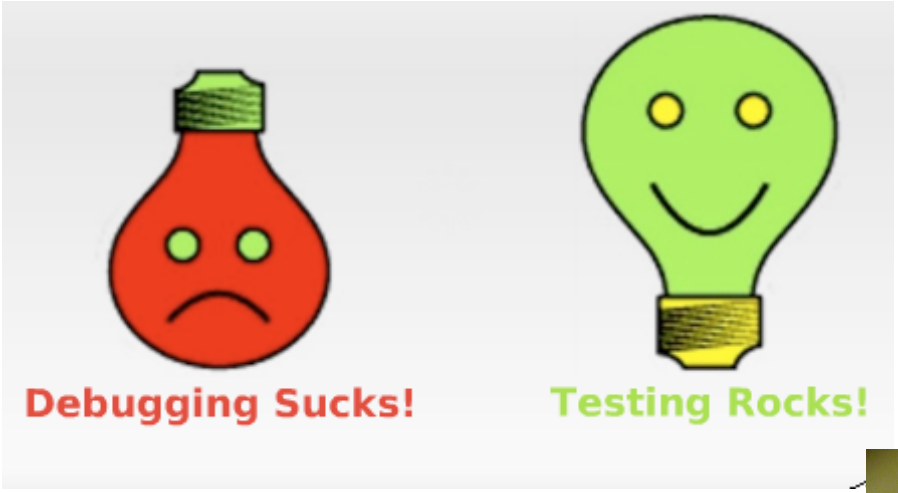
Test-Driven Development

CS 169 Spring 2012

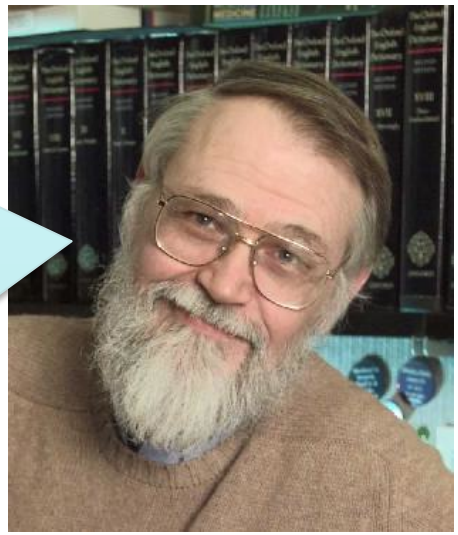
Armando Fox, David Patterson,
Koushik Sen

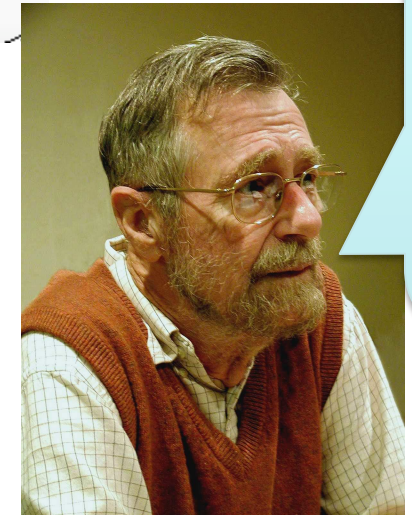
Testing Overview
(ELLS §5.1)

Armando Fox



Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.





Testing can never demonstrate the _____ of errors in software, only their _____

Survey Finds 58% of Software Bugs Result from Test Infrastructure and Process, Not Design Defects

Developers Prefer Taxes to Dealing with Software Testing

Sunnyvale, Calif. — June 2, 2010 Electric Cloud®, the leading provider of software production management (SPM) solutions, today released the results of a survey conducted in partnership with Osterman Research showing that the majority of software bugs are attributed to poor testing procedures or infrastructure limitations rather than design problems. Additionally, the software test process is generally considered an unpleasant process, with software development professionals rating the use of their companies' test systems more painful than preparing taxes.

