Advanced Testing Concepts (in Ruby on Rails)

Software Engineering II
WS 2020/21

Enterprise Platform and Integration Concepts

Agenda

Advanced Concepts & Testing Tests

- **Setup and Teardown**
- Test Data
- Test Doubles

Setup and Teardown: RSpec

As a developer using RSpec
I want to execute code before and after test blocks
So that I can control the environment in which tests are run

```
before(:example) # run before each test block
before(:context) # run one time only, before all of the examples in a group

after(:example) # run after each test block
after(:context) # run one time only, after all of the examples in a group
```

Setup RSpec – before(:example)


```ruby
class Thing
  def widgets
    @widgets = []
  end
end

describe Thing do
  before(:example) do
    @thing = Thing.new
  end

  describe "initialized in before(:example)" do
    it "has 0 widgets" do
      expect(@thing.widgets.count).to eq(0)
    end
  end
end
```

- `before(:example)` blocks are run before each example
- `:example` scope is also available as `:each`

- https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks

Setup RSpec – before(:context)


```ruby
class Thing
  ... #as before

describe Thing do
  before(:context) do
    @thing = Thing.new
  end

  context "initialized in before(:context)" do
    it "can accept new widgets" do
      @thing.widgets << Object.new
    end

    it "shares state across examples" do
      expect(@thing.widgets.count).to eq(1)
    end
  end
end
```

- `before(:context)` blocks are run before all examples in a group
- `:context` scope is also available as `:all`
- **Warning:** Mocks are only supported in `before(:example)`

- https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks

Teardown RSpec

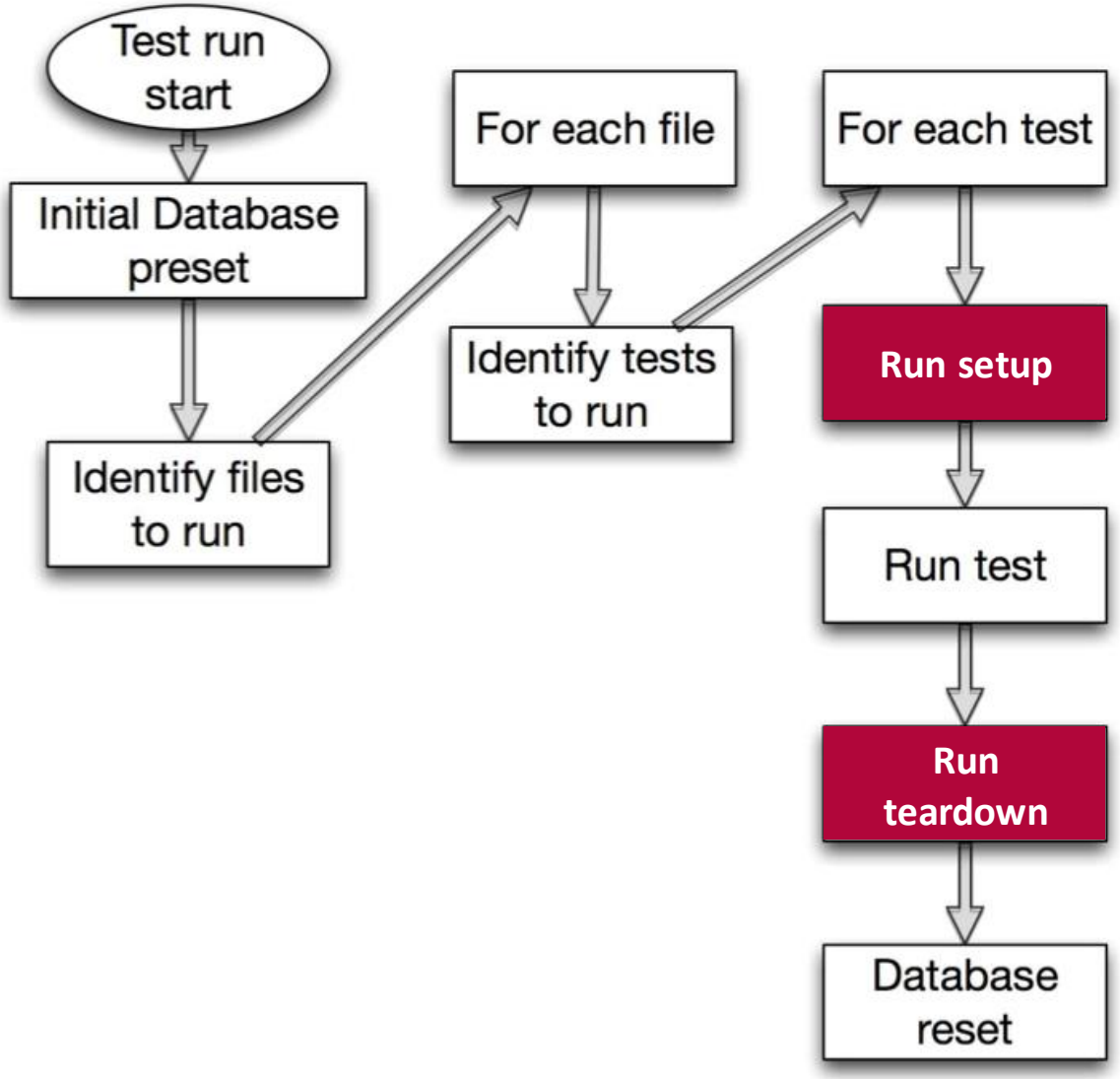
```ruby
describe "Test the website with a browser" do
  before(:context) do
    @browser = Watir::Browser.new
  end

  it "should visit a page" do
    ...
  end

  after(:context) do
    @browser.close
  end
end
```

