Test-Driven Development

Dr. Christoph Steindl
Senior IT Architect and Method Exponent
Certified ScrumMaster
Christoph_Steindl@at.ibm.com
Copyright Notice and Usage Information

- This presentation contains material from books, talks and presentations of a number of people, partly updated, extended and rephrased.
- All rights remain with the original authors. None of these rights are directly or indirectly transferred to the author of this presentation.
- You may use or forward this presentation only after informing the author of this presentation by email. Doing otherwise is dishonest.
Agenda

- Good Engineering Practices
- What is Test-Driven Development?
- How to do TDD at the Unit-Test Level?
- How to do TDD at the Acceptance-Test Level?
- Project Automation – Continuous Integration
- Live Demo
- Questions and Answers
Good Engineering Practices

1. Having a source code control system
   - E.g. CVS
2. Doing code reviews before checking in code
   - Pair programming
   - Code reviews
3. Checking in at least daily (even if it leads to build failures)
4. Having daily builds
5. Doing unit tests
6. Having an automated test harness for unit tests
   - E.g. Junit
7. Using test-driven development at the automated test harness level
8. Doing continuous builds
   - Continuously after unit tested and checked in
   - E.g. CruiseControl
9. Doing acceptance test driven development
   - Apply the acceptance tests once code is checked in
   - E.g. FIT, Fitnesse
10. Refactoring
XP practices

- Coding circle
- Team circle
- Process circle
- Product circle

Software development practices at heart of XP

Less clearly defined organizational practices around them
What is Test-Driven Development?

In principle, it is just about writing the test before the program.

But in consequence, it leads the developer to

- **first** think about “how to use” the component (why do we need the component, what’s it for?)
- and **only then** about “how to implement”.

So, it’s a testing technique as well as a design technique

- It results into components that are easy to test.
- It results into components that are easy to enhance and adapt.

In the end, there is no code without a test.

The developer can tell at any time

- whether everything still works as it should, or
- what exactly does no longer work as it once did.
Motivation

- If you intend to test after you’ve developed the system, you won’t have the time for testing. Write the tests before the code!

- If things get complicated, you might fear that „the system“ doesn’t work. Execute the tests and get positive feedback (everything still works) or get pointed to the bit that does not / no longer work.

- If you’re overwhelmed by the complexity, you get frustrated. Start with the simplest thing and proceed in tiny steps!

- If you don’t have tests for the code, you shouldn’t use it / ship it. This can’t happen if you write the test first (so you reach better test coverage than with functional tests).

- If performance is only considered late, you won’t be able to just „add a little more performance“ to the system. Re-use unit tests for performance tests even during development and don’t start with performance tests late in the project!
Red – Green – Refactor

Red
- Write a little test that doesn’t work (and perhaps doesn’t even compile at first).

Green
- Make the test work quickly (committing whatever sins necessary)

Refactor
- Eliminate all of the duplication created in merely getting the test to work, improve the design.
Why?

The test is the executable specification.
- You start thinking about the goal first, then about the possible implementations.
- You understand the program’s behavior by looking at the tests. The tests tell you more than just an API description, they show the dynamics, how to use the API.

You develop just enough.
- You get to the goal as quick as possible.
- You don’t develop unnecessary code.
- There is no code without a test.
- There is no test without a user requirement.

Once you get one test working, you know it is working now and forever.
- You use the tests as regression tests.

The tests give us the courage to refactor.
- You can prove that everything still works after the refactoring by simply executing the tests.

It’s more fun that way, it reduces fear.
How?

Don’t start with objects (or design, or ...), start with a test.
- Write new code only if an automated test has failed.
- First think of the goal, the required functionality.
- Then think of the perfect interface for that operation (from the outside, black-box view).

Run the test often – very often.
- To determine whether you’ve reached the goal.
- To catch any bugs that have crawled back in.

Make little steps (during coding and refactoring)
- So little that you feel comfortable with them.
- Make them larger if you feel.
- Make them smaller if you don’t proceed with your expected velocity.
Use Tools

- Framework for automating the unit tests
  - E.g. Junit

- Integrated development environment
  - For writing tests, using auto-completion and generation of missing code.
  - For running the tests
  - For refactoring
  - E.g. Eclipse

- Build environment
  - For executing tests automatically and during the build process
  - For computing code coverage
  - For generating test reports
  - E.g. Maven
Iterative Software Development

- At the start of the iteration, the customer explains the expected functionality to the team. The customer prioritizes for business value and urgency, the team estimates the effort and cost.
- Then the team brainstorms the necessary tasks to implement the functionality, details the estimation and team members select their tasks.
- During the iteration, the team holds short 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

- Don’t wait until the software has been written in order to start with 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 in order to verify 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

Validates the requirements

- Is that really what we need? (“Do we build the right system?”, not just “Do we build the system in the right way?”)
- Typically, validation only occurs at the end of the project (where it’s too late).