Fifty-eight percent of respondents pointed to problems in the testing process or infrastructure as the cause of their last major bug found in delivered or deployed software, not design defects.

Specifically, the survey found:

- Completely automated software testing environments are still rare, with just 12 percent of software development organizations using fully automated test systems. Almost 10 percent reported that all testing was done manually.

Testing Today

- Before/Waterfall
  - developers finish code, some ad-hoc testing
  - "toss over the wall to Quality Assurance [QA]"
  - QA people manually poke at software
- Today/Agile
  - testing is part of *every* Agile iteration
  - developers responsible for testing own code
  - testing tools & processes highly automated;
  - QA/testing group improves *testability* & *tools*

Testing Today

- Before/Waterfall
  - developers finish code, some ad-hoc testing
- [illegible]
  - developers responsible for testing own code
  - testing tools & processes highly automated;
  - QA/testing group improves *testability* & *tools*

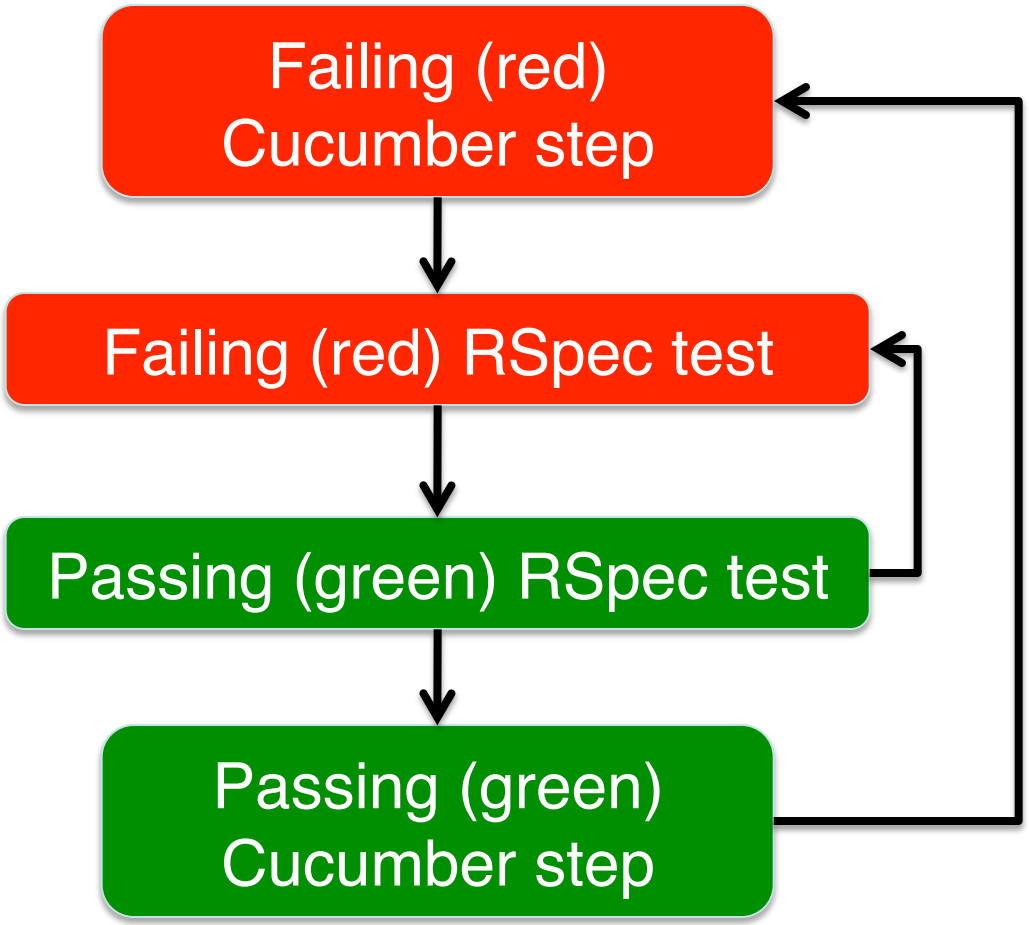
*Software Quality is the result of a good **process**, rather than the responsibility of one specific group*

