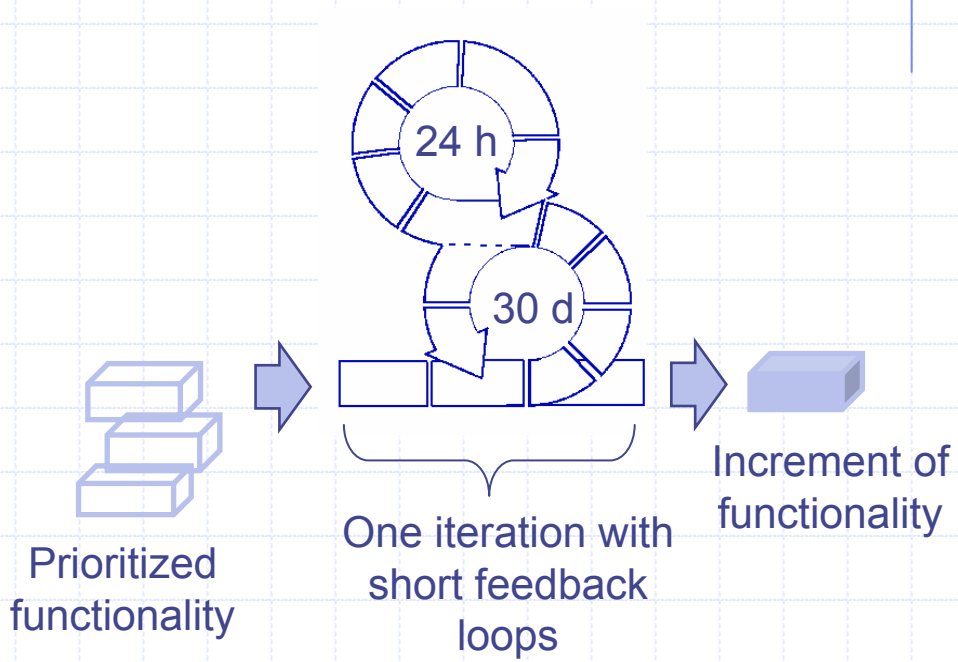
Test-Driven Development at the Acceptance Testing Level

