Unit tests with Java and Spock

Write readable and maintainable tests

Adrien CAUBEL

February 8, 2022

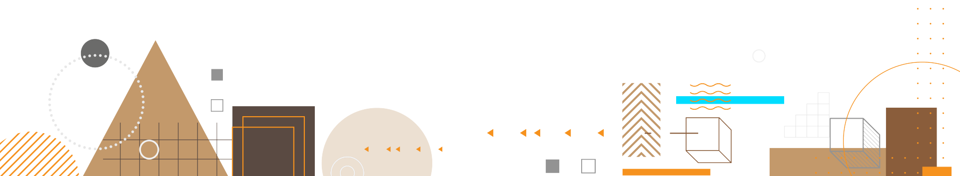


Table of contents

1. Overview and configurations
2. Writing tests with Spock
3. Data Driven Testing with Spock
4. Interaction Based Testing
5. Use third-party libraries

Part I

Overview and configurations

Plan

1. Spock concisely
2. Maven configurations
   - Project configuration
   - JUnit 5 dependency
   - Groovy dependency
   - Spock dependency
   - Full configuration
3. Eclipse Configurations
4. Checkout the configuration
   - Create JUnit test
   - Create Spock test
5. Integration in GitLab CI
   - Example
   - Configuration

Spock concisely

Spock is

- a testing and specification framework
- a beautiful and highly expressive specification language

Spock is compatible with

- JUnit thanks to the JUnit Runner
- most IDEs
- most build tools
- most continuous integration servers

`https://spockframework.org/`

Plan

1. Spock concisely
2. Maven configurations
   - Project configuration
   - JUnit 5 dependency
   - Groovy dependency
   - Spock dependency
   - Full configuration
3. Eclipse Configurations
4. Checkout the configuration
   - Create JUnit test
   - Create Spock test
5. Integration in GitLab CI
   - Example
   - Configuration

Project configuration

```xml
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>3.9.0</version>
         <configuration>
            <source>17</source>
            <target>17</target>
         </configuration>
      </plugin>
   </plugins>
</build>
```

JUnit 5 dependency

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-engine</artifactId>
   <version>5.8.2</version>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-api</artifactId>
   <version>5.8.2</version>
   <scope>test</scope>
</dependency>
```

JUnit 5 dependency

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>3.0.0-M1</version>
         <configuration>
            <includes>
               <include>**/*Test</include>
               <include>**/*Spec</include>
            </includes>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Groovy dependency

```xml
<dependency>
   <groupId>org.codehaus.groovy</groupId>
   <artifactId>groovy</artifactId>
   <version>3.0.9</version>
</dependency>
```

Configuration Groovy

```xml
<build>
   <plugins>
      <plugin>
         <groupId>org.codehaus.gmavenplus</groupId>
         <artifactId>gmavenplus-plugin</artifactId>
         <version>1.13.1</version>
         <executions>
            <execution>
               <goals>
                  <goal>compileTests</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

Spock dependency

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.spockframework</groupId>
         <artifactId>spock-bom</artifactId>
         <version>2.0-groovy-3.0</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

```
<dependency>
   <groupId>org.spockframework</groupId>
   <artifactId>spock-core</artifactId>
</dependency>
```

Full configuration

Based on the documentation

https://gist.github.com/adrien1212/2497ad62af0be75e28dc4dce1e3c1c3d

Documentation

https://github.com/groovy/GMavenPlus/wiki/Examples#spock-2-and-junit

Plan

1. Spock concisely
2. Maven configurations
   - Project configuration
   - JUnit 5 dependency
   - Groovy dependency
   - Spock dependency
   - Full configuration
3. **Eclipse Configurations**
4. Checkout the configuration
   - Create JUnit test
   - Create Spock test
5. Integration in GitLab CI
   - Example
   - Configuration