BDD+TDD: The Big Picture

- Behavior-driven design (BDD)
  - develop user stories to describe features
  - via Cucumber, user stories become *acceptance tests* and *integration tests*
- Test-driven development (TDD)
  - *step definitions* for new story,may require new code to be written
  - TDD says: write unit & functional tests for that code *first,* ***before*** the code itself
  - that is: write tests for *the code you wish you had*

Cucumber & RSpec

- Cucumber describes *behavior* via features & scenarios (*behavior driven* design)
- RSpec tests individual modules that contribute to those behaviors (*test driven* development)



Which are true about BDD & TDD:
a) requirements drive implementation
b) they're used only within Agile development
c) they embrace & deal with change

- ☐ Only (a)
- ☐ Only (a) & (b)
- ☐ Only (a) & (c)
- ☐ (a), (b) and (c)

FIRST, TDD, and Getting Started With RSpec

(ELLS §5.2)

Armando Fox

Unit tests should be FIRST

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-checking
- **T**imely

Unit tests should be FIRST

- **F**ast: run (subset of) tests quickly (since you'll be running them *all the time*)
- **I**ndependent: no tests depend on others, so can run *any subset* in *any order*
- **R**epeatable: run N times, get same result (to help isolate bugs and enable automation)
- **S**elf-checking: test can *automatically* detect if passed (*no human checking* of output)
- **T**imely: written about the same time as code under test (with TDD, written *first!*)

RSpec, a Domain-Specific Language for testing

- DSL: small programming language that simpifies one task at expense of generality
  - examples so far: migrations, regexes, SQL
- RSpec tests are called *specs*, and inhabit `spec` directory

  `rails generate rspec:install` creates structure

`app/models/*.rb`	`spec/models/*_spec.rb`
`app/controllers/*_controller.rb`	`spec/controllers/*_controller_spec.rb`
`app/views/*/*.html.haml`	(use Cucumber!)

Example: calling TMDb

- New RottenPotatoes feature: add movie using info from TMDb (vs. typing in)
- How should user story steps behave?

```
When I fill in "Search Terms" with "Inception"
And I press "Search TMDb"
Then I should be on the RottenPotatoes homepage
...
```

Recall Rails Cookery #2:

adding new feature ==

new route+new controller method+new view

The Code You Wish You Had

What should the *controller method* do that receives the search form?

1. it should call a method that will search TMDb for specified movie
2. if match found: it should select (new) "Search Results" view to display match
3. If no match found: it should redirect to RP home page with message

http://pastebin.com/kJxjwSF6

The method that contacts TMDb to search for a movie should be:

- ☐ A class method of the Movie model
- ☐ An instance method of the Movie model
- ☐ A controller method
- ☐ A helper method

The TDD Cycle: Red–Green–Refactor

(ELLS §5.3)

Armando Fox

Test-First development

- Think about one thing the code *should* do
- Capture that thought in a test, which fails
- Write the simplest possible code that lets the test pass
- Refactor: DRY out commonality w/other tests
- Continue with next thing code should do

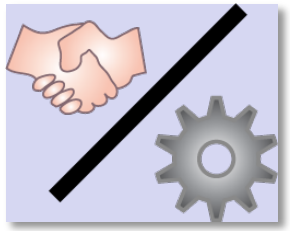
Red – Green – Refactor

Aim for "always have working code"

TDD for the Controller action: Setup

- Add a route to **config/routes.rb**

```
# Route that posts 'Search TMDb' form
post '/movies/search_tmdb'
```



  - Convention over configuration will map this to `MoviesController#search_tmdb`

- Create an empty view:

```
touch app/views/movies/search_tmdb.html.haml
```

- Replace fake "hardwired" method in **movies_controller.rb** with empty method:

```
def search_tmdb
end
```

What model method?

