CELESTIAL
TDD USING FITNESSE

Workshop

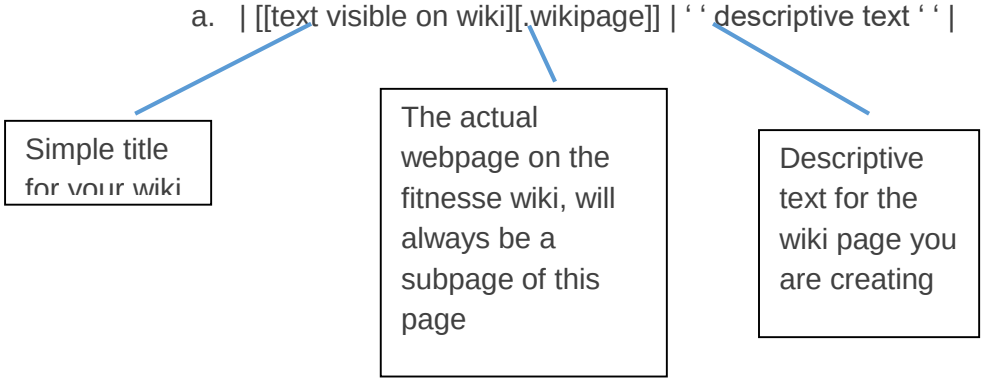
OVERVIEW

This document outlines a series of labs that are all TDD based. It starts with two duplicate exercises one will be using JUnit the other fitnesse. The purpose of the duplication is to clearly demonstrate TDD using something like fitnesse.

CREATING A WIKI PAGE

Start fitnesse with the following command

1. Java –jar fitnesse-standalone.jar -p 8090
    a. This will start fitnesse on port 8090 on your machine
2. Using a browser open the fitnesse wiki with localhost:8090
3. Select **Edit** (this can be found at the top left of the wiki page)
4. Examine the page carefully, you will see several line entries as shown here
    a. | [[text visible on wiki][.wikipage]] | ‘ ‘ descriptive text ‘ ‘ |



5. If the .wikipage entry refers to a page that doesn’t exist yet, it will be created once you save the current page and click question mark hyperlink the fitnesse wiki has created for you.
6. Once you create the new wiki page, you can select Edit (top left of the wiki page) and begin to mark it up using the wiki commands
7. Before a test a can be run from a page, you must enable the test feature of the page
    a. Select Tools/Properties
    b. On this page you will see numerous options, Under Page Type, select Test, then select Save Properties

INVERTED ECHO

JUnit/NUnit approach

1. Start a new project
2. Create new JUnit/NUnit test class, call it InvertedEchoTest in the package tdd_with_junit
3. Define a new test method called testInvertedText()
4. Your tests should specify an input value and an expected out value
5. Now define and implement a class that InvertedEchoTest will exercise, call the class InvertedEcho
6. DO NOT CHANGE THE TEST TO SUITE YOUR CODE

Fitnesse approach

Creating the wiki page

1. Let's begin by setting up a wiki page for this test, so make sure fitnesse is running and available on the port you specified above (8090)
2. Create a new wiki page using the technique described above, use the following fields
    - Page text: Reverse a string
    - Wikipage: .reversestring
    - Description: This test reverses a string literal no translation
3. Once the page has been created, select Edit on the new page
4. You will be faced with the following entries
    - **!contents -R2 -g -p -f –h**
5. You need to add the following entries
    - **!define TEST_SYSTEM {slim}**
        - Tells fitnesse you are using the slim protocol
    - **!path E:\work\software\netbeans\fitnesse_tests\dist\fitnesse_tests.jar**
        - Points to the fixture, you can have multiple path entries each one pointing to specific jar file

Defining the test script

Your wiki page should be in the Edit mode from above, if not repeat steps 3 .. 5 in Creating the wiki page

