Interaction Based Testing

Interaction-based testing is a design and testing technique that emerged in the Extreme Programming (XP) community in the early 2000's. Focusing on the behavior of objects rather than their state, it explores how the object(s) under specification interact, by way of method calls, with their collaborators.

For example, suppose we have a `Publisher` that sends messages to its `Subscriber`'s:

```
class Publisher {
    List<Subscriber> subscribers
    void send(String message)
}

interface Subscriber {
    void receive(String message)
}

class PublisherSpec extends Specification {
    Publisher publisher = new Publisher()
}
```

How are we going to test `Publisher`? With state-based testing, we can verify that the publisher keeps track of its subscribers. The more interesting question, though, is whether a message sent by the publisher is received by the subscribers. To answer this question, we need a special implementation of `Subscriber` that listens in on the conversation between the publisher and its subscribers. Such an implementation is often called a *mock object*.

While we could certainly create a mock implementation of `Subscriber` by hand, writing and maintaining this code can get unpleasant as the number of methods and complexity of interactions increases. This is where mocking frameworks come in: They provide a way to describe the expected interactions between an object under specification and its collaborators, and can generate mock implementations of collaborators that verify these expectations.

The Java world has no shortage of popular and mature mocking frameworks: JMock, EasyMock, Mockito, to name just a few. Although each of these tools can be used together with Spock, we decided to roll our own mocking framework, tightly integrated with Spock's specification language. This decision was

How Are Mock Implementations Generated?

Like m[illegible]va mocking framev[illegible]Spock uses

driven by the desire to leverage all of Groovy's capabilities to make interaction-based tests easier to write, more readable, and ultimately more fun. We hope that by the end of this chapter, you will agree that we have achieved these goals.

Except where indicated, all features of Spock's mocking framework work both for testing Java and Groovy code.

JDK dynamic proxies (when mocking interfaces) and CGLIB proxies (when mocking classes) to generate mock implementations at runtime. Compared to implementations based on Groovy meta-programming, this has the advantage that it also works for testing Java code.

Creating Mock Objects

Mock objects are created with the `MockingApi.Mock()` method [1]. Let's create two mock subscribers:

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

Alternatively, the following Java-like syntax is supported, which may give better IDE support:

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

Here, the mock's type is inferred from the variable type on the left-hand side of the assignment.

Note

If the mock's type is given on the left-hand side of the assignment, it's permissible (though not required) to omit it on the right-hand side.

Mock objects literally implement (or, in the case of a class, extend) the type they stand in for. In other words, in our example `subscriber` *is-a* `Subscriber`. Hence it can be passed to statically typed (Java) code that expects this type.

Default Behavior of Mock Objects

Initially, mock objects have no behavior. Calling

methods on them is allowed but has no effect other than returning the default value for the method's return type (`false`, `0`, or `null`). An exception are the `Object.equals`, `Object.hashCode`, and `Object.toString` methods, which have the following default behavior: A mock object is only equal to itself, has a unique hash code, and a string representation that includes the name of the type it represents. This default behavior is overridable by stubbing the methods, which we will learn about in the Stubbing section.

Injecting Mock Objects into Code Under Specification

After creating the publisher and its subscribers, we need to make the latter known to the former:

```
class PublisherSpec extends Specifica
    Publisher publisher = new Publish
    Subscriber subscriber = Mock()
    Subscriber subscriber2 = Mock()

    def setup() {
        publisher.subscribers << subs
        publisher.subscribers << subs
    }
}
```

We are now ready to describe the expected interactions between the two parties.

Lenient vs. Strict Mocking Frameworks

Like Mockito, we firmly believe that a mocking framework should be lenient by default. This means that unexpected method calls on mock objects (or, in other words, interactions that aren't relevant for the test at hand) are allowed and answered with a default response. Conversely, mocking frameworks like EasyMock and JMock are strict by default, and throw an exception for every unexpected method call. While strictness enforces rigor, it can also lead to over-specification, resulting in brittle tests that fail with every other internal code change. Spock's mocking framework makes it easy to describe only what's relevant about an interaction, avoiding the over-specification trap.

