RMock user guide

Daniel Brolund <daniel.brolund(at)agical.com>

Joakim Ohlrogge <joakim.ohlrogge(at)agical.com>

Copyright © 2005-2007 Agical AB

Table of Contents

1. The RMock 2.0.0 user guide

1.1. RMock in short

RMock is a Java based framework that supports interaction based and state based testing through a variety of features. Both expectations and asserts use the same powerful matching framework, which provides a vast library of expressions ready-to-use, as well as the possibility to easily add your own custom ones.

For interaction based testing, RMock can create several different types of test-doubles to enable testing of one class at a time by "mocking" out its dependencies. The expectations on the test-doubles can be setup in a variety of ways, of which some are nested strict or loose ordering, multiplicity control, exception throwing and more.

To improve your productivity the error messages of RMock are very clear and explicit. All expectations are presented with their satisfaction state, and for more nasty problems some advice for your code is offered.

1.2. Getting started

It is assumed that you are somewhat familiar with the following:

-Java

-JUnit

-Test-driven development

-Mock objects

Download the following:

-The latest rmock-2.0.0.jar (http://sourceforge.net/projects/rmock/)

-The junit-3.8.1.jar (http://www.ibiblio.org/maven/junit/jars/junit-3.8.1.jar)

-The cglib-nodep-2.1_2.jar (http://www.ibiblio.org/maven/cglib/jars/cglib-nodep-2.1_2.jar)

-JDK1.3.1 or later (http://java.sun.com/downloads)

Install the JDK and put the downloaded jars on your development classpath

1.3. The basics

This section goes through the basics of RMock. It should give you enough information to be productive with RMock and to be able to interpret the different types of feedback RMock gives.

1.3.1. Assert that

In traditional state-based testing the *assertion* is essential. In the standard RMockTestCase assertXxxx from the TestCase of JUnit can be used, but RMock encourages you to use the **assertThat(...)** method instead. The reason for this is that it is more flexible, more extensible and more expressive in its error reports.

1.3.1.1. Using expressions with assertThat

The **assertThat(...)** can be used to make powerful assertions:

```
        assertThat( false, is.FALSE );
        assertThat( true, is.TRUE );
        assertThat( true, is.not( is.FALSE ) );
        assertThat( "Hello", is.eq( "Hello" ) );
        assertThat( "Hello", is.instanceOf( String.class ) );
```

```
        assertThat( "Hello", is.containing("ell") );
        assertThat( "Hello", is.endingWith("ello") );
        assertThat( "Hello", is.startingWith( "Hell" ) );
        assertThat( 666, is.gt( 665 ) );
        assertThat( 666, is.lt( 667 ) );
        assertThat( 666, is.eq( 666 ) );

        Object object = new Object();
        assertThat( object, is.same(object));
        assertThat( object, is.not( is.same(new Object())));
```

1.3.1.2. Assertion failed feedback

The simplest form of *feedback* that RMock provides is whether an assertion **fails** or **passes**.

In the example below we make an assertion than can't pass.

```
        assertThat(true, is.FALSE);
```

RMock reports this error condition by throwing this exception

```
ASSERTION FAILED!
<true>
does not pass the expression:
<isFalse(<null>)>
----------------------------------------------------------------
Still in setup state! (startVerification has not yet been called)
0 expectation(s) have not yet been matched
(indicated by '->' in the listing below)
----------------------------------------------------------------
        Unordered section:root {
             Unordered section:main {
             }
             Unordered section:defaults {
             }
        }
```

1.3.2. Expect exception to be thrown

Often when testing a class a certain exception should be thrown by that class. To avoid repetitive and boring try-catch blocks, RMock allows you to specify the expected exception using an expression.

1.3.2.1. How to use expectThatExceptionThrown

Here we create a **String**-array with two elements, but we try to access an element outside that array. The implementation should then throw an **ArrayIndexOutOfBoundsException**.

```
        String stringArray = new String {"a", "b"};
        Class exceptionClass = ArrayIndexOutOfBoundsException.class;
        expectThatExceptionThrown(is.instanceOf(exceptionClass));
        String string = stringArray666;
```

By just declaring what exception is expected, RMock handles the boring try-catch and provides you with a nice error message if the expected exception isn't thrown.

1.3.3. Expressions

Every RMockTestCase has a member variable named **is**. This variable is actually a **ConstraintFactory**, a helper to create common constraints to use in assertions, method call validations, and more.

