Spock 101

Testing with Groovy

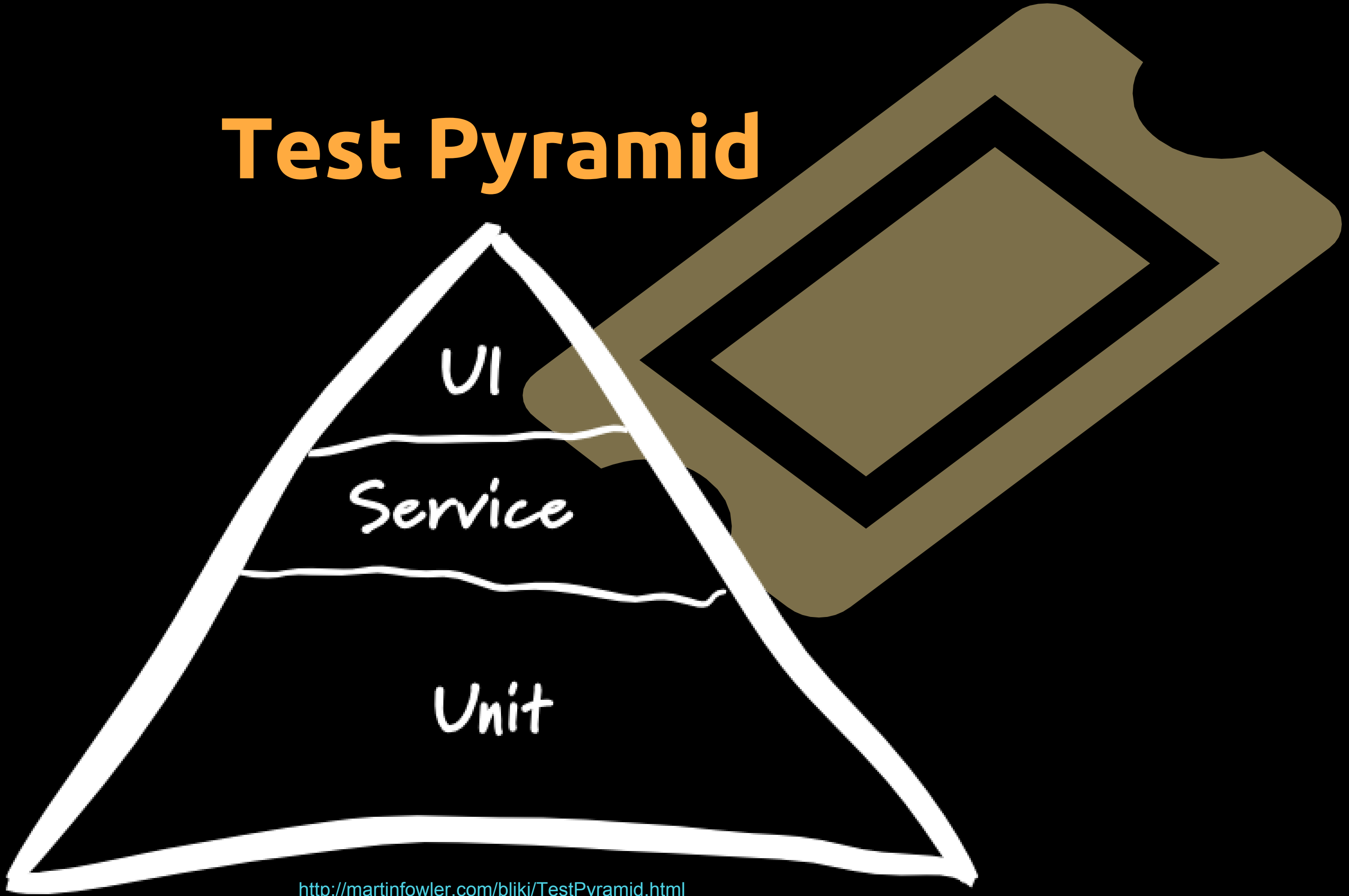
Testing Benefits

- Software reliability
- Good software design
- Confidence
- Safe refactoring
- Bugs reduction
- Documentation

Why testing is a good idea?

- Are quick to run
- Explain behaviour
- Verify functionality
- Identify bugs
- Test interaction between components

Test Pyramid



http://martinfowler.com/bliki/TestPyramid.html

F.I.R.S.T properties of Unit Testing

- **Fast**
  - Many hundred or thousands per second.
- **Isolates**
  - Failure reasons become obvious.
- **Repeatable**
  - Run repeatedly in any order, any time.
- **Self-validating**
  - No manual validation required
- **Timely**
  - Written before the code

TDD

Test-Driven Development is a programming discipline whereby programmers drive the design and implementation of their code by using unit tests.