Dr. Christoph Steindl

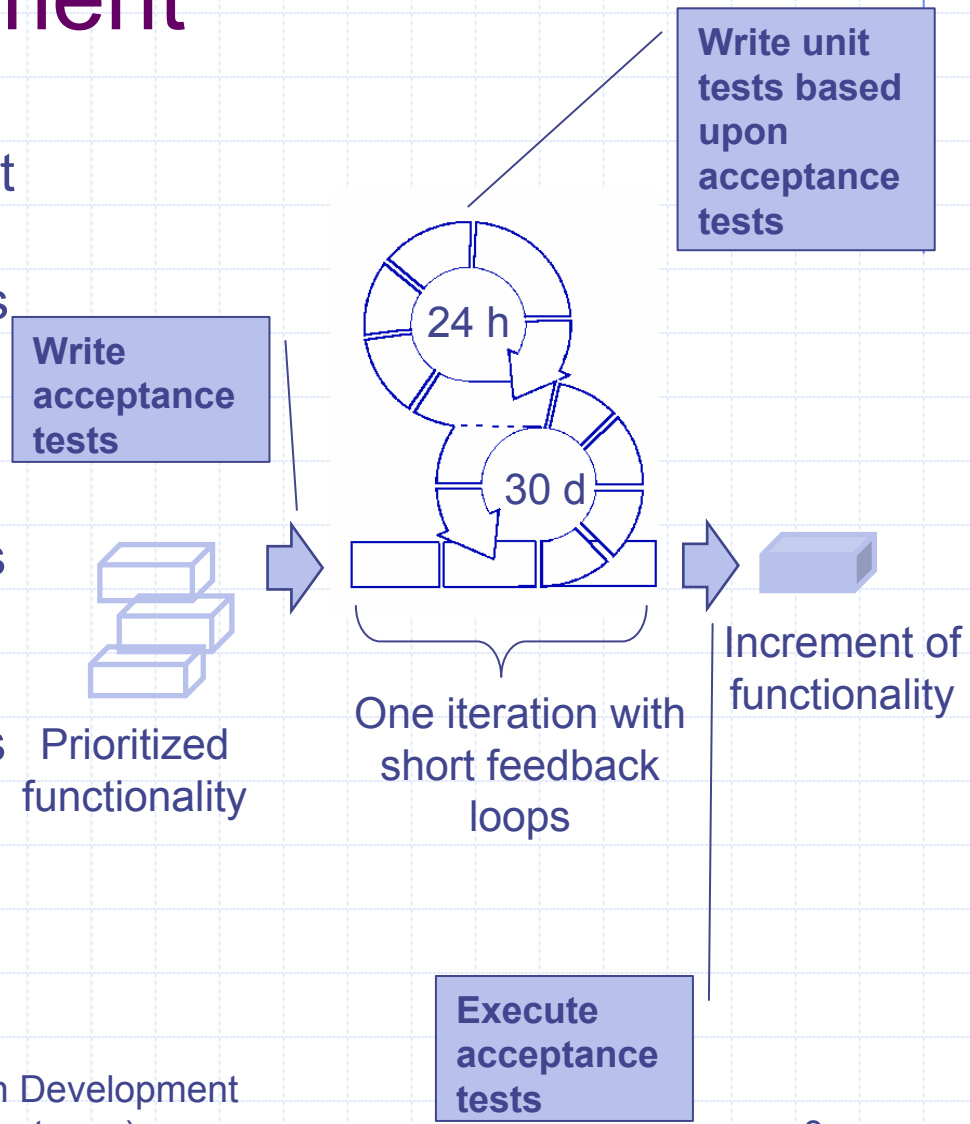
Iterative Software Development

- At the start of the iteration, the customer explains the expected functionality to the team. The customer sets the priorities, the team estimates the effort.
- Then the team thinks about the necessary tasks to implement the functionality, details the estimation and assigns the tasks.
- During the iteration, the team holds 15 minutes status meetings in order to discuss the current tasks, the achievements and the problems.
- At the end of the iteration, the team demonstrates to the customer the increment of potentially shippable functionality.



Acceptance Testing with Iterative Software Development

- Don‘t wait until the software has been written in order start with the acceptance testing.
- Write the acceptance tests as soon as possible.
- Build the unit tests upon the acceptance tests.
- Execute the acceptance tests during the iteration to understand the progress.
- Execute the acceptance tests at the end of the iteration for verifying that the requested functionality has been built.



Why?

- Better collaboration between customer and developer, faster feedback
- Customer / Business / User / Domain Expert
  - Specifies the requirements
    - In the language of the business, focusing on the scenarios, the flow of events, the dynamic behavior
    - In an executable form
    - Where the execution can be automated
    - Before the requirements are implemented
  - Verifies the requirements
- Targets errors not found by unit testing
  - Requirements are mis-interpreted by developer
  - Interface of software is not as intended
  - Modules don‘t integrate with each other

Approaches to Acceptance Testing

- Manually
  - User exercises the system manually using his creativity
  - But:
    - The developers don‘t know the goal, the tests that the system has to pass
      ➔ this approach does not support Test-First-Design
    - Expensive, due to manual effort which has to be repeated whenever the system changes
    - Errors may be overlooked (no automated verification whether the actual matches the expected)
    - There can be big arguments about the pass/fail decision
- GUI Capture & Replay
  - Capture user events (mouse, keyboard) in modifiable script, abstracting from screen coordinates to GUI objects
  - But:
    - GUI has to exist, so this approach does not support Test-First-Design
    - Tools are expensive
    - Tests are brittle, have to be re-captured if the system changes
- Framework for automating the functional tests
  - E.g. FIT, FitNesse
  - Easy for user to describe the requirements themselves (no programming, just text), easy for developer to glue the requirements to the business logic
    - **Tests are written before the code, so this approach supports Test-First-Design**
    - **Inexpensive because the frameworks are open source, execution of the tests can be automated.**
    - **The developer has a clear goal to achieve!**
  - But:
    - No capture & replay possible

What is FIT?

- An open source framework (under the GPL) created by Ward Cunningham: http://fit.c2.com
  - Supports Java, .NET, Python, Lisp, Scheme, Ruby, Perl, C++
  - has enough logic to parse HTML, run tests, capture results and output them as a modified HTML document
  - For data-driven tests (input – processing – output) where the tests look like spreadsheets
- The customer write tests as HTML tables.
- The framework interprets the tables, the glue code passes the values to the test code, the test code exercises the business logic.
- The customer documents the test with free text between the tables (which is ignored by the framework).

