Issue #02

Uncle John's Bathroom Reader

RSpec Basics 2

by Speedy Spec

Continuing the theme of RSpec Basics from last time around, in this edition of **Uncle John's Bathroom Reader**, we're going to discuss **mocking**. Just like our last edition, this is by no means comprehensive and you can find more details and links at the bottom of this flyer.

The concept of **mocking** and the concept of setting **expectations** are very similar in the RSpec world, let's just talk about mocking for now. In the following example, we have a **`User`** object, upon which we want to add a **`premium?`** method.

```
describe User do
  describe '#premium?' do
    # Using `#be_premium` which is an RSpec helper that invokes `#premium?`
    it { should be_premium }
  end
end
```

In an earlier version of the application, we might have had **`User#state`** return **`:premium`** and check that in **`premium?`** but here we want an **`AccountState`** object to encapsulate more complex behaviors.

```
describe '#premium?' do
  let(:state) do
    state = mock('AccountState Paid Mock')
    state.stub(:paid? => true)
    state
  end

  before :each do
    subject.stub(:state).and_return(state) # Alternate stubbing style
  end

  it { should be_premium }
end
```

In the above example we create an **`AccountState`** mock object and then use RSpec's **`#stub`** method to make the mock object behave the way we want it to behave for our example. We then use the **`#stub`** method again to attach it to our User object (the subject).

You might wonder why we didn't use a tool like FactoryGirl to create a **`User`** and **`AccountState`** object. For a unit test we should avoid using slow/expensive factories. We want the **`premium?`** method to ask the **`AccountState`** object as few questions as possible. Methods which interrogate an object too much are usually candidates for refactoring or are an indication that objects are too coupled in their implementations.

That's it for this issue of **Uncle John's Bathroom Reader**, now go wash your hands.

RSpec Mockery

Mocking methods:

```
m = mock('Mock Object')
m.stub(:upgrade).and_return(false)
```

```
m = mock('Mock Object 2')
m.stub(:upgrade => false)
```

```
m = mock('Mock Object 3',
                :upgrade => false)
```

Setting Responses:

```
m = mock('Mock Object 4')
m.stub(:raiseit).and_raise(ErrorObj)
m.stub(:throwit).and_raise(:failboat)
m.stub(:yieldit).and_yield(:value)
```

When to Factory, and when to mock:

Generally in unit tests you should be using mocking more than factories. In integration tests however, it's usually more useful to use factories provided by FactoryGirl.

If you find yourself stubbing multiple methods on a mock object, think about whether using a factory makes sense, compared to refactoring/simplifying the method that is being tested.

RSpec Mocks code and documentation: https://github.com/rspec/rspec-mocks

Collection of tips and tricks for writing better specs: http://betterspecs.org/

Send comments and suggestions for future issues to: **ujbr@lookout.com**