1. You can't write any production code until you have first written a failing unit test.
2. You can't write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You can't write more production code than is sufficient to pass the currently failing unit test.

Spock Framework

https://spockframework.github.io/spock/docs

https://github.com/spockframework/spock

Spock Test

```groovy
import spock.lang.Specification

class TriangleSpec extends Specification {

    def exercises = new Exercises()

    def "Calculate triangle area using the given base and height"() {
        given: "the base and height"
        def base = 3
        def height = 2

        when: "calculate triangle area"
        def area = exercises.calculateTriangleArea(base, height)

        then: "area must be the expected one"
        area == 3
    }
}
```

Fixture Methods

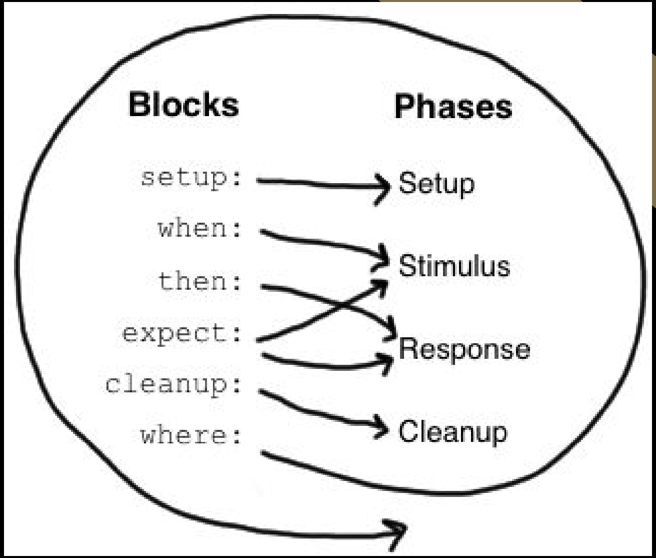
```
def setup() {}            // run before every feature method
def cleanup() {}          // run after every feature method
def setupSpec() {}        // run before the first feature method
def cleanupSpec() {}      // run after the last feature method
```

Feature Methods

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

Blocks

Spock has built-in support for implementing each of the conceptual phases of a feature method.



Data Driven Tests (I)

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

       where:
       a | b || c
       3 | 5 || 5
       7 | 0 || 7
       0 | 0 || 0
   }
}
```

It is executed before the feature method.

Runs as only one test

Data Driven Tests (II)

```groovy
import spock.lang.Unroll

class DataDrivenSpec extends Specification {
   @Unroll
   def "maximum of two numbers (max(#a, #b) == #c)"() {
       expect:
       Math.max(a, b) == c

       where:
       a | b || c
       3 | 5 || 5
       7 | 0 || 0
       0 | 0 || 0
   }
}
```

- maximum of two numbers (max(3, 5) == 5)
- maximum of two numbers (max(7, 0) == 0)

```
Math.max(a, b) == c
     |    |  |  |  |
     7    7  0  |  0
                false
```

- maximum of two numbers (max(0, 0) == 0)

Runs as three different tests

Data Driven Tests (III)

Data Pipes

```
where:
a << [3, 7, 0]
b << [5, 0, 0]
c << [5, 7, 0]
```

Data Variable Assignment

```
where:
a = 3
b = Math.random() * 100
c = a > b ? a : b
```

Multi-Variable Data Pipes

```
where:
[a, b, c] << sql.rows("select a, b, c from maxdata")
```

Testing Exceptions

```
when:
stack.pop()

then:
thrown(EmptyStackException)
notThrown(IllegalAccessException)
```

```
when:
stack.pop()

then:
def e = thrown(EmptyStackException)
// EmptyStackException e = thrown()
e.cause == null
```

Mocks (I)

- Test interactions with collaborators
- Mock objects have no behaviour
- They only return default value for the method’s return type (false, 0, or null)
- They work with Java code
- Mock objects literally implement (or, in the case of a class, extend) the type they stand in for

Mocks (II)

```groovy
class PublisherSpec extends Specification {

   Publisher publisher = new Publisher()
   Subscriber subscriber = Mock()
   def subscriber2 = Mock(Subscriber)

   def setup() {
       publisher.subscribers << subscriber
       publisher.subscribers << subscriber2
   }

