(Web) Testing for Pythonistas

C. Titus Brown
Grig Gheorghiu

This tutorial is for you

Ask questions.

Ask about specifics.

Demand demos.

(Grig and I are secure in our abilities)

Rough outline

Why test?

Testing strategies.

Low-level testing tool overview.

UI and acceptance tests.

Continuous integration.

Open Q&A.

Online resources

Code etc. will be posted after tutorial.

'testing-in-python' mailing list.

Us, in person & via e-mail.

O'Reilly E-book, "Functional Web Testing"

Why write automated tests?

1) Because you want your code to work.

Why write automated tests?

1) Because you want your code to work.
2) Because you want your code to meet customer expectations.

Why write automated tests?

1) Because you want your code to work.
2) Because you want your code to meet customer expectations.
3) Because you want to simplify your (programming) life.

Why write automated tests?

1) Because you want your code to work.
2) Because you want your code to meet customer expectations.
3) Because you want to simplify your (programming) life.
4) Because you often over- or under-design your code.

Goal: become "test infected"

Integrate "the testing way" into your daily life.

It is now actually psychologically uncomfortable for me to write code that is difficult to test.

(This speaks to tools as well.)

Some test guidelines

Keep your testing infrastructure as simple and stupid as possible.

Testing should help you write & debug your other code, not become a major part of the codebase in and of itself...

Always be ready to throw out your test code!

Some test guidelines

Start small.

Build tests incrementally.

Smoke tests are absurdly useful.

Test simply, at many levels.

Focus on actual problem areas (existing bugs).

