Developing Applications in a TDD environment

This workshop will help you become familiar with setting up FitNesse, the FitNesse environment, understanding the relationship between FitNesse and your code, writing fixtures and developing applications in a TDD lifecycle.

On the previous day the delegates created a number of Wiki pages. The pages were populated with FitNesse tables. They were written against working systems. So the data in them is correct. Your job is to construct code against these working tables.

Setting up FitNesse

Tools:

Fitnesse.jar	Download, use following link http://fitnesse.org/FrontPage.FitNesseDevelopment.DownLoad Latest version will suffice	Create a directory off the root called \java\fitnesse Copy fitnesse.jar into that directory Run java –jar fitnesse.jar If run command fails, the JRE is not on the PATH, this needs to be set
Fitsharp DLLs	Download, use the following link https://github.com/jediwhale/fitsharp/downloads Look for version *release 1.8 for .net 4.0*	Unzip the file into \java\fitnesse\dotnet4
Visual Studio for C# or any other tool that supports C# development	Download from Microsoft or other supplier	Install anywhere you want
MS Office Excel		

Testing the setup

To ensure that the setup has worked do the following

1. From the command line,, navigate to \java\fitnesse
2. Run java –jar fitnesse.jar –p 8090
3. Open a browser

4. Type the following URL http://localhost:8090

If all goes well, you should get the Fitnesse home page running on the local machine.

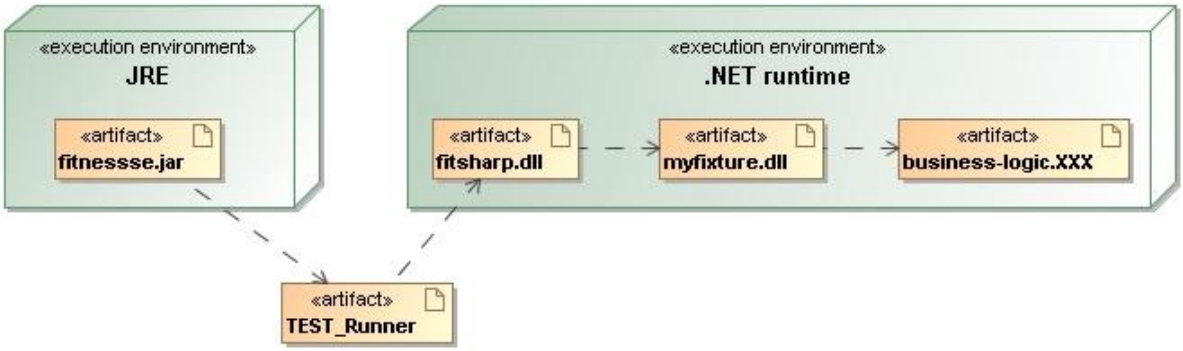
Course exercises

All solutions to the exercises in this document can be found in the following directories (all being subdirectories of where fitnesse is installed)

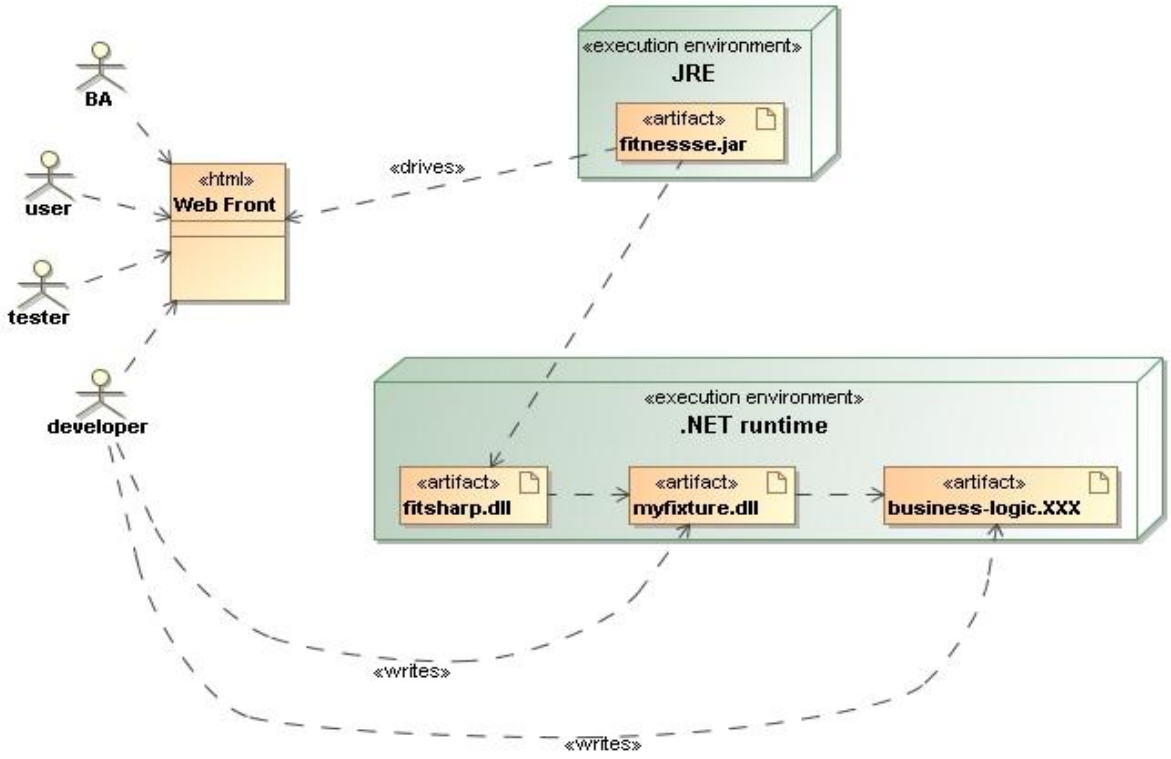
- C# source code; look in the directory ./ fitnesse/software_csharp_code
- Wiki pages (completed); look in the directory ./fitnesse/wikipages
- Tested, correct and runable applications used in the fitnesse test; look in the directory ./fitnesse/fitnesse_learning_tests

