**Hands-On Lab**

# Creating Mock Objects

Contents

[Creating Mock Objects 1](#_Toc357078576)

[Lab Introduction 3](#_Toc357078577)

[Using Mocks 3](#_Toc357078578)

[Task 1 – Modify Your Code To Use An Interface 3](#_Toc357078579)

[Task 2 – Modify Your Business Layer 5](#_Toc357078580)

[Task 3 – Create Your Mock Object 5](#_Toc357078581)

[Task 4 – Setting Up Tests To Use The Mock Object 10](#_Toc357078582)

[Task 5 – Create Unit Tests 10](#_Toc357078583)

# Lab Introduction

In this lab you will learn the basics mock object to simulate code of real objects. This technique is used to isolate your tests to your business layer without the need to know or access your data persistence layer.

**Objectives**

In this Hands-On Lab, you will learn:

* How to write code so it is mock friendly
* How to create a mock test object
* How to run tests using mock the mock object

**Duration**

Suggested time to complete this Hands-On Lab is approximately 30-45 minutes.

**Setup – Starting Materials**

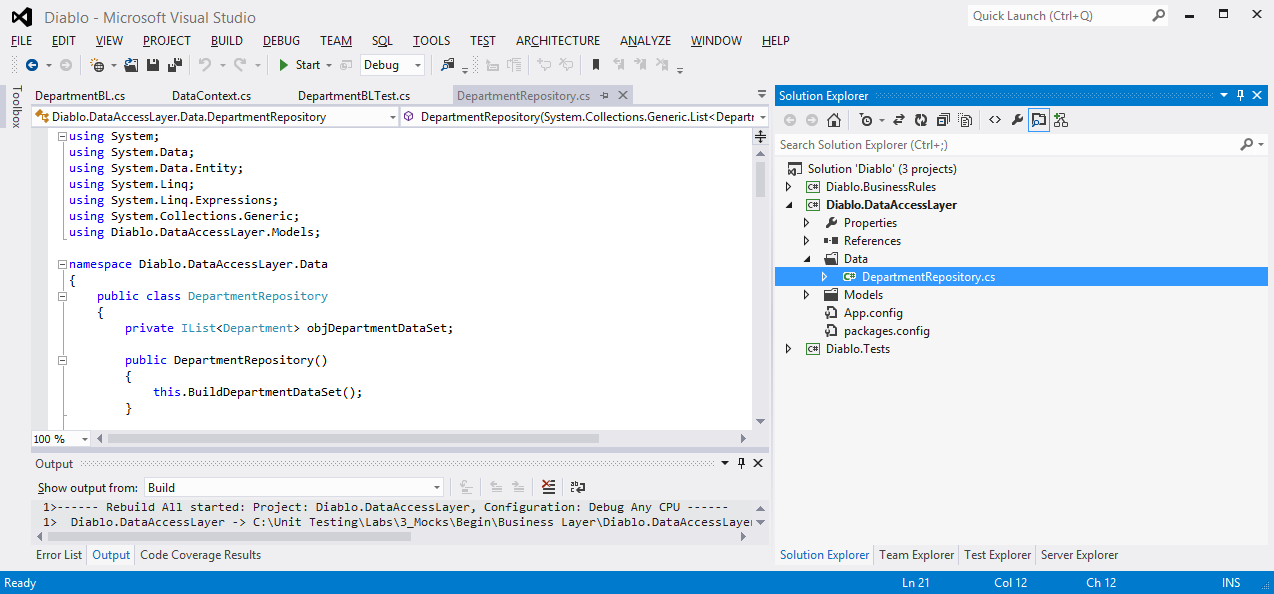
This Hands-On Lab includes the starting materials that are available under the root courseware folder on your machine. Open the folder **C:\Unit Testing\Labs\3\_Mocks.**

**Note**: This lab requires Visual Studio 2012

# Using Mocks

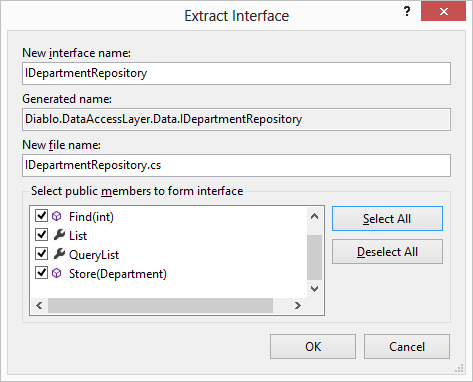
## Task 1 – Modify Your Code To Use An Interface

1. Open Visual Studio 2012
2. Open solution file in the 3\_Mock\Begin\MVC\Diablo.sln
3. Open DataAccessLayer project > Data > DepartmentRepository.cs



This class is used to interact with the data model using the repository factory model. In essence it what your business layer will call to do CRUD operations and retrieve data. It is at this layer that we want to create a mock for in our unit tests.

