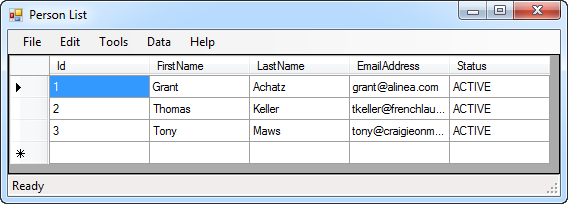
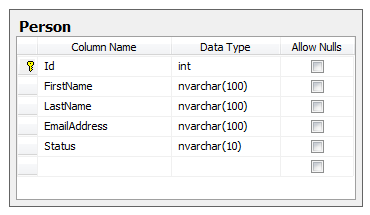
# Refactor Person to Use the Domain Model and Repository Patterns

For this lab, you’ll start with a simple Windows Forms application that is tightly coupled to the database and refactor it to use the Domain Model pattern to represent the business objects and the Repository pattern to encapsulate the data access logic.

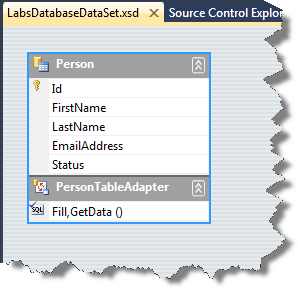
The application selects Person records from the database and displays them in a grid on a form.



The database design consists of a single table named Person.

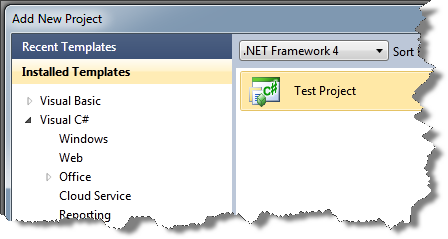


All database access is through a typed-dataset named LabsDatabaseDataSet.xsd.

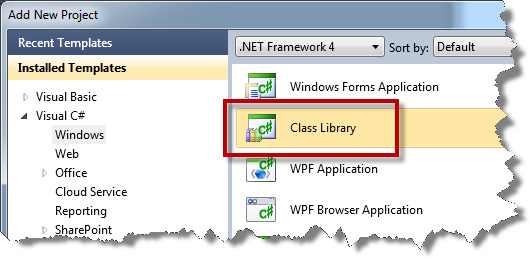


## Create Business Tier and Unit Test Projects

The current application has all of its logic in a single project. For small applications, this isn’t a problem but when your application is starting to grow or you know it’s going to be complex, you’ll want to start breaking the project apart. The current application also has no unit tests which will make it difficult to maintain. Let’s take care of that by adding some new projects to the solution.

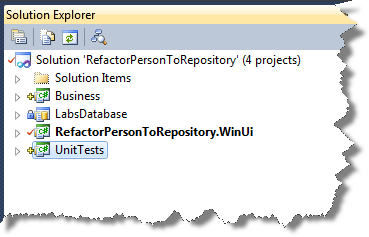


* Add a **Test Project** to the solution named **UnitTests**



* Add a **Class Library** project to the solution named **Business**
* Create a **Project Reference** from the UnitTests project to the Business project
* Create a **Project Reference** from the **RefactorPersonToRepository.WinUi** project to the Business project

Your solution should now look similar to the image below.

