Multi-thread tests with Lmock

Summary

Lmock provides a support to validate applications implying multiple threads.

Such threads can:

- Either be created by the tests
- Or be spawned by objects indirectly invoked by the tests

In both cases, Lmock introduces a number of concepts and methods that simplify the tracking of those threads and allow to inject mocked invocations.

Like the "traditional" mocking framework offered by Lmock, you can use either some "classical" programming paradigms to design multi-threaded tests or rely on a more friendly form, using masquerades.

Main concepts

The multi-thread support in Lmock is pretty rich, thus a little bit more complex to put in place than simple – mono-threaded – tests. It introduces a number of ideas presented here-in.

Testing thread vs. threads under test

The first main idea is that a test always include a testing thread and some threads under test.

The testing thread is typically the one launched by your testing framework, running the test method.

Typically, this thread (and **no other thread than this one**) will:

- create the mock objects,
- define the scenario and the stubs
- begin and end the test

The threads under test are the threads that actually implement the tested objects. They may:

- be the testing thread itself
- be created by the testing thread
- be indirectly created by the testing thread or other threads under test, as a consequence of the test execution

Actors

The basic principle of Lmock is to identify the threads under test on the fly: whenever a thread invokes a mock, Lmock checks if it knows it. If not, it tries to associate it to a scenario and a set of stubs (defined by the testing thread). Such a set is denoted as an "**actor** of the test".

Quite obviously, if a thread under test never invokes any mock object, it cannot be fetched, thus does not contribute to the test from Lmock standpoint.

The association of a thread with its corresponding actor is helped by specific **thread checkers** defined

by the user. Lmock comes with a number of default checkers, defined by `com.vmware.lmock.checker.ThreadChecker`:

- `equalTo(thread)`: the thread is explicitly specified by the user
- `threadsCalled(name)`: the corresponding thread should have the specified `name`.
- `instancesOf(class)`: the class of the corresponding thread is given by `class`.
- `anyThread`: any thread will match

Of course, you may also define your own checker if ever none of the above helper suits your expectations.

The definition of an actor is achieved by specific methods given by `com.vmware.lmock.mt.Actor`:

- `anActorForThread(thread)`: explicitly tell the `thread` to associate to the actor under construction
- `anActorForThreadLike(checker)`: the actor will be associated to the very first thread that:
  - Invokes the method of a mock
  - Whose profile matches the specified checker

Also, Lmock defines `anActorForCurrentThread()`, which is expected to represent the testing thread.

In fact, this actor is managed by default by Lmock and **should not** be explicitly used in the tests.

Scenarios and tests

Once you have defined the actors contributing to the test, you must associate them to scenarios and stubs.

Lmock is quite flexible for this, in the sense that:

- Several actors may share the same scenario, or (non exclusive) use different scenarios (which are considered as being played in parallel)
- Several actors may rely on the same set of stubs, or use their own stubs

Any combination is possible, in fact... The multi-thread support in Lmock was thought with different use-cases in mind:

- Your threads are independent from each other, but you know what each of them exactly does: each thread/actor has its own scenario
- Your threads are inter-dependent... The result is that their execution leads to a predictable sequence of invocations: create a single scenario associated to every thread/actor

In both cases, the actors may use a common set of stubs or rely on their own set if they have some very specific behaviors.

With the “classical” framework you can easily express such associations thanks to two methods:

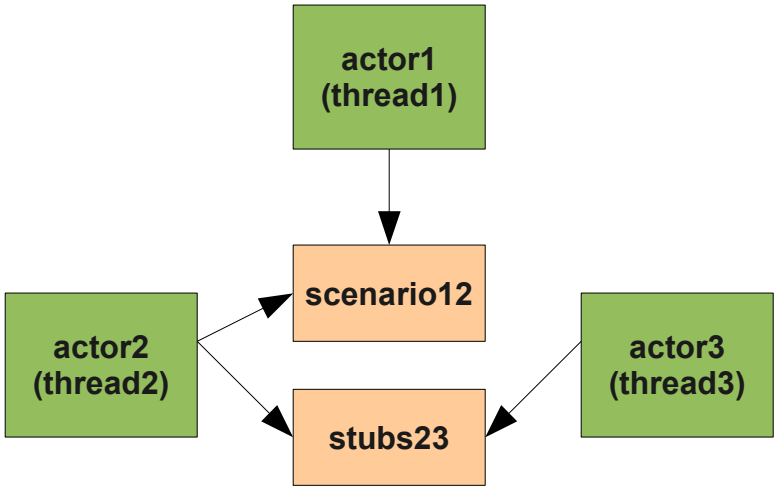
- `actor.following(scenario)`: your `actor` is associated to the given `scenario`
- `actor.using(stubs)`: your `actor` relies on the given `stubs` to operate

Creating a story with multiple actors

When the scenarios, stubs and actors are properly created, the testing thread can start the story, using `Story.create`. The only difference with the basic `Story.create` method is that the method receives the list of actors implied in the story instead of the scenario/stubs.

The following example gives an illustration of a typical sequence to create a test implying multiple threads. More specifically, this test defines:

- One scenario (scenario12), to which two actors (actor1 and actor2) contribute
- One set of stubs (stubs23), referenced by two actors (actor2 and actor3)



```
import com.vmware.lmock.impl.Scenario;
import com.vmware.lmock.mt.Actor;
import static com.vmware.lmock.mt.Actor.*;
import com.vmware.lmock.impl.Story;
import com.vmware.lmock.impl.Stubs;
import org.junit.Test;
public class SimpleMTExample {
    @Test
    public void testWithMultipleThread() {
        // TODO DEFINE ACTUAL THREADS HERE
        Thread thread1 = new Thread();
        Thread thread2 = new Thread();
        Thread thread3 = new Thread();

        Scenario scenario12 = new Scenario() {
            {
                // TODO FILL IN THE SCENARIO HERE
            }
        };
        Stubs stubs23 = new Stubs() {
            {
                // TODO FILL IN THE STUBS HERE
            }
        };
        Actor actor1 = anActorForThread(thread1).following(scenario12);
        Actor actor2 = anActorForThread(thread2).
            following(scenario12).using(stubs23);
        Actor actor3 = anActorForThread(thread3).using(stubs23);
        Story story = Story.create(actor1, actor2, actor3);
        story.begin();
        // TODO IMPLEMENT THE TEST
        story.end();
    }
}
```

Checking test results and errors

To understand how Lmock manages multiple threads, you must first understand the basic issue raised

by this type of test.

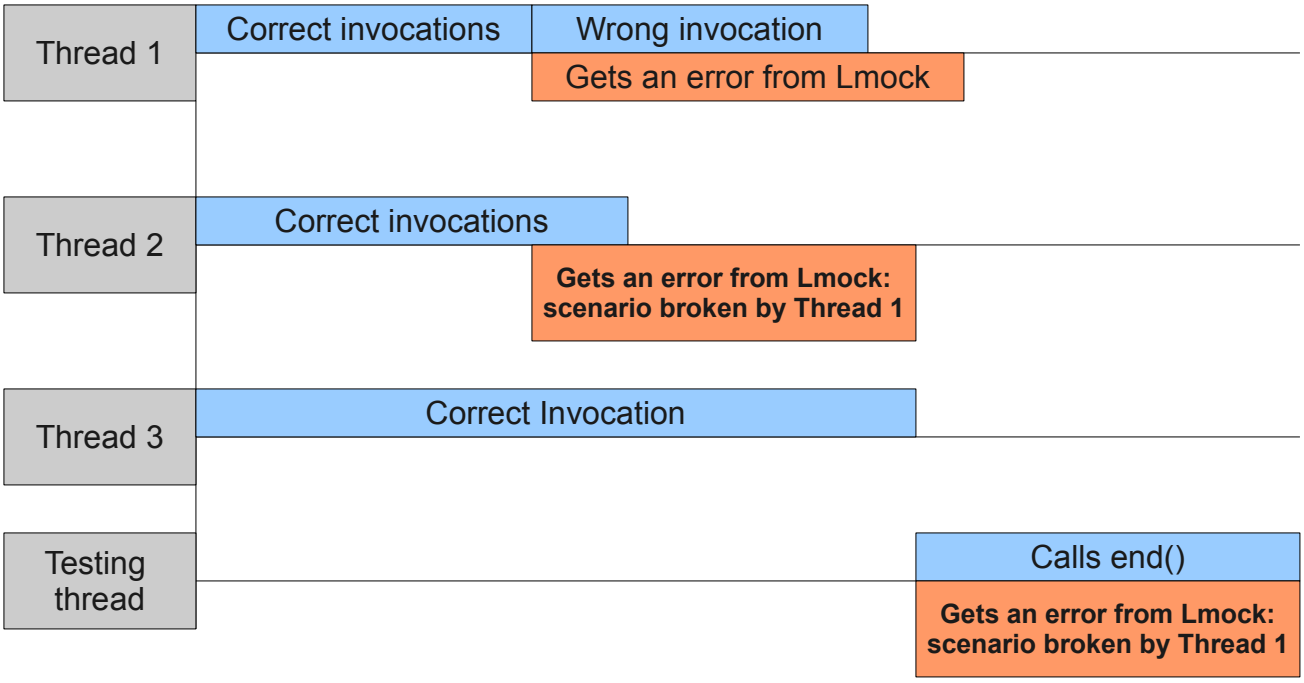
Assume a test including:

- The testing thread
- A thread under test, that fails for whatever reason

When the thread under test fails, this failure is reported by an exception or an error. The issue is that at this level, the thrown error is never trapped by the testing thread and so is not properly reported.

Lmock solves that issue by “guarding” the errors reported by any thread under test and forwarding that error when the test completes (i.e. upon invocation of the `end()` method). Moreover, the thread may contribute to a scenario shared with other threads. In that case, the other threads will also be interrupted whenever they invoke a mock.

Next is an illustration of the guarding mechanism:



Thread 1 and Thread 2 share the same scenario.
Thread 3 complies with a different scenario

In this example, all the threads first perform valid invocations to mock objects. When *Thread 1* makes a wrong invocation (i.e. not complying with the scenario) it immediately gets an error from Lmock. *Thread 2* will then get a similar error when making the next invocation to a mock. *Thread 3* doesn't share its scenario with Thread 1, thus doesn't see any error at all. Finally, the error caused by Thread 1 is reported to the user when the testing thread invokes `end()`.

Masquerades and roles

The mutli-thread support of Lmock extends to masquerades. For mono-threaded tests, the developer of the test is requested to specify “what is going to happen” (e.g. `willInvoke(1).of(...)...`). When

dealing with multiple threads, the user is also given the opportunity to say “for who it is going to happen” (i.e. `john.willInvoke(1).of(...)...`). This new feature introduces the concept of **role**.

A role is basically associated to one or several actors (thus making the link with the threads under test), referring to a common scenario and set of stubs built by the role upon new directives.

For example, assume that `actor1` and `actor2` are associated to the same role `commonRole`. Then, by saying:

```
commonRole.willInvoke(1).of(foo).bar();
commonRole.willReturn(true).when(foo).equals(anotherFoo);
commonRole.willInvoke(2).of(anotherFoo).bar();
```

The result is that actor1 and actor2 share a common scenario;

Invocation of `foo.bar()` must occur once

Invocation of `anotherFoo.bar()` must occur twice

And use a stub saying that `foo` equals `anotherFoo`.

The fact of constructing a specific pool of expectations and stubs for a each actor is a default behavior of the masquerades. You may also wish to share either a scenario or a set of stubs, or both, between several actors. This is achieved by saying `shareStubsWith` and `shareScenarioWith`.

