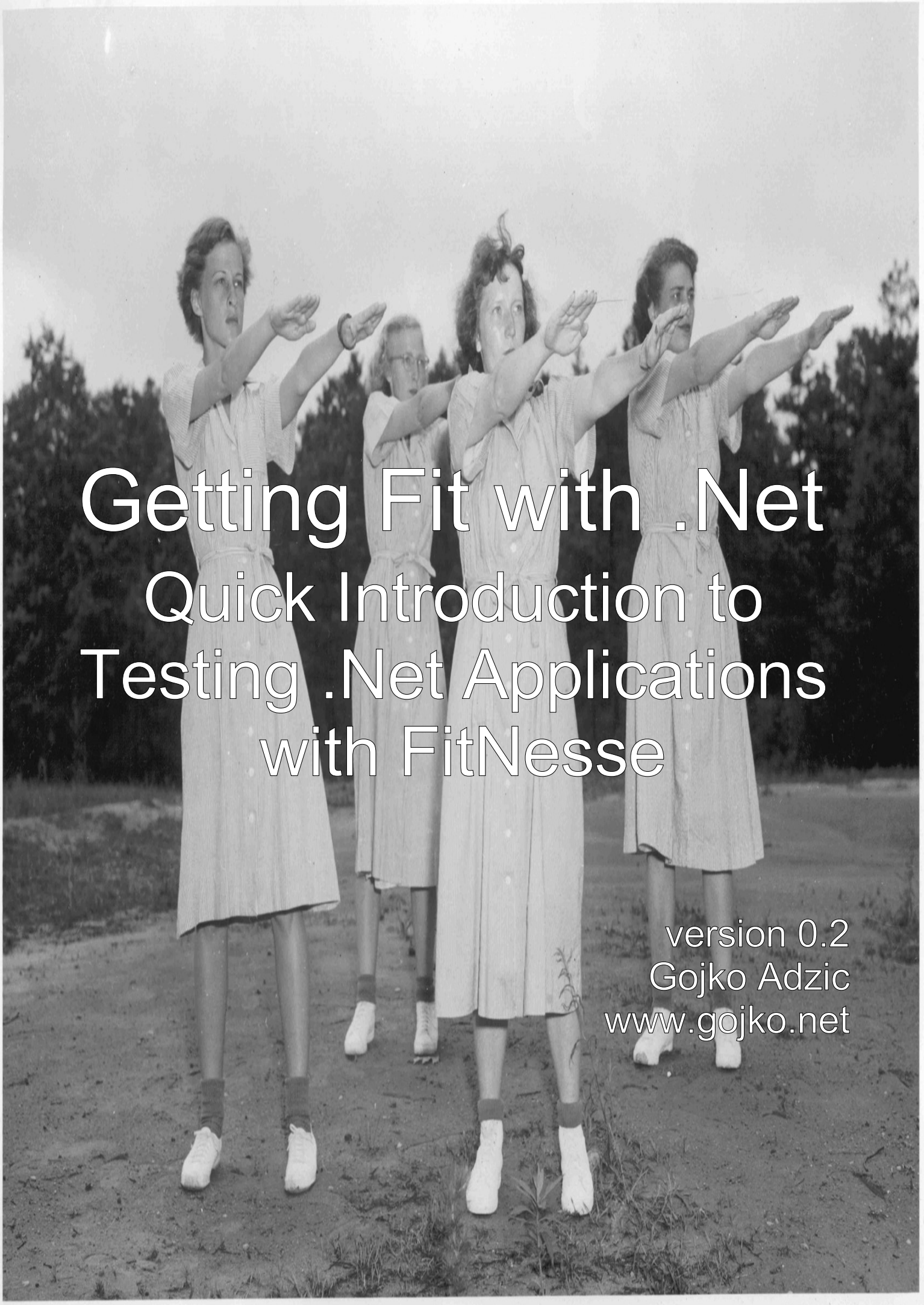


Getting Fit with .Net

Quick Introduction to Testing .Net Applications with FitNesse

version 0.2

Gojko Adzic
www.gojko.net

Table of Contents

Redistributing

Version

0.2 – 16. February 2007, covering Fitnesse.Net 1.1 (build 20070128)

If you suspect that this document is outdated, check if there is a more recent version on www.gojko.net.

Version History

0.2 – 16. February 2007, covering Fitnesse.Net 1.1 (build 20070128)

- minor corrections and updates for Fitnesse.Net 1.1
- new chapter Writing better test scripts, covering DoFixture, working with arrays and business objects

0.1 – 26. December 2006, covering build 20060530 of Fitnesse.Net

Corrections, comments, updates and source code

If you are interested in corrections, updates and new versions of this guide, I suggest visiting http://gojko.net from time to time. You can also subscribe to the RSS feed for updates on http://gojko.net/feed and keep up to date with recent developments. Or, alternatively, you can drop me an e-mail and I'll notify you about important updates – my contact details are on the last page.

The source code for the examples and Wiki pages from this guide can be downloaded from http://gojko.net/FitNesse – please feel free to leave any comments about this guide on that page, or notify me about errors/corrections. Your help in building this guide will be greatly appreciated.

Links

All links are clickable (if you are reading this on your screen - on the other hand, if you printed this document and links still work, please get in touch - I'd like to hear how you did it).

Cover photo

The cover photo is a royalty-free picture by the Horton group: http://www.sxc.hu/profile/hortongrou.

Introduction

FitNesse is a great Web-based collaboration tool for software testing, which can really help to test-drive the code and build a framework for holding the project together during big changes and re-factoring. It makes writing and running automated tests easy and allows test-driven software teams to share knowledge and expectations.

Under the hub, FitNesse runs FIT (Framework for Integrated Testing). Both FitNesse and FIT are open-source tools, and together they are very popular as a testing framework in the Java community. Although FitNesse supports testing .Net code, some things don't quite work out of the box or do not follow official on-line documentation. However, the integration is stable, and I guarantee that the effort required to start using FitNesse is worth it.

This is a guide to help you get started with FitNesse. I will not try to make a case for automated tests or test-driven development here, nor explain all the benefits of FitNesse. Here are just a few advantages of using FIT/FitNesse combination for testing:

- It is easy to write complex tests
- Tests are easy to read and understand
- FitNesse promotes collaboration between team members (and customers)
- Test-specific code is very thin, and it looks much more like an integration layer then typical testing code

I will refer to FIT/FitNesse combination in this document simply as FitNesse – although it might not be 100% correct, it will simplify the story. This is a beginners guide and you will be working directly with FitNesse, and I do not want to confuse you from the start.

Why are we here?

You are here, I presume, because you are interested in automated testing, especially testing .Net code. I will make a few more assumptions and say that you know at least the basics of .Net development, expect to get some benefits from code test coverage, and have a project that you want to cover with tests. Also, you want to know what all the buzz with FitNesse is about.

I am here because I was in that same position recently, and spent a lot of time experimenting with FitNesse. I banged my head against the brick wall quite a few times, especially due the lack of good documentation about .Net FitNesse integration. Although the journey was not without problems, the results are really great – and I want to help you to cross those few first bridges easier.

This is by no means a comprehensive text-book on everything you can do with these tools. Consider it more as a short tourist guide to get you on your way to the wonderful world of FitNesse[1]. It covers:

- Setting up a FitNesse server for testing .Net code
- Writing basic tests, performing common tasks
- Saving time and effort with specialised test types
- Tips and tricks for writing better tests and making test pages easier to read
- Managing content with FitNesse
- Organising tests into test suites

IMPORTANT: FitNesse .Net integration has it's specifics, some things are different from the online documentation (which deals with the Java version) and some things just don't work out of the box. When I explain such a difference, I'll mark it like this block.

1 "Welcome to the wonderful world of FitNesse" is the default home page title on a FitNesse site built from scratch.

Getting started

First, get a copy of FitNesse and set it up. Download the latest version from http://www.FitNesse.org/FitNesse.Download (look for the ZIP file with binaries from the latest release). Unpack it somewhere on your disk, and make sure that you have Java 5 or 6 installed. (Yes, this is a .Net testing guide, but FitNesse requires Java to run. If you do not have a working Java VM on your system, get one from http://java.sun.com).

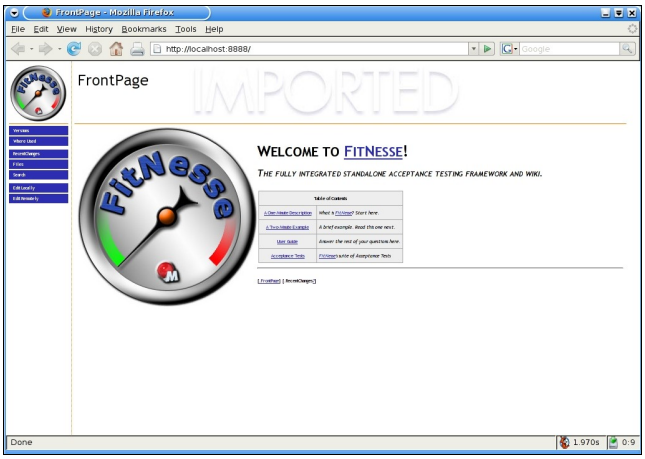
IMPORTANT: The original distribution already contains a *dotnet* folder with .Net integration – however be aware that it works only for .Net1 (at the time when I wrote this). To test .Net2 code, you have to download .Net2 binaries separately. The binaries for release 1.1 are a bit hard to find, as the link on http://www.FitNesse.org/FitNesse.DotNet has still not been updated to the correct URL. (when I wrote this, the link for Framework 2.0 Binaries lead to release 1.0). You can download the binaries of release 1.1 from http://sourceforge.net/project/showfiles.php?group_id=167811 and the source is on https://svn.sourceforge.net/svnroot/fitnessedotnet/tags/20070128/). Get *FitServer.exe*, *TestRunner.exe* and those DLLs and put them in a new folder – I suggest opening *dotnet2* alongside *dotnet* and storing the files there. I will use that path in the examples here, if you put them elsewhere, remember to change the path.