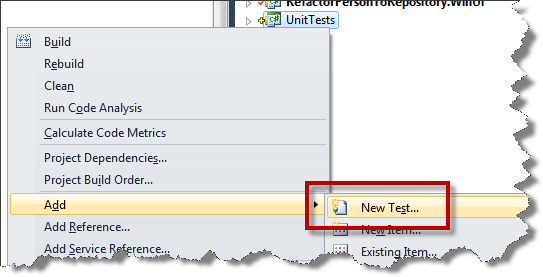


## Create Business-tier Objects for Person Operations

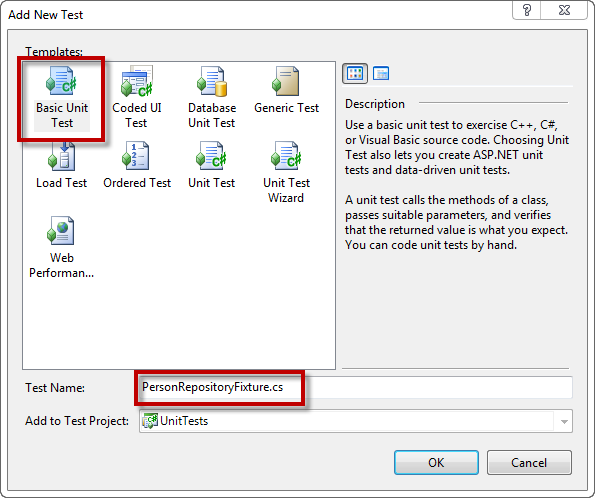
Now that you have a basic solution structure to work with, let’s start working on moving Person functionality in to the Business tier.

The current project has a tight coupling between the user interface, the database design for Person, and the database access implementation. This means that changes to the database immediately ripple in to the user interface tier and it also means that the user interface code is aware of *how* database access is done. For maintenance and testability, it’s a lot better if there’s a healthy layer of abstraction between the user interface, the business tier, and the data access implementation.

Ok. Since we’re doing test-driven development, everything always starts with a unit test.

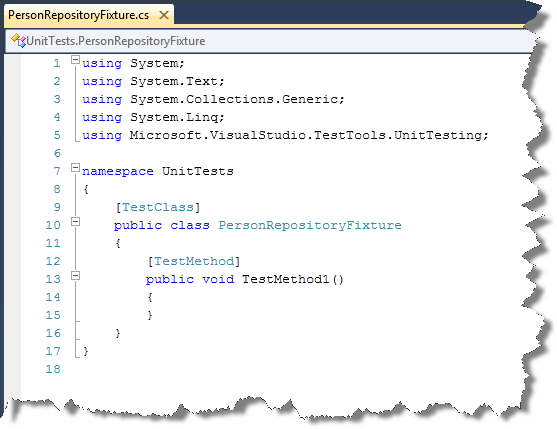


* Add a new test to the **UnitTests** project



* Choose **Basic Unit Test**
* Set the **Test Name** to **PersonRepositoryFixture.cs**
* Click **OK**

You should now see your new PersonRepositoryFixture unit test class.



We’ll start by writing a unit test for logic to get back all available Person records from the person repository. At the most basic level, our test should instantiate a person repository, call a method to get all the Person objects, verify that the result isn’t null, and verify that the result count isn’t zero. This isn’t the detail we really want in our test but we’ll start from here and work our way forward.

[TestClass]

public class PersonRepositoryFixture

{

[TestMethod]

public void PersonRepository\_GetAll()

{  
 DeleteAllPersonsFromDatabase();

CreateTestPersonsInDatabase();

IPersonRepository repository = new PersonRepository();

IList<IPerson> results = repository.GetAll();

Assert.IsNotNull(results, "Repository returned null results.");

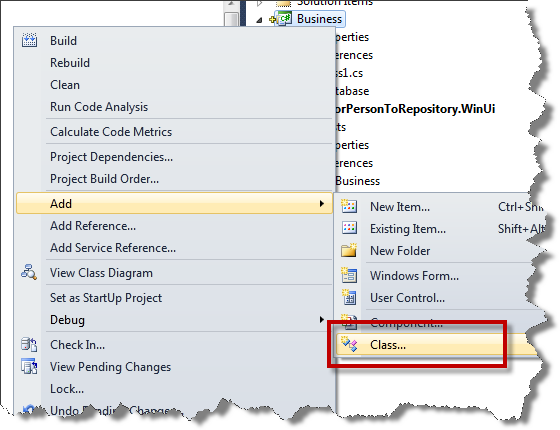
Assert.AreNotEqual<int>(results.Count, 0, "Result count was zero.");

}

}

* Add the highlighted code (shown above) to the **PersonRepositoryFixture**
* Try to compile the code

Ok. As you probably guessed, that code didn’t compile. It didn’t even come close. Let’s start adding code to the **Business** project so that this thing compiles.



