Easier testing with EasyMock

Imitate interfaces, classes, and exceptions with an open source mock-object framework

Skill Level: Intermediate

Elliotte Rusty Harold (elharo@ibiblio.org)
Software Engineer
Cafe au Lait

28 Apr 2009

Join Elliotte Rusty Harold for a look at some hard unit tests made easy through mock objects — more specifically, the EasyMock framework. This open source library saves you time and helps make your mock-object code concise and legible.

Test-driven development is a critical component of software development. If code isn't tested, it's broken. All code must be tested, and ideally the tests should be written before the model code is. But some things are easier to test than others. If you're writing a simple class to represent currency, it's easy to test that you can add $1.23 to $2.28 and get $4.03 and not $3.03 or $4.029999998. It's not much harder to test that it's impossible to create a currency such as $7.465. But how do you test the method that converts $7.50 to €5.88 — especially when the exchange rate is found by connecting to a live database with information that's updated every second? The correct result of `amount.toEuros()` can change every time you run the program.

The answer is *mock objects*. Instead of connecting to a real server that provides up-to-the-minute exchange-rate information, the test connects to a mock server that always returns the same exchange rate. Then you have a predictable result that you can test. After all, the goal is to test the logic in the `toEuros()` method, not whether the server is sending the correct values. (Let the developers who built the server worry about that.) This kind of mock object is sometimes called a *fake*.

Mock objects can also be useful for testing error conditions. For example, what happens if the `toEuros()` method tries to retrieve the latest exchange rate, but the

network is down? You could unplug the Ethernet cable from your computer and then run your test, but it's a lot less labor-intensive to write a mock object that simulates a network failure.

Mock objects can also be used to spy on the behavior of a class. By placing assertions inside the mock code, you can verify that the code under test is passing the correct arguments to its collaborators at the right time. A mock can let you see and test private parts of a class without exposing them through otherwise unnecessary public methods.

Finally, mock objects help remove large dependencies from a test. They make tests more unitary. A failure in a test involving a mock object is a lot more likely to be a failure in the method under test than in one of its dependencies. This helps isolate the problem and makes debugging simpler.

EasyMock is an open source mock object library for the Java programming language that helps you quickly and easily create mock objects for all these purposes. Through the magic of dynamic proxies, EasyMock enables you to create a basic implementation of any interface with just one line of code. By adding the EasyMock class extension, you can create mocks for classes too. These mocks can be configured for any purpose, ranging from simple dummy arguments for filling out a method signature to multi-invocation spies that verify a long sequence of method calls.

Introducing EasyMock

I'll start with a concrete example to demonstrate how EasyMock works. Listing 1 is the hypothesized `ExchangeRate` interface. Like any interface, it simply says what an instance does without specifying how it does it. For instance, it doesn't say whether the exchange-rate data comes from Yahoo finance, the government, or elsewhere.

Listing 1. ExchangeRate

```
import java.io.IOException;

public interface ExchangeRate {

    double getRate(String inputCurrency, String outputCurrency) throws IOException;

}
```

Listing 2 is the skeleton of the putative `Currency` class. It's actually fairly complex, and it might well contain bugs. (I'll spare you the suspense: there are bugs — quite a few in fact.)

Listing 2. Currency class

```
import java.io.IOException;
```

```
public class Currency {

    private String units;
    private long amount;
    private int cents;


    public Currency(double amount, String code) {
        this.units = code;
        setAmount(amount);
    }

    private void setAmount(double amount) {
        this.amount = new Double(amount).longValue();
        this.cents = (int) ((amount * 100.0) % 100);
    }

    public Currency toEuros(ExchangeRate converter) {
        if ("EUR".equals(units)) return this;
        else {
            double input = amount + cents/100.0;
            double rate;
            try {
                rate = converter.getRate(units, "EUR");
                double output = input * rate;
                return new Currency(output, "EUR");
            } catch (IOException ex) {
                return null;
            }
        }
    }

    public boolean equals(Object o) {
        if (o instanceof Currency) {
            Currency other = (Currency) o;
            return this.units.equals(other.units)
                    && this.amount == other.amount
                    && this.cents == other.cents;
        }
        return false;
    }

    public String toString() {
        return amount + "." + Math.abs(cents) + " " + units;
    }

}
```

Something important about the design of the `Currency` class might not be obvious at first glance. The exchange rate is passed in from *outside* the class. It is not constructed inside the class. This is critical to enable me to mock out the exchange rate so the tests can run without talking to the real exchange-rate server. It also enables client applications to supply different sources of exchange-rate data.

