LDTS 2021/2022

Rui Maranhão

Last Lecture

- Test-first
- Types of tests
- Unit Testing
    - Method testing
    - Test doubles

Test Doubles

https://martinfowler.com/articles/mocksArentStubs.html

- isolate units
- reduce dependencies between teams
- reduce the overhead of testing set up, e.g. in memory databases
- simulate infrequent, or difficult to generate, test cases, e.g. the server is down

Stubs and Mocks

what are the differences

How to test a method that sends a message to an email service?

do not want to send emails during tests

Stubs

```
public interface MailService {
  public void send (Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();

  public void send (Message msg) {
    messages.add(msg);
  }

  public int numberSent() {
    return messages.size();
  }
}
```

(http://martinfowler.com/articles/mocksArentStubs.html)

```
class OrderStateTester…

  public void testOrderSendsMailIfUnfilled() {
    // set up
    Order order = new Order(TALISKER, 51);
    Warehouse warehouse = new Warehouse();
    warehouse.setInventory(TALISKER, 50);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);

    // execute
    order.fill(warehouse);

    // verify
    assertFalse(order.isFilled())
    assertEquals(1, mailer.numberSent());
  }
```

(http://martinfowler.com/articles/mocksArentStubs.html)

Stubs are objects that provide a canned answer to calls made during the test

Stubs use
state verification

Mocks

```
class OrderInteractionTester {
    @Mocked
    MailService mailer;

    @Test
    public void testOrderSendsMailIfUnfilled() {
        // set up
        Order order = new Order(TALISKER, 51);
        Warehouse warehouse = new Warehouse();
        warehouse.setInventory(TALISKER, 50);
        order.setMailer(mailer);

        new Expectations() {{
            mailer.send((Message) withNotNull());
        }};

        // execute
        order.fill(warehouse);

        // verify
        assertFalse(order.isFilled());
    }
}
```

(adapted from http://martinfowler.com/articles/mocksArentStubs.html)

Are pre-programmed with expectations which form a specification of the calls they are expected to receive

Mocks use behaviour verification

Today

- Test doubles
  - JMockit
  - Spock
- Object testing

JMockit

http://www.baeldung.com/jmockit-101

http://www.baeldung.com/jmockit-expectations

http://jmockit.org/tutorial.html

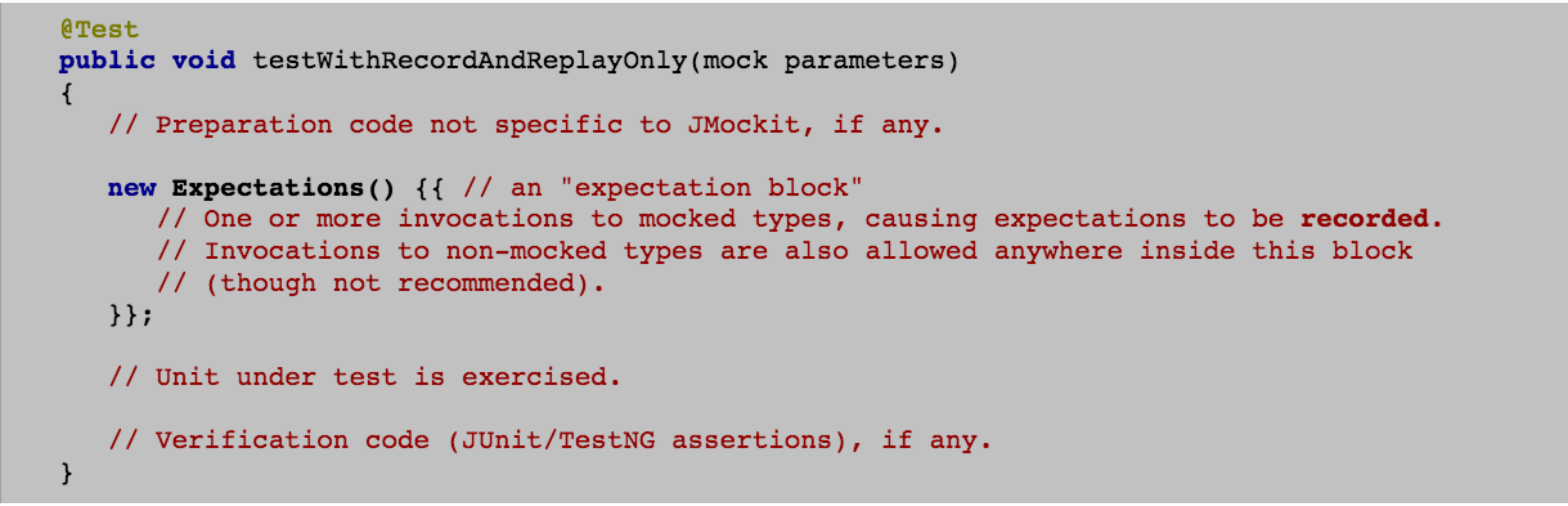
The record-replay-verify model

```java
@Test
public void someTestMethod()
{
   // 1. Preparation: whatever is required before the code under test can be exercised.
   ...
   // 2. The code under test is exercised, usually by calling a public method.
   ...
   // 3. Verification: whatever needs to be checked to make sure the code exercised by
   //    the test did its job.
   ...
}
```

```
// "Dependency" is mocked for all tests in this test class.
// The "mockInstance" field holds a mocked instance automatically created for use in each test.
@Mocked Dependency mockInstance;

@Test
public void doBusinessOperationXyz(@Mocked final AnotherDependency anotherMock)
{
   ...
   new Expectations() {{ // an "expectation block"
      ...
      // Record an expectation, with a given value to be returned:
      mockInstance.mockedMethod(...); result = 123;
      ...
   }};
   ...
   // Call the code under test.
   ...
   new Verifications() {{ // a "verification block"
      // Verifies an expected invocation:
      anotherMock.save(any); times = 1;
   }};
   ...
}
```