A constraint must implement this interface:

```
interface Constraint :
 com.agical.rmock.core.match.Expression
{
    getName(  ) : String
    getReference(  ) : Reference
}
```

As seen above all constraints are *expressions*. The expression definition is shown below

```
interface Expression :
{
    and( Expression ) : Expression
    or( Expression ) : Expression
    xor( Expression ) : Expression
    passes( Object ) : boolean
    describeWith( ExpressionDescriber ) : void
}
```

When using constraints you typically create them using the standard constraint factory provided by RMock:

```
interface ConstraintFactory :
{
    lt( double ) : Expression
    lt( byte ) : Expression
    lt( short ) : Expression
    lt( int ) : Expression
    lt( long ) : Expression
    lt( Object ) : Expression
    lt( char ) : Expression
    lt( float ) : Expression
    eq( double ) : Expression
    eq( float ) : Expression
    eq( char ) : Expression
    eq( long ) : Expression
    eq( boolean ) : Expression
    eq( Object ) : Expression
    eq( short ) : Expression
    eq( int ) : Expression
    eq( byte ) : Expression
    not( Expression ) : Expression
    le( short ) : Expression
    le( int ) : Expression
    le( char ) : Expression
    le( long ) : Expression
    le( byte ) : Expression
    le( double ) : Expression
    le( float ) : Expression
    le( Object ) : Expression
    ge( float ) : Expression
    ge( double ) : Expression
    ge( byte ) : Expression
    ge( short ) : Expression
    ge( long ) : Expression
```

```
    ge( Object ) : Expression
    ge( int ) : Expression
    ge( char ) : Expression
    containing( String ) : Expression
    same( Object ) : Expression
    instanceOf( Class ) : Expression
    gt( short ) : Expression
    gt( float ) : Expression
    gt( long ) : Expression
    gt( Object ) : Expression
    gt( byte ) : Expression
    gt( int ) : Expression
    gt( double ) : Expression
    gt( char ) : Expression
    startingWith( String ) : Expression
    endingWith( String ) : Expression
}
```

1.3.3.1. Using the constraint factory of RMockTestCase

Shown here is a disection of the assertThat used in conjunction with the constraint factory.

```
        assertThat( true, is.eq(true) );
        // is the equivalent of
        assertThat( true, is.TRUE );
        // is the equivalent of
        ConstraintFactory constraintFactory = is;
        assertThat( true, constraintFactory.TRUE );
        // is the equivalent of
        Expression expression = constraintFactory.TRUE;
        assertThat( true, expression );
```

1.3.4. Mocking and verifying

An important component of any intercation based framework is the ability to create *mocks*. A mock is different from a stub in that a mock can verify that what was expected to happen actually happened.

The mock test-double is very strict; it verifies that whatever messages are sent to the mock are expected and also that **all** expected messages are actually received before the test is finished.

In RMock you create mocks with one of the *mock(...)* method.

1.3.4.1. The simplest thing that could possibly be mocked

Every test starts in the *set up* state. In the set up state you set up *expectations*. Expectations are a specification of what you expect to happen to a mock as a result of running the test. Lets begin by creating a mock and set up some expectations on it:

```
        Runnable runnable = (Runnable)mock(Runnable.class);
```

Create a new mock for the interface (Runnable). The mock object is an instance of the Runnable interface (created behind the scenes in runtime by cglib, but that is another story). We only use the interface for setting up the methods expected to be invoked on it. This way we get a strong typing of the methods we set up, and we also get refactoring support, e.g. if we change the name of a method, the set up also changes.

Set up a method that the mock will require being called when in verification mode. This way we say: "**Runnable** expects one and only one call to the method run() with no parameters":

```

```

```
        runnable.run();

        startVerification();
```

The last statement *startVerification* changes the state of all mocks from *recording* to verifying. In an RMock test everything above *startVerification* is *set up code* meaning that the code sets up expectations on the mocks. We finish the test by fulfilling the expectations. In a real world test the runnable would typically be passed to the actual object we want to test as a constructor agument, via a setter or as a parameter to some method.

```
        runnable.run();
```

As we make the invocation RMock verifies that there is a matching expectation for the call. Had there been no expectation RMock would have failed the test instantly.

After the test method has finished, all used mock objects are automatically verified according to what was set up on them. Since the runnable object was set-up to expect one and only one call to run(), the verification in this test case will pass.

