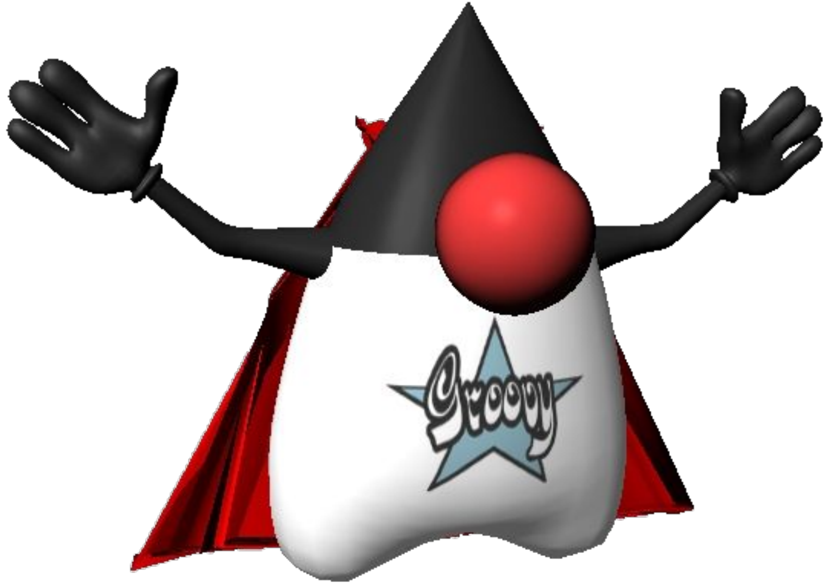
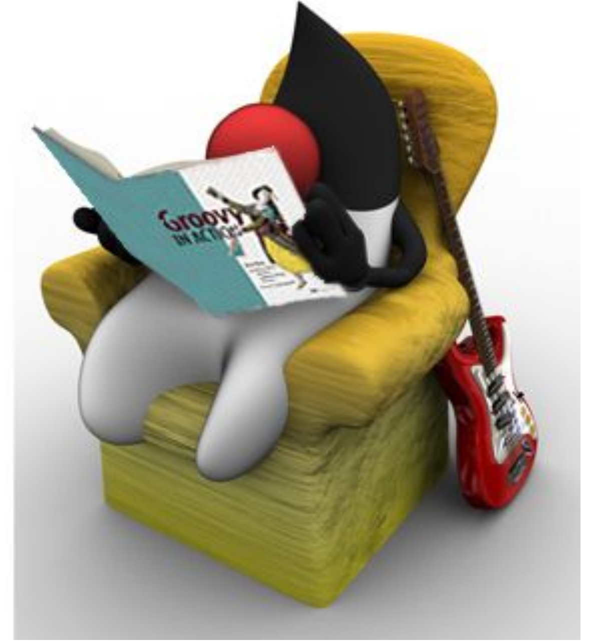
Groovy & Spock





Grooovy

- диалект Java (в отличие от Scala), почти любой код на Java является валидным
- динамический язык с котролем типов во время исполнения (int c = “” //cast error)
- широкий набор импортов по умолчанию
- primitives - по факту почти всегда используются объекты-обёртки
- def - синоним типа Object
- методы всегда возвращают значение - void = null
- коллекции по умолчанию сохраняют порядок инициализации - в Java коллекции могут выдавать совершенно разный порядок при разных запусках
  `int[] array = [1,2,3]` VS `int[] array = {1, 2, 3}`
- Нет try-with-resources, зато есть @AutoCleanup в Spock
- Closure - это Lambda, которая умеет менять внешние переменные
- == - это compareTo (для сравниваемых объектов) или equals иначе
  таким образом, == может быть несимметричным, например, для GString vs String
  для сравнивания ссылок используйте метод **is**: `128.is(128)`
- позволяет перегружать операторы
- необязательные: `;` в конце, **return**, **скобки** при вызове функции, **public** классы и методы
- реализует множественное наследование с помощью **trait** (аналог интерфейса в Java)
- **switch** оператор позволяет использовать почти любые условия для сравнения (isCase метод), например, тип объекта, сравнение по equals, вхождение в коллекцию, удовлетворение регулярному выражению и даже просто Closure)
- необязательная декларация для checked exceptions