- Calling TMDb is responsibility of the model... but no model method exists to do this yet!
- No problem...we'll use a seam to test the *code we wish we had ("CWWWH")*, `Movie.find_in_tmdb`
- Game plan:
  - Simulate POSTing search form to controller action.
  - Check that controller action *tries to call* `Movie.find_in_tmdb` with data from submitted form.
  - The test will fail (red), because the (empty) controller method *doesn't* call find_in_tmdb.
  - Fix controller action to make green.

http://pastebin.com/zKnwphQZ

Seams

- A place where you can change your app's *behavior* without editing the *code*.
  (Michael Feathers, *Working Effectively With Legacy Code*)
- Useful for testing: *isolate* behavior of some code from that of other code it depends on.
- `should_receive` uses Ruby's open classes to create a seam for isolating controller action from behavior of (possibly buggy or missing) `Movie.find_in_tmdb`
- Rspec *resets* all mocks & stubs after *each example* (keep tests Independent)

How to make this spec green?

- Expectation says controller action should call `Movie.find_in_tmdb`
- So, let's call it!

http://pastebin.com/DxzFURiu

The spec has *driven* the creation of the controller method to pass the test.

- But shouldn't `find_in_tmdb` *return* something?

Test techniques we know

```
obj.should_receive(a).with(b)
```

Optional!

Which is FALSE about `should_receive`?

- ☐ It provides a stand-in for a real method that doesn't exist yet
- ☐ It would override the real method, even if it did exist
- ☐ It can be issued either before or after the code that should make the call
- ☐ It exploits Ruby's open classes and metaprogramming to create a seam

More Controller Specs and Refactoring

(ELLS §5.4)

Armando Fox

Where we are & where we're going: "outside in" development

- Focus: write *expectations* that drive development of controller method
  - Discovered: must *collaborate* w/model method
  - Use outside-in recursively: *stub* model method in this test, write it later
- Key idea: *break dependency* between method under test & its collaborators
- Key concept: *seam*—where you can affect app behavior without editing code

The Code You Wish You Had

What should the *controller method* do that receives the search form?

1. it should call a method that will search TMDb for specified movie
2. if match found: it should select (new) "Search Results" view to display match
3. If no match found: it should redirect to RP home page with message

"it should select Search Results view to display match"

- Really 2 specs:

1. It should decide to render Search Results
   - more important when different views could be rendered depending on outcome
2. It should make list of matches available to that view

- New *expectation* construct:

  `obj.should` *match-condition*

  - Many built-in matchers, or define your own

Should & Should-not

- Matcher applies test to receiver of *should*

`count.should == 5`	Syntactic sugar for `count.should.==(5)`
`5.should(be.<(7))`	`be` creates a lambda that tests the predicate expression
`5.should be < 7`	Syntactic sugar allowed
`5.should be_odd`	Use `method_missing` to call `odd?` on 5
`result.should include(elt)`	calls Enumerable#include?
`result.should match(/regex/)`	
`should_not` also available	

```
result.should render_template('search_tmdb')
```

Checking for rendering

- After `post :search_tmdb`, `response()` method returns controller's *response object*
- `render_template` matcher can check what view the controller tried to render

http://pastebin.com/C2x13z8M

- Note that this view has to exist!
  - `post :search_tmdb` will try to do the whole MVC flow, including rendering the view
  - hence, controller specs can be viewed as *functional testing*

Test techniques we know

```
obj.should_receive(a).with(b)
```

```
obj.should match-condition
```

Rails-specific extensions to RSpec:

```
response()
render_template()
```

Which of these, if any, is *not* a valid use of `should` or `should_not`?

