Global Knowledge®

Doubles and Mocking

Estimated time for completion: 60 minutes

Overview

Creating test doubles by hand is tedious and time consuming. Mocking frameworks enable these classes to be created automatically, without much effort on the part of the developer. This significantly reduces the work involved in creating large libraries of unit tests that require many test doubles. Various mocking frameworks are in use today; this lab will use Moq from Clarius Consulting.

Goals

- Using Moq to generate test stubs
- Handling exceptions when using test stubs
- Creating test stubs that raise exceptions
- Using mock objects to verify behaviour (optional)

Notes

You will use the Moq mocking framework to help create additional unit tests for the blog service that you worked with in the previous lab.

Part 1 – Using Moq to generate test stubs

In this part, you will write additional unit test for the blog service that you worked with in the previous lab. These unit tests will require test stubs, which will be generated by Moq.

Steps

1. Locate the C:\Labs\Work\Doubles and Mocking\Before folder and then open the TFSBlog solution (alternatively, if you completed it, you may continue with the solution from the previous Dependency Injection and Inversion of Control lab).
2. In the TBlogServiceTests project, delete the DummyBlogRepository.cs file; this will no longer be needed as a replacement will be generated by Moq.

3. Open BlogServiceTest.cs and locate the unit test. The DummyBlogRepository will need to be replaced with a mock.
4. Add a folder called Library to the TBlogServiceTests project and then create a folder within the Library folder called Moq.
5. Unzip the Moq redistributable (Moq.4.0.10827.Final.zip) and copy the contents of the Moq.4.0.10827\NET40 folder to the TBlogServiceTests\Library\Moq folder.
6. In the TBlogServiceTests project, right-click the new Moq folder and then select Add | Existing Item. Browse to the TBlogServiceTests\Library\Moq folder, change the file type to All Files (*.*), select all files and then click Add.
7. Add a reference in the TBlogServiceTests project to the Moq assembly (Moq.dll) and then add a using directive for the Moq namespace in BlogServiceTest.cs.
8. Add a using directive for the TFSBlogRepository namespace in BlogServiceTest.cs.
9. You are now ready to use Moq. The next task is to create a mock implementation of an IBlogRepository, to replace the DummyBlogRepository class. To do this, add a field to the BlogServiceTest class of type Mock<IBlogRepository>. Call the field blogRepository but do not initialise it.
10. Add a test initialisation method to the BlogServiceTest class and initialise the new blogRepository field (remember to add the TestInitialize attribute).

```
private Mock<IBlogRepository> blogRepository;

[TestInitialize]
public void TestInitialize()
{
   this.blogRepository = new Mock<IBlogRepository>();
}
```

11. You now have a mock implementation of an IBlogRepository to use in unit tests. However, the existing unit test requires the mock object to return a collection of blogs when the GetBlogs method is called, so the next task is to enable this.
12. Add a variable at the start of the unit test of type IQueryable<BlogModel.Blog>. Call it blogs, and then initialise it by creating a new List<BlogModel.Blog> and populating it with two BlogModel.Blog instances (each blog instance needs a reference to a BlogModel.User instance). Finally, call AsQueryable on the list.

```
IQueryable<BlogModel.Blog> blogs = new List<BlogModel.Blog>
{
   new BlogModel.Blog
      { User = new BlogModel.User { Name = "Cal" } },
   new BlogModel.Blog
      { User = new BlogModel.User { Name = "Michael" } }
}
.AsQueryable();
```

13. This collection of blogs needs to be returned when the GetBlogs method is called on the mock object. To achieve this, call the Setup method on the blogRepository field, passing an action delegate that has one parameter of type IBlogRepository. Call GetBlogs on the delegate parameter, and call Returns on the return value of the call to Setup. Pass the collection of blogs as the argument to Returns.
14. Finally, replace the DummyBlogRepository in the BlogService constructor with the value of the Object property of the blogRepository field.

```
[TestMethod]
public void GetBlogs_LessThanPageSizeBlogs_ReturnsAllBlogs()
{
   IQueryable<BlogModel.Blog> blogs = new List<BlogModel.Blog>
   {
      new BlogModel.Blog
         { User = new BlogModel.User { Name = "Cal" } },
      new BlogModel.Blog
         { User = new BlogModel.User { Name = "Michael" } }
   }
   .AsQueryable();

   this.blogRepository.Setup(
      b => b.GetBlogs() ).Returns( blogs );

   BlogService target =
      new BlogService( this.blogRepository.Object );

   IEnumerable<Blog> actual = target.GetBlogs();

   Assert.AreEqual( 2, actual.Count() );
}
```

15. Run the unit test; it should pass.

Part 2 – Handling exceptions when using test stubs

Here, you will see how to handle exceptions that occur in code that is being called by test stubs.