Мульти-методы

В Java перегруженные методы вызываются в зависимости от статической информации на этапе компиляции. В Groovy метод находится в процессе исполнения.

```
int method(String arg) {
   return 1;
}
int method(Object arg) {
   return 2;
}

Object o = "Object";
assert 1 == method(o) //In Java 2


assert 2 == method((Object)o)
```

Свойства (properties)

Если модификатор доступа не указан для поля, это значит, что это не поле, а свойство, у которого автоматически появляются методы доступа и изменения.

```
class Person {
    String name
}
```

===>

```
class Person {
    private String name
    public String getName() {name}
    public void setName(String name) {
        this.name = name
    }
}
```

Если очень хочется получить package-private поле, это возможно

```
class Person {
    @PackageScope String name
}
```

Строки

В Groovy строки заключаются в апострофы.
Поэтому **char** там надо приводить явно.

String name	String syntax	Interpolated	Multiline	Escape character
Single quoted	`'...'`			`\`
Triple single quoted	`'''...'''`		+	`\`
Double quoted	`"..."`	+		`\`
Triple double quoted	`"""..."""`	+	+	`\`
Slashy	`/.../`	+	+	`\`
Dollar slashy	`$/.../$`	+	+	`$`

Строки могут представлять имя метода:

```
def prop = 'a'
def meth = 'size'
def map = [a: [1,2]]
assert
map."$prop"."$meth"() == 2
```

Числа

```
int i
m(i)

void m(long l) {
  println "in m(long)" //Java
}

void m(Integer i) {
  println "in m(Integer)" //Groovy
}
```

```
println 1.abs() // 1
-1.abs() //-1
println (-1).abs() //NPE
println ((-1).abs()) //1
```

```
5/3; //1 in Java
5/3 //1.67 in Groovy
5.intdiv(3) //1 in Groovy, quicklier than {int i = 5/3}

5**1.7 //15.43
```

```
assert 2.5.toInteger()  ==  2
assert 2.5 as Integer   ==  2
assert (int)2.5         ==  2

assert '5'.toInteger()  ==  5
assert '5' as Integer   ==  5
assert (int)'5'         == 53
```

Коллекции

List

```
def numbers = [1,2,3,4,5,6,7]
assert numbers instanceof List
assert numbers.size() == 7
assert numbers[0,2,4..6] ==
                       [1,3,5,6,7]
```

Range

```
def range = 0..5
assert (0..5).collect() == [0, 1, 2, 3, 4, 5]
assert (0..<5).collect() == [0, 1, 2, 3, 4]
assert (0..5) instanceof List
assert (0..5).size() == 6
```

Map

```
def numbers = [1: 'one', 2: 'two']
assert numbers[1] == 'one'

def key = 'name'
person = [(key): 'Guillaume']
assert person.containsKey('name')

def map=[:]
map.get("a", []) << 5
assert map == [a:[5]]
```

Object creation

```
class Foo {
    def a, b
}

def foo = new Foo(a: '1', b: '2')
assert foo.a == '1'

interface X {
    void f()
    void g(int n)
}
x = [ f: {println "f called"} ] as X
```

Операторы для работы с коллекциями

Spread collections

```
def items = [4,5]
def list = [1,2,3,*items,6]
assert list == [1,2,3,4,5,6]

def m1 = [c:3, d:4]
def map = [a:1, b:2, *:m1]
assert map == [a:1, b:2, c:3, d:4]
```

Subscript

```
def list = [0,1,2,3,4]
assert list[2] == 2
list[2] = 4
assert list[0..2] == [0,1,4]
list[0..1] = [6,6,6]
assert list == [6,6,6,4,3,4]
assert list[-1..0] == list.reverse()
```

Spread (null-safe)

```
cars = [
   new Car(make: 'Peugeot', model: '508'),
   null,
   new Car(make: 'Renault', model: 'Clio')]
assert cars*.make == ['Peugeot', null, 'Renault']
assert null*.make == null
```

Spread arguments

```
int function(int x, int y, int z) {x*y+z}
def args = [4,5,6]
assert function(*args) == 26

args = [4]
assert function(*args,5,6) == 26
```

Groovy Truth

- Non-zero numbers
- Non-empty strings
- Non-empty maps
- Non-empty collections
- Non-empty arrays
- Non-empty iterators
- Non-empty enumerators
- Matcher has at least one match
- Boolean is true
- Non-null objects
- asBoolean()

`String.toBoolean()` Converts the given string into a Boolean object. If the trimmed string is "true", "y" or "1" (ignoring case) then the result is true otherwise it is false.