Once you have downloaded everything, just start *run.bat*. You should see something like this:

```
FitNesse (20060719) Started...
port:                  80
root page:             FitNesse.wiki.FileSystemPage at ./FitNesseRoot
logger:                none
authenticator:         FitNesse.authentication.PromiscuousAuthenticator
html page factory: FitNesse.html.HtmlPageFactory
page version expiration set to 14 days.
```

IMPORTANT: FitNesse works as a Web application with it's own web server, and will try to take port 80 by default. If you are like me, that port is already taken, so open *run.bat* in any editor and add *-p 8888* to the command. (You can replace 8888 with some other free port on your system. I will use 8888 in the examples - so if you use another one, remember to enter the correct port when trying out the examples).

Open http://localhost:8888/ and you should see the welcome page.



Congratulations, FitNesse is up and running (to shut it down later, just kill the window or press Ctrl+C). Now, let's write some tests.

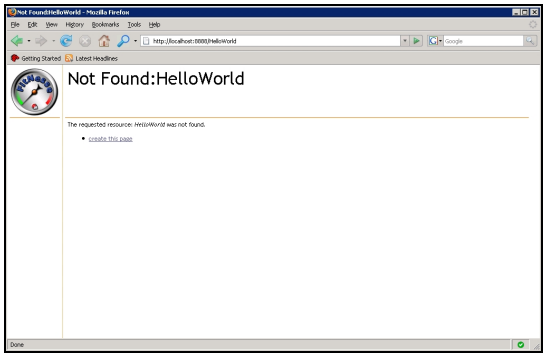
Hello, World

FIT reads HTML files, looks for tables, and uses data in the tables to execute tests and compare results to expectations. FitNesse is there to help with building those pages, as a collaborative Wiki[2] site with helpful mark-up shortcuts.

Let's write a simple class that joins strings and test it. Create a new project in Visual Studio (or whatever you are using to write .Net applications), and add a reference to *fit.dll* from *dotnet2* subfolder of FitNesse. Add the following class to the project:

```
namespace NetFit {
      public class ConcatenateStrings: fit.ColumnFixture{
            public string firstString;
            public string secondString;
            public string Concatenate()
            {
                  return firstString + " " + secondString;
            }
      }
}
```

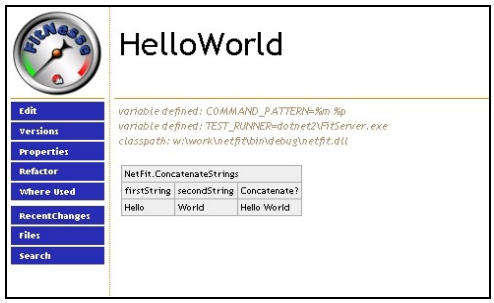
Now go to http://localhost:8888/HelloWorld - you should see a screen similar to the one on the right – telling you that there is no *HelloWorld* page (yet) and offering to create a new page. Click on the link, and FitNesse will open the page editor – a big text box above several buttons.



Just paste the following code into the text box and click on *Save (*replace *examples\netfit.dll* with the full path to the your project's DLL):

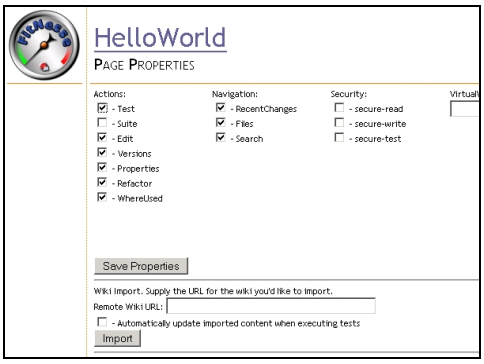
```
!define COMMAND_PATTERN {%m %p}
!define TEST_RUNNER {dotnet2\FitServer.exe}
!path examples\netfit.dll

!|NetFit.ConcatenateStrings|
|firstString|secondString|Concatenate?|
|Hello|World|Hello World|
```

FitNesse will now create a new page and display it:

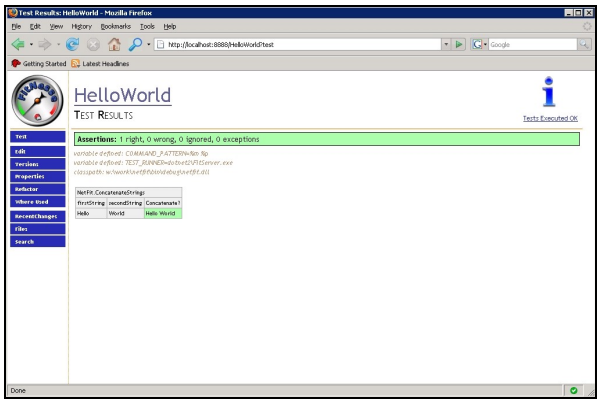


2 Wikis are Web systems for publishing information, typically intended for collaborative use, allowing people to easily create and edit pages using a simple mark-up syntax. Wikipedia is a popular example, you must have seen at least one Wiki by now, so working with FitNesse should not feel strange.

Next, you have to tell FitNesse that this is a test page – click on the properties button on the left, check the *Test* check-box and then click on *Save Properties*. Page properties basically define what the user can do with that page (controls which buttons will be offered on the left-hand menu).



When the page is loaded again, you will notice a new button on the left – *Test*. Click on it and FitNesse will run the test.



Ok – that was your first FitNesse test in .Net, and it passed. Hurrah! Now let's go back and see what really happened.

The class has two public string fields and a public method – FitNesse can set and get values of public fields and properties, and it can call public parameterless methods (so the fields are actually a way to pass parameters to tests). The class extends *fit.ColumnFixture*, which tells FitNesse how to execute tests – just leave it like that for now, I'll explain it a bit later.

Now look the page code. First three lines tell FitNesse to use the test runner in *dotnet2* folder, with your DLL.

```
!define COMMAND_PATTERN {%m %p}
!define TEST_RUNNER {dotnet2\FitServer.exe}
!path examples\netfit.dll
```

The next three lines define the table:

```
!|NetFit.ConcatenateStrings|
|firstString|secondString|Concatenate?|
|Hello|World|Hello World|
```

Table rows are created by using the pipe symbol (|) to separate cells. The first line (table header) contains the test class name. FitNesse automatically converts CamelCase[3] names into links, and we don't want a link in this case, so the exclamation mark (!) before the first cell in the table header tells FitNesse not to play around with our table data.

The second row contains column headers, listing two field names and the method name. Notice the question mark on the end of the method name. If the header does not end with a question mark, the column defines input parameters. If the header ends with a question mark, the column defines assertions. This particular table tells FitNesse to set values of *firstString* and *secondString* and then check the result of *Concatenate()*. Rows after the headers define combinations of input parameters and expected values. This table has only one row with values, only one assertion is executed.

3 See http://en.wikipedia.org/wiki/CamelCase

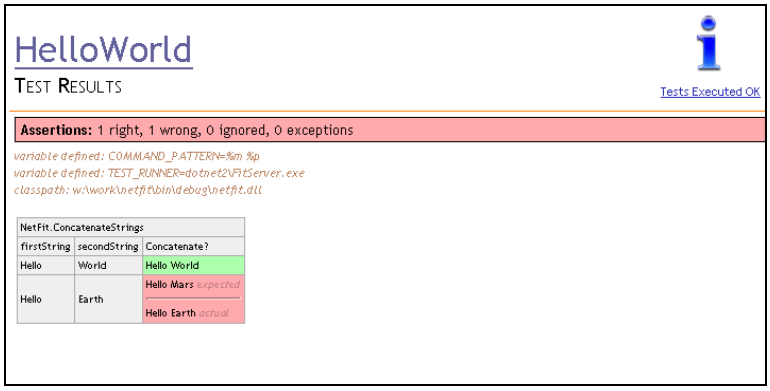
So, in fact, the executed test is equivalent to the following xUnit-like code:

```
ConcatenateStrings c=new  ConcatenateStrings();
c.firstString=”Hello”;
c.secondString=”World”;
AssertEqual(c.Concatenate(),”Hello World”);
```

A single table can contain many tests – edit the page again and add another row to the table:

```
!|NetFit.ConcatenateStrings|
|firstString|secondString|Concatenate?|
|Hello|World|Hello World|
|Hello|Earth|Hello Mars|
```

This time, the tests failed. You can see the expected and the actual result in the second row, clearly marked red (as is the page header). The symbol on the right still happily tells you that the tests executed OK (which is a bit strange, as they failed, but who am I to complain).



Now, go and play!

Take a few minutes and experiment – build tests for some other classes, try reading and writing properties and fields (just add a question mark after the name of the field or property). To assert that an exception will be thrown, write *error* in the cell. There are a few other special keywords which you can use in tables, including *null* and *blank* (equal to an empty string). Take a note that comparing with an empty cell does nothing – FitNesse will just print the actual value of the appropriate field. If you want to check whether some value is actually an empty string, write *blank* into the cell.

You can put more than one table on a single page, in order to test different classes or combinations of parameters, but remember to split them with at least one blank line, so that FitNesse does not join their rows together and build a single table.

IMPORTANT: if you spot a question mark after the class or field name, and you did not put it there, check if it looks like a link (underlined). If it had CamelCase structure, FitNesse built a “missing page” link from the class/field name. To prevent FitNesse from converting CamelCase names to links, put an exclamation mark (!) before the first pipe in the table.

You can also correct the previous table either by changing the expectation to *Hello Earth* or the second parameter to *Mars* (or alternatively, change the class code to return *Hello Mars*).