* Add a class to the **Business** project named **PersonRepository**
* Add a class to the **Business** project named **Person**

Let’s implement the Person class first.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Business

{

public class Person

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string EmailAddress { get; set; }

public string Status { get; set; }

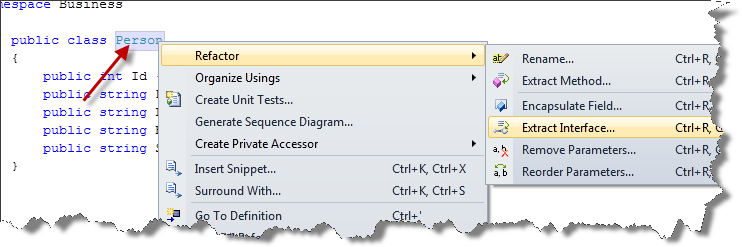
}

}

* Open **Person.cs**
* Add the highlighted code as shown above

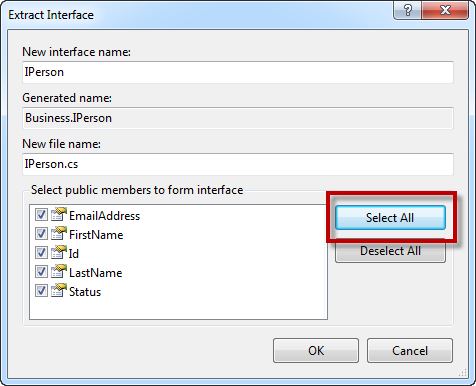
We’ve got the Person class done. (That was easy enough.) If you remember back to the structure of the unit test, the test actually works against **IPerson** rather than **Person**. Not only are we going to be introducing the Repository pattern and testability to this application, we’re going to use interface-driven programming to add some ‘loose coupling’ in to the app. This will also help us to maintain the application over time and decreases the direct dependencies between our tiers.

We probably could have started out by creating an IPerson interface file in the Business project but there’s a helpful feature in Visual Studio called **Extract Interface**. This allows you to take an existing class and create an interface that matches the public signature of the class.



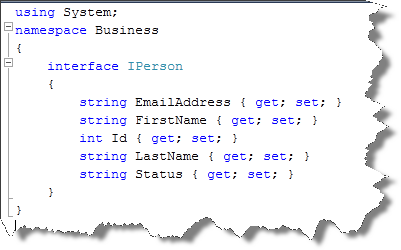
* **Right-click** on the word **Person** in the class definition as shown above
* Choose **Refactor | Extract Interface…**

You should now see the **Extract Interface** dialog. This dialog allows you to choose which properties and methods you want to move to the interface. In our this case, we want all the properties.

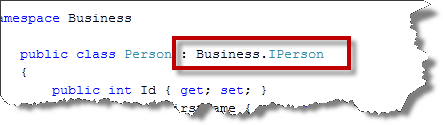


* Click **Select All**
* Click **OK**

You should now see the new IPerson interface that has been added to the Business project.



If you go back to the Person class, you’ll see that this refactoring command automatically added the “: IPerson” that is required to make the Person class officially implement the IPerson interface.



using System;

namespace Business

{

public interface IPerson

{

string EmailAddress { get; set; }

string FirstName { get; set; }

int Id { get; set; }

string LastName { get; set; }

string Status { get; set; }

}

}

* Make **IPerson** public as highlighted above

Now you’ll create the **PersonRepository** class.

public class PersonRepository

{

public IList<IPerson> GetAll()

{

throw new NotImplementedException();

}

}

* Go to the **PersonRepository** class
* Add the highlighted code
* Use **Extract** **Interface** to create an interface named **IPersonRepository**

using System;

namespace Business

{

public interface IPersonRepository

{

System.Collections.Generic.IList<IPerson> GetAll();

}

}

* Make **IPersonRepository**’s visibilty **public** as highlighted above

Looking at IPersonRepository, there’s another refactoring that we should probably do. This won’t be the only domain object that we’ll need to work with using a repository. Whenever you have the same methods working on a different type, it calls out for an interface that uses **generics**.