1. Right click on the public class DepartmentRepository > Refactor > Extract Interface. Select all of the public methods and click OK.



This creates a new IDepartmentRepository.cs file with the extracted IDepartmentRepository interface. It also modifies your DepartmentRepository class to use the new interface.

## Task 2 – Modify Your Controller

Now that our class has an interface we should modify the controller to use the interface type and accept a mocked object.

1. Open the MVCWebApp project > Controllers > DepartmentController.cs. Modify the first few lines of the DepartmentController class to read like following:

|  |
| --- |
| public class DepartmentController : Controller  {  private IDepartmentRepository objRepository;    public DepartmentBL()  {  this.objRepository = new DataAccessLayer.Data.DepartmentRepository();  }  public DepartmentController(IDepartmentRepository departmentRepository)  {  this.objRepository = departmentRepository;  }  *(the rest of the code..)* |

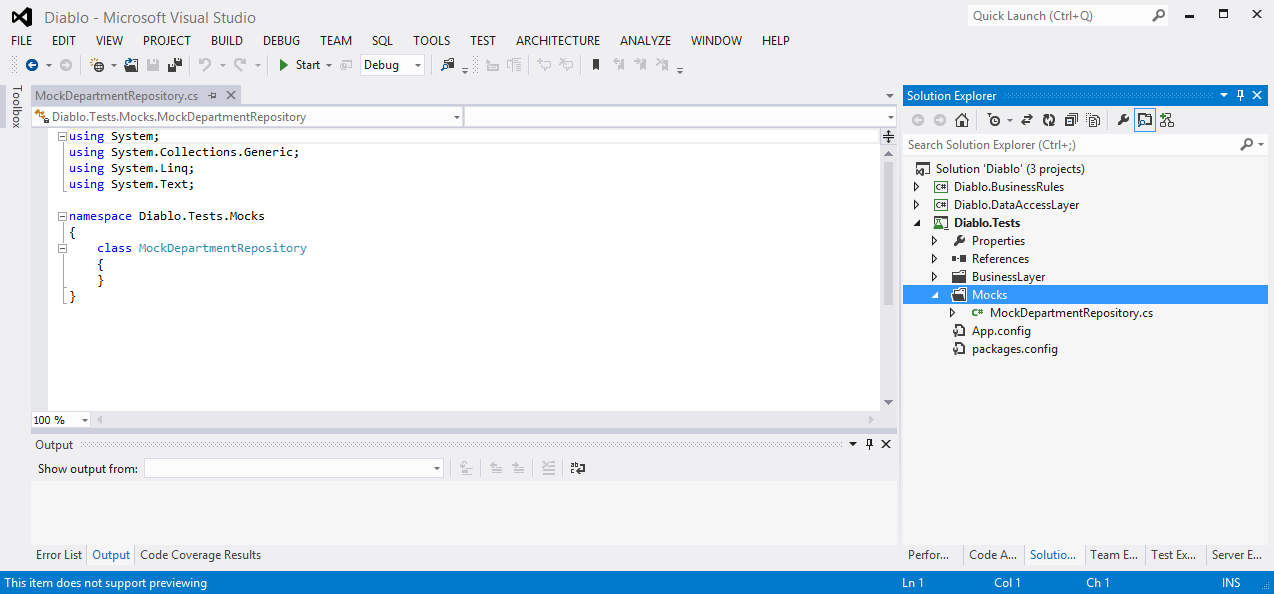
First we changed the objRepository from DepartmentRepository to IDepartmentRepository.

Secondly we created a new constructor that includes the IDepartmentRepository parameter. If we use this constructor when we instantiate the DepartmentController object, then we set the global objRepository variable to the that input parameter. This obviously assumes that the parameter is of the type of IDepartmentRepository.

## Task 3 – Create Your Mock Object

Now we are ready to create our mocked DepartmentRepository object for our unit tests to run.

1. Open the Diablo.Tests project.
2. Create a new a new class file called “MockDepartmentRepository.cs” in the Mocks folder.



1. Modify the class by adding the two using statements above the namespace for our Diablo.DataAccessLayer.Data and Diablo.DataAccessLayer.Data and make sure our class is implementing the IDepartmentRepository interface.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using Diablo.DataAccessLayer.Data;  using Diablo.DataAccessLayer.Models;  namespace Diablo.Tests.Mocks  {  class MockDepartmentRepository: IDepartmentRepository  {  }  } |

If you try and compile your project you will receive an error because you have not implemented the methods that are required by the interface.