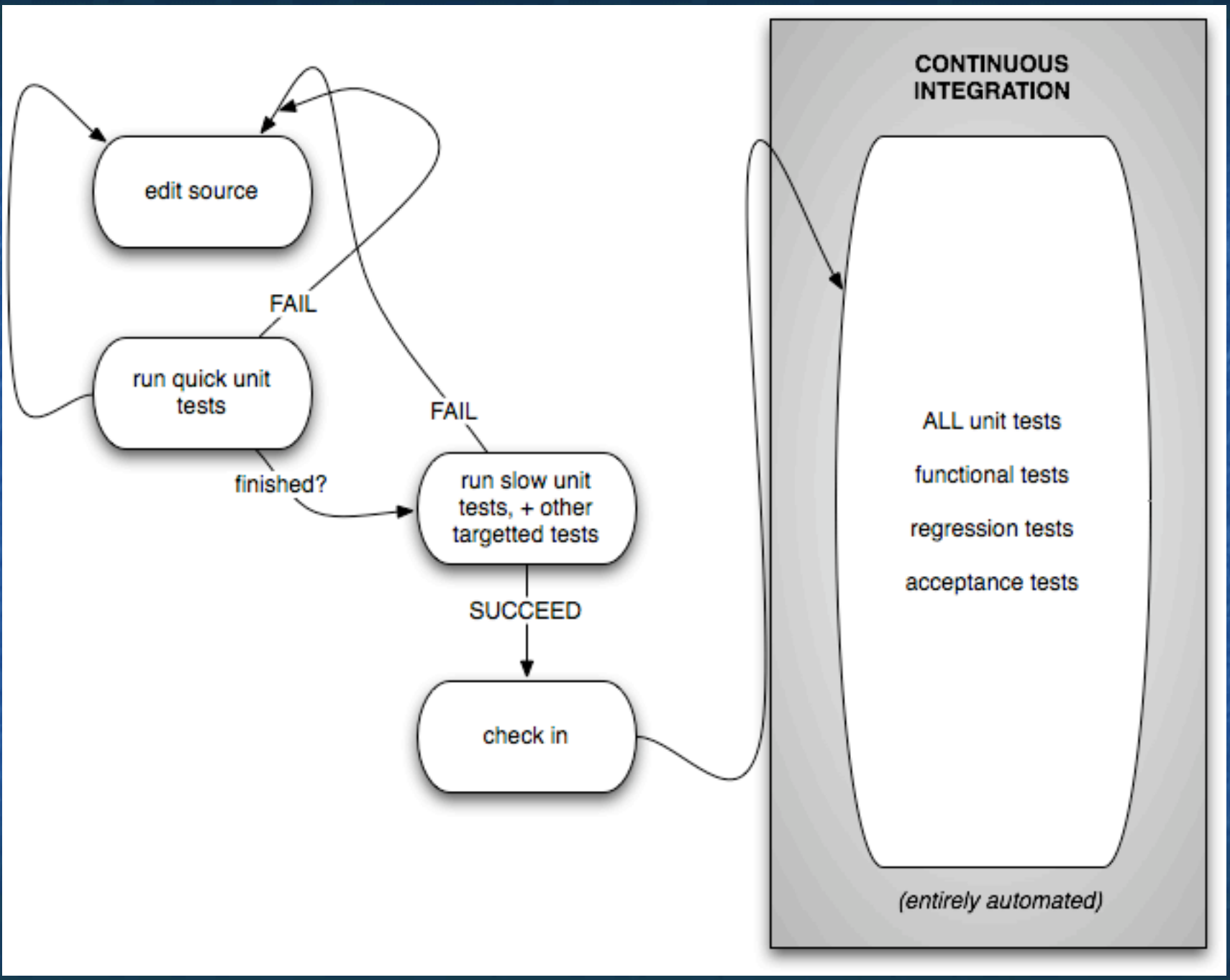
Using testing as part of your process

Use tests personally, even if other people don't.

Manage up and across.

Automate what can be easily automated, nothing more. (Corollary: plan for testability)

Start small and KISS.


edit source
FAIL
run quick unit tests
FAIL
finished?
run slow unit tests, + other targetted tests
SUCCEED
check in
CONTINUOUS INTEGRATION
ALL unit tests
functional tests
regression tests
acceptance tests
(entirely automated)

Testing Taxonomy

(only somewhat useful)

Unit tests

Functional tests

Regression tests

User interface tests

Acceptance tests

Integration tests

Continuous integration

Performance tests

Testing Strategies

"TDD" vs "SDT" or "TED"

Test Driven Development: write tests first

"TDD" vs "SDT" or "TED"

Test Driven Development: write tests first

Stupidity Driven Testing: write tests after you do something stupid, to make sure you never make the same mistake again.

(a.k.a. "Test Enhanced Development" ;)

Constrain your code

Document your expectations, internally and externally.

Use assert more. (internal)

Write unit, functional, and regression tests in terms of expectations. (external)

"This function better do X, or else..."

"Tracer bullet" development

Write an end-to-end test
(e.g. Web browser -> database & back)

1) If it doesn't work, go no further!
2) Exercises your setup and teardown code.
3) Gives you a base from which to expand.

Build a "test umbrella"

One command -> all tests.

1) Integrated reporting
2) Ease of use and "startup"
3) No memory required

(nose, py.test)

Using regression tests

Before you can ask why did that change?

...you must be able to know

Did something change?

Refactoring

Incrementally change code while keeping the codebase working.

Refactoring becomes stress free with reasonably thorough testing.

Code coverage

Use code coverage (line coverage) to target new tests.

People will tell you that code coverage is not a good metric.

They're right, in theory.

But if you're not running every line of code in your application at least once during your tests, you can't know if that line works.

Continuous integration

Set your tests up to run automatically with a single click, in several environments.

Yes, this helps you figure out if your code still works.

But it also helps you in many less obvious ways:

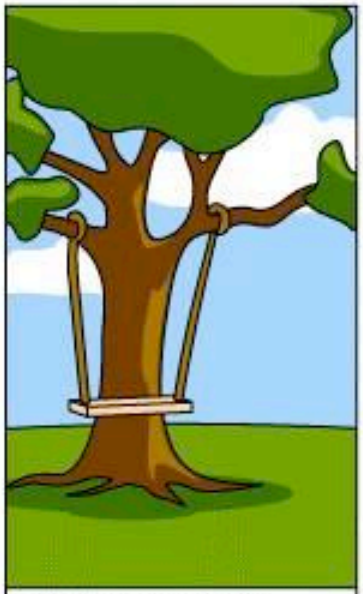
1) Did you introduce an environment dependency?
2) Did you add a new package/library requirement?
3) Environment & package requirements become explicit
4) Indisputable and impersonal evidence that something is broken!

