|  |
| --- |
| An Introduction to MOQIntroduction This post introduces how we can mock our dependencies and carry on with an uninterrupted unit testing using MOQ. Background Many a times, we see that we have to depend on many external dependencies while development or unit testing our code for example, depending on a group of web service or WCF services providing us some data, stored procedure for some magic values or a Flat/XML file system.  In such a case, it's very difficult to code or unit test your application without hard coding the dependencies reason being those dependencies might not been in place or developed yet or we have been provided only with contracts with no concrete implementation.  In this post, I have decided to take up a simple example of a banking transaction module consisting of entities like Customer class, TransactionManager in business logic layer.  Transaction manager depends on a web service call for validating Social security number for the customer and a data base call for identifying the authentication of the customer. Using the Code 1. We have the following web service contract:  using System;    public interface IValidateSSN  {  bool Validate(string SSN);  }  2. We have the following Data access layer contract:  public interface IValidateCustomer  {  bool Validate(string CustomerID);  }  3. We have the following Customer class:  public class Customer  {  public string SSN{get;set;}  public string FirstName{get;set;}  public string LastName{get;set;}  }  4. We have transaction helper class:  public class TransactionHelper  {  public bool DoTransaction(Customer cust)  {  return true;  }  }  5. We have TransactionManager class:  public class TransactionManager  {  public bool ValidateCustomer(Customer customer)  {  //need to validate customer SSN  // IValidateCustomer.Validate(Custom.SSN);  // but no concrete impementation at place  // need to validate Customer authentication but no concrete implementation  //IValidateCustomer.Validate(Customer.CustomID);    TransactionHelper helper = new TransactionHelper();  bool res=helper.DoTransaction((customer));    return res;  }  }  Completing the unit testing of TransactionManager's validateCustomer method using MOQ . Download MOQ from <http://code.google.com/p/moq/>  Create a test project in VS2010 and refer *MOQ.dll* and your assembly in the test class file:  using MOQ.dll;  using System.Linq;    [TestClass()]  public class TransactionManagerTest  {  [TestMethod()]  public void ValidateCustomerTest()  {  TransactionManager target = new TransactionManager  Customer customer = new Customer();  bool expected = false;  bool actual;    customer.CustomrID = "33333";  var IValidateCustomer = new Mock<ivalidatecustomer>();  IValidateCustomer.Setup(m=>m.Validate(customer.CustomrID)).Returns(true);    var ivalidatessn = new Mock<ivalidatessn>();  ivalidatessn.Setup((m => m.Validate(customer.SSN))).Returns(true);    bool res=target.ValidateCustomer(customer);    Assert.AreEqual(res,true);  }  } Note Here MOQ is responsible for creating a proxy/fake for us and we have used setup method to set the expected results of the fake object.In order to Mock a property/method of a class, we need to mark it with virtual keyword. Interfaces are mockable by default. It comes very handy when you have contracts with you but no concrete implementation or your implementation/behavior is volatile. Conclusion We avoided the two dependencies with the help of MOQ, which according to the contract/interface created a mock object for us and helped us to carry on with our unit testing. |