Testing with Mock Objects

You saw earlier that the stubbing technique lets you unit test portions of code by isolating them from the environment (such as by stubbing a web server, the filesystem, a database, and so on).

**What about fine-grained isolation**, such as being able to isolate a method call to another class? Can you achieve this without deploying huge amounts of energy that would negate the benefits of having tests?

The answer is yes. The technique is called ***mock objects.***

**The strategy allows you to unit test at the finest possible level**. You develop method by method after having provided unit tests for each method.

# Introducing Mock Objects

Testing in isolation offers strong benefits, such as:

* the ability to test code that has not yet been written (as long as you at least have an interface to work with).
* In addition, testing in isolation helps teams unit test one part of the code without waiting for all the other parts.
* The biggest advantage is the ability to write focused tests that test a single method, with no side effects resulting from other objects being called from the method under test.

Writing small, focused tests is a tremendous help; small tests are easy to understand and do not break when other parts of the code are changed.

Remember that one of the benefits of having a suite of unit tests is the courage it gives you to refactor mercilessly. If you have large tests, and your refactoring introduces a bug, several tests will fail. That

result will tell you that there is a bug somewhere, but you will not know where. With fine-grained tests, potentially fewer tests are affected, and they provide precise messages that pinpoint the cause of the failure.

*Mock objects* (*mocks*, for short) are perfectly suited for testing a portion of code logic in isolation from the rest of the code. Mocks replace the objects with which your methods under test collaborate, thus offering a layer of isolation. In that sense, they are similar to stubs. **The similarity ends there**, however, **because mocks do not implement any logic: they are empty shells** that provide methods **that** **let the tests control the behavior of all the business methods of the faked class**.

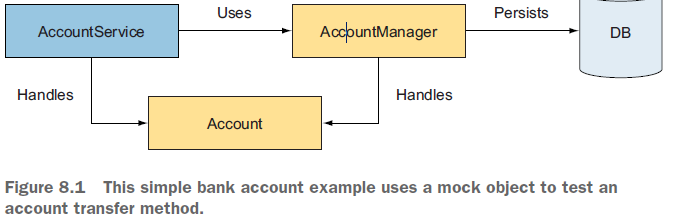
**A stub is created with a predetermined behavior**, even a very simple one, **and this behavior cannot be changed when running different tests**. A mock does not have a predetermined behavior. When running a test, you are setting the expectations on the mock before effectively using it. You may run different tests, and you may reinitialize a mock and set different expectations on it. The pattern for testing with a mock is this:

*initialize mock > set expectations > execute test > verify assertions*.

# Unit Testing with Mock Objects

Imagine a very simple use case in which we want to be able to make a bank transfer from one

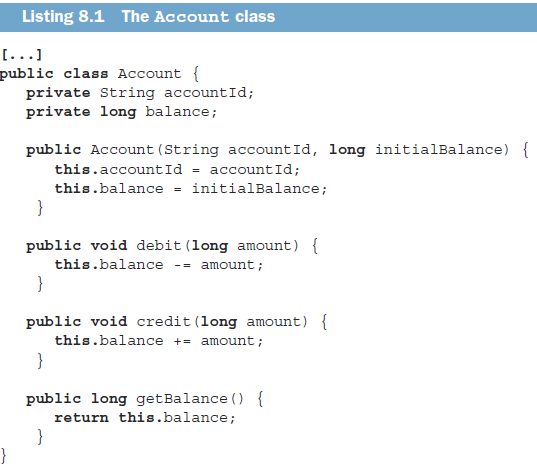
account to another:



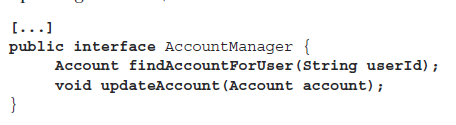
The AccountService class offers services related to Account objects and uses AccountManager to persist data to the database (by using JDBC, for example).

The service of interest to us materializes itself via the AccountService.transfer method, which makes the transfer.

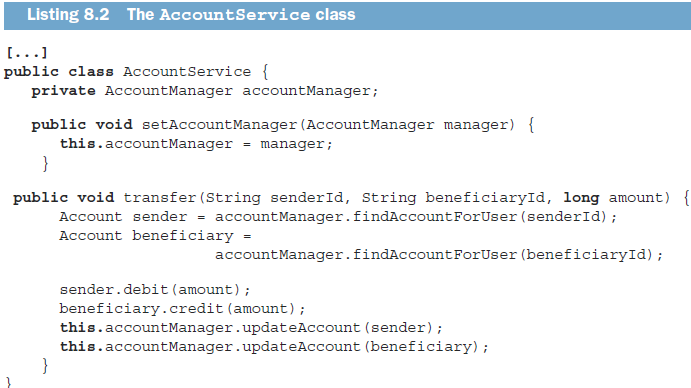
Without mocks, testing the AccountService.transfer behavior would imply setting up a database, presetting it with test data, deploying the code inside the container (Java EE application server, for example), and so forth. Although this process is required to ensure that the application works end to end, **it is too much work** **when you want to unit test only your code logic**.



