Mocking in Java: jMock vs. EasyMock

by Jean Tessier

This document shows how to do common mocking tasks in Java using both jMock and EasyMock.

Throughout, I use the terminology defined by Gerard Meszaros in his book xUnit Test Patterns.

All example were tested with JUnit 3.8.2, jMock 2.2.4, EasyMock 2.3, and EasyMock classextension 2.2.2 using JDK 1.5.0_13.

Table of Contents

DISCLAIMER

I like jMock a lot more than I like EasyMock. The DSL for specifying expectations in jMock takes some getting used to, but it is more expressive than the one in EasyMock. And while I don't like anonymous inner classes, I like static imports even less. :-)

Introduction

Mocking allows you to isolate a class or method and test it in isolation. You replace all of its collaborator with *mocks* that essentially simulate the normal environment of the *SUT* (*System Under Test*). Mocks replace the SUT's *DOCs* (*Depended-On Components*) and give you fine control on how the SUT interacts with its environment and what messages it gets back from it.

jMock

jMock focuses on explicitly specifying the behavior of the mocks using a specialized *DSL* (*Domain-Specific Language*) embeded in the Java code. The notation takes some getting used to, but it makes the specification of behavior stand out in the test code.

EasyMock

EasyMock takes a record/replay approach. You first train the mock by making the expected method calls on it yourself. You then switch the mock into replay-mode before exercising the SUT. Specifying the behavior is just regular method calls on a typed Java object.

Lowest Overhead Mock

EasyMock lets you create individual mocks inside your test method with very little framework machinery involved. In jMock, you always need at least a *context* object or extend a jMock superclass, as you'll see in a moment.

jMock	EasyMock
n/a	see below

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class Single_EasyMock extends TestCase {
    public void testSome() {
        SomeInterface mockSome = createMock(SomeInterface.class);
        // Program the mock here
        replay(mockSome);

        // Setup SUT with mock and exercise here
        verify(mockSome);
    }
}
```

Using One or More Mocks Together

You can control multiple mocks together. You use a special object to create and group the mocks. EasyMock calls it *control*, jMock calls it *context*. You use the control/context to validate the group of mocks as a unit.

jMock's `mock()` and EasyMock's `createMock()` methods can take an optional `String` parameter that is used to name the mock. This is needed to distinguish two mocks of the same type. It is also used to refer to the mock in failure messages, so you can use it to make these messages more expressive by clearly identifying the mock with the mismatched expectations.

In the code below, I've made the control/context a field and I create it in `setUp()`. I could put the verification in `tearDown()`, for symmetry, but that could potentially mask another error: if the test fails in the middle for some reason, JUnit will call `tearDown()` before all expectations have been met. `tearDown()` will end up throwing an exception because of the missing expectations and the root cause of the test failure will be lost. This is why I override `runTest()` instead in the code below.

jMock

```
import junit.framework.TestCase;
import org.jmock.Mockery;

public class Multiple_jMock extends TestCase {
    private Mockery context;

    protected void setUp() throws Exception {
        super.setUp();

        context = new Mockery();
    }

    protected void runTest() throws Throwable {
        super.runTest();
        context.assertIsSatisfied();
    }

    public void testSome() {
        SomeInterface mockSome = context.mock(SomeInterface.class);
        SomeOtherInterface mockSomeOther = context.mock(SomeOtherInterface.class);
        // Program the mocks here
```

EasyMock

```
import junit.framework.TestCase;
import org.easymock.EasyMock;
import org.easymock.IMocksControl;

public class Multiple_EasyMock extends TestCase {
    private IMocksControl control;

    protected void setUp() throws Exception {
        super.setUp();

        control = EasyMock.createControl();
    }

    protected void runTest() throws Throwable {
        super.runTest();
        control.verify();
    }

    public void testSome() {
        SimpleInterface mockSome = control.createMock(SimpleInterface.class);
        SomeOtherInterface mockSomeOther = control.createMock(SomeOtherInterface.class);
        // Setup SUT with mocks and exercise here
    }
}
```

```
        // Program the mocks here
        control.replay();