Integrating testing into your customer interaction process

Rationale: the point of programming is (usually) to deliver something that a customer actually wants. (...not what they think they want...)

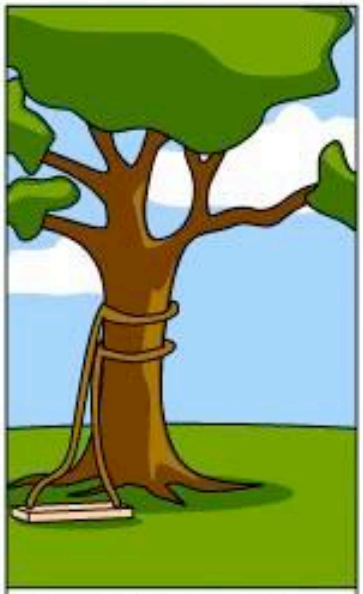
This is a communication problem.

How the customer explained it



How the Project Leader understood it

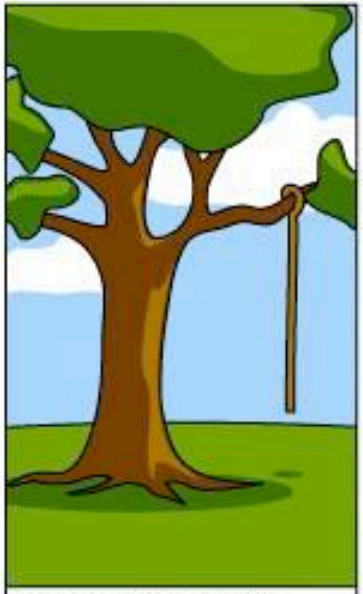
How the Analyst designed it



How the Programmer wrote it

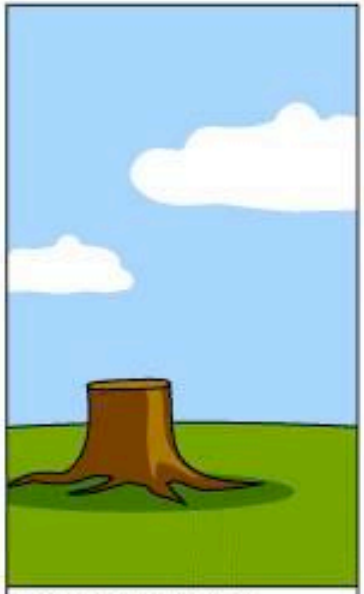
How the Business Consultant described it