- `after(:context)` blocks are run after all examples in a group
- For example to clean up

Test Run



- Rails Test Prescriptions. Noel Rappin. 2010. p. 37. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Agenda

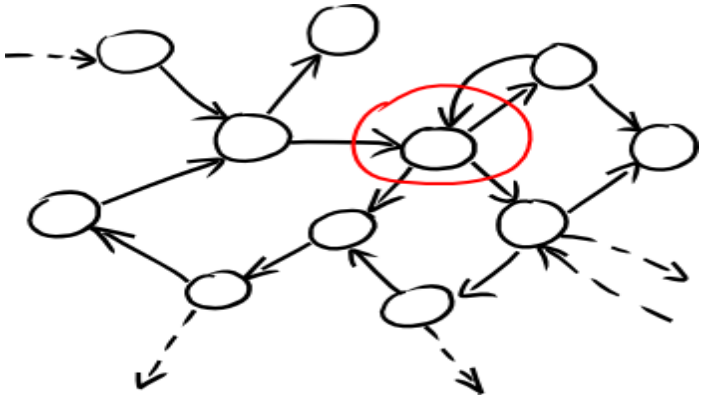
Advanced Concepts & Testing Tests

- Setup and Teardown
- **Test Data**
- Test Doubles

Isolation of Test Cases

Tests should be independent

- If a bug in a model is introduced:
  Only tests related to this model should fail
- Allow localization of bug

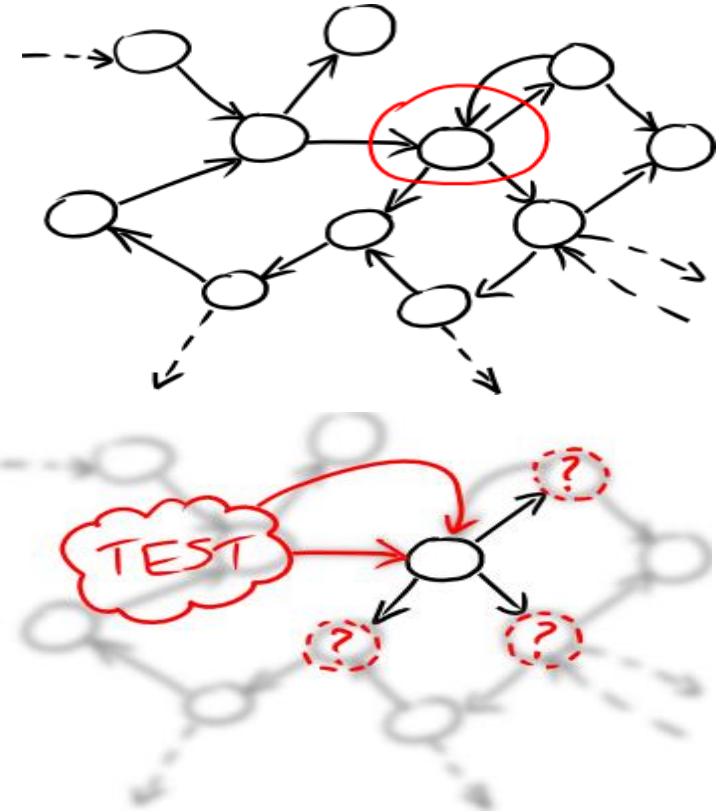


Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Isolation of Test Cases

Achieving Test Case Isolation

- Don't write complex tests
- **Don’t share complex test data**
- Don’t use complex objects



Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Test Data Overview

Two main ways to provide data to test cases:

Fixtures

- Fixed state at the beginning of a test
- Assertions can be made against this state

Factories

- Blueprints for models
- Used to generate test data locally in the test

Fixture Overview

Fixtures for testing

- Fixed Sample data/objects
- Populate testing database with **predefined data** before test run
- Stored in database independent files (e.g. `test/fixtures/<name>.yml`)

```
# test/fixtures/users.yml
david: # Each fixture has a name
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: Front-end engineer
```

- http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html
- http://guides.rubyonrails.org/testing.html

Drawbacks of Fixtures

Fixtures are global

- Only one set of data, every test has to deal with all test data

Fixtures are spread out

- Own directory
- One file per model -> data for one test is spread out over many files
- Tracing relationships is challenging

Fixtures are distant

- Fixture data is not immediately available in the test
- `expect(users(:ernie).age + users(:bert).age).to eq(20) #why 20?`

Fixtures are brittle

- Tests rely on fixture data, they break when data is changed
- Data requirements of tests may be incompatible

Test Data Factories

Test data should be

- **Local**: Defined as closely as possible to the test
- **Compact**: Easy and quick to specify; even for complex data sets
- **Robust**: Independent from other tests

One way to achieve these goals: **Data factories**

Defining Factories

```ruby
# This will guess the User class
FactoryBot.define do
  factory :user do
    first_name { "John" }
    last_name { "Doe" }
    admin false
  end

  # This will use the User class
  # (Admin would have been guessed)
  factory :admin, class: User do
    first_name { "Admin" }
    last_name { "User" }
    admin true
  end
end
```

We use FactoryBot

- Rich set of features around
  - Creating objects
  - Connecting objects
- Rails automatically loads `spec/factories.rb` and `spec/factories/*.rb`

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Using Factories

- Different strategies: *build, create* (standard), *attributes_for*

```ruby
# Returns a User instance that's _not_ saved
user = build(:user)

# Returns a _saved_ User instance
user = create(:user)

# Returns a hash of attributes that can be used to build a User instance
attrs = attributes_for(:user)

# Passing a block will yield the return object
create(:user) do |user|
  user.posts.create(attributes_for(:post))
end
```

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Attributes


```ruby
# Lazy attributes
factory :user do
  activation_code { User.generate_activation_code }
  date_of_birth { 21.years.ago }
end

# Dependent attributes
factory :user do
  first_name { "Joe" }
  email { "#{first_name}.#{last_name}@example.com".downcase }
end

# override the defined attributes by passing a hash/dict
create(:user, last_name: "Doe").email
# => "joe.doe@example.com"
```

The opposite of lazy is eager evaluation

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Associations


```
factory :post do
  # If factory name == association name, the factory name can be left out.
  author
End

factory :post do
  # specify a different factory or override attributes
  association :author, factory: :user, last_name: "Writely“
End

# Builds and saves a User and a Post
post = create(:post)
post.new_record?              # => false
post.author.new_record?       # => false

# Builds and saves a User, and then builds but does not save a Post
post = build(:post)
post.new_record?              # => true
post.author.new_record?       # => false
```