        // Setup SUT with mocks and exercise here
    }
}
```

Note that EasyMock.replay() and EasyMock.verify() are vararg methods, so you could do away with the *control* and simply pass them the full list of mocks. But using the *control* to keep track of all the mocks is less error-prone.

Pulling the Context or Control into a Superclass

When using jMock, you can extend MockObjectTestCase to inherit the *context* automatically. You don't interact with the *context* directly, but you use utility methods of MockObjectTestCase with maching names that delegate to the encapsulated *context*.

jMock

```
import org.jmock.integration.junit3.MockObjectTestCase;

public class Inherited_jMock extends MockObjectTestCase {
    public void testSome() {
        SomeInterface mockSome = mock(SomeInterface.class);
        // Program the mocks here

        // Setup SUT with mocks and exercise here
    }
}
```

EasyMock

EasyMock does not have an equivalent, but you can write your own base class and hide the *control* within it.

```
import junit.framework.TestCase;
import org.easymock.IMocksControl;
import org.easymock.EasyMock;

public abstract class EasyMockTestCase extends TestCase {
    private IMocksControl control;

    protected void setUp() throws Exception {
        super.setUp();

        control = EasyMock.createControl();
    }

    protected void runTest() throws Throwable {
        super.runTest();
        control.verify();
    }
    protected <T> T createMock(Class<T> clazz) {
        return control.createMock(clazz);
    }

    protected <T> T createMock(String name, Class<T> clazz) {
        return control.createMock(name, clazz);
    }

    protected void replay() {
        control.replay();
    }
}

public class Inherited_EasyMock extends EasyMockTestCase {
    public void testSome() {
        SimpleInterface mockSome = createMock(SimpleInterface.class);
        // Program the mocks here
        replay();

        // Setup SUT with mocks and exercise here
    }
}
```

Again, you do not want to call verify() from tearDown() as it might hide the test method's reason for failing, if it failed before all expectations had been met. This is why I had to override runTest() below, so verification happens only of everything else in the test method was successful.

Mocking Classes

Both jMock and EasyMock only mock interfaces by default. You need slightly different setup if you want to mock classes. Both can mock abstract or concrete classes, but no final classes. Also, neither can mock a method that has been marked final.

Luckly, in both cases, class mocking is a superset of interface mocking, so if you setup the test for class mocking, you can mock interfaces in the same test as well.

I suspect the creators of both frameworks made it this way to motivate people to code to interfaces instead of implementations.

jMock

```
import org.jmock.integration.junit3.MockObjectTestCase;
import org.jmock.lib.legacy.ClassImposteriser;
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.classextension.EasyMock.*;
```

```
public class Class_jMock extends MockObjectTestCase {
    protected void setUp() throws Exception {
        super.setUp();

        setImposteriser(ClassImposteriser.INSTANCE);
    }

    public void testSome() {
        SomeClass mockSome = mock(SomeClass.class);
        // Program the mocks here

        // Setup SUT with mocks and exercise here
    }
}
```

```
public class Class_EasyMock extends TestCase {
    public void testSome() {
        SomeClass mockSome = createMock(SomeClass.class);
        // Program the mocks here
        replay(mockSome);

        // Setup SUT with mocks and exercise here

        verify(mockSome);
    }
}
```

or, if you're using an explicit *context*, the jMock guys recommend you do this:

```
import junit.framework.TestCase;
import org.jmock.Mockery;
import org.jmock.lib.legacy.ClassImposteriser;

public class ClassExplicit1_jMock extends TestCase {
    private Mockery context = new Mockery() {{
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    protected void runTest() throws Throwable {
        super.runTest();
        context.assertIsSatisfied();
    }

    public void testSome() {
        SomeClass mockSome = context.mock(SomeClass.class);
        // Program the mocks here

        // Setup SUT with mocks and exercise here
    }
}
```

Personally, I try to initialize all state in `setUp()` as a general rule, so I would write it the following way instead, and remove the need for an anonymous inner class.