- ☐ `result.should_not be_empty`
- ☐ `5.should be <=> result`
- ☐ `result.should_not match /^D'oh!$/`
- ☐ All of the above are valid uses

More Controller Specs and Refactoring, continued

(ELLS §5.4)

Armando Fox

It should make search results available to template

- Another rspec-rails addition: `assigns()`
  - pass symbol that names controller instance variable
  - returns value that controller assigned to variable

- D'oh! our current code *doesn't set any instance variables:*
- TCWWWH: list of matches in `@movies`

http://pastebin.com/DxzFURiu

http://pastebin.com/4W08wL0X

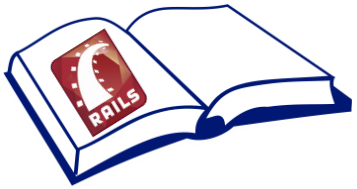
Two new seam concepts

- `stub`
  - similar to `should_receive`, but not expectation
  - `and_return` optionally controls return value
- `mock`: create dumb "stunt double" object
  - stub individual methods on it:

```
m = mock('movie1')  m.stub
  (:title).and_return('Rambo')
```

  - shortcut: `m=mock('movie1',:title=>'Rambo')`

each seam enables just enough functionality for some *specific* behavior under test



Test Cookery #1

- Each spec should test *just one behavior*
- Use seams as needed to isolate that behavior
- Determine which expectation you'll use to check the behavior
- Write the test and make sure it fails for the right reason
- Add code until test is green
- Look for opportunities to refactor/beautify

Test techniques we know

```
obj.should_receive(a).with(b).and_return(c)
obj.stub(a).and_return(b)

d = mock('impostor')
```

Optional!

```
obj.should match-condition
```

Rails-specific extensions to RSpec:

```
assigns(:instance_var)
response()
render_template()
```

`should_receive` combines _____
and _____,
whereas `stub` is only _____.

- ☐ A mock and an expectation; a mock
- ☐ A mock and an expectation; an expectation
- ☐ A seam and an expectation; an expectation
- ☐ A seam and an expectation; a seam

Fixtures and Factories

(ELLS §5.5)

Armando Fox

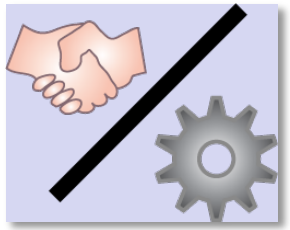
When you need the real thing

Where to get a real object: http://pastebin.com/N3s1A193

- Fixture: statically preload some known data into database tables
- Factory: create only what you need per-test

Fixtures

- database wiped & reloaded before *each spec*
  - add `fixtures :movies` at beginning of `describe`
  - **`spec/fixtures/movies.yml`** are `Movies` and will be added to **`movies`** table



- Pros/uses
  - truly static data, e.g. configuration info that never changes
  - easy to see all test data in one place
- Cons/reasons not to use
  - Introduces dependency on fixture data

Factories

- Set up “helpers” to quickly create objects with default attributes, as needed per-test
- Example: FactoryGirl gem http://pastebin.com/bzvKG0VB
  - or just add your own code in **`spec/support/`**
- Pros/uses:
  - Keep tests Independent: unaffected by presence of objects they don’t care about
- Cons/reasons not to use:
  - Complex relationships may be hard to set up (but may indicate too-tight coupling in code!)

Pitfall: *mock trainwreck*

- Goal: test searching for movie by its director or by awards it received

```
a = mock('Award', :type => 'Oscar')
d = mock('Director',
    :name => 'Darren Aronovsky'
m = mock('Movie', :award => a,
    :director => d)
```

...etc...

```
m.award.type.should == 'Oscar'
m.director.name.split(/ +/).last.should
  == 'Aronovsky'
```

Which of the following kinds of data, if any, should *not* be set up as fixtures?