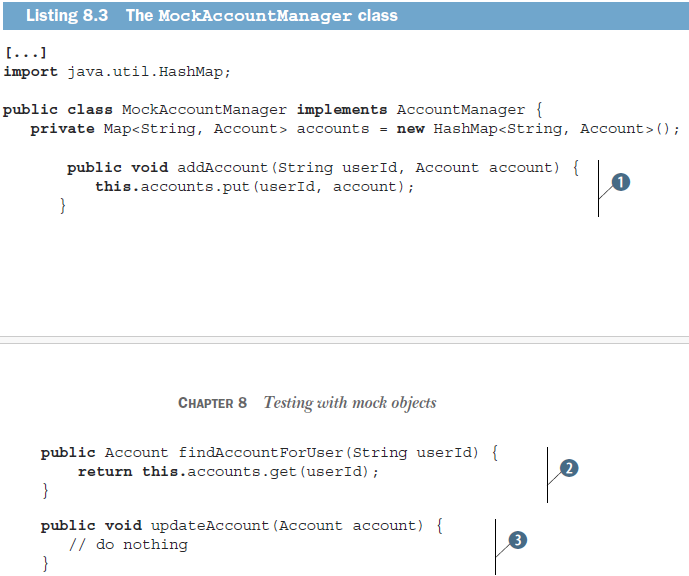
The AccountManager interface manages the life cycle and persistence of Account objects (limited to finding accounts by ID and updating accounts):



the transfer method is designed for transferring money between two accounts. It uses the previously defined AccountManager interface to find the debit and credit accounts by ID and to update them.



We want to be able to unit test the AccountService.transfer behavior. For that purpose, until the implementation of the AccountManager interface is ready, we use a mock implementation of the AccountManager interface because the transfer method is using this interface, and we need to test it in isolation.



In this listing:

 The addAccount method uses an instance variable to hold the values to return (1). Because we have several Account objects that we want to be able to return, we store them in a HashMap. This step makes the mock generic and able to support different test cases. One test could set up the mock with one account, another test could set it up with two accounts or more, and so forth.

 We implement a method to retrieve the account from the accounts map (2). We can retrieve only accounts that were added earlier.

 The updateAccount method does nothing for now, and it does not return any value (3). Thus, we do nothing.

JUnit best practices: Do not write business logic in mock objects

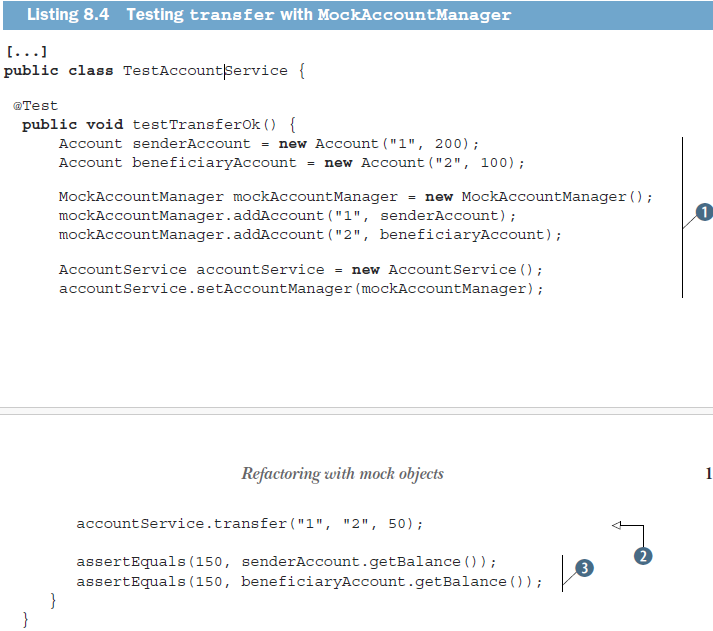
The single most important rule to follow when writing a mock is that it should not have any business logic: **a mock must be a dumb object** **that does only what the test tells it to do**. In other words, it is purely driven by the tests. This characteristic is precisely the opposite of stubs, which contain all the logic (see chapter 7).

This point has two nice corollaries. First, mock objects can be generated easily. Second, because mock objects are empty shells, they are too simple to break and do not need testing themselves.

Now we are ready to write a unit test for AccountService.transfer. The following listing shows a typical test that uses a mock.

During the test setup, we create the MockAccountManager object and define what it should return when it’s called for the two accounts we manipulate (the sender and beneficiary accounts). Practically, setting the expectations of the mock-AccountManager object by adding two accounts to it transforms it into our own defined mock.

As stated earlier, one of the characteristics of a mock is that when you run a test, you are setting the expectations on the mock before effectively using it. We have succeeded in testing the AccountService code in isolation from the other domain object, AccountManager, which in this case did not exist but could have been implemented with JDBC in real life.



JUnit best practices: Test only what could possibly break

You may have noticed that we did not mock the Account class. The reason is that this data-access-object class does not need to be mocked; it does not depend on the environment, and it is very simple. **The other tests use the Account object, so they test it indirectly**. If this class failed to operate correctly, the tests that rely on Account would fail and alert us to the problem.