Mocking

Mocking is the act of describing (mandatory) interactions between the object under specification and its collaborators. Here is an example:

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

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

Read out aloud: “When the publisher sends a ‘hello’ message, then both subscribers should receive that message exactly once.”

When this feature method gets run, all invocations on mock objects that occur while executing the `when` block will be matched against the interactions described in the `then:` block. If one of the interactions isn’t satisfied, a (subclass of) `InteractionNotSatisfiedError` will be thrown. This verification happens automatically and does not require any additional code.

Interactions

Let’s take a closer look at the `then:` block. It contains two *interactions*, each of which has four distinct parts: a *cardinality*, a *target constraint*, a *method constraint*, and an *argument constraint*:

```
1 * subscriber.receive("hello")
|   |            |        |
|   |            |       argument const
|   |          method constraint
|   target constraint
cardinality
```

Is an Interaction Just a Regular Method Invocation?

Not quite. While an interaction looks similar to a regular method invocation, it is simply a way to express which method invocations are expected to occur. A good way to think of an interaction is as a regular expression that all incoming invocations on mock objects are matched against. Depending on the circumstances, the interaction may match zero, one, or multiple invocations.

Cardinality

The cardinality of an interaction describes how often a method call is expected. It can either be a fixed number or a range:

```
1 * subscriber.receive("hello")
0 * subscriber.receive("hello")
(1..3) * subscriber.receive("hello")
(1.._) * subscriber.receive("hello")
(_..3) * subscriber.receive("hello")
_ * subscriber.receive("hello")
```

Target Constraint

The target constraint of an interaction describes which mock object is expected to receive

the method call:

```
1 * subscriber.receive("hello") // a call to 'subscriber'
1 * _.receive("hello")          // a call to any mock object
```

Method Constraint

The method constraint of an interaction describes which method is expected to be called:

```
1 * subscriber.receive("hello") // a method named 'receive'
1 * subscriber./r.*e/("hello")  // a method whose name matche
                                // (here: method name starts
```

When expecting a call to a getter method, Groovy property syntax *can* be used instead of method syntax:

```
1 * subscriber.status // same as: 1 * subscriber.getStatus()
```

When expecting a call to a setter method, only method syntax can be used:

```
1 * subscriber.setStatus("ok") // NOT: 1 * subscriber.status
```

Argument Constraints

The argument constraints of an interaction describe which method arguments are expected:

```
1 * subscriber.receive("hello")      // an argument that is eq
1 * subscriber.receive(!"hello")     // an argument that is un
1 * subscriber.receive()             // the empty argument lis
1 * subscriber.receive(_)            // any single argument (i
1 * subscriber.receive(*_)           // any argument list (inc
1 * subscriber.receive(!null)        // any non-null argument
1 * subscriber.receive(_ as String) // any non-null argument
1 * subscriber.receive({ it.size() > 3 }) // an argument that
                                          // (here: message l
```

Argument constraints work as expected for methods with multiple arguments:

```
1 * process.invoke("ls", "-a", _, !null, { ["abcdefghiklmnopq
```

When dealing with vararg methods, vararg syntax can also be used in the corresponding interactions:

```
interface VarArgSubscriber {
    void receive(String... messages)
}

...

subscriber.receive("hello", "goodbye")
```

Spock Deep Dive: Groovy Varargs

Groovy allows any method whose last parameter has an array type to be called in vararg style. Consequently, vararg syntax can also be used in interactions matching such methods.

Matching Any Method Call

Sometimes it can be useful to match “anything”, in some sense of the word:

```
1 * subscriber._(*_)      // any method on subscriber, with an
1 * subscriber._          // shortcut for and preferred over t

1 * _._                   // any method call on any mock objec
1 * _                     // shortcut for and preferred over t
```

Note

Although `(_.._) * _._(*_) >> _` is a valid interaction declaration, it is neither good style nor particularly useful.

Strict Mocking

Now, when would matching any method call be useful? A good example is *strict mocking*, a style of mocking where no interactions other than those explicitly declared are allowed:

```
when:
publisher.publish("hello")

then:
1 * subscriber.receive("hello") // demand one 'receive' call
_ * auditing._                  // allow any interaction with
0 * _                           // don't allow any other inte
```

`0 *` only makes sense as the last interaction of a `then:` block or method. Note the use of `_ *` (any number of calls), which allows any interaction with the auditing component.