- ☐ Movies and their ratings
- ☐ The TMDb API key
- ☐ The application's time zone settings
- ☐ Fixtures would be fine for all of these

TDD for the Model & Stubbing the Internet

(ELLS §5.6–5.7)

Armando Fox

What should model method find_in_tmdb do?

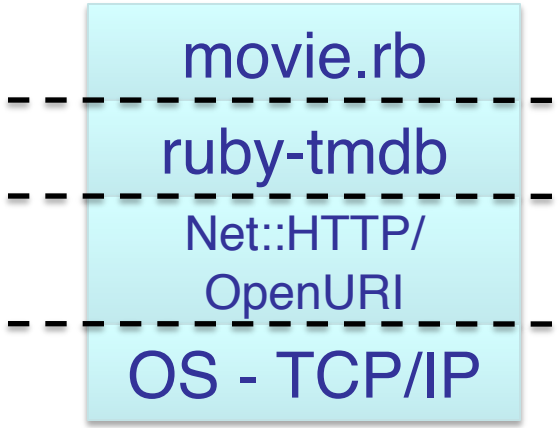
- It should call TmdbRuby gem with title keywords
  - If we had no gem: It should directly submit a RESTful URI to remote TMDb site
- What if TmdbRuby gem signals error?
  - API key is invalid
  - API key is not provided
- *Explicit vs. implicit* requirements
- Use *context & describe* to divide up tests

http://pastebin.com/ELQfC8Je

Review

- Implicit requirements derived from explicit
  - by reading docs/specs
  - as byproduct of designing classes
- We used 2 different stubbing approaches
  - case 1: we *know* TMDb will *immediately* throw error; want to test that we catch & convert it
  - case 2: need to *prevent* underlying library from contacting TMDb at all
- `context` & `describe` group similar tests
  - in book: using `before(:each)` to setup common preconditions that apply to whole group of tests

Where to stub in Service Oriented Architecture?




```
Movie.stub(:find_in_tmdb).
  and_return(...)
TmdbMovie.stub(:find).with(...).
  and_return(...)
Net::HTTP.stub(:get).
  with(...).
  and_return(...)
```

Rule of thumb:

- for unit testing, stub nearby, for maximum *isolation* of class under test (Fast, Independent)
- for integration testing, stub *far away*, to test as many interfaces as possible

Parse contrived request
Return canned value(s)

Stubbing the Internet

- Almost always ≥1 way to test it.
- Correct seam depends on *focus of test*
  - …ensure model working right?
  - …ensure model can deal with external error?
  - …ensure model's collaborators working right?
- You can create your own test stubs
  - *observe* service's behavior (read API docs, try it yourself)
  - *mimic* behavior in your stubs
  - (aside…this is how the autograder is tested!)

Test techniques we know

```
obj.should_receive(a).with(b).and_return(c)
                .with(hash_including 'k'=>'v')
obj.stub(a).and_raise(SomeClass::SomeError)

d = mock('impostor')

obj.should raise_error(SomeClass::SomeError)
describe, context
```

Rails-specific extensions to RSpec:

```
assigns(:instance_var)
response()
render_template()
```

Which statement(s) are TRUE about Implicit requirements?

- ☐ They are often, but not always, derived from explicit requirements
- ☐ They apply only to unit & functional tests, not integration tests
- ☐ Testing them is lower priority than testing explicit requirements, since they don't come from the customer
- ☐ All of the above are true

Coverage, Unit vs. Integration Tests, Other Testing Concepts, and Perspectives

(ELLS §5.8–5.11)

Armando Fox

How much testing is enough?

