Test Doubles

Estimated time for completion: 60 minutes

Overview:

In this lab you will take the service you tested previously and add extra tests to it. Those tests require the creation of a number of "test doubles", these are tedious to create by hand so we will use Rhino Mocks to create the doubles

Goals:

- Understand the need for test doubles in the code
- Use Rhino Mocks to create the doubles

Lab Notes:

You will use Rhino Mocks to further test the Service you tested in the previous lab

Part 1: Using Rhino Mocks

In this part of the lab you will write code to further test the service.You will create doubles using Rhino Mocks to do this

1. The test project (**TBlogServiceTest**) currently contains a dummy class (**DummyBlogRepository**), delete this now.

2. In the **BlogServiceTest** class you will need to replace the **new DummyBlogRepository** with a mock, we will use Rhino Mocks to do this. You will need to create a Rhino Mocks repository, get this repository to create a mock of the **IBlogRepository** interface and then use this interface when calling the service
3. Using Rhino Mocks
4. In the test project create a lib directory
5. Copy the Rhino.Mocks.dll from the labs\lib directory to this projects lib directory and add a reference to Rhino to the project.
6. In the **BlogServiceTest** class add a data member of type **IBlogRepository** called **_repository**
7. Add a **Setup** method and mark it with the **TestInitialize** attribute.
8. In the **Setup** method initialize the _repository member by calling MockRepository.GenerateStub<IBlogRepository>();

```
[TestInitialize]
public void Initialize()
{
    _repository = MockRepository.GenerateStub<IBlogRepository>();
}
```

9. You now have a stub implementation of the interface ready to be used in the test
10. You will mock the **GetBlogs** method on the interface. To do this the method has to return a collection of **Blogs.**
11. In the test method create a **BlogModel.User** instance, this will be used by the **Blog**s you are about to create.
12. In the test method create and initialize a **List<Blog>** collection adding two Blogs, make sure each **Blog** is initialized with a **User**.

```
BlogModel.User user = new BlogModel.User();

List<BlogModel.Blog> blogs = new List<BlogModel.Blog>
            {
                new BlogModel.Blog {User = user},
                new BlogModel.Blog {User = user},
            };
```

13. You now need to tell Rhino what to do when the GetBlogs method is called
14. Add a call to **_repository.Stub**. This takes an action delegate with a single parameter of type **IBlogRepository**. Use this parameter to specify the method to call (**GetBlogs**), the value to pass, and the return value (the Question you created in a previous step)
15. Call the **BlogService** passing the stubbed interface
16. Run the tests, they should pass.

```
[TestMethod()]
public void GetBlogs_...()
{
    BlogModel.User user = new BlogModel.User();

    List<BlogModel.Blog> blogs = new List<BlogModel.Blog>
                {
                    new BlogModel.Blog {User = user},
                    new BlogModel.Blog {User = user},
                };

    _repository.Stub(r => r.GetBlogs()).Return(blogs.AsQueryable());
    BlogService target = new BlogService(_repository);
    IEnumerable<Blog> actual;
    actual = target.GetBlogs();
```

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

Part 2: More Mocking

In this part of the lab you will write more tests against the BlogService class

1. We would like to run other tests against the Service class. Without using a mocking framework this would require that we create a new Dummy type each time we want to mock, this is tedious and error prone and leads to a proliferation of classes in the test project. With a mocking framework however this is trivial
2. Add a new test, call it **GetBlogs_ShouldThrowAnException_WhenRepositoryIsNull**.
3. In this test have the stub return **null** when **GetBlogs** is called
4. Run the test, it should fail with a **NullReferenceException**
5. In the **TBlogService** project add a new exception class called **ServiceException**
6. In the **GetBlogs** method check for exceptions and throw a new **ServiceException** if any are detected.
7. In the test add an **ExpectedException** attribute for the **ServiceException**
8. Re-run the test, it should now succeed

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

return blogs;
```

```
[TestMethod()]
[ExpectedException(typeof(ServiceException))]
public void GetBlogs_ShouldThrowAnException_WhenRepositoryIsNull()
{
    _repository.Stub(r => r.GetBlogs()).Return(null);
    BlogService target = new BlogService(_repository);
    IEnumerable<Blog> actual;
    actual = target.GetBlogs();
}
```

Part 3: Having stubs throw exceptions

In the previous test GetBlogs threw an exception 'by accident' If the code was changed so that exceptions were caught then that test would fail, this makes the test fragile. What we need to is be able to ensure that the service responds correctly whenever GetBlogs throws an exception. To do that we need to write a test that always causes GetBlogs to throw an exception, we can do that with the stub

1. Add a new test, call it:

   GetBlogs_ShouldThrowAnException_WhenGetBlogsThrowsAnException()

2. This method will again mock the **GetBlogs** method on the **IBlogRepository** interface but now in such a way that it throws an exception. The **GetBlogs** method on the Service should catch that exception and throw its own **ServiceException**, we will assert that it does
3. In the new test add a call to **Stub** that says it throws an **Exception** – you will also need to tell it to call **IgnoreArguments**.