Several combinations



```
@Test
public void testWithRecordAndReplayOnly(mock parameters)
{
   // Preparation code not specific to JMockit, if any.

   new Expectations() {{ // an "expectation block"
      // One or more invocations to mocked types, causing expectations to be recorded.
      // Invocations to non-mocked types are also allowed anywhere inside this block
      // (though not recommended).
   }};

   // Unit under test is exercised.

   // Verification code (JUnit/TestNG assertions), if any.
}
```

Several combinations

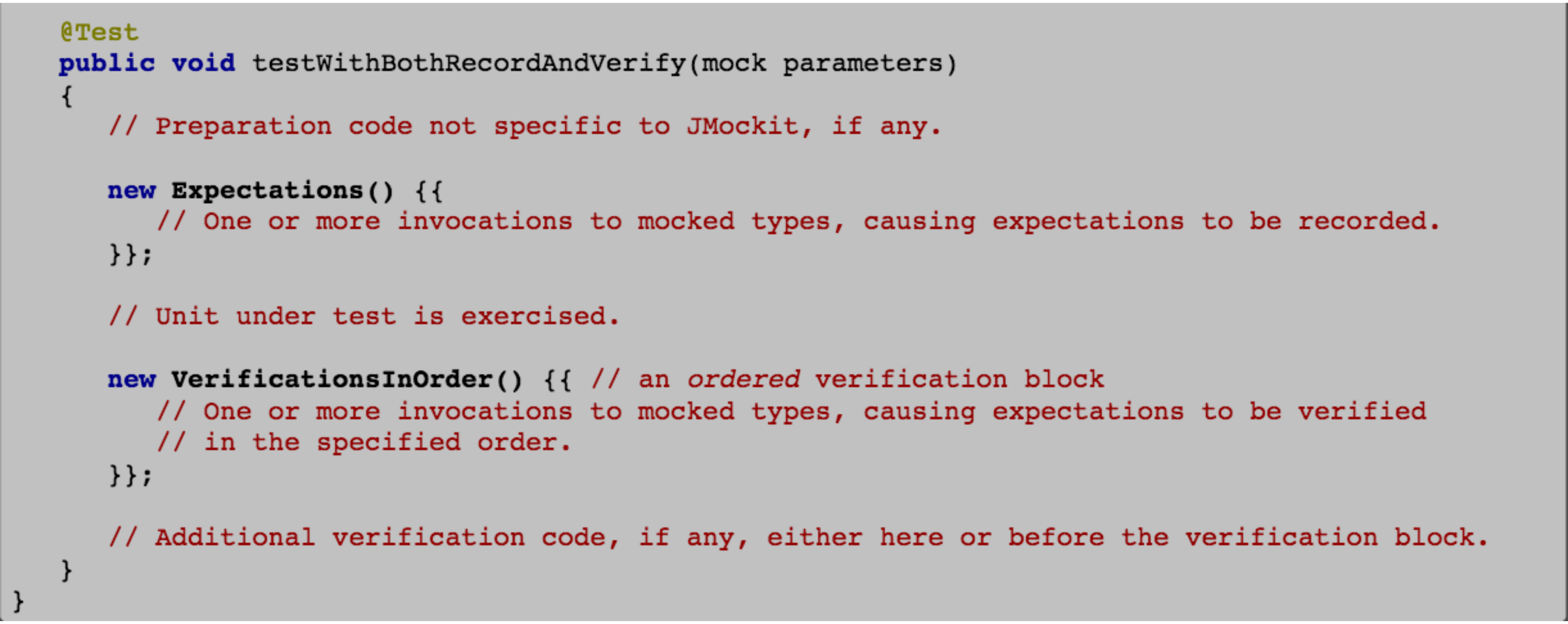
```
@Test
public void testWithReplayAndVerifyOnly(mock parameters)
{
   // Preparation code not specific to JMockit, if any.

   // Unit under test is exercised.

   new Verifications() {{ // a "verification block"
      // One or more invocations to mocked types, causing expectations to be verified.
      // Invocations to non-mocked types are also allowed anywhere inside this block
      // (though not recommended).
   }};

   // Additional verification code, if any, either here or before the verification block.
}
```

Several combinations



```
   @Test
   public void testWithBothRecordAndVerify(mock parameters)
   {
      // Preparation code not specific to JMockit, if any.

      new Expectations() {{
         // One or more invocations to mocked types, causing expectations to be recorded.
      }};

      // Unit under test is exercised.

      new VerificationsInOrder() {{ // an ordered verification block
         // One or more invocations to mocked types, causing expectations to be verified
         // in the specified order.
      }};

      // Additional verification code, if any, either here or before the verification block.
   }
}
```

Recording results for an expectation (1)

```
public class UnitUnderTest
{
(1)private final DependencyAbc abc = new DependencyAbc();

   public void doSomething()
   {
(2)   int n = abc.intReturningMethod();

      for (int i = 0; i < n; i++) {
         String s;

         try {
(3)         s = abc.stringReturningMethod();
         }
         catch (SomeCheckedException e) {
            // somehow handle the exception
         }

         // do some other stuff
      }
   }
}
```

Recording results for an expectation (2)