Регулярные выражения

```
def p = ~/foo/
assert p instanceof Pattern
```

```
def text = "some text to match"
def m = text =~ /match/
assert m instanceof Matcher
if (!m) { //m.find()
    throw new RuntimeException("Text not found!")
}
```

```
m = text ==~ /match/
assert m instanceof Boolean
if (m) {  //strict match
    throw new RuntimeException("Should not reach it!")
}
```

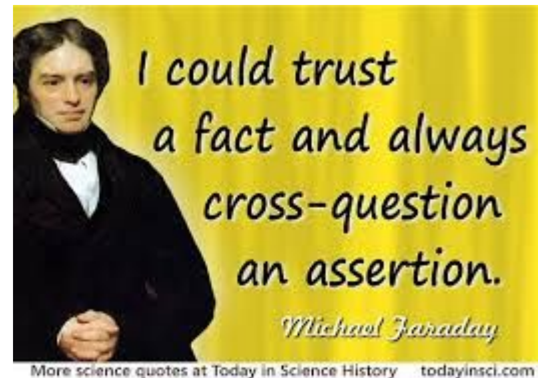
Еще немного операторов

- <=> spaceship compareTo()
- Элвис унарный `def s = k?.toString()`
- Элвис бинарный `def s = k?:"empty"`
- multiple assignment `def (a, b, c) = [1, 2]`
- membership `assert 'Emmy' in ['Grace','Rob','Emmy']`
- coercion `String s = 123 as String`
- diamond `List<String> strings = new LinkedList<>()`
- call

```
class MyCallable {
    int call(int x) { 2*x }
}
def mc = new MyCallable()
assert mc.call(2) == 4
assert mc(2) == 4
```

Power Assert

В Java, `assert` может быть разрешён через параметр JVM `-ea` или запрещён параметром `-da`. По умолчанию ассёрты в Java отключены.



В Groovy assert разрешён всегда и нет возможности его отключить.

Power assert портирован на JavaScript, Perl, .Net, etc.

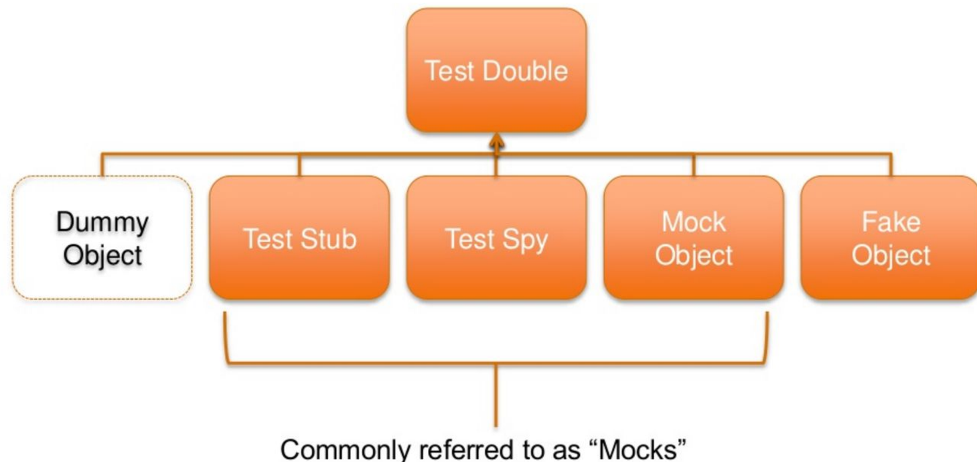
```
def x = 25
assert x + 5 == 31

// Output:
//
// Assertion failed:
// assert x + 5 == 31
//        | |    |
//        | 30   false
//        25
```

Почему тестирование важно

Первоначально тесты на groovy наследовались от GroovyTestCase, имели проверку на ожидаемое исключение shouldFail(exception, Closure), а также Mock & Stub возможности.