Listing 3 demonstrates a JUnit test that verifies that $2.50 is converted into €3.75 when the exchange rate is 1.5. EasyMock is used to create an `ExchangeRate` object that always supplies the value 1.5.

Listing 3. CurrencyTest class

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

```
import java.io.IOException;

public class CurrencyTest extends TestCase {

    public void testToEuros() throws IOException {
        Currency testObject = new Currency(2.50, "USD");
        Currency expected = new Currency(3.75, "EUR");
        ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
        EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
        EasyMock.replay(mock);
        Currency actual = testObject.toEuros(mock);
        assertEquals(expected, actual);
    }

}
```

Truth be told, Listing 3 failed the first time I ran it, as tests are wont to do. However, I fixed that bug before including the example here. This is why we do TDD.

Run this test and it passes. What happened? Let's look at the test line by line. First the test object and the expected result are constructed:

```
Currency testObject = new Currency(2.50, "USD");
Currency expected = new Currency(3.75, "EUR");
```

Nothing new yet.

Next I create a mock version of the `ExchangeRate` interface by passing the `Class` object for that interface to the static `EasyMock.createMock()` method:

```
ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
```

This is by far the weirdest part. Notice that at no point did I write a class that implements the `ExchangeRate` interface. Furthermore, there is absolutely no way the `EasyMock.createMock()` method can be typed to return an instance of `ExchangeRate`, a type it never knew about and that I created just for this article. And even if it did by some miracle return `ExchangeRate`, what happens when I need to mock an instance of a different interface?

The first time I saw this I almost fell out of my chair. I did not believe this code could possibly compile, and yet it did. There's deep, dark magic here, coming from a combination of Java 5 generics and dynamic proxies introduced way back in Java 1.3 (see Resources). Fortunately, you don't need to understand how it works to use it (and to be wowed by the cleverness of the programmers who invented these tricks).

The next step is just as surprising. To tell the mock what to expect, I invoke the method as an argument to the `EasyMock.expect()` method. Then I invoke `andReturn()` to specify what should come out as a result of calling this method:

```
EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
```

EasyMock records this invocation so it knows what to play back later.

> If you forget to call `EasyMock.replay()` before using a mock, you'll get an `IllegalStateException` with the not especially helpful error message: `missing behavior definition for the preceding method call`.

Next I get the mock ready to play back its recorded data by invoking the `EasyMock.replay()` method:

```
EasyMock.replay(mock);
```

This is the one piece of the design I find a little confusing. `EasyMock.replay()` does not actually replay the mock. Rather, it resets the mock so that the next time its methods are called it will begin replaying.

Now that the mock is prepared, I pass it as an argument to the method under test:

> **Mocking classes**
>
> Mocking out classes is harder from an implementation perspective. You can't create a dynamic proxy for a class. The standard EasyMock framework does not support mocks of classes. However, the EasyMock class extension uses bytecode manipulation to produce the same effect. The patterns in your code are almost exactly the same. Just import `org.easymock.classextension.EasyMock` instead of `org.easymock.EasyMock`. Class mocking also gives you the option to replace some of the methods in a class with mocks while leaving others intact.

```
Currency actual = testObject.toEuros(mock);
```

Finally, I verify that the answer is as expected:

```
assertEquals(expected, actual);
```

And that's all there is to it. Any time you have an interface that needs to return certain results for purposes of testing, you can just create a quick mock. It really is that easy. The `ExchangeRate` interface was small and simple enough that I could have easily written the mock class manually. However, the larger and more complex an interface becomes, the more onerous it is to write individual mocks for each unit test. EasyMock lets you create implementations of large interfaces like `java.sql.ResultSet` or `org.xml.sax.ContentHandler` in one line of code,

and then supply them with just enough behavior to run your tests.