How the project was documented

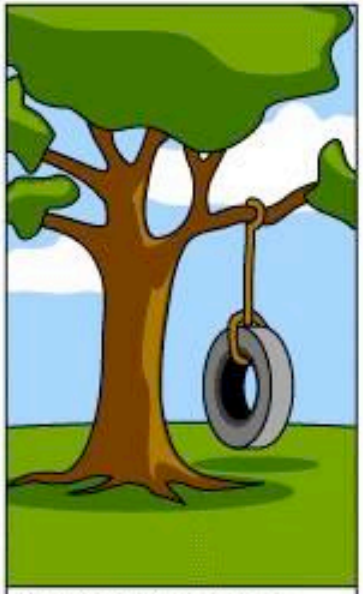


What operations installed

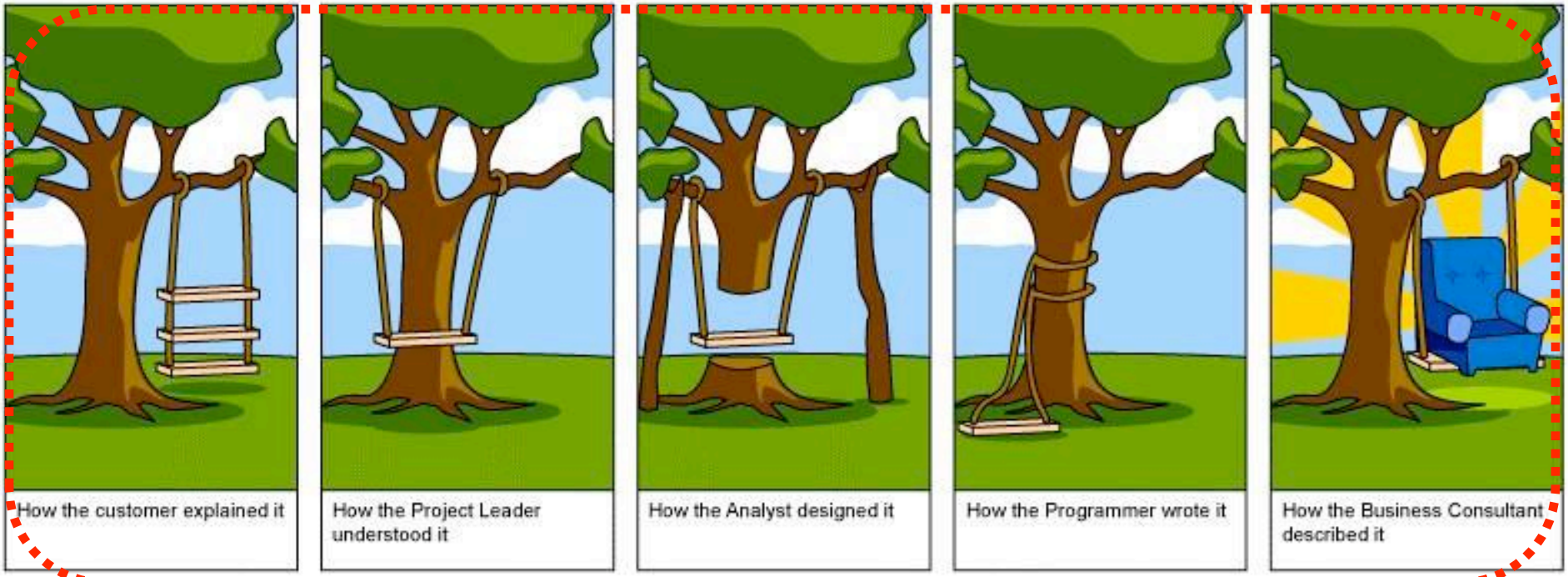
How the customer was billed

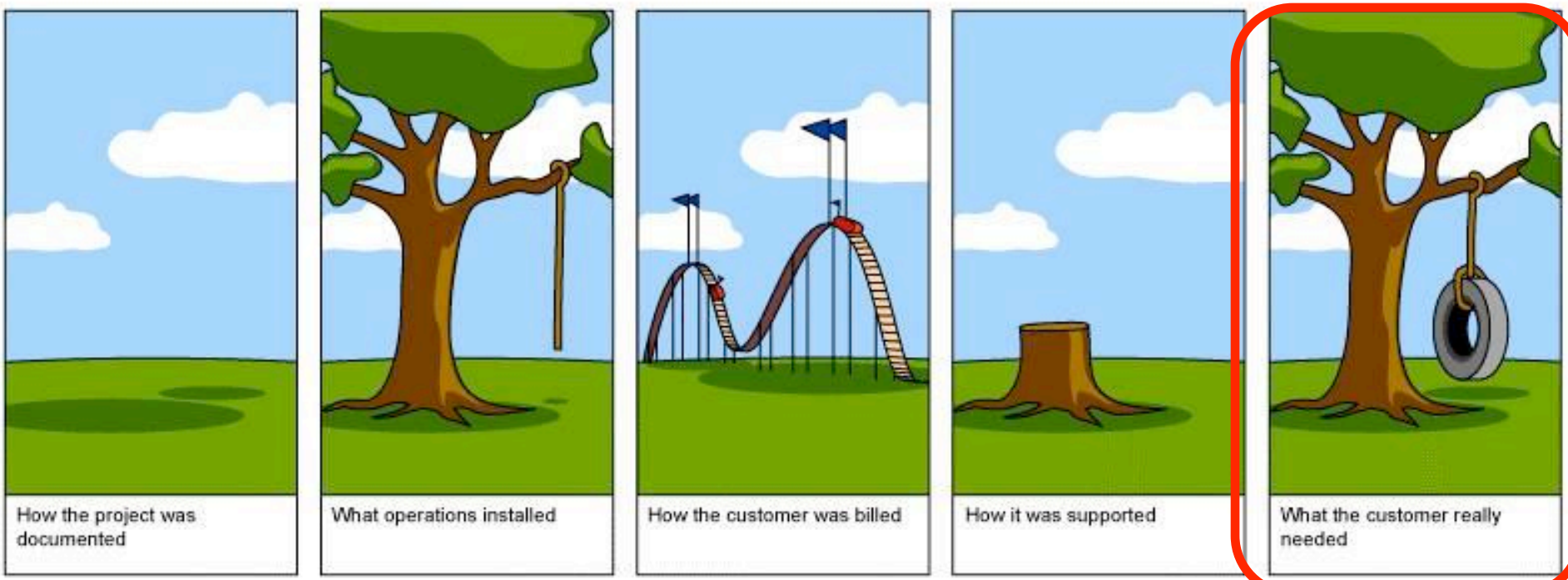


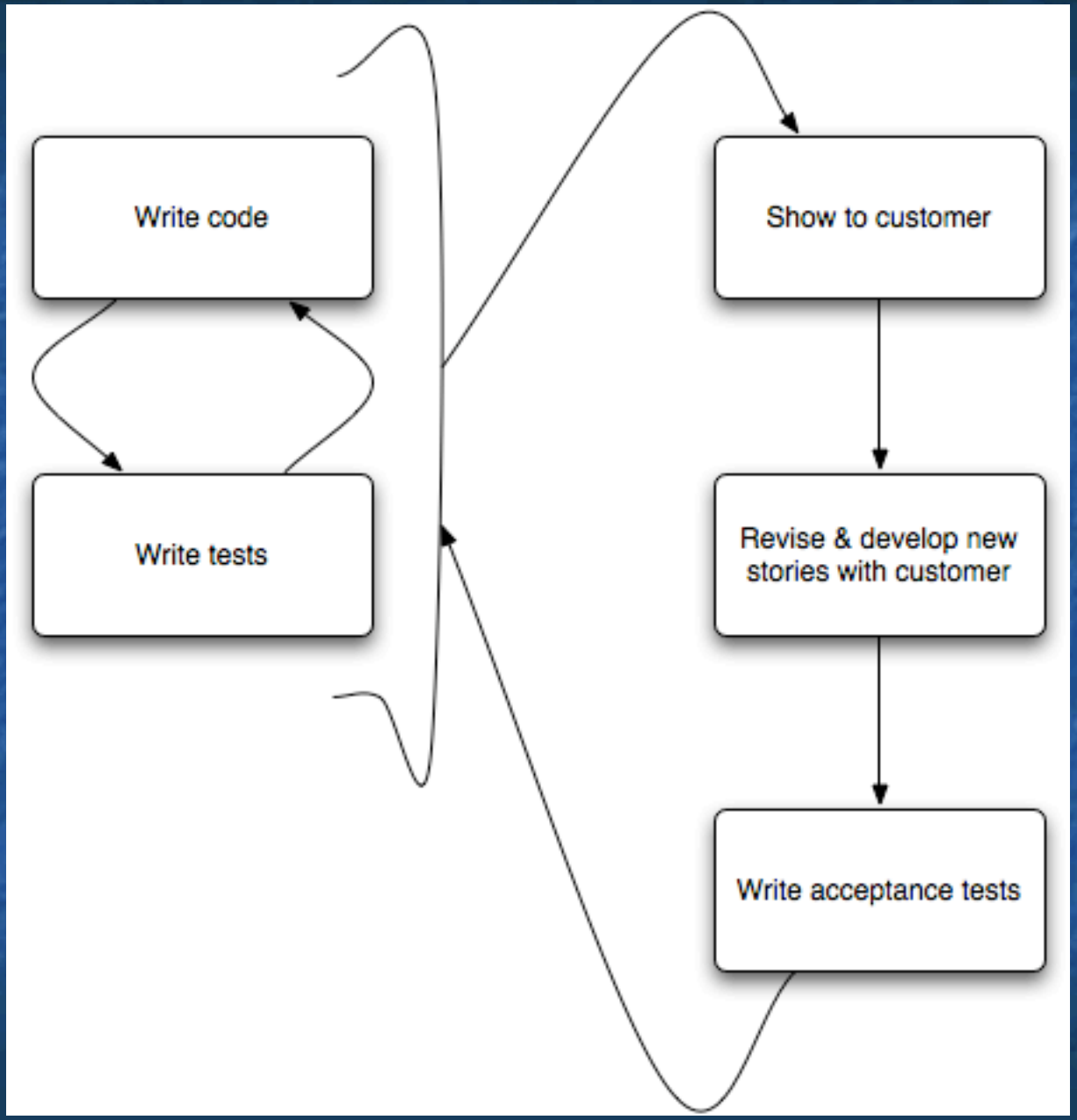
How it was supported



What the customer really needed


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it


How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed


Write code
Write tests
Show to customer
Revise & develop new stories with customer
Write acceptance tests

Summary

Testing should be an integral part of your process.

If it's not working for you, don't force it but don't give up, either!

Address real problems.

Simplify.

There is no try, only do.

"Low-level" testing tools

nose -- unit test discovery & execution

twill -- functional Web testing
(+ extensions)

wsgi_intercept -- talk directly to WSGI apps
(hard to explain...)

