EasyMock: Dynamic Mock Objects for JUnit

Tammo Freese
OFFIS
Escherweg 2
26121 Oldenburg
Germany
+49 441 9722 215
tammo.freese@offis.de

ABSTRACT
In Extreme Programming, unit testing is an integral activity of everyday software development. For isolating units in the tests, Mock Objects are often used to simulate collaborators of the units under test. However, writing and maintaining Mock Objects may become a tedious task. This paper presents the Java library EasyMock that dynamically generates Mock Objects for interfaces. This moves the specification of the Mock Objects into the test methods, avoids implementation mistakes, and eases refactoring.

Keywords
JUnit, Java, Mock Object, EasyMock, unit testing, test-driven development, refactoring

1 UNIT TESTING AND MOCK OBJECTS

Unit testing is the testing of program units in isolation. In practice, a unit often does not work in isolation, as it relies on other units. For unit testing, these collaborating units have to be simulated and controlled from within the test.

Mock Objects are replacements for domain code that emulate real behavior. In addition to this stub functionality, they verify assertions about their usage [5]. Every method call on the Mock Object is checked whether it was set as expectation before. After using the Mock Object, the method `verify()` checks whether the defined behavior has been used by the object under test.

The Mock Objects package [6] contains classes which ease the implementation of Mock Objects, and it provides ready-to-use Mock Object implementations for several APIs.

2 TEST DRIVEN DEVELOPMENT WITH JUNIT

Extreme Programming [1] does not only focus on unit testing, but advocates test-driven development. Every change of the code's observable behavior has to be motivated by a failing test.

JUnit [4] is a regression testing framework for Java written by Erich Gamma and Kent Beck. Using JUnit has several benefits. As the tests are written in Java, testing does not break the programming session, but integrates into it. A couple of assertions eases the construction of the tests. The tests may be composed into test hierarchies, and graphical and text interfaces provide immediate feedback.

Test-driven development includes refactoring, a technique of small, behavior-preserving steps which improve the design of the code [3].

As test-driven development is a central part of Extreme Programming, tools and techniques should not hinder it.

3 MOCK OBJECTS EXAMPLE

To show an implementation of a Mock Object and its usage with JUnit, we provide a test case for a search algorithm on a simple storage. Listing 1 shows the `Storage` interface. It has a method that returns a list of page titles as well a method that returns the page for a given title.

We want to test a class `Searcher` that finds all pages in a storage where the title contains a given string. Listing 2 shows a test case that uses a Mock Object to test the searcher in isolation. First, the Mock Object is created (Listing 2, part 1). Our test checks whether the search is able to find a single page. So we configure the Mock Object to return one page title when asked for the list of page titles, and to return a certain page when asked for this title (Listing 2, part 2). Then a searcher is configured to use the Mock Object as storage, and a search is performed (Listing 2, part 3). It is expected to return the page that our mock storage contains (Listing 2, part 4). Finally, we use the `verify()` method on the Mock Object to check whether the defined behavior has been used (Listing 2, part 5).

The implementation of the Mock Object is shown in Listing 3. To store the expectation parameter for the method `getPage()`, the class `ExpectationValue` from the Mock Objects package [6] is used.

Such expectation classes allow adding expected and actual values. If an actual value is added that was not specified as expectation, the expectation class throws an exception. Additionally, the `verify()` method of each expectation class verifies that all expected values have been used. Otherwise, it throws an exception.

As our Mock Object only verifies that the `getPage()` parameter was called, the `verify()` method of the Mock Object delegates to the `verify()` method of the expectation `getPage`.

```
public interface Storage {
  Page getPage(String name);
  String[] getPageNames();
}
```

Listing 1. Storage interface

```
public void testSearcher() {

  // (1)
  MockStorage mockStorage
    = new MockStorage();

  // (2)
  String[] pageNames=new String[] {"title"};
  Page page = new Page();

  mockStorage.setupPageNamesReturnValue
      (pageNames);
  mockStorage.setGetPageParameter("title");
  mockStorage.setGetPageReturnValue(page);

  // (3)
  Searcher searcher = new Searcher();
  searcher.setStorage(mockStorage);
  Page[] result = searcher.find("itl");

  // (4)
  assertEquals(1, result.length);
  assertEquals(page, result[0]);

  // (5)
  mockStorage.verify();
}
```

Listing 2. Test case using an implemented Mock Object

4 PROBLEMS WITH MOCK OBJECTS

Although Mock Objects are a big step towards isolated testing of program units, there are some disadvantages when they are implemented manually. Writing Mock Objects is not very difficult, but it is a tedious task that sometimes introduces errors. And as they are separate classes, their code has to be read to understand the test.