Note

`_ *` is only meaningful in the context of strict mocking. In particular, it is never necessary when *stubbing* an invocation. For example, `_ * auditing.record(_) >> "ok"` can (and should!) be simplified to `auditing.record(_) >> "ok"`.

Where to Declare Interactions

So far, we declared all our interactions in a `then:` block. This often results in a spec that reads naturally. However, it is also permissible to put interactions anywhere *before* the `when:` block that is supposed to satisfy them. In particular, this means that interactions can be declared in a `setup` method. Interactions can also be declared in any "helper" instance method of the same specification class.

When an invocation on a mock object occurs, it is matched against interactions in the interactions' declared order. If an invocation matches multiple interactions, the earliest declared interaction that hasn't reached its upper invocation limit will win. There is one exception to this rule: Interactions declared in a `then:` block are matched against before any other interactions. This allows to override interactions declared in, say, a `setup` method with interactions declared in a `then:` block.

Spock Deep Dive: How Are Interactions Recognized?

In other words, what makes an expression an interaction declaration, rather than, say, a regular method call? Spock uses a simple syntactic rule to recognize interactions: If an expression is in statement position and is either a multiplication (`*`) or a left-shift (`>>`, `>>>`) operation, then it is considered an interaction and will be parsed accordingly. Such an expression would have little to no value in statement position, so changing its meaning works out fine. Note how the operations correspond to the syntax for declaring a cardinality (when mocking) or a response generator (when stubbing). Either of them must always be present; `foo.bar()` alone will never be considered an interaction.

Declaring Interactions at Mock Creation Time (New in 0.7)

If a mock has a set of "base" interactions that don't vary, they can be declared right at mock creation time:

```
def subscriber = Mock(Subscriber) {
    1 * receive("hello")
    1 * receive("goodbye")
}
```

This feature is particularly attractive for *stubbing* and with dedicated Stubs. Note that the interactions don't (and cannot [2]) have a target constraint; it's clear from the context which mock object they belong to.

Interactions can also be declared when initializing an instance field with a mock:

```
class MySpec extends Specification {
    Subscriber subscriber = Mock {
        1 * receive("hello")
        1 * receive("goodbye")
    }
}
```

Grouping Interactions with Same Target (New in 0.7)

Interactions sharing the same target can be grouped in a `Specification.with` block. Similar to *declaring interactions at mock creation time*, this makes it unnecessary to repeat the target constraint:

```
with(subscriber) {
    1 * receive("hello")
    1 * receive("goodbye")
}
```

A `with` block can also be used for grouping conditions with the same target.

Mixing Interactions and Conditions

A `then:` block may contain both interactions and conditions. Although not strictly required, it is customary to declare interactions before conditions:

```
when:
publisher.send("hello")

then:
1 * subscriber.receive("hello")
publisher.messageCount == 1
```

Read out aloud: "When the publisher sends a 'hello' message, then the subscriber should receive the message exactly once, and the publisher's message count should be one."

Explicit Interaction Blocks

Internally, Spock must have full information about expected interactions *before* they take place. So how is it possible for interactions to be declared in a `then:` block? The answer

is that under the hood, Spock moves interactions declared in a `then:` block to immediately before the preceding `when:` block. In most cases this works out just fine, but sometimes it can lead to problems:

```
when:
publisher.send("hello")

then:
def message = "hello"
1 * subscriber.receive(message)
```

Here we have introduced a variable for the expected argument. (Likewise, we could have introduced a variable for the cardinality.) However, Spock isn't smart enough (huh?) to tell that the interaction is intrinsically linked to the variable declaration. Hence it will just move the interaction, which will cause a `MissingPropertyException` at runtime.

One way to solve this problem is to move (at least) the variable declaration to before the `when:` block. (Fans of *data-driven testing* might move the variable into a `where:` block.) In our example, this would have the added benefit that we could use the same variable for sending the message.

