Spock versus JUnit

Kostis Kapelonis
Athens Greece
JHUG 2 April 2016
A trailer/Quiz
Sample class that checks JPEG files

```java
public class ImageNameValidator {
    public boolean isValidImageExtension(String fileName) { ... }
}
```
Example Usage

ImageNameValidator v = ImageNameValidator();

v.isValidImageExtension("hello.jpg") -> true
v.isValidImageExtension("now.JPg") -> true
v.isValidImageExtension("s.JpEg") -> true
v.isValidImageExtension("wow.png") -> false
What would Spock do?
def "All kinds of JPEG file are accepted"() {
    given: "an image extension checker"
    ImageNameValidator v = new ImageNameValidator();

    expect: "that all jpeg filenames are accepted regardless of case"
    validator.isValidImageExtension(pictureFile)

    where: "sample image names are"
    pictureFile << GroovyCollections.combinations([["sample.", "Sample.", "SAMPLE."], ["j", 'J'], ["p", 'P'], ["e", 'E', '"'], ["g", 'G'])).join()
Test result

Finished after 0.484 seconds

<table>
<thead>
<tr>
<th>Runs: 72/1</th>
<th>Errors: 0</th>
<th>Failures: 0</th>
</tr>
</thead>
</table>

- Checking image name SAMPLE.jpeg (0.016 s)
- Checking image name sample.jpeg (0.000 s)
- Checking image name Sample.jpeg (0.000 s)
- Checking image name SAMPLE.jpeg (0.000 s)
- Checking image name sample.jpeg (0.000 s)
- Checking image name Sample.jpeg (0.000 s)
- Checking image name SAMPLE.jpeg (0.000 s)
- Checking image name sample.jpeg (0.000 s)
- Checking image name Sample.jpeg (0.000 s)
- Checking image name SAMPLE.jpeg (0.000 s)
Try the same with JUnit

The Spock class is 10 LOC and results in 72 test scenarios
Motivation

Why Spock? What is wrong with JUnit?
Spock history

• Created in 2008 by Peter Niederwieser (Gradle)
• Joined by Luke Daley (Gradle)
• Spock 1.0 released in 2015
• Default Test framework in Grails
• Used internally by Gradle, Groovy etc.
• Used by MongoDB, Tapestry, Netflix, JFrog
TestNG and JUnit
Spock (Something new)
Why Spock

Enterprise Testing (JUnit)

Can mock and stub (Mockito)

Behavior Driven (JBehave)

Spock
Why Spock

Value

Project Complexity/Size

Spock

JUnit
Spock for everything

Status quo

JUnit/TestNG

EasyMock/JMock/Mockito

Cucumber/JBehave

Growth of Testing needs

Spock for everything

Spock
Why Spock (parameterized tests)
Spock F.A.Q

First things first
Let’s make 2 things clear
Spock uses the JUnit runner

This means that it is compatible with all existing JUnit tools
Spock FAQ

• How do I include Spock tests in my project?
• How do I run Spock tests?
• How do I debug Spock tests?
• How do I get code Coverage?
• How do I integrate with Sonar?
• How do I ....?
How do I...?

Answer: “the same way you did with JUnit”
Let’s make 2 things clear
Spock can work with **Java**

In fact Spock is written in Java and only has a Groovy front-end (same as Gradle)
Unit tests have different needs

- **Development**
  - Core code compiles
  - Unit tests compile

- **Deployment/shipping**
  - Unit tests run

- **Production**
  - Core code runs
Spock is the default Grails test framework

But it is not tied to Grails, (as Gradle is not tied with Groovy)
Spock can work with Java!
Spock with Java

1. You can add Spock tests to an existing Java project
2. You can keep your JUnit tests
3. You can run them together
4. You can still use Maven, Intellij, Sonar, Eclipse etc.
Gradual Spock acceptance

1. Pure Java
   - Core: Java
   - Build Tests: JUnit, Maven

2. Spock/JUnit
   - Core: Java
   - Build Tests: JUnit, Spock, Maven

3. Spock only
   - Core: Java
   - Build Tests: Spock, Maven
Recap - Spock Facts

• Spock can test Java code
• Spock tests behave as JUnit tests.
Spock versus JUnit