* Use **Extract Interface** on **IPersonRepository** to create an interface named **IRepository**

using System;

namespace Business

{

public interface IRepository<T>

{

System.Collections.Generic.IList<T> GetAll();

}

}

* Make **IRepository** public
* Change the interface name from IRepository to **IRepository<T>**
* Change the return type on the **GetAll()** method to **IList<T>**

using System;

namespace Business

{

public interface IPersonRepository : IRepository<IPerson>

{

~~System.Collections.Generic.IList<IPerson> GetAll();~~

}

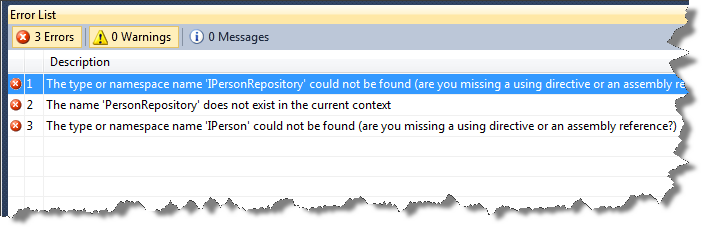
}

* Go back to **IPersonRepository.cs**
* Add the highlighted code as shown above
* Delete any code that is in ~~strikethrough~~

We’ve got the basics of the business tier done. Let’s do a compile to see what we still need to do.

* Compile the solution

Plenty of errors but mostly just missing **using** statements.



using System;

using System.Text;

using System.Collections.Generic;

using System.Linq;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Business;

namespace UnitTests

{

[TestClass]

public class PersonRepositoryFixture

{

…

* Go to **PersonRepositoryFixture**
* Add the missing using statement as shown above
* **Compile** the solution

I’d be surprised if the code compiles. You’re still missing definitions for **DeleteAllPersonsFromDatabase()** and **CreateTestPersonsInDatabase()**.

private void CreateTestPersonsInDatabase()

{

LabsDatabaseDataSet dataset = new LabsDatabaseDataSet();

dataset.Person.AddPersonRow("FN1", "LN1", "ln1@email.com", "status1");

dataset.Person.AddPersonRow("FN2", "LN2", "ln2@email.com", "status2");

dataset.Person.AddPersonRow("FN3", "LN3", "ln3@email.com", "status3");

using (PersonTableAdapter adapter = new PersonTableAdapter())

{

adapter.Update(dataset);

}

}

* Add the **CreateTestPersonsInDatabase()** method to **PersonRepositoryFixture.cs**

private void DeleteAllPersonsFromDatabase()

{

using (PersonTableAdapter adapter = new PersonTableAdapter())

{

var dataset = adapter.GetData();

if (dataset != null && dataset.Count > 0)

{

foreach (var row in dataset)

{

row.Delete();

}

adapter.Update(dataset);

}

}

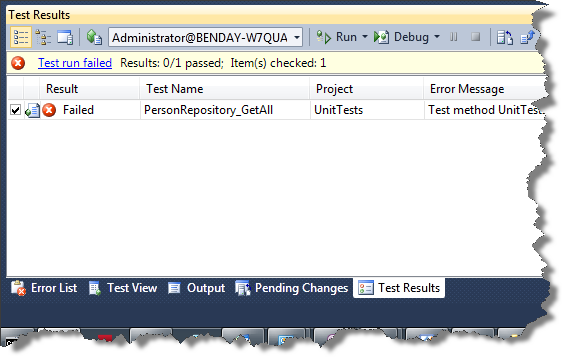
}

* Add the **DeleteAllPersonsFromDatabase()** method to **PersonRepositoryFixture.cs**

The solution should compile.

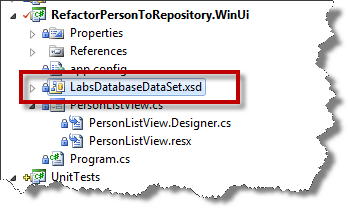
* Go to the **Test View** window
* Run the unit tests

No surprise here. The unit test doesn’t pass because there’s no implementation of the GetAll() method on the PersonRepository.