Another solution is to be explicit about the fact that variable declaration and interaction belong together:

```
when:
publisher.send("hello")

then:
interaction {
    def message = "hello"
    1 * subscriber.receive(message)
}
```

Since an `MockingApi.interaction` block is always moved in its entirety, the code now works as intended.

Scope of Interactions

Interactions declared in a `then:` block are scoped to the preceding `when:` block:

```
when:
publisher.send("message1")

then:
subscriber.receive("message1")
```

```
when:
publisher.send("message2")

then:
subscriber.receive("message2")
```

This makes sure that `subscriber` receives `"message1"` during execution of the first `when:` block, and `"message2"` during execution of the second `when:` block.

Interactions declared outside a `then:` block are active from their declaration until the end of the containing feature method.

Interactions are always scoped to a particular feature method. Hence they cannot be declared in a static method, `setupSpec` method, or `cleanupSpec` method. Likewise, mock objects should not be stored in static or `@Shared` fields.

Verification of Interactions

There a two main ways in which a mock-based test can fail: An interaction can match more invocations than allowed, or it can match fewer invocations than required. The former case is detected right when the invocation happens, and causes a `TooManyInvocationsError`:

```
Too many invocations for:

2 * subscriber.receive(_) (3 invocations)
```

To make it easier to diagnose why too many invocations matched, Spock will show all invocations matching the interaction in question (new in Spock 0.7):

```
Matching invocations (ordered by last occurrence):

2 * subscriber.receive("hello")   <-- this triggered the erro
1 * subscriber.receive("goodbye")
```

According to this output, one of the `receive("hello")` calls triggered the `TooManyInvocationsError`. Note that because indistinguishable calls like the two invocations of `subscriber.receive("hello")` are aggregated into a single line of output, the first `receive("hello")` may well have occurred before the `receive("goodbye")`.

The second case (fewer invocations than required) can only be detected once execution of the `when` block has completed. (Until then, further invocations may still occur.) It causes a `TooFewInvocationsError`:

```
Too few invocations for:

1 * subscriber.receive("hello") (0 invocations)
```

Note that it doesn't matter whether the method was not called at all, the same method was called with different arguments, the same method was called on a different mock object, or a different method was called "instead" of this one; in either case, a `TooFewInvocationsError` error will occur.

To make it easier to diagnose what happened "instead" of a missing invocation, Spock will show all invocations that didn't match any interaction, ordered by their similarity with the interaction in question (new in Spock 0.7). In particular, invocations that match everything but the interaction's arguments will be shown first:

```
Unmatched invocations (ordered by similarity):

1 * subscriber.receive("goodbye")
1 * subscriber2.receive("hello")
```

Invocation Order

Often, the exact method invocation order isn't relevant and may change over time. To avoid over-specification, Spock defaults to allowing any invocation order, provided that the specified interactions are eventually satisfied:

```
then:
2 * subscriber.receive("hello")
1 * subscriber.receive("goodbye")
```

Here, any of the invocation sequences `"hello" "hello" "goodbye"`, `"hello" "goodbye" "hello"`, and `"goodbye" "hello" "hello"` will satisfy the specified interactions.

In those cases where invocation order matters, you can impose an order by splitting up interactions into multiple `then:` blocks:

```
then:
2 * subscriber.receive("hello")

then:
1 * subscriber.receive("goodbye")
```

Now Spock will verify that both `"hello"`'s are received before the `"goodbye"`. In other words, invocation order is enforced *between* but not *within* `then:` blocks.

> **Note**
>
> Splitting up a `then:` block with `and:` does not impose any ordering, as `and:` is only meant for documentation purposes and doesn't carry any semantics.

Mocking Classes

Besides interfaces, Spock also supports mocking of classes. Mocking classes works just like mocking interfaces; the only additional requirement is to put `cglib-nodep-2.2` or higher and `objenesis-1.2` or higher on the class path. If either of these libraries is missing from the class path, Spock will gently let you know.

Stubbing

Stubbing is the act of making collaborators respond to method calls in a certain way. When stubbing a method, you don't care if and how many times the method is going to be called; you just want it to return some value, or perform some side effect, *whenever* it gets called.