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
Traditional Approaches to Acceptance Testing

- Manual Acceptance Testing
  - User exercises the system manually using his creativity
  - Disadvantages:
    - The developers don't know the goal, the tests that the system has to pass
    - this approach does not support TDD
    - 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, “the eye sees what it wants to see”)
    - There can be big arguments about the pass/fail decision

- Acceptance Testing with Capture & Replay (e.g. at the GUI level)
  - Tool captures events (e.g. mouse, keyboard) in modifiable script (e.g. hopefully abstracting from screen coordinates to GUI objects)
  - Disadvantages:
    - System (and GUI) has to exist, so this approach does not support TDD
    - Tools are expensive
    - Tests are brittle, have to be re-captured if the system changes
Agile Approach to Acceptance Testing

Framework for automating the functional tests

- E.g. FIT, FitNesse

Advantages:

- Easy for the user to describe the requirements themselves (no programming, just text)
- Easy for the developer to glue the requirements to the business logic
- Tests are written before the code, so this approach supports TDD
- Inexpensive because the frameworks are open source, execution of the tests can be automated.
- The developer has a clear goal to achieve!

Disadvantages:

- 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 behavior 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

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

---

copyright © Christoph Steindl  
Test-Driven Development 18
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

(c) Christoph Steindl
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 (TestDrivenDevelopment), 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 by a click on the web page.
- See the results of executing the tests on the web page.

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

TDD at the Acceptance-Test Level

(c) Christoph Steindl
Test-Driven Development 23
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 $^\text{SubPage}$ 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

(c) Christoph Steindl Test-Driven Development 25
Virtual Wikis

- FitNesse runs on the server \(\Leftrightarrow\) 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](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](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 JavaScript, 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/](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/](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

  - 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

```java
package net.sourceforge.jwebunit.sample;
import junit.framework.TestCase;
import com.meterware.httpunitWebResponse;
import com.meterware.httpunitWebConversation;
import com.meterware.httpunitWebForm;
import com.meterware.httpunitWebRequest;
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.jwebunitWebTestCase;
public class JWebUnitSearchExample extends WebTestCase {

    public void testSearch() {
        getTestContext().setBaseUrl("http://www.google.com");
        beginAt("/");
        setFormElement("q", "HttpUnit");
        submit("btnG");
        clickLinkWithText("HttpUnit");
        assertEquals("HttpUnit", getTitle());
        assertLinkPresentWithText("User's Manual");
    }
}
```
Project Automation

- Building # Compiling
- Integrate early
- Integrate often
- Make the build CRISP
- Automate (e.g. with Ant or Maven)
- Verify the build with tests
- Trigger build by schedule or event (like modified sources in the repository)
- Build continuously (e.g. with CruiseControl)

(c) Christoph Steindl

Test-Driven

http://www.pragmaticprogrammer.com/sk/auto

CRISP Builds
- Complete
- Repeatable
- Informative
- Schedulable
- Portable

Automation Checklist
- Create a one-step build process
- Build on a frequent schedule
- Write branch and release scripts
- Create an installer/deployer
- Monitor builds and applications
- Review and revise

Pragmatic Project Automation Summary Road Map

One-Step Builds
- Compile
- Test
- (On Command)

Scheduled Builds
- Checkout
- Compile & Test
- Email
- (Hourly)

Push-Button Releases
- Branch
- Test
- Package
- Release
- (Weekly)

Installation & Deployment
- Install
- (Monthly)

Monitoring
- Call/Phone / Pager
- Log4j
- Visual Devices
- RSS
- (Continuously)
Prerequisites and Benefits

Prerequisites:

- Keep a single place where all the source code lives and where anyone can obtain the current sources from (and previous versions).
- Automate the build process so that anyone can use a single command to build the system from the sources.
- Automate the testing so that you can run a good suite of tests on the system at any time with a single command.
- Make sure anyone can get a current executable which you are confident is the best executable so far.