FitNesse Environment

The basic architecture for a FitNesse environment is shown here



FitNesse.jar is the webserver. Test_Runner.exe is an adaptor that separates the FitNesse server from the SLIM executor. This allows FitNesse to function with many different SLIM executors. FitSharp.dll communicates with your C# code. The following model gives another view of this design


«execution environment»
JRE
«artifact»
fitnessse.jar
BA
user
tester
developer
«html»
Web Front
«drives»
«execution environment»
.NET runtime
«artifact»
fitsharp.dll
«artifact»
myfixture.dll
«artifact»
business-logic.XXX
«writes»
«writes»

Exercises

Create a C# project called FitNesse_tests for the first three labs (all classes should be defined in the namespace **FitNesse_tests**). When you are ready to do lab 4 create a project called BankingSystems (all classes except fixture classes should be defined in the namespace **BankingSystem**, the fixture code should be defined in the namespace **banking_system**).

Echo (found in the wiki page: ReverseString, lecturer lead)

In this lab, the table echoes whatever you put in the table in reverse character order. Open the Wiki page ReverseString and select Edit. In the edit frame create the table shown here

Inverted Echo	
Original Text	Inverted Text?
Hello World	
My name is Selvyn	nyvleS si eman yM

1. Create the following class

```
namespace FitNesse_tests
{
        public class InvertedEcho
        {
                private String itsOriginalText;

                public   InvertedEcho(){ }

                public   void     setOriginalText(String val)
                {
                        itsOriginalText = val;
                }

                public   String   invertedText()
                {
                        String result = ReverseString( itsOriginalText );
                        return result;
                }

                Private   string   ReverseString(string s)
                {
                        char[] arr = s.ToCharArray();
                        Array.Reverse(arr);
                        return new string(arr);
                }
        }
}
```

2. Notice the colour coding that we have used. The matching colours indicate what you need to write in order for your code to be a compliant fixture that FitNesse can call onto. The class name should match the table name without any spaces. Essentially FitNesse removes any spaces between the camal case words of a table name. If there are any fields after the table name, they are passed as parameters to the class constructor.
3. Each input column in a table can either have a matching data member in your fixture (introspection is used to populate such data members) or, an operation with same name as your table column (spaces removed) and prefixed with the word "**set**".
4. Each output column in a table can either have a matching data member in your fixture (introspection is used to read such data members) or, an operation with same name as your table column (spaces removed) and no prefix.
5. Visual studio will compile this down the moment you save it. Modify the wiki page so that it points to the compiled unit (dll or exe). This is the **!path** parameter. Save and test the page.
6. If all is well, you should have zero on warnings, exceptions and errors.

NOTE

You may find when you run the test that you get a number of errors. An easy way to determine the cause of the error is to expand the field ***decisionTable _XX_XX*** which can be under EXCEPTIONS on the wiki page. There is usually sufficient enough information to allow you to track down the problem.

Typical problems are

- Fixture not found. This is usually caused by a typo in the !path field in your wiki page
- Class name not found. This is usually a case of misspelling the class name so that it does not match the table name.
- Class name not found. Could be due to the class not being public.
- Class name not found. Could be due to an incorrect namespace.
- Class name not found. Could be due to you not importing the namespace into the wiki page.
- Method name not found. This is usually a case of misspelling the operation name in your fixture or, it may be a compound error because the class was not found.