```
_repository.Stub(r => r.GetBlogs())
    .IgnoreArguments(
    ).Throw(new NullReferenceException())
```

4. The full method looks like this

```
[TestMethod()]
[ExpectedException(typeof(ServiceException))]
public void GetBlogs_ShouldThrowAnException_WhenGetBlogsThrowsAnException()
{
    _repository.Stub(r => r.GetBlogs())
        .IgnoreArguments()
        .Throw(new NullReferenceException());
    BlogService target = new BlogService(_repository);
    IEnumerable<Blog> actual;
    actual = target.GetBlogs();
}
```

5. The important part for this lab is the call to **IgnoreArguments**, without this the stub will not throw the exception.
6. Run the tests, it should now succeed.

Part 4: Expectations

Is this part of the lab you will use expectations to prove that an audit method has been called.

1. Add a new **Audit** class library project, this will contain the audit interfaces
2. Add an **IAudit** interface to the new library with a single **Message** method that takes a **string**

3. Add a new **SimpleAudit** project. To this project add a reference to the **Audit** project.
4. Create a **ConsoleAudit** class in the **SimpleAudit** project that implements **IAudit**.
5. Implement the **Message** method to simply write the message to the console
6. To the **TBlogService** project add references to the **Audit** and **SimpleAudit** projects
7. Add an **IAudit _audit** member variable to the **BlogService** class in **TBlogService**
8. You now need to configure the **IoC** container for the **Service** class to reference **IAudit**
    a. In the **TBlogServiceHost**'s **app.config** file add a type entry for **IAudit**
    b. Add **namespace** entries for **Audit** and **SimpleAudit**
    c. Add **assembly** entries for **Audit** and **SimpleAudit**

```
<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 HERE"/>
      </constructor>
    </register>
    <register type="IAudit" mapTo="ConsoleAudit"/>
  </container>
</unity>
```

9. In the **BlogService** class's constructor add an **IAudit** parameter and in the default constructor add a call to the IoC container's **Resolve** method to resolve the audit instance
    a. Save the passed **IAudit** reference in the **_audit** variable
    b. Make sure you update all the necessary tests to pass null as the second parameter
10. In **GetBlogs** method if **_audit** is not null call the **_audit.Message** method
11. You now want to test that this is being called correctly
12. Add the **Audit** project as a reference to the **TBlogServiceTest** project
13. Make sure the tests compile and run
14. Add a new **Test** to the test project
15. In this test you will need to:

a. Create a new **MockRepository**
b. Create a **DynamicMock** of the **IBlogRepository** interface
c. Create a **StrictMock** of the **IAudit** interface
d. For the **IBlogRepository** mock create and expectation that **GetBlogs** is called and have it return a collection of **Blog**s
e. For the **IAudit** mock create an expectation that **Message** will be called
    i. This is a void method so you will need to use the lambda syntax
f. Ensure that the repository is in replay mode
g. Create an instance of the service passing the two mocks
h. Call **service.GetBlogs**
i. Verify that the **audit.Message** method was called

```
[TestMethod]
public void GetBlogs_ShouldProduceAnAuditMessage_WhenAnIAuditIsPassed()
{
    MockRepository mocks = new MockRepository();
    _repository = mocks.DynamicMock<IBlogRepository>();
    IAudit audit = mocks.StrictMock<IAudit>();

    BlogModel.User user = new BlogModel.User();
    List<BlogModel.Blog> blogs = new List<BlogModel.Blog>
                {
                    new BlogModel.Blog {User = user},
                    new BlogModel.Blog {User = user},
                };

    using (mocks.Record())
    {
        Expect.Call(() => audit.Message("Get Blogs"));
        Expect.Call(_repository.GetBlogs()).Return(blogs.AsQueryable());
    }

    using (mocks.Playback())
    {
        BlogService target = new BlogService(_repository, audit);
        target.GetBlogs();
    }
}
```