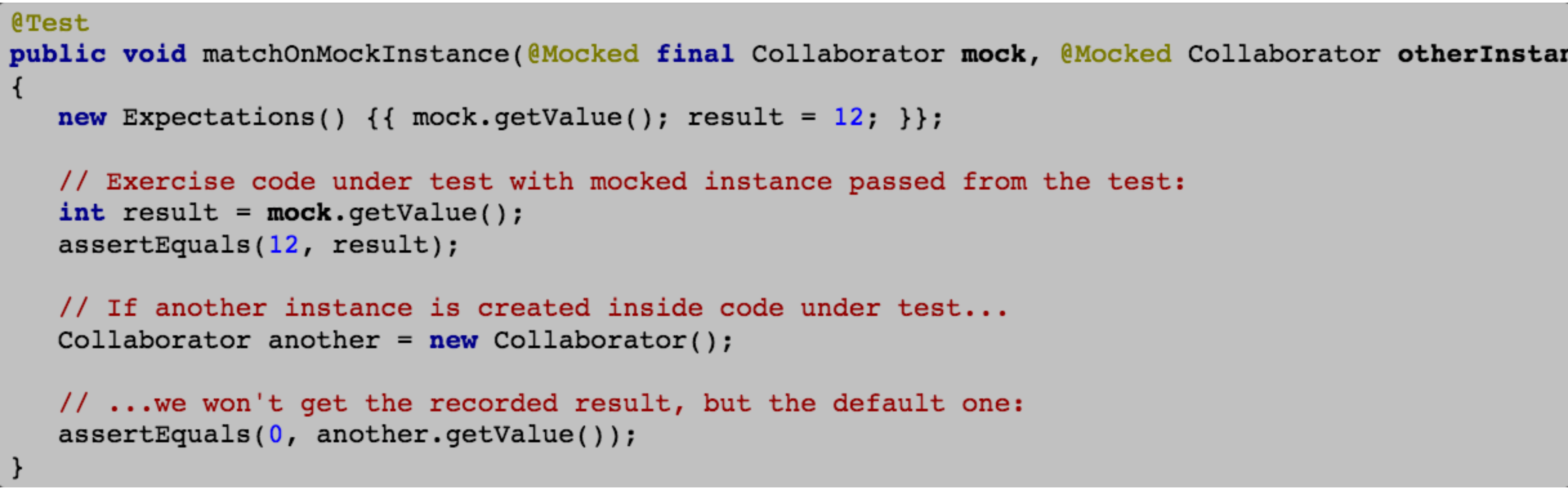
```
@Test
public void doSomethingHandlesSomeCheckedException(@Mocked final DependencyAbc abc) throws Except
{
   new Expectations() {{
(1)   new DependencyAbc();

(2)   abc.intReturningMethod(); result = 3;

(3)   abc.stringReturningMethod();
      returns("str1", "str2");
      result = new SomeCheckedException();
   }};

   new UnitUnderTest().doSomething();
}
```

Declaring multiple mocked instances



```java
@Test
public void matchOnMockInstance(@Mocked final Collaborator mock, @Mocked Collaborator otherInstar
{
   new Expectations() {{ mock.getValue(); result = 12; }};

   // Exercise code under test with mocked instance passed from the test:
   int result = mock.getValue();
   assertEquals(12, result);

   // If another instance is created inside code under test...
   Collaborator another = new Collaborator();

   // ...we won't get the recorded result, but the default one:
   assertEquals(0, another.getValue());
}
```

Instances created with a given constructor

```
@Test
public void newCollaboratorsWithDifferentBehaviors(@Mocked Collaborator anyCollaborator)
{
   // Record different behaviors for each set of instances:
   new Expectations() {{
      // One set, instances created with "a value":
      Collaborator col1 = new Collaborator("a value");
      col1.doSomething(anyInt); result = 123;

      // Another set, instances created with "another value":
      Collaborator col2 = new Collaborator("another value");
      col2.doSomething(anyInt); result = new InvalidStateException();
   }};

   // Code under test:
   new Collaborator("a value").doSomething(5); // will return 123
   ...
   new Collaborator("another value").doSomething(0); // will throw the exception
   ...
}
```

or

```
@Test
public void newCollaboratorsWithDifferentBehaviors(
   @Mocked final Collaborator col1, @Mocked final Collaborator col2)
{
   new Expectations() {{
      // Map separate sets of future instances to separate mock parameters:
      new Collaborator("a value"); result = col1;
      new Collaborator("another value"); result = col2;

      // Record different behaviors for each set of instances:
      col1.doSomething(anyInt); result = 123;
      col2.doSomething(anyInt); result = new InvalidStateException();
   }};

   // Code under test:
   new Collaborator("a value").doSomething(5); // will return 123
   ...
   new Collaborator("another value").doSomething(0); // will throw the exception
   ...
}
```

Using the "any" fields for argument matching

```java
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   final DataItem item = new DataItem(...);

   new Expectations() {{
      // Will match "voidMethod(String, List)" invocations where the first argument is
      // any string and the second any list.
      abc.voidMethod(anyString, (List<?>) any);
   }};

   new UnitUnderTest().doSomething(item);

   new Verifications() {{
      // Matches invocations to the specified method with any value of type long or Long.
      abc.anotherVoidMethod(anyLong);
   }};
}
```

Using the "with" methods for argument matching

```java
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   final DataItem item = new DataItem(...);

   new Expectations() {{
      // Will match "voidMethod(String, List)" invocations with the first argument
      // equal to "str" and the second not null.
      abc.voidMethod("str", (List<?>) withNotNull());

      // Will match invocations to DependencyAbc#stringReturningMethod(DataItem, String)
      // with the first argument pointing to "item" and the second one containing "xyz".
      abc.stringReturningMethod(withSameInstance(item), withSubstring("xyz"));
   }};

   new UnitUnderTest().doSomething(item);

   new Verifications() {{
      // Matches invocations to the specified method with any long-valued argument.
      abc.anotherVoidMethod(withAny(1L));
   }};
}
```