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Agenda

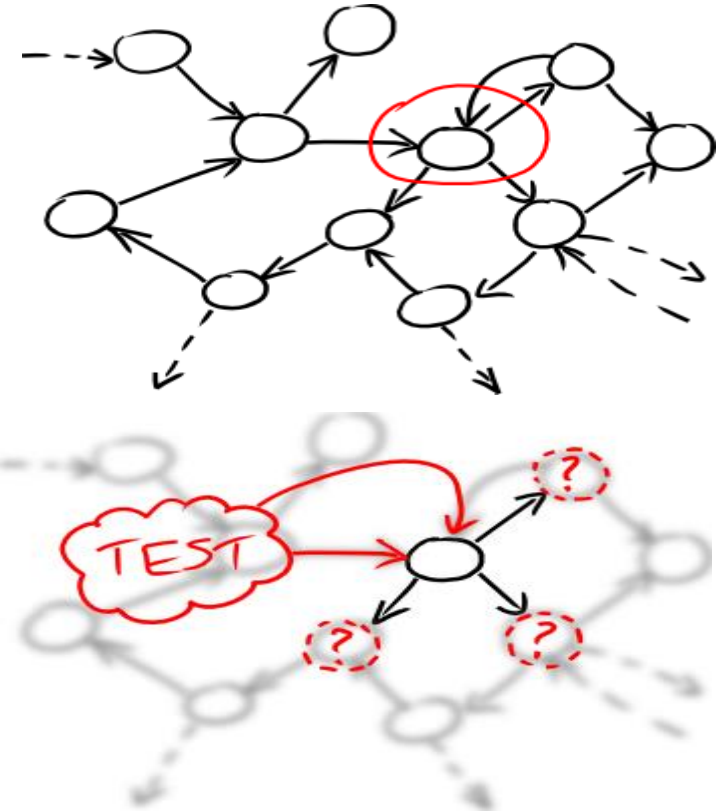
Advanced Concepts & Testing Tests

- Setup and Teardown
- Test Data
- **Test Doubles**

Isolation of Test Cases

Achieving Test Case Isolation

- Don't write complex tests
- Don’t share complex test data
- **Don’t use complex objects**



Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Test Doubles

Generic term for object that stands in for a real object during a test

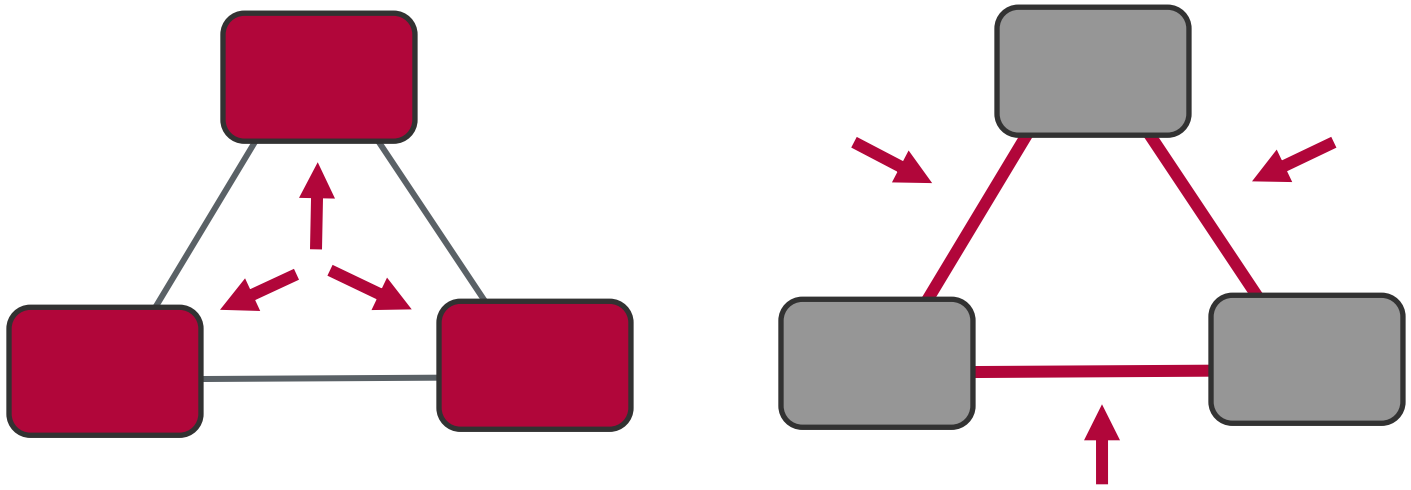
- Think “stunt double”
- Purpose: automated testing

Used when

- Real object is unavailable
- Real object is difficult to access or trigger
- Real object is slow or expensive to run
- An application state is required that is challenging to create

Verifying Behavior During a Test

- Usually: test system state **after** a test
  - Only the result of a call is tested, intermediate steps are not considered
- Test doubles: Possibility to test **detailed system behavior**
  - E.g. How often a method is called, in which order, with which parameters



Ruby Test Double Frameworks

Many (Ruby) frameworks available:

- RSpec-mocks (http://github.com/rspec/rspec-mocks)
- Mocha (https://github.com/freerange/mocha)
- FlexMock (https://github.com/jimweirich/flexmock)

A collection of mocking frameworks (as well as many others):

- https://www.ruby-toolbox.com/categories/mocking

We recommend **RSpec-Mocks** as it shares a common syntax with RSpec

`require("rspec/mocks/standalone")` imports the mock framework.
Useful for exploring in *irb*.

