Next Level Spock

Getting the most out of testing

What we'll talk about

- Unit tests as "Executable Specifications"
- Extending Spock
- Spock and Grails 2.0
- Hidden Treasures

Executable Specifications

Why only functional & acceptance tests?

Executable Specifications?

"An executable specification captures the functional behavior of a system. […] a complete executable specification must first include design **documentation**. There must be an **executable** model, and there must be a **verification** environment within which to run the model."

"Executable Specs: What Makes One,and How are They Used?"
Peter J. Schubert, Lev Vitkin and Frank Winters
http://delphi.com/pdf/techpapers/2006-01-1357.pdf

Traditional Tools

- Cucumber
- FitNesse
- JBehave
- EasyB
- Many more…

Natural Language Based

Given I am at the login page
When I enter the username “admin”
And I enter the password “admin”
And I click the login button
Then I am taken to admin page

Natural Language Based

Given I am **at** the **login page**
When I **enter** the **username** "**admin**"
And I **enter** the **password** "**admin**"
And I **click** the **login button**
Then I am **taken** to **admin page**

information is parsed out

Pros of Natural Language

- Stories are in “user” language
- Readable by “the business”
- Clearly serve as documentation
- **Forces** developers to think like users
- Nice looking reports

Pros of Natural Language

- Stories are in “user” language
- Readable by “the business”
- Clearly serve as documentation
- **Forces** developers to think like users
- Nice looking reports (that no one reads)

Cons of Natural Language

- Requires pattern matching/parsing (much indirection)
- It's a different environment (i.e. context switching cost)
- Expensive compared to typical unit tests
- Little tool support / analysis
- *Sometimes* is too much effort

Natural Language Executable Specs

Primary strength is communicating / collaborating with non programmers and projecting a user oriented view of the system.

Excellent fit for <u>acceptance testing</u> due to audience.

Natural Language Executable Specs

They **definitely** have their place.

But, there **IS** a cost in development overhead.
(jury is out on maintenance overhead)

Unit vs. Acceptance Testing

"Acceptance tests are not unit tests. Unit tests are written *by* programmers *for* programmers [...]. Acceptance tests are written *by* the business *for* the business."

"Unit tests and acceptance tests are documents first, and tests second. Their primary purpose is to formally document the design, structure and behavior of the system".

Robert C. Martin - The Clean Coder - 2011
http://www.informit.com/store/product.aspx?isbn=0137081073

Unit vs. Acceptance Testing

The purpose is the same.
(documents first, tests second)

The difference is audience, and scope.
(by programmers, for programmers)

Unit vs. Acceptance Testing

Why do we think about them
in such different ways?

We should aim for just as much clarity.

We need a tool that...

- Promotes readability and clarity (like natural language tools)
- Promotes "user" thinking (context, stimulus, expectations)
- Programmer productivity focussed
- Familiar environment and good tool support

We need a tool that...

- Promotes readability and clarity (like natural language tools)
- Promotes "user" thinking (context, stimulus, expectations)
- Programmer productivity focussed
- Familiar environment and good tool support

We need cheap programmer oriented executable specifications

Enter
Spock & Groovy

Spock

- Programmer focussed
- Familiar environment (classes, methods etc.)
- Promotes clarity (structural blocks)
- Removes noise (concise syntax)
- Good tool support

Spock

- Programmer focussed
- Familiar environment (classes, methods etc.)
- Promotes clarity (structural blocks)
- Removes noise (concise syntax)
- Good tool support

A good middle ground for programmer executable specs

Previous Natural Language Example

Given I am **<u>at</u>** the **<u>login page</u>**
When I **<u>enter</u>** the **<u>username</u>** "**<u>admin</u>**"
And I **<u>enter</u>** the **<u>password</u>** "**<u>admin</u>**"
And I **<u>click</u>** the **<u>login button</u>**
Then I am **<u>taken</u>** to **<u>admin page</u>**

Spock & Geb

```
class LoginSpec extends GebSpec {
    def "user can login with good credentials"() {
        given:
        to LoginPage

        expect:
        at LoginPage

        when:
        username = "admin"
        password = "admin"

        and:
        loginButton.click()

        then:
        at AdminPage
    }
}
```

Geb is a Groovy web automation library
http://www.gebish.org

Geb is...