IMPORTANT: Don't be alarmed if you get a *NullReferenceException* during tests. FitNesse can throw that exception if it does not find an appropriate field or method to call – so check if you misspelled a column header.

Writing tests

FitNesse determines how to execute tests based on the *Fixture* class which they inherit. *ColumnFixture*, used in the previous example, is executed by setting and reading fields and calling methods specified in column headers, depending on whether they end with a question mark. By now, you should be familiar with it. *ColumnFixture* is very simple, but quite powerful – think of it as a FitNesse equivalent of the Swiss army knife.

Each table is a test unit, so the context (field and property values in the *Fixture*) is preserved while a single table is being processed. This can be applied to test calculations which produce complex results with multiple properties. A typical example would be to set the input parameters, execute a method which populates several properties, and then check those properties.

Here is an example – the *TextMachine* counts words and characters in a string:

```
class TextMachine{
      public int words;
      public int chars;
      public TextMachine(String text){
            chars=text.Length;
            words=text.Split(", \r\n\t".ToCharArray(),
                  StringSplitOptions.RemoveEmptyEntries).Length;
      }
}
```

Let's build a simple FitNesse test for it:

```
namespace NetFit {
      public class AnalyseText:fit.ColumnFixture {
            private TextMachine tm;
            public int words { get { return tm.words; } }
            public int chars { get { return tm.chars; } }
            public string input;
            public bool calculate() {
                  tm = new TextMachine(input);
                  return true;
            }
      }
}
```

And here is the table (remember to copy .Net2 environment lines if you put this into a new page):

```
!|NetFit.AnalyseText|
|input|calculate?|words?|chars?|
|The quick brown fox jumps over the lazy dog|true|9|43|
|So Long, and Thanks for All the Fish|true|8|36|
```

The table should look like this on the screen:

NetFit.AnalyseText			
input	calculate?	words?	chars?
The quick brown fox jumps over the lazy dog	true	9	43
So Long, and Thanks for All the Fish	true	8	36

In this case, the column with *calculate* method is not actually used to check any assertions, just to execute the method call. It can, however, be used to check for processing exceptions – so it is a good practice to return some value (i.e. true) on the end and then check for that in the table.

Tables are always processed top-down and left-to-right, so it's safe to check for *words* and *chars* after the *calculate?* column.

This technique is very useful for testing persistent data – pass the primary key value, execute a method to retrieve a row of data, then check individual properties.

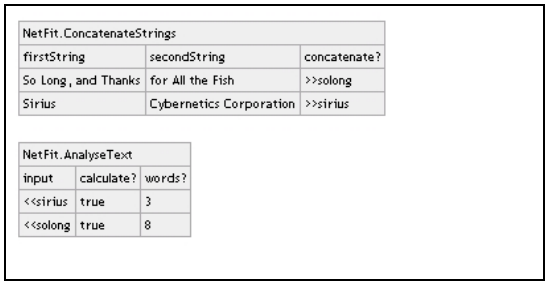
Similar technique can be used to store data in the database, in order to set up the test environment.

Writing scripts

A single page can contain multiple tables, so you can quickly build complex test scripts by combining tables. Tables are always executed top-down, and it is save to presume that a previous table has already been executed – for example, you can use one table to prepare data, and then execute various tests on that database with several other tables. To pass data between tests, use local variables – called 'symbols' in FitNesse.

IMPORTANT: .Net implementation of symbols differs significantly from the on-line documentation (which explains the Java implementation). Symbols are defined using a special syntax in cells instead of column headers.

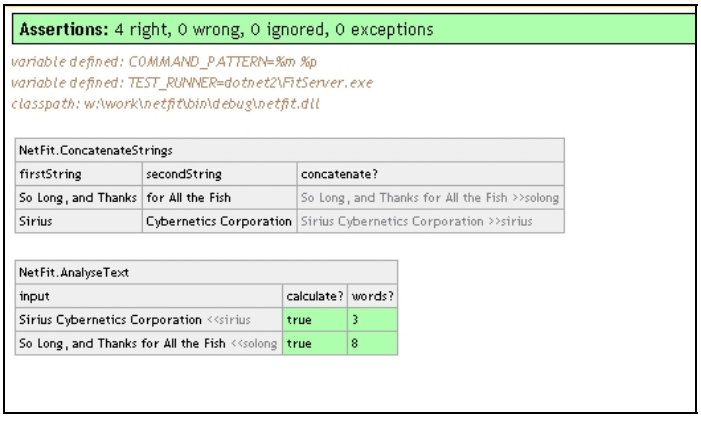
To store a value into a symbol, put >> and the symbol name into the cell. To read the symbol value, put << before the symbol name. The picture on the right shows an example, a combination of previous two tests.



NetFit.ConcatenateStrings		
firstString	secondString	concatenate?
So Long, and Thanks	for All the Fish	>>solong
Sirius	Cybernetics Corporation	>>sirius

NetFit.AnalyseText		
input	calculate?	words?
<<sirius	true	3
<<solong	true	8

When the test is executed, current symbol values are printed next to symbol operations, so you can see exactly what is going on:



Assertions: 4 right, 0 wrong, 0 ignored, 0 exceptions

variable defined: COMMAND_PATTERN=%m %p
variable defined: TEST_RUNNER=dotnet2\FitServer.exe
classpath: w:\work\netfit\bin\debug\netfit.dll

NetFit.ConcatenateStrings		
firstString	secondString	concatenate?
So Long, and Thanks	for All the Fish	So Long, and Thanks for All the Fish >>solong
Sirius	Cybernetics Corporation	Sirius Cybernetics Corporation >>sirius

NetFit.AnalyseText		
input	calculate?	words?
Sirius Cybernetics Corporation <<sirius	true	3
So Long, and Thanks for All the Fish <<solong	true	8

You can also use symbol values in comparisons (to check if a calculated value is equal to the current symbol value – just put <<*symbol* in the column body).

Symbols are very useful when testing against a database – you can use one column fixture to create a new customer record and return the primary key, then load the primary key into a symbol and continue using that symbol as the customer identifier in other tests.

Saving time and effort with specialised test types

Although column fixtures are quite versatile, other *Fixture* types are better suited for specific tasks. *RowFixture* is excellent for testing referential information, data coming out of the database and methods that produce arrays of objects. *ActionFixture* is great for test scripts which do not have a repetitive structure.

Batch comparisons: *RowFixture*

Using *RowFixture* instead of *ColumnFixture* for batch data comparisons can save a lot of effort, since both the fixture code and the test tables are more compact and easier to write - especially if you already have an object-relational mapping layer. With *RowFixture*, you do not have to re-declare properties in the *Fixture* class (which you would have to do if *ColumnFixture* was used), and you do not have pollute test tables with columns that just read the data.

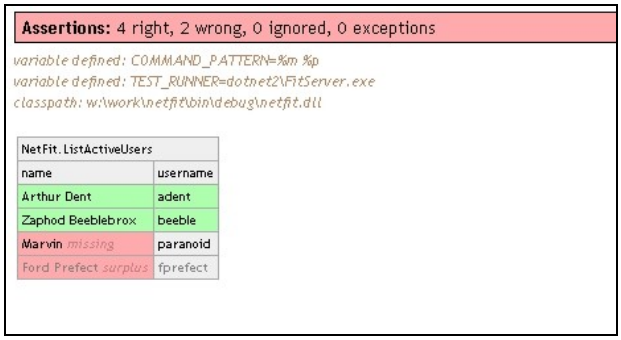
RowFixture is an abstract class with two methods – *GetTargetClass* and *Query*. Instead of specifying your own properties and methods, like with *ColumnFixture*, you just need to override those two methods to write a *RowFixture* test. *GetTargetClass* specifies the type of objects in the array, and *Query* retrieves the array of objects. FitNesse will execute the *Query* method and then compare results with the table, matching rows with array elements. The order of elements is not important, but the test will fail if the array does not contain all specified rows, if it contains more objects than rows, and if any of the rows cannot be matched. Here is an example:

```
namespace NetFit {
      public class User{
            public int id;
            public String name;
            public String username;
            private User(int id, String name, String username){
                  this.id = id;
                  this.name = name;
                  this.username = username;
            }
            private static User[] hhgg=new User[]{
                  new User(1,"Arthur Dent","adent"),
                  new User(2,"Ford Prefect","fprefect"),
                  new User(3,"Zaphod Beeblebrox","beeble")
            };
            public static User[] GetList(){return hhgg;}
      }
      public class ListActiveUsers:fit.RowFixture {
            public override Type GetTargetClass(){
                  return typeof(User);
            }
            public override object[] Query() {
                  return User.GetList();
            }
      }
}
```

The following test table has one object more, and one of the actual objects is missing:

```
!|NetFit.ListActiveUsers|
|name|username|
|Arthur Dent|adent|
|Zaphod Beeblebrox|beeble|
|Marvin|paranoid|
```

When this test is run, FitNesse will match first two rows, report that Marvin is missing, and that Ford Prefect was seen, but not expected.



You can use symbols in the row tables – both for storing values and for comparing them.

IMPORTANT: Put a question mark after the column name to specify that a property should not be used for row matching, just for comparing expected values to matched objects. If a row fixture tests database values, all columns except the primary key should typically have a question mark on the end. Also, any columns that are used to store values into symbols will have to be marked like this.