You try

String Concatenation (found in wiki page ConcatenateStrings)

In this lab, the table concatenates two strings. The table is defined the wiki page ConcatenateStrings

Concatenate Strings		
Left String	Right String	Concatenate?
Hello	World	HelloWorld
Some other	Value	**$value=**
$value	Night	OneDay

Notice use of wiki symbol

1. Define the fixture for this table, you should be able to complete this in 30 minutes. The lecturer will give you the solution after 20 minutes into the exercise.

Loan Calculator (found in wiki page LoanCalculator)

In this lab you are going to define and create two tables, the first allowing you to supply some basic figures for a loan (such as loan amount, period and interest rate) in which the result should be a figure indicating the monthly repayments. The second lab allowing you to invoke predefined operations on a fixture. This whole exercise is more complex than the previous exercise because you have to define the tables and the data. Use Excel to define the table, copy it into FitNesse and use the format button to convert the excel sheet into a fitnesse table.

NOTES

Use the following characteristics for your first table, a decision table

1. Table name: **Loan Calculator**
2. Field names: (in this order) Loan Amount, Period, Percentage Down, Fixed Rate, Monthly Payments (this is a result column)
3. Create some data for your table (ask the lecturer to give you an expected value for the monthly payments or use excel's functions if you know how to)

For the second part of the question you are going to use a fitnesse script table rather than a decision table

4. Table name: **Loan Calculator Script** with parameters (in this order), principal, length of loan in months, interest rate and initial down payment
5. The following operations are available
    a. **Monthly Payments**; returns the payments per month (ppm),
    b. **Monthly Payments Match**; takes an amount as input and returns true if it matches the monthly payments,
    c. **Initial Loan**; returns the initial loan amount less the initial down payment,
    d. **Initial Loan Minus**; returns an initial loan amount but allows you to specify the down payment as a percentage of the original loan
6. Use the script keywords (such as check, reject, ensure, show etc) to drive the underlying fixture. Try using wiki symbols; $<symbol name>= to create and assign a value to a symbol and, $<symbol name> to use a symbol.

Your tasks

The first thing you need to do for the Loan Calculator decision table is to create a class called LoanCalculator. The Payments column should return a value using the following formula

```
double result = 0;
double loanMinusDP = loanAmount - (percentageDown / 100 * loanAmount);
double monthlyRate = fixedRate / 12 / 100;
```

*result = monthlyRate * loanMinusDP / (1 - Math.Pow(monthlyRate + 1, -1 * (period * 12)));*
return result;

1. Use setters and getters for the fields in the table (do not depend on a class attribute name matching a table field name, this reduces your ability to control the input and output of table data).
2. Once you have written your fixture, test it using the wiki page. If it passes move on to the next part of the exercise otherwise, correct your code until it passes the FitNesse test.

You are now going to write a fixture for a script table

1. Create a class called LoanCalculatorScript, it needs two constructors, one with no parameters and the other with the following parameters
    a. double principle
    b. int period
    c. double interest, this should be stored as a percentile (i.e. the value passed must be divided by 100 before it is stored as an attribute of the class)
    d. double downPaymentPercentile
2. There are four operations defined in the table shown in the wiki page, each one will need to be implemented in the fixture

Monthly Payments	public String montlyPayments(); returns the montly payments given a principle, period and fixed interest rate (copy the code from the previous exercise) Can be used with the following wiki script commands • Show • Check [not] • reject • <function call> • ensure
Monthly Payments Match	Public bool monthlyPaymentsMatch(double expectedResult); returns true if expectedResult matches the monthly payments that are created by the values that initiated the script. Can be used with the following wiki script commands • show • check [not] • reject • <function call> • ensure
Initial Loan	public String initialLoan(); returns the amount that was originally borrowed Can be used with the following wiki script commands • show

	• check [not] • reject • <function call> • ensure
Initial Loan Minus DP	public String initialLoanMinus(double percentageDown); returns the initial amount borrowed minus a percentage down payment of percentageDown can be used with the following wiki script commands • show • check [not] • reject • <function call> • ensure

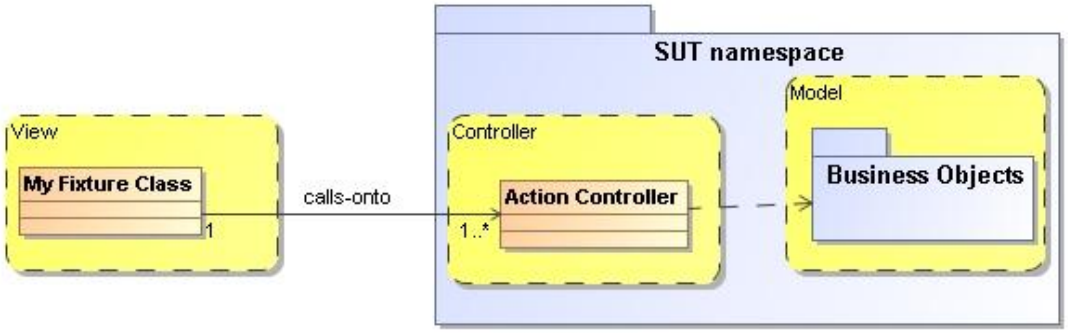
3. Develop each of the operations shown above, use the wiki page to test the functions that you have implemented.
4. On the wiki page, you will see that there is a symbol in the first table called ppm, this is holding the payments per month. This can be used to test that your script table is functioning correctly.