1.3.4.2. Setting up expectations

When a mock is created it is in the *set-up* state. Or rather, you can only create mocks when RMock is in the set-up state. RMock is in the set-up state in the beginning of each test-method.

You can set up an expectation on the mock simply by calling the method you expect to be invoked on the mock:

```
        Runnable runnable = (Runnable)mock(Runnable.class);
        runnable.run();
```

RMock sees the above expectation like this

```
        Unordered section:root {
           Unordered section:main {
->            0(1) runnable.run()
           }
           Unordered section:defaults {
           }
        }
```

That means that one call to run is expected and currently zero invocations have been detected.

We change the state from *set up* to *verify* by calling RMocks startVerification method:

```
        startVerification();
        runnable.run();
```

and RMock records that one invocation has been made:

```
        Unordered section:root {
           Unordered section:main {
              1(1) runnable.run()
           }
           Unordered section:defaults {
           }
```

```
        }
```

Which satisfies all expectations in this test.

1.3.4.3. Naming mocks

As seen in the previous example RMock assigns a name to the mock automatically. Sometimes it can be useful to assign a different name to a mock though.

RMock uses the name to identify the mock so it has to be unique. Don't worry, RMock detects and fails the test if an id is reused.

The example below shows how two mocks are created, one with an autogenerated name and one that is explicitly assigned a name upon creation:

```
        Runnable runnable = (Runnable)mock(Runnable.class);
        Runnable someOtherRunnable = (Runnable)mock(Runnable.class,
        runnable.run();
        someOtherRunnable.run();
```

This is what RMock sees:

```
        Unordered section:root {
           Unordered section:main {
->            0(1) runnable.run()
->            0(1) someOtherRunnable.run()
           }
           Unordered section:defaults {
           }
        }
```

We fulfill the expected invocation on *someOtherRunnable*

```
        startVerification();
        someOtherRunnable.run();
```

and RMock records that *someOtherRunnable* has all invocations fulfilled while *runnable* is still unsatisfied:

```
        Unordered section:root {
           Unordered section:main {
->            0(1) runnable.run()
              1(1) someOtherRunnable.run()
           }
           Unordered section:defaults {
           }
        }
```

1.3.4.4. Unexpected invocation failure

When a mock receives a message that is not setup to be received that is called an *unexpected* invocation.

In the example below we create a mock without expectations and invoke it unexpectedly in the *verify* state.

```
```

```
        List mockedList = (List)mock(List.class, "mockedList");

        startVerification();

        mockedList.add("unexpected string");
```

RMock reports this error condition by throwing this exception

```
UNEXPECTED!
No expectation matched: mockedList.add(<unexpected string>)
----------------------------------------------------------------
0 expectation(s) have not yet been matched
(indicated by '->' in the listing below)
----------------------------------------------------------------
        Unordered section:root {
             Unordered section:main {
             }
             Unordered section:defaults {
             }
        }
```

Such mocks are typically passed to or *injected* into other objects in order to verify that they are invoked according to your expectations.

1.3.4.5. Unsatisfied expectation failure

When expectations are not fulfilled for the duration of the test RMock reports an *unsatisfied* error.

In the example below we create a mock and record an expectation that we won't fulfill in the *verify* state.

```
        List mockedList = (List)mock(List.class, "mockedList");

        mockedList.add("expected string");
        startVerification();
```

RMock reports this error condition by throwing this exception

```
UNSATISFIED!
----------------------------------------------------------------
1 expectation(s) have not yet been matched
(indicated by '->' in the listing below)
----------------------------------------------------------------
        Unordered section:root {
             Unordered section:main {
->                0(1) mockedList.add(eq(<expected string>))
             }
             Unordered section:defaults {
             }
        }
```

As you see, RMock makes sure that everything you expect actually happens.

1.3.4.6. Modify the expectation multiplicity

Create a new mock for the interface (Runnable) and set up the run-method as before. Lets assume we expect the run() method to be called several times but not a particular number of times. This could be illustrated with a set up call to run() for the number of times expected, OR we can use the modify functionality of RMock:

```
        Runnable runnable = (Runnable)mock(Runnable.class);
        runnable.run();
        modify().multiplicity(expect.atLeastOnce());
```

Now the runnable expects at least one call to the run() method. **expect** is a multiplicity expectation factory available on **com.agical.rmock.extension.junit.RMockTestCase**