There are no explicit parameters to the *Query* method, but you can pass parameters to row fixtures using a different technique – fixture arguments. Pass arguments by adding cells to the first row of the table, after the class name. Arguments can be accessed in the *Fixture* code through the protected string array *Args*. Let's modify the example and add a parameter to the query. First, add a parametrised *Get* method and another user table to the *User* class:

```
private static User[] jedi = new User[]{
      new User(1,"Obi-Wan Kenobi","kenobi"),
      new User(2,"Qui-Gon Jinn","quigonn"),
      new User(3,"Mace Windu","windu")
};
public static User[] GetList(String list){
      if ("jedi".Equals(list)) return jedi;
      return hhgg;
}
```

Next, create a new row fixture that uses parameters:

```
public class ListActiveUsersWithParam : fit.RowFixture {
      public override Type GetTargetClass() {
            return typeof(User);
      }
      public override object[] Query() {
            return User.GetList(Args[0]);
      }
}
```

Then create test tables, with parameters in table headers:

```
!|NetFit.ListActiveUsersWithParam|hhgg|
|name|username|
|Arthur Dent|adent|
|Zaphod Beeblebrox|beeble|
|Ford Prefect|fprefect|

!|NetFit.ListActiveUsersWithParam|jedi|
|name|username|
|Obi-Wan Kenobi|kenobi|
|Mace Windu|windu|
|Qui-Gon Jinn|quigonn|
```

Parameters are not limited to row fixtures – you can use them in other fixture types in the same way.

IMPORTANT: Symbols cannot currently be used as *Fixture* parameters – but you can read any symbol value in fixture code with static method *Fixture.Recall(symbolName)*. See the part about additional shortcuts on the page 28 of this document to use symbols as fixture parameters directly.

Simple scripts: *ActionFixture*

Column fixtures allow you to define parameter values and execute methods, but they require you to set the structure for each test by defining column headers. When the test script does not have a repetitive structure, you will end up creating a new table (and possibly a new fixture class) for each test case step. *ActionFixture* is a much better solution for that – it allows you to execute a relatively free-form test script in a single table, with a single test fixture class.

Here is an example: after the customer fills in user details and submits the registration form, the system has to execute a credit check before activating his account.

This is the test script that we want to try:

- customer enters his details and submits the registration form
- verify that the customer record is stored correctly
- verify that customer is inactive
- verify that there is a pending credit check
- approve account
- verify that credit check is finished
- verify that the customer is active.

We will use a simple in-memory customer "database":

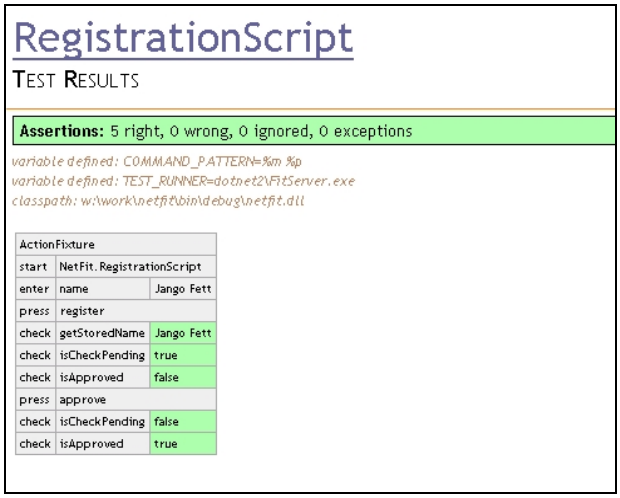
```
class Customer{
	public string name;
	public bool approved;
	public Customer(string name){
		this.name = name;
	}
}

class CustomerDatabase{
	private List<Customer> customers = new List<Customer>();
	private Queue<Customer> creditChecks = new Queue<Customer>();
	public int Register(Customer c){
		int nextPos = customers.Count;
		customers.Add(c);
		creditChecks.Enqueue(c);
		return nextPos;
	}
	public bool IsCheckPending(int customerId){
		return creditChecks.Contains(customers[customerId]);
	}
	public void Approve(){
		creditChecks.Dequeue().approved = true;
	}
	public Customer GetCustomer(int customerId) {
			return customers[customerId];
	}
}
```

Here is the FitNesse fixture for the test:

```
public class RegistrationScript:fit.Fixture {
      public string name;
      private CustomerDatabase cd = new CustomerDatabase();
      private int customerId;
      public void Register(){
            customerId=cd.Register(new Customer(name));
      }
      public bool IsCheckPending(){
            return cd.IsCheckPending(customerId);
      }
      public bool IsApproved(){
            return cd.GetCustomer(customerId).approved;
      }
      public string GetStoredName(){
            return cd.GetCustomer(customerId).name;
      }
      public void Approve(){
            cd.Approve();
      }
}
```

And finally, here is the test table:

```
!|ActionFixture|
|start|NetFit.RegistrationScript|
|enter|name|Jango Fett|
|press|register|
|check|getStoredName|Jango Fett|
|check|isCheckPending|true|
|check|isApproved|false|
|press|approve|
|check|isCheckPending|false|
|check|isApproved|true|
```



The executed test should look like the picture on the right. Notice that the fixture class does not extend *ActionFixture*, but *fit.Fixture*, and that the table uses *fit.ActionFixture* as the test class, not our *RegistrationScript*. Action fixtures script other classes, and the scripted class appears in the second row, after the *start* keyword.

Script commands use metaphors from user-interfaces: *enter*, *press* and *check*. *Enter* modifies a field or property value – like a column without the question mark in the *ColumnFixture*. *Check* reads the value of a property or a field, or executes a method and checks the result – it is like a column with a question mark in the *ColumnFixture*. *Press* is the equivalent of columns in *ColumnFixture* that are just used to execute methods.

Notice how compact this script is – both the code and the table. To perform the same check with *ColumnFixture*, we would have to write a separate table for each step, and pass the customer identifier using symbols.

Using ready-made classes to set symbol values: *StringFixture* and others

There are many other useful *Fixture* classes, both in the basic FIT framework and the popular *FitLibrary* extension, but I will keep this introduction short and let you discover those utilities yourself. (I'll provide a few pointers where to look on the end of this document.) However, I'll just mention another very useful group of classes from an internal test suite. They are: *StringFixture*, *IntFixture*, *DoubleFixture*, *LongFixture*, *BoolFixture*, *FloatFixture* and *DecimalFixture*. All these classes have a field called *Field* which can be used to set and test values of the appropriate type. They are great as utilities when you want to set symbol values. Here is an example:

```
|StringFixture|
|field|field?|
|Vogon Constructor Fleet|>>who|
```

Working with data-transfer objects

FitNesse requires all test fixtures to extend the *Fixture* class, which puts you in a tough spot if you want to connect data transfer objects with table values, because a test fixture cannot easily inherit the structure from a data-transfer class. You have already seen that *RowFixtures* can return arrays of objects, which is ideal for checking values of data-transfer objects without wrapping them into properties of the fixture class. However, populating data-transfer objects is not that easy. There is a good solution, if the data transfer classes can be extended: a fixture can tell FitNesse to connect test tables to another object – *GetTargetObject* method is used for that. By default, it returns the current fixture object, but you can override it return an embedded data-transfer object. That will connect test tables directly to the data-transfer class.

There is, however, one possible problem with this approach: you have to map all the columns to the embedded object – even the test methods. The best way to do this would be to extend the data-transfer class, add test methods, and then use the extended class as the fixture target. The fixture class then just acts as a shell around the extended data-transfer object, telling FitNesse how to run the tests. But, if the data-transfer class is sealed (cannot be extended), you will have to re-declare all the properties in the fixture class and map them to an embedded object, or copy them manually.

Here is an example: we want to test the *CreditCardValidator* service, which validates credit cards. Instead of passing individual properties of the card, we use *CreditCardDTO* data transfer class. To make things simpler, we will just check if the card number has 16 digits, if the post code has at least five characters and if the owner name has at least six characters.

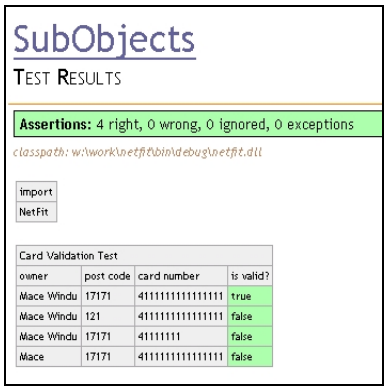
```
public class CardValidator
{
            public static bool IsValid(CreditCardDTO cc) {
                  return (
                  cc.cardNumber.Length==16
                  && cc.postCode.Length>4
                  && cc.owner.Length>5);
            }
}
public class CreditCardDTO
{
      public String cardNumber;
      public String owner;
      public String postCode;
}
```

The test fixture class will be a *ColumnFixture*, but will not declare any of the properties nor assertions itself. Instead, we will extend the data transfer class and add the test method there.

```
public class CardValidationTest: fit.ColumnFixture {
      class ExtendedCreditCardDTO:CreditCardDTO {
            public bool isValid() {
                  return CardValidator.IsValid(this);
            }
      }
      private ExtendedCreditCardDTO card = new ExtendedCreditCardDTO();
      public override object GetTargetObject() {
            return card;
      }

}
```

The test table can now directly access *ExendedCardDTO* methods and properties:

```
!|Card Validation Test|
|owner|post code|card number|is valid?|
|Mace Windu|17171|4111111111111111|true|
|Mace Windu|121|4111111111111111|false|
|Mace Windu|17171|41111111|false|
|Mace|17171|4111111111111111|false|
```



Making test pages easier to read

One of the original goals of FIT and FitNesse was to enable non-technical users to collaborate on writing tests. Several syntax tricks and a bit of smart formatting can make the test pages much more readable – closer to the English language than to C#.

Use names that are easy to read – FitNesse will find the right .Net equivalent

When mapping column values to class names, properties, variables and methods, FitNesse does a case-insensitive search and ignores blanks. So, instead of:

```
|ConcatenateStrings|
|firstString|secondString|Concatenate?|
|Hello|World|Hello World|
```

you can write:

```
|Concatenate Strings|
|first string|second string|concatenate?|
|Hello|World|Hello World|
```

The test will look much better on the screen, it will be easier to read, and splitting the name into several words solves the problem of automatic CamelCase conversion into links.

