PyCon 2008,
Chicago, USA.

Jonathan Hartley

Getting Started with Test Driven Development (TDD) in Python

(aka TDD: Brooks' Silver Bullet?)

tartley@tartley.com
http://tartley.com

Aims of this talk

To demystify the process of getting started with test-driven development (TDD) in Python.

I'll create a small test-driven project from scratch, using an Extreme Programming model, and show:

- How to do it
- The benefits it can provide
- Pitfalls to avoid

Example Project: Thumbnailer

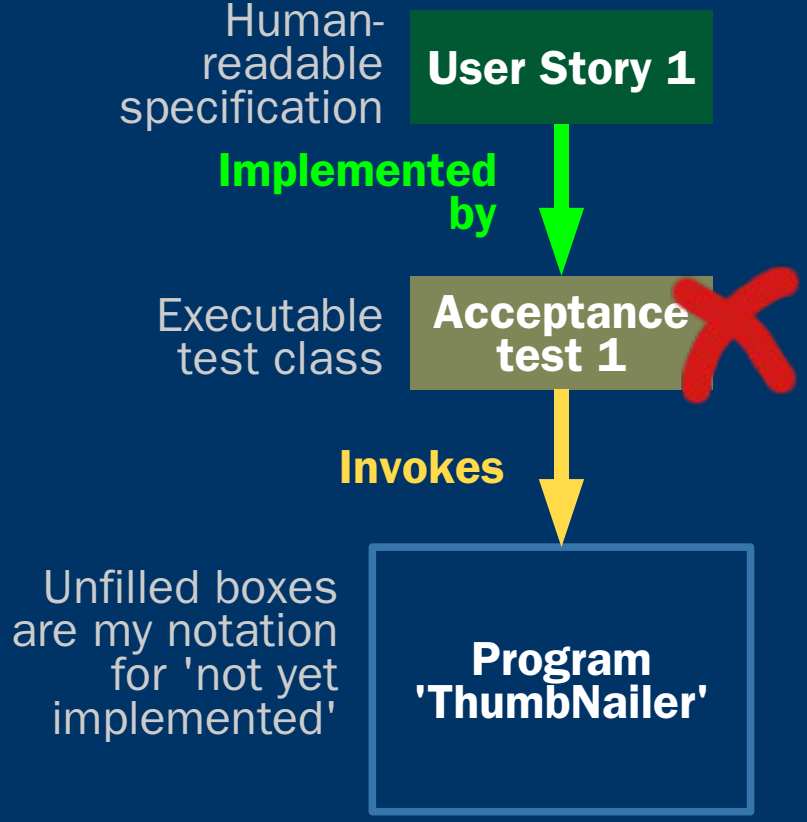
To scan a directory of images and use the Python Image Library (PIL) to generate thumbnails in a new subdirectory.

I want to develop the project in a test-driven way.

What do I do?

Step 1: Write an Acceptance Test

Also known as a functional, integration, system-level, scenario or end-to-end test.



Is an executable version of your human-readable specification documents.

A program which made this test pass can be considered to perfectly fulfil the spec.

But we haven't yet written any such program...

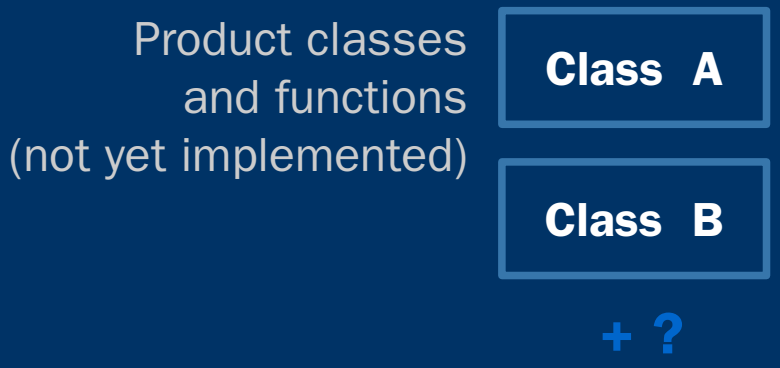
See source code s00 to s04

Test-driven means tests first

Write acceptance tests *before* writing or designing the program under test.