6 Reasons why Spock is better
1. Test structure

Spock enforces the setup-trigger-assert paradigm
A good JUnit test

```java
@Test
public void oneSensorIsTriggered() {
    FireEarlyWarning fireEarlyWarning = new FireEarlyWarning();
    int triggeredSensors = 1;

    fireEarlyWarning.feedData(triggeredSensors);
    WarningStatus status = fireEarlyWarning.getCurrentStatus();

    assertTrue("Alarm sounds", status.isAlarmActive());
    assertFalse("No notifications", status.isFireDepartmentNotified());
}
```
Arrange- Act-assert Pattern

```java
@Test
public void oneSensorIsTriggered() {
    FireEarlyWarning fireEarlyWarning = new FireEarlyWarning();
    int triggeredSensors = 1;

    fireEarlyWarning.feedData(triggeredSensors);
    WarningStatus status = fireEarlyWarning.getCurrentStatus();

    assertTrue("Alarm sounds", status.isAlarmActive());
    assertFalse("No notifications", status.isFireDepartmentNotified());
}
```
What happens in real life
@Test
public void sentinelSet() {
    Jedis j = new Jedis(sentinel.getHost(), sentinel.getPort());

    try {
        Map<String, String> parameterMap = new HashMap<String, String>();
        parameterMap.put("down-after-milliseconds", String.valueOf(1234));
        parameterMap.put("parallel-syncs", String.valueOf(3));
        parameterMap.put("quorum", String.valueOf(2));
        j.sentinelSet(MASTER_NAME, parameterMap);

        List<Map<String, String>> masters = j.sentinelMasters();
        for (Map<String, String> master : masters) {
            if (master.get("name").equals(MASTER_NAME)) {
                assertEquals(1234, Integer.parseInt(master.get("down-after-milliseconds")));
                assertEquals(3, Integer.parseInt(master.get("parallel-syncs")));
                assertEquals(2, Integer.parseInt(master.get("quorum")));
            }
        }
        parameterMap.put("quorum", String.valueOf(1));
        j.sentinelSet(MASTER_NAME, parameterMap);
    } finally {
        j.close();
    }
}
def "If one sensor is active the alarm should sound as a precaution"() {
    given: "that only one fire sensor is active"
    FireEarlyWarning fireEarlyWarning = new FireEarlyWarning()
    int triggeredSensors = 1

    when: "we ask the status of fire control"
    fireEarlyWarning.feedData(triggeredSensors)
    WarningStatus status = fireEarlyWarning.getCurrentStatus()

    then: "only the alarm should be triggered"
    status.alarmActive
    !status.fireDepartmentNotified
}
Spock blocks

- **given**: Creates initial conditions
- **setup**: An alternative name for given:
- **when**: Triggers the action that will be tested
- **then**: Examines results of test
- **and**: Cleaner expression of other blocks
- **expect**: Simpler version of then:
- **where**: Parameterized tests
- **cleanup**: Releases resources
Given – Expect example

def "An empty basket has no weight "() {
    given: "an empty basket"
    Basket basket = new Basket()

    expect: "that the weight is 0"
    basket.currentWeight == 0
}

2. Test readability

Spock tests read like English sentences
def "If one sensor is active the alarm should sound as a precaution"() {
    given: "that only one fire sensor is active"
        [...code here...]
    when: "we ask the status of fire control"
        [...code here...]
    then: "only the alarm should be triggered"
        [...code here...]
}
Enterprise applications
Enterprise applications

- Big codebase (200k+ LOC)
- No developer knows all parts
- Original authors are not in the team
- In development for 2+ years
- In production for 3+ years
Unit tests are specifications
JUnit reports – usual case

<table>
<thead>
<tr>
<th>BadTest</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>scenario1</td>
<td></td>
</tr>
<tr>
<td>scenario2</td>
<td></td>
</tr>
</tbody>
</table>

[REJECTED]
JUnit reports - boring

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensorsAreTriggered</td>
<td>Passed</td>
</tr>
<tr>
<td>everythingIsOk</td>
<td>Passed</td>
</tr>
<tr>
<td>oneSensorIsTriggered</td>
<td>Passed</td>
</tr>
<tr>
<td>twoSensorsAreTriggered</td>
<td>Passed</td>
</tr>
</tbody>
</table>

REJECTED
# FireSensorSpec

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>If all sensors are inactive everything is ok</td>
<td></td>
</tr>
<tr>
<td>If one sensor is active the alarm should sound as a precaution</td>
<td></td>
</tr>
<tr>
<td>If more than one sensors are active then we have a fire</td>
<td></td>
</tr>
</tbody>
</table>
Supercharge your test reports
Spock native reports