1. Add the following entries
    - | import |
    - | tdd_with_fitnesse |

        - This should match the java package name into which your fixture is defined
    - | InvertedEcho |
        - This is the name of our fixture – a Java class
    - | Original Text | Inverted Text? |
        - The first field, the one without a ? is an input field, you must define an operation on your fixture class (InvertedEcho in our case) with the following name setOriginalText (see below in defining the fixture). You can have as many input fields as you want but they must all conform to this rule if you want to pass data from the wiki to your fixture.
        - The second field, the one with the ? is an output field. You must define an operation on your fixture class (InvertedEcho in our case) with the following name InvertedText (see below in defining the fixture). You can have as many input fields as you want but they must all conform to this rule if you want to pass data from the wiki to your fixture.
    - |Hello World| |
        - The first field is an input value to be passed to the fixture
        - The second is left blank here, by doing this you are indicating to firnesse that you would like to see what the output value is without doing a comparison against an expected value
    - |Hello World|dlroW olleH|
        - The first field is an input value to be passed to the fixture
        - The second is set to dlroW olleH, by doing this you are indicating to firnesse to test the output value against an expected value dlroW olleH
    - You can create as many row entries as you feel are necessary to give you complete test coverage
2. Once your tests are in place, click the Save button at the bottom of the page.
3. Run the test by clicking Test on the upper left of the wiki page

Example of what your wiki edit page should look like

```
!contents -R2 -g -p -f –h

!define TEST_SYSTEM {slim}

#!path E:\work\software\netbeans\fitnesse_tests\dist\fitnesse_tests.jar

| import        |

| fitnesse_tests |
```

|Inverted Echo |

|Original Text|Inverted Text?|

|Hello World | |

|Hello World |dlroW olleH |

Defining the fixture

We are now going to define the fixture for the above wiki page

1. Within the same JUnit project Inverted Echo from above create a new class, call it InvertedEcho within a package called tdd_with_fitnesse
    - Fitnesse fixtures must be Java Beans (POJO)
2. Add an attribute called itsOriginalText of type String, it should be private
3. Add a setter for this attribute but it should have the following signature
    - public void originalText(String str)
4. Add a new getter operation called InvertedText with the following signature
    - public String InvertedText()
5. Implement an empty method and run the test from the fitnesse wiki
    - The tests should fail
6. Now implement the method to reverse the string
    - When you think the code works test it by running it from the fitnesse wiki page
7. When the tests pass, you have completed your first TDD approach to developing software

TIMESHEET EXAMPLE

This is a more complex example than the InvertedEcho example but the process is the same. In this example, you will be given the main steps but you must work out the detail. We will provide the test data.

The scenario is as follows

A small company has been using an excel type application to capture timesheet data for quite some time giving rise to a number of obscure reporting and output formats for the time fields. They have a plan to move to a COTS product but need to import all the all data from the existing system. At some point the data will be exported out of the original application an imported into the COTS product. The COTS product expects the data to be correct and in the correct format – time entries must be in 24hr format, hourly rate must be integers, hours worked and pay must be in a decimal format of two decimal points.

Unfortunately, your organization no longer has access to the original code base that was used to create the custom timesheet application and the only fields that are present are – Check In Time, Check Out Time and Hourly Rate, the Hours Worked and Pay must be calculated using a small transformation application that will be written in Java.

- The amount to be paid = (hours + minutes worked) * hourly rate
- The test script must accept the check in time and check out time. These times must be specified in the following format HH:MM [<AM|PM>], where AM|PM not specified AM is assumed, otherwise 24hr notation may be used or any combination.
- The hourly is a decimal value
- The hours worked should be as a decimal value

Your activities

1. Define your wiki, its wiki title should be Time Sheet, wiki page timesheet, and description is some free text of your choice
    a. The fixture name will be TimeSheet
    b. Package name timesheet_with_fitnesse
2. Define your fixture
3. Complete the rest of the business logic in the file TimeFormatter.java
4. The test data is

Check in time	Check out time	Hourly rate	Check In Time Normalised?	Check out time Nornalised?	Hours worked?	Pay?
9:00 AM	5:00 PM	7	09:00	17:00	8.00	56.00
10:30 AM	12:30PM	4	10:30	12:30	2.00	8.00
9:45AM	12:45 PM	10	09:45	12:45	3.00	30.00
9:30 AM	12:15	10	09:30	12:15	2.75	27.50
9:30	12:15 PM	10	09:30	12:15	2.75	27.50
09:00 AM	15:00	8	09:00	15:00	6.00	48.00
09:00	3:00 PM	8	09:00	15:00	6:00	48.00
9:05 PM	3:30 PM	7	09:05	15:30	ERROR	NIL

EXTENDING THE TIMESHEET EXAMPLE

You are going to add a new feature to the timesheet application. There is now a requirement for you to first register the name, employee ID, and hourly rate of each employee. Once the employee details have been loaded into the system, your timesheet fitnesse test script above must be modified so it has the following columns

Check in time	Check out time	Employee ID	Check In Time Normalised?	Check Out Time Normalised?	Hours worked?	Pay?