Not much more than aggressive use of Groovy language features to remove noise and duplication, on top of a library for interacting with browsers.

Focus is on clarity of intent instead of browser mechanics.

The primary goal is low cost, low ceremony, executable specifications for web applications.

Geb is...

a Domain Specific Language.

You can, and should, apply the same concepts to your unit tests to make them more specification like.

You can, and should, use DSL techniques to focus on clarity of <u>intent</u> instead of mechanics.

Example

grails Link Generator Spec

What happened?

- Removed duplication
- Extracted setup/detail out of feature methods
- Utilised statefulness of tests
- Focussed on differences between feature methods
- Isolated *how* and focussed on *what*

The Test DSL

```groovy
class LinkGeneratorWithUrlMappingsSpec extends Specification {
  def baseUrl = "http://myserver.com/foo"
  def context = null
  def path = "welcome"
  def action = [controller:'home', action:'index']

  def mappings = {
    "/${this.path}"(this.action)
  }

  def link = new LinkedHashMap(action)

  protected getGenerator() {
    def generator = new DefaultLinkGenerator(baseUrl, context)
    def evaluator = new DefaultUrlMappingEvaluator(new MockServletContext())
    // apply url mappings
    generator
  }

  protected getUri() {
    generator.link(link)
  }
}
```

The Test DSL

```
class LinkGeneratorWithUrlMappingsSpec extends Specification {
  def baseUrl = "http://myserver.com/foo"
  def context = null
  def path = "welcome"
  def action = [controller:'home', action:'index']

  def mappings = {
    "/${this.path}"(this.action)
  }

  def link = new LinkedHashMap(action)

  protected getGenerator() {
    def generator = new DefaultLinkGenerator(baseUrl, context)
    def evaluator = new DefaultUrlMappingEvaluator(new MockServletContext())
    // apply url mappings
    generator
  }

  protected getUri() {
    generator.link(link)
  }
}
```

inputs with defaults

The Test DSL

```
class LinkGeneratorWithUrlMappingsSpec extends Specification {
  def baseUrl = "http://myserver.com/foo"
  def context = null
  def path = "welcome"
  def action = [controller:'home', action:'index']

  def mappings = {
    "/${this.path}"(this.action)
  }

  def link = new LinkedHashMap(action)

  protected getGenerator() {
    def generator = new DefaultLinkGenerator(baseUrl, context)
    def evaluator = new DefaultUrlMappingEvaluator(new MockServletContext())
    // apply url mappings
    generator
  }

  protected getUri() {
    generator.link(link)
  }
}
```

actions as

properties

The Tests

```
class LinkGeneratorWithUrlMappingsSpec extends Specification {
  def "link is prefixed by the deployment context …"() {
    when: context = "/bar"
    then: uri == "$context/$path"
  }

  def "absolute links are prefixed by the base url …"() {
    when: context = "/bar"
    and: link.absolute = true
    then: uri == "$baseUrl/$path"
  }

  def "absolute links are generated when a relative …"() {
    when: context = null
    then: uri == "$baseUrl/$path"
  }
}
```

Looks like a specification,
not a traditional unit test

More Examples

Techniques

- Utilise state (i.e. ivars with defaults)
- Getter methods that construct using state
- Default method params
- Named arguments
- Configuration "callback" closures
- @Lazy for one time construction

and more...

Techniques

- Utilise state (i.e. ivars with defaults)
- Getter methods that construct using state
- Default method params
- Named arguments
- Configuration callback closures
- @Lazy for one time construction

Standard Groovy DSL implementation techniques

In short...

By creating a micro DSL for the **how** of the test, we can use specification like language for the **what** of the test in order to get the kind of clarity of intent we find in acceptance tests.

Why?

