Unit testing with mock objects

Improve your unit tests by replacing your collaborators with mock objects

Alexander Chaffee
William Pietri

November 01, 2002

Mock objects are a useful way to write unit tests for objects that act as mediators. Instead of calling the real domain objects, the tested object calls a mock domain object that merely asserts that the correct methods were called, with the expected parameters, in the correct order. However, when the tested object must create the domain object, we are faced with a problem. How does the tested object know to create a mock domain object instead of the true domain object? In this article, software consultants Alexander Day Chaffee and William Pietri present a refactoring technique to create mock objects based on the factory method design pattern.

Unit testing has become widely accepted as a "best practice" for software development. When you write an object, you must also provide an automated test class containing methods that put the object through its paces, calling its various public methods with various parameters and making sure that the values returned are appropriate.

When you're dealing with simple data or service objects, writing unit tests is straightforward. However, many objects rely on other objects or layers of infrastructure. When it comes to testing these objects, it is often expensive, impractical, or inefficient to instantiate these collaborators.

For example, to unit test an object that uses a database, it may be burdensome to install, configure, and seed a local copy of the database, run your tests, then tear the local database down again. Mock objects provide a way out of this dilemma. A mock object conforms to the interface of the real object, but has just enough code to fool the tested object and track its behavior. For example, a database connection for a particular unit test might record the query while always returning the same hardwired result. As long as the class being tested behaves as expected, it won't notice the difference, and the unit test can check that the proper query was emitted.

Mock in the middle

The common coding style for testing with mock objects is to:
• Create instances of mock objects
• Set state and expectations in the mock objects
• Invoke domain code with mock objects as parameters
• Verify consistency in the mock objects

While this pattern is very effective for many cases, sometimes the mock object cannot be passed into the object being tested. Instead, that object is designed to either create, look up, or otherwise obtain its collaborator.

For instance, the tested object may need to obtain a reference to an Enterprise JavaBean (EJB) component or remote object. Or the tested object may make use of objects with side effects that may not be desirable in unit testing, like File objects that delete files.

Common wisdom suggests that this situation provides an opportunity to refactor your object to make it more test-friendly. For instance, you may change the method signature so that the collaborator object is passed in.

Nicholas Lesiecki points out that refactoring is not always desirable, nor does it always result in code that is cleaner or easier to understand. In many cases, changing the method signature so the collaborator becomes a parameter will result in a confusing, untested snarl of code inside the method's original callers.

The heart of the problem is that the object is obtaining these objects "on the inside." Any solution must apply to all occurrences of this creation code. To solve this problem, Lesiecki uses a lookup aspect or a creation aspect. In this solution, the code that performs the lookup is replaced automatically with code that returns a mock object instead.

Because AspectJ is not an option for some, we offer an alternative approach in this article. Because this is, at root, a refactoring, we will follow the presentation convention established by Martin Fowler in his seminal book, Refactoring: Improving the Design of Existing Code (see Related topics). (Our code is based on JUnit, the most popular unit testing framework for Java programming, though it is by no means JUnit-specific.)

Refactoring: Extract and override factory method

A refactoring is a code change that leaves the original functionality intact, but changes the design of the code so that it's cleaner, more efficient, and easier to test. This section offers a step-by-step description of the Extract and Override factory method refactoring.

Problem: The object being tested creates a collaborator object. This collaborator must be replaced with a mock object.

Code before refactoring

class Application {
    ...  
    public void run() {
        View v = new View();
        v.display();
    }
    ...
**Solution:** Extract the creation code into a factory method, override this factory method in a test subclass, and then make the overridden method return a mock object instead. Finally, if practical, add a unit test that requires the original object's factory method to return an object of the correct type:

**Code after refactoring**

```java
class Application {
    public void run() {
        View v = createView();
        v.display();
    }

    protected View createView() {
        return new View();
    }
}
```

This refactoring enables the unit test code shown in Listing 1:

**Listing 1. Unit test code**

```java
class ApplicationTest extends TestCase {
    MockView mockView = new MockView();

    public void testApplication() {
        Application a = new Application() {
            protected View createView() {
                return mockView;
            }
        };
        a.run();
        mockView.validate();
    }

    private class MockView extends View {
        boolean isDisplayed = false;

        public void display() {
            isDisplayed = true;
        }

        public void validate() {
            assertTrue(isDisplayed);
        }
    }
}
```