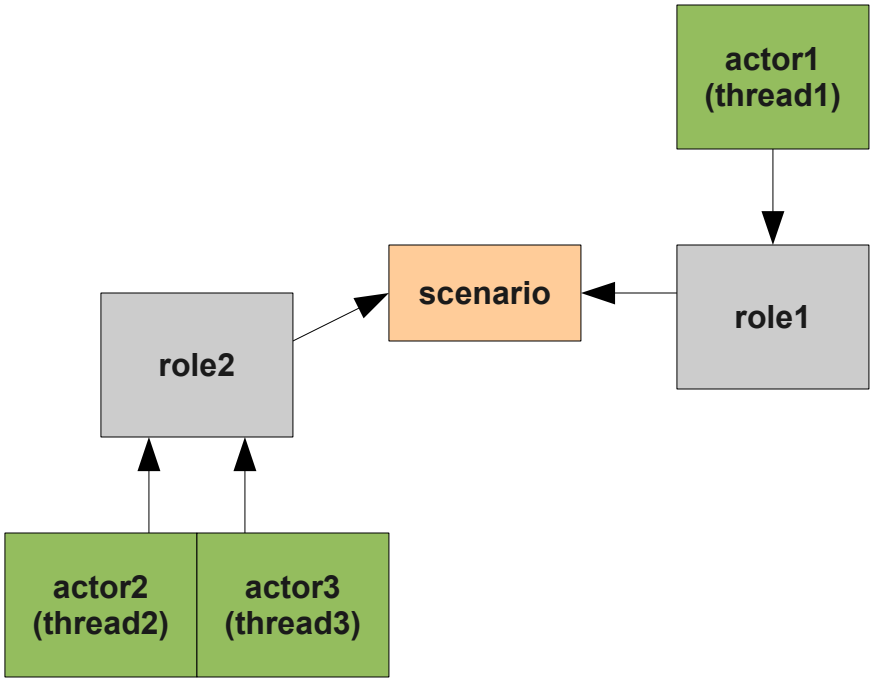
For example, let's consider the following combination:

- A first actor, actor1, is sharing its scenario with two other actors (actor2 and actor3)
- The first actor is using its own set of stubs
- The two other actors share their stubs

This can be implemented by:

- Specifying one role (role1) for actor1
- Specifying one role (role2) for actor2 and actor3
- Let role1 and role2 share their scenario

As illustrated in the next figure:



In terms of code, this can be achieved as follows:

```
import com.vmware.lmock.masquerade.Role;
import static com.vmware.lmock.masquerade.Schemer.*;
import com.vmware.lmock.mt.Actor;
import static com.vmware.lmock.mt.Actor.*;
import org.junit.Test;
public class SimpleMTExampleWithMasquerade {
    @Test
    public void testWithMultipleThread() {
        // TODO DEFINE ACTUAL THREADS HERE
        Thread thread1 = new Thread();
        Thread thread2 = new Thread();
        Thread thread3 = new Thread();

        Actor actor1 = anActorForThread(thread1);
        Actor actor2 = anActorForThread(thread2);
        Actor actor3 = anActorForThread(thread3);

        Role role1 = new Role(actor1);
        Role role2 = new Role(actor2, actor3);
        role2.shareScenarioWith(role1);

        begin(role1, role2);
        // TODO: write the directives, using roleX.will...
        // AND RUN THE TEST.
        end();
    }
}
```

The default role

Lmock implicitly manages a role for the testing thread, called the **default role**. This means that you do not have to declare such a role and rely on the "traditional" directives to define a test including this role (i.e. `willInvoke`... clauses).

However, you may need to access the default role if you want to share its scenario or stubs with other roles. This is possible, thanks to `Role.defaultRole()`, as illustrated below:

```
import static com.vmware.lmock.masquerade.Schemer.*;
…
@Test
public void aTest() {
    final Thread thread1 = new Thread(new Runnable() {
        public void run() {
            joe.ping(jack);
        }
    });

    Role role1 = aRoleForThread(thread1);
    begin(role1);
    role1.shareScenarioWith(defaultRole());
    // Now that the two roles share the same scenario, using role1.willInvoke
    // or simply willInvoke is the same.
    willInvoke(2).of(joe).ping(jack);
    thread1.start();
    thread1.join(JOIN_TIMESOUT_AFTER);
    joe.ping(jack);
    end();
}
```