Import namespaces

Instead of specifying fully qualified names of test classes, you can import a namespace using the *Import* table. Start the table with a single word – *import*, and specify namespaces in following rows, one in each row. Here is the Hello World example again:

```
!|import|
|NetFit|

!|Concatenate Strings|
|first string|second string|concatenate?|
|Hello|World|Hello World|
```

You need to import the namespace only once for the entire page (actually, once for the entire test suite, I'll explain that later).

IMPORTANT: The include directive is, in fact, a special test table, defined in *fit.dll*. Although on-line documentation does not mention any specific pre-conditions for the include directive, if it does not work for you in .Net, add a line containing *!path dotnet2/fit.dll* to the start of the page. This will make sure that basic FIT library is included in the search for fixtures.

Clean up the mess and start with a fresh Wiki

Unless you are going to develop FitNesse, you might want to clear the existing content and start from a fresh web site. Edit *run.bat* and *add -r MyTests* into the command line, then restart FitNesse. This will install a blank Wiki into the *MyTests* folder, and you can then fill it in with your pages.

Configure FitNesse to run .Net tests by default

FitNesse has a special page, */root*, for storing global definitions. You might not see the *Edit* buttons on the left when you browse to that page, but the URL */root?edit* will open the *root* page straight in edit mode. I suggest adding the following few lines into that page to define the .Net runner and load basic .Net DLLs, so FitNesse will run .Net tests by default:

```
!define COMMAND_PATTERN {%m %p}
!define TEST_RUNNER {dotnet2\FitServer.exe}
!define PATH_SEPARATOR {;}
!path dotnet2\*.dll
```

IMPORTANT: When I did this, the browser just waited and waited... the default content seems to hang the server when *root* page is edited. However, if you create a fresh Wiki (which I suggest you do anyway), as explained in *Clean up the mess*, the problem goes away.

Notice that *PATH_SEPARATOR* is also defined, and that just the basic libraries from *dotnet2* folder are loaded – I suggest that you keep project-specific paths in your tests for now, not in the *root* page. When I explain how to organise tests into test suites, I'll give you a better option for defining the path for project-specific DLL paths.

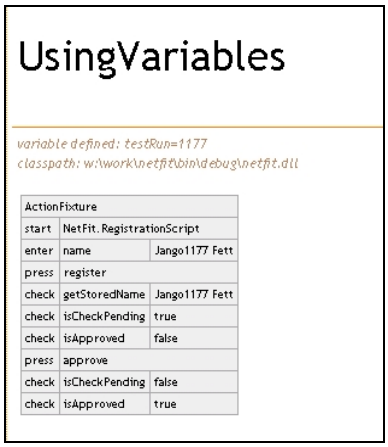
Use variables for string macro replacement

Variables in FitNesse are similar to named macros in programming languages. They are a great tool to quickly parametrise the entire page (or a set of pages) – for example when the tests depend on a file system path or URL of some internal test server, which can change in the future, or when each test run should use a different sequence number. We use them to make sure that tests will not attempt to create customers with duplicate data in consecutive runs, if we do not have time to clean the test database between runs.

Variables can be set with the *!define* directive – you have already used them to set the test runner and DLL path. Variables are read by enclosing the variable name in *${ }*, anywhere in the Wiki source. Here is the customer registration test script again, but with a variable used in the name:

```
!define testRun {1177}

!|ActionFixture|
|start|NetFit.RegistrationScript|
|enter|name|Jango${testRun} Fett|
|press|register|
|check|getStoredName|Jango${testRun} Fett|
|check|isCheckPending|true|
|check|isApproved|false|
|press|approve|
|check|isCheckPending|false|
|check|isApproved|true|
```

The difference between variables and symbols is in the processing time. Variables are processed by FitNesse while building HTML pages – they are not available in the Fixture code, but can be used to build parts of cell content. Symbols, on the other hand, are processed while executing tests – so they are available in the Fixture code, but cannot be used to build parts of cells. Also, the type of variables is irrelevant – they can be used in text or numerical fields, or even as parts of those fields. Type information is important for symbols – create a test that loads 111.1 into a symbol using *StringFixture*, and then compare it to 111.1 using *DecimalFixture*, and the test will fail.



Also, you can clearly see when a symbol is used, but the page will not show that a variable is used in cell content, it will just display a comment where the variable is defined.

Use comments to provide information or disable tables

Any text outside of tables is just ignored – so you can write explanations, include images, provide links to more information or modify those test pages in any way you feel would improve the understanding – this is one of the best features of FitNesse. Easily providing such flexible contextual information is not what most testing frameworks can be proud of.

To quickly disable a test table, and turn it into a comment, just add a row with |*Comment*| on top of it. Any tables beginning with a row containing the single word Comment are ignored during testing.

IMPORTANT: Again, like includes, comments are actually tests – if adding the Comment row on top does not give you the expected results, add *dotnet2/fit.dll* to the path.

There is another type of comment, which can be used in the source of wiki pages – add a hash (#) to the start of any line in the Wiki source to turn it into a comment – it will not display on the Web page.

Load custom cell handlers for simpler comparisons

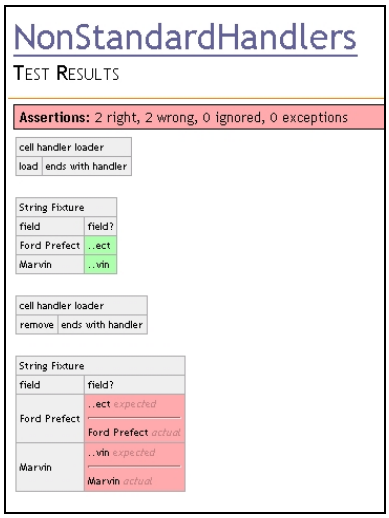
FitNesse uses cell handlers to "understand" what you wrote in table cells. The default cell handler just interprets the data literally, but the symbol retrieval cell handler looks up the symbol value. There are a few non-standard cell handlers which you can use to write comparisons easier. A good example is *EndsWithHandler* – which checks whether a string ends with a given substring. The syntax for this handler is simple – put two dots and then the substring in the cell.

These non-standard handlers have to be loaded on demand because they can obstruct expected behaviour of other functions – when *EndsWithHandler* is loaded, two dots on the beginning of a string have a special meaning – they are no longer interpreted literally.

To load a non-standard handler, just add a table with *CellHandlerLoader* in the first row, and then add rows with two cells – first cell should contain the keyword *load*, and the second cell should contain the class name of the handler. Here is an example:

```
|cell handler loader|
|load|ends with handler|

|String Fixture|
|field|field?|
|Ford Prefect|..ect|
|Marvin|..vin|
```

Other interesting non-standard cell handlers are *IntegralRangeHandler* (checks if a number is in a numeric interval given as *min..max*), *StartsWithHandler* (similar to *EndsWithHandler*, but checks for strings from the left; the syntax is *substring..*), *SubstringHandler* (checks for substrings anywere; can replace *EndsWithHandler* and *StartsWithHandler*, and the allowed syntax is *..substring*, *substring..* or *..substring..*).



As you can see, all these handlers act on two dots, so to avoid confusion you might want to unload them when the test is over. To do that, use *remove* keyword in the Cell Handler Loader, followed by the class name. Two other keywords can be used – *clear* drops all active cell handlers, and *loaddefaults* will load the default handlers again.

The picture on the right shows an example test run – *EndsWithHandler* is loaded first, then we test if the two strings end with correct three characters. After that, *EndsWithHandler* is unloaded, so the repeated test (same table as above) will fail this time, as FitNesse will try to match *..ect* and *..vin* literally.

IMPORTANT: Numeric comparisons do not work as explained in the on-line documentation – although it's written that you can use >10 to check if a value is larger than 10, there is no handler for that expression in the current FitNesse.Net build – see page 28 for a possible solution.

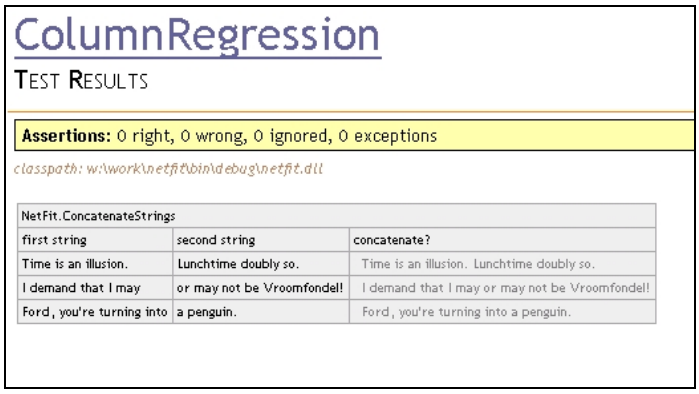
Writing regression tests

If you leave a comparison cell blank, FitNesse will just print the current value of the relevant field, property or method – without actually comparing anything. Along with the feature originally intended for Excel import, this enables you to quickly snapshot current functionality of complex calculations, and build regression tests. Here is a quick example, again using string concatenation.