- Bad: “Until time to ship”
- A bit better: (Lines of code) / (Lines of tests)
  - *code-to-test* ratio 1.2–1.5 not unreasonable
  - often *much higher* for production systems
- Better question: “How thorough is my testing?”
  - Formal methods (later in semester)
  - Coverage measurement
  - We focus on the latter, though the former is gaining steady traction

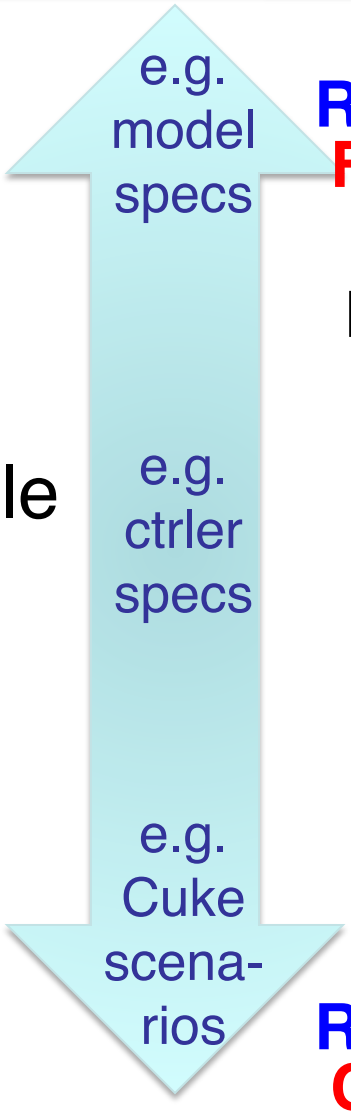
Measuring Coverage—Basics

```
class MyClass
 def foo(x,y,z)
  if x
    if (y && z) then bar(0) end
   else
    bar(1)
   end
 end
 def bar(x) ; @w = x ; end
end
```

- S0: every method called
- S1: every method *from every call site*
- C0: every statement
  - Ruby SimpleCov gem
- C1: every branch in both directions
- C1+decision coverage: every *subexpression* in conditional
- C2: every path (difficult, and disagreement on how valuable)

What kinds of tests?

- Unit (one method/ class)
- Functional or module (a few methods/ classes)
- Integration/system



Runs fast **High coverage**
Fine resolution
Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces
Runs slow **Low coverage**
Coarse resolution

Other testing terms you may hear

- Mutation testing: if introduce deliberate error in code, does some test break?
- Fuzz testing: 10,000 monkeys throw random input at your code
  - Find ~20% MS bugs, crash ~25% Unix utilities
  - *Tests app the way it wasn't meant to be used*
- DU-coverage: is every pair <define x/use x> executed?
- Black-box vs. white-box/glass-box

Going to extremes

- ✗ "I kicked the tires, it works"
- ✗ "Don't ship until 100% covered & green"
- ☑ TRUTH: use coverage to identify untested or undertested parts of code
- ✗ "Focus on unit tests, they're more thorough"
- ✗ "Focus on integration tests, they're more realistic"
- ☑ TRUTH: each finds bugs the other misses

TDD vs. Conventional debugging

Conventional	TDD
Write 10s of lines, run, hit bug: break out debugger	Write a few lines, with test first; know immediately if broken
Insert printf's to print variables while running repeatedly	Test short pieces of code using expectations
Stop in debugger, tweak/set variables to control code path	Use mocks and stubs to control code path
Dammit, I thought for sure I fixed it, now have to do this all again	Re-run test automatically

- Lesson 1: TDD uses same skills & techniques as conventional debugging—but more productive (FIRST)

- Lesson 2: writing tests *before* code takes more time up-front, but often less time overall

Which of these is POOR advice for TDD?

- ☐ Mock & stub early & often in unit tests
- ☐ Aim for high unit test coverage
- ☐ Sometimes it's OK to use stubs & mocks in integration tests
- ☐ Write code first, then tests, as long as you don't wait too long after code written

TDD Summary

- Red – Green – Refactor, and always have working code
- Test *one* behavior at a time, using seams
- Use `it` "placeholders" or `pending` to note tests you know you'll need
- Read & understand coverage reports
- "Defense in depth": don't rely too heavily on any *one* kind of test