For the sake of the following examples, let's modify the `Subscriber`'s `receive` method to return a status code that tells if the subscriber was able to process a message:

```
interface Subscriber {
    String receive(String message)
}
```

Now, let's make the `receive` method return `"ok"` on every invocation:

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

Read out aloud: "*Whenever* the subscriber receives a message, *make* it respond with 'ok'."

Compared to a mocked interaction, a stubbed interaction has no cardinality on the left end, but adds a *response generator* on the right end:

```
subscriber.receive(_) >> "ok"
|          |       |     |
|          |       |     response generator
|          |       argument constraint
|          method constraint
target constraint
```

A stubbed interaction can be declared in the usual places: either inside a `then:` block, or anywhere before a `when:` block. (See *Where to Declare Interactions* for the details.) If a mock object is only used for stubbing, it's common to declare interactions *at mock creation time* or in a `setup:` block.

Returning Fixed Values

We have already seen the use of the right-shift (>>) operator to return a fixed value:

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

To return different values for different invocations, use multiple interactions:

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

This will return `"ok"` whenever `"message1"` is received, and `"fail"` whenever `"message2"` is received. There is no limit as to which values can be returned, provided they are compatible with the method's declared return type.

Returning Sequences of Values

To return different values on successive invocations, use the triple-right-shift (>>>) operator:

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

This will return `"ok"` for the first invocation, `"error"` for the second and third invocation, and `"ok"` for all remaining invocations. The right-hand side must be a value that Groovy knows how to iterate over; in this example, we've used a plain list.

Computing Return Values

To compute a return value based on the method's argument, use the the right-shift (>>) operator together with a closure. If the closure declares a single untyped parameter, it gets passed the method's argument list:

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

Here `"ok"` gets returned if the message is more than three characters long, and `"fail"` otherwise.

In most cases it would be more convenient to have direct access to the method's

arguments. If the closure declares more than one parameter or a single *typed* parameter, method arguments will be mapped one-by-one to closure parameters [4]:

```
subscriber.receive(_) >> { String message -> message.size() >
```

This response generator behaves the same as the previous one, but is arguably more readable.

If you find yourself in need of more information about a method invocation than its arguments, have a look at `org.spockframework.mock.IMockInvocation`. All methods declared in this interface are available inside the closure, without a need to prefix them. (In Groovy terminology, the closure *delegates* to an instance of `IMockInvocation`.)

Performing Side Effects

Sometimes you may want to do more than just computing a return value. A typical example is throwing an exception. Again, closures come to the rescue:

```
subscriber.receive(_) >> { throw new InternalError("ouch") }
```

Of course, the closure can contain more code, for example a `println` statement. It will get executed every time an incoming invocation matches the interaction.

Chaining Method Responses

Method responses can be chained:

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

This will return `"ok", "fail", "ok"` for the first three invocations, throw `InternalError` for the fourth invocations, and return `ok` for any further invocation.

Combining Mocking and Stubbing

Mocking and stubbing go hand-in-hand:

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

When mocking and stubbing the same method call, they have to happen in the same

interaction. In particular, the following Mockito-style splitting of stubbing and mocking into two separate statements will *not* work:

```
setup:
subscriber.receive("message1") >> "ok"

when:
publisher.send("message1")

then:
1 * subscriber.receive("message1")
```

As explained in *Where to Declare Interactions*, the `receive` call will first get matched against the interaction in the `then:` block. Since that interaction doesn't specify a response, the default value for the method's return type (`null` in this case) will be returned. (This is just another facet of Spock's lenient approach to mocking.). Hence, the interaction in the `setup:` block will never get a chance to match.

> **Note**
>
> Mocking and stubbing of the same method call has to happen in the same interaction.

Other Kinds of Mock Objects (New in 0.7)

So far, we have created mock objects with the `MockingApi.Mock` method. Aside from this method, the `MockingApi` class provides a couple of other factory methods for creating more specialized kinds of mock objects.

Stubs

A *stub* is created with the `MockingApi.Stub` factory method:

```
def subscriber = Stub(Subscriber)
```

Whereas a mock can be used both for stubbing and mocking, a stub can only be used for stubbing. Limiting a collaborator to a stub communicates its role to the readers of the specification.