Stub: заменяет метод кодом, который возвращает заданный результат (тестирование состояния)

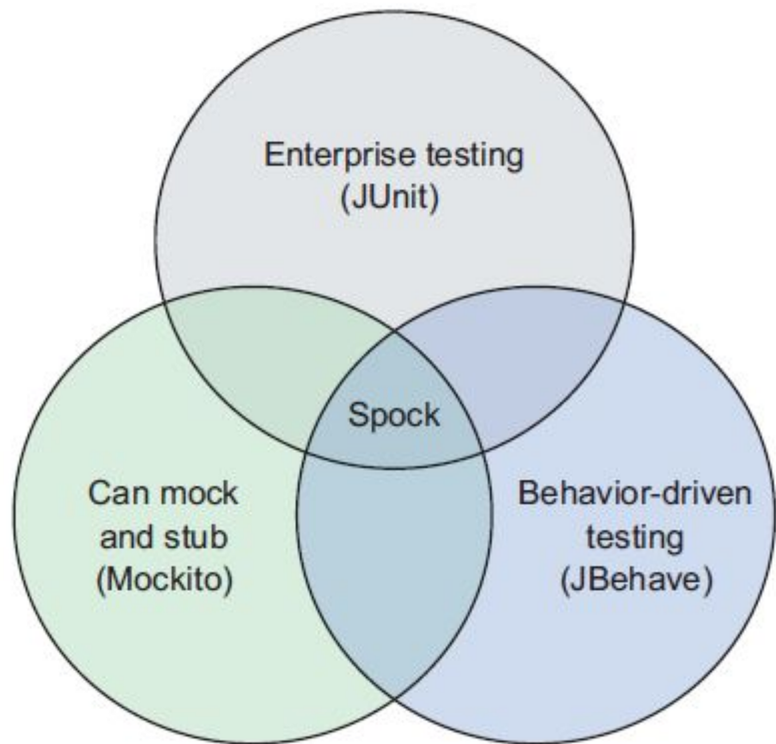
Mock: stub вместе с проверкой условия, что этот stub был вызван (тестирование поведения)



Spock создан в 2008 в Gradleware.

Martin Fowler
21 August 2013

Given-When-Then is a style of representing tests - or as its advocates would say - specifying a system's behavior using SpecificationByExample. It's an approach developed by Dan North and Chris Matts as part of Behavior-Driven Development (BDD). It appears as a structuring approach for many testing frameworks such as Cucumber.



Из чего состоят Spock тесты

Класс с тестами - Specification
Тестовый метод - Feature (позволяет указывать имя на английском языке)
Тестируемый объект - @Subject
Описание спецификации - @Title/@Narrative

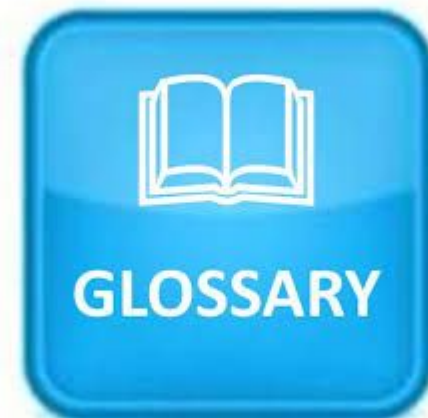


Table 4.1 Available Spock blocks

Spock block	Description	Expected usage
`given:`	Creates initial conditions	85%
`setup:`	An alternative name for `given:`	0% (I use `given:`)
`when:`	Triggers the action that will be tested	99%
`then:`	Examines results of test	99%
`and:`	Cleaner expression of other blocks	60%
`expect:`	Simpler version of `then:`	20%
`where:`	Parameterized tests	40%
`cleanup:`	Releases resources	5%

```groovy
class GivenWhenThenSpec extends Specification {
    def "test adding a new item to a set"() {
        given: "four items set"
        def items = [4, 6, 3, 2] as Set

        when: "add an item to the set"
        items << 1

        then: "set size is five"
        items.size() == 5
    }
}
```