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

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

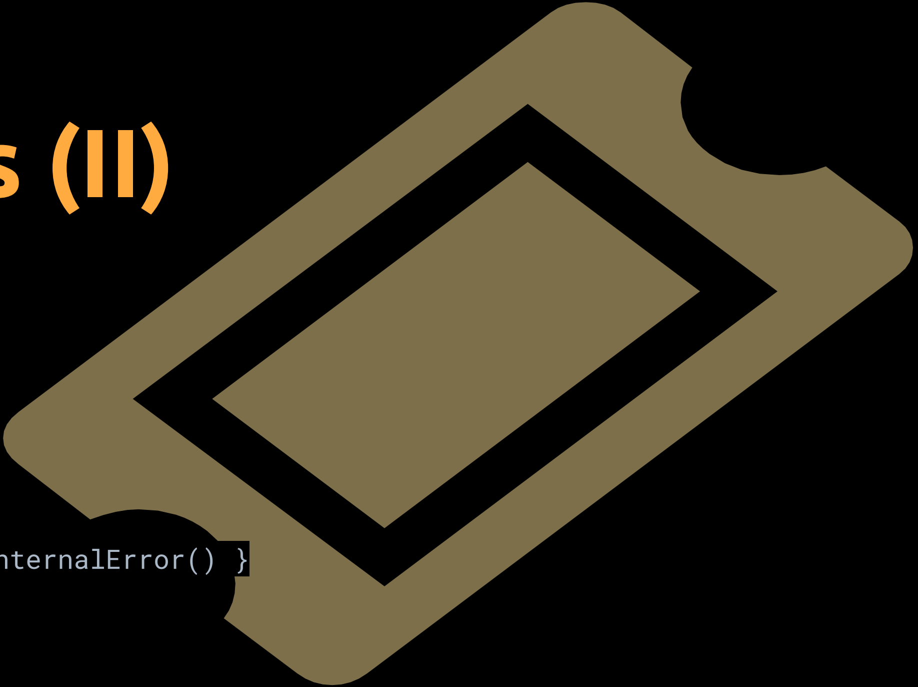
Mocks (III)

```
1 * subscriber.receive("hello")           // exactly one call
0 * subscriber.receive("hello")           // zero calls
(1..3) * subscriber.receive("hello")      // between one and three calls (inclusive)
(1.._) * subscriber.receive("hello")      // at least one call
(_..3) * subscriber.receive("hello")      // at most three calls
_ * subscriber.receive("hello")           // any number of calls, including zero

1 * _.receive("hello")                    // a call to any mock object
1 * subscriber./r.*e/("hello")            // a method whose name matches the given regular expression
1 * subscriber.status                     // same as: 1 * subscriber.getStatus()

1 * subscriber.receive("hello")           // an argument that is equal to the String "hello"
1 * subscriber.receive(!"hello")          // an argument that is unequal to the String "hello"
1 * subscriber.receive()                  // the empty argument list (would never match in our example)
1 * subscriber.receive(_)                 // any single argument (including null)
1 * subscriber.receive(*_)                // any argument list (including the empty argument list)
1 * subscriber.receive(!null)             // any non-null argument
1 * subscriber.receive(_ as String)       // any non-null argument that is-a String
1 * subscriber.receive({ it.size() > 3 }) // an argument that satisfies the given predicate

1 * subscriber._(*_)                      // any method on subscriber, with any argument list
1 * subscriber._                          // shortcut for and preferred over the above

1 * _._                                   // any method call on any mock object
1 * _                                     // shortcut for and preferred over the above
```

Stubs (I)

- Make collaborators respond to methods in a certain way
- Return fixed values
- Perform some side effect
- They don’t care about interactions
- Mock can be used for stubbing
- Stub cannot be used for mocking

Stubs (II)



```
class PublisherSpec extends Specification {

        Publisher publisher = new Publisher()
        Subscriber subscriber = Stub()
        def subscriber2

        def setup() {
            subscriber.receive("message1") >> "OK"
            subscriber2 = Stub(Subscriber) {
                receive("message1") >> { throw new InternalError() }
            }

            publisher.subscribers << subscriber
            publisher.subscribers << subscriber2
        }

        def "should send messages to all subscribers"() {
            when:
            publisher.send("message1")

            then:
            thrown InternalError
        }
}
```

Stubs (III)

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

```
def subscriber = Stub(Subscriber) {
   receive("message1") >> "ok"
   receive("message2") >> "fail"
}
```

```
subscriber.receive(_) >>> ["ok", "error", "error", "ok"]
subscriber.receive(_) >> { String message -> message.size() > 3 ? "ok" : "fail" }
subscriber.receive(_) >> { throw new InternalError("ouch") }
subscriber.receive(_) >>> ["ok", "fail", "ok"] >> { throw new InternalError() } >> "ok"
```

Let’s start working!

https://github.com/ticketbis/spock-workshop

Thank You!

We are hiring!

...and we are remote friendly!

Aritz Águila
@duiraritz

Álvaro Salazar
@xala3pa

Endika Santamaría
@katxorro87

Imanol Pinto
@Pahint

itjobs@ticketbis.com

@TicketbisEng

TICKETBIS ENGINEERING