Eclipse Configurations

Disclaimer

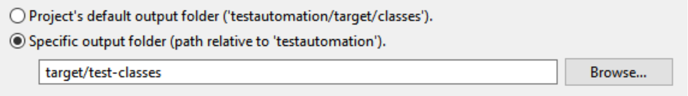
This configuration is done in January 2022 the Eclipse version 12-2022

1. Install *Groovy Development Tools* to
   - Compile Groovy script before run it in test
     `MySpec.groovy` → `Right Click` → `Run As` → `Junit Test`
   - Else, you need to compile your Groovy script each time with Maven
2. Configure Groovy compiler
   - Go to `Windows` → `Preferences` → `Groovy : Compiler`
   - Switch to 3.0.9 compiler

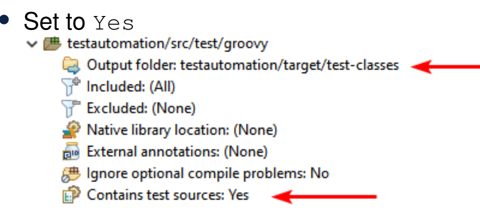
Eclipse Configurations

Configure the output folder for the Groovy tests like Java tests

- Output folder
  - `Build Path` → `Configure Build Path` → `Source` → `monprojet/src/test/groovy`
  - Setup the output folder



- Contains test sources
  - Set to `Yes`



Plan

1. Spock concisely
2. Maven configurations
   - Project configuration
   - JUnit 5 dependency
   - Groovy dependency
   - Spock dependency
   - Full configuration
3. Eclipse Configurations
4. Checkout the configuration
   - Create JUnit test
   - Create Spock test
5. Integration in GitLab CI
   - Example
   - Configuration

Create JUnit test

Create a new test classe

```
MyProject
 ├─ src/main/java
 ├─ src/test/java
 │   └─ FirstTest.java
 └─ pom.xml
```

```
import
    org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class FirstTest {
   @Test
   public void firstAssert() {
      Assertions
         .assertEquals(10, 10);
   }
}
```

Launch the test with Maven

- Run the following command `mvn clean install test`
- and check the result

```
[INFO] ------------------------------------------
[INFO]  T E S T S
[INFO] ------------------------------------------
[INFO] Running FirstTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, [...]
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Create Spock test

To create a test with Spock we need to :

- Create a new directory `/src/test/groovy`
- Create a Groovy script `SecondSpec.groovy`

Create a new test classe

```
MonProjet
 ├── src/main/java
 ├── src/test/java
 │    └── FirstTest.java
 ├── src/main/groovy
 │    └── SecondSpec.groovy
 └── pom.xml
```

```
import spock.lang.Specification

class SecondSpec extends
    Specification {
   def "one plus one equal two"() {
     expect:
     1 + 1 == 2
   }
}
```

Launch the test with Maven

- Run the following command `mvn clean install test`
- and check the results (JUnit + Spock)

```
[INFO] -------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------
[INFO] Running FirstTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, [...]
[INFO] Running SecondSpec
[INFO] Tests run: 1, Failures: 0, Errors: 0, [...]
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Launch the test with Eclipse

This step ensures you that you will be able to run Groovy test without recompiling the Maven project after each modification.

- Run the `SecondSpec` test with Eclipse
  - `SecondSpec.groovy` → `Right Click` → `Run As` → `Junit Test`
  - Check that the test succeeds
- Set the expectation $1 + 1 == 2$ to $1 + 1 == 3$
- Run again the `SecondSpec` test with Eclipse
  - `SecondSpec.groovy` → `Right Click` → `Run As` → `Junit Test`
  - Check that the test fails

SecondSpec [Runner: JUnit 5] (0,267 s)
one plus one equals two (0,267 s)