Stubs

- Method call on the real object does not happen
- Returns a predefined value if called
- Strict by default (error when messages received that have not been allowed)

```
dbl = double(“user”)
allow(dbl).to receive_messages( :name => “Fred”, :age => 21 )
expect (dbl.name).to eq(“Fred”) #this is not really a good test :)
dbl.height #raises error (even if your original object had that property)
```

- Alternatively, if all method calls should succeed: **Null object double**

```
dbl = double(“user”).as_null_object
dbl.height # this is ok! Returns itself (dbl)
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/null-object-doubles

Spies

- Stubs with *Given-When-Then* structure
- Allows to expect that a message has been received after the message call

```
dbl = spy("user")
dbl.height
dbl.height
expect(dbl).to have_received(:height)
```

This pattern for tests is also called **arrange-act-assert**

- Alternatively, spy on specific messages of real objects

```
user = User.new
allow(user).to receive(:height)          # Given a user
user.measure_size                        # When I measure the size
expect(user).to have_received(:height) # Then height is called
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/spies

Mocks

Mocks are Stubs with attitude

- Demands that mocked methods are called

```
book = double("book", :title => "The RSpec Book")
expect(book).to receive(:open).once # 'once' is default
book.open # this works
book.open # this fails
```

- Or as often as desired

```
user = double("user")
expect(user).to receive(:email).exactly(3).times
expect(user).to receive(:level_up).at_least(4).times
expect(user).to receive(:notify).at_most(3).times
```

- If test ends with expected calls missing, it fails!

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/configuring-responses/returning-a-value

Stubs vs. Mocks

HPI

Stub (passive)

- Returns a predetermined value for a method call

```
dbl = double("a user")
allow(dbl).to receive (:name) => { "Fred" }
expect (dbl.name).to eq("Fred") #this is not really a good test :)
```

Mock (more aggressive)

- In addition to stubbing: set a “message expectation”
- If expectation is not met, i.e. method is not called: test failure

```
dbl = double(“a user”)
expect(dbl).to receive(:name)
dbl.name      #without this call the test would fail
```

In **RSpec** the *allow* keyword refers to a stub, *expect* to a mock.
This will vary by framework.

Stubs don‘t fail your tests, mocks can!

Partially Stubbing Instances

- Sometimes you want only part of objects to be stubbed
  - Only expensive methods might need stubbing
- **Extension of a real object** instrumented with stub behaviour
- “Partial test double” (in RSpec terminology)

```
s = "a user name" # s.length == 11
allow(s).to receive(:length).and_return(9001)
expect (s.length).to eq(9001) # the method was stubbed
s.capitalize! # this still works, only length was stubbed
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/partial-test-doubles

Method Stubs with Parameters

- Test that methods are called with **correct parameters**
- Failure when calling stub with wrong parameters
- A mock/expectation will only be satisfied when called (and arguments match)

```
calc = double("calculator")
allow(calc).to receive(:double).with(4).and_return(8)
expect(calc.double(4)).to eq(8) # this works
```

- Calling mock with wrong parameters fails:

```
dbl = double("spiderman")
# anything matches any argument
expect(dbl).to receive(:injury).with(1, anything, /bar/)
dbl.injure(1, 'lightly', 'car') # this fails, "car" does not match /bar/
```

These are only a few of the matchers *RSpec-mocks* provides

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/setting-constraints/matching-arguments

Raising Errors

- A stub can raise an error when it receives a message
- Allow easier **testing of exception handling**

```
dbl = double()
allow(dbl).to receive(:foo).and_raise("boom")
dbl.foo # This will fail with:

#     Failure/Error: dbl.foo
#     RuntimeError:
#       boom
```

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/configuring-responses/raising-an-error

Verifying Doubles

- **Stricter** alternative to normal doubles
- Check that methods being stubbed are present on underlying object
- Verify that provided arguments are supported by method signature