scotch -- record & replay Web traffic

figleaf -- code coverage analysis

Testing tools for programmers must be simple, easy to deploy & use, and configurable. Otherwise people won't use them.

Functional Web testing with twill

go http://www.google.com/

formvalue 1 q “google query statistics”
submit
show

forms, cookies, redirects, http basic auth, link following, link checking, http code assertions, test for text presence/nonpresence, tidy checking, etc.

no JavaScript support ;(

twill is Python

```
from twill.commands import *
go('http://www.google.com/')
showforms()
formvalue('1', 'q', "google query statistics")
submit()
show()
```

All base twill commands directly accessible from Python. Additionally, a "nice" wrapper around mechanize is available.

wsgi_intercept lets twill talk directly to WSGI apps.

```
app = get_wsgi_app()

import twill
twill.add_wsgi_intercept('localhost', 80, lambda: app)

twill.commands.go('http://localhost/')
...
twill.remove_wsgi_intercept('http://localhost/')
```

This is fairly close in concept to paste.fixture, but you can use the same script for testing a direct WSGI connection as you can for testing your whole "Web stack" (server + middleware + app).

(Will show you demo)

scotch lets you record & replay WSGI data, and generate twill scripts too.

```
app = get_wsgi_app()

import scotch.recorder
recorder = scotch.recorder.Recorder(app)
serve_wsgi(recorder)
...

app = get_wsgi_app()
for record in recorder.record_holder:
    print record.replay(app)
```

figleaf is a code coverage recording tool.

```
import figleaf
figleaf.start()
...
figleaf.stop()
figleaf.write_coverage('.figleaf')
```

Intended for programmatic use; can take coverage from multiple runs/deployments/platforms, and intersect/union results. Can retrieve from running Web server.

(Works fine as a replacement for coverage.py, too)