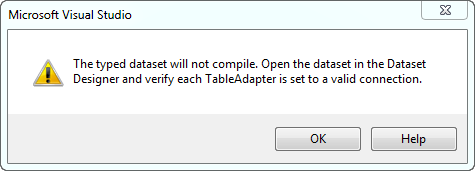
## Remove the Data Access Logic from the User Interface

The purpose of the Repository pattern is to encapsulate data access logic for specific domain model objects. In our case, we’re going to be using the PersonRepository to handle our create, read, update, and delete operations for the Person object. At the moment, our data access logic is in the Windows user interface project and we need to move that logic to the business tier.



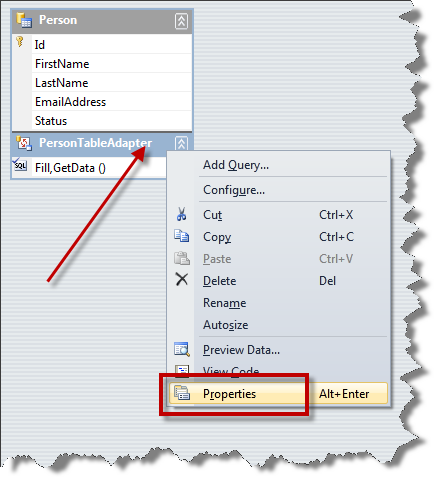
* Go to the **RefactorPersonToRepository.WinUi** project and locate the **LabsDatabaseDataSet.xsd** typed-dataset file
* **Right-click** LabsDatabaseDataset.xsd and choose **Cut**
* Go to the **Business** project, **right-click** Business and choose **Paste**

You should see a warning dialog saying that the dataset will not compile. This is an irritating anti-feature of typed datasets – they’re very difficult to move between projects.



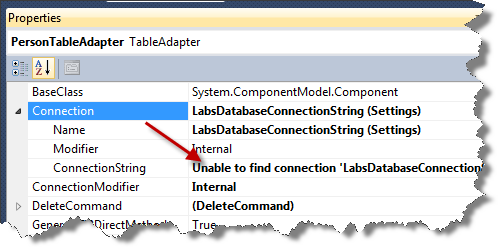
* Click **OK** on the dialog
* **Double-click** on the new **LabsDatabaseDataSet.xsd** in the Business project to open the designer

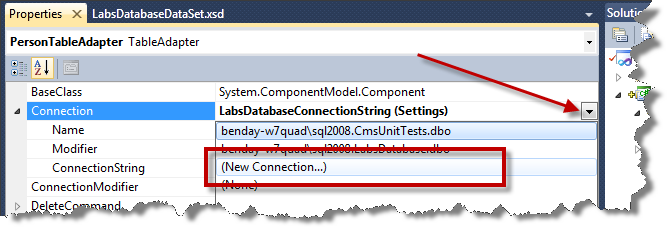
We’re going to have to do a little surgery on our typed dataset’s connection to make it work again.

****

* **Right-click** on the **PersonTableAdapter** and choose **Properties**

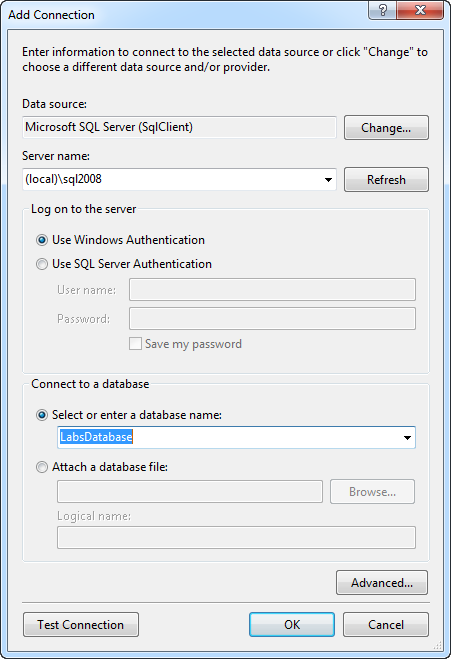
In the **Properties** dialog, you’ll see an error under **Connection** complaining about the LabsDatabaseConnectionString.

