Mock Controls with EasyMock

Most people who have used EasyMock before are familiar with the types of mock objects. There is the basic mock object, created with EasyMock.createMock(MyInterface.class). There is the strict mock that validates both the method calls and the order they occur. Finally there is the nice mock which returns a basic default (false, 0 or null) if an unexpected method call occurs. However, most developers I have come across don't use mock controls much. This tutorial will introduce you to the concepts of mock controls and show you an example of where testing without them may not be as safe as you think.

The Example

Mock objects validate the execution of method calls across a single object. However what if you have a scenario where the order of method calls matters across objects? Take the example below:

```
...
public void runReport(int userId) {
    // load values by userId and where Active Flag = Y
    ReportRecords records = reportDAO.getActiveUsersByUserID(userId);

    // generate report based on the records returned
    processReport(records);

    // Removes users after reports have been run
    userDAO.deleteUserByUserID(userId);
}
...
```

If I went to refactor the above method and switched the calls to

```
...
public void runReport(int userId) {
    ReportRecords records = null;

    // Removes users after reports have been run
    userDAO.deleteUserByUserID(userId);

    // generate report based on the records returned
    processReport(records);

    // load values by userId and where Active Flag = Y
    records = reportDAO.getActiveUsersByUserID(userId);
}
...
```

the report would no longer work because we would delete the users before running the report on them (nevermind the NullPointerException that may be thrown from processReport). In situations like the one above, the order of method calls across objects matters. So how do we address this issue with EasyMock? Enter mock controls.

The Wrong Way To Test

Mock controls allow you to couple mock objects together. Now, coupling is usually considered a bad thing in software development. However, in this case, we need to have a relationship between mock objects so that we can establish who's methods were called first. Let's look at a test case for the above scenario without mock controls first.

```
...
public void testRunReport() {
    // class to be tested
    ReportRunner runner = new ReportRunner();

    // create and inject our mock objects
    UserDAO mockUserDao = EasyMock.createMock(UserDao.class);
    ReportDAO mockReportDao = EasyMock.createMock(ReportDAO.class);
    runner.setUserDao(mockUserDao);
    runner.setReportDao(mockReportDao);

    // record our behaviour
    expect(reportDAO.getActiveusersByUserID(123)).andReturn(new ReportRecords());
    userDao.deleteUserByUserID(123);

    // replay our behaviour for the test
    EasyMock.replay(reportDao, userDao);

    // run the test
    runner.runReport(123);

    // verify the results of the test
    EasyMock.verify(reportDao, userDao);
}
...
```

Now if we actually try the above test case, we will find that it will pass for the first example. The issue arises when we try the second example. It still passes. This test doesn't truly verify that our refactoring hasn't broken existing code (one of the advantages of unit testing in the first place).

The Right Way To Test

Let's create a test case using mock controls to address the issue.

```
...
public void testRunReport() {
    // class to be tested
    ReportRunner runner = new ReportRunner();

    // create our mock control
    IMockControl mockMaker = EasyMock.createStrictControl();

    // create and inject our mock objects
    UserDAO mockUserDao = mockMaker.createMock(UserDao.class);
    ReportDAO mockReportDao = mockMaker.createMock(ReportDAO.class);
    runner.setUserDao(mockUserDao);
    runner.setReportDao(mockReportDao);

    // record our behaviour
    expect(reportDAO.getActiveusersByUserID(123)).andReturn(new ReportRecords());
    userDao.deleteUserByUserID(123);

    // replay our behavior for the test
    mockMaker.replay();

    // run the test
    runner.runReport(123);
```

```
    // verify the results of the test
    mockMaker.verify();
}
...
```

So what is different? In the new test case, we are using a strict mock control (mock controls have they same three types as mock objects: regular, nice and strict) to create our mock objects. By doing this, EasyMock will not only verify that the methods we record were executed, but in the correct order across objects. You should also notice that the recording of the behavior has not changed. That is all the same. Finally, instead of having to specify all the mock objects to replay and verify, our mock control kept track of what was recorded where so all we have to do is tell it to replay and verify.

When you try the new test case, it will pass for the first example, but fail for the second. Our refactoring safety net has been cast and we can refactor at will.

Conclusion

So when do you use mock controls vs mock objects? Mock controls are great for maintaining a relationship between objects. I personally also think the code is a bit cleaner given that you don't have to list out a long list of objects to replay and verify.

I hope this helps you in your unit testing endeavors. Questions and thoughts are always encouraged in the comments!