```
class Post
  attr_accessor :title, :author, :body
end

post = instance_double("Post") # reference to the class Post
allow(post).to receive(:title)
allow(post).to receive(:message).with (‘a msg’) # this fails (not defined)
```

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/verifying-doubles

Test Doubles Pro and Contra

Disadvantages

- Mock objects need to **accurately model mocked object behavior**
- Risk to test a value set by a test double (false positives)
- Run out of sync with real implementation
  - Brittle while refactoring

Best practice: try to minimize the amount of mocked objects. *(why?)*

Advantages

- Test focused on behavior
- Speed (e.g. not having to use an expensive database query)
- Isolation of tests

Summary

Test run steps

- Setup & teardown
- Test run process
- Test Data
  - Principles
  - Fixtures vs factories

Test doubles

- Use cases & goals
- Mocks
- Stubs
- Spy
- Pros & Cons

Testing Tests

Software Engineering II
WS 2020/21

Enterprise Platform and Integration Concepts

Agenda

Advanced Concepts & **Testing Tests**

- Test Coverage
- Fault Seeding
- Mutation Testing
- Metamorphic Testing

Test Coverage

Most commonly used metric for evaluating test suite quality

- Test coverage = executed code during test suite run ÷ all code * 100
  - e.g. 85 loc / 100 loc = 85% test coverage

Line coverage

- Absence of line coverage indicates potential problems
- **(High) line coverage means very little**
- In combination with good testing practices, coverage might say something about test suite reach
- Circa 100% test coverage is a by-product of BDD

Measuring Code Coverage

Most common approaches

- Line coverage
- Branch coverage

Tool

- SimpleCov Ruby tool
- Uses line coverage

```
if (i > 0); i += 1 else i -= 1 end
```

-> 100% line coverage even if one branch is not executed

Test Tips

Independence

- Of external test data
- Of other tests (and test order)

Repeatability

- Same results each test run
- Potential Problems
  - Dates, e.g. Timecop (https://github.com/travisjeffery/timecop)
  - Random numbers
  - Type and state of test database
  - Type of employed library depending on system architecture

Test Tips

Clarity

- Test **purpose should be immediately clear**
- Tests should be small, simple, readable
- Make it clear how the test fits into the larger test suite

Worst case:

```
it "sums to 37" do
  expect(37).to eq(User.all_total_points)
end
```

Better:

```
it "rounds total points to nearest integer" do
  User.add_points(32.1)
  User.add_points(5.3)
  expect(37).to eq(User.all_total_points)
end
```

Test Tips

Conciseness

- Use the minimum amount of code and objects
- But: Clear beats short
- Writing the minimum required amount of tests for a feature

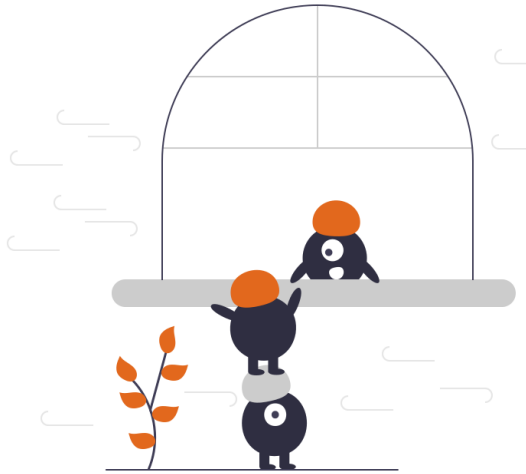
-> Test suite will be faster

```
def assert_user_level(points, level)
  user = User.create(points: points)
  expect(level).to eq(user.level)
end

it test_user_point_level
  assert_user_level(   0, "novice")
  assert_user_level(   1, "novice")
  assert_user_level( 500, "novice")
  assert_user_level( 501, "apprentice")
  assert_user_level(1001, "journeyman" )
  assert_user_level(2001, "guru")
  assert_user_level( nil, "novice")
end
```

- Rails Test Prescriptions. Noel Rappin. 2010. p. 277. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Conciseness: #Assertions per Test



If a single model method call results in many model changes:

- High number of assertions **->** High clarity and cohesion
- High number of assertions **->** Low test independence

- **Use context & describe and have single assertion per test**

Test Tips

Robustness

- Underlying code is correct **->** test passes
- Underlying code is wrong **->** test fails
- *Example:* view testing