Additional issues arise with respect to test-driven development and refactoring. The interface that the unit under test uses is neither directly visible nor used in the test case. It is only visible as a substring in the Mock Object's name.

If we add a method to the interface, we have to implement it in the mock object before our code may be compiled again. When we change a method name, we have to change the names of related expectation setting methods, too. When we delete a method, we have to care about deleting it on the mock object and erasing its occurrences.

When we introduce a parameter object, we have to change the expectation handling inside the Mock Object. And as a Mock Object is often used in several test cases, it is difficult to see which of the Mock Object's methods are still used.

These problems are not valid for standard API interfaces, as they do not change often, if at all. However, they are valid for iterative development, where interfaces inside our applications do change quite often. So the usage of manually implemented Mock Objects may hinder refactoring.

```
import com.mockobjects.ExpectationValue;

class MockStorage implements Storage {
  private Page pageReturnValue;
  private ExpectationValue getPage
   = new ExpectationValue("getPage");
  private String[] pageNamesReturnValue;

  // interface methods
  public Page getPage(String name){
    getPage.setActual(name);
    return pageReturnValue;
  }
  public String[] getPageNames() {
    return pageNamesReturnValue;
  }

  // verify usage
  public void verify() {
    getPage.verify();
  }

  // methods that define the behavior
  public void setGetPageParameter(String n){
    getPage.setExpected(n);
  }
  public void setGetPageReturnValue(Page p){
    pageReturnValue = p;
  }
  public void setupPageNamesReturnValue
      (String[] n) {
    pageNamesReturnValue = n;
  }
}
```

Listing 3. Implemented Mock Object

5 EASYMOCK EXAMPLE

The EasyMock library [2] provides simple Mock Objects without having to implement them. They are generated dynamically at runtime.

The definition of the Mock Object behavior in EasyMock is different to that using an implemented Mock Object. As an example, Listing 4 shows a version of Listing 2 adapted to use the EasyMock library. We will now explain all the changes step by step.

As EasyMock builds Mock Objects at runtime, there cannot be any additional implementation for the Mock Objects, and methods like `verify()` cannot be defined on the Mock Object itself. So the Mock Object is split in two parts: The EasyMock Mock Object and the EasyMock Mock Control. The EasyMock Mock Object (in short: the mock) is an implementation of the interface to simulate. In contrast to a Mock Object, it has no additional methods.

The EasyMock Mock Control (in short: the control) has methods for controlling its associated mock.

To get a Mock Object for our `Storage`, first we obtain a control via a factory method `mockControlFor()` on `EasyMock`:

```
MockControl control
= EasyMock.mockControlFor(Storage.class);
```

The mock is returned by the control:

```
Storage mockStorage
= (Storage) control.getMock();
```

To define a method call as an expectation, we simply call the method on the mock, and define the return value via the control:

```
mockStorage.getPageNames();
control.setReturnValue(pageNames);
```

We repeat this for every behavior that we want to specify:

```
mockStorage.getPage("title");
control.setReturnValue(page);
```

As the mock is used to record the expected method calls, it does not behave like a Mock Object yet. To switch to the Mock Object behavior, we have to activate the mock via its control:

```
control.activate();
```

Now we may use the mock in our test, just like an implemented Mock Object. So parts 3 and 4 of the test case in Listing 2 do not change.

As the mock only implements the interface, the `verify()` method is not implemented there, but on the control. So part 5 of Listing 2 changes from

```
mock.verify();
```

to

```
control.verify();
```

The test case got a little bit longer than before. To define the Mock Object, we need two lines of code instead of one, and we need the activation. However, by adding these two lines of code, we have defined the Mock Object directly within the test case. We do not need the Mock Object implementation shown in Listing 3 anymore.

6 MORE EASYMOCK FEATURES

EasyMock allows more than setting return values for each method call.

As a first example, we show how to set up the mock to throw a `Throwable` instead of returning a value:

```
mock.getPage(title);
control.setThrowable(new Error());
```

If we expect a method call to occur not only once, we can specify the behavior for any number of calls. In simple cases where the return value does not change, we specify the number of expected calls as a second parameter to the return value definition:

```
mock.getPage(title);
control.setReturnValue(page, 3);
```

A similar approach is available for `Throwables`:

```
mock.getPage(title);
control.setThrowable(new Error(), 3);
```

A call to a `void` method may be expected to happen a defined number of times, too:

```
mock.update(); // a void method
control.setVoidCallable(2);
```

If `setVoidCallable()` is omitted, exactly one call is expected. Therefore an alternative way to specify the same behavior as in the preceding example is:

```
mock.update();
mock.update();
```