**Roles**

This design introduces the following roles played by objects in the system:

- **Target object:** The object being tested
- **Collaborator object:** The object created or obtained by the target
• **Mock object**: A subclass (or implementation) of the collaborator that follows the mock object pattern

• **Specialization object**: A subclass of the target that overrides the creation method to return a mock instead of a collaborator

**Mechanics**

A refactoring consists of many small, technical steps. Together, these are called the *mechanics*. If you follow the mechanics closely like a cookbook recipe, you should be able to learn the refactoring without much trouble.

1. Identify all occurrences of code that create or obtain the collaborator.
2. Apply the *Extract Method* refactoring to this creation code, creating the factory method (discussed on page 110 of Fowler's book; see the Related topics section for more information).
3. Assure that the factory method is accessible to the target object and its subclasses. (in the Java language, use the `protected` keyword).
4. In your test code, create a mock object implementing the same interface as the collaborator.
5. In your test code, create a specialization object that extends (specializes) the target.
6. In the specialization object, override the creation method to return a mock object that accommodates your test.
7. Optional: create a unit test to assure that the original target object's factory method still returns the correct, non-mock object.

**Example: ATM**

Imagine you are writing the tests for a bank's Automatic Teller Machine. One of those tests might look like Listing 2:

**Listing 2. Initial unit test, before mock object introduction**

```java
public void testCheckingWithdrawal() {
    float startingBalance = balanceForTestCheckingAccount();
    AtmGui atm = new AtmGui()
    insertCardAndInputPin(atm);
    atm.pressButton("Withdraw");
    atm.pressButton("Checking");
    atm.pressButtons("1", "0", "0", "0", "0");
    assertContains("$100.00", atm.getDisplayContents());
    atm.pressButton("Continue");
    assertEquals(startingBalance - 100,
    balanceForTestCheckingAccount());
}
```

In addition, the matching code inside the `AtmGui` class might look like Listing 3:
Listing 3. Production code, before refactoring

```java
private Status doWithdrawal(Account account, float amount) {
    Transaction transaction = new Transaction();
    transaction.setSourceAccount(account);
    transaction.setDestAccount(myCashAccount());
    transaction.setAmount(amount);
    transaction.process();
    if (transaction.successful()) {
        dispense(amount);
    }
    return transaction.getStatus();
}
```

This approach will work, but it has an unfortunate side effect: the checking account balance is lower than when the test started, making other testing more difficult. There are ways to solve that, but they all increase the complexity of the tests. Worse, this approach also requires three round trips to the system in charge of the money.

To fix this problem, the first step is to refactor `AtmGui` to allow us to substitute a mock transaction for the real transaction, as shown in Listing 4 (compare the boldface source code to see what we're changing):

Listing 4. Refactoring `AtmGui`

```java
private Status doWithdrawal(Account account, float amount) {
    Transaction transaction = createTransaction();
    transaction.setSourceAccount(account);
    transaction.setDestAccount(myCashAccount());
    transaction.setAmount(amount);
    transaction.process();
    if (transaction.successful()) {
        dispense(amount);
    }
    return transaction.getStatus();
}
```

```java
protected Transaction createTransaction() {
    return new Transaction();
}
```

Back inside the test class, we define the `MockTransaction` class as a member class, as shown in Listing 5:

Listing 5. Defining `MockTransaction` as a member class

```java
private MockTransaction extends Transaction {
    private boolean processCalled = false;

    // override process method so that no real work is done
    public void process() {
        processCalled = true;
        setStatus(Status.SUCCESS);
    }

    public void validate() {
        assertTrue(processCalled);
    }
}
```
And finally, we can rewrite our test so that the tested object uses the `MockTransaction` class rather than the real one, as shown in Listing 6.