Если вам трудно написать описание блока, это может значить, что ваш тест делает сложные вещи

этот блок должен быть как можно проще, он описывает тестируемое действие

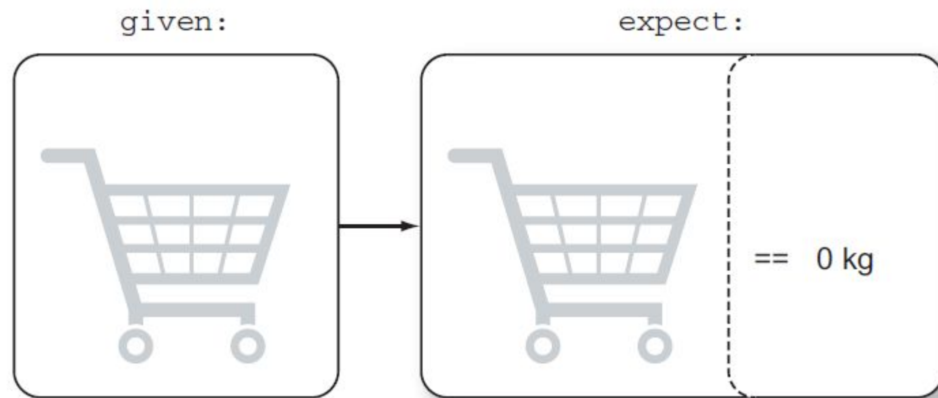
Золотое правило unit тестов: они должны проверять только одну вещь

Всегда включайте в ваши тесты описание блоков и создавайте тестовые методы с именем, которое легко читается.

Тесты должны быть короткими и понятными. Иногда для лучшего понимания стоит использовать методы-хелперы для создания дублёров и для проверки состояния.

Запомните, что множественные вызовы с одним объектом можно группировать с помощью Groovy-with: obj.with { actions }, а множественные проверки можно выполнять с помощью Spock-with: with(obj) { assertions }. Последний with может быть перенесён в метод-хелпер, осуществляющий общие проверки для более одного теста.

expect блок обычно заменяет пару блоков when/then




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

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

expect: block performs the assertion of the test

- where-блок должен быть последним блоком (возможен and: блок)
- возможно явно определить типы параметров, указав их в качестве аргументов тестового метода
- таблица данных должна содержать 2 или более колонок
- @Unroll позволяет построить более детальный отчет, но не меняет логики выполнения теста

```
@Unroll def 'checkPassword(#password) valid=#valid : #comment'() {
   given:
   PasswordValidator validator = new PasswordValidator()

   expect:
   validator.validate(password) == valid

   where:
   password                | valid | comment
   'pwd'                   | false | 'too short'
   'very long password'    | false | 'too long'
   'password'              | false | 'not enough strength'
   'h!Z7abcd'              | true  | 'valid password'
}
```

Какие классы стоит замещать в процессе тестирования

Как правило, вы должны замещать все зависимые классы, которые удовлетворяют условиям:

- делают юнит тесты непредсказуемыми
- имеют сайд-эффект
- создают зависимости от внешнего окружения
- замедляют тест
- требуют эмулировать поведение, которое трудно воспроизвести на реальной системе

Тестируемый класс - всегда реальный класс без инструментации.