How to Use FIT?

- **Fixtures** are types of HTML tables with a specific behaviour of interpreting the values in the table.
- **ColumnFixture**: maps columns in the test data to fields or methods; a new column fixture is created for each table that uses one.
- **ActionFixture**: executes the command in the first column
  - **start** aClass: create an object of aClass to work with
  - **enter** aMethod anArgument: invoke the method on the object
  - **press** aMethod: invoke the method on the object (without parameters)
  - **check** aFunction aValue: invoke the (parameterless) function and compare return value with the specified value
- **RowFixture**: invokes methods on the objects and compares the returned values to those in the table
  - **binds** the columns to variables and methods by reflection.
  - **Executes the functions** to get the result rows which will be checked.
  - **matches** the expected and result rows and check the matches.
  - **marks** missing and surplus rows

eg.Division		
numerator	denominator	quotient()
1000	10	100.0000
-1000	10	-100.0000
1000	7	142.85715
1000	.00001	100000000
4195835	3145729	1.3338196

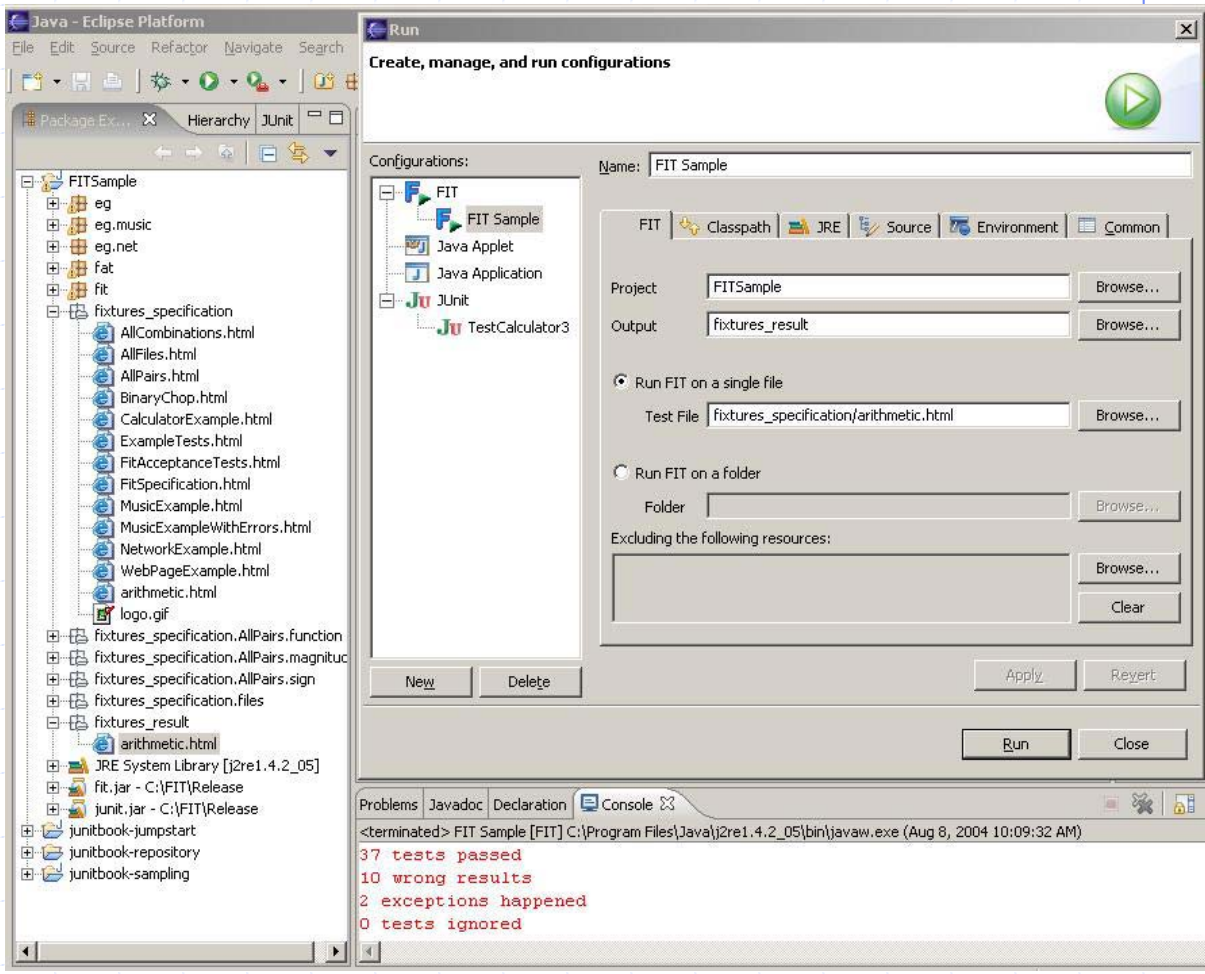
fit.ActionFixture		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	37