In some cases, we want our mock to behave differently on subsequent calls. As an example, we configure our mock object to throw an error two times, and to return the page the next three times:

```
mock.getPage(title);
control.setThrowable(new Error(), 2);
mock.getPage(title);
control.setReturnValue(page, 3);
```

There is a shortcut for this definition. As every behavior definition on the control is mapped to the last method call on the mock, the second call on the mock may be omitted:

```
mock.getPage(title);
control.setThrowable(new Error(), 2);
control.setReturnValue(page, 3);
```

Setting a return value or `Throwable` without specifying a number of times is interpreted as expecting it for exactly one call. If we only care about a method call to happen, but not about the number of calls, we may use

```
control.setReturnValue(page,
  MockControl.ONE_OR_MORE_CALLS)
```

The method call is now allowed to happen an unlimited number of times, but if it is never used, `verify()` will fail.

`MockControl.ONE_OR_MORE_CALLS` is also the key to use EasyMock for the definition of stubs where we do not care about how often, if at all, the defined behavior is used. To define a stub, `MockControl.ONE_OR_MORE_CALLS` has to be used in all behavior definitions, and `verify()` must not be called at the end of the test.

Finally, it is possible to reset a mock to its initial state for reusing it in several test cases:

```
control.reset();
```

```
public void testSearcher() {

  //(1)
  MockControl control
   = EasyMock.mockControlFor(Storage.class);
  Storage mockStorage
   = (Storage) control.getMock();

  //(2)
  String[] pageNames=new String[] {"title"};
  Page page = new Page();

  mockStorage.getPageNames();
  control.setReturnValue(pageNames);
  mockStorage.getPage("title");
  control.setReturnValue(page);

  control.activate();

  //(3)
  Searcher searcher = new Searcher();
  searcher.setStorage(mockStorage);
  Page[] result = searcher.find("itl");

  //(4)
  assertEquals(1, result.length);
  assertEquals(page, result[0]);

  //(5)
  control.verify();
}
```

Listing 4. Test case using EasyMock

7 EXPERIENCES

Our experiences using EasyMock are positive. The time for implementing the Mock Objects is saved, and implementation errors are avoided. All the information needed to understand a test case is available in the test code itself.

EasyMock is ideal for often-changing interfaces inside the application, as it handles changes to the interface quite well.

As the behavior definition is fixed, we cannot use EasyMock in all cases. So we also use implemented Mock Objects where appropriate.

The only drawback of EasyMock is that tests using it are harder to read than tests using implemented Mock Objects, as there are no self-explaining method names for the behavior definitions.

8 RELATED WORK

As mentioned in the previous section, EasyMock cannot be used in all cases. Whenever more freedom and special checks are needed, either a manual implementation or reusing an existing Mock Object is recommended. For the implementations, the Mock Objects library [6] should be used.

While EasyMock generates Mock Objects at runtime, it is also possible to generate Mock Objects as source code. At the time of writing, there are two code generators for Mock Objects. MockMaker [8] has a command line interface as well as a GUI. MockCreator [6] integrates into Visual Age for Java.

Code generation shares some advantages with EasyMock: Implementation errors are avoided, and as the behavior of the generated implementation is known, all the information that is needed to understand a test case is provided in the test method itself.

However, code generation has some disadvantages regarding refactoring. If a method is renamed, all the expectation setting methods will be renamed in the next generation steps, too. As the test code uses the old names of the expectation setting methods, the code won't even compile.

9 FUTURE WORK

At the time of writing, EasyMock (internal release 0.85) is only able to handle interfaces, and it does only work with Java version 1.3.1 and above.

Our future plans are to allow EasyMocks for classes, and to provide an sequence check for the method calls.

ACKNOWLEDGEMENTS

Thanks to Tim Mackinnon, Steve Freeman and Philip Craig for the Mock Objects idea. Without it, EasyMock would not exist. And thanks to all EasyMock users for their valuable feedback. Finally, thanks to Frank Westphal and Jürgen Schlegelmilch for their valuable comments on this paper.

REFERENCES

1. Beck, Kent. *Extreme Programming Explained.* Addison-Wesley, 2000.
2. EasyMock home page. http://www.easymock.org.
3. Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
4. JUnit home page. http://www.junit.org.
5. Mackinnon, T., Freeman, S., and Craig, P. Endo-Testing: Unit Testing with Mock Objects. In*: Extreme Programming Examined*, pages 287-301. Addison-Wesley, 2001.
6. Mock Objects home page. http://www.mockobjects.com.
7. MockCreator home page. http://www.abstrakt.de/en/mockcreator.html
8. MockMaker home page. http://mockmaker.sourceforge.net.