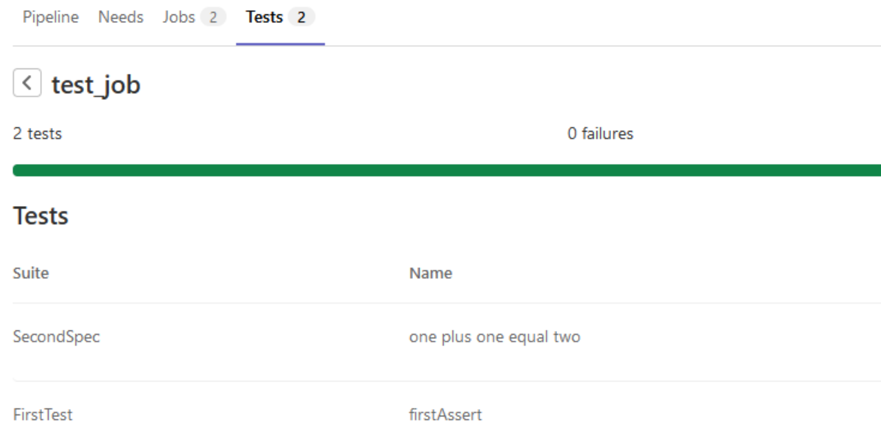
SecondSpec [Runner: JUnit 5] (0,499 s)
one plus one equals two (0,499 s)

Plan

1. Spock concisely
2. Maven configurations
   - Project configuration
   - JUnit 5 dependency
   - Groovy dependency
   - Spock dependency
   - Full configuration
3. Eclipse Configurations
4. Checkout the configuration
   - Create JUnit test
   - Create Spock test
5. Integration in GitLab CI
   - Example
   - Configuration

Example

Spock tests, like all JUnit tests, are managed by the CI/CD provided by GitLab.



Configuration

This integration does not require any addition to the default job
https://docs.gitlab.com/ee/ci/unit_test_reports.html#maven

```
java:
  stage: test
  script:
    - mvn verify
  artifacts:
    when: always
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml
        - target/failsafe-reports/TEST-*.xml
```

Part II

Writing tests with Spock

Plan

6 Introduction

7 Fields
- Fields and Shared Fields

8 Fixture Methods

9 Feature Methods
- Conceptual phases
- Blocks
  - Blocks' advantages
  - When and Then Blocks
  - When-Then block VS Expect block
  - Cleanup block
  - Where block
- Comparison of JUnit test and Spock test

10 Helper methods

11 Assert multiple expectations together

Documentation

`https://spockframework.org/spock/docs/2.0/index.html`

This part summarizes the section *Spock Primer* of the documentation

Plan

6 Introduction

7 Fields
- Fields and Shared Fields

8 Fixture Methods

9 Feature Methods
- Conceptual phases
- Blocks
  - Blocks' advantages
  - When and Then Blocks
  - When-Then block VS Expect block
  - Cleanup block
  - Where block
- Comparison of JUnit test and Spock test

10 Helper methods

11 Assert multiple expectations together

Fields and Shared Fields

- Objects stored into instance fields are not shared between feature methods
- To share an object between feature methods declare a `@Shared` field

```
def number = 5;

def "first"() {
    when: number = 6
    then: number == 6
}

def "second"() {
    expect: number == 5
}
```

```
@Shared number = 5;

def "first"() {
    when: number = 6
    then: number == 6
}

def "second"() {
    expect: number == 6
}
```

Plan

6 Introduction

7 Fields
- Fields and Shared Fields

8 **Fixture Methods**

9 Feature Methods
- Conceptual phases
- Blocks
  - Blocks' advantages
  - When and Then Blocks
  - When-Then block VS Expect block
  - Cleanup block
  - Where block
- Comparison of JUnit test and Spock test

10 Helper methods

11 Assert multiple expectations together

Fixture Methods

- Responsible for setting up and cleaning up the environment
- All fixture methods are optional
- Equivalent to JUnit

Spock	JUnit 5
setup()	@BeforeEach
cleanup()	@AfterEach
setupSpec()	@BeforeAll
cleanupSpec()	@AfterAll

```
def setupSpec() // runs before the first feature method
def setup()   // runs before every feature method
def cleanup() // runs after every feature method
def cleanupSpec() // runs after the last feature method
```