- Writing tests afterwards leads to poor tests (miss your own blind spots, and psychological disincentive to find problems)
- Writing tests first allows you to focus on how you want the product to behave, thus refining and clarifying the specification, without being prematurely influenced by product design ideas.

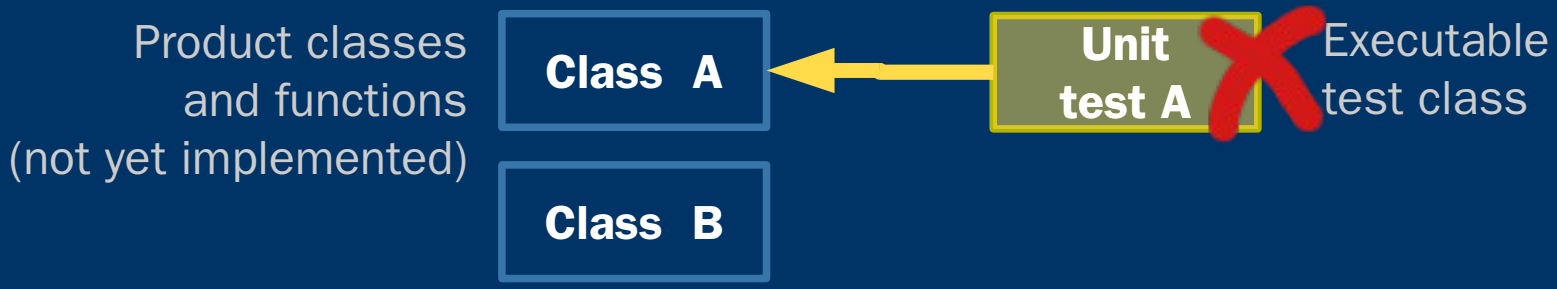
Step 2: Start your product design



Plan out just enough functionality to make your first acceptance test pass.

Use your failing acceptance test to drive the area of the code which you work on.

Step 3: Write one unit test



A unit test is an executable test class, just like an acceptance test.

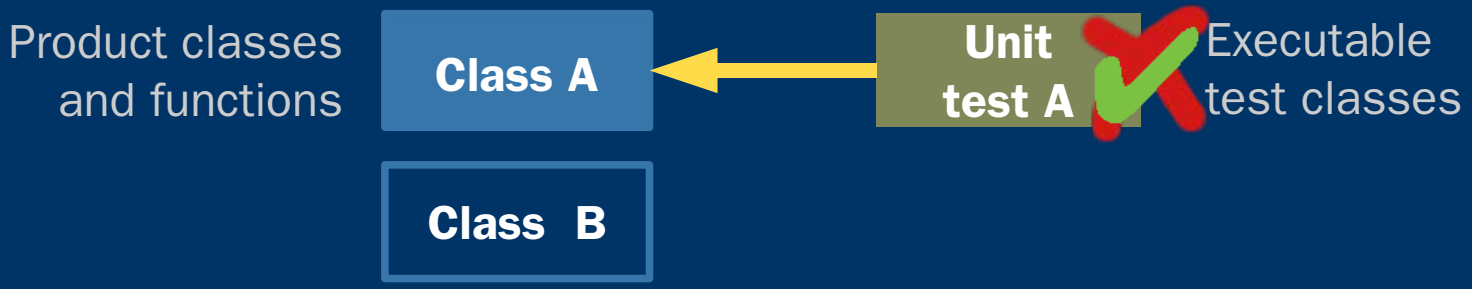
But unit tests do not invoke the whole program - they test individual functions and classes at the lowest level.

Again, test first is important.

Run your unit test to watch it fail

See source code s05

Step 4: Start coding (Finally!)