Invoke method run() one or several times since that is what is expected.

```
        runnable.run();
        runnable.run();
        runnable.run();
        runnable.run();
        runnable.run();
        runnable.run();
        runnable.run();
```

1.3.4.7. Expect multiplicity range

The expectation multiplicity can be modified in several ways. The **expect** member of RMockTest case is used to change the expected multiplicity of an expectation. The member is an instance of

```
com.agical.rmock.core.match.multiplicity.MultiplicityFactory
```

and looks like this:

```
interface MultiplicityFactory :
{
    from( int ) : MultiplicityFactory$LimitableMultiplicity
    once(  ) : Multiplicity
    atLeastOnce(  ) : Multiplicity
    atLeast( int ) : Multiplicity
    exactly( int ) : Multiplicity
    atMost( int ) : Multiplicity
    atMostOnce(  ) : Multiplicity
}
```

Besides the more obvious usages of the **expect** member you can define a multiplicity range like this:

```
        Runnable runnable = (Runnable)mock(Runnable.class);
        runnable.run();
        modify().multiplicity(expect.from(2).to(5));
```

Now the runnable expects 2, 3, 4, or 5 calls to the run() method.

To make the expectation pass we need to invoke the method at least twice and at most five times:

```
        runnable.run();
        runnable.run();
        runnable.run();
        runnable.run();
```

1.3.4.8. Modifying the arguments of the expectation

Consider the following class;

```
package com.agical.rmock.doc.basics;

public interface MethodInterface {
    void oneArgument( String string );
    void twoArguments( String string, Object object );
}
```

To be able to mock calls where the actual argument is hard to setup exactly, or maybe it is just irrelevant, RMock provides several ways of modifying the expected arguments. The most allowing way is of course to allow anything:

```
        MethodInterface mi = (MethodInterface)mock(MethodInterface.class);
        mi.oneArgument("Cannot set this up exactly");
        modify().args(is.ANYTHING);
```

As you see, we use **is**, the same expression factory as in the assertThat() method, to modify the argument expectations.

To fulfill the expectation we can pass anything that Java can compile to the method:

```
        mi.oneArgument("Any String or null");
```

1.3.4.9. How to modify only some arguments

Consider the same class as previously. If you want to change one expectation but keep the other you can do like this:

```
        MethodInterface mi = (MethodInterface)mock(MethodInterface.class);
        Object importantObj = new Object();
        mi.twoArguments("Cannot setup this exactly", importantObj);
        modify().args(is.ANYTHING, is.AS_RECORDED);
```

To fulfill the expectation we can pass anything as the first argument, but the second needs to be as recorded:

```
        mi.twoArguments("Any String or null here", importantObj);
```

The setup above would be the equivalent of this:

```
        mi.twoArguments("Cannot setup this exactly either", importantObj);
        modify().args(is.ANYTHING);
```

i.e. *you can omit modifications in the end of the argument list if they are supposed to be as recorded*. This setup has exactly the same expectations on the passed arguments:

```
        mi.twoArguments("Any String or null here, too", importantObj);
```

1.3.5. How to make a mock throw an exception

Mocks and intercepts can be configured to throw an exception to enable testing exception handling in the class under test.

1.3.5.1. Throw an exception from a method without parameters

Consider the following source

```
    Runnable runnable = (Runnable)mock(Runnable.class, "runnable");
    RuntimeException runtimeException =
        new RuntimeException( "My faked exception" );
    runnable.run();
    modify().throwException( runtimeException );
    startVerification();

    try {
        runnable.run();
    } catch (RuntimeException actualException) {
        assertThat(actualException, is.same(runtimeException));
    }
```

Here we create a mocked Runnable and set it up to throw an exception when called. When the method is invoked, instead of just returning, it throws the specified exception. In this example we catch it using using a standard catch-clause, but we could just as well have used the **expectThatExceptionThrown(is.instanceOf(RuntimeException.class));**

1.3.6. Sections

Sections can be used to ensure the call order among mocks, e.g. you might want the validation to be made before committing data.

Most of the time you don't need sections at all but sometimes you want to specify that all or a subset of events occur in a specific order.

You may also want to group expectations in sections in order to find the expectations you look for more easily.

RMock also allows you to append expectation to a previously defined section, which enables you to easily avoid repetitive setup code when all you want is a small variation.

In this chapter we will show some examples of how to use sections and how they affect the way expectations are matched.

1.3.6.1. The default sections

Sections is a large topic but even if you are not consiously using sections you are using them as soon as you specify an expectation.