Fixture Methods

- `setupSpec()` and `cleanupSpec()` cannot reference field annoted with `@Shared`

```
@Shared number

def setupSpec() {
   number = 7
}

def "second"() {
   expect: number == 7
}
```

Plan

6 Introduction

7 Fields
- Fields and Shared Fields

8 Fixture Methods

9 Feature Methods
- Conceptual phases
- Blocks
  - Blocks' advantages
  - When and Then Blocks
  - When-Then block VS Expect block
  - Cleanup block
  - Where block
- Comparison of JUnit test and Spock test

10 Helper methods

11 Assert multiple expectations together

Conceptual phases

Conceptually, a feature method consists of four phases:

1. Set up the feature's fixture - [Given]
2. Provide a *stimulus* to the system under specification - [When]
3. Describe the *response* expected from the system - [Then]
4. Clean up the feature's fixture

```
def "pushing an element on the stack"() {
  // blocks go here
}
```

Name your tests

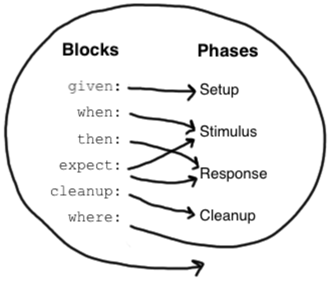
- Choose a good name
- Free to write : string

Blocks

The feature methods are structured in the following blocks

- `given`
- `when`
- `then`
- `expect`
- `cleanup`
- `where`

Each block is mapped to a conceptual phases of a feature method



Blocks’ advantages

- Before you could follow the
  - `Given-When-Then` structure
  - `Arrange-Act-Assert` structure
  - or another homemade structure
- Now, we have a common structure for testing

Advantages

- Tests are structured
- These structure is common to all developers
- Improve the **readability** and **maintainability** of tests
- $\Rightarrow$ Tests have their own "grammar"

When and Then Blocks

- The `when` and `then` blocks always occur together
- A feature method may contain multiple pairs of when-then blocks.

```
when: // stimulus
then: // response
```

```
when:
  stack.push(elem)

then:
  !stack.empty
  stack.size() == 1
  stack.peek() == elem
```

Conditions

- `then` and `expect` receive conditions
- Conditions are written as plain boolean expressions
- When condition is violated Spock provide a feedback

```
Condition not satisfied:

stack.size() == 2
|     |      |
|     1      false
[push me]
```

There are two types of conditions

- *implicit* conditions
  - essential ingredient of `then` blocks and `expect` blocks
  - expressions in these blocks are implicitly treated as conditions
- *explicit* conditions
  - to use conditions in other places
  - use Groovy's `assert` keyword

When-Then block VS Expect block

An `expect` block

- is more limited than a `then` block
- it may only contain conditions and variable definitions
- usefull to describe stimulus and expected response in a single expression

```
when:
def x = Math.max(1, 2)

then:
x == 2
```

```
// Prefered

expect:
Math.max(1, 2) == 2
```

- use `when-then` to describe methods with side effects
- and `expect` to describe purely functional methods.

Cleanup block

- Is used to free any resources used by a feature method
- Is run even if the feature method has produced an exception

```
given:
  def file = new File("/some/path")
  file.createNewFile()
  // ...
cleanup:
  file.delete()
```

- Object-level specifications usually don't need a `cleanup` method
- Automatically reclaimed by the garbage collector
- Might use a `cleanup` block
  - to clean up the file system
  - to close a database connection
  - to shut down a network service

Where block

- Always comes last in a method
- May not be repeated
- Used to write data-driven feature methods

```
def "computing the maximum of two numbers"() {
  expect:
    Math.max(a, b) == c

  where:
    a << [5, 3]
    b << [1, 9]
    c << [5, 9]
}
```

This `where` block creates two "versions" of the feature method

- First with $a = 5$, $b = 1$ and $c = 5$
- Second with $a = 3$, $b = 9$ and $c = 9$