Benefits

- **Faster Feedback**: Most integration bugs manifest themselves the same day they were introduced.
- **Better Context**: You typically know where to look for the reason of the problem (or you don’t add the buggy feature to the product).
CruiseControl

- Is a continuous integration server
  - Runs in the background
  - Runs the “build cycle” (at configurable intervals):
    - Reload configuration file (in case of changes)
    - Determine if a build is necessary (by querying the source control system)
    - Build
      - Check out the project files
      - Compile the sources
      - Execute the tests
      - Package the deliverables
    - Create a log file
    - Send notifications of success or failure
- Presents the build results as a web page (by the “build results JSP” which can be deployed into a Tomcat server).
- Integrates with Junit, Ant, Maven and CVS (and other source control systems).
How to

On the developer’s machine (as usually, nothing new here)
- Develop the system
- Check in code

On the build server
- Create a directory for the builds (e.g. "mkdir c:\builds")
- Change into the directory (e.g. "cd c:\builds")
- Check out the project (e.g. "cvs co triangle")
- Create a directory for the build results (optionally on another server, e.g. "mkdir c:\builds\buildresults")
  - this is where Maven will deploy the results of the build
- Create a directory for the log files (e.g. "mkdir c:\builds\cc-logs")
  - this is where CruiseControl will create its log files
- Let maven create the cruisecontrol.xml (e.g. "maven cruisecontrol")
- Add additional publishers (e.g. for RSS feeds)
- Let maven start CruiseControl (e.g. "maven cruisecontrol:run")

The source control system acts as the intermediary between the developer and the build server.
- Relax and wait for email notifications to arrive.
Check out

Continuous Integration Server

Check in

Source Control System

Development Machine

C:\development

checkout
Maven
CruiseControl
RSS-Feed

C:\builds
triangle
buildresults
cc-logs
buildstatus

C:\develop
triangle

:pserver:anonymous@localhost:/sandbox
<?xml version="1.0" encoding="UTF-8"?>

<cruisecontrol>
  <project name="testing-triangle">
    <bootstrappers>
      <currentbuildstatusbootstrapper
        file="C:\builds/cc-logs/currentbuildstatus.txt">
      </currentbuildstatusbootstrapper>
    </bootstrappers>
  </project>
  <modificationset>
    <cvs localWorkingCopy="C:\builds/triangle"
      cvsroot=":pserver:anonymous@localhost:/sandbox">
    </cvs>
  </modificationset>
</cruisecontrol>

- Name
- Bootstrappers
- Modification set
CruiseControl.xml (2/3)

```xml
<schedule interval="60">
  <maven goal="scm:update-project|clean test|site site:fsdeploy"
    projectfile="C:\builds/triangle/project.xml"
    mavenscript="C:\PROGRA~1\APACHE~1\MAVEN1~1.0/bin/maven">
  </maven>
</schedule>

<log dir="C:\builds/cc-logs/testing-triangle">
  <merge dir="C:\builds/triangle/target/test-reports">
  </merge>
</log>
```

- Schedule interval
- Maven for build
- Log

Continuous Integration
<publishers>
  <currentbuildstatuspublisher
    file="C:\builds/cc-logs/currentbuildstatus.txt">
  </currentbuildstatuspublisher>
  <htmlemail logdir="C:\builds/cc-logs/testing-triangle"
    mailhost="localhost"
    css="C:\cruisecontrol-2.1.6/reporting/jsp/css/cruisecontrol.css"
    subjectprefix="[BUILD]" returnaddress="christoph_steindl@at.ibm.com"
    defaultsuffix="@localhost"
    xsldir="C:\cruisecontrol-2.1.6/reporting/jsp/xsl">
    <map address="christoph_Steindl@at.ibm.com" alias="steindl">
    </map>
    <failure address="christoph_steindl@at.ibm.com">
    </failure>
  </htmlemail>
  <XSLTLogPublisher directory="c:\builds\buildstatus",
    outfilename="trianglebuildstatus.rss" xsltfile="buildstatus.xsl" />
</publishers>
Maven / CruiseControl files

- **Maven’s „project.xml“ (under version control)**
  - Deploy the build result with the target „site:fsdeploy“ (file system deployment)
    <siteDirectory>c:\builds\buildresults</siteDirectory>
  - You can also deploy the results to a web server (with „site:deploy“)

- **Maven’s „build.properties“ (specific for each developer / machine)**
  - maven.cruisecontrol.checkout.dir=C:\builds
  - maven.cruisecontrol.logs.dir=c:\builds\cc-logs

- **CruiseControl’s „cruisecontrol.xml“**
  - Generated by Maven (maven cruisecontrol)
  - Added RSS publisher (under <publishers>):
    <XSLTLogPublisher directory="c:\builds\buildstatus"
      outfilename="trianglebuildstatus.rss" xsltfile="buildstatus.xsl" />