Summary:

Created on Sun Jan 25 23:39:45 EET 2015 by Kostis

<table>
<thead>
<tr>
<th>Executed features</th>
<th>Failures</th>
<th>Errors</th>
<th>Skipped</th>
<th>Success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Features:

If all sensors are inactive everything is ok

*Given:* that all fire sensors are off
*When:* we ask the status of fire control
*Then:* no alarm/notice should be triggered

If one sensor is active the alarm should sound as a precaution

*Given:* that only fire sensor is active
*When:* we ask the status of fire control
*Then:* only the alarm should be triggered

If more than one sensors are active then we have a fire

*Given:* that two fire sensors is active
*When:* we ask the status of fire control
*Then:* alarm is triggered and the fire department is notified

APPROVED
Work with non-developers
Reports readable by Testers

Summary:

Created on Sun Jan 25 23:39:45 EET 2015 by Kostis

<table>
<thead>
<tr>
<th>Executed features</th>
<th>Failures</th>
<th>Errors</th>
<th>Skipped</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Features:

If all sensors are inactive everything is ok

Given: that all fire sensors are off
When: we ask the status of fire control
Then: no alarm/notification should be triggered

If one sensor is active the alarm should sound as a precaution

Given: that only fire sensor is active
When: we ask the status of fire control
Then: only the alarm should be triggered

If more than one sensors are active then we have a fire

Given: that two fire sensors is active
When: we ask the status of fire control
Then: alarm is triggered and the fire department is notified
Tests readable by Business Analysts

**Summary:**

Created on Sun Jan 25 23:39:45 EET 2015 by Kostis

<table>
<thead>
<tr>
<th>Executed features</th>
<th>Failures</th>
<th>Errors</th>
<th>Skipped</th>
<th>Success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

**Features:**

- **If all sensors are inactive everything is ok**
  - **Given:** that all fire sensors are off
  - **When:** we ask the status of fire control
  - **Then:** no alarm/notification should be triggered

- **If one sensor is active the alarm should sound as a precaution**
  - **Given:** that only fire sensor is active
  - **When:** we ask the status of fire control
  - **Then:** only the alarm should be triggered

- **If more than one sensors are active then we have a fire**
  - **Given:** that two fire sensors is active
  - **When:** we ask the status of fire control
  - **Then:** alarm is triggered and the fire department is notified
3. Failed tests

Spock knows the context of failed tests
This is a killer feature
A build fails – now what?
JUnit knows only actual result
JUnit knows only actual result

java.lang.AssertionError: 4 times (2 plus 3) is 20 expected:<20> but was:<25>

at com.manning.spock.MultiplierTest.combinedOperationsTest(MultiplierTest.java:22)
Spock knows the context
Spock knows the context

Failure Trace

Condition not satisfied:

```
multi.multiply(4, adder.add(2, 3)) == 20
  |   |   |   |   |
  |  25 |   5   false
  | com.manning.spock.Adder@691a0e79
com.manning.spock.Multiplier@38d9e447
```
Both sides of assert are analyzed
A realistic example

JUnit assert

java.lang.AssertionError: Expected same result expected:<2> but was:<5>
  at com.manning.spock.chapter2.NormalAssert.methods(NormalAssert.java:42)

Caught: Assertion failed:

Groovy assert

assert wordDetector.feedText(text).duplicatesFound().size() == 5

[are, They] 2 false
They are alone. They are a dying race.
com.manning.spock.chapter2.WordDetector@552ee43b
com.manning.spock.chapter2.WordDetector@552ee43b
4. Built-in mocking

JUnit needs Mockito so no JUnit example to compare
Why we need Stubs and Mocks

Real system

Unit test
Our Scenario
Simple Stubbing

given: “a shopping basket”
Basket basket = new Basket()
and: "an empty warehouse"

WarehouseInventory inventory = Stub(WarehouseInventory)
inventory.isEmpty() >> true
basket.setWarehouseInventory(inventory)
inventory.isEmpty() >> true