Как создать имитацию объекта (Mock)

public <T> T Mock(Class<T> type)

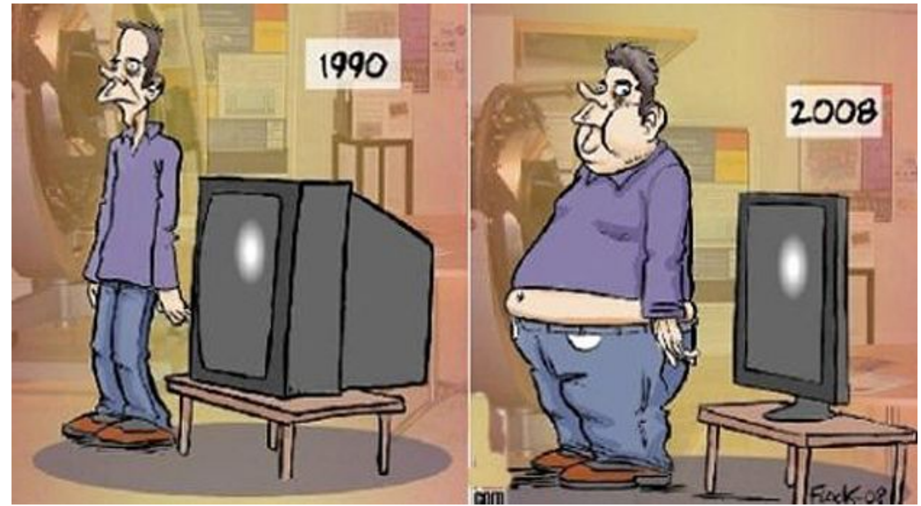
Creates a mock with the specified type.

```
Date date = Mock(Date. class)
```

```
Date date = Mock(Date)
```

```
def date = Mock(Date)
```

```
Date date = Mock()
```



Как создать заглушку (Stub)

Интересно, стаб пустой или нет?

```
given: "default stubbed object"
List list = Stub()

expect: "stub returns default value"
!list.size()
!list.empty


given: "empty stubbed list"
List list = Stub()
list.empty >> true

expect: "list is empty"
list.empty


given: "empty stubbed list"
List list = Stub{isEmpty() >> true}

expect: "list is empty"
list.empty
```

В отличие от Mockito, Spock поддерживает частичный matching аргументов, где некоторые аргументы указаны явно, я некоторые используют matchers.

```
given: "partially matched arguments"
Map<String, String> map = Mock()
map.put(_, 'ok') >> 'ko'

expect: "'ok' value results in 'ko'"
map.put(null, 'ok') == 'ko'
map.put(null, 'ok') == 'ko'
map.put('key', 'ok') == 'ko'

and: "not 'ok' value results in null"
map.put('key', 'ko') == null
map.put('key', null) == null
```

Вы, наверное, спросите, почему здесь Mock, а не Stub?

Как указать результат для искусственного метода?

```
given: "stubbed callable"
Callable<Integer> callable = Stub()
callable.call() >> 1 >> 2 >> 3

expect: "callable returns numbers"
callable.call() == 1
callable.call() == 2
callable.call() == 3
callable.call() == 3
```

```
given: "stubbed callable"
Callable<Integer> callable = Stub()
callable.call() >>> [1, 2, 3]

expect: "callable returns numbers"
callable.call() == 1
callable.call() == 2
callable.call() == 3
callable.call() == 3
```

```
given: "stubbed callable"
Callable<Integer> callable = Stub()
callable.call() >>> [1, 2] >>
        {throw new RuntimeException('fail')} >> 5

expect: "callable returns numbers"
callable.call() == 1
callable.call() == 2

when: "call to throw RuntimeException"
callable.call()

then: "RuntimeException is thrown"
thrown RuntimeException

and: "callable returns numbers"
callable.call() == 5
callable.call() == 5
```

```
given: "stubbed mock"
Callable<Integer> callable = Mock()
2 * callable.call() >> 1
1 * callable.call() >>
        {throw new RuntimeException('fail')}
_ * callable.call() >> 2

expect: "callable returns 2 numbers (1)"
callable.call() == 1
callable.call() == 1

when: "call to throw RuntimeException"
callable.call()

then: "RuntimeException is thrown"
thrown RuntimeException

and: "callable returns numbers"
callable.call() == 2
callable.call() == 2
```

Как эмулировать метод без возвращаемого результата

```
given: "mocked runnable"
Runnable runnable = Mock()
2 * runnable.run()
1 * runnable.run() >>
      {throw new RuntimeException('fail')}
_ * runnable.run()

when: "run to execute without exceptions"
runnable.run()
runnable.run()

then: "no exceptions thrown"
noExceptionThrown()

when: "run to throw RuntimeException"
runnable.run()

then: "RuntimeException is thrown"
thrown RuntimeException

when: "run to execute without exceptions"
runnable.run()
runnable.run()

then: "no exceptions thrown"
noExceptionThrown()
```