```
import junit.framework.TestCase;
import org.jmock.Mockery;
import org.jmock.lib.legacy.ClassImposteriser;
```

```
public class ClassExplicit2_jMock extends TestCase {
    private Mockery context;

    protected void setUp() throws Exception {
        super.setUp();

        context = new Mockery();
        context.setImposteriser(ClassImposteriser.INSTANCE);
    }

    protected void runTest() throws Throwable {
        super.runTest();
        context.assertIsSatisfied();
    }

    public void testSome() {
        SomeClass mockSome = context.mock(SomeClass.class);
        // Program the mocks here

        // Setup SUT with mocks and exercise here
    }
}
```

jMock uses an Imposteriser to create the mocks. MockObjectTestCase uses one that works for interfaces only by default, so you have to replace it in your own setUp() method if you need to mock classes.

EasyMock uses a completely different implementation altogether, so all you have to do is change the import statements (and use a separate JAR where the other implementation resides).

Expecting A Method To Return A Value

Let's actually do something with our mocks. Here is a simple Cache that delegates operations to an underlying Map. In real life, it would do additional processing around its interactions with the underlying storage.

```
import java.util.Map;

public class Cache {
    private Map<Integer, String> underlyingStorage;

    public Cache(Map<Integer, String> underlyingStorage) {
        this.underlyingStorage = underlyingStorage;
    }
    public String get(int key) {
        return underlyingStorage.get(key);
    }

    public void add(int key, String value) {
        underlyingStorage.put(key, value);
    }

    public void remove(int key) {
        underlyingStorage.remove(key);
    }

    public int size() {
        return underlyingStorage.size();
    }

    public void clear() {
        underlyingStorage.clear();
    }
}
```

We will write tests for `Cache` and mock `Map`. The cache is called the *SUT* for *System Under Test*. We want to show how it interacts with the mock and how we can control the mock to influence the SUT.

Here is an example showing a simple method call to a method which returns some value.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testMethodWithReturnValue() {
        final int expectedValue = 42;

        final Map mockStorage = mock(Map.class);

        checking(new Expectations() {{
            one (mockStorage).size();
                will(returnValue(expectedValue));
        }});
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

import java.util.Map;

public class CacheTest_EasyMock extends TestCase {
    public void testMethodWithReturnValue() {
        int expectedValue = 42;

        Map mockStorage = createMock(Map.class);
        expect(mockStorage.size()).andReturn(expectedValue);
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        Object actualValue = sut.size();
        assertSame(expectedValue, actualValue);
        Cache sut = new Cache(mockStorage);
        int actualValue = sut.size();
        assertSame(expectedValue, actualValue);
    }
}
```

jMock uses an anonymous inner class to set expectations using a DSL defined in the class Expectations. Local values that we plan to use as part of the expectations have to be marked final. If we used instance variables, they wouldn't have to be final. For example, we could make mockStorage an instance variable, create the mock in setup(), and still set expectations in the test method.

Because the expectation on the method call and the expectation on the returned value are in separate statements, Java cannot check that the type of the returned value is the same as the return type for the method. There will be an exception thrown at runtime if they are not compatible, so it is not a total loss.

```
        verify(mockStorage);
    }
}
```

EasyMock sets expactations using a DSL defined as static methods on class EasyMock.

EasyMock uses separate notations for methods that return something and methods that don't return a value (void). If the method returns something, you have to surround the expectation with expect() and specify a returned value.

Expecting A Method Returning A Value To Throw An Exception

Both jMock and EasyMock can throw exceptions as part of mocking method calls. This lets you simulate error conditions very easily.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testMethodWithReturnValueThrowsAnException() {
        final Exception expectedException = new RuntimeException();

        final Map mockStorage = mock(Map.class);

        checking(new Expectations() {{
            one (mockStorage).size();
                will(throwException(expectedException));
        }});
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

import java.util.Map;

public class CacheTest_EasyMock extends TestCase {
    public void testMethodWithReturnValueThrowsAnException() {
        Exception expectedException = new RuntimeException();

        Map mockStorage = createMock(Map.class);
        expect(mockStorage.size()).andThrow(expectedException);
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        try {
```

```
        Cache sut = new Cache(mockStorage);
        try {
            sut.size();
            fail("Should have thrown the exception");
        } catch (RuntimeException actualException) {
            assertSame(expectedException, actualException);
        }
    }
}
```

Like before, because the expectation on the method call and the expectation on the thrown exception are in separate statements, Java cannot check that the type of the exception matches one thrown by the method. There will be an exception thrown at runtime if they are not compatible, so it is not a total loss.