“When the method isEmpty() is called, ignore the real object and return true”
given: "a basket, a TV and a camera"
Product tv = new
    Product(name:"bravia",price:1200,weight:18)
Product camera = new
    Product(name:"panasonic",price:350,weight:2)
Basket basket = new Basket()

and:"a warehouse with partial availability"
WarehouseInventory inventory =
    Stub(WarehouseInventory) {
        isProductAvailable("bravia",1) >> true
        isProductAvailable("panasonic",1) >> false
        isEmpty() >> false
    }
isProductAvailable("bravia",1) >> true

“When the method isProductAvailable() is called with these arguments, return true”
Argument Matchers

WarehouseInventory inventory = Stub(WarehouseInventory)
inventory.isProductAvailable(_, 1) >> true
basket.setWarehouseInventory(inventory)

(Mockito does not support partial matchers)
isProductAvailable(_,1) >> true

“When the method isProductAvailable() is called with any first argument and 1 as second argument then return true”
Method call count

and: "a warehouse with fluctuating stock levels"

WarehouseInventory inventory =
    Stub(WarehouseInventory);

inventory.isProductAvailable( "bravia", _ ) >>> true >> false

inventory.isEmpty() >>> [false, true]

basket.setWarehouseInventory(inventory)
inventory.isEmpty() >>> [false, true]

“When the method isEmpty() is called the first time return false. The second time it is called return true”
Basket basket = new Basket()

and: "a fully stocked warehouse"
WarehouseInventory inventory = Stub(WarehouseInventory)
inventory.isProductAvailable(_, _) >> true
basket.setWarehouseInventory(inventory)

and: "a shipping calculator that charges 10 dollars for each product"
ShippingCalculator shippingCalculator = Stub(ShippingCalculator)
    shippingCalculator.findShippingCostFor(_, _) >> { Product
        product, int count -> 10 * count}
    basket.setShippingCalculator(shippingCalculator)
```
shippingCalculator.findShippingCostFor( _, _) >> { Product
  product, int count -> 10 * count}
```

“When the method is called with any two arguments, ignore the first argument, multiply the second with 10 and return the result”
5. Parameterized tests

Common in big enterprise applications
The need for parameterized tests
Understanding parameterized tests

is Valid extension?

- jpg: yes
- png: yes
- tiff: no
- gif: no
def "Valid images are PNG and JPEG files"() {
    given: "an image extension checker"
    ImageNameValidator validator = new ImageNameValidator()

    expect: "that only valid filenames are accepted"
    validator.isValidImageExtension(pictureFile) == validPicture

    where: "sample image names are"
    pictureFile || validPicture
    "scenery.jpg" || true
    "house.jpeg" || true
    "car.png" || true
    "sky.tiff" || false
    "dance_bunny.gif" || false
}

APPROVED
Tabular design

<table>
<thead>
<tr>
<th>pictureFile</th>
<th>validPicture</th>
<th>error</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scenario.jpg is valid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>house.jpg is valid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>car.png is valid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sky.tif is invalid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ new variable +</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ new scenario +</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dance_bunny.gif is invalid</td>
<td></td>
</tr>
</tbody>
</table>
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
        });
    }

    private int fInput;
    private int fExpected;

    public FibonacciTest(int input, int expected) {
        fInput = input;
        fExpected = expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
JUnit limitations 1/2

• The test class must be polluted with fields that represent inputs.
• The test class must be polluted with fields that represent outputs.
• A special constructor is needed for all inputs and outputs.
JUnit limitations 2/2

• Test data comes into a two-dimensional object array (which is converted to a list).
• Test data and test descriptions are in different places
• Cannot easily use two tests in the same class
Alternatives

• TestNG addresses some of these limitations
• https://github.com/TNG/junit-dataprovider
• https://github.com/Pragmatists/junitparams
• https://github.com/piotrturski/zohhak
• Developers avoid using parameterized tests and keep copying-pasting the same code
Business Analysts love tables