Build a test table without expected results, just type in the input values and leave the third column blank.

```
!|NetFit.ConcatenateStrings|
|first string|second string|concatenate?|
|Time is an illusion.|Lunchtime doubly so.||
|I demand that I may| or may not be Vroomfondel!||
|Ford, you're turning into| a penguin.||
```

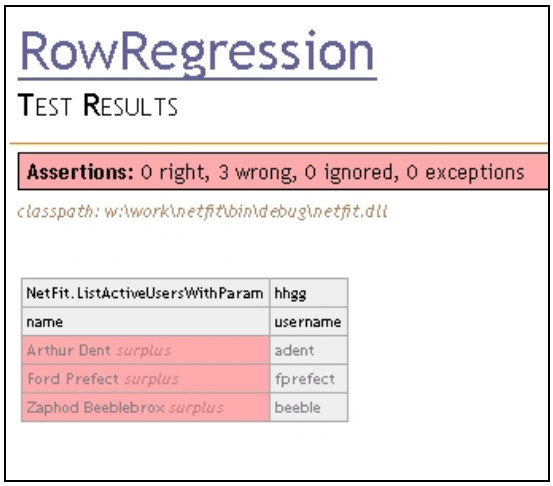
Run the test – it will not fail, but just print out calculated values in the third column. Now select the entire table in the browser (directly from the rendered page, not from the HTML source nor Wiki source), and copy it. Internet Explorer allows you to get just a few rows at a time, but some versions of Firefox require you to select the entire table in order to copy it properly. Try to paste what you copied into Notepad – you should see the table, with column values separated by tabs. Edit the test page again – replace the old table with tab-separated content from the clipboard, and then click on the *Spreadsheet to FitNesse* button. That should turn the tab-separated table into a FitNesse table, and even add the exclamation mark to the first row automatically.



You can also build data regression tests with *RowFixtures* – just a bit harder than in the previous example. Write the table header with data structure, but without any data rows.

```
!|NetFit.ListActiveUsersWithParam|hhgg|
|name|username|
```

Now execute the test – it will fail, printing everything that the *RowFixture* actually returned. Again, copy the table into the clipboard. Paste it into an editor – Notepad will do just fine. Do a global search & replace of *surplus* into an empty string (you can also include one blank before *surplus*). After the replacement is done, edit the page, paste the new table and press *Spreadsheet to FitNesse* again, and that's it – you have built the regression test!



Be gone with all those pipes

Although FitNesse Wiki syntax is really simple, you do not have to use it to write scripts. You can write your tables in Excel (or almost any other spreadsheet program), and then just copy them into FitNesse like the tables in regression examples. Clipboard automatically picks up data from most spreadsheet programs in tab-separated format, which can be directly converted to FitNesse with *Spreadsheet to FitNesse* button. If your spreadsheet program behaves differently, it should be able to export tab-separated files.

You can also convert a FitNesse table to tab-separated data with *FitNesse to Spreadsheet* button in page editor, and then copy that into Excel for editing.

Managing Wiki content

FitNesse is not just a test-server. It is also a great collaboration content management tool, which allows you to add requirements, specifications, useful links and supporting documentation to the tests, and exchange ideas with the team. It is no coincidence that tests tables can look more like English text than a code-related script.

Formatting text

FitNesse is a Wiki – a relatively free-form content management system that allows users to build pages and link them together. Instead of using HTML directly, Wikis use a special markup syntax – you have already seen pipes (|) used to create tables. Here are a few more interesting markup symbols:

Markup	***Effect***
!1	Apply Heading 1 style to the rest of the line
!2	Apply Heading 2 style to the rest of the line
!3	Apply Heading 3 style to the rest of the line
!c	Align to centre
----	Horizontal line (4 or more dashes)
!img *url*	Display image from *url*
!img-l url	Display image, left aligned
!img-r url	Display image, right aligned
'''*text*'''	Bold - three single quotes enclosing *text* on each side.
''*text*''	Italics – two single quotes enclosing *text* on each side.
#	Comment – ignore the rest of the line

Check out http://FitNesse.org/FitNesse.MarkupLanguageReference for a detailed reference of the Wiki markup language of FitNesse.

Links

FitNesse automatically recognises most links and builds proper HTML code for them – external links should just begin with http:// and internal links are built from CamelCase words – beginning with a single capital letter and containing at least one more capital letter. If the url ends with .gif or .jpg, FitNesse will automatically replace the url with the image. You can create additional links yourself by putting *[[label][url]]* anywhere in on the page. This can be used to create links which FitNesse does not recognise (if the word is not in CamelCase), or to change the default label for the link.

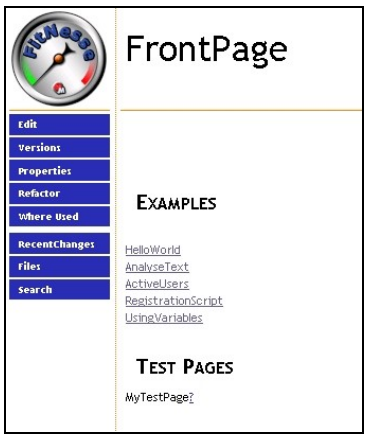
Preventing Wiki formatting

FitNesse does a lot of formatting on it's own, most of the times guessing the right thing to do. However, in some cases you explicitly want to prevent 'smart' formatting. For example, formatting should not be applied to code examples, class names, and generally to test tables.

You already know that you can use an exclamation mark (!) to prevent any smart formatting of table contents. However, some basic formatting (such as variable replacement) will still be done. If you want to prevent all formatting, enclose the text into !- and -!. To prevent FitNesse from parsing and formatting large blocks of text, enclose those blocks into three curly braces ({{{ and }}}) – you should typically do this with code examples, but you can use that trick to enclose any pre-formatted block of text.

Managing pages

Pages are units of content in FitNesse – and you will use pages to group related tests (and create testing scripts). Putting tables on a single page guarantees that they will be executed in sequence, in a specific order (top-down, and left to right).



To create a new page, either write the page name after http://localhost:8888/ or edit one of the existing pages and add the page name to the content. FitNesse will automatically create a link from CamelCase words, or print a question mark link if the referenced page does not exist. So, try this:

- click on the top-left dial icon to go to the homepage (or open http://localhost:8888/)
- click on the *Edit* button on the left
- add MyTestPage to the bottom of the page content
- click on Save

When the homepage loads again, it will display MyTestPage on the bottom, and a question mark with a link after that. Click on the link and you will open the new page in edit mode – now enter some lines, a heading, maybe even a test table. Click on *Save* and that's it – the new page is in the Wiki, with a working link from the home page. You can now click on *Properties* and specify actions which can be executed on the page.

What if there are no buttons?

If the *Properties* button is not there when you need it, just add *?properties* to the URL. If there is no *Edit* button there, add *?edit* to the URL. Sometimes *?edit* will help create the page if FitNesse does not offer it (for example, when the page name is not a CamelCase word – though it's best to keep to CamelCase page names, as some versions of FitNesse throw a *NullPointerException* when you try to edit such a page).

Group common pages into sub-wikis

Sub-wikis are the FitNesse equivalent of Web folders or C# namespaces– they can be used to group related pages and control their common properties (I'll explain this in more detail in the chapter about test suites). Instead of a slash (/), the dot(.) is used to separate levels of hierarchy in FitNesse. So, for example, http://localhost:8888/CustomerTests.OpenNewAccount leads to *OpenNewAccount* page in *CustomerTests* sub-wiki.

All relative links from *OpenNewAccount* will lead to pages in *CustomerTests* sub-wiki. To go to the top level, prefix a page name with a dot. So the link to *.FrontPage* leads always to the homepage of the site. Similar, the carret (^) leads to a sub-level (*^OpenNewAccount* link on *CustomerTests page leads to CustomerTests.OpenNewAccount*).

Create a couple of pages in the *CustomerTests* sub-wiki and then open the main page of the sub-wiki: http://localhost:8888/CustomerTests. Put just one line containing *!contents* into the page body and save it. This page will display links to all the other pages in the sub-wiki. Using *!contents* is a quick and easy way to build tables of contents for the sub-wikis, so that you do not have to add links for each page manually. Add *-R* after !*contents* to build the table of contents for the complete hierarchy – including contents of all sub-sub-wikis.

Define common content with special pages

Special pages *PageHeader* and *PageFooter* are included on the top and bottom of all other pages in a sub-wiki. Edit those pages to define common headers or links that are always displayed.

Organising tests into test suites

FitNesse supports grouping tests into test suites with sub-wikis. Putting tests into a test suite allows you to:

- Execute them together with a single click
- Manage their files easier as a group
- Define common properties like DLL paths only once
- Define common actions that should be executed before and after each test
- Define common actions that should be executed before and after the entire suite

Creating test suites

Any sub-wiki can be turned into a test suite by enabling the *Suite* property for the main page of the sub-wiki. When you do that, the *Suite* button will appear on the top of the left column, and pressing that button will run all tests in the suite.

Let's go through an example – we'll create a test suite with two tests. Point your browser to http://localhost:8888/TestSuite.ConcatenateStrings – create that page, mark it as a test and put the following table into it:

```
!|Concatenate Strings|
|first string|second string|concatenate?|
|Hello|World|Hello World|
```

Now point your browser to http://localhost:8888/TestSuite.ListActiveUsers – create the page, mark it as a test and paste the following content into it:

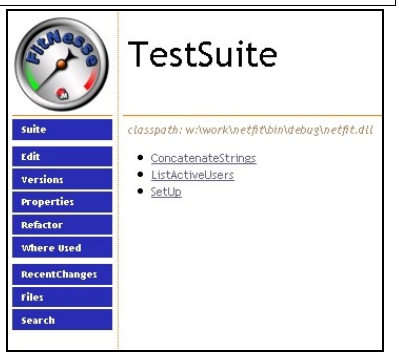
```
!|List Active Users With Param|jedi|
|name|username|
|Obi-Wan Kenobi|kenobi|
|Mace Windu|windu|
|Qui-Gon Jinn|quigonn|
```

Notice that there are no DLL path definitions nor namespace imports this time – just the tests. All common properties will be defined globally for the entire test suite.