```
describe "the signin process", :type => :feature do
  it "signs me in (text version)" do
    visit '/dashboard'
    expect(page).to have_content “My Projects”
  end
  # version below is more robust against text changes
  it "signs me in (css selector version)" do
    visit '/dashboard'
    expect(page).to have_css "h2#projects"
  end
end
```

Test Tips

Robustness

- Reusable code increases robustness
- E.g. constants instead of magic numbers

```
def assert_user_level(points, level)
  user = User.make(:points => points)
  expect(level).to eq(user.level)
end

def test_user_point_level
  assert_user_level(User::NOVICE_THRESHOLD + 1, "novice")
  assert_user_level(User::APPRENTICE_THRESHOLD + 1, "apprentice")
  # ...
end
```

- Be aware of tests that always pass regardless of underlying logic

- Rails Test Prescriptions. Noel Rappin. 2010. p. 278. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Troubleshooting

Reproduce the error

- **Write a test!** (and send it to someone else?)

Inspect recent changes

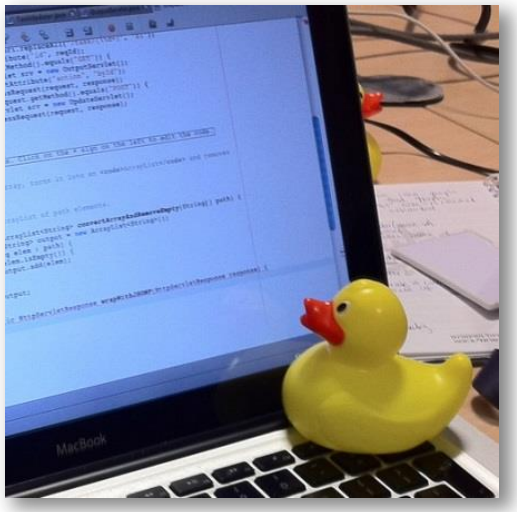
- Isolate commit/change that causes failure

Isolate the failure

- `thing.inspect`
- Add additional assertions to your test
- `save_and_open_page` (take a snapshot of a page)

Explain to someone else

- Rubber duck debugging



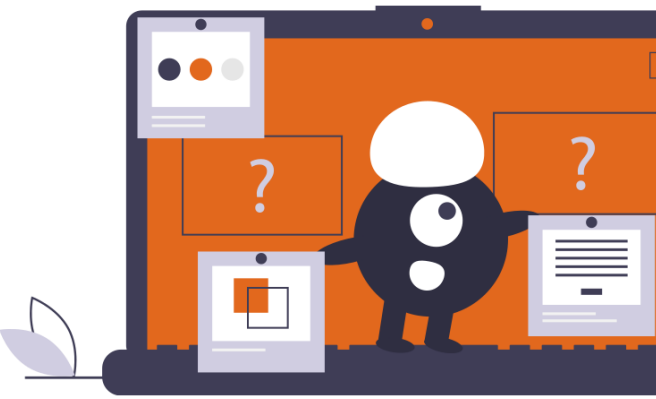
Also refer to "regression testing" aka "non regression testing" *(why?)*

http://commons.wikimedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg

Manual Fault Seeding

Conscious introduction of faults into the program

- Run tests
- Minimum 1 test should fail



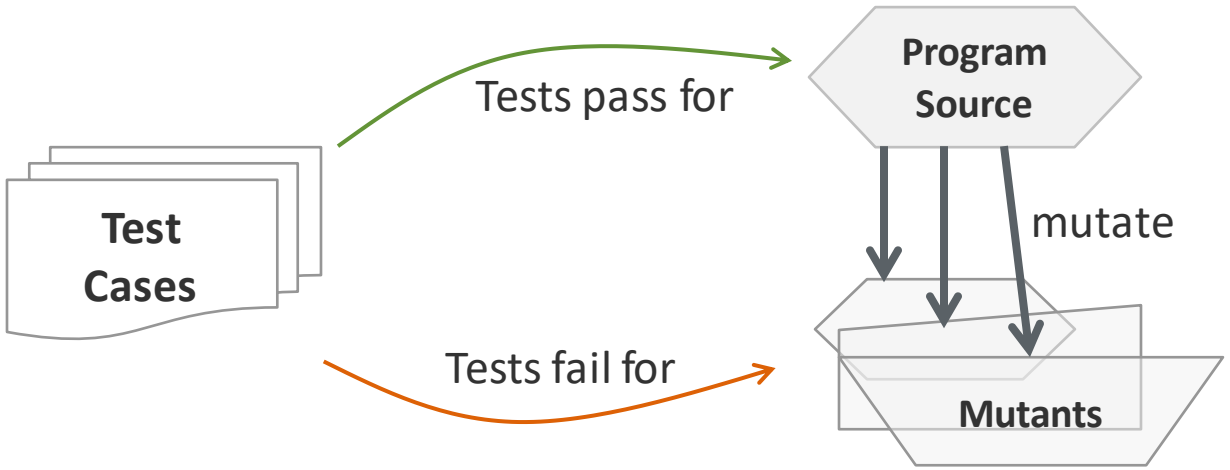
If no test fails, then a test is missing

- Possible even with 100% line coverage
- Asserts functionality coverage

Mutation Testing

Mutant: Modified version of the program with small change

- Tests correctly cover code -> Test should notice change and fail



next_month:

```
if month > 12 then
  year += month / 12
  month = month % 12
end
```

```
if not month > 13 then
  year -= month / 12
  month = month % 12
end
```

- **Mutation Coverage**: How many mutants did not cause a test to fail? Asserts functionality & behavior coverage
  - For Ruby: *Mutant* (https://github.com/mbj/mutant)

Metamorphic Testing

When testing, often hard to find test oracle

- Establish whether a test has passed or failed
- Require understanding of input-output-relation
- May be more convenient to **reason about relations between outputs**

Compare outputs of program runs

- Describe inherent behavior of the program
- No need to know exact outputs

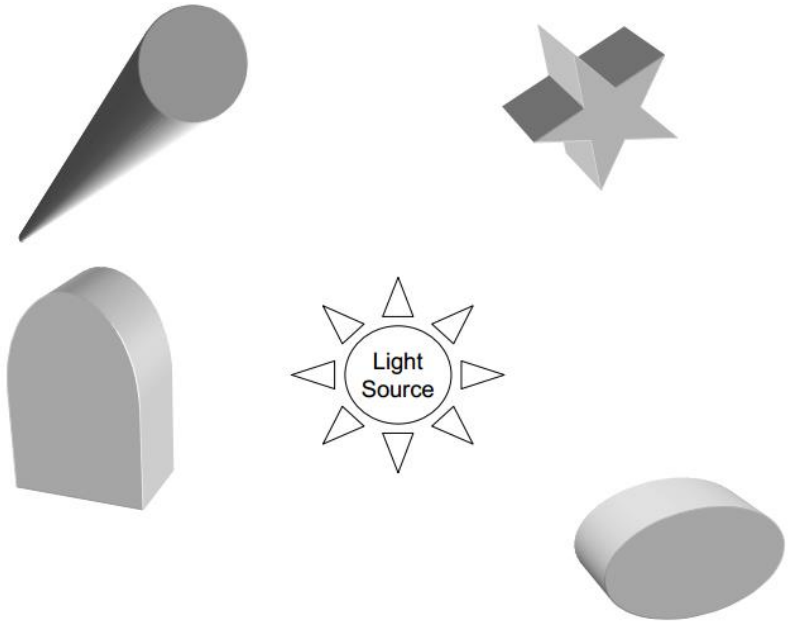
Example: Rendering Lighting

Not easy to verify all pixels were rendered correctly

Use relations of outputs for test cases

Position of light source changes

- Points closer to light source will be brighter
  - Exception: White pixels
- Points further away from light source will be darker
  - Exception: Black pixels
- Points hidden behind other objects don't change brightness



Summary

Test Quality

- Test Coverage
- Fault Seeding
- Mutation Testing
- Metamorphic Testing

Further Reading

- http://betterspecs.org – Collaborative RSpec best practices documentation effort
- *Everyday Rails Testing with RSpec* by Aaron Sumner, leanpub
- *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends* by David Chelimsky et al.
- *Quizzes*
  - http://www.codequizzes.com/rails/rails-test-driven-development/controller-specs
  - http://www.codequizzes.com/rails/rails-test-driven-development/model-specs