Проверка вызова методов с указанным поведением

```
given: "mocked runnable"
Runnable runnable = Mock()
runnable.run() >>
      { throw new RuntimeException('fail') }

when: "run to throw RuntimeException"
runnable.run()

then: "Exception is thrown and run called once"
thrown RuntimeException
1 * runnable.run()
```

Stop

Expected exception java.lang.RuntimeException,
but no exception was thrown

```
given: "mocked runnable"
Runnable runnable = Mock()

when: "run to throw RuntimeException"
runnable.run()

then: "RuntimeException is thrown and run
                              called once"
thrown RuntimeException
1 * runnable.run() >>
      { throw new RuntimeException('fail') }
```

Этот код - одновременно и проверка на одиночный вызов, и установка stub-поведения.

Проверка порядка вызова методов

```
given: "mocked runnable"
Runnable runnable = Mock()

when: "some methods run"
runnable.run()
runnable.run()
runnable.hashCode()

then: "the methods run in expected
     count, and unspecified order"
1 * runnable.hashCode()
0 * runnable.toString()
2 * runnable.run()
```

```
given: "mocked runnable"
Runnable runnable = Mock()

when: "some methods run"
runnable.run()
runnable.toString()

then: "at first, run() method runs"
1 * runnable.run()

then: "second method is toString()"
1 * runnable.toString()
```

Matcher _

```
given: "two mocked fake objects"
Runnable r = Mock()
Callable c = Mock()

and: "tested runnable"
Runnable runnable = {r.run(); c.call(); r.run(); c.toString()}

when: "runnable runs"
runnable.run()

then: "run runs twice, call runs once, nothing else runs except c.toString()"
2 * r.run()
1 * c.call() >> 5
_ * c.toString() >> '7'
0 * _
```

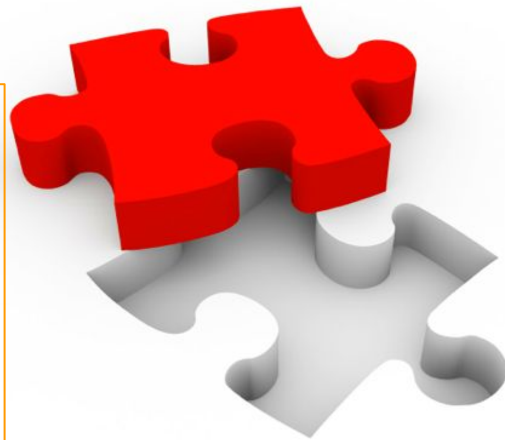
Тесты, в которых есть _ в качестве матчера могут оказаться слишком снисходительными к багам. В критических участках кода лучше обходиться без них и использовать специфичные матчеры вплоть до точных значений.

Другие матчеры (not null, type matcher, Closure)

```
given: "mocked runnable"
Runnable runnable = Mock()

when: "twice compared"
runnable.equals(runnable)
runnable.equals(new Object())

then: "parameter is not null"
2 * runnable.equals(!null)
```

```
class Test {
   public void func(String str) {}
   public void func(int number) {}
}
```

```
given: "mocked Test"
Test test = Mock()

when: "test.func(String) run"
test.func('5')
test.func('6')
test.func('7')
test.func('8')
test.func(null)

then: "only test.func(String) run"
4 * test.func(   as String)
1 * test.func(null)
0 * _
```



```
given:
ComplexChecker mock = Mock()
def man = new Manager(mock)

when:
man.call('Peter', 5)

then:
1 * mock.check({name -> name.size() > 4}, {number -> number % 2 == 0})
```

Заключение

@Issue

@Ignore / @IgnoreRest
@IgnoreIf({ os.windows })
@IgnoreIf({ env.containsKey('SKIP_TESTS') })
@Requires

@Timeout

@AutoCleanup

ты не задашь
мне вопросик?!