Fixtures and Factories *(ELLS §6.5)*

Armando Fox

When you need the real thing

Where to get a real object:

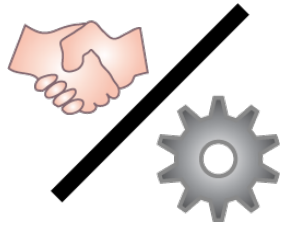
http://pastebin.com/N3s1A193

•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 Moviesand 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

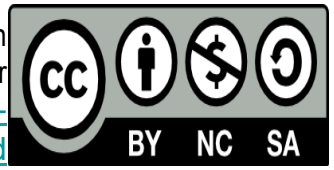
```
a = mock('Award', :type => 'Oscar')
d = mock('Director',
   :name => 'Darren Aronovsky'
m = mock('Movie', :award => a,
   :director => d)
...etc...
m.award.type.should == 'Oscar'
m.director.name.split(/ +/).last.should
  == 'Aronovsky'
```

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 §6.6–6.7)*

Armando Fox



What should model method find_in_tmdb do?

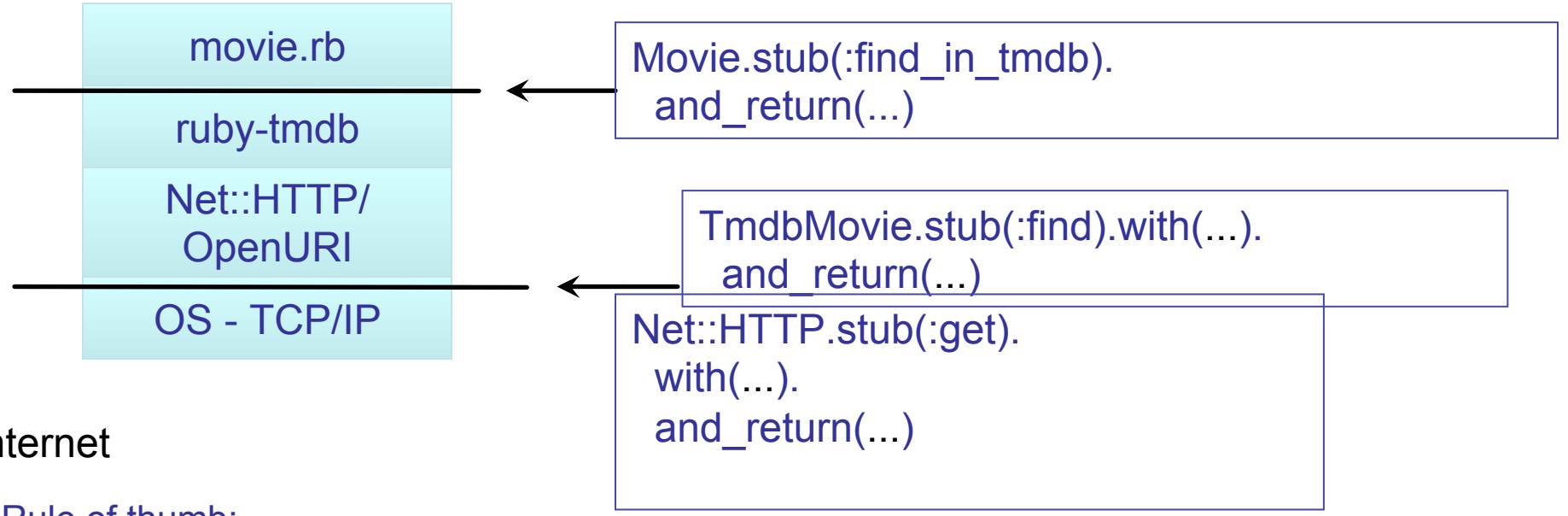
- It should call TmdbRuby gem with title keywords
- If we had no gem: It should directly submit a RESTful URI to remote TMDb site
- What if TmdbRuby gem signals error?
- API key is invalid
- API key is not provided
- *Explicit vs. implicit* requirements
- Use *context & describe* to divide up tests

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

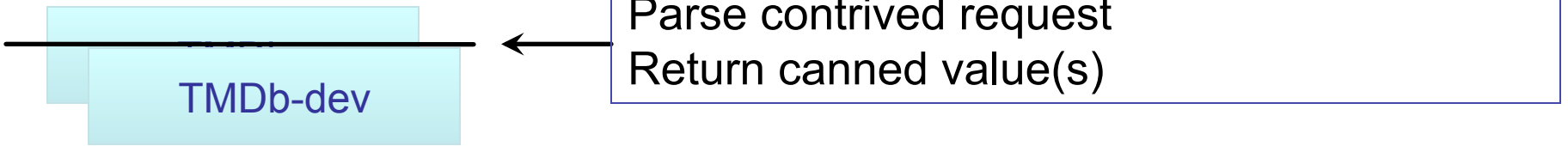
Where to stub in Service Oriented Architecture?



Internet

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



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 §6.8–6.11)*

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

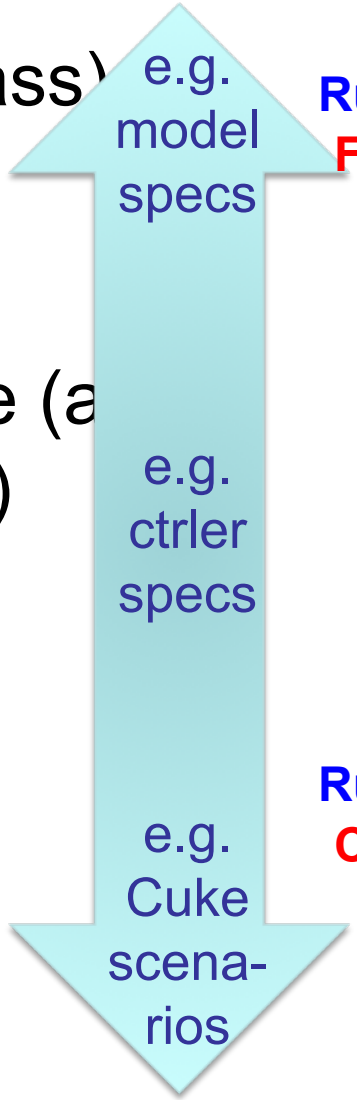
Measuring Coverage—Basics

```
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
- ☐ Unit tests give you higher confidence of system correctness than integration tests

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

NOTE

Students will not be tested on material in this section past “TDD Summary”.