If the employee ID is not found in the loaded employees table, the test should fail by indicating NIL hours worked and NIL Pay. Employee IDs must be unique and of the following format EMP###-YYMMDD - ### number 001..999, YYMMDD date when employment began

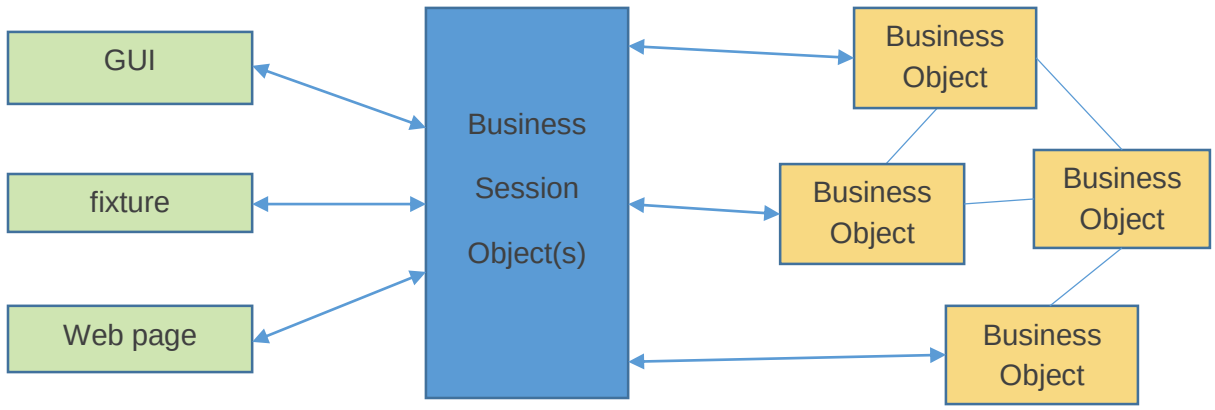
To load employees use the following test script (we will call this fixture Load Employees)

Employee Name	Employee ID	Hourly Rate	Loaded?

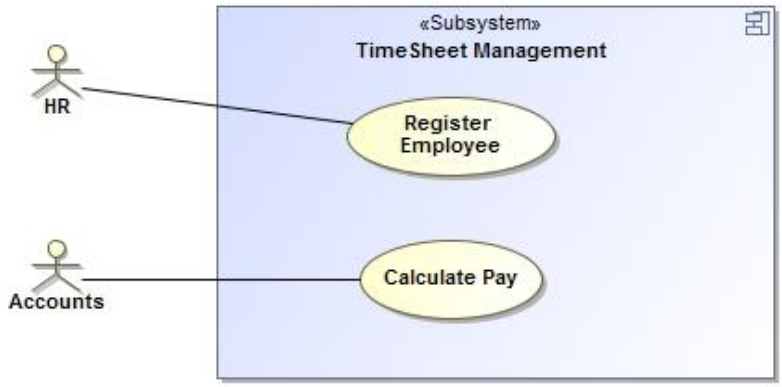
Two criteria must be met for an employee record to be loaded

1. If an Employee ID is not unique Loaded should result in NO otherwise it should be YES.
2. If the Employee ID does not conform to the format described above, Loaded should result in NO otherwise it should be YES.

At this point you should have realized that this problem cannot be solved with one class. The first thing to note is that you can no longer place all the logic in the fixture. In actual fact it's bad practice to place your logic in the fixture. The fixture should be used as a gateway to your business classes. What we actually want to do is to create a separation between your business logic and the fitnesse harness as shown here

