EasyMock 2.3 Readme

Documentation for release 2.3 (July 9 2007)

EasyMock 2 is a library that provides an easy way to use Mock Objects for given interfaces. EasyMock 2 is available under the terms of the MIT license.

Mock Objects simulate parts of the behavior of domain code, and are able to check whether they are used as defined. Domain classes can be tested in isolation by simulating their collaborators with Mock Objects.

Writing and maintaining Mock Objects often is a tedious task that may introduce errors. EasyMock 2 generates Mock Objects dynamically - no need to write them, and no generated code!

EasyMock 2 Benefits

- Hand-writing classes for Mock Objects is not needed.
- Supports refactoring-safe Mock Objects: test code will not break at runtime when renaming methods or reordering method parameters
- Supports return values and exceptions.
- Supports checking the order of method calls, for one or more Mock Objects.

EasyMock 2 Drawbacks

- EasyMock 2 does only work with Java 2 Version 5.0 and above.

EasyMock by default supports the generation of Mock Objects for interfaces only. For those who would like to generate Mock Objects for classes, there is an extension available at the EasyMock home page.

Installation

1. Java 2 (at least 5.0) is required.
2. Unzip the EasyMock zip file (`easymock2.3.zip`). It contains a directory `easymock2.3`. Add the EasyMock jar file (`easymock.jar`) from this directory to your classpath.

To execute the EasyMock tests, add `tests.zip` and the JUnit 4.1 jar to your class path and start `'java org.easymock.tests.AllTests'`.

The source code of EasyMock is stored in the zip file `src.zip`.

Usage

Most parts of a software system do not work in isolation, but collaborate with other parts to get their job done. In a lot of cases, we do not care about using collaborators in unit testing, as we trust these collaborators. If we *do* care about it, Mock Objects help us to test the unit under test in isolation. Mock Objects replace collaborators of the unit under test.

The following examples use the interface `Collaborator`:

```
package org.easymock.samples;

public interface Collaborator {
    void documentAdded(String title);
    void documentChanged(String title);
    void documentRemoved(String title);
    byte voteForRemoval(String title);
    byte[] voteForRemovals(String[] title);
}
```

Implementors of this interface are collaborators (in this case listeners) of a class named `ClassUnderTest`:

```
public class ClassUnderTest {
    // ...
    public void addListener(Collaborator listener) {
        // ...
    }
    public void addDocument(String title, byte[] document) {
        // ...
    }
    public boolean removeDocument(String title) {
        // ...
    }
    public boolean removeDocuments(String[] titles) {
        // ...
    }
}
```

The code for both the class and the interface may be found in the package `org.easymock.samples` in `samples.zip`.

The following examples assume that you are familiar with the JUnit testing framework. Although the tests shown here use JUnit in version 3.8.1, you may as well use JUnit 4 or TestNG.

The first Mock Object

We will now build a test case and toy around with it to understand the functionality of the EasyMock package. `samples.zip` contains a modified version of this test. Our first test should check whether the removal of a non-existing document does **not** lead to a notification of the collaborator. Here is the test without the definition of the Mock Object:

```
package org.easymock.samples;

import junit.framework.TestCase;

public class ExampleTest extends TestCase {

    private ClassUnderTest classUnderTest;
    private Collaborator mock;

    protected void setUp() {
        classUnderTest = new ClassUnderTest();
        classUnderTest.addListener(mock);
    }

    public void testRemoveNonExistingDocument() {
        // This call should not lead to any notification
        // of the Mock Object:
        classUnderTest.removeDocument("Does not exist");
    }
}
```

For many tests using EasyMock 2, we only need a static import of methods of `org.easymock.EasyMock`. This is the only non-internal, non-deprecated class of EasyMock 2.