Steps

1. Add another unit test called GetBlogs_NullRepository_ThrowsException. This unit test should follow the lines of the first unit test, but should pass a value of null as the argument in the call to the Returns method (to resolve the ambiguity, specify the type parameter for the Returns method as <IQueryable<BlogModel.Blog>).
2. Run the unit tests; the new test should fail with an ArgumentNullException.
3. In the TBlogService project, add a new exception class called ServiceException.
4. In the GetBlogs method of the BlogService class, catch any exceptions that occur and throw a new ServiceException instead.

```
Blogs blogFactory = new Blogs();
IEnumerable<Blog> blogs;
try
{
   blogs = blogFactory.GetBlogs( this.repository, 10, 1 );
}
catch
{
   throw new ServiceException();
}

return blogs;
```

5. In the new unit test, add an ExpectedException attribute for ServiceException.

```
[TestMethod]
[ExpectedException( typeof( ServiceException ) )]
public void GetBlogs_NullRepository_ThrowsException()
{
   this.blogRepository.Setup( b => b.GetBlogs() )
      .Returns<IQueryable<BlogModel.Blog>>( null );

   BlogService target =
      new BlogService( this.blogRepository.Object );

   IEnumerable<Blog> actual = target.GetBlogs();

   Assert.AreEqual( 2, actual.Count() );
}
```

6. Run the unit tests again; they should now pass.

Part 3 – Creating test stubs that raise exceptions

In the previous part, an exception was created indirectly by arranging for GetBlogs to return a value of null. If the implementation of the blog service was modified so that the argument null exception was handled at the location where it occurred, then the test would fail, as the catch handler would not be executed. The test is therefore fragile, and may not be reliable. A better approach would be for the test stub to raise an exception, thereby ensuring that an exception would always occur. This would enable the exception handling in the GetBlogs method to be tested in a more reliable manner. Here, you will configure the test stub to raise an exception and test that the service responds appropriately.

Steps

1. Add a unit test called GetBlogs_ExceptionRaised_ThrowsServiceException. This unit test should follow the lines of the last unit test, but this time it should call Throws instead of Returns. Pass a NullReferenceException as the argument.

```
[TestMethod]
[ExpectedException( typeof( ServiceException ) )]
public void GetBlogs_ExceptionRaised_ThrowsServiceException()
{
   this.blogRepository.Setup( b => b.GetBlogs() )
     .Throws( new NullReferenceException() );

   BlogService target =
      new BlogService( this.blogRepository.Object );

   IEnumerable<Blog> actual = target.GetBlogs();

   Assert.AreEqual( 2, actual.Count() );
}
```

2. Run the unit tests; they should all pass.

Part 4 – Using mock objects to verify behaviour (optional)

In this part, you will use mocking to verify that an expected action occurs during the execution of the test. Specifically, you will verify that an audit method is called when GetBlogs is called.

Steps

1. Add a new class library project named Audit to the TFSBlog solution.
2. Delete the file Class1.cs and then add a new interface named IAudit.
3. Make the IAudit interface public and then add a single member, a method called Message, which should return void and have a single parameter of type string.

4. Add another new class library project named SimpleAudit to the TFSBlog solution and add a reference to the Audit project.
5. Rename Class1.cs and the class it contains to ConsoleAudit.
6. Implement the IAudit interface on the ConsoleAudit class and then implement the Message method (simply write the value of the parameter to the console).
7. In the TBlogService project, add references to the Audit and SimpleAudit projects.
8. Next, in the BlogService class, add a field of type IAudit called audit.
9. The Unity configuration in App.config now needs updating to support auditing.
    a. Open the App.config file in the TBlogServiceHost project.
    b. Add assembly and namespace mappings for Audit and for SimpleAudit.
    c. Add a mapping between the IAudit interface and the ConsoleAudit type.
    d. The final Unity configuration should resemble the following:

```
<unity
   xmlns="http://schemas.microsoft.com/practices/2010/unity">

   <assembly name="TFSBlogRepository" />
   <assembly name="Audit" />
   <assembly name="SimpleAudit" />

   <namespace name="TFSBlogRepository" />
   <namespace name="Audit" />
   <namespace name="SimpleAudit" />

   <container>

      <register type="IBlogRepository" mapTo="BlogRepository">
         <constructor>
            <param
               name="connectionString"
               value="<connection string>" />
         </constructor>
      </register>

      <register type="IAudit" mapTo="ConsoleAudit" />

   </container>

</unity>
```

10. Now that Unity has been configured, auditing can be added to the service. In the non-default constructor of the BlogService class, add a parameter of type IAudit and use this to initialise the IAudit field. Finally, update the default constructor to provide the additional argument by making another call to Resolve.

11. In the GetBlogs method, if the value of the IAudit field is not null, call its Message method, passing a suitable message.
12. You will now find that your tests no longer compile, as the BlogService no longer provides a constructor with a single parameter. To complete the auditing update, add a reference to the Audit project in the TBlogServiceTests project and then fix the tests by passing null for the second argument.
13. Verify that the solution builds without errors or warnings and that all tests pass.
14. The final task is to verify that the Message method is being called by GetBlogs.
    a. Add another unit test called GetBlogs_IAuditNotNull_MessageCalled.
    b. In the new test, create an IAudit mock and use this in the constructor of the BlogService.
    c. Call GetBlogs on the service.
    d. Verify that the Message method was called exactly once, with the correct message, and that the GetBlogs method of the repository was also called exactly once.

```
[TestMethod]
public void GetBlogs_IAuditNotNull_MessageCalled()
{
   Mock<IAudit> audit = new Mock<IAudit>();

   BlogService target = new BlogService(
      this.blogRepository.Object, audit.Object );

   target.GetBlogs();

   audit.Verify( a => a.Message( "GetBlogs" ), Times.Once() );
   this.blogRepository.Verify(
      b => b.GetBlogs(), Times.Once() );
}
```

    e. Run the unit tests; they should all pass.

Solution

The final solution code for this lab can be found in C:\Labs\Work\Doubles and Mocking\After.