Using the null value to match any object reference

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   ...
   new Expectations() {{
      abc.voidMethod(anyString, null);
   }};
   ...
}
```

only applicable when at least one explicit argument matcher
(either a "with" method or an "any" field) is used for the expectation

Specifying invocation count constraints

```
@Test
public void someTestMethod(@Mocked final DependencyAbc abc)
{
   new Expectations() {{
      // By default, at least one invocation is expected, i.e. "minTimes = 1":
      new DependencyAbc();

      // At least two invocations are expected:
      abc.voidMethod(); minTimes = 2;

      // 1 to 5 invocations are expected:
      abc.stringReturningMethod(); minTimes = 1; maxTimes = 5;
   }};

   new UnitUnderTest().doSomething();
}

@Test
public void someOtherTestMethod(@Mocked final DependencyAbc abc)
{
   new UnitUnderTest().doSomething();

   new Verifications() {{
      // Verifies that zero or one invocations occurred, with the specified argument value:
      abc.anotherVoidMethod(3); maxTimes = 1;

      // Verifies the occurrence of at least one invocation with the specified arguments:
      DependencyAbc.someStaticMethod("test", false); // "minTimes = 1" is implied
   }};
}
```

Delegates: specifying custom results

```java
@Test
public void delegatingInvocationsToACustomDelegate(@Mocked final DependencyAbc anyAbc)
{
   new Expectations() {{
      anyAbc.intReturningMethod(anyInt, null);
      result = new Delegate() {
         int aDelegateMethod(int i, String s)
         {
            return i == 1 ? i : s.length();
         }
      };
   }};

   // Calls to "intReturningMethod(int, String)" will execute the delegate method above.
   new UnitUnderTest().doSomething();
}
```

Delegates: specifying custom results

```
@Test
public void delegatingConstructorInvocations(@Mocked Collaborator anyCollaboratorInstance)
{
   new Expectations() {{
      new Collaborator(anyInt);
      result = new Delegate() {
         void delegate(int i) { if (i < 1) throw new IllegalArgumentException(); }
      };
   }};

   // The first instantiation using "Collaborator(int)" will execute the delegate above.
   new Collaborator(4);
}
```

Spock's Interaction-based testing

MustRead

http://spockframework.org/spock/docs/1.0/interaction_based_testing.html

Spock cheatsheet: http://jakubdziworski.github.io/java/groovy/spock/2016/05/14/spock-cheatsheet.html

Junit comparison

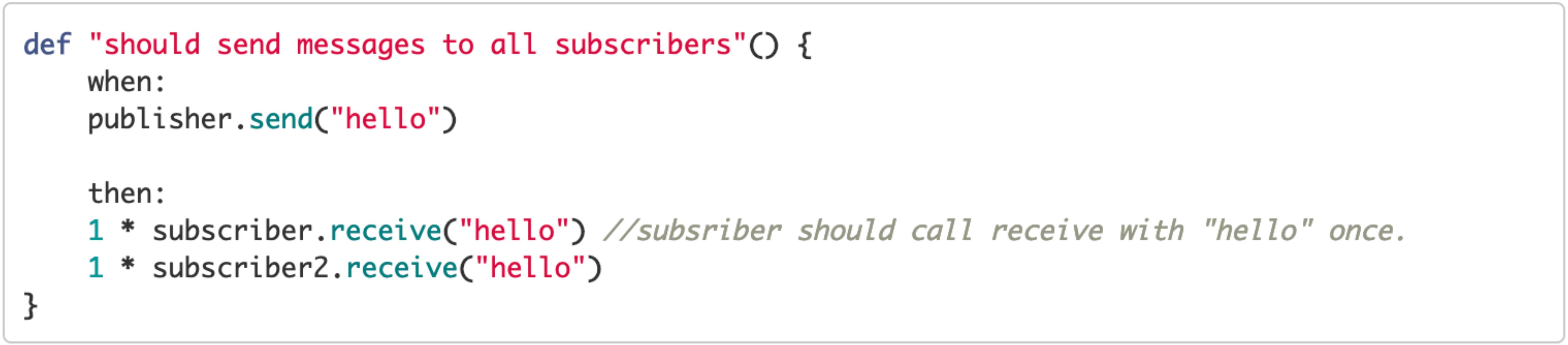
Spock	JUnit
Specification	Test class
setup()	@Before
cleanup()	@After
setupSpec()	@BeforeClass
cleanupSpec()	@AfterClass
Feature	Test
Feature method	Test method
Data-driven feature	Theory
Condition	Assertion
Exception condition	@Test(expected=…)
Interaction	Mock expectation (e.g. in Mockito)

Create Mock

- In spock, mocks are lenient
  - i.e., return default value for undefined mock calls

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

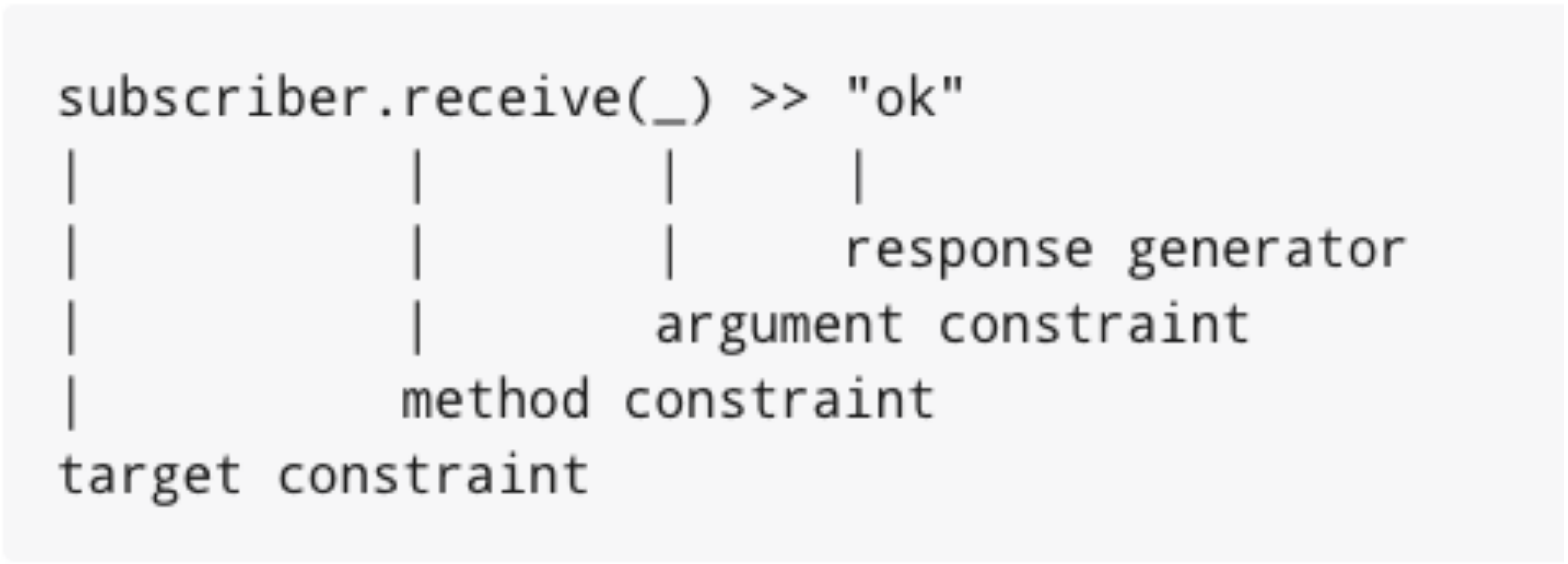
Using Mocks (Interactions)



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

    then:
    1 * subscriber.receive("hello") //subsriber should call receive with "hello" once.
    1 * subscriber2.receive("hello")
}
```