Now open the main page of the sub-wiki: http://localhost:8888/TestSuite – edit (or create) the page, and paste the following content (replace the DLL path with the appropriate path on your system):

```
!path examples\netfit.dll
!contents -R
```

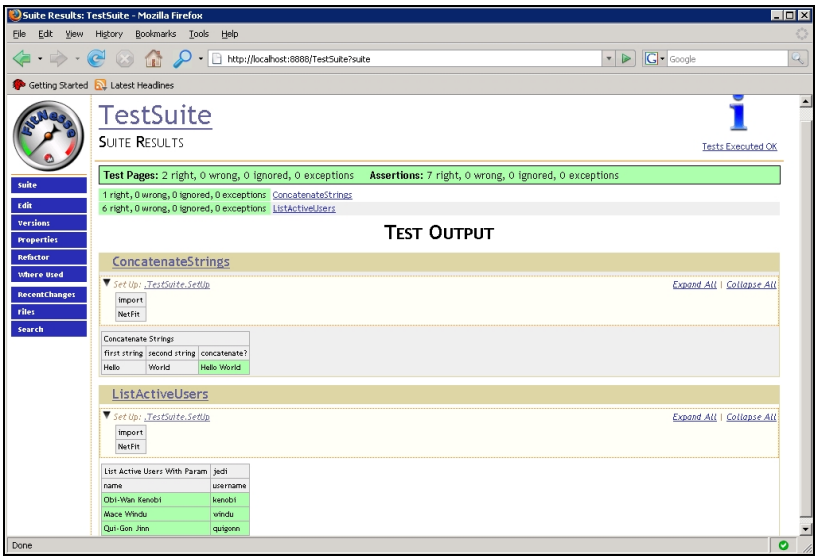
Now click on *Properties* on the left, and check the *Suite* checkbox, then save the properties. A suite button will appear on the left, and the page will display sub-wiki contents. Notice that the DLL path is defined in the main suite page – that will automatically load the referenced DLL for all tests in the suite.



Next, open http://localhost:8888/TestSuite.SetUp and put the following content into it (no need to mark it as a test):

```
!|import|
|NetFit|
```

SetUp is executed before each test – so individual tests do not have to import the namespace.
The test suite is ready now. Go to http://localhost:8888/TestSuite and click on the *Suite* button on the left. All tests in the suite will run, and you will see a summary report on the top, with test results below:



Common actions

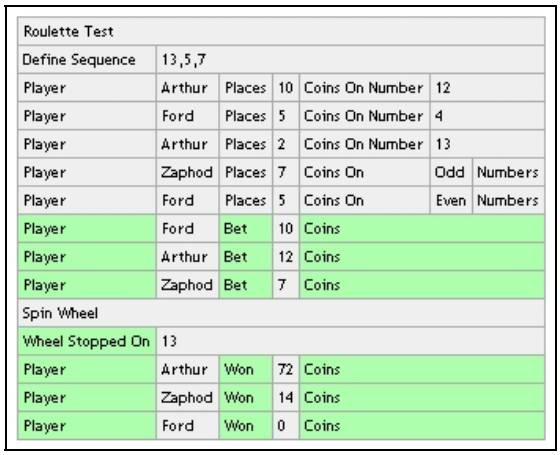
In addition to *SetUp*, you can also create a *TearDown* page (it will be executed after each test). There is also an option to create *SuiteSetUp* and *SuiteTearDown* pages, which will be executed when the entire suite starts/ends. Common content, as mentioned in the section on sub-wikis, can also be defined with *PageHeader* and *PageFooter*, and we also used the main suite page and the *root* site page to add common definitions. So, which of them should you use for what? Here are a few simple guidelines:

- Common HTML content should be put into *PageHeader* and *PageFooter*, as they are pasted directly into the page code. *SetUp* and *TearDown* are included into special, framed sections.
- Global path and runner definitions, such as including basic Fit libraries and setting up a .Net2 runner can be added to */root* if you use FitNesse only for .Net testing.
- Project-specific DLL paths should be added to the test suite main page. FitNesse looks for path definitions up the hierarchy, so this ensures that correct DLLs will be loaded for all the tests.
- Variables global for the entire test suite scope (utility URLs, randomiser sequence numbers etc) should be added to *SetUp*. *SuiteSetUp* is not included directly into the test page content, so FitNesse will not see variables defined in *SuiteSetUp* on individual test pages.
- Package includes and initialisations like setting up a database connection pool can be added to either *SuiteSetUp* or *SetUp* – however, I strongly suggest using *SetUp* for that - this makes sure that the environment will be properly initiated for all the tests if they are executed separately. *SuiteSetUp* will not be run when you execute individual test pages in the suite.
- Only the actions that really have to be executed once and only once for the entire suite run should be added into *SuiteSetUp*.

Writing better test scripts

So far, you know all the basics of FitNesse – before you continue the journey on your own, I'd like to show you how to efficiently write test scripts. This chapter will introduce one more test class type – the versatile *DoFixture*, and a couple of shortcuts and tricks which will enable you to write test scripts quicker and make them easier to read.

From ActionFixture to DoFixture



Roulette Test						
Define Sequence	13,5,7					
Player	Arthur	Places	10	Coins On Number	12	
Player	Ford	Places	5	Coins On Number	4	
Player	Arthur	Places	2	Coins On Number	13	
Player	Zaphod	Places	7	Coins On	Odd	Numbers
Player	Ford	Places	5	Coins On	Even	Numbers
Player	Ford	Bet	10	Coins		
Player	Arthur	Bet	12	Coins		
Player	Zaphod	Bet	7	Coins		
Spin Wheel						
Wheel Stopped On	13					
Player	Arthur	Won	72	Coins		
Player	Zaphod	Won	14	Coins		
Player	Ford	Won	0	Coins		

DoFixture is a part of *FitLibrary*, a popular extension to Fit, which is distributed in the main FitNesse.Net package. Instead of following a UI metaphor, *DoFixture* table looks more like a story – rows do not have to follow the same structure, and most lines can seem like English sentences.

Rows two to seven in the picture on the right set-up the test – populating a Roulette table with coins. Most of them would have to be written as two or three 'enter' and 'press' actions of *ActionFixture*. Rows 8-10 test if the coins have been appropriately placed, corresponding to combinations of 'enter' and 'check' actions. Row 11 is like a single 'press' action in the *ActionFixture*, and the following four rows are again compound checks. All in all, the equivalent action fixture would be at least twice as long.

Each row of the table (apart from the header one) corresponds to a single method of the Fixture class. Odd cells are combined to produce a method name, and even cells are used as parameters. So, the third row actually calls method *PlayerPlacesCoinsOnNumber* with parameters *"Arthur", 10* and *12*. Methods that just set-up data don't return anything, and the comparison methods return a bool value – if that value is true, the row is coloured green, and the test passes. If the value is false, the test will fail (and the row will be coloured yellow).

Here is the source code for the test class (download the complete source code for the Roulette Game from http://gojko.net/fitnesse):

```
public class RouletteTest:fitlibrary.DoFixture {
      private FixedRouletteAlgorithm algorithm;
      private RouletteGame game;
      public RouletteTest() {
            algorithm=new FixedRouletteAlgorithm();
            game= new RouletteGame(algorithm);
      }
      public void PlayerPlacesCoinsOnNumber(string player, int coins,int field)
      {
            game.PlaceBet(player,new NumberBet(coins,field));
      }
      public void PlayerPlacesCoinsOnNumbers(string player, int coins,
                  string numtype) {
            game.PlaceBet(player,
            new OddEvenBet(coins,numtype.ToLower().Equals("odd")));
      }
      public bool PlayerWonCoins(String player, int won) {
            return game.GetWinnings(player)==won;
      }
      public bool WheelStoppedOn(int number) {
                  return game.GetWheelPosition()==number;
      }
```

```
	public void SpinWheel() {
		game.SpinWheel();
	}
	public void DefineSequence(int[] sequence) {
		algorithm.SetSpins(sequence);
	}
	public bool PlayerBetCoins(String player, int coins) {
		return game.GetTotalBet(player)==coins;
	}
}
```

IMPORTANT: Notice how a comma-separated list of values was automatically converted into an array for the *DefineSequence* method – this is not a property of DoFixture and you can use that trick with all fixture classes.

If you read the example carefully, you might have noticed that there was no API to report the actual value of a failed test – change the table so that one of the tests fails, and you will see the row in red, but will not know what actually happened.

DoFixture provides a way to report both expected and actual value – prefix the row content with *check* and put the expected value in the last cell.

To use the check keyword, change test-methods in the previous example to return expected values:

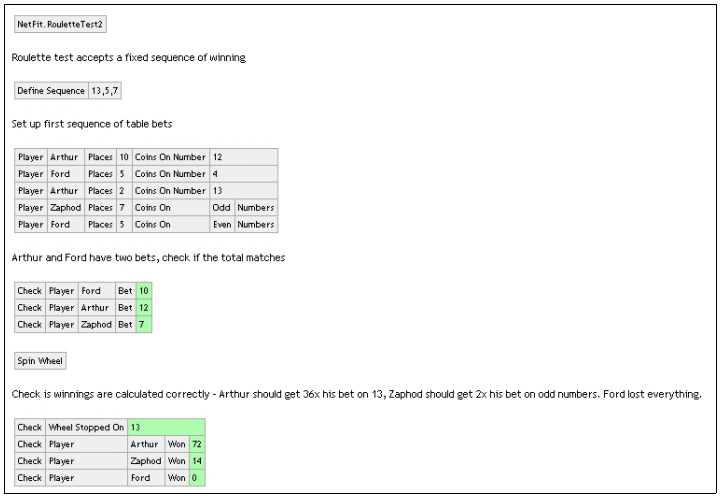
Roulette Test2					
Define Sequence	13,5,7				
Player	Arthur	Places	10	Coins On Number	12
Player	Ford	Places	5	Coins On Number	4
Player	Arthur	Places	2	Coins On Number	13
Player	Zaphod	Places	7	Coins On	Odd Numbers
Player	Ford	Places	5	Coins On	Even Numbers
Check	Player	Ford	Bet	10	
Check	Player	Arthur	Bet	12	
Check	Player	Zaphod	Bet	7	
Spin Wheel					
check	Wheel Stopped On	13			
Check	Player	Arthur	Won	72	
Check	Player	Zaphod	Won	14	
Check	Player	Ford	Won	17 *expected* 0 *actual*	

```
public int PlayerWon(String player) {
	return game.GetWinnings(player);
}
public int WheelStoppedOn() {
	return game.GetWheelPosition();
}
public int PlayerBet(String player) {
	return game.GetTotalBet(player);
}
```

Split the table to make tests more readable

If the DoFixture is started by the first table on a page, it takes over page processing. This allows you to split the rows into sub-tables, and Fitnesse will treat them as a single table.