By default RMock specifes a few sections for you, and they look like this in e.g. error messages:

```
        Unordered section:root {
           Unordered section:main {
           }
           Unordered section:defaults {
           }
        }
```

You see the *main* and *defaults* sections that are contained in a *root* section:

As you can see in the message all of these sections are *unordered* which means that expectations can match in any order from those sections. We talk more about the *defaults* section later since it's usage is fairly advanced. The next few sections we will focus on the *main* section.

1.3.6.2. Defining an ordered section

Sections are created by invoking one of the methods on the section factory:

```
interface SectionFactory :
{
    unordered( String ) : Section
    ordered( String ) : Section
}
```

as an argument to the beginSection()/endSection() methods of the RMockTestCase. Expectations set up within an ordered section must be invoked in that order.

```
        MethodInterface mi =
            (MethodInterface)mock(MethodInterface.class, "mi");
        beginSection(s.ordered("ordered section"));
        {
            mi.oneArgument("First");
            mi.twoArguments("Second", "Any object");
        }
        endSection();
```

RMock first sees the expectations like this:

```
        Unordered section:root {
           Unordered section:main {
              Ordered section:ordered section {
->               0(1) mi.oneArgument(eq(<First>))
-> (N/A)         0(1) mi.twoArguments(eq(<Second>), eq(<Any object>))
              }
           }
           Unordered section:defaults {
           }
        }
```

The "arrow" indicates that an expectation is not fulfilled. (N/A) indicates that the ordered section need other expectations to be fulfilled before this expectation can be accepted.

After the first method is invoked the expectations change to this state:

```
        Unordered section:root {
           Unordered section:main {
              Ordered section:ordered section {
                 1(1) mi.oneArgument(eq(<First>))
->               0(1) mi.twoArguments(eq(<Second>), eq(<Any object>))
              }
           }
           Unordered section:defaults {
           }
        }
```

Now there is only one expectation left to fulfill, and after that is done it will look like this:

```
        Unordered section:root {
```

```
           Unordered section:main {
              Ordered section:ordered section {
                 1(1) mi.oneArgument(eq(<First>))
                 1(1) mi.twoArguments(eq(<Second>), eq(<Any object>))
              }
           }
           Unordered section:defaults {
           }
        }
```

No arrows indicate that RMock expects no more calls to its expectations.

Ordered and unordered sections can be nested to any level, but generally there is a great chance that you have design problems in your code if you need to use that ability.

1.3.6.3. Using the defaults section

The default section can be used for occasions when you want to setup general expectations that are used "if all else fails", usually to cover for small variations in general scenarios. They are appended last in the expectation list, even if they are set up first. Usually you allow these setups to be called any number of times, including zero times.

To use this feature you have to *append* to the *defaults* section, like this:

```
        MethodInterface mi =
            (MethodInterface)mock(MethodInterface.class, "mi");
        appendToSection("defaults");
        {
            mi.twoArguments("Not so important variation", ", dude!");
            modify().multiplicity(
                    expect.from(0)).args(is.ANYTHING, is.ANYTHING);
        }
        endSection();
        mi.oneArgument("Important call");
```

RMock will see the set up like this:

```
        Unordered section:root {
           Unordered section:main {
->            0(1) mi.oneArgument(eq(<Important call>))
           }
           Unordered section:defaults {
              0(*) mi.twoArguments(anything(<null>), anything(<null>))
           }
        }
```

The somewhat constructed implementation might look like this:

```
        mi.oneArgument("Important call");
        if( System.currentTimeMillis()%2 == 1 ) {
            mi.twoArguments("Yada", "Yada");
        }
```

In a more realistic testing scenario the defaults section would be configured in the setUp() method, and the tests can focus on clarifying the intentions for the code.

And after the run the expectations look like this.

```
        Unordered section:root {
```

```
            Unordered section:main {
               1(1) mi.oneArgument(eq(<Important call>))
            }
            Unordered section:defaults {
               0(*) mi.twoArguments(anything(<null>), anything(<null>))
            }
         }
```

Has the defaults section been called? It depends on the system clock. Since RMocks documentation is actually generated from live data in the test cases, it will vary from time to time.

1.4. The different types of test-doubles

This sections contains more detailed RMock features for creating test-doubles of different kinds.