<table>
<thead>
<tr>
<th>Current pressure</th>
<th>Fire sensors</th>
<th>Radiation sensors</th>
<th>Audible alarm</th>
<th>A shutdown is needed</th>
<th>Evacuation within x minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>0</td>
<td>0, 0, 0</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
<td>0, 0, 0</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>150</td>
<td>3</td>
<td>0, 0, 0</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>110.4, 0.3, 0.0</td>
<td>Yes</td>
<td>Yes</td>
<td>1 minute</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>45.3, 10.3, 47.7</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>155</td>
<td>0</td>
<td>0, 0, 0</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>170</td>
<td>0</td>
<td>0, 0, 0</td>
<td>Yes</td>
<td>Yes</td>
<td>3 minutes</td>
</tr>
<tr>
<td>180</td>
<td>0</td>
<td>110.4, 0.3, 0.0</td>
<td>Yes</td>
<td>Yes</td>
<td>1 minute</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>110.4, 300, 0.0</td>
<td>Yes</td>
<td>Yes</td>
<td>1 minute</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>110.4, 1000, 0.0</td>
<td>Yes</td>
<td>Yes</td>
<td>1 minute</td>
</tr>
<tr>
<td>155</td>
<td>4</td>
<td>0, 0, 0</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>170</td>
<td>1</td>
<td>45.3, 10.3, 47.7</td>
<td>Yes</td>
<td>Yes</td>
<td>3 minutes</td>
</tr>
</tbody>
</table>
Convert Specs directly into code

where: "possible nuclear incidents are:"

<table>
<thead>
<tr>
<th>pressure</th>
<th>fireSensors</th>
<th>radiation</th>
<th>alarm</th>
<th>shutDown</th>
<th>evacuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>0</td>
<td>[]</td>
<td>false</td>
<td>false</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td>1</td>
<td>[]</td>
<td>true</td>
<td>false</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td>3</td>
<td>[]</td>
<td>true</td>
<td>true</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>[110.4f, 0.3f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
<td>[45.3f, 10.3f, 47.7f]</td>
<td>false</td>
<td>false</td>
<td>-1</td>
</tr>
<tr>
<td>155</td>
<td>0</td>
<td>[0.0f, 0.0f, 0.0f]</td>
<td>true</td>
<td>false</td>
<td>-1</td>
</tr>
<tr>
<td>170</td>
<td>0</td>
<td>[0.0f, 0.0f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>3</td>
</tr>
<tr>
<td>180</td>
<td>0</td>
<td>[110.4f, 0.3f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>[110.4f, 300f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>[110.4f, 1000f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>155</td>
<td>4</td>
<td>[0.0f, 0.0f, 0.0f]</td>
<td>true</td>
<td>true</td>
<td>-1</td>
</tr>
<tr>
<td>170</td>
<td>1</td>
<td>[45.3f, 10.3f, 47.7f]</td>
<td>true</td>
<td>true</td>
<td>3</td>
</tr>
</tbody>
</table>
JUnit and Spock LOC (same test)
6. Extra Enterprise features

Spock is ready for the Enterprise.
public class SampleTest {
    @Test
    void login()

    @Test
    void createOrder()

    @Test
    void viewOrder()
}

Classic scenario
Tests should run in order

If login fails no need to continue
Tests should be isolated

But that is true only for pure unit tests. Functional tests have sometimes different needs.
Spock @Stepwise

Used on class. If a test fails all other methods are ignored
Using Stepwise

@Stepwise
class SpringRestSpec extends Specification {

    def "Simple status checker"() {
        [...code here...]
    }

    def "Cleaning all products"() {
        [...code here...]
    }

    def "Creating a product"() {
        [...code here...]
    }
}

Using Stepwise
JUnit @Ignore

Very simple. On/Off switch to enable/disable tests
@IgnoreIf({ os.windows })

This test will run on Linux/Mac but not Win
This test will not run if this system variable is present
Spock @Ignore

Use any condition that returns a boolean
@IgnoreIf({ new CreditCardProcessor().online() })

This test will not run if a staging server is down
More Spock features

• Mocking/Interaction testing
• Lifecycle methods
• Timeouts
• Data pipes/Data generators
• Exception catching
• Functional tests with Geb
• Documentation annotations
• Spy Objects
• Spock extensions
Summary – Why Spock
Cut your unit test code size by 50%

Groovy itself if very concise and not as verbose as Java
Enforce a clear structure in your tests

Using Spock blocks given, when, then etc.
Make your tests readable by business analysts

Spock allows you to adopt an English like flow in your tests
Embrace (and not fear) parameterized tests

Spock has a DSL for data tables mapping directly program specifications
Use tests as specifications

Spock reports explain fully the test case
Use built-in mocking/stubbing

Spock can mock classes and interfaces (Groovy and Java)
Instant insight on failed builds

Spock gives you the full context when a test fails
Cover unit, integration and functional tests

Spock has explicit facilities for all types of testing
Bring Spock in your Enterprise
The end

www.codepipes.com