

Unit Testing J2EE from JRuby

Evan Light

<http://evan.tiggerpalace.com>



Who I am

- Professional developer since 1996
- Java since 1999
- J2EE since 2000
- Ruby since 2006

Some yutz
with Keynote and a
remote control

Pop Quiz

But why test?

Find bugs

Prove out design

Simplify refactoring

Document behavior

Document behavior?

Time for a visit
to the eye doctor

#1

```
import junit.cookbook.Money;

@Test public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

From the [JUnit Cookbook](#)

#2

```
import "junit.cookbook.Money"

describe "Money" do
  describe "in Swiss Francs" do
    before do
      @twelve = Money.new 12, "CHF"
      @fourteen = Money.new 14, "CHF"
      @expected = Money.new 26, "CHF"
    end

    it "should be additive" do
      @twelve.add(@fourteen).should == @expected
    end
  end
end
```

Why Ruby?

Why not?

Why do we learn new
programming
languages?

Better runtime performance

Greater developer productivity

Better user experience

Expand technical outlook

Why?

To earn money!!!

Story Time, Part I

(a.k.a How I found Ruby)

The Setup

- Secure enterprise document mgmt system
- 6 person team
- Cannot add people
- Cannot reduce scope
- Cannot reduce quality

The J2EE Estimate

- 6 person-year schedule estimate
- 6 people / 6 person-years = 1 year
- Customer already waited 1 year!

How can we do better?

Increase productivity

Enter Ruby on Rails

The Outcome

3 people x 3 months =

50% of project

JRuby



Ruby

... and then some

Ruby objects are Java objects

Java garbage collection

Java threading

Ruby on Rails apps as WARs

... and **Java Integration**

But u want to put ur
Ruby in my **Java**?!

Not exactly...

We're here to talk about J2EE testing

(And it only took him 31 slides to say that)

My “favorite”

EJBs

EJB 3.0

Container DI

Encourages difficult-to-test idioms

- Private fields

- Private static final fields

```
@Stateless
public class DeepThought implements DeepThoughtLocal {

    private static final Logger LOGGER = Logger.getLogger(DeepThought.class);

    @Resource(name="theAnswer")
    private String theAnswer;

    @Resource(name="planetName")
    private String planetName;

    @Resource
    private SessionContext ctx;

    public String computeQuestion() {
        PlanetConstructor pc = ctx.lookup("planetConstructor/local");
        LOGGER.info("Preparing to build " + planetName);
        Planet planet = pc.buildPlanetToDeriveQuestion(planetName);
        LOGGER.info("Completed construction of " + planetName);
        return planet.derive(theAnswer);
    }
}
```

How do you test that?!

Solution #1
Don't write it

What if I
inherited it?

Solution #2
Work around it

You could...

... write a pre-compiler and integrate it into your ANT/Maven tasks, that would use Java reflection to introspect on each Java class under test. The precompiler could generate code for a setter and getter wrapper for each non-public field and a wrapper for each non-public method that, in turn, tell the Java SecurityManager that it is kosher to use Java reflection to access/invoke the respective non-public field/method and then do their getting/setting/invoking via reflection. You could even write this as part of an Eclipse-plugin that generates the code earlier so that you could have access to these wrappers while you're writing your tests in the first place. Hey, that may even give you type-checking while you're writing your tests which would be awfully nice. But wouldn't the resulting test code be a little awkward to write and not particularly readable for developers down the line?

Solution #3: JRuby

With a little help
from jrsplenda

This just works

```
deepThought.planetName = "Earth"
```

So how do you test this?

```
@Stateless
public class DeepThought implements DeepThoughtLocal {

    private static final Logger LOGGER = Logger.getLogger(DeepThought.class);

    @Resource(name="theAnswer")
    private String theAnswer;

    @Resource(name="planetName")
    private String planetName;

    @Resource
    private SessionContext ctx;

    public String computeQuestion() {
        PlanetConstructor pc = ctx.lookup("planetConstructor/local");
        LOGGER.info("Preparing to build " + planetName);
        Planet planet = pc.buildPlanetToDeriveQuestion(planetName);
        LOGGER.info("Completed construction of " + planetName);
        return planet.derive(theAnswer);
    }
}
```

One step at a time

jrsplenda

JRuby helpers to simplify testing Java code

Hosted on GitHub

<http://github.com/elight/jrsplenda/tree/master>

Designed for Mocha (hence the name)

Field and Method helpers

At **runtime**, generates wrappers to:

- Set non-public Fields

- Invoke non-public Methods

- JRuby provides non-public Field getters

Caveats

Cannot wrap a final Field

As of JRuby 1.1.2, non-public field getters

Provided for Java instance Fields

Not provided for Java class Fields

Caveats (cont'd)

May cause:

Diarrhea

Upset stomach

Test setup of DI