****

****

* On **Connection**, clickthearrow to open the drop down list.
* Choose **(New Connection…)** from the list

Now you’ll provide a new database connection for the dataset.

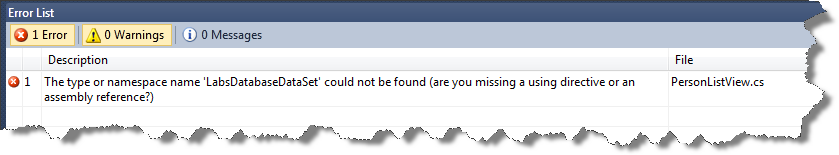
****

* Set **Server name** to the name of your server. (Example: **(local)\sql2008** or **(local)\sqlexpress**)
* Choose **LabsDatabase** from the **Select or enter a database name** box
* Click **OK**

The LabsDatabaseDataSet.xsd has now been successfully moved from the user interface project to the business tier project.

* Build the solution

There’s a problem in PersonListView.cs where the code is pointing to the old LabsDatabaseDataSet.



* Go to **PersonListView.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using Business;

using Business.LabsDatabaseDataSetTableAdapters;

namespace RefactorPersonToRepository.WinUi

{

public partial class PersonListView : Form

* Add the highlighted code as shown above
* Compile the solution
* Fix any other build problems related to PersonTableAdapter
* Compile the solution

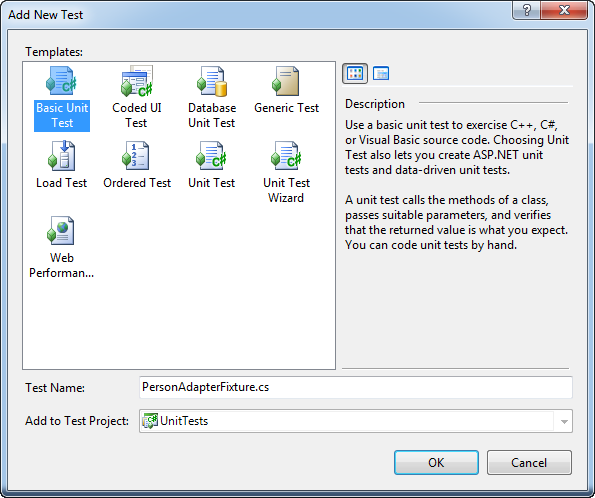
At this point, the solution should build.

## Convert PersonRow to Person and vice versa

When we’re finished with this lab, the user interface will be using the Person class rather than LabsDatabaseDataSet.PersonRow. The PersonRepository will be responsible for calling LabsDatabaseDataSet in order to select PersonRows from the database and will convert the PersonRow instances in to Person object instances.

That conversion logic will need a unit test and our implementation of the conversion will use a design pattern called the **Adapter** **Pattern**. The **Adapter Pattern** contains the logic for taking an instance of one type of object and converting it to another object.

* Go to the **UnitTests** project
* **Right-click** on the **UnitTests** project, choose **Add… | New Test…**



* Choose **Basic Unit Test**
* Set the **Test name** to **PersonAdapterFixture.cs**
* Click **OK**

[TestMethod]

public void PersonAdapter\_PersonRowToPerson()

{

Business.LabsDatabaseDataSet.PersonRow fromPerson =

new LabsDatabaseDataSet().Person.NewPersonRow();

int expectedId = 1234;

string expectedFirstName = "TestFirstName";

string expectedLastName = "TestLastName";

string expectedEmail = "TestEmail";

string expectedStatus = "TestStatus";

fromPerson.Id = expectedId;

fromPerson.FirstName = expectedFirstName;

fromPerson.LastName = expectedLastName;

fromPerson.EmailAddress = expectedEmail;

fromPerson.Status = expectedStatus;

PersonAdapter adapter = new PersonAdapter();

Person toPerson = new Person();

adapter.Adapt(fromPerson, toPerson);

UnitTestUtility.AssertAreEqual(fromPerson, toPerson);

}

* Add the code shown above to **PersonAdapterFixture**

Did you notice the **UnitTestUtility** reference in the previous chunk of code? If you’ve got chunks of utility code that you’ll be re-using throughout your unit tests, it helps to keep everything organized if you put the utility methods in to their own class.