1.4.1. Fake and intercept

1.4.1.1. Default return values

We start demonstrating fakeAndIntercept by faking this interface

```
interface DemoReturnTypesInterface :
{
    getBoolean(  ) : boolean
    getByte(  ) : byte
    getShort(  ) : short
    getChar(  ) : char
    getInt(  ) : int
    getLong(  ) : long
    getFloat(  ) : float
    getDouble(  ) : double
}
```

We start the verification before using the demoInterface variable. We expect the interface to return default-values for the different return types.

```
         demoInterface = (DemoReturnTypesInterface)fakeAndIntercept(
                     DemoReturnTypesInterface.class, "demoInterface");
         startVerification();

         assertThat(demoInterface.getBoolean(), is.eq(false));
         assertThat(demoInterface.getByte(), is.eq((byte)0));
         assertThat(demoInterface.getShort(), is.eq((short)0));
         assertThat(demoInterface.getChar(), is.eq((char)0));
         assertThat(demoInterface.getInt(), is.eq(0));
         assertThat(demoInterface.getLong(), is.eq((long)0));
         assertThat(demoInterface.getFloat(), is.eq((float)0));
         assertThat(demoInterface.getDouble(), is.eq((double)0));
```

As can be seen we don't have to setup any expectations in order to use the faked interface.

1.4.1.2. Interception

When invocations are made on a faked and intercepted interface before startVerification is called an interception is

made.

After startVerification is called interceptions work as expectations on mocks. That means that from then on, for intercepted methods, only the explicitly expected invocations are allowed. All methods that are not intercepted still return default values as demonstrated earlier.

```
        demoInterface = (DemoReturnTypesInterface)fakeAndIntercept(
                    DemoReturnTypesInterface.class, "demoInterface");
        // intercept getLong()
        demoInterface.getLong();
        modify().multiplicity(expect.exactly(2)).returnValue(4711L);
        startVerification();

        assertThat(demoInterface.getLong(), is.eq((long)4711));
        assertThat(demoInterface.getLong(), is.eq((long)4711));
        // getInt is not intercepted and will return a default value
        assertThat(demoInterface.getInt(), is.eq(0));
```

As seen in the code listing we can modify the interception. In this case we modied getLong() to return 4711 two times.

1.4.2. Test doubles for classes

The first rule of test-doubles for concrete classes is: *don't create test-doubles for concrete classes*!

That said, RMock allows it and in this section we will explain how and some special features and tricks related to test-doubles for classes.

There are two basic ways to create test-doubles for classes:

1. mocking - The same as mocking an interface except that instead of dynamically implementing an interface RMock *subclasses* the class you want to mock. This means that **final** classes can't be mocked and that the constructor of the mocked class will need to be run. A fact that can and will eventually bite you.

Once the mock is created it works the same as if it would be a mocked interface with the limitation that **final** methods cannot be mocked.

2. intercepting - Intercepting a class allows you to setup expectations on some methods of a class. All other methods work as if the class would have been created as usual with new. Intercepting also works by subclassing so the same limitations as for mocking classes apply when intercepting them.

A typical example when intercepting can be used is to intercept a stream in order to guarantee that close is called while the rest of the methods function normally.

1.4.2.1. Mocking a class with a default constructor

We mock an instance of java.util.ArrayList and expect clear to be called once

```
        List mockedArrayList = (List)mock(ArrayList.class);
        mockedArrayList.clear();
```

As with interfaces an expectation is recorded

```
        Unordered section:root {
           Unordered section:main {
->            0(1) arrayList.clear()
           }
           Unordered section:defaults {
```

```
        }
    }
```

1.4.2.2. Intercepting classes with a default constructor

We intercept an instance of java.util.ArrayList and expect clear to be called once

```
    List interceptedArrayList = (List) intercept(ArrayList.class);
    interceptedArrayList.clear();
```

As with interfaces an expectation is recorded

```
    Unordered section:root {
       Unordered section:main {
->        0(1) arrayList.clear()
       }
       Unordered section:defaults {
       }
    }
```

We fulfil the clear expectation and we add objects to the list and check it's size.

```
    startVerification();
    assertThat(interceptedArrayList.size(), is.eq(0));
    interceptedArrayList.add("hello");
    interceptedArrayList.add("world!");
    assertThat(interceptedArrayList.size(), is.eq(2));
    // up until here the list works as expected
    interceptedArrayList.clear();
    // Since clear is intercepted clear is not
    // forwarded to the implementation,
    // hence the list is not cleared.
    assertThat(interceptedArrayList.size(), is.eq(2));
```

Note that since clear is intercepted the call is never forwarded to the list-implementation so the list is not cleared!