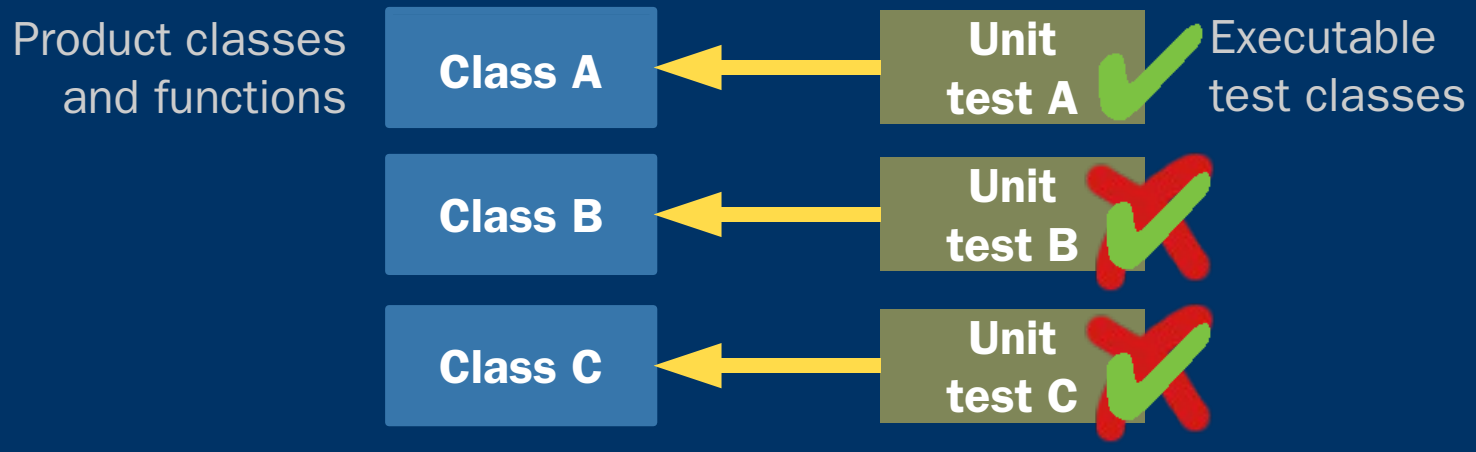


Use your unit test failures to improve your product classes, until all unit tests pass.

Note here the most prominent disadvantage of TDD – we have done a lot of work creating tests, and only now get to start work on our product.