Parts of Interactions



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

Cardinality

```
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
                                     // (rarely needed; see 'Strict Mocking')
```

Strict Mocking

- *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 on 'subscriber'
_ * auditing._                  // allow any interaction with 'auditing'
0 * _                           // don't allow any other interaction
```

Constraints

Target

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

Method

```
1 * subscriber.receive("hello") // a method named 'receive'
1 * subscriber./r.*e/("hello")  // a method whose name matches the given regular expression
                                // (here: method name starts with 'r' and ends in 'e')
```

Argument

```
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 examp
le)
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
                                          // (here: message length is greater than 3)
```

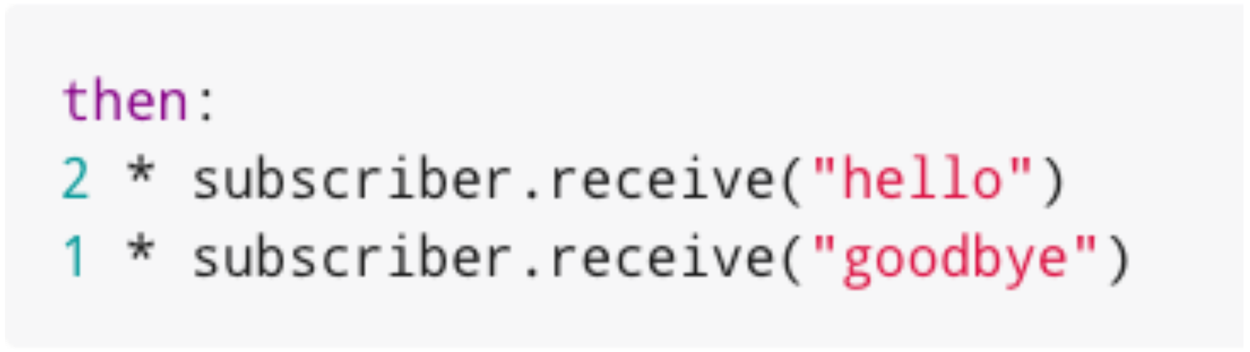
Specify mock calls at creation

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

Group interactions

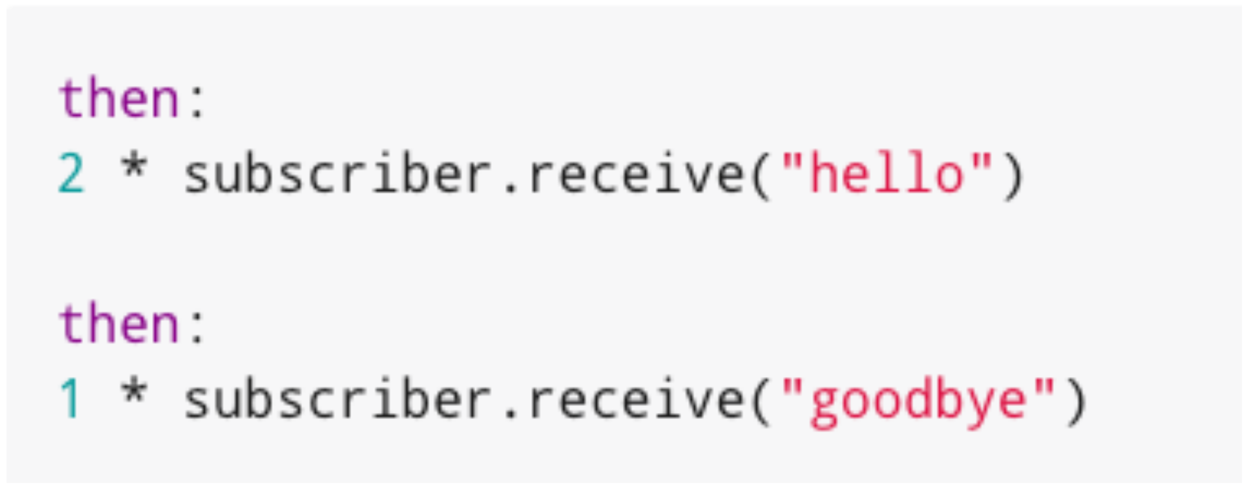
```
with(mock) {
    1 * receive("hello")
    1 * receive("goodbye")
}
```

Invocation Order



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

vs



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

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

Mock’s Expected Value

- Do not have cardinality (matches invocation any times)