```
@computer = DeepThought.new  
wrap_java_fields @computer  
  
# Now pretend to be the EJB container  
@computer.theAnswer = "42"  
@computer.planetName = "Earth"  
@computer.ctx = @session_context
```

Mock objects at 50k feet

Special kind of stub

Set expectations on a mock

Inject mock into path of code to be tested

Test fails if expectation is not met

Good read: [Mock Roles, Not Objects](#)

jrsplenda mock helper

Borrowed from Ola Bini's JtestR

Create mock Java objects within JRuby

Set expectations in JRuby

DI mocks from JRuby into Java

Exercise Java code from JRuby

In future releases of jrsplenda

Partial mocks

Until implemented, tests must include
additional stubbing

Support for other Ruby mocking frameworks

Test setup of mocking

```
CLASSES_TO MOCK = [  
    "magarathean.PlanetConstructor",  
    "magarathean.Planet",  
    "javax.ejb.SessionContext"  
]
```

```
CLASSES_TO MOCK.each { |c| splenda_mock_attr c }
```

We've made Java
testing simpler

Now let's make it more
expressive

Remember this?

```
import "junit.cookbook.Money"

describe "Money" do
  describe "in Swiss Francs" do
    before do
      @twelve = Money.new 12, "CHF"
      @fourteen = Money.new 14, "CHF"
      @expected = Money.new 26, "CHF"
    end

    it "should be additive" do
      @twelve.add(@fourteen).should == @expected
    end
  end
end
```

RSpec

<http://rspec.info/>

Ruby Example-driven (Testing) DSL

Couples documentation with behavior

Walkthrough

```
import "junit.cookbook.Money"

describe "Money" do
  describe "in Swiss Francs" do
    before do
      @twelve = Money.new 12, "CHF"
      @fourteen = Money.new 14, "CHF"
      @expected = Money.new 26, "CHF"
    end

    it "should be additive" do
      @twelve.add(@fourteen).should == @expected
    end
  end
end
```

Putting it all together

```

include Java

require 'rubygems'
require 'jrsplenda'

describe "A computer that can compute the Ultimate Answer" do
  include JRSplenda::AllHelpers

  CLASSES_TO_MOCK = [
    "magarathian.PlanetConstructor",
    "magarathian.Planet",
    "javax.ejb.SessionContext"
  ]

  before do
    # Creates @planet_constructor and @session_context mocks
    CLASSES_TO_MOCK.each { |c| splenda_mock_attr c }

    @computer = DeepThought.new
    wrap_java_fields @computer

    # Now pretend to be the EJB container
    @computer.theAnswer = "42"
    @computer.planetName = "Earth"
    @computer.ctx = @session_context

    # For now, jrsplenda requires that you stub the whole impl
    # In the next release, jrsplenda should provide "Partial Mocks"
    @session_context.stubs(:lookup)
      .and_returns(@planet_constructor)
    @planet_constructor.stubs(:buildPlanetToDeriveQuestion)
      .and_returns(@planet)
    @planet.stubs(:derive).and_returns("What is 6 times 9?")
  end

  it "should lookup a Planet Constructor" do
    @session_context.expects(:lookup)
      .with("planetConstructor/local")
      .and_returns(@planet_constructor)
    @computer.computeQuestion()
  end

  it "should build Earth" do
    @planet.expects(:buildPlanetToDeriveQuestion)
      .with(@computer.planetName)
    @computer.computeQuestion()
  end

  it "should have Earth derive the Ultimate Question" do
    @planet.expects(:derive)
  end
end

```

Specification (beginning)

```
include Java

require 'rubygems'
require 'jrsplenda'

describe "A computer that can compute the Ultimate Answer" do
  include JRSpplenda::AllHelpers

  CLASSES_TO MOCK = [
    "magarathean.PlanetConstructor",
    "magarathean.Planet",
    "javax.ejb.SessionContext"
  ]
end
```

Example setup

```
before do
  # Creates @planet_constructor and @session_context mocks
  CLASSES_TO_MOCK.each { |c| splenda_mock_attr c }

  @computer = DeepThought.new
  wrap_java_fields @computer

  # Now pretend to be the EJB container
  @computer.theAnswer = "42"
  @computer.planetName = "Earth"
  @computer.ctx = @session_context

  # For now, jrsplenda requires that you stub the whole impl
  # In the next release, jrsplenda should provide "Partial Mocks"
  @session_context.stubs(:lookup)
    .and_returns(@planet_constructor)
  @planet_constructor.stubs(:buildPlanetToDeriveQuestion)
    .and_returns(@planet)
  @planet.stubs(:derive).and_returns("What is 6 times 9?")
end
```

The examples

```
it "should lookup a Planet Constructor" do
  @session_context.expects(:lookup)
    .with("planetConstructor/local")
    .and_returns(@planet_constructor)
  @computer.computeQuestion()
end

it "should build Earth" do
  @planet_constructor.expects(:buildPlanetToDeriveQuestion)
    .with(@computer.planetName)
  @computer.computeQuestion()
end

it "should have Earth derive the Ultimate Question" do
  @planet.expects(:derive)
  @computer.computeQuestion()
end
end
```



By PhotoGraham on Flickr

[Creative Commons License](#)