The next section demonstrates that **writing unit tests with mocks leads to refactoring the code under test**—**and that this process is a good thing.**

# Refactoring with Mock Objects

Some people used to say that unit tests should be fully transparent to the code under test and that runtime code should not be changed to simplify testing. *This is wrong!*

**Unit tests are first-class users of the runtime code** and deserve the same consideration as any

other user**. If the code is too inflexible for the tests to use, we should correct the code.**

Consider the following piece of code created by one of the engineers at Tested Data Systems. This engineer is taking care of the implementation of the Account-Manager class that we previously mocked until it is fully available:

**public class** DefaultAccountManager1 implements AccountManager {

**private static final** Log logger = LogFactory.getLog(DefaultAccountManager1**.class**);

**public** Account findAccountForUser(String userId) {

logger.debug("Getting account for user [" + userId + "]");

ResourceBundle bundle =

PropertyResourceBundle.getBundle("technical");

String sql = bundle.getString("FIND\_ACCOUNT\_FOR\_USER");

// Some code logic to load a user account using JDBC

[...]

}

[...]

}

In this listing:

 We create a Log object.

 We retrieve an SQL command.

We can see two issues, both of which relate to code flexibility and the ability to resist change:

* The first problem is that it is not possible to decide to use a different Log object, as it is created inside the class. For testing, for example, you probably would like to use a Log that does nothing, but you cannot do so.

As a rule, **a class like this one should be able to use whatever Log it is given**. The goal of this class is not to create loggers, but to perform some JDBC logic.

* The same goal applies to PropertyResourceBundle. It may sound OK right now, but what

happens if we decide to use XML to store the configuration? Again, it should not be the goal of this class to decide what implementation to use.

**An effective design strategy is to pass to an object any other object that is outside its immediate business logic**. **Ultimately, as you move up in the calling layers, the decision to use a given logger or configuration should be pushed to the top level**. **This strategy provides the best possible code flexibility and ability to cope with changes.** And as we all know; change is the only constant. Taking these issues into consideration, the Tested Data Systems engineer who created the code **will need to refactor it.**

## Refactoring Example

Refactoring all code so that domain objects are passed around can be time consuming. You may not be ready to refactor the entire application just to be able to write a unit test.

Fortunately, **an easy refactoring technique lets you keep the same interface for the code but allows it to be passed domain objects that it should not create**. As proof, the following listing shows what the refactored DefaultAccountManager1 class could look like:

**public class** DefaultAccountManager2 **implements** AccountManager {

**private Log logger;**

**private Configuration configuration;**

**public** DefaultAccountManager**2() {**

**this**(LogFactory.getLog(DefaultAccountManager2.**class**),

**new** DefaultConfiguration(**"technical"**));

}

**public** DefaultAccountManager2(Log logger, Configuration configuration) {

**this.logger** = logger;

**this.configuration** = configuration;

}

**public** Account findAccountForUser(String userId) {

**this.logger**.debug(**"Getting account for user ["** + userId + **"]"**);

**this.configuration**.getSQL(**"FIND\_ACCOUNT\_FOR\_USER"**);

// Some code logic to load a user account using JDBC

[...]

}

[...]

}

we swap the PropertyResourceBundle class from the previous listing in favor of a new Configuration field. This swap makes the code more flexible because it introduces an interface (which will be easy to mock), and the implementation of the Configuration interface can be anything we want (including using resource bundles).

The design is better now because we can use and reuse the DefaultAccount-Manager2 class with any implementation of the Log and Configuration interfaces (if we use the constructor that takes two parameters). The class can be controlled from the outside (by its caller).

We have a no-args constructor and a constructor with arguments. The no-args constructor initializes the logger and configuration field members with default implementations.

## Refactoring Considerations

With this refactoring, we have provided a trap door for controlling the domain objects from the tests. We retain backward compatibility and pave an easy refactoring path for the future. Calling classes can start using the new constructor at their own pace.

**Design patterns in action: Inversion of Control (IoC)**

**Applying the IoC pattern to a class means removing the creation of all object instances for which this class is not directly responsible and passing any needed instances instead. The instances may be passed by a specific constructor or a setter, or as parameters of the methods that need them. It becomes the responsibility of the calling code to set these domain objects correctly on the class that is called.**

IoC makes unit testing a breeze. To prove the point, here’s how easily we can write a test for the findAccountByUser method.

**public void** testFindAccountByUser() {

MockLog logger = **new** MockLog();

MockConfiguration configuration = **new** MockConfiguration();

configuration.setSQL("SELECT \* [...]");

DefaultAccountManager2 am = **new** DefaultAccountManager2(logger,

configuration);

Account account = am.findAccountForUser("1234");

// Perform asserts here

[...]

}

We have been able to completely control the logging and configuration behavior from outside the code to test, in the test code. As a result, the code is more flexible and allows for any logging and configuration implementation to be used.

One last point to note is that if we write the test first, we will automatically design the code to be flexible. Flexibility is a key point in writing a unit test. If we test first, we will not incur the cost of refactoring the code for flexibility later.

# Mocking an HTTP Connection