```
def subsriber = Stub(Subscriber)
...
subscriber.receive(_) >> "ok"
```

- Whenever subscriber receives a message, make it respond ‘ok’

Returning different values on successive calls

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

Interesting; not related to mocks, though

- Extensions

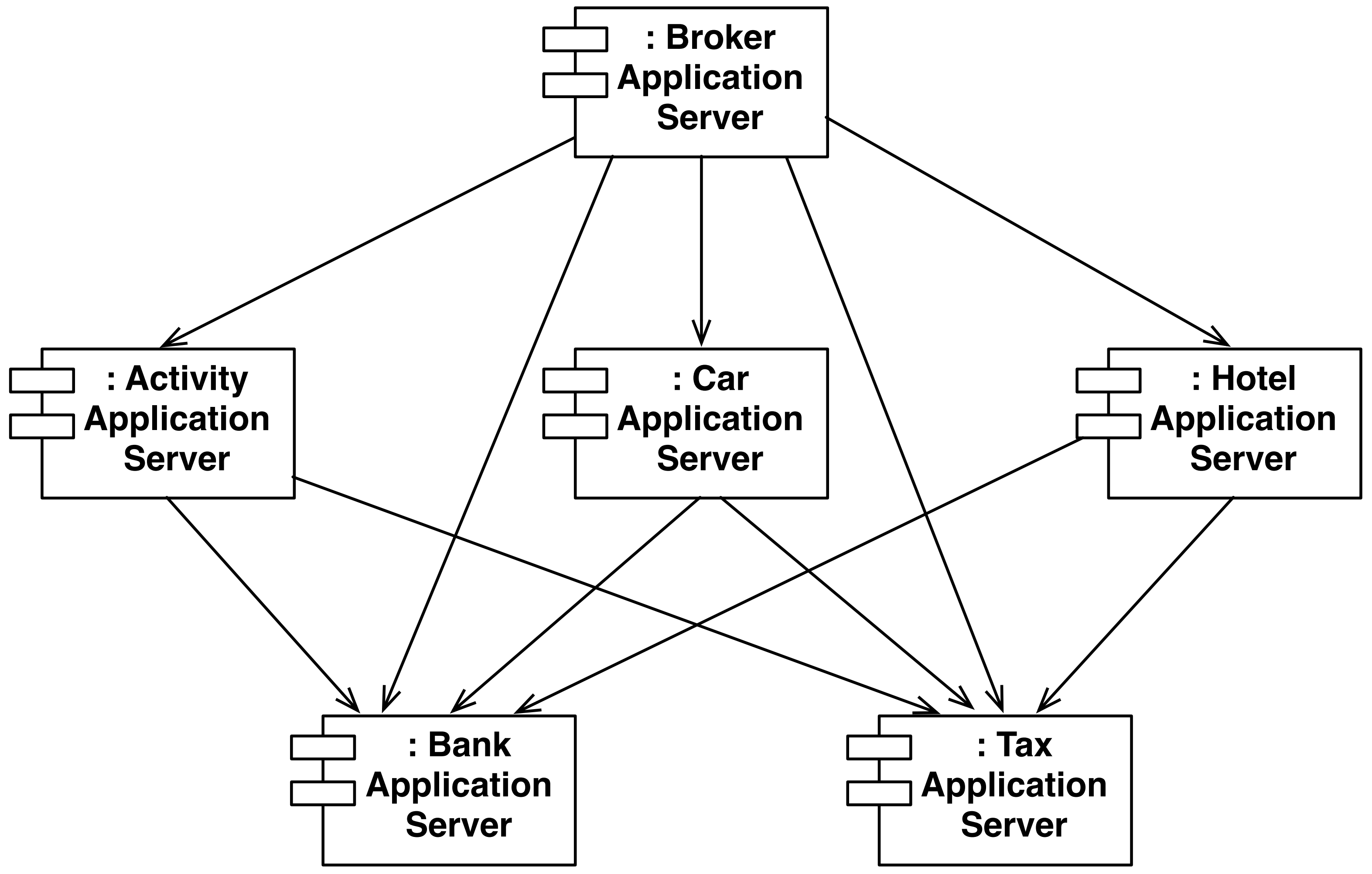
```
@Ignore(reason = "TODO")
@IgnoreRest
@IgnoreIf({ spock.util.environment.Jvm.isJava5()) })
@Requires({ os.windows })
@Timeout(5)
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
@Title("This tests if..."
@Narrative("some detailed explanation")
@Issue("http://redmine/23432")
@Subject
```

http://spockframework.org/spock/docs/1.3-RC1/extensions.html

Where to declare Interactions?

- **then:** often results in a spec that reads naturally
- Permissible to put them anywhere, including **setup**/ **given**
- When an invocation on a mock object occurs, it is matched against interactions' declared order