```
            sut.size();
            fail("Should have thrown the exception");
        } catch (RuntimeException actualException) {
            assertSame(expectedException, actualException);
        }

        verify(mockStorage);
    }
}
```

EasyMock is no better than jMock when it comes to the type safety of thrown exceptions.

Expecting A Method With void Return Type

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testVoidMethod() {
        final Map mockStorage = mock(Map.class);

        checking(new Expectations() {{
            one (mockStorage).clear();
        }});

        Cache sut = new Cache(mockStorage);
        sut.clear();
    }
}
```

jMock uses the exact same notation, regardless of whether the method

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

import java.util.Map;

public class CacheTest_EasyMock extends TestCase {
    public void testVoidMethod() {
        Map mockStorage = createMock(Map.class);
        mockStorage.clear();
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        sut.clear();

        verify(mockStorage);
    }
}
```

EasyMock uses separate notations for methods that return something returns a value or not.

and methods that don't return a value (void). If the method is a void method, you must **not** use expect().

Expecting A Method With void Return Type To Throw An Exception

Remember that both jMock and EasyMock can only check the typesafety of the thrown exception at runtime.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testVoidMethodThrowsAnException() {
        final Exception expectedException = new RuntimeException();

        final Map mockStorage = mock(Map.class);

        checking(new Expectations() {{
            one (mockStorage).clear();
                will(throwException(expectedException));
        }});

        Cache sut = new Cache(mockStorage);
        try {
            sut.clear();
            fail("Should have thrown the exception");
        } catch (RuntimeException actualException) {
            assertSame(expectedException, actualException);
        }
    }
}
```

Again, jMock's notation is exactly the same.

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

import java.util.Map;

public class CacheTest_EasyMock extends TestCase {
    public void testVoidMethodThrowsAnException() {
        Exception expectedException = new RuntimeException();

        Map mockStorage = createMock(Map.class);
        mockStorage.clear();
        expectLastCall().andThrow(expectedException);
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        try {
            sut.clear();
            fail("Should have thrown the exception");
        } catch (RuntimeException actualException) {
            assertSame(expectedException, actualException);
        }

        verify(mockStorage);
    }
}
```

When dealing with a void method in EasyMock, you have to use expectLastCall() to set expectations on it, like throwing an exception in this case.

Checking Parameters

If you're expecting specific values for parameters, just write them out as part of the expectations. jMock and EasyMock will use `==` for primitive types, recursively compare array contents, and use `equals()` for everything else.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
  public void testExactParams() {
    final int expectedKey = 42;
    final String expectedValue = "forty-two";

    final Map mockStorage = mock(Map.class);

    checking(new Expectations() {{
      one (mockStorage).put(expectedKey, expectedValue);
        will(returnValue(true));
    }});

    Cache sut = new Cache(mockStorage);
    sut.add(expectedKey, expectedValue);
  }
}
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class CacheTest_EasyMock extends TestCase {
  public void testExactParams() {
    int expectedKey = 42;
    String expectedValue = "forty-two";

    Map mockStorage = createMock(Map.class);
    expect(mockStorage.put(expectedKey, expectedValue)).andReturn(true);
    replay(mockStorage);

    Cache sut = new Cache(mockStorage);
    sut.add(expectedKey, expectedValue);

    verify(mockStorage);
  }
}
```

Fuzzy Matching Parameters

You can relax expectations on the parameter values using a wide range of boolean functions. Both jMock and EasyMock allow you to combine *matchers* using special boolean arithmetic functions. Each also allows you to define your own custom *matchers* if you need to. See their respective documentation for the full range of *matchers* that come with each framework.