> **Note**
>
> If a stub invocation matches a *mandatory* interaction (like `1 * foo.bar()`), an `InvalidSpecException` is thrown.

Like a mock, a stub allows unexpected invocations. However, the values returned by a stub in such cases are more ambitious:

- For primitive types, the primitive type's default value is returned.
- For non-primitive numerical values (like `BigDecimal`), zero is returned.
- For non-numerical values, an "empty" or "dummy" object is returned. This could mean an empty String, an empty collection, an object constructed from its default constructor, or another stub returning default values. See class `org.spockframework.mock.EmptyOrDummyResponse` for the details.

A stub often has a fixed set of interactions, which makes *declaring interactions at mock creation time* particularly attractive:

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

Spies

(Think twice before using this feature. It might be better to change the design of the code under specification.)

A *spy* is created with the `MockingApi.Spy` factory method:

```
def subscriber = Spy(SubscriberImpl, constructorArgs: ["Fred"
```

A spy is always based on a real object. Hence you must provide a class type rather than an interface type, along with any constructor arguments for the type. If no constructor arguments are provided, the type's default constructor will be used.

Method calls on a spy are automatically delegated to the real object. Likewise, values returned from the real object's methods are passed back to the caller via the spy.

After creating a spy, you can listen in on the conversation between the caller and the real object underlying the spy:

```
1 * subscriber.receive(_)
```

Apart from making sure that `receive` gets called exactly once, the conversation between the publisher and the `SubscriberImpl` instance underlying the spy remains unaltered.

When stubbing a method on a spy, the real method no longer gets called:

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

Instead of calling `SubscriberImpl.receive`, the `receive` method will now simply return `"ok"`.

Sometimes, it is desirable to both execute some code *and* delegate to the real method:

```
subscriber.receive(_) >> { String message -> callRealMethod()
```

Here we use `callRealMethod()` to delegate the method invocation to the real object. Note that we don't have to pass the `message` argument along; this is taken care of automatically. *`callRealMethod()`* returns the real invocation's result, but in this example we opted to return our own result instead. If we had wanted to pass a different message to the real method, we could have used `callRealMethodWithArgs("changed message")`.

Partial Mocks

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Spies can also be used as partial mocks:

```
// this is now the object under specification, not a collabor
def persister = Spy(MessagePersister) {
  // stub a call on the same object
  isPersistable(_) >> true
}

when:
persister.receive("msg")

then:
// demand a call on the same object
1 * persister.persist("msg")
```

Groovy Mocks (New in 0.7)

So far, all the mocking features we have seen work the same no matter if the calling code is written in Java or Groovy. By leveraging Groovy's dynamic capabilities, Groovy mocks offer some additional features specifically for testing Groovy code. They are created with

the `MockingApi.GroovyMock()`, `MockingApi.GroovyStub()`, and `MockingApi.GroovySpy()` factory methods.

> **When Should Groovy Mocks be Favored over Regular Mocks?**
>
> Groovy mocks should be used when the code under specification is written in Groovy *and* some of the unique Groovy mock features are needed. When called from Java code, Groovy mocks will behave like regular mocks. Note that it isn't necessary to use a Groovy mock merely because the code under specification and/or mocked type is written in Groovy. Unless you have a concrete reason to use a Groovy mock, prefer a regular mock.

Mocking Dynamic Methods

All Groovy mocks implement the `GroovyObject` interface. They support the mocking and stubbing of dynamic methods as if they were physically declared methods:

```
def subscriber = GroovyMock(Subscriber)

1 * subscriber.someDynamicMethod("hello")
```

Mocking All Instances of a Type

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Usually, Groovy mocks need to be injected into the code under specification just like regular mocks. However, when a Groovy mock is created as *global*, it automagically replaces all real instances of the mocked type for the duration of the feature method [3]:

```
def publisher = new Publisher()
publisher << new RealSubscriber() << new RealSubscriber()

def anySubscriber = GroovyMock(RealSubscriber, global: true)

when:
publisher.publish("message")

then:
2 * anySubscriber.receive("message")
```

Here, we set up the publisher with two instances of a real subscriber implementation. Then we create a global mock of the *same* type. This reroutes all method calls on the real subscribers to the mock object. The mock object's instance isn't ever passed to the

publisher; it is only used to describe the interaction.