1. Add the required methods so that is in complaince to the structure of the implemented interface.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using Diablo.DataAccessLayer.Data;  using Diablo.DataAccessLayer.Models;  namespace Diablo.MvcWebApp.Tests.Mocks  {  class MockDepartmentRepository : IDepartmentRepository  {  public IList<Department> List  {  get  {  return new System.Collections.Generic.List<Department>();  }  }  public IQueryable<Department> QueryList  {  get  {  return new System.Collections.Generic.List<Department>().AsQueryable();  }  }  public void Store(Department department)  {    }  public Department Find(int id)  {  return new Department();  }  public int SaveChanges()  {  return 1;  }  public void Delete(int id)  {    }  }  } |

You will notice that some of our methods require a return type. For those we want to return their respective simple or complex object types. For example, the public Department Find(int id) returns a Department object. This obviously makes sense because if we search for a specific department we will want to return the Department object. From a unit testing perspective we are concerned that the method returns a Departement object but not necessarly concerned about it’s contents. So in this case we simply return a new department object (return new Department();).

Finally we need to add a few new properties to the mock object so we can check certain behaviors in our assertions when writing our unit tests. For example, if the code we are testing is suppose to delete a Department object, we want to make sure that the Delete method is called on the DepartmentRepository class. If the code we are testing does not execute the Delete method then our code is wrong and it would create a unexpected results. A simple way to check and make sure our methods are being executed is to add boolean properties to the class. So if the Delete method is executed we would create a “IsDeleteCalled = true” to the method execution.

|  |
| --- |
| public void Delete(int id)  {  this.IsDeleteCalled = true;  } |

Then in our unit test assertions we can would check to make sure that the IsDeleteCalled is True.

|  |
| --- |
| Assert.IsTrue(objMockRepository.IsDeleteCalled); |

1. Modify the MockDepartmentRepository.cs file so that it looks like this:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using Diablo.DataAccessLayer.Data;  using Diablo.DataAccessLayer.Models;  namespace Diablo.MvcWebApp.Tests.Mocks  {  class MockDepartmentRepository : IDepartmentRepository  {  public bool IsStoreCalled { get; set; }  public bool IsSaveChangesCalled { get; set; }  public bool IsDeleteCalled { get; set; }  public bool IsFindCalled { get; set; }  public bool IsListCalled { get; set; }  public bool IsQueryListCalled { get; set; }  public IList<Department> List  {  get  {  IsListCalled = true;  return new System.Collections.Generic.List<Department>();  }  }  public IQueryable<Department> QueryList  {  get  {  IsQueryListCalled = true;  return new System.Collections.Generic.List<Department>().AsQueryable();  }  }  public void Store(Department department)  {  this.IsStoreCalled = true;  }  public Department Find(int id)  {  this.IsFindCalled = true;  if (id == 99)  {  return null;  }  else  {  return new Department();  }  }  public int SaveChanges()  {  this.IsSaveChangesCalled = true;  return 1;  }  public void Delete(int id)  {  this.IsDeleteCalled = true;  }  }  } |

1. Save and build the test project to ensure you do not have any errors.

## Task 4 – Setting Up Tests To Use The Mock Object

Now that our mock object is in place we need to create the tests so that we pass the mocked object into our business layer that we are testing.

1. Open the DepartmentControllerTest.cs file in the Diablo.Tests project > Controllers folder.
2. Add the following code to DepartmentControllerTest class:

|  |
| --- |
| [TestClass]  public class DepartmentBLTest  {  MockDepartmentRepository objMockRepository;  DepartmentController objController;  [TestInitialize]  public void TestInitialize()  {  objMockRepository = new MockDepartmentRepository();  objController = new DepartmentController(objMockRepository);  }  [TestCleanup]  public void TestCleanUp()  {  objMockRepository = null;  objController = null;  }  } |

In the TestInitialize() method you are creating a new MockDepartmentRepository object and passing it into the DepartmentController constuctor.

|  |
| --- |
| objMockRepository = new MockDepartmentRepository();  objController = new DepartmentController(objMockRepository); |

This tells the controller to use MockDepartmentRepository object rather than the actual DepartmentRepository object.

## Task 5 – Create Unit Tests

1. Open the DepartmentControllerTest.cs file in the Diablo.Tests project > Controller folder.
2. Cut and paste the following method directly under the TestCleanUp() method:

|  |
| --- |
| [TestCleanup]  public void TestCleanUp()  {  objMockRepository = null;  objBusinessObject = null;  }  [TestMethod]  [TestCategory("Unit Tests")]  [TestProperty("Controller", "Department")]  public void DepartmentController\_Index\_Success()  {  //arrange  objMockRepository = new Mocks.MockDepartmentRepository();  objController = new DepartmentController(objMockRepository);  //act  var result = objController.Index();  //assert  Assert.IsTrue(objMockRepository.IsListCalled);  Assert.IsInstanceOfType(((ViewResult)result).Model, typeof(IList<Department>));    result = null;  } |

We are now executing the business layer methods while isolating it from the data persistence layer.

Notice the first assert line:

|  |
| --- |
| //assert  Assert.IsTrue(objMockRepository.IsListCalled); |

This is checking to make sure that when we called the Controller Index; method that it in return called the List() method on the DepartmentRepository class.

Continue to use this method for the remaining controller methods.