In both cases, you must either use exact values for all parameters, or use matchers for all parameters. You cannot mix and match matchers with explicit values. Use jMock's `with(equalTo(...))` or EasyMock's `eq(...)` matchers for matching exact values.

jMock

```
import static org.hamcrest.Matchers.*;
```

EasyMock

```
import junit.framework.TestCase;
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testFuzzyParams() {
        final int expectedKey = 42;
        final String expectedValue = "forty-two";

        final Map mockStorage = mock(Map.class);

        checking(new Expectations() {{
            one (mockStorage).put(with(greaterThan(40)), with(containsString("two")));
                will(returnValue(true));
        }});

        Cache sut = new Cache(mockStorage);
        sut.add(expectedKey, expectedValue);
    }
}
```

jMock uses Hamcrest matchers to check actual parameters against expectations. New versions of JUnit are also moving to Hamcrest matchers instead of all the assert...() methods.

```
import static org.easymock.EasyMock.*;

public class CacheTest_EasyMock extends TestCase {
    public void testFuzzyParams() {
        int expectedKey = 42;
        String expectedValue = "forty-two";

        Map mockStorage = createMock(Map.class);
        expect(mockStorage.put(gt(40), find("two"))).andReturn(true);
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        sut.add(expectedKey, expectedValue);

        verify(mockStorage);
    }
}
```

EasyMock does not use Hamcrest matchers, but has a rich API that serves the same purpose.

Ignoring Irrelevant Return Values

Our example Cache class discards the value it gets from Map.put() and Map.remove(). jMock does not force us to put expectations on things that are irrelevant to the test at hand. If we're testing Cache.add(), it does not matter what the underlying call to Map.put() returns. With EasyMock, we have to fully specify everything, whether it matters or not.

jMock

```
public class CacheTest_jMock extends MockObjectTestCase {
    public void testIgnoreReturnValue() {
        final int expectedKey = 42;
        final String expectedValue = "forty-two";

        final Map mockStorage = mock(Map.class);
```

EasyMock

```
        checking(new Expectations() {{
            one (mockStorage).put(expectedKey, expectedValue);
        }});

        Cache sut = new Cache(mockStorage);
        sut.add(expectedKey, expectedValue);
    }
}
```

Notice the absence of will(returnValue(...)). jMock will supply an innocuous default value as appropriate (false in this case).

There is no equivalent in EasyMock.

Ignoring Method Calls

Both jMock and EasyMock let you ignore certain methods. Essentially, if you ignore a method, the mock does not care how many times it gets called, or even if it gets called at all.

Say we add a new logAndClear() method on the cache that logs the size of the cache before it clears it.

```
public class Cache {
    // ...

    public void logAndClear() {
        log("Clearing cache that had " + size() + " entries.");
        underlyingStorage.clear();
    }

    private void log(String message) {/* ... */}
}
```

A test method might care about Map.clear() getting called and not care about the logging behavior (that would be the topic for another test method). By keeping the test focused on the clearing logic and ignoring the logging logic, it will not break if we decide to change the implementation of logAndClear() to not include the size of the cache.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;
```

```
public class CacheTest_jMock extends MockObjectTestCase {
  public void testIgnoreMethodCall() {
    final Map mockStorage = mock(Map.class);

    checking(new Expectations() {{
      one (mockStorage).clear();
      ignoring (mockStorage).size();
        will(returnValue(42));
    }});

    Cache sut = new Cache(mockStorage);
    sut.logAndClear();
  }
}
```

With jMock, we don't have to specify the return value for `Map.size()`, as we saw in the previous section.

```
public class CacheTest_EasyMock extends TestCase {
  public void testIgnoreMethodCall_withStub() {
    Map mockStorage = createMock(Map.class);
    mockStorage.clear();
    expect(mockStorage.size()).andStubReturn(42);
    replay(mockStorage);