> **Note**
>
> A global mock can only be created for a class type. It effectively replaces all instances of that type for the duration of the feature method.

Since global mocks have a somewhat, well, global effect, it's often convenient to use them together with `GroovySpy`. This leads to the real code getting executed *unless* an interaction matches, allowing you to selectively listen in on objects and change their behavior just where needed.

Mocking Constructors

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Global mocks support mocking of constructors:

```
def anySubscriber = GroovySpy(RealSub

1 * new RealSubscriber("Fred")
```

Since we are using a spy, the object returned from the constructor call remains unchanged. To change which object gets constructed, we can stub the constructor:

```
new RealSubscriber("Fred") >> new Rea
```

Now, whenever some code tries to construct a subscriber named Fred, we'll construct a subscriber named Barney instead.

> **How Are Global Groovy Mocks Implemented?**
>
> Global Groovy mocks get their super powers from Groovy meta-programming. To be more precise, every globally mocked type is assigned a custom meta class for the duration of the feature method. Since a global Groovy mock is still based on a CGLIB proxy, it will retain its general mocking capabilities (but not its super powers) when called from Java code.

Mocking Static Methods

(Think twice before using this feature. It might be better to change the design of the code under specification.)

Global mocks support mocking and stubbing of static methods:

```
def anySubscriber = GroovySpy(RealSubscriber, global: true)
```

```
1 * RealSubscriber.someStaticMethod("hello") >> 42
```

The same works for dynamic static methods.

When a global mock is used solely for mocking constructors and static methods, the mock's instance isn't really needed. In such a case one can just write:

```
GroovySpy(RealSubscriber, global: true)
```

Advanced Features (New in 0.7)

Most of the time you shouldn't need these features. But if you do, you'll be glad to have them.

A la Carte Mocks

At the end of the day, the `Mock()`, `Stub()`, and `Spy()` factory methods are just precanned ways to create mock objects with a certain configuration. If you want more fine-grained control over a mock's configuration, have a look at the `org.spockframework.mock.IMockConfiguration` interface. All properties of this interface [5] can be passed as named arguments to the `Mock()` method. For example:

```
def person = Mock(name: "Fred", type: Person, defaultResponse
```

Here, we create a mock whose default return values match those of a `Mock()`, but whose invocations aren't verified (as for a `Stub()`). Instead of passing `ZeroOrNullResponse`, we could have supplied our own custom `org.spockframework.mock.IDefaultResponse` for responding to unexpected method invocations.

Detecting Mock Objects

To find out whether a particular object is a Spock mock object, use a `org.spockframework.mock.MockDetector`:

```
def detector = new MockDetector()
def list1 = []
def list2 = Mock(List)

expect:
!detector.isMock(list1)
detector.isMock(list2)
```

A detector can also be used to get more information about a mock object:

```
def mock = detector.asMock(list2)

expect:
mock.name == "list2"
mock.type == List
mock.nature == MockNature.MOCK
```

Further Reading

To learn more about interaction-based testing, we recommend the following resources:

- Endo-Testing: Unit Testing with Mock Objects

  Paper from the XP2000 conference that introduces the concept of mock objects.

- Mock Roles, not Objects

  Paper from the OOPSLA2004 conference that explains how to do mocking *right*.

- Mocks Aren't Stubs

  Martin Fowler's take on mocking.

- Growing Object-Oriented Software Guided by Tests

  TDD pioneers Steve Freeman and Nat Pryce explain in detail how test-driven development and mocking work in the real world.

Footnotes

[1] For additional ways to create mock objects, see *Other Kinds of Mock Objects (New in 0.7)* and *A la Carte Mocks*.

[2] The `subscriber` variable cannot be referenced from the closure because it is being declared as part of the same statement.

[3] You may know this behavior from Groovy's MockFor and StubFor facilities.

[4] The destructuring semantics for closure arguments come straight from Groovy.

[5] Because mock configurations are immutable, the interface contains just the properties' getters.