eg.music.Display					
title	artist	album	year	time()	track()
Scarlet Woman	Weather Report	Mysterious Traveller	1974	5.72	6 of 7
American Tango	Weather Report	Mysterious Traveller	1974	3.70	2 of 7

- Run FIT Fixtures
  - Within Eclipse: http://www4.ncsu.edu/~cho/articles/FitRunner.html
  - For .NET: http://storytestrunner.sourceforge.net/

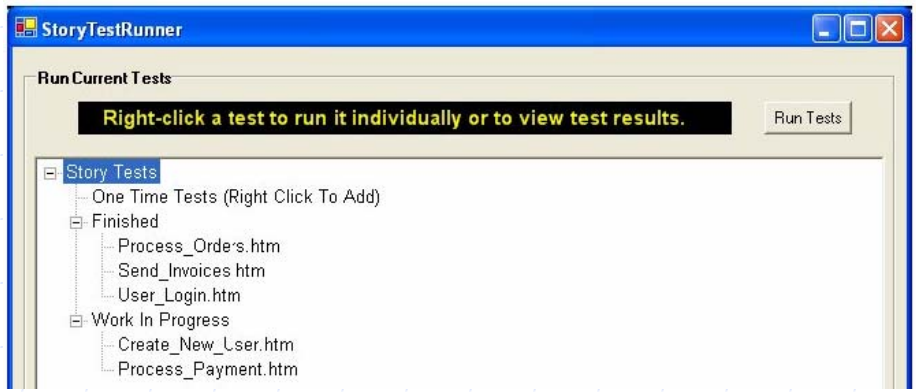
Eclipse plugin FitRunner (http://www4.ncsu.edu/~cho/articles/FitRunner.html)

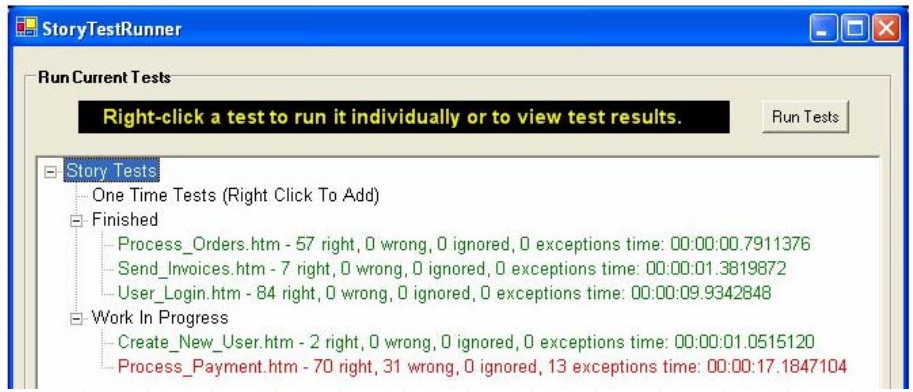
- Install plugin, add fit.jar and junit.jar to project's classpath
- Define input fixtures
- Define output directory
- Run them
- View the result



.NET StoryTestRunner (http://storytestrunner.sourceforge.net/)

- StoryTestRunner is a C# based application that runs .NET FIT fixtures.
  - Runs all story tests in specified locations (specified in an XML config file)
  - Ability to add additional tests for 'one-time execution'
  - Context menu lets tests be run individually.
  - Summary results are displayed in the tree control once a test is complete.
  - Detailed results are displayed using a context menu.





What is a Wiki?

