Interest Equation*

$$P = C(1 + r)^t$$

*where*

*P = future value*

*C = initial capital*

*r = interest rate*

*t = number of years invested*

The specification seems quite clear (it is a mathematical function, after all), so I start hacking away at once. Here is the Java code:

```
public class CompoundInterestCalculator
```

```

{
    public static double calculateCompoundInterest(double capital,
        double interestRate, int years)
    {
        double value = capital * (1 + interestRate);
        value = Math.pow(value, years);
        return value;
    }
}

```

After doing a quick manual test (taking numbers we can calculate in our head, like 10% and 1 year) we can check this code in and go on to our next task, hoping that the test team will not bother us, can't we?

As this article tries to demonstrate some points, the code does of course have a serious bug. So, let's write a JUnit test to find it:

```

package testbuch;

import junit.framework.TestCase;

public class CompoundInterestCalculatorTest extends TestCase
{
    public void testCalculateCompoundInterest()
    {
        double result = CompoundInterestCalculator.calculateCompoundInterest(
            100, 0.1, 1);
        assertEquals(110.0, result, 0.01);
        result = CompoundInterestCalculator.calculateCompoundInterest(
            100, 0.1, 10);
        assertEquals(259.37, result, 0.01);
    }
}

```

In case you don't know what the third value in the assert commands means, it defines how exact the comparison has to be (the double values will not be exact because of rounding).

The first test works just fine, but the second breaks:

```
junit.framework.AssertionFailedError: expected:<259.37> but
was:<2.5937424601000034E20> at junit.framework.Assert.fail(Assert.java:47)
```

Wow, that is quite a large number, and after just ten years of saving. The error lies in the calculation. We split it into two rows and therefore the multiplication with the initial capital was executed first and only then did we execute Math.pow() call.

Here is the corrected version:

```

    public static double calculateCompoundInterest(double capital,
        double interestRate, int years)
    {
        double value = (1 + interestRate);
        value = Math.pow(value, years);
        value = capital * value;
        return value;
    }
}

```

Okay, you might say, so we found a bug by using JUnit. Still no big deal!

And right you might be. But I am a tester and I know that there are defects and there are defects. A tester needs to test for two things:

- does the software work correctly
- does the software do the right thing

The first point seems to be alright, but for the second I have to say: I do not know!

### ***TestFirst Approach***

To understand whether the code above does the right thing, we would need to know how it is used. It does return doubles for once. Therefore:

```
System.out.println(CompoundInterestCalculator.  
    calculateCompoundInterest(100, 0.1, 1));
```

110.00000000000001

This might be okay if it is used only for further calculations, but is definitely not what we want to show on a GUI. And what about entering negative numbers? We could use them to calculate backwards (how much would I need to achieve some capital), but most likely negative values are to be rejected. Therefore we need to clarify our requirements. And we need more tests. In this case it is very easy to calculate new test data but in real live test data is usually hard to get. And even now we see how the test data is included in our test code. I once sad three days with a requirements engineering consultant over our requirements and learned how hard it can be to express oneself clearly and without double meanings. Being lazy, we take the easy way out and only define some examples. We do this in Fitnesse, of course.

initialCapital	interestRate	numberYears	futureValue
0.00	0.00	0	0.00
0.00	0.10	1	0.00
100.00	0.10	0	100.00
100.00	0.10	1	110.00
100.00	0.10	10	259.37
100.00	0.10	-1	error

**Figure 0.1: Example Values**

Taking the values from the figure we do know much more and can adapt our method to throw an exception if a negative value is entered for the number of years. Further more, we can refactor our test to letting FitNesse call the test code:

```
package testbuch;  
import fit.ColumnFixture;  
  
public class CompoundInterestCalculatorFixture extends ColumnFixture  
{  
    public double initialCapital, interestRate;  
    public int numberYears;  
  
    public double futureValue()  
    {  
        int value = (int) (CompoundInterestCalculator.calculateCompoundInterest(  
            initialCapital, interestRate, numberYears) * 100);  
        return value / 100.0;  
    }  
}
```

This new test is very simple and the good news is that it stays the way it is, not matter how much test data we have. Adding another test now just means to add another row to the table, the code itself is not touched!

# CompoundInterest

## TEST RESULTS

**Assertions:** 6 right, 0 wrong, 0 ignored, 0 exceptions

*classpath: C:\x\workspace\Tester\*

*classpath: C:\x\workspace\Tester\lib\\**

testbuch.CompoundInterestCalculatorFixture			
initialCapital	interestRate	numberYears	futureValue?
0.00	0.00	0	0.00
0.00	0.10	1	0.00
100.00	0.10	0	100.00
100.00	0.10	1	110.00
100.00	0.10	10	259.37
100.00	0.10	-1	error

**Figure 0.2**

And the best thing is that we can let the customer or the business analyst define these tests. Now it becomes their responsibility to define exactly what the software is to do. No more “I wanted something else” as they have to define their wishes with mathematical precision now!

## Conclusion

This article demonstrated how to use a data-driven test to improve Java code by providing test cases in a table. These tests define the requirement much more precise than words can do. The example used in this article refactored a “normal” JUnit test to a FitNesse driven test, showing how much more easy testing can get.