    Cache sut = new Cache(mockStorage);
    sut.logAndClear();

    verify(mockStorage);
  }
}
```

With EasyMock, we must specify a return value for `Map.size()`.

Ignoring Whole Objects

In addition, jMock and EasyMock let you ignore all calls to a given mock, making it perfect for creating fake objects on the fly. Fake objects may be required by the API of the SUT but are otherwise irrelevant to the test at hand.

For example, a test method may try to exercise some part of the `Cache` without caring what happens on the underlying storage. The storage cannot be null, but we can create a fake storage that will do nothing.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

public class IgnoringObject_jMock extends MockObjectTestCase {
  public void testIgnoreObject() {
    final Map mockStorage = mock(Map.class);

    checking(new Expectations() {{
      ignoring (mockStorage);
    }});
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.classextension.EasyMock.*;

public class CacheTest_EasyMock extends TestCase {
  public void testIgnoreObject() {
    Map mockStorage = createNiceMock(Map.class);
    replay(mockStorage);

    Cache sut = new Cache(mockStorage);
    sut.logAndClear();
```

```
        Cache sut = new Cache(mockStorage);
        sut.logAndClear();
    }
}
```

```
        verify(mockStorage);
    }
}
```

Innocuous Default Values

jMock and EasyMock can supply return values for you if you do not care about the actual returned value. They will automatically return zero for numerical values and false for boolean values.

For this example, we will expand `Cache` to be a `UserCache`, as show here:

```
import java.util.logging.Logger;

public class UserCache {
    private Storage underlyingStorage;
    private Logger logger;

    public UserCache(Storage underlyingStorage, Logger logger) {
        this.underlyingStorage = underlyingStorage;
        this.logger = logger;
    }

    public UserRecord getAndLog(int key) {
        UserRecord result = underlyingStorage.get(key);
        logger.log(key + " --> \"" + result + "\"");
        return result;
    }

    public UserRecord getAndLogName(int key) {
        UserRecord result = underlyingStorage.get(key);
        logger.log(key + " --> \"" + result.getLastName() + ", " + result.getFirstName() + "\"");
        return result;
    }
}

public interface Storage {
    UserRecord get(int key);
}

public interface Logger {
    void log(String message);
}

public interface UserRecord {
    String getFirstName();
    String getLastName();
}
```

We want to test that calling getAndLog() actually writes something to the logs, without caring about the details of what gets written.

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

public class UserCacheTest_jMock extends MockObjectTestCase {
    public void testInnocuousValue() {
        final int key = 42;

        final Storage mockStorage = mock(Storage.class);
        final Logger mockLogger = mock(Logger.class);

        checking(new Expectations() {{
            one (mockStorage).get(key);
            one (mockLogger).log(with(any(String.class)));
        }});

        UserCache sut = new UserCache(mockStorage, mockLogger);
        sut.getAndLog(key);
    }
}
```

For methods that return an object type, jMock generates a mock for that type on the fly so it can continue returning innocuous values across chains of calls. And it returns an empty string for String.

The exact same test would work with getAndLogName() because result would turn out to be an ignored mock.

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class UserCacheTest_EasyMock extends TestCase {
    public void testInnocuousValue() {
        int expectedKey = 42;

        Storage mockStorage = createNiceMock(Storage.class);
        Logger mockLogger = createMock(Logger.class);
        mockLogger.log(isA(String.class));
        replay(mockStorage, mockLogger);

        UserCache sut = new UserCache(mockStorage, mockLogger);
        sut.getAndLog(expectedKey);

        verify(mockStorage, mockLogger);
    }
}
```

For methods that return an object type, EasyMock returns null, even for String. And the only choice is to ignore the entire method call, you cannot place an expectation on the method getting called and still get the innocuous default values.

The test would not work with getAndLogName() because result would be null and building the message would throw a NullPointerException.

Mocking Repeated Method Calls

Both jMock and EasyMock let you specify how many times you expect a given method to be called. They both let you specified exact numbers or constraints like "at least" or "at most" so many calls.