* Create a **Class** in the **UnitTests** project named **UnitTestUtility.cs**

Now you’ll add an implemention of that AssertAreEqual() method.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Business;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTests

{

public static class UnitTestUtility

{

public static void AssertAreEqual(

LabsDatabaseDataSet.PersonRow expected,

IPerson actual)

{

Assert.AreEqual<int>(expected.Id, actual.Id, "Id");

Assert.AreEqual<string>(

expected.FirstName, actual.FirstName, "FirstName");

Assert.AreEqual<string>(

expected.LastName, actual.LastName, "LastName");

Assert.AreEqual<string>(

expected.EmailAddress, actual.EmailAddress, "EmailAddress");

Assert.AreEqual<string>(

expected.Status, actual.Status, "Status");

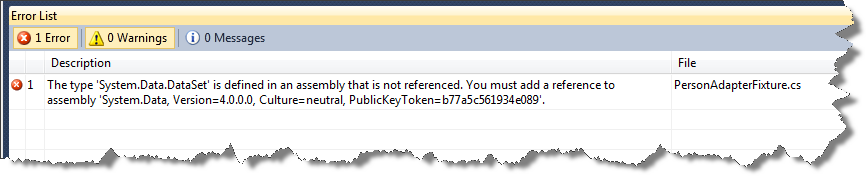
}

}

}

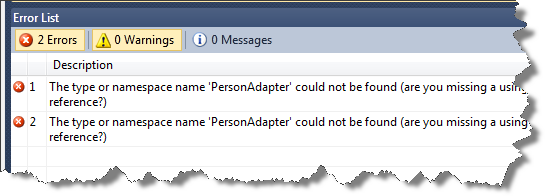
* Add the highlighted code to **UnitTestUtility.cs** as shown above
* Compile the solution

There’s a missing reference in the unit test project.



* Add a reference from the **UnitTests** project to **System.Data.dll**
* Add a reference from the **UnitTests** project to **System.Data.DataSetExtensions.dll**
* Add a reference from the **UnitTests** project to **System.Xml.dll**
* Compile the solution

The only remaining errors should be because **PersonAdapter** doesn’t exist.



* Go to the **Business** project
* Create a new **class** named **PersonAdapter**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Business

{

public class PersonAdapter

{

public void Adapt(LabsDatabaseDataSet.PersonRow fromValue,   
 IPerson toValue)

{

toValue.EmailAddress = fromValue.EmailAddress;

toValue.FirstName = fromValue.FirstName;

toValue.Id = fromValue.Id;

toValue.LastName = fromValue.LastName;

toValue.Status = fromValue.Status;

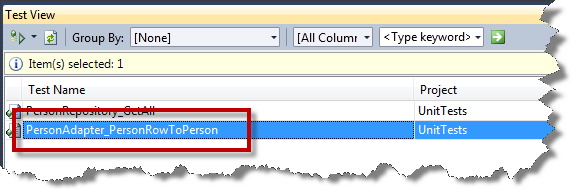
}

}

}

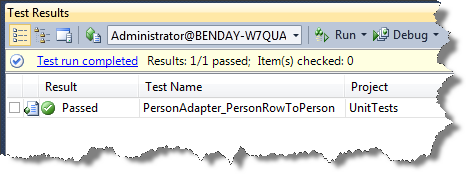
* Add the implementation for the **Adapt()** method as shown above
* Compile the solution

The solution should compile.



* Run the **PersonAdapter\_PersonRowToPerson** unit test

The unit test should pass.