Comparison of JUnit test and Spock test

```
@Test
public void givenTwoAndTwo_whenAdding_thenIsFour() {
  int first = 2, second = 2; // Given
  int result = first + second; // When
  assertTrue(result == 4) // Then
}
```

```
def "two plus two should equal four"() {
   given:
      int left = 2
      int right = 2
   when:
      int result = left + right
   then:
      result == 4
}
```

Plan

6. Introduction
7. Fields
   - Fields and Shared Fields
8. Fixture Methods
9. Feature Methods
   - Conceptual phases
   - Blocks
     - Blocks' advantages
     - When and Then Blocks
     - When-Then block VS Expect block
     - Cleanup block
     - Where block
   - Comparison of JUnit test and Spock test
10. Helper methods
11. Assert multiple expectations together

Helper methods

```
def "offered PC matches preferred configuration"() {
  when:
    def pc = shop.buyPc()

  then:
    pc.vendor == "Sunny"
    pc.clockRate >= 2333
    pc.ram >= 4096
    pc.os == "Linux"
}
```

- Avoid grow large method
- Avoid duplicated code
- $\Rightarrow$ Introduce one or more helper methods to factoring out

Helper methods

Two points need to be considered when creating a helper method

- implicit conditions must be turned into explicit conditions
  $\Rightarrow$ use the `assert` keyword
- must have return type `void`

```
def "offered PC matches preferred configuration"() {
  when:
    def pc = shop.buyPc()
  then:
    matchesPreferredConfiguration(pc)
}

void matchesPreferredConfiguration(pc) {
  assert pc.vendor == "Sunny"
  assert pc.clockRate >= 2333
  assert pc.ram >= 4096
  assert pc.os == "Linux"
}
```

Plan

6 Introduction

7 Fields
- Fields and Shared Fields

8 Fixture Methods

9 Feature Methods
- Conceptual phases
- Blocks
  - Blocks' advantages
  - When and Then Blocks
  - When-Then block VS Expect block
  - Cleanup block
  - Where block
- Comparison of JUnit test and Spock test

10 Helper methods

11 Assert multiple expectations together

Assert multiple expectations together

- Normal expectations fail the test on the first failed assertions
- Sometimes it is helpful to collect these failures before failing the test

```
def "my test"() {
    expect:
    verifyAll {
        2 == 3
        4 == 5
    }
}
```

```
condition not satisfed :
2 == 3
condition not satisted :
4 == 5
```

```
def "my test"() {
    expect:
        2 == 3
        4 == 5
}
```

```
condition not satisfed :
2 == 3
```

Part III

Data Driven Testing with Spock

Plan

12 Introduction
- Definition
- Documentation

13 Data Tables

14 Isolated Execution of Iterations

15 Syntactic Variations
- Double Pipe
- Data Pipes

Definition

Data Driven Testing

- Is a software testing method in which test data is stored in table or spreadsheet format
- Is useful because we provide multiple data sets for a single test and an individual test is created with each data

Documentation

Documentation

`https://spockframework.org/spock/docs/2.0/index.html`

This part summarizes the section *Data Driven Testing* of the documentation

Plan

12 Introduction
- Definition
- Documentation

13 Data Tables

14 Isolated Execution of Iterations

15 Syntactic Variations
- Double Pipe
- Data Pipes

Data Tables

```
class MathSpec extends Specification {
  def "maximum of two numbers"(int a, int b, int c) {
    expect:
     Math.max(a, b) == c

    where:
     a | b | c
     1 | 3 | 3
     7 | 4 | 7
     0 | 0 | 0
  }
}
```

- The first line is the *table header*, declares the data variables
- The subsequent lines are *table rows*, the corresponding values
- For each row, an *iteration* of feature method will get executed

Plan

12 Introduction
- Definition
- Documentation

13 Data Tables

14 Isolated Execution of Iterations

15 Syntactic Variations
- Double Pipe
- Data Pipes

Isolated Execution of Iterations

Iterations are isolated from each other

- Each iteration gets its own instance of the specification class
- The `setup` and `cleanup` methods will be called before and after each iteration