- Clearly documents the behavior of what is being tested
- Promotes "single responsibility"
- Forces "user" thinking (i.e. inputs and outputs instead of implementation)
- Long term maintainability (reduced duplication)

Extending Spock

JUnit Rules

JUnit RULES

- (Anti-)Pattern: Common test logic goes into base class
- How many base classes can you have?
- JUnit 4.7 introduces *Rules*

JUnit Rules

"A TestRule is an alteration in how a test method, or set of test methods is run and reported. A TestRule may add additional checks that cause a test that would otherwise fail to pass, or it may perform necessary setup or cleanup for tests, or it may observe test execution to report it elsewhere."

"[TestRules are] easily shared between projects and classes."

JUnit 4.9 Javadoc
http://kentbeck.github.com/junit/javadoc/4.9/org/junit/rules/TestRule.html

Rule Examples

- TemporaryFolder
- ErrorCollector

Rule Examples

- TemporaryFolder
- ErrorCollector

Demo

Advanced Rules

- @ClassRule (JUnit 4.9)
- RuleChain (JUnit 4.10)

Advanced Rules

- @ClassRule (JUnit 4.9)
- RuleChain (JUnit 4.10)

Demo

Spock & Grails

Now a first class Citizen

Spock for Grails 1.x works by **duplicating** classes from Grails core.

Over time they have gotten out of sync.

Grails 2.0 features a brand new unit testing infrastructure based on mixins instead of inheritance.

This means that Spock for Grails now uses the **exact** same code as the regular plain JUnit tests.

Plain JUnit

```
package org.example

import grails.test.mixin.*

@TestFor(PostController)
class PostControllerTests {
    void testIndex() {
        controller.index()
        assert "/post/list" == response.redirectedUrl
    }
    ...
}
```

Spock

```
package org.example

import grails.test.mixin.*

@TestFor(PostController)
class PostControllerSpec extends spock.lang.Specification {
    def "Index action should redirect to list page"() {
        when:
        controller.index()

        then:
        response.redirectedUrl == "/post/list"
    }
    ...
}
```

Demo

Spock & GRails 2.0

This means that...

There's no reason not to use Spock for your Grails unit, integration and functional (with Geb) tests.

For more...

http://grails.org/doc/2.0.0.RC1/guide/testing.html

Extending Spock

Spock Extensions

Spock Extensions

- *The* way to extend Spock
- Two flavors:
  - Annotation-driven
  - Global
- Abilities:
  - Register interceptors
  - Register listeners
  - Alter the spec's *model*

Extensions small...

- @Ignore/@IgnoreRest/@IgnoreIf
- @Timeout
- @Stepwise
- @AutoCleanup
- @RevertMetaClass
- @Use
- @Rule/@ClassRule

...AND LARGE

- spock-spring
- spock-guice
- spock-tapestry
- spock-unitils
- spock-arquillian

Extension vs. Integration

- spock-grails
- spock-griffon
- spock-maven

Report extension

```
class AccountSpec extends Specification {
    def "withdraw an amount"() {
        given: "an account with a balance of five euros"
        def account = new Account(5.0)

        when: "two euros are withdrawn"
        account.withdraw(2.0)

        then: "three euros remain in the account"
        account.balance == 3.0
    }
}
```

Report extension

```
class AccountSpec extends Specification {
    def "withdraw an amount"() {
        given: "an account with a balance of five euros"
        def account = new Account(5.0)

        when: "two euros are withdrawn"
        account.withdraw(2.0)

        then: "three euros remain in the account"
        account.balance == 3.0
    }
}
```

No such extension?
Let's write one!

Hidden Treasures

Configuring Spock

- ~/.spock/SpockConfig.groovy
- -Dspock.configuration="/prj/MyConfig.groovy"

```
runner {
    filterStackTrace true
    include org.prj.tests.Database
    exclude org.prj.tests.UI
    optimizeRunOrder true
}
```

- Extensible, POJO-based configuration

Include/Exclude

Demo

Optimize Run Order

Demo

IDE Diff Dialog

Demo