|  |
| --- |
|  |
| Isolation |
| Mocks, Fakes, and Dependency Injection |

# 

# Isolation

## Objectives

After completing this lab, you should understand how to:

* Isolate a class from its dependencies
* Write unit tests using fakes and mocks
* Use basic features of an IoC container

## Overview

This lab will require some refactoring to reach the point where you can isolate an object for proper testing. There is an OrderProcessor in this application that was written without unit tests. We plan to add new features to the class, but first want to get the existing code under test (so we can be sure the changes don’t break any existing code).

# Getting Started

1. Open the **OrderProcessing** solution in the before folder of the lab.
2. **Run** the application to make sure everything is in working order (you should see some messages on the screen about an order being processed and saved).
3. Open the **OrderProcessor.cs** class. How many dependencies do you see? Is there 2? Could there be 3? We’ll find out as we start testing…

# Test Saving an Order

1. Add a new project to the solution – a **Test Project** named **OrderProcesssing.Tests**
2. Delete the existing .cs file from the new test project.
3. Add a **project reference** from the test project to the OrderProcessing project.
4. Add a new class to the test project – **OrderProcessorTests**.**cs**
5. Add a [TestInitialize] method to the class to to instantiate a private OrderProcessor field.

[TestClass]

public class OrderProcessorTests

{

private OrderProcessor \_processor;

[TestInitialize]

public void TestInitialize()

{

\_processor = new OrderProcessor();

}

}

At this point we want to test that the OrderProcessor saves an order by passing an Order into the data access layer. The way the class is currently designed makes this difficult, because the save operation is a side-effect that happens inside the processor.

Our first step will be to remove the hard dependency on the data access layer. We’ll do this in a way that will allow us to test the OrderProcessor, and still allow the application (Program.cs) to work without modifications.

1. Open **OrderDataAccess.cs**.
2. Use an Extract Interface refactoring to produce an **IOrderDataAccess** interface with a SaveOrder method. You can leave the interface definition in the same file as OrderDataAccess.

public interface IOrderDataAccess

{

void SaveOrder(Order order);

}

1. Back in OrderProcessor.cs, add a private field to the class of type **IOrderDataAccess** and named \_orderDataAccess.
2. Add a **default constructor** to OrderProcessor that initializes \_orderDataAccess to a concrete OrderDataAccess object.
3. Add **a second constructor** to OrderProcess that takes an IOrderDataAccess parameter and assigns the parameter to \_orderDataAccess.

The class should now look a bit like the following.

public class OrderProcessor

{

private IOrderDataAccess \_orderDataAccess;

public OrderProcessor()

{

\_orderDataAccess = new OrderDataAccess();

}

public OrderProcessor(IOrderDataAccess orderDataAccess)

{

\_orderDataAccess = orderDataAccess;

}

. . .

}

1. Change the **SaveOrder** method in the OrderProcessor to use the private field instead of instantiating a class.

private void SaveOrder(Order order)

{

\_orderDataAccess.SaveOrder(order);

}

1. **Run** the application to ensure everything still works in the console window.

We haven’t change the behavior of the application, hopefully, but we can write a test against a better isolated class.

1. In the **Test** project, add a new class named **FakeOrderDataAccess**. The class should implement **IOrderDataAccess**. A saved order can be published to a public list of orders.

public class FakeOrderDataAccess : IOrderDataAccess

{

public void SaveOrder(Order order)

{

SavedOrders.Add(order);

}

public List<Order> SavedOrders = new List<Order>();

}

1. Write a test named SavesAValidOrder that will ensure the processor calls SaveOrder on the fake.

Your test class might look like the following now.

[TestClass]

public class OrderProcessorTests

{

private OrderProcessor \_processor;

**private FakeOrderDataAccess \_orderDataAccess;**

[TestInitialize]

public void TestInitialize()

{

**\_orderDataAccess = new FakeOrderDataAccess();**

**\_processor = new OrderProcessor(\_orderDataAccess);**

}

**[TestMethod]**

**public void SavesAValidOrder()**

**{**

**var order = new Order() { Origin = "MD", Destination="WA" };**

**\_processor.ShipOrder(order);**

**Assert.AreSame(order, \_orderDataAccess.SavedOrders[0]);**

**}**

}

1. **Run** the tests to ensure everything works.
2. Write a test to ensure the processor does not save an invalid order (an order with no destination).