Testing exceptions

One of the more popular uses of mocks is to test exceptional conditions. For example, you can't easily create a network failure on demand, but you can create a mock that imitates one.

The `Currency` class is supposed to return `null` when `getRate()` throws an `IOException`. Listing 4 tests this:

Listing 4. Testing that a method throws the right exception

```
public void testExchangeRateServerUnavailable() throws IOException {
   Currency testObject = new Currency(2.50, "USD");
   ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
   EasyMock.expect(mock.getRate("USD", "EUR")).andThrow(new IOException());
   EasyMock.replay(mock);
   Currency actual = testObject.toEuros(mock);
   assertNull(actual);
}
```

The new piece here is the `andThrow()` method. As you probably guessed, this simply sets the `getRate()` method to throw the specified exception when invoked.

You can throw any kind of exception you like — checked, runtime, or error — as long as the method signature supports it. This is especially helpful for testing extremely unlikely conditions (out-of-memory error or class def not found, for example) or conditions that indicate virtual machine bugs (such as no UTF-8 character encoding available).

Setting expectations

EasyMock doesn't just provide canned answers in response to canned input. It can also check that the input is what it's supposed to be. For example, suppose the `toEuros()` method had the bug shown in Listing 5, where it's returning a result in euros but getting the exchange rate for Canadian dollars. This could make or lose someone a lot of money.

Listing 5. A buggy toEuros() method

```
public Currency toEuros(ExchangeRate converter) {
    if ("EUR".equals(units)) return this;
    else {
        double input = amount + cents/100.0;
        double rate;
        try {
            rate = converter.getRate(units, "CAD");
            double output = input * rate;
            return new Currency(output, "EUR");
```

```
        } catch (IOException e) {
            return null;
        }
    }
}
```

However, I don't need an additional test for this. Listing 4's `testToEuros` will already catch this bug. When you run this test with the buggy code in Listing 4, the test fails with this error message:

```
"java.lang.AssertionError:
  Unexpected method call getRate("USD", "CAD"):
    getRate("USD", "EUR"): expected: 1, actual: 0".
```

Notice that this is not an assertion I made. EasyMock noticed that the arguments I was passing didn't add up and flunked the test case.

By default, EasyMock only allows the test case to call the methods you specify with the arguments you specify. Sometimes this is a little too strict though, so there are ways to loosen this up. For example, suppose I did want to allow any string to be passed to the `getRate()` method, rather than just `USD` and `EUR`. Then I could specify that I expect `EasyMock.anyObject()` instead of the explicit strings, like so:

```
EasyMock.expect(mock.getRate(
       (String) EasyMock.anyObject(),
       (String) EasyMock.anyObject())).andReturn(1.5);
```

I can be a little pickier and specify `EasyMock.notNull()` to allow only non-`null` strings:

```
EasyMock.expect(mock.getRate(
        (String) EasyMock.notNull(),
        (String) EasyMock.notNull())).andReturn(1.5);
```

Static type checking will prevent non-`Strings` from being passed to this method. However, now I allow `Strings` besides `USD` and `EUR`. You can use regular expressions to be even more explicit with `EasyMock.matches()`. Here I require a three-letter, upper-case ASCII `String`:

```
EasyMock.expect(mock.getRate(
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"),
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"))).andReturn(1.5);
```

Using `EasyMock.find()` instead of `EasyMock.matches()` would accept any `String` that contained a three-capital-letter sub-`String`.

EasyMock has similar methods for the primitive data types:

- `EasyMock.anyInt()`
- `EasyMock.anyShort()`
- `EasyMock.anyByte()`
- `EasyMock.anyLong()`
- `EasyMock.anyFloat()`
- `EasyMock.anyDouble()`
- `EasyMock.anyBoolean()`

For the numeric types, you can also use `EasyMock.lt(x)` to accept any value less than `x`, or `EasyMock.gt(x)` to accept any value greater than `x`.