Plan

12 Introduction
- Definition
- Documentation

13 Data Tables

14 Isolated Execution of Iterations

15 Syntactic Variations
- Double Pipe
- Data Pipes

Double Pipe

```
class MathSpec extends Specification {
  def "maximum of two numbers"() {
    expect:
      Math.max(a, b) == c

    where:
      a | b || c
      1 | 3 || 3
      7 | 4 || 7
      0 | 0 || 0
  }
}
```

- Method parameters can be ommited
  - You can also omit some parameters and specify others, for example to have them typed
- Inputs and expected outputs can be separated with a double pipe symbol

Data Pipes

```
...
where:
  a << [1, 7, 0]
  b << [3, 4, 0]
  c << [3, 7, 0]
```

- Connects a data variable to a data provider
  - the data provider holds all values for the variable, one per iteration
- Any object that Groovy knows how to iterate over can be used as a data provider
  - `Collection`
  - `String`
  - `Iterable`
  - objects implementing the `Iterable` contract

Part IV

Interaction Based Testing

Plan

16 Introduction
- Definition
- Documentation
- When we need Mocking
- Mock implementation with Spock

17 Mocking
- Definition
- The application
- Creating Mock Objects
- Injecting Mock Objects into Code Under Specification
- Creating the test

18 Stubbing
- Definition
- Returning Fixed Values
- Returning Sequences of Values
- Computing Return Values
- Chaining Method Responses
- Returning a default response

Definition

Interaction Based Testing

- Checks how different objects interact with each other
- Help verify the functionality of code that depends on the interaction between multiple classes or interfaces
- The interaction testing uses a *mock object* to check that the expected behavior happened

Documentation

Documentation

`https://spockframework.org/spock/docs/2.0/index.html`

This part summarizes the section *Interaction Based Testing* of the documentation

When we need Mocking

Mock object is useful when you

- want to **test interactions** between a class under test and a particular interface.
- the execution of a method passes outside of that method, into another object : dependencies
- have complicated object as a parameter, and it would be a pain to simply instantiate this object

See also :
`https://odetocode.com/blogs/scott/archive/2008/05/01/mocks-its-a-question-of-when.aspx`

Mock implementation with Spock

- Java world provides of popular and mature mocking frameworks
  - Mockito
  - EasyMock
  - ...
- These frameworks can be used together with Spock
- But Spock integrates its own mocking framework
  - all features of Spock's mocking framework work both for testing Java and Groovy code

Plan

16 Introduction
- Definition
- Documentation
- When we need Mocking
- Mock implementation with Spock

17 Mocking
- Definition
- The application
- Creating Mock Objects
- Injecting Mock Objects into Code Under Specification
- Creating the test

18 Stubbing
- Definition
- Returning Fixed Values
- Returning Sequences of Values
- Computing Return Values
- Chaining Method Responses
- Returning a default response

Definition

Definition

- Mock objects are simulated objects that mimic the behavior of real objects

The application

Publisher

- subscribers : List<Subscriber>
- messageCount : int

+ send(message : String) void

<<interface>>
Subscriber

+ receive(message : String)

```
void send(String message){
   for(Subscriber s : subscribers) {
      s.receive(message);
   }
   messageCount++
}
```

Creating Mock Objects

- We create two fake implementations of `Subscriber`

```
def subscriber = Mock(Subscriber)
def subscriber2 = Mock(Subscriber)
```

Or

```
Subscriber subscriber = Mock()
Subscriber subscriber2 = Mock()
```

Injecting Mock Objects into Code Under Specification

- We set the `Publisher` with the fake `Subscriber`

```
class PublisherSpec extends Specification {
 Publisher publisher = new Publisher()
 Subscriber subscriber = Mock()
 Subscriber subscriber2 = Mock()

 def setup() {
  publisher.subscribers << subscriber // <=> List.add()
  publisher.subscribers << subscriber2
 }
```