  - Except for those in a **then:** block
    - This are matched first

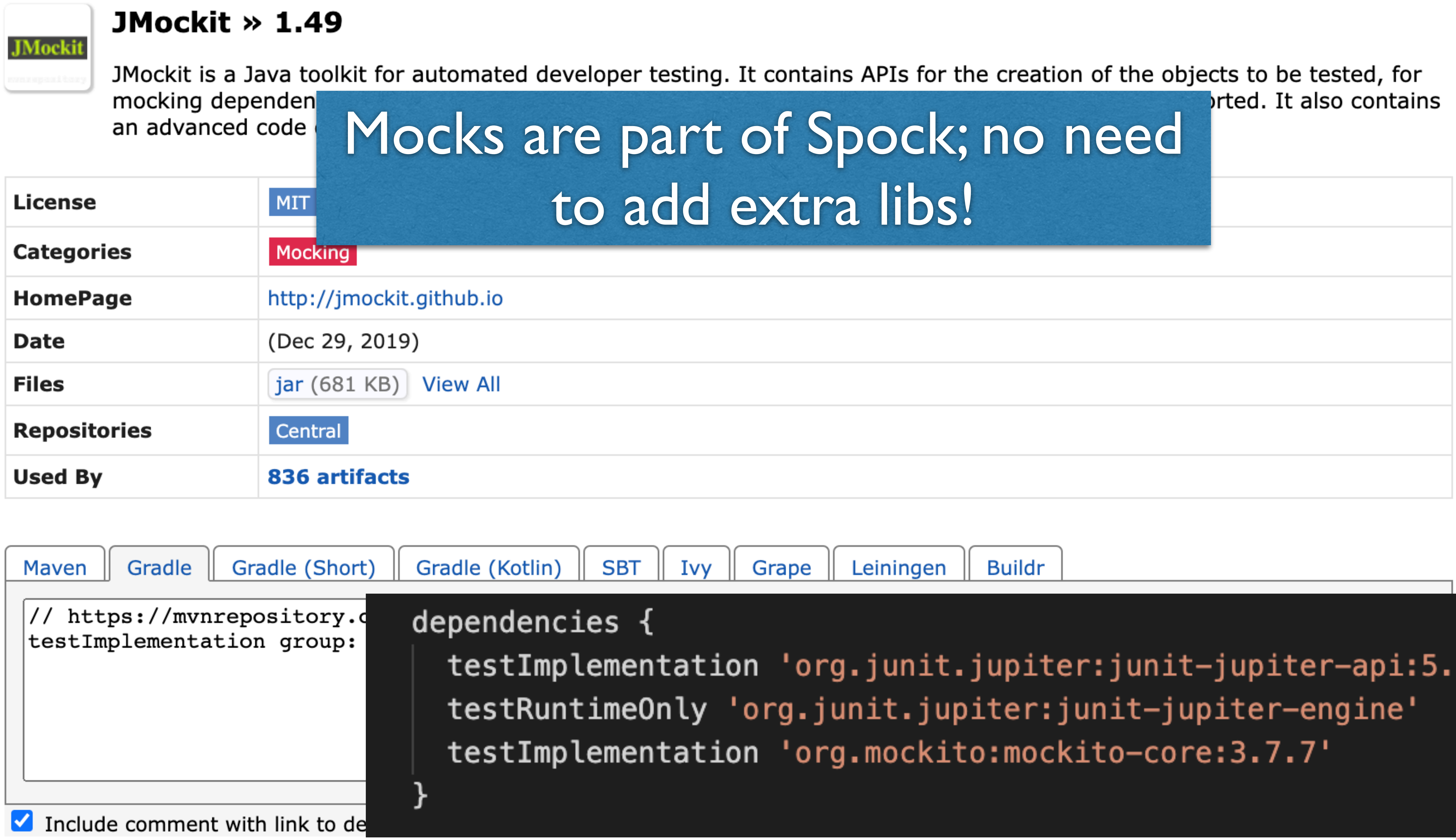
JMockit/Spock: An example


: Broker Application Server
: Activity Application Server
: Car Application Server
: Hotel Application Server
: Bank Application Server
: Tax Application Server

reduce dependencies between teams

simulate infrequent, or difficult to generate, test cases

lower cost to run the tests



JMockit » 1.49

JMockit is a Java toolkit for automated developer testing. It contains APIs for the creation of the objects to be tested, for mocking dependen… …rted. It also contains an advanced code …

License	MIT
Categories	Mocking
HomePage	http://jmockit.github.io
Date	(Dec 29, 2019)
Files	jar (681 KB) View All
Repositories	Central
Used By	836 artifacts

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
// https://mvnrepository.c
testImplementation group:
```