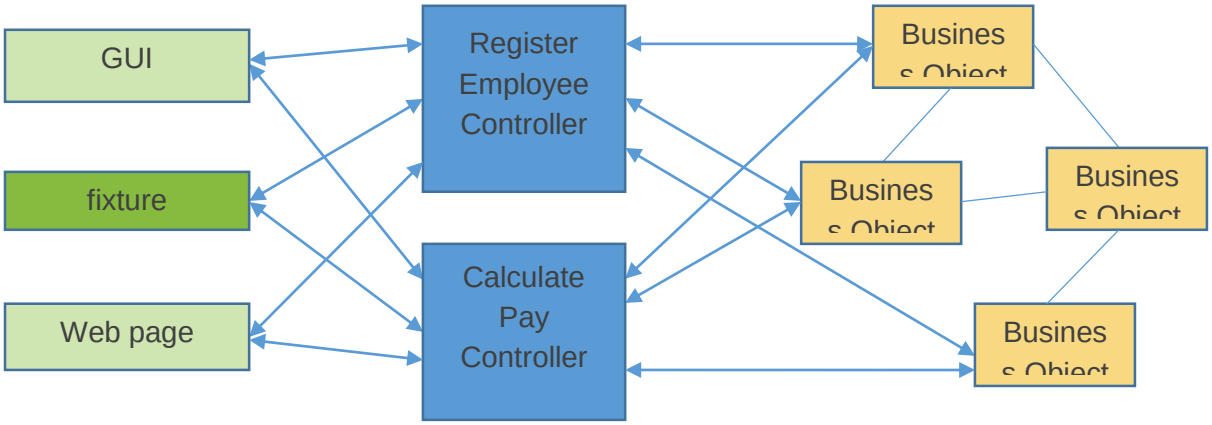


A better way to understand this model is to think of it in terms of UML use cases



A use case represents the sequence of interactions between the actor and the system to achieve a business goal. When we translate UML models to code, it is good practice to create a class for each use case. We call these <<controllers>>. They are essentially objects that coordinate business objects to achieve the business goal.

So our above model could be modelled as follows

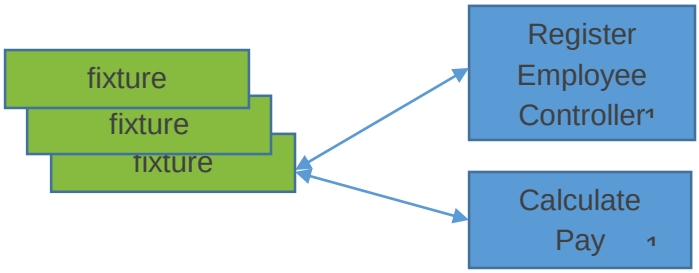


Wiring your model

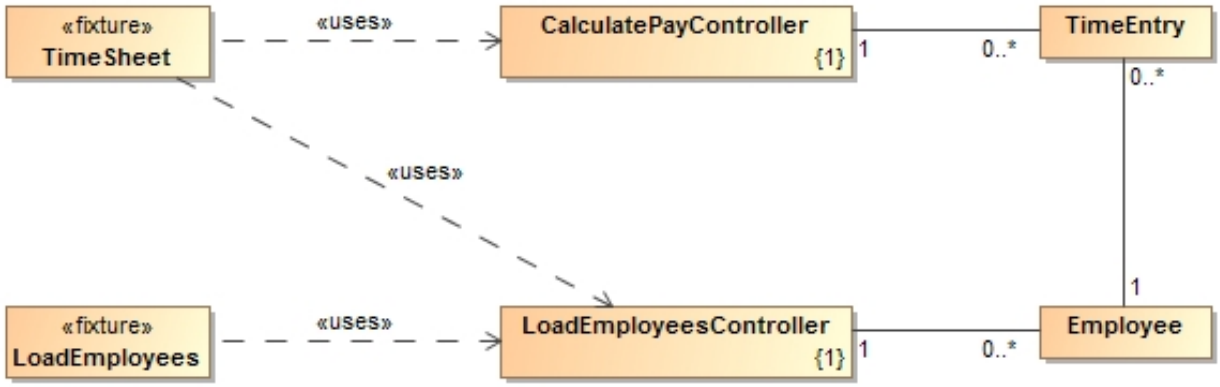
You will write two fixtures, LoadEmployees and TimeSheet. LoadEmployees will ask RegisterEmployeeController to create Employee objects. RegisterEmployeeController will ensure that the business objects are available to the CalculatePayController. CalculatePayController is used by the TimeSheet fixture.

A fitnesse fixture is active for the lifetime of a fitnesse test page. The best but not the only approach is to code the controllers as singletons, each being accessed by several fixtures. Each of these fixtures are called from a single fitnesse test page, as the model below shows.

By wiring your model like this, it is possible to pass data between fixture tests on the same fitnesse page.



The design for your application can be described with this class model



Employee	Holds all the data from each loaded Employee
TimeEntry	Holds all he data from each row on the TimeSheet. Each object should be linked to one Employee.