- A minimalistic Content Management System
  - Everyone can change every page
  - Changes are visible immediately (but are under version control in case that you damage something)
  - There are abbreviations for often used HTML tags
  - Whenever a word is combined of several others (TestFirstDesign), it becomes a link to a new page. When the link is activated the first time, you can fill the (originally) empty page.
- First Wiki by Ward Cunningham: http://c2.com/cgi/wiki

What is FitNesse?

- An open source framework (under the GPL) created by Robert Martin et al.: http://fitnesse.org
  - Supports Java, .NET, C++
  - Combines FIT with a Wiki Web for writing the Fixtures (HTML tables)
  - Supports sub wikis for managing multiple projects
  - Supports virtual wikis for defining tests on the server (accessible by all) but for running them locally (within the development environment).
  - Versions pages, searches pages, supports simple refactorings (rename, move, delete page)
- A collaborative testing and documentation tool.
  - It provides a *very simple* way for teams to:
    - collaboratively create documents,
    - specify tests,
    - and run those tests and suites of those tests
- A web server:
  - It requires **no configuration or setup**.
  - Just run it and then direct your browser to the machine where it is running.
- A wiki:
  - You can *easily* create:
    - New Documents and pages.
    - Hyperlinks
    - Lists
    - Tables

How to use FitNesse?

- Install and start
- Define project on the FitNesse Wiki
- Write acceptance tests on the FitNesse Wiki.
- Write the glue code, the unit tests and the business logic in your favorite IDE.
- Execute the acceptance tests.
- But:
  - Not so tightly integrated into the automated build process, i.e. no test coverage computed out of the box,…
  - Not so tightly integrated into the IDE, i.e. no end-to-end debugging

Standard FitNesse Fixtures

- ColumnFixture: operates on a single object; each row loads a data structure (domain object) and then invokes functions upon it, often used for test object creation.
- RowEntryFixture: (like ColumnFixture) to add a bunch of data to a database, or to call a function over and over again with different arguments.
- RowFixture: to match all the rows from a simple query, independent of order. Each row is the data of a domain object, all rows are matched, missing and surplus rows are reported; often used to check the results of a query (where the query is built into the fixture or taken from a known static variable)
- ParametricRowFixture: (like RowFixture) additionally you can pass arguments into the RowFixture
- ActionFixture: to write a script that emulates a user interface
- CommandLineFixture: to execute shell commands in multiple threads
- HtmlFixture: to examine and navigate html pages
- SummaryFixture: displays a summary of all tests on a page; often added to TearDown
- TableFixture: lets you access the cells in a table by row and column
- TimedActionFixture: (like ActionFixture) additionally with visual feedback on how long certain functions take to execute

Sub Wikis and Test Suites

- A normal wiki is a collection of pages with a flat structure. All the pages are peers.
  - Add a top-level page simply by placing a WikiWord on an existing top-level page, and then clicking on the ?
- FitNesse allows you to create sub wikis. Each wiki page can be the parent of an entire new wiki.
  - Create a sub wiki page by the ^*SubPage* syntax, and then clicking on the ?
- Each wiki (and sub wiki) can have its own
  - ClassPath
  - PageHeader, PageFooter
  - SetUp, TearDown
  - SuiteSetup, SuiteTearDown
- Test Suites
  - A Test Suite executes all tests in the sub wiki (tree of pages)
  - SetUp and TearDown pages are invoked for each page of the suite.
  - To wrap an entire suite, define the operations on pages SuiteSetUp and SuiteTearDown

Virtual Wikis

- Fitnesse runs on the server ⇔ the developer works on his own machine.
  - Virtual Wikis allow to run local code still under development using a central set of shared test pages.
  - This is helpful in testing code before check-in.
- How to…
  - The developer starts FitNesse on his own machine, points one of his local pages to a sub-wiki on the global FitNesse server.
  - The entire sub-wiki from the global server then appears on the developer's local machine -- just as if the developer had written the pages there. But the pages are really still on the server.
  - Pressing the **Test** button on such a page, causes the test to be executed *locally*.
  - The developer can create ClassPath pages on *his* machine that allow the acceptance tests to be run in his local environment.
  - Thus, each developer can set up his own local environment and create a set of ClassPath pages that bind that environment to his wiki.
  - Then he can use Virtual Wiki to merge the remote acceptance tests to his local ClassPath environment.
- See http://fitnesse.org/FitNesse.MarkupVirtualWiki for details
  - Set page property VirtualWiki URL to include the pages of the sub wiki as children of the current page.