One possible solution:

[TestMethod]

public void DoesNotSaveAnInvalidOrder()

{

var order = new Order() { Origin = "MD", Destination = "" };

try

{

\_processor.ShipOrder(order);

}

catch(ArgumentException){}

Assert.AreEqual(0, \_orderDataAccess.SavedOrders.Count);

}

# Test Logging

1. We want to make sure the order processor writes to the log when processing an order. Follow the same process as the last section to isolate the order processor from the logger. Ensure Program.cs is still working without a modification, and write the test to make sure a log message is saved.

If you run into problems, ask the instructor for help, or look at the solution in the After folder. The test itself will look something like the following.

[TestMethod]

public void Logs\_Message\_When\_Processing()

{

var order = new Order() { Origin = "MD", Destination = "WA" };

\_processor.ShipOrder(order);

Assert.IsTrue(\_fakeLogger.LogMessages[0].Contains("Order"));

}

# Testing Time Stamps

In this section we want to write a test to ensure the processor adds a correct time stamp to the Order. Tests like this can be tricky, because the processor is currently using DateTime.Now, which gives it a dependency on the system clock. System dependencies are not always a problem, but in this case it presents a difficulty for testing. One way to fix this is to use an abstraction like ISystemClock. We’ll explore an alternate approach.

1. Add a static class to the **OrderProcessing** project named **SystemTime**.
2. Give the class a static property named Now that returns a Func<DateTime>.

public static class SystemTime

{

public static Func<DateTime> Now = () => DateTime.Now;

}

1. Write a test that manipulates SystemTime.Now to produce a constant value and Asserts that the ship date matches the constant value. The test will fail until you change **InitiateShipping** to use SystemTime.Now() instead of DateTime.Now (but that’s ok, in this case we can write the failing test first).

[TestMethod]

public void Records\_Correct\_Ship\_Date()

{

SystemTime.Now = () => new DateTime(2008, 6, 1);

var order = new Order() { Origin = "MD", Destination = "WA" };

\_processor.ShipOrder(order);

Assert.AreEqual(SystemTime.Now(), order.ShipDate);

}

# Using Moq

Instead of writing fakes, we’ll use Moq to create Moq objects.

1. Right click the References node of your **test** project. Use “**Manage NuGet Packages**” to download Moq.
2. Add a new class to your test project – **OrderProcessorTestsWithMoq**.

We’ll rewrite the tests we wrote earlier using Moq instead of fakes.

1. Like the last test class, setup a **TestInitialize** method and private variables to hold an OrderProcessor, a mock data access layer, and a mock logger. Instead of using the fake classes we created earlier, **use Mock<T>** (where T is IOrderDataAccess and ILogger).

private OrderProcessor \_processor;

private Mock<IOrderDataAccess> \_mockDataAccess;

private Mock<ILogger> \_mockLogger;

[TestInitialize]

public void TestInitialize()

{

\_mockDataAccess = new Mock<IOrderDataAccess>();

\_mockLogger = new Mock<ILogger>();

\_processor = new OrderProcessor(\_mockDataAccess.Object, \_mockLogger.Object);

}

There are a few different ways to use Moq and mock objects. One approach is to use the Verify API to ensure a method was called.

1. Try the **following** test and running the tests to see what happens.

[TestMethod]

public void Logs\_Message\_When\_Processing()

{

var order = new Order() { Origin = "MD", Destination = "WA" };

\_processor.ShipOrder(order);

\_mockLogger.Verify(l => l.Log("log"));

}

1. The test should fail (because we are saying to expect a parameter “log”). **Replace** the last line of code with the following.

\_mockLogger.Verify(l => l.Log(It.IsAny<string>()));

1. **Run** the test again – it should pass.