(c) Christoph Steindl  Test-Driven Development 37
maven.username=Administrator

#Location of cruise control installation - for finding the reporting directory
maven.cruisecontrol.home=C:\cruisecontrol-2.1.6

#Where to checkout/update code temporarily. Default value is ${basedir}/checkout.
maven.cruisecontrol.checkout.dir=C:\builds

#Seconds in between builds for this project. Default value is 300.
maven.cruisecontrol.schedule.interval=60

#Default value is scm:update-project|clean test|site:deploy.
maven.cruisecontrol.goals=scm:update-project|clean test|site site:fsdeploy

#Default value is the domain name of the first email in ${pom.build.nagEmailAddress}.
maven.cruisecontrol.mail.defaultsuffix=@localhost

#Default value is ${maven.build.dir}/cc-logs.
maven.cruisecontrol.logs.dir=c:\\builds\\cc-logs

#Default value is localhost.
#maven.cruisecontrol.mail.host=
#Default value is [BUILD].
#maven.cruisecontrol.mail.subjectprefix=
#Config file to update/create. Default value is ${basedir}/cruisecontrol.xml.
#maven.cruisecontrol.config=
#Template file to use in generating the cruisecontrol.xml file.
#maven.cruisecontrol.template=

- User name
- Home of CruiseControl
- Schedule interval
- Goals for build
- Mail suffix
eXtreme Feedback Devices (XFDs)

- XFDs are inexpensive to build and operate, they add fun and color to the workspace, and most important they are effective in providing the team with feedback on key items and getting them to act upon it.
- XFDs help you to achieve and maintain focus on what is most important to the organization at any given time.
- By making the feedback mechanism fun you help to draw attention to it.
- By making the feedback broadly available you send out a signal that the objectives are important and that many people care about, and depend upon, their achievement.
Quote of the day

„By the time it was released, Microsoft Windows NT 3.0 consisted of 5.6 million lines of code spread across 40,000 source files. A complete build took as many as 19 hours on several machines, but the NT development team still managed to build every day (Zachary, 1994). Far from being a nuisance, the NT team attributed much of its success on that huge project to their daily builds.“

Steve McConnell

http://www.stevemcconnell.com/ieeesoftware/bp04.htm
References

- Frank Westphal: Testgetriebene Entwicklung, dpunkt.verlag, 2004
- 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
- Martin Fowler: Continuous Integration 
  http://www.martinfowler.com/articles/continuousIntegration.html
- Continuous Integration with CruiseControl.NET and Draco.NET 
  http://www.theserverside.net/articles/showarticle.tss?id=ContinuousIntegration
Online References

Unit Testing:
- xUnit: http://www.xprogramming.com/software.htm
- JUnit: http://junit.org
- NUnit: http://sourceforge.net/projects/nunit/

Automated Build:
- Ant: http://ant.apache.org/
- Grand: http://www.ggtools.net/grand/
- NAnt: http://nant.sourceforge.net/

Integrated Development Environment:
- Eclipse: http://eclipse.org
- SharpDevelop: http://www.icsharpcode.net/OpenSource/SD/

Build Environment:
- Maven: http://maven.apache.org/
Online References

FIT: http://fit.c2.com/

Fitnesse: http://fitnesse.org/
  - http://sourceforge.net/projects/fitnesse

Continuous Build Environment:
  - CruiseControl: http://cruisecontrol.sourceforge.net/
  - CruiseControl.NET: http://ccnet.thoughtworks.com/
  - DamageControl: http://damagecontrol.codehaus.org/
  - Draco.NET: http://draconet.sourceforge.net/
  - IntegrationGuard: http://iguard.sourceforge.net/
  - Gump: see Google
Online References

TDD
- http://c2.com/cgi/wiki?TestFirstDesign
- http://www.testdriven.com/
- http://www.objectmentor.com/writeUps/TestDrivenDevelopment
- „Aim, Fire“:
  http://www.computer.org/software/homepage/2001/05Design/index.htm

Continuous Integration:
- http://c2.com/cgi/wiki?ContinuousIntegration
- Daily Build and Smoke Test:
  http://www.stevemcconnell.com/ieeesoftware/bp04.htm
- XFDs:
  http://www.developertesting.com/managed_developer_testing/000036.html
- Continuous Integration WebLog:
  http://weblogs.asp.net/mnissen/category/5211.aspx
- Automated Continuous Integration and the Ambient Orb