☑ Include comment with link to de

Mocks are part of Spock; no need to add extra libs!

```
dependencies {
  testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
  testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'
  testImplementation 'org.mockito:mockito-core:3.7.7'
}
```

Test Methods + Class

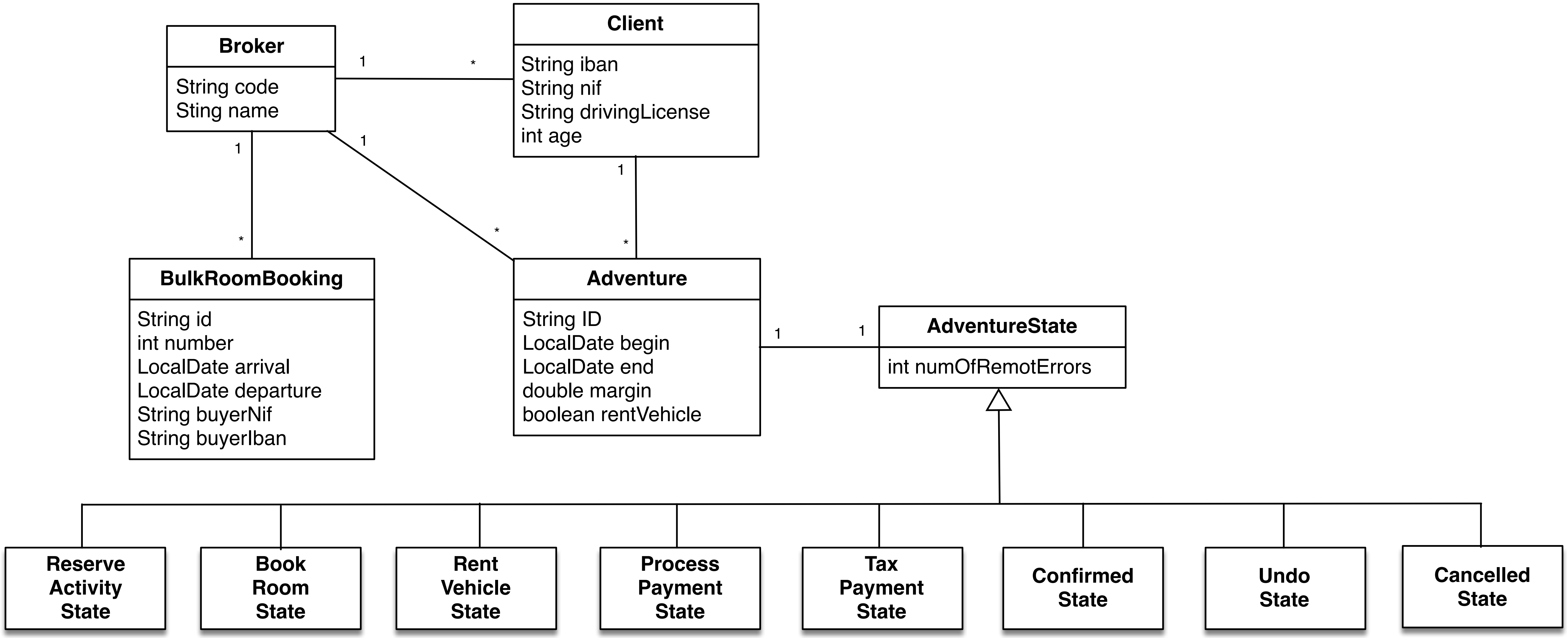
Test sequences of methods after testing methods

Example

Which sequences should be tested?

The life cycle
of a Adventure

Broker Module



Generate sequences

we need to define a state diagram

```
create;
```

Initial State

```
create;
process;
```

Activity Reserved

```
create;
process;
process;
```

Room Booked

```
create;
process;
process;
process;
```

Car Rented

```
create;
process;
process;
process;
process;
```

Payment Processed

…

AdventureSequenceTest.java

```java
@Test
public void successSequence(@Mocked final TaxInterface taxInterface, @Mocked final BankInterface
bankInterface,
        @Mocked final ActivityInterface activityInterface, @Mocked final HotelInterface
roomInterface,
        @Mocked final CarInterface carInterface) {
    new Expectations() {
        {
            ActivityInterface.reserveActivity((RestActivityBookingData) this.any);
            this.result = ACTIVITY_CONFIRMATION;

            HotelInterface.reserveRoom((RestRoomBookingData) this.any);
            this.result = ROOM_CONFIRMATION;

            CarInterface.rentCar((CarInterface.Type) this.any, this.anyString, this.anyString,
this.anyString,
                    (LocalDate) this.any, (LocalDate) this.any, this.anyString);
            this.result = RENTING_CONFIRMATION;

            BankInterface.processPayment((RestBankOperationData) this.any);
            this.result = PAYMENT_CONFIRMATION;

            TaxInterface.submitInvoice((RestInvoiceData) this.any);
            this.result = INVOICE_DATA;

            AdventureSequenceTest.this.activityReservationData.getPaymentReference();
            this.result = REFERENCE;

            AdventureSequenceTest.this.activityReservationData.getInvoiceReference();
            this.result = REFERENCE;

            AdventureSequenceTest.this.rentingData.getPaymentReference();
            this.result = REFERENCE;

            AdventureSequenceTest.this.rentingData.getInvoiceReference();
            this.result = REFERENCE;
```

AdventureSequenceSpockTest.groovy

```groovy
def 'success sequence'() {
    given: 'an adventure with rent vehicle as #car'
    def adventure = new Adventure(broker, ARRIVAL, end, client, MARGIN, hotel, car)
    and: 'an activity reservation'
    activityInterface.reserveActivity(_) >> bookingActivityData

    and: 'a room booking'
    if (hotel != Adventure.BookRoom.NONE) {
        hotelInterface.reserveRoom(_) >> bookingRoomData
    }
    and: 'a car renting'
    if (car != Adventure.RentVehicle.NONE) {
        carInterface.rentCar(*_) >> rentingData
    }

    and: 'a bank payment'
    bankInterface.processPayment(_) >> PAYMENT_CONFIRMATION
    and: 'a tax payment'
    taxInterface.submitInvoice(_) >> INVOICE_DATA
    and: 'the correct return of the data associated with each reservation and payment'
    activityInterface.getActivityReservationData(ACTIVITY_CONFIRMATION) >>
bookingActivityData
    if (car != Adventure.RentVehicle.NONE) {
        carInterface.getRentingData(RENTING_CONFIRMATION) >> rentingData
    }
    if (hotel != Adventure.BookRoom.NONE) {
        hotelInterface.getRoomBookingData(ROOM_CONFIRMATION) >> bookingRoomData
    }
    bankInterface.getOperationData(PAYMENT_CONFIRMATION)

    when: 'the life cycle of the adventure'
```