Test Doubles and Mocking

Estimated time for completion: 45 minutes

Overview:

In this lab you will take a refactored project and write unit tests that provide alternate implementations of the dependencies (test doubles). You will initially write them by hand and then use a Mocking Framework to automate their creation

Goals:

- Understand the use of test doubles in unit testing
- Learn how to write different kinds of test doubles
- Learning how to use a Mocking framework to generate test doubles

Lab Notes:

N/A

Part 1: Examine the starter solution

The first part of the lab is to familarise yourself with the assembly that contains the code to be tested. If you have previously done the "Design for Testing" lab it is the same project that we are taking forward to introduce tests now that we have decoupled the dependencies.

1. Open **Before/TestDoubles.sln**
2. For this exercise we will be concentrating on writing tests for the SMSService and CRM. Look at these classes and notice their dependencies (injected in their constructors)

Part 2: Writing Test Doubles

In this part of the lab you will add the test project and write tests using custom written test doubles

1. Add a new test project to the solution called **Services.Tes**t and reference the **Services** assembly
2. Rename the generated test class to `CRMTests`
3. Create a test to check that when `SendCustomerMessage` is called the `CRM` sends an `SMS` to the contacts returned by the repository
4. You will need a version of `IContactRepository` that returns data under your control. As it is returning specific data to drive the test, this test double is known as a stub. Create a class called `StubContactRepository` that implements `IContactRepository`

5. In the stub repository you only need to implement the `GetAll` method as this is the only one used by the SUT. The other methods can throw `NotImplementedException`
6. Create a member variable of type array of `Contact` with three contacts in it (you don't need any special values for the contacts) and return this array from `GetAll`
7. You will also need a way to verify that the contacts are sent to the `SMSService`. So you will need an implementation of `IDeliveryService` that records the data sent to it – this kind of test double is called a spy. So create a class called `SpyDeliveryService` that implements `IDeliveryService`
8. In the spy, create a readonly property Contacts of type `IEnumerable<Contact>` and in the `Send` method assign the passed contacts to this property
9. Now we have our test doubles we can write the test. Create an instance of each of the test doubles as local variables in the test
10. Create an instance of the `CRM` passing your doubles as the first two parameters to the constructor. The last parameter can be `null` as we are not testing email functionality in this test
11. Invoke the `SendCustomerMessage` method of the `CRM` with a delivery method of `SMS` and an empty message template. This should drive the test doubles correctly
12. Use `CollectionAssert.AreEquivalent` to verify that the contacts returned by the repository `GetAll` method are the same as those captured by the spy. The only awkward thing here is that `CollectionAssert` methods assume `ICollection` rather than `IEnumerable` so you will have to convert the parameters using `ToList`

```
[TestMethod]
public void
   SendCustomerMessage_WhenInvoked_SendsSMSToContactsFromRepository()
{
    var repo = new StubContactRepository();
            var deliveryService = new SpyDeliveryService();

            var crm = new CRM(repo, deliveryService, null);

            crm.SendCustomerMessage(DeliveryMethod.SMS, "");

            CollectionAssert.AreEquivalent(repo.GetAll().ToList(),
   deliveryService.Contacts.ToList());
        }
```

13. Compile and run your test – you should see it pass even though none of the real dependencies are in place.

Part 3: Using a Mocking Framework to Create the Test Doubles

Rolling your own test doubles can sometimes be the simplest solution but often it ends up with more code for you to maintain. It ius normally better to generally use a mocking framework and save rolling your own test doubles for places where you would otherwise have to fight the mocking framework. In this part of the lab you will use the mocking framework Rhino Mocks to generate your test doubles

1. We are going to use Rhino Mocks to generate the test doubles so use **NuGet** to add a package reference to the test project for Rhino Mocks
2. In the `CRM` test comment out the creation of the `StubContactRepository` – we are going to use Rhino Mocks to generate the contact repository dynamically
3. Use the `static GenerateStub` method of `MockRepository` to generate a stub implementation of `IContactRepository`. Remember that currently this stub method has no method implementations
4. We need some data for the stub to return so create a local variable of type `List<Contact>` with three empty `Contacts`
5. Stub the repository's `GetAll` method returning the list of contacts

```
List<Contact> contacts = new List<Contact>()
                         {
                             new Contact(),
                             new Contact(),
                             new Contact(),
                         };

var repo = MockRepository.GenerateStub<IContactRepository>();

repo.Stub(r => r.GetAll())
    .Return(contacts);
```