For this test the Mock data access object is playing the role of a stub – it implements the interface but doesn’t do anything. We are treating the Mock logger as a real mock – something that records the calls made to it so we can inspect and verify them later.

We can also program expectations into Moq – that is how it should behave when it receives a call. Setups are typically used to specify return values, but can also be used to setup callbacks and exceptions. A callback is useful if you want to capture the exact parameter passed to a method.

1. Rewrite the test with the following code and run it.

[TestMethod]

public void Logs\_Message\_When\_Processing()

{

var order = new Order() { Origin = "MD", Destination = "WA" };

string logMessage = null;

\_mockLogger.Setup(l => l.Log(It.IsAny<string>()))

.Callback((string message) => logMessage = message);

\_processor.ShipOrder(order);

Assert.IsTrue(logMessage.Contains("Order"));

}

1. To test a valid order, we’ll also use the Setup API. When you use Setup, you can use VerifyAll to assert all setups were called as expected.

[TestMethod]

public void SavesAValidOrder()

{

var order = new Order() { Origin = "MD", Destination = "WA" };

\_mockDataAccess.Setup(dal => dal.SaveOrder(order));

\_processor.ShipOrder(order);

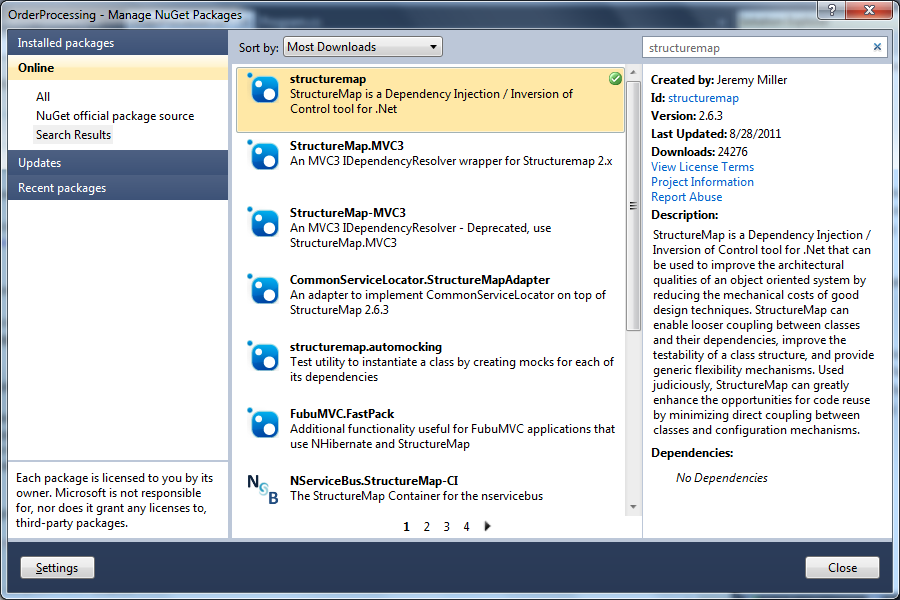
\_mockDataAccess.VerifyAll();

}

1. Make sure both the above tests are passing, and that Program.cs is still running properly.

# Using StructureMap

1. In the **OrderProcessing** project, use NuGet to add **StructureMap** to the project.



1. Add a static method in the **Program** class to configure StructureMap.

private static IContainer ConfigureContainer()

{

var container = new Container(c =>

{

c.For<ILogger>().Use<Logger>();

c.For<IOrderDataAccess>().Use<OrderDataAccess>();

});

return container;

}

1. Use the container in the Main method of the Program to build an OrderProcessor with injected dependencies.

static void Main(string[] args)

{

var container = ConfigureContainer();

var order = new Order()

{

Origin = "MD",

Destination = "WA"

};

var processor = container.GetInstance<OrderProcessor>();

processor.ShipOrder(order);

}

1. Remove the default constructor from OrderProcessor.
2. Run the application and watch it work!

# Conclusion

Congratulations! In this module you learned how to break dependencies, create fake and mock objects, and use an inversion of control container.