

# Agile Development and Testing

PyCon '06, Feb 23rd, Addison, TX

**Grig Gheorghiu, [grig@gheorghiu.net](mailto:grig@gheorghiu.net)**

**C. Titus Brown, [titus@caltech.edu](mailto:titus@caltech.edu)**

<http://agiletesting.blogspot.com/>

<http://www.advogato.org/person/titus/>

**All of our code, all of our tests, and links to buildbot and other external resources will be made available on <http://agile.idyll.org/> by the end of the conference.**

## Introduction

1. what is agile?
  - the "testing pyramid"
  - functionality implemented as "stories"
  - automated tests
  - collaboration/"whole team" approach
  - "tracer bullet" development
  - "Test-Enhanced Development"
2. lessons learned:
  - continuous integration is key
  - remote pair programming works well
  - whole team approach to both development and testing
  - no code "thrown over the wall" to QA

## MailOnnaStick

1. Overview: browse, search, and comment across multiple sources of e-mail.
2. Technologies:
  - CherryPy
  - Durus
  - Commentary
  - jwzthreading
  - py.lib logging, xml generation
  - quixote.html HTML escaping code
3. Implementation timeline
  - started with simple Web interface to browse mailboxes
  - added search function
  - added commenting function

- refactored to allow for multiple mailboxes per mail source (Mail dir, IMAP);
  - currently refactoring to allow for better database storage
  - next step: better choice? or use? of database.
4. Basic architecture: mos, mosweb, mailsearch
  5. Demo.

## Using Trac and Subversion for collaborative development

1. Trac
  - Wiki
  - defect tracking system
  - source browser
  - roadmap
  - can link between everything
  - timeline is very useful
2. Subversion
  - source code control system similar to CVS but more advanced
  - standard layout: trunk, branch, tags
  - lots of tools available
  - some sort of code management system is essential. period.
  - SvnReporter does a nice job of e-mail notifications, etc.

## Unit testing:

1. theory
  - completely automated tests that should run *quickly*;
  - test small, individual "units" of function;
  - generally test functionality of a specific interface or function
  - generally don't test full functional "path" through code
  - may take some setup (load/reset database; build mock object; etc.) in advance of test.
2. using nose: unit test structure
  - easy to build ad hoc unit tests, but nose gives you a *structure* within which you can easily select subsets of tests, add tests, and see what tests didn't pass.
  - based on concepts from py.test; written by Jason Pellerin.
  - some examples
  - built on top of unittest.py, so can integrate into setup.py
  - path handling is a bitch...
3. running subtests and doing timing stuff
  - very useful command line option to select a subset of tests
  - hacking nose to do timing analysis/display.
4. lessons learned:
  - the easier to run the better
  - unit tests serve as explicit interface documentation
  - writing unit tests for other people's code is a good way to learn the code!

## FitNesse Acceptance Testing

1. FitNesse
  - more user friendly variant of Ward Cunningham's FIT framework
  - "business facing" or "customer facing" tests, as opposed to "code facing" tests (e.g. unit tests)
  - tests are expressed as stories ("storytests")
  - higher level than unit tests
  - FitNesse tests make sure you "write the right code"
  - unit tests make sure you "write the code right"
  - wiki format encourages collaboration
  - Fit/FitNesse brings together business customers, developers and testers, and it forces them to focus on the core business rules of the application.
  - James Shore: "Done right, FIT fades into the background"
2. Writing FitNesse tests
  - PyFIT is Python port of FIT/FitNesse
  - tests are written in tabular format, with inputs and expected outputs
  - fixtures are a thin layer of glue code that tie the test tables to the application
  - tests can be executed from both the wiki and the command line.
3. lessons learned:
  - acceptance tests written with FitNesse can be seen as an alternative GUI into the business logic of the applications
  - GUI tests are fragile and need frequent changes to keep up with changes in the UI

## Regression testing with TextTest

1. TextTest is a tool for "behavior based" acceptance testing
  - generates and then compares log files to known "gold standard", displays differences.
  - documentation on project's home page is plentiful, but lacks howtos
2. Lessons learned:
  - log at multiple levels to separate, discrete log files
  - we're not really using TextTest right.

twill functional tests; coverage analysis; profiling.

1. theory: scripting remote HTTP.
2. using twill to browse other Web sites.
  - twill scripts are simple to write (examples)
  - interactive browsing demo (example)
  - twill tests are just scripts that should not fail
  - extending twill with Python is easy (example)
  - other packages: mechanize, webtest, webunit, zope.testbrowser, mechanoid.
3. writing automated twill tests: setup/teardown and WSGI intercept (FIGURE)
  - setup/teardown for a Web server is annoying.
  - wsgi\_intercept lets you run everything in a single process: coverage and profiling are made *much* easier.

4. twill tests can provide extensive code coverage to your Web code.
5. twill can also automate a particular user path through your site, letting you profile and enhance specific targeted site functions.

### Testing AJAX Web sites with Selenium

1. functional/acceptance testing at the user interface level for Web/AJAX application
  - uses an actual browser driven via JavaScript!
  - needs to be deployed on the same server as the app under test for JS security reasons
2. Writing Selenium tests
  - tests written as HTML tables
  - tables are small programs, with actions etc.
  - tools exist for recording tests: Selenium IDE, XPath Checker and XPather Firefox extensions
  - also a "driven" mode where an external program can drive Selenium via XML-RPC
  - Selenium tests can be added to continuous integration process
3. Lessons learned
  - Selenium tests are fairly brittle, like all GUI tests
  - Not fun to write (although IDEs help!)
  - Only known way for mankind to test AJAX!

### doctest

1. testing with doctest
  - "literate testing" or "executable documentation"
  - unit tests expressed as stories with context
  - many projects generate documentation from doctests (Django, Zope3)
  - epydoc makes it trivial to show!
  - test list -- set of unit tests for a given module;
  - test map -- set of unit tests functions that exercise a given application function/method
  - easy to generate automatically
2. Lessons learned
  - unit test duplication (unit tests, doctests, etc.) is not necessarily a bad thing

### buildbot I: continuous integration

1. Continuous integration
  - automate time consuming tasks, get immediate *public* feedback about results of changes
  - the more often you build & test your software, the better.
2. Using buildbot
  - written in Twisted; hard to configure but works really well
  - can be deployed behind Apache for access control purposes
  - master/slave on multiple servers

- run slaves with as many different hardware/software combos as you can
  - run as many types of tests as possible
  - (the more aspects of the app you cover, the better protected you are)
3. Lesson learned
- running unit/acceptance tests as separate user/different environment will uncover *all kinds* of environment specific issues. trust us.

buildbot II: automating everything, and recording everything.

1. automating Selenium UI tests with VNC and Firefox.
2. collating logs and test results.
  - provides a useful, detailed *record* of where things succeeded, failed.
3. recording logs and test results across runs: Web, billboard.
  - refer back to prior runs
  - billboard: get an integrated view of all of the tests

Wrap up & Conclusions

1. Holistic testing is the way to go. No one test type does it all...
  - unit testing of basic code
  - Functional/acceptance testing of business/user logic
  - regression testing
2. Make it as easy as humanly possible to run all these types of tests automatically, **or they will not get run**