See the picture on the right for an example, functionally equivalent to the previous one. Notice how the test is much more readable – rows are split into logical groups, and comments between rows explain test steps.


NetFit.RouletteTest2

Roulette test accepts a fixed sequence of winning

Define Sequence	13,5,7

Set up first sequence of table bets

Player	Arthur	Places	10	Coins On Number	12	
Player	Ford	Places	5	Coins On Number	4	
Player	Arthur	Places	2	Coins On Number	13	
Player	Zaphod	Places	7	Coins On	Odd	Numbers
Player	Ford	Places	5	Coins On	Even	Numbers

Arthur and Ford have two bets, check if the total matches

Check	Player	Ford	Bet	10
Check	Player	Arthur	Bet	12
Check	Player	Zaphod	Bet	7

Spin Wheel

Check is winnings are calculated correctly - Arthur should get 36x his bet on 13, Zaphod should get 2x his bet on odd numbers. Ford lost everything.

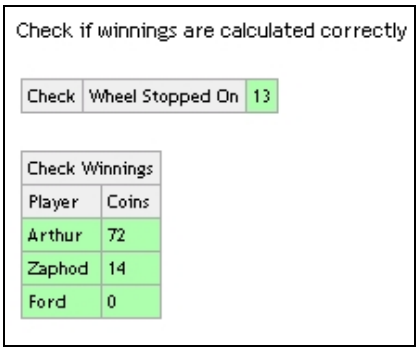
Check	Wheel Stopped On	13		
Check	Player	Arthur	Won	72
Check	Player	Zaphod	Won	14
Check	Player	Ford	Won	0

IMPORTANT: You cannot use Fitnesse keywords and symbols in DoFixture, but there are replacements for most of them. For example, you cannot use a blank cell to print the actual value without any testing, but you can achieve the same thing by prefixing a row with *show* keyword. Likewise, *error* keyword cannot be used to expect that a test will throw an exception, but you can prefix the row with *reject*. Note that in the current Fitnesse.Net build *reject* does not fail if there was no exception.

Embed other fixture types to write even more compact tests

When *DoFixture* takes control of the page, you can use other fixture types normally – the control will be returned to *DoFixture* after the table is finished.

However, *DoFixture* provides yet another useful shortcut – you can embed other fixture types into a *DoFixture* by returning them from methods. For example, the winnings could be checked much more efficiently if they were in a *RowFixture*. Instead of comparing individual values, just create a method that returns a *RowFixture* and list winnings in a table. Fit will automatically report any surplus or missing players and compare winnings.



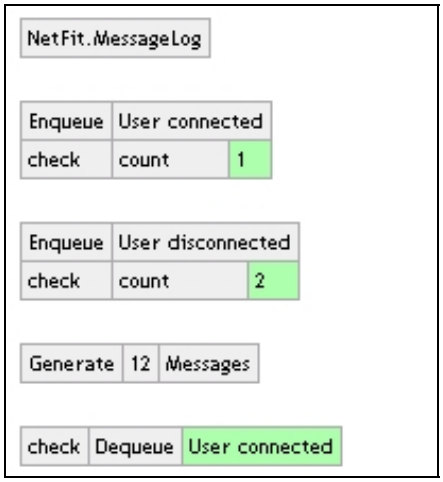

```
public RowFixture CheckWinnings() {
      return new WinningsRowFixture(game);
}
```

Wrap business objects in three lines

In many previous examples the Fixture just wraps around a business object and forwards the calls, acting as a glue between Fitnesse and your objects. *DoFixture* helps again – just set the protected *mySystemUnderTest* property to your business object, and call methods of that object directly. It does not have to sub-class the *Fixture*. Here is a short example that tests how .Net Queues work:

```
public class MessageLog:fitlibrary.DoFixture {
      Queue<string> queue=new Queue<string>();
      public MessageLog() {
            mySystemUnderTest=queue;
      }
      public void GenerateMessages(int count) {
            for (int i = 0; i < count; i++)
                  queue.Enqueue("M" + count);
      }
}
```

If there is no matching method in the test class, appropriate method of the *system under test* will be called. So, in the example on the right, *Enqueue* and *Dequeue* calls are forwarded directly to the Queue object, likewise the count property is read directly from the queue. *Generate 12 Messages* row matches a test class method, so that method is called. This way you can mix and match – wrap a business object in three lines and augment it with test-specific methods.



Important: In the online documentation (for Java version), the same effect is achieved with a method – *setSystemUnderTest()*. This method does not exist in the .Net version, and is replaced with *mySystemUnderTest* property.

Use business objects in table cells

So far, we have used standard .Net types like strings and decimal numbers in cell values – but FitNesse can also utilise your own business classes in cells. Instead of converting strings into business objects manually, define a static method *Parse* to convert strings into business objects, and FitNesse will do the rest. Overriding *Equals* and *ToString* is typically a good idea, as it enables FitNesse to compare values and print them in cells. Here is the message queue example again, but with business objects instead of strings:

```
public class Event {
      public string Name;
      public Event(string eventName) {
            this.Name = eventName;
      }
      public static Event Parse(string s) {
            return new Event(s);
      }
      public override string ToString() {
            return Name;
      }
      public override bool Equals(object obj) {
            if (obj is Event) {
                  return ((Event)obj).Name.Equals(Name);
            }
            return false;
      }
}
public class EventLog:fitlibrary.DoFixture {
      private Queue<Event> queue=new Queue<Event>();
      public EventLog() {
            mySystemUnderTest=queue;
      }
      public void GenerateEvents(int count) {
            for (int i=0; i<count; i++)
                  queue.Enqueue(new Event("E"+count));
      }
}
```

This test table looks like the previous one – but test rows two and six work directly on *Event* objects, not strings. In the second row, FitNesse uses '*Test Event 1*' to generate an *Event* using static *Parse* method of that class. In the sixth row, the object is compared to the dequeued value using *Equals*.

NetFit.EventLog		
Enqueue	Test Event 1	
check	Count	1
Generate	10	Events
check	Count	11
check	Dequeue	Test Event 1

Some other shortcuts

In order to write tests for a large .Net project more efficiently, I developed a few extensions for .Net Fit/FitNesse library:

- numeric comparisons >, >=, < and <= can be used in cells for checks
- Nullable types like *int?* and *bool?* work properly
- !blank keyword can be used to check if a string is not blank
- symbols (<<*name*) can be used as fixture parameters. Fixture will get the symbol value directly.
- Fixtures can handle object arguments (not just string arguments) with protected *ArgsObjects* array

Those patches are not part of standard FitNesse distribution, but are free to use, and can be downloaded from http://gojko.net/fitnesse.

Continuing the journey

Congratulations, you are now well on your way through the wonderful world of FitNesse. You should have enough knowledge (and code that can be copied/pasted and used as a template) to continue the journey on your own. Here are just a few pointers where to go next:

- Online documentation: http://www.FitNesse.org – containing the official User Guide (Java version), full reference of Wiki markup syntax and further examples.
- FIT online documentation: http://fit.c2.com/ - the engine powering FitNesse under the hub. Contains additional documentation, FAQ and more examples.
- Online acceptance tests for the .Net implementation: http://FitNesse.org/FitNesse.DotNet.SuiteAcceptanceTests - a very good source of ideas. Browse through it to find new features and see how to use them.
- FitNesse Yahoo group: http://tech.groups.yahoo.com/group/fitnesse/ - online discussion forum, mailing list and a file repository. This is where to ask for help.
- Subversion repository for FitNesse.Net on SourceForge: https://svn.sourceforge.net/svnroot/fitnessedotnet/trunk – browse the latest source code for FitNesse .Net integration and see how things really work.
- FitLibrary: http://fitlibrary.sourceforge.net/ - lots of additional useful Fixture types.
- Mike Stockdale's page on FitNesse.Net plans: http://www.syterra.com/FitNesseDotNet.html
- Blogs with good articles on Fitnesse and .Net: http://codebetter.com/blogs/jeremy.miller/, http://www.cornetdesign.com/ and http://xman892.blogspot.com/
- Integrating FitNesse and Nant: http://sourceforge.net/projects/fnessenanttasks/
- Integrating FitNesse and CC.Net: http://fitnesse.codebetter.com/blogs/jeffrey.palermo/archive/2005/09/13/131914.aspx
- "FIT for developing software", the reference book about FIT (ISBN: 0321269349)
- Examples from that book ported to .Net: http://www.vlagsma.com/fitnesse/
- Storyteller - new tool for efficient management of automated testing with FitNesse – http://storyteller.tigris.org/.

About the author

Gojko Adzic is an all-round software architect/programmer with extensive hands-on experience in a wide variety of technologies and platforms, ranging from Python utilities for mobile devices to J2EE trading systems. His software story so far includes equity and energy trading, mobile content delivery, e-commerce, on-line betting and complex configuration management. He has also been involved in several IT newspapers and magazines, with more than 200 published articles about programming, operating systems, Internet and new technologies, and held the position of Editor-in-Chief of PC World in Serbia for two years. He is currently based in United Kingdom, where he helps companies evaluate and utilise new technologies, and build better software.

In his free time, Gojko maintains a blog about programming on www.gojko.net. Feel free to contact him by sending an e-mail to gojko@gojko.com.