See source code s06 & s07

Step 5: Add more Unit Tests

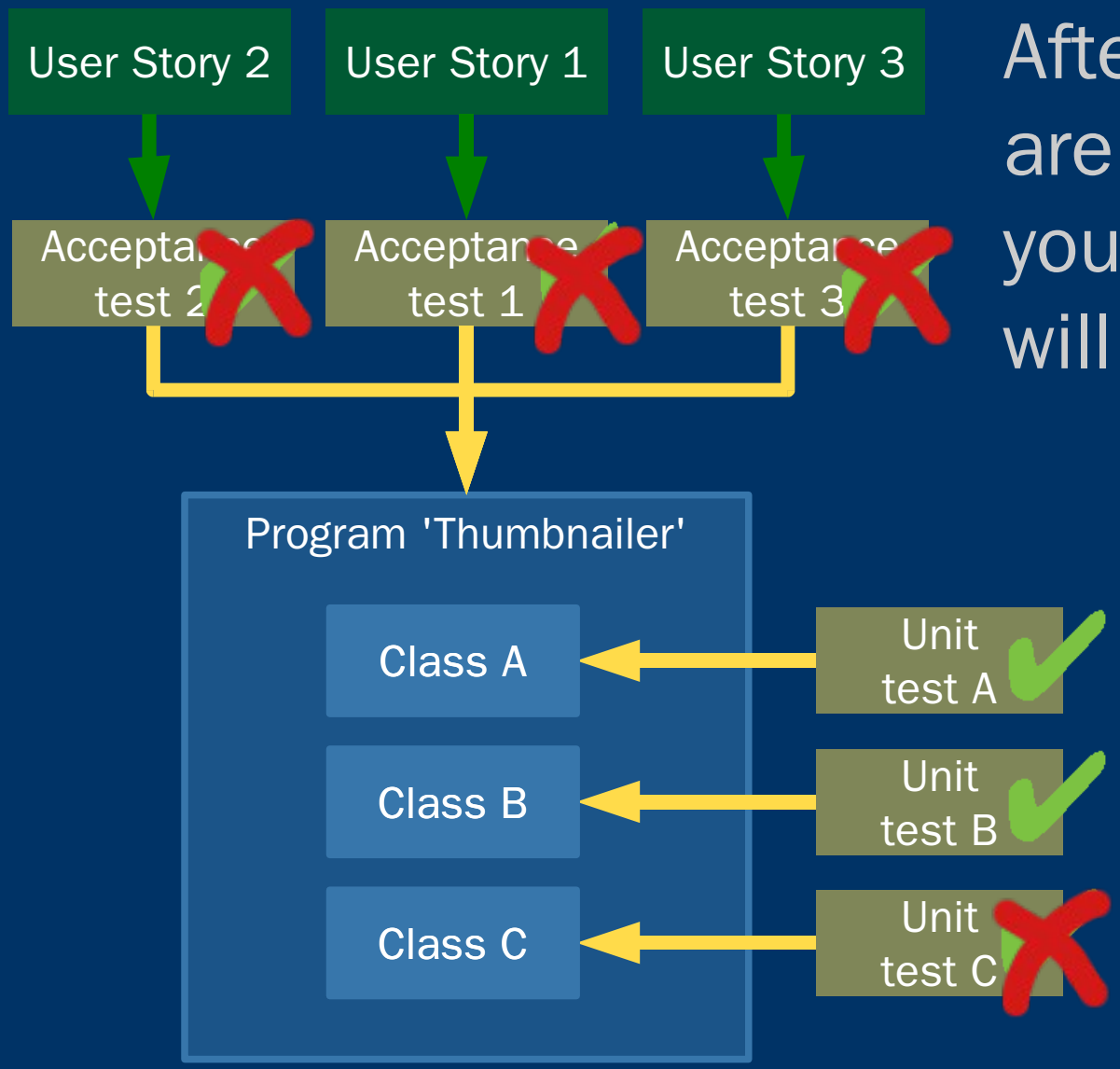


Always use the current acceptance test failure to drive which classes you work on next.

Improve your unit tests, and make new ones, to cover the classes you are about to implement.

Implement and improve the product code again, to make the unit tests pass.