Open New Account (found in the wiki page OpenNewAccountPage)

Part lead by lecturer

This lab is spread over two wiki pages but, can be tested from this page (because it is a suite) or from each page individually. Study the wiki pages carefully and the case study notes before proceeding.

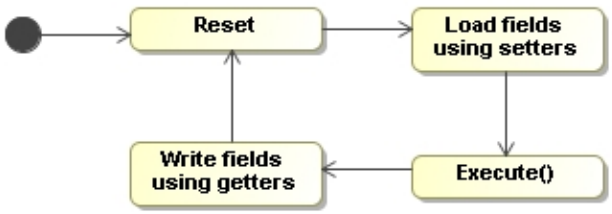
1. Familiarise yourself with the case study (document entitled Open New Account Workflow)
2. Because of time constraints and the amount of code required to make this lab work, we have supplied you with some code. Before you look at the supplied code we will describe the design of the application
    a. A fixture is a piece of code that sits between the FIT runner and your code. Its sole purpose is to allow an external test unit to drive your SUT (System Under Test).
    b. It is wise to avoid placing any of your business logic in the fixture, the fixture should simply call onto operations in the SUT.
    c. To avoid contaminating your fixture with business logic or the business logic with fixture code, we will follow the MVC pattern. The structure will be as follows



d. We have supplied you with the working classes shown in the class diagram plus some ancillary helper classes, here are the classes you need to implement

ApplicationForm	Holds all the information on the application (it will ultimately control the application form lifecycle)
OpenNewAccount	A fixture for the Open New Account table
DepositTake	A fixture for the Deposit Taken table
OpenNewAccountController	The adaptor that will sit between the fixtures and the SUT. Ultimately it forms part of the interface to the SUT

3. Begin with OpenNewAccount. Develop the fixture for this table (instructor lead)
   a. Begin by simply writing the fixture, getting it to build and you able to test it from the wiki pages. The FitNesse test should fail but there should be no exceptions specifying that the class or methods cannot be found.
   b. You need to return several values to the fitnesse table. Unlike the previous examples we need to set the values of several fields before they are passed back to fitnesse. To do this we need to add an extra operation *public void Execute()* to our fixture class. This operation is called once all the input fields on a table have been set. After this operation is executed, each output field in the fitnesse table is populated by calling the getter operations of the required fields.



   c. Write some code in the Execute() method to fix the errors that fitnesse is reporting. This will involve using the supplied classes and adding code the other classes (ApplicationForm and OpenNewAccountController).

   d. Focus your efforts on the controller class, we have put some of the most crucial code in place, add your code where you see the comment TODO:
   e. Each time you add a significant amount of code, use fitnesse to test how far you have progressed.
   f. Repeat steps D and E until the tests pass.
   g. Once the code passes, refactor (you should be able to move some of the code out of the controller into the ApplicationForm class.
4. Now move onto the DepositTaken fixture (you're on your own)
   a. Begin by simply writing the fixture, getting it to build and you able to test it from the wiki pages. The FitNesse test should fail but there should be no exceptions specifying that the class or methods cannot be found.
   b. Again we need to add our functionality to the Execute() method..
   c. Write some code in the Execute() method to fix the errors that fitnesse is reporting. This will involve using the supplied classes and adding code the other classes (ApplicationForm and OpenNewAccountController).
   d. Focus your efforts on the controller class, we have put some of the most crucial code in place, add your code where you see the comment TODO:
   e. Each time you add a significant amount of code, use fitnesse to test how far you have progressed.
   f. Repeat steps D and E until the tests pass.
   g. Once the code passes, refactor (you should be able to move some of the code out of the controller into the ApplicationForm class.

NOTE

We have written the code for this lab and placed in a number directories that can be found on your machines {lecturer will direct you to where the code can be found}. The code has been developed in stages 1 to 5

1. At the end of this stage, both fixtures are written and code compiles, OpenNewAccount.Execute(), OpenNewAccountController. verifyFormDetails() and OpenNewAccountController. addCustomerDetails() should all be written; maybe not to completion
2. At the end of this stage, the application form status should have been handled
3. At the end of this stage, we have linked the data from OpenNewAccount table to the DepositTaken table
4. At the end of this stage, all the code is in place, we now need to refactor (if it needs to be done)