Testing the User Interface

- HtmlFixture (http://fitnesse.org/FitNesse.HtmlFixture)
  - is used to exercise and test web pages
  - permits to make assertions about the structure of a page and to navigate between pages
  - can fire java script, submit forms, "click" links, etc
  - lets you navigate this structure and name the elements as you go
  - At any given time some element in the structure is the "current" element and the commands apply to this element for the most part (called the "focus“).
  - Some commands are only legal if the focus has a certain type. For example, Submit applies only to anchor elements and form elements.
- HttpUnit (http://httpunit.sourceforge.net/)
  - emulates the relevant portions of browser behavior, including form submission, JavaScript, basic http authentication, cookies and automatic page redirection
  - allows Java test code to examine returned pages either as text, an XML DOM, or containers of forms, tables, and links
  - makes it easy to write Junit tests that very quickly verify the functioning of a web site
  - models the http protocol so you deal with request and response objects
- HtmlUnit (http://htmlunit.sourceforge.net/)
  - Similar to HttpUnit, but models the returned document so that you deal with pages and forms and tables
  - Supports http/https, POST/GET, partial JavaScript, basic http authentication, cookies, proxy server
  - Makes it easy to submit forms, click on buttons, walk the DOM model of the html document
- jWebUnit (http://jwebunit.sourceforge.net/)
  - Evolved from combined use of HttpUnit and Junit
  - provides a high-level API for navigating a web application combined with a set of assertions to verify the application's correctness (includes navigation via links, form entry and submission, validation of table contents)

Junit/HttpUnit ⇔ jWebJunit

```
package net.sourceforge.jwebunit.sample;

import junit.framework.TestCase;
import com.meterware.httpunit.WebResponse;
import com.meterware.httpunit.WebConversation;
import com.meterware.httpunit.WebForm;
import com.meterware.httpunit.WebRequest;

public class SearchExample extends TestCase {

   public void testSearch() throws Exception {
      WebConversation wc = new WebConversation();
      WebResponse resp = wc.getResponse(
        "http://www.google.com");
      WebForm form = resp.getForms()[0];
      form.setParameter("q", "HttpUnit");
      WebRequest req = form.getRequest("btnG");
      resp = wc.getResponse(req);
      assertNotNull(resp.getLinkWith("HttpUnit"));
      resp = resp.getLinkWith("HttpUnit").click();
      assertEquals(resp.getTitle(), "HttpUnit");
      assertNotNull(resp.getLinkWith("User's Manual"));
   }
}
```

```
package net.sourceforge.jwebunit.sample;

import net.sourceforge.jwebunit.WebTestCase;

public class JWebUnitSearchExample extends
      WebTestCase {

   public JWebUnitSearchExample(String name) {
      super(name);
   }

   public void setUp() {
      getTestContext().setBaseUrl("http://www.google.com");
   }

   public void testSearch() {
      beginAt("/");
      setFormElement("q", "httpunit");
      submit("btnG");
      clickLinkWithText("HttpUnit");
      assertTitleEquals("HttpUnit");
      assertLinkPresentWithText("User's Manual");
   }
}
```

References

- Kent Beck: Test-Driven Development: By Example, Addison-Wesley, 2002.
- David Astels: Test-Driven Development: A Practical Guide, Prentice Hall, 2003.
- Vincent Massol: Junit in Action, Manning Publications, 2003.
- J. B. Rainsberger: Junit Recipes, Manning Publications, 2004.
- Andrew Hunt, David Thomas: Pragmatic Unit Testing, Pragmatic Bookshelf, 2004.
- Johannes Link, Peter Fröhlich: Unit Tests mit Java, dpunkt.verlag, 2002.
- Stefan Roock: Akzeptanztests mit FIT und Fitnesse, http://www.stefanroock.de/downloads/Fitnesse.pdf
- Mark Windholz: Fit & Fitnesse, http://www.objectwind.com/present/FitNesse.htm

Online References

- FIT: http://fit.c2.com/
  - http://fit.c2.com/wiki.cgi?JavaDownloads
  - http://fit.c2.com/wiki.cgi?DotNetDownloads
- Fitnesse: http://fitnesse.org/
  - http://sourceforge.net/projects/fitnesse