6. Change the `CollectionAssert` to use the local variable as the expected result
7. Rerun the test a verify it still passes
8. Stubs are very straightforward to create with Rhino Mocks, however, if the dependency doesn't return anything then a stub cannot drive the test conditions. To verify behavior in this situation we need mock objects. Add a new unit test file to the project and call it `SMSServiceTests`. We are going to verify that the `SMSService` calls the `SMSGateway` once for each `Contact` from the repository
9. Create a test that verifies that `Send` calls the gateway correctly for a contact passed
    a. Create a `Contact` array with data for the `Name` and `CellPhone`
    b. Create a string for a message template (must have a `{0}` placeholder)
    c. Create a string with the name inserted into the message template as the expected message
10. We need to track the expected state so create an instance of the `MockRepository`
11. Using the `MockRepository` instance create a dynamic mock for `ISMSGateway`

```
var mocks = new MockRepository();
string phone = "123";
var contact = new Contact {FirstName = "Rich", CellPhone = phone};
string msgTemplate = "Foo {0}";
string expectedMessage = "Foo Rich";

var gateway = mocks.DynamicMock<ISMSGateway>();
```

12. Now we need to set the expectations using the `Expect` static class. Set an expectation that the `smsGateway` has the `Send` method called on it with the contact's phone and the message resolved from the template

```
Expect.Call(() => gateway.Send(phone, expectedMessage));
```

13. Create an instance of the `SMSService`. You are now ready to check the expected conditions are met as the SMS service is called
14. Call the `ReplayAll` method of the mock repository
15. Call the `Send` method of the SMS service
16. Call the `VerifyAll` method of the mock repository

```
Expect.Call(() => gateway.Send(phone, expectedMessage));

var sms = new SMSService(gateway);

mocks.ReplayAll();

sms.Send(new Contact[] { contact }, msgTemplate);

mocks.VerifyAll();
```

17. Compile and run your test – the test should pass
18. Finally we are going to ensure that the `smsService` is called for each `Contact` passed to the `Send` method – create a test named appropriately
19. Create an instance of the `MockRepository`
20. Create an array of `Contact` objects
21. Create a dynamic mock for the `ISMSGateway`

```
var mocks = new MockRepository();

Contact[] contacts = new[]
    {
        new Contact(),
        new Contact(),
        new Contact(),
        new Contact(),
        new Contact(),
    };
var gateway = mocks.DynamicMock<ISMSGateway>();
```

22. Set an expectation that the `Send` method of the SMS Gateway will be called the same times as elements in the contact array. We don't care about the parameters passed so state that arguments should be ignored

```
Expect.Call(() => gateway.Send(null, null))
      .IgnoreArguments()
      .Repeat
      .Times(contacts.Length);
```

23. Create the instance of the `SMSService`
24. Call `ReplayAll`
25. Call `Send` on the SMS service with the contact array and a string with a `{0}` placeholder
26. Call `VerifyAll`

```
var sms = new SMSService(gateway);

mocks.ReplayAll();

sms.Send(contacts, "foo {0}");

mocks.VerifyAll();
```

27. Compile and run the test, the test should pass
28. However, here we can see a danger with tests that pass first time – change the expectation that the method will be called one time less than the number of items in the contact array and rerun the test – notice the test still passes. This is because dynamic mocks can only trace if something was called – it doesn't track how many times. For this you must use a strict mock
29. Change the call on the `MockRepository` instance from `DynamicMock` to `StrictMock`. Compile and re-run the test – notice that the test now fails.
30. Change the expectation back to be the number of items in the array and re-run the test – the test should now pass again. If you ever need to count the number of executions rememeber to use a strict mock

---

Solutions

after\TestDoubles.sln