1.4.2.3. Forwarding invocations to the implementation

We use the same setup as in the previous section with one important addition: This time we modify the action of the clear() invocation to forward to the underlying implementation.

```
    List interceptedArrayList = (List) intercept(ArrayList.class);
    interceptedArrayList.clear();
    modify().forward();
```

```
    Unordered section:root {
       Unordered section:main {
->        0(1) arrayList.clear()
       }
       Unordered section:defaults {
       }
    }
```

We put data in the list as before:

```
        startVerification();
        assertThat(interceptedArrayList.size(), is.eq(0));
        interceptedArrayList.add("hello");
        interceptedArrayList.add("world!");
        assertThat(interceptedArrayList.size(), is.eq(2));
        interceptedArrayList.clear();
        // Since clear the clear invocation is forwarded
        // to the underlying implementation
        // the list size should be 0 after clear
        assertThat(interceptedArrayList.size(), is.eq(0));
```

This time the intercepted call is forwarded to the list-implementation and the list is cleared.

And the invocation to clear has been registered:

```
        Unordered section:root {
           Unordered section:main {
              1(1) arrayList.clear()
           }
           Unordered section:defaults {
           }
        }
```

1.4.2.4. Mocking classes with different non default constructors

When mocking, intercepting or fake-and-intercepting a class that has no default constructor, RMock needs to know what arguments to provide to the constructor, and sometimes even the constructor signature to use (more on that later).

Consider the following class:

```
package com.agical.rmock.doc.features;

public class DifferentConstructors {
    String arg1 = "";
    Object arg2 = "";

    public DifferentConstructors(String arg1) {
        this.arg1 = arg1;
    }

    public DifferentConstructors(String arg1, String arg2) {
        this.arg1 = arg1;
        this.arg2 = arg2;
    }
    public String getString() {
        return arg1 + arg2;
    }
}
```

If we want to mock it we need to provide it with some arguments, like this:

```
        DifferentConstructors differentConstructors =
            (DifferentConstructors)mock(DifferentConstructors.class,
                                        new Object {"Only first arg" },
                                        "classWithDifferentConstructors");
        differentConstructors.getString();
        modify().forward();
        startVerification();
```

```
        assertThat( differentConstructors.getString(),
                    is.eq("Only first arg") );
```

RMock analyses the argument provided in the Object-array and tries to match a constructor. In this case it is easy since there is only one constructor taking one argument.

1.4.2.5. Mocking classes with overloaded constructors

Now consider the following class:

```
package com.agical.rmock.doc.features;

public class OverloadedConstructors {
    Object arg = "";

    public OverloadedConstructors(Object arg) {
        this.arg = "As object: " + arg;
    }

    public OverloadedConstructors(String arg) {
        this.arg = arg;
    }

    public Object getArg() {
        return arg;
    }
}
```

RMock tries to guess what constructor is intended to use by matching the types of the arguments provided, and for most cases this is enough, like in this case where we want to use the String constructor:

```
        OverloadedConstructors overloadedConstructors1 =
            (OverloadedConstructors)mock(OverloadedConstructors.class,
                                         new Object {"String arg"},
                                         "classWithOverloadedConstructors1");
```

When the constructors are overloaded and the default selection is not what is desired, the construtor has to be pointed out explicitly. This is done by also providing the constructor signature as a Class array:

```
        OverloadedConstructors overloadedConstructors2 =
            (OverloadedConstructors)mock(OverloadedConstructors.class,
                                         new Class {Object.class},
                                         new Object {"Wants to be an object"},
                                         "classWithOverloadedConstructors2");
```

Note that this also works for primitives, for instance **int.class**, **double.class** etc. In the end we can verify that the intended constructors actually was called:

```
        overloadedConstructors1.getArg();
        modify().forward();
        overloadedConstructors2.getArg();
        modify().forward();
        startVerification();
        assertThat( overloadedConstructors2.getArg(),
                    is.eq("As object: Wants to be an object") );
        assertThat( overloadedConstructors1.getArg(),
                    is.eq("String arg") );
```

Generally, if you need to use these really detailed and advanced features on your own code, you should probably go

over your design again.

A. Appendix

A.1. JUnit

RMock is based on the testing framework JUnit, http://www.junit.org. Currently version 3.8.1 is supported.