**Listing 6. Using the MockTransaction class**

```java
MockTransaction mockTransaction;

public void testCheckingWithdrawal() {
    mockTransaction = new MockTransaction();

    AtmGui atm = new AtmGui() {
        protected Transaction createTransaction() {
            return mockTransaction;
        }
    };

    insertCardAndInputPin(atm);
    atm.pressButton("Withdraw");
    atm.pressButton("Checking");
    atm.pressButtons("1", "0", "0", "0", "0");
    assertContains("$100.00", atm.getDisplayContents());
    atm.pressButton("Continue");

    assertEquals(100.00, mockTransaction.getAmount());
    assertEquals(TEST_CHECKING_ACCOUNT, mockTransaction.getSourceAccount());
    assertEquals(TEST_CASH_ACCOUNT, mockTransaction.getDestAccount());
    mockTransaction.validate();
}
```

This solution yields a test that is slightly longer, but is only concerned with the immediate behavior of the class being tested, rather than the behavior of the entire system that lies beyond the ATM's interface. That is, we no longer check that the final balance of the test account is correct; we would check that function in the unit test for the `Transaction` object, not the `AtmGui` object.

**Note:** According to its inventors, a mock object is supposed to perform all of its own validation inside its `validate()` method. In this example, for clarity, we left some of the validation inside the test method. As you grow more comfortable using mock objects, you will develop a feel for how much validation responsibility to delegate to the mock.

### Inner class magic

In Listing 6, we used an anonymous inner subclass of `AtmGui` to override the `createTransaction` method. Because we only had to override one simple method, this was a concise way to achieve our goal. If we were overriding multiple methods or sharing the `AtmGui` subclass between many tests, it could pay to create a full (non-anonymous) member class.

We also used an instance variable to store a reference to the mock object. This is the simplest way to share data between the test method and the specialization class. It is acceptable because our test framework is not multithreaded or reentrant. (If it were, we would have to protect ourselves with `synchronized` blocks.)
Finally, we defined the mock object itself as a private inner class of the test class -- often a convenient approach because it is clearer to put the mock right next to the test code that uses it, and because inner classes have access to the instance variables of their surrounding class.

**Better safe than sorry**

Because we overrode the factory method to write this test, it turns out that we no longer have any test coverage of the original creation code (which is now inside the base class's factory method). It may be beneficial to add a test that covers this code explicitly. This can be as simple as invoking the base class's factory method and asserting that the returned object is of the correct type. For example:

```java
AtmGui atm = new AtmGui();
Transaction t = atm.createTransaction();
assertTrue(!(t instanceof MockTransaction));
```

Note that the inverse, `assertTrue(t instanceof Transaction)` would not suffice, because a `MockTransaction` is a `Transaction` as well.

**From factory method to abstract factory**

At this point, you may be tempted to go one step further and replace the factory method with a full-fledged abstract factory object, as detailed in *Design Patterns* by Erich Gamma, et al. (see Related topics). In fact, many would have started this approach with a factory object, rather than a factory method -- we did, but soon backed away.

Introducing a third object type (role) into the system has some potential disadvantages:

1. It increases the complexity without a corresponding increase in functionality.
2. It may force you to change the public interface to the target object. If an abstract factory object must be passed in, then you must add a new public constructor or mutator.
3. Many languages have conventions attached to the concept of "factory" that may lead you astray. For instance, in the Java language, factories are often implemented as static methods; this is not appropriate in this situation.

Remember, the whole point of this exercise is to make the object easier to test. Often, designing for testability can push the object's API toward a cleaner, more modular state. But it can be taken too far. Test-driven design changes should not pollute the public interface of the original object.

In the ATM example, as far as the production code is concerned, the `AtmGui` object only ever makes the one type of `Transaction` object (the real kind). The test code would like it to produce a different type (a mock). But forcing the public API to accommodate factory objects or abstract factories, just because the test code wants it to, is the wrong design. If production code has no need to instantiate many types of this collaborator, then adding that ability will make the resulting design needlessly hard to understand.
Related topics

- The source of the factory method and abstract factory design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1997) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides -- also known as the Gang of Four.
- If you’re new to the world of design patterns, you may also want to take this free, two-part tutorial:
  - [Java design patterns 101](developerWorks, January 2002) introduces patterns.
- The [Refactoring Home Page](developerWorks) maintained by Martin Fowler is a great resource for tidy programmers.