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

public class ExampleTest extends TestCase {

    private ClassUnderTest classUnderTest;
    private Collaborator mock;

}
```

To get a Mock Object, we need to

1. create a Mock Object for the interface we would like to simulate,
2. record the expected behavior, and
3. switch the Mock Object to replay state.

Here is a first example:

```
    protected void setUp() {
        mock = createMock(Collaborator.class); // 1
        classUnderTest = new ClassUnderTest();
        classUnderTest.addListener(mock);
    }

    public void testRemoveNonExistingDocument() {
        // 2 (we do not expect anything)
        replay(mock); // 3
        classUnderTest.removeDocument("Does not exist");
    }
```

After activation in step 3, `mock` is a Mock Object for the `Collaborator` interface that expects no calls. This means that if we change our `ClassUnderTest` to call any of the interface's methods, the Mock Object will throw an `AssertionError`:

```
java.lang.AssertionError:
  Unexpected method call documentRemoved("Does not exist"):
     at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
     at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:44)
     at $Proxy0.documentRemoved(Unknown Source)
     at org.easymock.samples.ClassUnderTest.notifyListenersDocumentRemoved(ClassUnderTest.java:74)
     at org.easymock.samples.ClassUnderTest.removeDocument(ClassUnderTest.java:33)
     at org.easymock.samples.ExampleTest.testRemoveNonExistingDocument(ExampleTest.java:24)
     ...
```

Adding Behavior

Let us write a second test. If a document is added on the class under test, we expect a call to `mock.documentAdded()` on the Mock Object with the title of the document as argument:

```
    public void testAddDocument() {
        mock.documentAdded("New Document"); // 2
        replay(mock); // 3
        classUnderTest.addDocument("New Document", new byte[0]);
    }
```

So in the record state (before calling `replay`), the Mock Object does *not* behave like a Mock Object, but it records method calls. After calling `replay`, it behaves like a Mock Object, checking whether the expected method calls are really done.

If `classUnderTest.addDocument("New Document", new byte[0])` calls the expected method with a wrong argument, the Mock Object will complain with an `AssertionError`:

```
java.lang.AssertionError:
  Unexpected method call documentAdded("Wrong title"):
    documentAdded("New Document"): expected: 1, actual: 0
     at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
     at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:44)
     at $Proxy0.documentAdded(Unknown Source)
     at org.easymock.samples.ClassUnderTest.notifyListenersDocumentAdded(ClassUnderTest.java:61)
     at org.easymock.samples.ClassUnderTest.addDocument(ClassUnderTest.java:28)
     at org.easymock.samples.ExampleTest.testAddDocument(ExampleTest.java:30)
     ...
```

All missed expectations are shown, as well as all fulfilled expectations for the unexpected call (none in this case). If the method call is executed too often, the Mock Object complains, too:

```
java.lang.AssertionError:
  Unexpected method call documentAdded("New Document"):
    documentAdded("New Document"): expected: 1, actual: 1 (+1)
     at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
     at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:44)
     at $Proxy0.documentAdded(Unknown Source)
     at org.easymock.samples.ClassUnderTest.notifyListenersDocumentAdded(ClassUnderTest.java:62)
     at org.easymock.samples.ClassUnderTest.addDocument(ClassUnderTest.java:29)
     at org.easymock.samples.ExampleTest.testAddDocument(ExampleTest.java:30)
    ...
```

Verifying Behavior

There is one error that we have not handled so far: If we specify behavior, we would like to verify that it is actually used. The current test would pass if no method on the Mock Object is called. To verify that the specified behavior has been used, we have to call `verify(mock)`:

```
    public void testAddDocument() {
        mock.documentAdded("New Document"); // 2
        replay(mock); // 3
        classUnderTest.addDocument("New Document", new byte[0]);
        verify(mock);
    }
```

If the method is not called on the Mock Object, we now get the following exception:

```
java.lang.AssertionError:
  Expectation failure on verify:
    documentAdded("New Document"): expected: 1, actual: 0
     at org.easymock.internal.MocksControl.verify(MocksControl.java:70)
     at org.easymock.EasyMock.verify(EasyMock.java:536)
     at org.easymock.samples.ExampleTest.testAddDocument(ExampleTest.java:31)
    ...
```

The message of the exception lists all missed expectations.

Expecting an Explicit Number of Calls

Up to now, our test has only considered a single method call. The next test should check whether the addition of an already existing document leads to a call to `mock.documentChanged()` with the appropriate argument. To be sure, we check this three times (hey, it is an example ;-)):

```
    public void testAddAndChangeDocument() {
        mock.documentAdded("Document");
        mock.documentChanged("Document");
        mock.documentChanged("Document");
        mock.documentChanged("Document");
        replay(mock);
        classUnderTest.addDocument("Document", new byte[0]);
        classUnderTest.addDocument("Document", new byte[0]);
        classUnderTest.addDocument("Document", new byte[0]);
        classUnderTest.addDocument("Document", new byte[0]);
        verify(mock);
    }
```

To avoid the repetition of `mock.documentChanged("Document")`, EasyMock provides a shortcut. We may specify the call count with the method `times(int times)` on the object returned by `expectLastCall()`. The code then looks like:

```
public void testAddAndChangeDocument() {
    mock.documentAdded("Document");
    mock.documentChanged("Document");
    expectLastCall().times(3);
    replay(mock);
    classUnderTest.addDocument("Document", new byte[0]);
    classUnderTest.addDocument("Document", new byte[0]);
    classUnderTest.addDocument("Document", new byte[0]);
    classUnderTest.addDocument("Document", new byte[0]);
    verify(mock);
}
```

If the method is called too often, we get an exception that tells us that the method has been called too many times. The failure occurs immediately at the first method call exceeding the limit:

```
java.lang.AssertionError:
  Unexpected method call documentChanged("Document"):
    documentChanged("Document"): expected: 3, actual: 3 (+1)
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:44)
        at $Proxy0.documentChanged(Unknown Source)
        at org.easymock.samples.ClassUnderTest.notifyListenersDocumentChanged(ClassUnderTest.java:67)
        at org.easymock.samples.ClassUnderTest.addDocument(ClassUnderTest.java:26)
        at org.easymock.samples.ExampleTest.testAddAndChangeDocument(ExampleTest.java:43)
    ...
```

If there are too few calls, `verify(mock)` throws an `AssertionError`:

```
java.lang.AssertionError:
  Expectation failure on verify:
    documentChanged("Document"): expected: 3, actual: 2
        at org.easymock.internal.MocksControl.verify(MocksControl.java:70)
        at org.easymock.EasyMock.verify(EasyMock.java:536)
        at org.easymock.samples.ExampleTest.testAddAndChangeDocument(ExampleTest.java:43)
    ...
```

Specifying Return Values

For specifying return values, we wrap the expected call in `expect(T value)` and specify the return value with the method `andReturn(Object returnValue)` on the object returned by `expect(T value)`.

As an example, we check the workflow for document removal. If `ClassUnderTest` gets a call for document removal, it asks all collaborators for their vote for removal with calls to `byte voteForRemoval(String title)` value. Positive return values are a vote for removal. If the sum of all values is positive, the document is removed and `documentRemoved(String title)` is called on all collaborators:

```
public void testVoteForRemoval() {
    mock.documentAdded("Document");   // expect document addition
    // expect to be asked to vote for document removal, and vote for it
    expect(mock.voteForRemoval("Document")).andReturn((byte) 42);
    mock.documentRemoved("Document"); // expect document removal
    replay(mock);
    classUnderTest.addDocument("Document", new byte[0]);
```

```
        assertTrue(classUnderTest.removeDocument("Document"));
        verify(mock);
    }

    public void testVoteAgainstRemoval() {
        mock.documentAdded("Document");   // expect document addition
        // expect to be asked to vote for document removal, and vote against it
        expect(mock.voteForRemoval("Document")).andReturn((byte) -42);
        replay(mock);
        classUnderTest.addDocument("Document", new byte[0]);
        assertFalse(classUnderTest.removeDocument("Document"));
        verify(mock);
    }
```

The type of the returned value is checked at compile time. As an example, the following code will not compile, as the type of the provided return value does not match the method's return value:

```
    expect(mock.voteForRemoval("Document")).andReturn("wrong type");
```

Instead of calling `expect(T value)` to retrieve the object for setting the return value, we may also use the object returned by `expectLastCall()`. Instead of

```
    expect(mock.voteForRemoval("Document")).andReturn((byte) 42);
```

we may use

```
    mock.voteForRemoval("Document");
    expectLastCall().andReturn((byte) 42);
```

This type of specification should only be used if the line gets too long, as it does not support type checking at compile time.

Working with Exceptions

For specifying exceptions (more exactly: Throwables) to be thrown, the object returned by `expectLastCall()` and `expect(T value)` provides the method `andThrow(Throwable throwable)`. The method has to be called in record state after the call to the Mock Object for which it specifies the `Throwable` to be thrown.

Unchecked exceptions (that is, `RuntimeException`, `Error` and all their subclasses) can be thrown from every method. Checked exceptions can only be thrown from the methods that do actually throw them.

Creating Return Values or Exceptions

Sometimes we would like our mock object to return a value or throw an exception that is created at the time of the actual call. Since EasyMock 2.2, the object returned by `expectLastCall()` and `expect(T value)` provides the method `andAnswer(IAnswer answer)` which allows to specify an implementation of the interface `IAnswer` that is used to create the return value or exception.

Inside an `IAnswer` callback, the arguments passed to the mock call are available via `EasyMock.getCurrentArguments()`. If you use these, refactorings like reordering parameters may break your tests. You have been warned.

Changing Behavior for the Same Method Call

It is also possible to specify a changing behavior for a method. The methods `times`, `andReturn`, and `andThrow` may be chained. As an example, we define `voteForRemoval("Document")` to

- return 42 for the first three calls,
- throw a `RuntimeException` for the next four calls,
- return -42 once.

```
expect(mock.voteForRemoval("Document"))
    .andReturn((byte) 42).times(3)
    .andThrow(new RuntimeException(), 4)
    .andReturn((byte) -42);
```

Relaxing Call Counts

To relax the expected call counts, there are additional methods that may be used instead of `times(int count)`:

`times(int min, int max)`
  to expect between `min` and `max` calls,
`atLeastOnce()`
  to expect at least one call, and
`anyTimes()`
  to expected an unrestricted number of calls.

If no call count is specified, one call is expected. If we would like to state this explicitely, `once()` or `times(1)` may be used.

Strict Mocks

On a Mock Object returned by a `EasyMock.createMock()`, the order of method calls is not checked. If you would like a strict Mock Object that checks the order of method calls, use `EasyMock.create`*`Strict`*`Mock()` to create it.

If an unexpected method is called on a strict Mock Object, the message of the exception will show the method calls expected at this point followed by the first conflicting one. `verify(mock)` shows all missing method calls.

Switching Order Checking On and Off

Sometimes, it is necessary to have a Mock Object that checks the order of only some calls. In record phase, you may switch order checking on by calling `checkOrder(mock, true)` and switch it off by calling `checkOrder(mock, false)`.

There are two differences between a strict Mock Object and a normal Mock Object:

1. A strict Mock Object has order checking enabled after creation.
2. A strict Mock Object has order checking enabled after reset (see *Reusing a Mock Object*).

Flexible Expectations with Argument Matchers

To match an actual method call on the Mock Object with an expectation, `Object` arguments are by default compared with `equals()`. This may lead to problems. As an example, we consider the following expectation:

```
String[] documents = new String[] { "Document 1", "Document 2" };
expect(mock.voteForRemovals(documents)).andReturn(42);
```

If the method is called with another array with the same contents, we get an exception, as `equals()` compares object identity for arrays:

```
java.lang.AssertionError:
  Unexpected method call voteForRemovals([Ljava.lang.String;@9a029e):
    voteForRemovals([Ljava.lang.String;@2db19d): expected: 1, actual: 0
    documentRemoved("Document 1"): expected: 1, actual: 0
    documentRemoved("Document 2"): expected: 1, actual: 0
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:44)
        at $Proxy0.voteForRemovals(Unknown Source)
        at org.easymock.samples.ClassUnderTest.listenersAllowRemovals(ClassUnderTest.java:88)
        at org.easymock.samples.ClassUnderTest.removeDocuments(ClassUnderTest.java:48)
        at org.easymock.samples.ExampleTest.testVoteForRemovals(ExampleTest.java:83)
    ...
```

To specify that only array equality is needed for this call, we may use the method `aryEq` that is statically imported from the `EasyMock` class:

```
String[] documents = new String[] { "Document 1", "Document 2" };
expect(mock.voteForRemovals(aryEq(documents))).andReturn(42);
```

If you would like to use matchers in a call, you have to specify matchers for all arguments of the method call.

There are a couple of predefined argument matchers available.

`eq(X value)`
    Matches if the actual value is equals the expected value. Available for all primitive types and for objects.
`anyBoolean()`, `anyByte()`, `anyChar()`, `anyDouble()`, `anyFloat()`, `anyInt()`, `anyLong()`, `anyObject()`, `anyShort()`

Matches any value. Available for all primitive types and for objects.

`eq(X value, X delta)`
: Matches if the actual value is equal to the given value allowing the given delta. Available for `float` and `double`.

`aryEq(X value)`
: Matches if the actual value is equal to the given value according to `Arrays.equals()`. Available for primitive and object arrays.

`isNull()`
: Matches if the actual value is null. Available for objects.

`notNull()`
: Matches if the actual value is not null. Available for objects.

`same(X value)`
: Matches if the actual value is the same as the given value. Available for objects.

`isA(Class clazz)`
: Matches if the actual value is an instance of the given class, or if it is in instance of a class that extends or implements the given class. Available for objects.

`lt(X value)`, `leq(X value)`, `geq(X value)`, `gt(X value)`
: Matches if the actual value is less/less or equal/greater or equal/greater than the given value. Available for all numeric primitive types and `Comparable`.

`startsWith(String prefix), contains(String substring), endsWith(String suffix)`
: Matches if the actual value starts with/contains/ends with the given value. Available for `String`s.

`matches(String regex), find(String regex)`
: Matches if the actual value/a substring of the actual value matches the given regular expression. Available for `String`s.

`and(X first, X second)`
: Matches if the matchers used in `first` and `second` both match. Available for all primitive types and for objects.

`or(X first, X second)`
: Matches if one of the matchers used in `first` and `second` match. Available for all primitive types and for objects.

`not(X value)`
: Matches if the matcher used in `value` does not match.

`cmpEq(X value)`
: Matches if the actual value is equals according to `Comparable.compareTo(X o)`. Available for all numeric primitive types and `Comparable`.

`cmp(X value, Comparator comparator, LogicalOperator operator)`
: Matches if `comparator.compare(actual, value) operator 0` where the operator is <,<=,>,>= or ==. Available for objects.

Defining your own Argument Matchers

Sometimes it is desirable to define own argument matchers. Let's say that an argument matcher is needed that matches an exception if the given exception has the same type and an equal message. It should be used this way:

```
IllegalStateException e = new IllegalStateException("Operation not allowed.")
expect(mock.logThrowable(eqException(e))).andReturn(true);
```

Two steps are necessary to achieve this: The new argument matcher has to be defined, and the static method `eqException` has to be declared.

To define the new argument matcher, we implement the interface `org.easymock.IArgumentMatcher`. This interface contains two methods: `matches(Object actual)` checks whether the actual argument matches the given argument, and `appendTo(StringBuffer buffer)` appends a string representation of the argument matcher to the given string buffer. The implementation is straightforward:

```
import org.easymock.IArgumentMatcher;

public class ThrowableEquals implements IArgumentMatcher {
    private Throwable expected;

    public ThrowableEquals(Throwable expected) {
        this.expected = expected;
    }

    public boolean matches(Object actual) {
        if (!(actual instanceof Throwable)) {
            return false;
        }
        String actualMessage = ((Throwable) actual).getMessage();
        return expected.getClass().equals(actual.getClass())
                && expected.getMessage().equals(actualMessage);
    }

    public void appendTo(StringBuffer buffer) {
        buffer.append("eqException(");
        buffer.append(expected.getClass().getName());
        buffer.append(" with message \"");
        buffer.append(expected.getMessage());
        buffer.append("\")");

    }
}
```

The method `eqException` must create the argument matcher with the given Throwable, report it to EasyMock via the static method `reportMatcher(IArgumentMatcher matcher)`, and return a value so that it may be used inside the call (typically `0`, `null` or `false`). A first attempt may look like:

```
public static Throwable eqException(Throwable in) {
    EasyMock.reportMatcher(new ThrowableEquals(in));
    return null;
}
```

However, this only works if the method `logThrowable` in the example usage accepts `Throwables`, and does not require something more specific like a `RuntimeException`. In the latter case, our code sample would not compile:

```
     IllegalStateException e = new IllegalStateException("Operation not allowed.")
     expect(mock.logThrowable(eqException(e))).andReturn(true);
```

Java 5.0 to the rescue: Instead of defining `eqException` with a `Throwable` as parameter and return value, we use a generic type that extends `Throwable`:

```
public static <T extends Throwable> T eqException(T in) {
     reportMatcher(new ThrowableEquals(in));
     return null;
}
```

Reusing a Mock Object

Mock Objects may be reset by `reset(mock)`.

Using Stub Behavior for Methods

Sometimes, we would like our Mock Object to respond to some method calls, but we do not want to check how often they are called, when they are called, or even if they are called at all. This stub behavoir may be defined by using the methods `andStubReturn(Object value)`, `andStubThrow(Throwable throwable)`, `andStubAnswer(IAnswer answer)` and `asStub()`. The following code configures the MockObject to answer 42 to `voteForRemoval("Document")` once and -1 for all other arguments:

```
     expect(mock.voteForRemoval("Document")).andReturn(42);
     expect(mock.voteForRemoval(not(eq("Document")))).andStubReturn(-1);
```

Nice Mocks

On a Mock Object returned by `createMock()` the default behavior for all methods is to throw an `AssertionError` for all unexpected method calls. If you would like a "nice" Mock Object that by default allows all method calls and returns appropriate empty values (`0`, `null` or `false`), use `create`*`Nice`*`Mock()` instead.

Object Methods

The behavior for the three object methods `equals()`, `hashCode()` and `toString()` cannot be changed for Mock Objects created with EasyMock, even if they are part of the interface for which the Mock Object is created.

Checking Method Call Order Between Mocks

Up to this point, we have seen a mock object as a single object that is configured by static methods on the class `EasyMock`. But many of these static methods just identify the hidden control of the Mock Object and delegate to it.

A Mock Control is an object implementing the `IMocksControl` interface.

So instead of

```
IMyInterface mock = createStrictMock(IMyInterface.class);
replay(mock);
verify(mock);
reset(mock);
```

we may use the equivalent code:

```
IMocksControl ctrl = createStrictControl();
IMyInterface mock = ctrl.createMock(IMyInterface.class);
ctrl.replay();
ctrl.verify();
ctrl.reset();
```

The IMocksControl allows to create more than one Mock Object, and so it is possible to check the order of method calls between mocks. As an example, we set up two mock objects for the interface `IMyInterface`, and we expect the calls `mock1.a()` and `mock2.a()` ordered, then an open number of calls to `mock1.c()` and `mock2.c()`, and finally `mock2.b()` and `mock1.b()`, in this order:

```
IMocksControl ctrl = createStrictControl();
IMyInterface mock1 = ctrl.createMock(IMyInterface.class);
IMyInterface mock2 = ctrl.createMock(IMyInterface.class);

mock1.a();
mock2.a();

ctrl.checkOrder(false);

mock1.c();
expectLastCall().anyTimes();
mock2.c();
expectLastCall().anyTimes();

ctrl.checkOrder(true);

mock2.b();
mock1.b();

ctrl.replay();
```

Naming Mock Objects

Mock Objects can be named at creation using `createMock(String name, Class toMock)`, `createStrictMock(String name, Class toMock)` or `createNiceMock(String name, Class toMock)`. The names will be shown in exception failures.

Backward Compatibility

EasyMock 2 contains a compatibility layer so that tests using EasyMock 1.2 for Java 1.5 should work without any modification. The only known differences are visible when failures occur: there are small changes in the failure messages and stack traces, and failures are now reported using Java's `AssertionError` instead of JUnit's `AssertionFailedError`.

EasyMock 2.1 introduced a callback feature that has been removed in EasyMock 2.2, as it was too complex. Since EasyMock 2.2, the `IAnswer` interface provides the functionality for callbacks.

EasyMock Development

EasyMock 1.0 has been developed by Tammo Freese at OFFIS. The development of EasyMock is now hosted on SourceForge to allow other developers and companies to contribute.

Thanks to the people who gave feedback or provided patches, including Nascif Abousalh-Neto, Dave Astels, Francois Beausoleil, George Dinwiddie, Shane Duan, Wolfgang Frech, Steve Freeman, Oren Gross, John D. Heintz, Dale King, Brian Knorr, Dierk Koenig, Chris Kreussling, Robert Leftwich, Patrick Lightbody, Johannes Link, Rex Madden, David McIntosh, Karsten Menne, Bill Michell, Stephan Mikaty, Ivan Moore, Ilja Preuss, Justin Sampson, Markus Schmidlin, Richard Scott, Joel Shellman, Jiří Mareš, Alexandre de Pellegrin Shaun Smith, Marco Struck, Ralf Stuckert, Victor Szathmary, Henri Tremblay, Bill Uetrecht, Frank Westphal, Chad Woolley, Bernd Worsch, and numerous others.

Please check the EasyMock home page for new versions, and send bug reports and suggestions to the EasyMock Yahoo!Group. If you would like to subscribe to the EasyMock Yahoo!Group, send a message to easymock-subscribe@yahoogroups.com.

Release Notes

EasyMock Version 2.3 (July 9 2007)

Changes since 2.2:

- French documentation
- Matchers for Comparable parameters
- Decimal comparison fix
- Mock Objects can now be named
- Include Bill Michell's ThreadLocal fix
- Converted EasyMock's unit tests to JUnit 4

Changes since 2.1:

- answers for expected calls can now be created at call time via `andAnswer(IAnswer answer)` and `andStubAnswer(IAnswer answer)`

- `callback(Runnable runnable)` has been removed, for callbacks, please switch to `andAnswer(IAnswer answer)` and `andStubAnswer(IAnswer answer)`
- `replay()`, `verify()` and `reset()` now accept multiple mock objects as arguments

Changes since 2.0:

- arguments passed to the mock object are now available in callbacks via `EasyMock.getCurrentArguments()`
- fixed bug reported in http://groups.yahoo.com/group/easymock/message/558
- earlier failing if unused matchers were specified

Changes since 1.2:

- support for flexible, refactoring-safe argument matchers
- no mock control is needed for single Mock Objects
- stub behavior replaces default behavior
- support for call order checking for more than one mock, and to switch order checking on and off
- support for callbacks
- EasyMock now throws `java.lang.AssertionError` instead of `junit.framework.AssertionFailedError` so that it is now independent from the testing framework, you may use it with JUnit 3.8.x, JUnit 4 and TestNG
- deprecated old API

Test Coverage

EasyMock always keeps 100% test coverage. Clover reports are available here.