When checking a long sequence of expectations, you can capture the results or arguments of one method call and compare it to the value passed into another method call. And finally, you can define custom matchers that check pretty much any detail about the arguments you can imagine, though the process to do so is somewhat complex. However, for most tests the basic matchers like `EasyMock.anyInt()`, `EasyMock.matches()`, and `EasyMock.eq()` suffice.

Strict mocks and order checking

EasyMock doesn't just check that expected methods are called with the right arguments. It can also verify that you call those methods and only those methods, in the right order. This checking is not performed by default. To turn it on, call `EasyMock.verify(mock)` at the end of your test method. For example, Listing 6 will now fail if the `toEuros()` method invokes `getRate()` more than once:

Listing 6. Check that getRate() is called only once

```
public void testToEuros() throws IOException {
    Currency expected = new Currency(3.75, "EUR");
    ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);
    EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);
    EasyMock.replay(mock);
    Currency actual = testObject.toEuros(mock);
    assertEquals(expected, actual);
    EasyMock.verify(mock);
}
```

Exactly how much such checking `EasyMock.verify()` does depends on which of the available modes it operates in:

- **Normal — `EasyMock.createMock()`**: All of the expected methods must be called with the specified arguments. However, the order in which these methods are called does not matter. Calls to unexpected methods

cause the test to fail.

- **Strict — `EasyMock.createStrictMock()`**: All expected methods must be called with the expected arguments, in a specified order. Calls to unexpected methods cause the test to fail.
- **Nice — `EasyMock.createNiceMock()`**: All expected methods must be called with the specified arguments in any order. Calls to unexpected methods do *not* cause the test to fail. Nice mocks supply reasonable defaults for methods you don't explicitly mock. Methods that return numbers return `0`. Methods that return booleans return `false`. Methods that return objects return `null`.

Checking the order and number of times a method is called is even more useful in larger interfaces and larger tests. For example, consider the `org.xml.sax.ContentHandler` interface. If you were testing an XML parser, you'd want to feed in documents and verify that the parser called the right methods in `ContentHandler` in the right order. For example, consider the simple XML document in Listing 7:

Listing 7. A simple XML Document

```
<root>
  Hello World!
</root>
```

According to the SAX specification, when the parser parses this document, it should call these methods in this order:

1. `setDocumentLocator()`
2. `startDocument()`
3. `startElement()`
4. `characters()`
5. `endElement()`
6. `endDocument()`

However, just to make matters interesting, the call to `setDocumentLocator()` is optional; parsers are allowed to call `characters()` more than once. They don't need to pass the maximum contiguous run of text in a single call, and in fact most don't. This is difficult to test with traditional methods, even for a simple document like Listing 7, but EasyMock makes it straightforward, as shown in Listing 8:

Listing 8. Testing an XML parser

```
import java.io.*;
import org.easymock.EasyMock;
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;
import junit.framework.TestCase;

public class XMLParserTest extends TestCase {

    private  XMLReader parser;

    protected void setUp() throws Exception {
        parser = XMLReaderFactory.createXMLReader();
    }

    public void testSimpleDoc() throws IOException, SAXException {
        String doc = "<root>\n  Hello World!\n</root>";
        ContentHandler mock = EasyMock.createStrictMock(ContentHandler.class);

        mock.setDocumentLocator((Locator) EasyMock.anyObject());
        EasyMock.expectLastCall().times(0, 1);
        mock.startDocument();
        mock.startElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"),
                (Attributes) EasyMock.anyObject());
        mock.characters((char[]) EasyMock.anyObject(),
                EasyMock.anyInt(), EasyMock.anyInt());
        EasyMock.expectLastCall().atLeastOnce();
        mock.endElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"));
        mock.endDocument();
        EasyMock.replay(mock);

        parser.setContentHandler(mock);
        InputStream in = new ByteArrayInputStream(doc.getBytes("UTF-8"));
        parser.parse(new InputSource(in));

        EasyMock.verify(mock);
    }
}
```

This test demonstrates several new tricks. First, it uses a strict mock so that order is required. You wouldn't want the parser calling `endDocument()` before `startDocument()`, for instance.