If we return to our Cache example and add the following logAndClear() method that logs and clears only if the cache is not empty:

```
public class Cache {
   // ...

   public void conditionalLogAndClear() {
      if (size() > 0) {
         logAndclear();
      }
   }
}
```

jMock

```
import org.jmock.Expectations;
import org.jmock.integration.junit3.MockObjectTestCase;

import java.util.Map;

public class CacheTest_jMock extends MockObjectTestCase {
   public void testMultipleCalls() {
      final Map mockStorage = mock(Map.class);

      checking(new Expectations() {{
         exactly(2).of (mockStorage).size();
         will(returnValue(42));
         one (mockStorage).clear();
      }});

      Cache sut = new Cache(mockStorage);
      sut.conditionalLogAndClear();
   }
}
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class CacheTest_EasyMock extends TestCase {
   public void testMultipleCalls() {
      Map mockStorage = createMock(Map.class);
      expect(mockStorage.size()).andReturn(42).times(2);
      mockStorage.clear();
      replay(mockStorage);

      Cache sut = new Cache(mockStorage);
      sut.conditionalLogAndClear();

      verify(mockStorage);
   }
}
```

Checking that Calls Happen In Sequence on One Mock

By default, both jMock and EasyMock do not order the expectations. The calls do not have to occur in the same order they are specified in the test. But each framework provides a mechanism for checking that things happen in a given sequence.

jMock

```
import org.jmock.Expectations;
import org.jmock.Sequence;
import org.jmock.integration.junit3.MockObjectTestCase;

public class CacheTest_jMock extends MockObjectTestCase {
    public void testSequenceOnOneMock() {
        final Map mockStorage = mock(Map.class);
        final Sequence clearSequence = sequence("clear");

        checking(new Expectations() {{
            one (mockStorage).size();
                inSequence(clearSequence);
                will(returnValue(42));
            one (mockStorage).clear();
                inSequence(clearSequence);
        }});

        Cache sut = new Cache(mockStorage);
        sut.logAndClear();
    }
}
```

With jMock, you can control which individual calls are on or off the sequence by including or omitting `inSequence()` statements.

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;

public class CacheTest_EasyMock extends TestCase {
    public void testSequenceOnOneMock() {
        Map mockStorage = createStrictMock(Map.class);
        expect(mockStorage.size()).andReturn(42);
        mockStorage.clear();
        replay(mockStorage);

        Cache sut = new Cache(mockStorage);
        sut.logAndClear();

        verify(mockStorage);
    }
}
```

With EasyMock, you can control which calls are on or off the sequence by calling `EasyMock.checkOrder(mock, boolean)` repeatedly to turn on or off sequence ordering.

Checking that Calls Happen In Sequence Across Mocks

jMock and EasyMock have slightly different mechanism for checking call sequences involving multiple mocks.

jMock

```
import org.jmock.Expectations;
import org.jmock.Sequence;
import org.jmock.integration.junit3.MockObjectTestCase;
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;
import org.easymock.IMocksControl;
```

```
public class UserCacheTest_jMock extends MockObjectTestCase {
    public void testSequenceOnTwoMocks() {
        final int key = 42;

        final Storage mockStorage = mock(Storage.class);
        final Logger mockLogger = mock(Logger.class);
        final Sequence getSequence = sequence("get");

        checking(new Expectations() {{
            one (mockStorage).get(key);
                inSequence(getSequence);
            one (mockLogger).log(with(any(String.class)));
                inSequence(getSequence);
        }});

        UserCache sut = new UserCache(mockStorage, mockLogger);
        sut.getAndLog(key);
    }
}
```

jMock does not see a difference between a sequence with one mock or multiple mocks.

You can create as many independent sequences as you need.

```
public class UserCacheTest_EasyMock extends TestCase {
    public void testSequenceOnTwoMocks() {
        int expectedKey = 42;

        IMocksControl control = createStrictControl();

        Storage mockStorage = control.createMock(Storage.class);
        Logger mockLogger = control.createMock(Logger.class);
        UserRecord mockUser = control.createMock(UserRecord.class);
        expect(mockStorage.get(expectedKey)).andReturn(mockUser);
        mockLogger.log(isA(String.class));
        control.replay();

        UserCache sut = new UserCache(mockStorage, mockLogger);
        sut.getAndLog(expectedKey);

        control.verify();
    }
}
```

EasyMock uses the *control* to coordinate action across multiple mocks. You can control which calls are on or off the sequence by calling IMockControls.checkOrder(boolean) repeatedly.

You need to use separate controls if you need to check more than one independent sequences.

Providing Side Effects in Mocks

Recently, I came upon a case where the SUT was using a collaborator to populate some data structure. When mocking said collaborator, I needed to replicate the behavior to populate the structure so the SUT could proceed. I think of this as a code smell, but it was too hard to refactor on the spot, so I wanted to put a test in place and maybe come back to it at some time in the future.

Both jMock and EasyMock let you provide behavior that gets run as part of mocking a method call.

Assume following Populator:

```
import java.util.List;
```

```
public interface Populator {
    void populate(List list);
}
```

And the SUT is this `Client` class:

```
import java.util.List;
import java.util.ArrayList;

public class Client {
    private final Populator populator;

    public Client(Populator populator) {
        this.populator = populator;
    }

    public int callPopulateAndReturnSize() {
        List list = new ArrayList();
        populator.populate(list);
        return list.size();
    }
}
```

Here are tests for `callPopulateAndReturnSize()`.

jMock

```
import org.jmock.Expectations;
import org.jmock.api.Invocation;
import org.jmock.integration.junit3.MockObjectTestCase;
import org.jmock.lib.action.CustomAction;

import java.util.List;

public class SideEffect_jMock extends MockObjectTestCase {
    public void testSideEffect() {
        final Populator mockPopulator = mock(Populator.class);

        checking(new Expectations() {{
            one (mockPopulator).populate(with(any(List.class)));
                will(new CustomAction("Add random value to list") {
                    public Object invoke(Invocation invocation) throws Throwable {
                        ((List) invocation.getParameter(0)).add(new Object());
                        return null;
                    }
```

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.EasyMock.*;
import org.easymock.IAnswer;

import java.util.List;

public class SideEffect_EasyMock extends TestCase {
    public void testSideEffect() {
        Populator mockPopulator = createMock(Populator.class);
        mockPopulator.populate(isA(List.class));
        expectLastCall().andAnswer(new IAnswer() {
            public Object answer() throws Throwable {
                ((List) getCurrentArguments()[0]).add(new Object());
                return null;
            }
        });
        replay(mockPopulator);
```

```
            });
        }});

        Client sut = new Client(mockPopulator);
        int actualSize = sut.callPopulateAndReturnSize();
        assertTrue(actualSize > 0);
    }
}
```

```
        Client sut = new Client(mockPopulator);
        int actualValue = sut.callPopulateAndReturnSize();
        assertTrue(actualValue > 0);

        verify(mockPopulator);
    }
}
```

In both cases, having the inner class definition inlined into the expectations makes the code hard to read. You'd be better off extracting it to a factory method. We left it in there for these examples so that you could compare this expectation to other we've shown before.

Partial Mocking

EasyMock lets you mock only parts of a class so you can test the behavior of methods that call other methods on the same object instead of external collaborators.

In the test below, we verify that when we call the logAndClear() method, it calls the log() method on the SUT.

jMock

n/a

EasyMock

```
import junit.framework.TestCase;
import static org.easymock.classextension.EasyMock.*;

import java.lang.reflect.Method;
import java.util.Map;

public class PartialMocking_EasyMock extends TestCase {
    public void testPartialMocking() throws Exception {
        Map mockStorage = createMock(Map.class);
        mockStorage.clear();
        expect(mockStorage.size()).andStubReturn(42);

        Cache sut = createMock(Cache.class, new Method[] {Cache.class.getMethod("log", String.class)});
        sut.log(isA(String.class));

        replay(mockStorage, sut);

        sut.setUnderlyingStorage(mockStorage);
        sut.logAndClear();

        verify(mockStorage);
    }
```

	`}`
There is no equivalent in jMock.	EasyMock uses reflection to find out which methods to mock and which ones to leave as implemented. If you are using `Class.getMethod()` to locate the method to mock, as in the example, you can mock any public methods of the SUT, whether it is inherited or declared on its concrete class, but you cannot mock any package-level or protected or private methods. If you are using `Class.getDeclaredMethod()` to locate the method to mock, you can mock any methods of the SUT's concrete class, whether it is public or package-level or protected or private, but you cannot mock any inherited methods.

This document was first written on 2008-05-29. It was last updated on 2008-11-20.