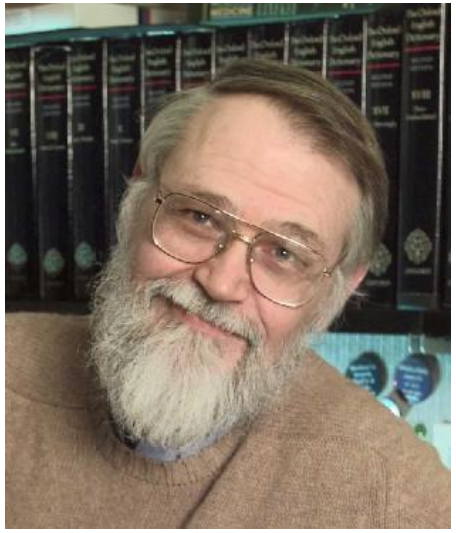
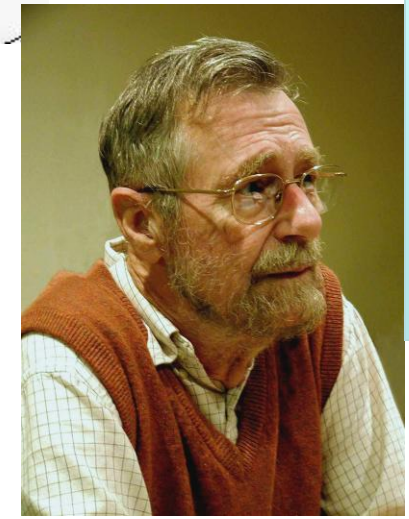
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.





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

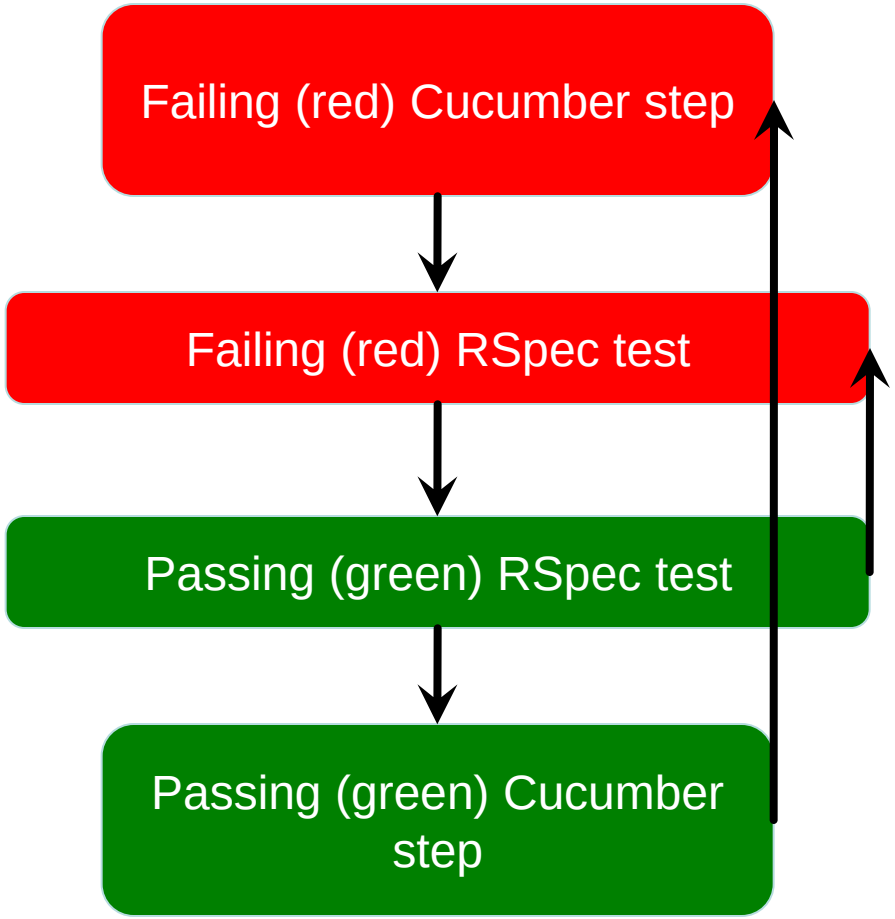
*Software Quality is the result of a good **process**, rather than the responsibility of one specific group*

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 §6.2)*

Armando Fox

© 2012 Armando Fox & David Patterson
Licensed under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

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

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

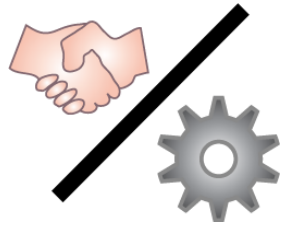
Red – Green – Refactor

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

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

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 @

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