Creating the test

```
def "should send messages to all subscribers"() {
  when:
    publisher.send("hello")

  then:
    1 * subscriber.receive("hello")
    1 * subscriber2.receive("hello")
}
```

- **When** the publisher sends a ’hello’ message
- **Then** both subscribers should receive that message exactly once

Plan

16 Introduction
- Definition
- Documentation
- When we need Mocking
- Mock implementation with Spock

17 Mocking
- Definition
- The application
- Creating Mock Objects
- Injecting Mock Objects into Code Under Specification
- Creating the test

18 Stubbing
- Definition
- Returning Fixed Values
- Returning Sequences of Values
- Computing Return Values
- Chaining Method Responses
- Returning a default response

Stubbing

- Is the act of making collaborators respond to method calls in a certain way
- We don't care if and how many times the method is going to be called
  - we just want it to return some value
  - or perform some side effect

Example

Whenever the subscriber receives a message, make it respond with 'ok'

Returning Fixed Values

- Whenever the subscriber receives a message, make it respond with 'ok'

```
subscriber.receive(_) >> "ok"
```

- Return different values for different invocations
    - *ok* whenever *message1* is received
    - *fail* whenever *message2* is received

```
subscriber.receive("message1") >> "ok"
subscriber.receive("message2") >> "fail"
```

Returning Sequences of Values

- Use triple-right-shift >>>
- Return different values on successive invocations
  - *ok* for the first invocation
  - *error* for the second invocation
  - *error* for the third invocation
  - *ok* for all remaining invocations

```
subscriber.receive(_) >>> ["ok", "error", "error", "ok"]
```

Computing Return Values

- Return value based on the method's argument
  - *ok* if the message is more than three characters
  - *fail* otherwise

```
subscriber.receive(_) >>
{ args -> args[0].size() > 3 ? "ok" : "fail" }
```

Or

```
subscriber.receive(_) >>
{ String message -> message.size() > 3 ? "ok" : "fail" }
```

- Method arguments will be mapped one-by-one to closure parameters
- Behaves the same as the previous one, but is arguably more readable

Chaining Method Responses

- Method responses can be chained
  - *ok* for the first invocation
  - *fail* for the second invocation
  - *ok* for the third invocation
  - throw `InternalError` for the fourth invocation
  - *ok* for any further invocation

```
subscriber.receive(_)
>>> ["ok", "fail", "ok"]
>> { throw new InternalError() }
>> "ok"
```

Returning a default response

- Don't really care what you return
- But you must return a non-null value
- Use _

```
subscriber.receive(_) >> _
```

- This will use the same logic to compute a response as `Stub`

Part V

Use third-party libraries

Plan

19 Introduction

20 Selenium
- Include Maven dependency
- Preparing our tests
- Writing out tests

Introduction

Use framework

- As with Junit, we can use third-party framework
- Spock accept their uses
- By adding the Maven dependency

Plan

19 Introduction

20 Selenium
- Include Maven dependency
- Preparing our tests
- Writing out tests

Include Maven dependency

```
<dependency>
   <groupId>org.seleniumhq.selenium</groupId>
   <artifactId>selenium-java</artifactId>
   <version>4.1.1</version>
</dependency>
```

Preparing our tests

```
class SeleniumSpec extends Specification {
   @Shared WebDriver driver

   def setupSpec() {
      System.setProperty("webdriver.gecko.driver",
                         "D:/.../geckodriver.exe");
      driver = new FirefoxDriver()
   }
   def cleanupSpec() {
      if(driver != null) { driver.close() }
   }

   /* Methodes de test */
}
```

- `driver` is a shared field
- `setupSpec` is run once before the first test
- `cleanupSpec` is run once after the last test

Writing out tests

```
def "selenium integration with spock"() {
    when:
     driver.get "https://spockframework.org/"

    then:
     driver.title == "Spock"
}

def "selenium integration with spock bis"() {
    when:
     driver.get "https://selenium.dev"

    then:
     driver.title == "Selenium"
}
```