Second, the methods I'm testing all return `void`. That means I can't pass them as arguments to `EasyMock.expect()`, as I did with `getRate()`. (EasyMock fools the compiler about a lot of things, but it isn't quite smart enough to fool the compiler into believing that `void` is a legal argument type.) Instead, I just invoke the void method on the mock, and EasyMock captures the result. If I need to change some detail of the expectation, then I call `EasyMock.expectLastCall()` immediately after invoking the mock method. Also, notice that you can't just pass any old `Strings` and `ints` and arrays as the expectation arguments. All of these must be wrapped with `EasyMock.eq()` first so their values can be captured in the expectation.

Listing 8 uses `EasyMock.expectLastCall()` to adjust the number of times methods are expected. By default, methods are expected once each. However, I make `setDocumentLocator()` optional by invoking `.times(0, 1)`. This says the method must be called between 0 and 1 times. Of course, you can change these arguments to expect methods to be called 1 to 10 times, 3 to 30 times, or any other range you like. For `characters()`, I really don't know how many times it will be

called, except that it must be called at least once, so I expect it `.atLeastOnce()`. If this were a non-`void` method, I could have applied `times(0, 1)` and `atLeastOnce()` to the expectations directly. However, because the methods being mocked return `void`, I must target them with `EasyMock.expectLastCall()` instead.

Finally, notice the use of `EasyMock.anyObject()` and `EasyMock.anyInt()` for the arguments to `characters()`. This accounts for the many different ways a parser is allowed to pass text into a `ContentHandler`.

Mocks and reality

Is EasyMock worth it? It does nothing that couldn't be done by manually writing mock classes, and in the case of manually written classes your project would be a dependency or two leaner. For instance, Listing 3 is one case where a manually written mock using an anonymous inner class could be almost as compact and might be more legible to developers who aren't yet familiar with EasyMock. However, it's a deliberately short example for the purpose of an article. When mocking larger interfaces such as `org.w3c.dom.Node` (25 methods) or `java.sql.ResultSet` (139 methods and growing), EasyMock is a huge time saver that produces much shorter and more legible code at a minimal cost.

Now a word of caution: mock objects can be taken too far. It is possible to mock out so much that a test always passes even when the code is seriously broken. The more you mock, the less you're testing. Many bugs exist in dependent libraries and in the interactions between one method and the methods it calls. Mocking out dependencies can hide a lot of bugs you'd really rather find. Mocks should not be your first choice in any situation. If you can use the real dependency, do so. A mock is an inferior replacement for the real class. However, if you can't reliably and automatically test with the real class for any reason, then testing with a mock is infinitely superior to not testing at all.

Resources

Learn

- Dynamic Proxy Classes: EasyMock is implemented using the dynamic-proxy classes introduced way back in Java 1.3.
- "Unit testing with mock objects" (Alexander Chaffee and William Pietri, developerWorks, November 2002): A general introduction to the concept of mock objects using manually written mocks.
- "Endo-Testing: Unit Testing with Mock Objects" (Tim Mackinnon, Steve Freeman and Philip Craig, 2000): This paper, presented at the XP2000 conference, coined the term *mock objects*.
- "*Evolutionary architecture and emergent design*: Test-driven design, Part 1" (Neal Ford, developerWorks, February 2009): Find out how test-driven development can improve the overall design of your code.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- EasyMock: Install the EasyMock system.
- cglib 2.1: An open source byte code manipulation library you'll need to install to use the EasyMock class extension.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Elliotte Rusty Harold
Elliotte Rusty Harold is originally from New Orleans, to which he returns periodically in search of a decent bowl of gumbo. However, he resides in the University Town Center neighborhood of Irvine with his wife Beth and cats Charm (named after the quark) and Marjorie (named after his mother-in-law). His Cafe au Lait Web site has become one of the most popular independent Java sites on the Internet, and his spin-off site, Cafe con Leche, has become one of the most popular XML sites. His most recent book is *Refactoring HTML*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.