Step 6: Add More Acceptance Tests

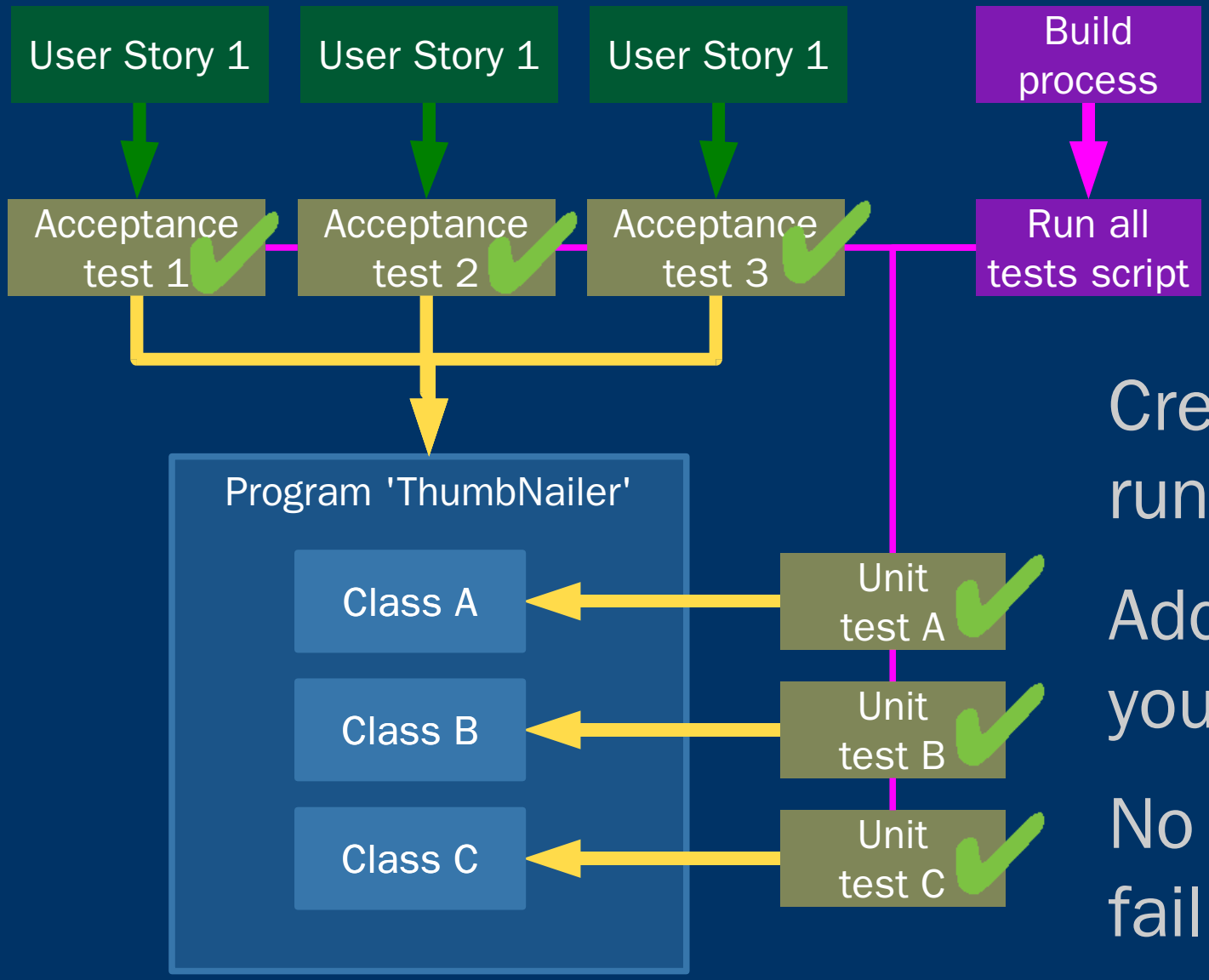


After several unit tests are passing, eventually your acceptance test will also start to pass.

Then you should add a new acceptance test, and iterate the process.

See source code s08

Step 7: Integrate into build process



Create a script to run all tests

Add this script to your build process

No check-in with failing tests

More advanced scenarios

- Run a continuous build server, to flush out race conditions or other occasionally failing tests.
- Integration builds can verify that changes work on several platforms.
- Tests can measure performance, giving instant feedback if behaviour-preserving changes introduce unexpected time or memory costs.
- Distributed builds for speed

Brooks' Silver Bullet?

- Brooks stated there would be 'no silver bullet' to the problems of complexity inherent to software and the development process.
- Tests allow objective measurement of whether code changes are an improvement or not.
- Without such measurement, in a system of sufficient complexity, we are just guessing that each change is an overall improvement.
- With objective measurement, we can eliminate bad changes, creating a ratchet to ensure that changes always act to improve our product.

Disadvantages and pitfalls

- Writing and maintaining test code takes time. (eg. 4:1 ratio of test code to product)
- Sporadically failing tests, caused by race conditions, non-deterministic behaviour or external influences, can be troublesome. (try continuous builds)
- Some things are hard to write tests for, especially acceptance tests of GUI. (use a framework such as Selenium for web apps)
- Tests for large projects can take a long time to run, reducing agility. (try distributed builds)

Advantages

- Assurance that new code really works.
- Instant feedback if changes break old code.
- Eliminates the need for project integration test phases – development is sustainable.
- Enables massive, painless refactoring.
- Product code has better external interfaces, and better internal design.
- Tests form up-to-date and unambiguous docs of product code's abilities and intended usage.
- Passing acceptance tests means provably conforming to an unambiguous, predefined spec.
- Breaks down large, daunting tasks into controlled, well-understood steps.
- Reduces developer gold-plating

PyCon 2008,
Chicago, USA.

Jonathan Hartley

Thanks for listening.

EOF

tartley@tartley.com
http://tartley.com