We now have a unit test and a method for adapting single PersonRow objects in to Person objects. When we write the GetAll() method of the PersonRepository, we’ll need to adapt multiple instances in a single call. We need some extra methods on the PersonAdapter and unit tests for those methods.

Here’s the structure of the unit test that will adapt an entire Person data table in to a list of IPerson objects.

[TestMethod]

public void PersonAdapter\_PersonTableToIListOfPerson()

{

LabsDatabaseDataSet dataset = new LabsDatabaseDataSet();

UnitTestUtility.CreatePersonRow(

dataset,

1234,

"FN1", "LN1",

"email1@email.com", "Status1");

UnitTestUtility.CreatePersonRow(

dataset,

4567,

"FN2", "LN2",

"email2@email.com", "Status2");

PersonAdapter adapter = new PersonAdapter();

IList<IPerson> toPersons = new List<IPerson>();

adapter.Adapt(dataset.Person, toPersons);

AssertAreEqual(dataset.Person, toPersons);

}

* Add the code to the **PersonAdapterFixture** class

In order to keep our tests maintainable and to save us some extra typing, here’s another utility method for **UnitTestUtility**. This one will create PersonRow instances and add them to a dataset.

public static void CreatePersonRow(

LabsDatabaseDataSet dataset, int id,

string firstName, string lastName,

string email,

string status)

{

Business.LabsDatabaseDataSet.PersonRow fromPerson =

dataset.Person.NewPersonRow();

fromPerson.Id = id;

fromPerson.FirstName = firstName;

fromPerson.LastName = lastName;

fromPerson.EmailAddress = email;

fromPerson.Status = status;

dataset.Person.AddPersonRow(fromPerson);

}

* Add the code to the **UnitTestUtility** class

This is a utility method that will allow us to assert that a collection of PersonRows is the same as a collection of IPerson objects.

public static void AssertAreEqual(

LabsDatabaseDataSet.PersonDataTable expected,

IList<IPerson> actual)

{

Assert.IsNotNull(expected, "Expected was null.");

Assert.IsNotNull(actual, "Actual was null.");

Assert.AreEqual<int>(expected.Count, actual.Count,

"The number of items does not match.");

for (int index = 0; index < expected.Count; index++)

{

AssertAreEqual(expected[index], actual[index]);

}

}

* Add the code to the **UnitTestUtility** class

Now that we have a unit test, let’s go implement the adapter code in PersonAdapter.

public void Adapt(

LabsDatabaseDataSet.PersonDataTable fromPersons,

IList<IPerson> toPersons)

{

IPerson toPerson;

foreach (var fromPerson in fromPersons)

{

toPerson = new Person();

Adapt(fromPerson, toPerson);

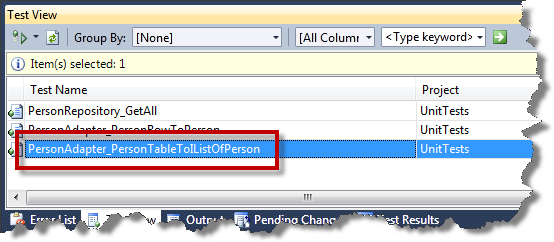
toPersons.Add(toPerson);

}

}

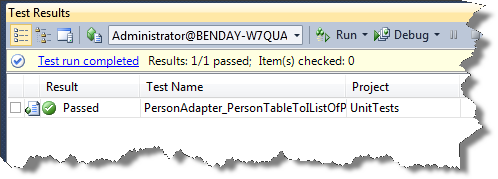
* Add the code above to the **PersonAdapter** class

Now that you have implemented the PersonAdapter method, you should run the unit test.



* Run the **PersonAdapter\_PersonTableToIListOfPerson** unit test

The unit test should pass.



## Implement the Person Repository for the GetAll method

Finally. It’s time to implement PersonRepository’s GetAll() method. Since we’ve already written so much of the supporting logic, the implementation of GetAll() is going to be pretty easy. All you need to do is call the typed-dataset’s PersonTableAdapter to retrieve the records from the database and then pass the results to our PersonAdapter.

**Side note:** You may have noticed that PersonTableAdapter and PersonAdapter both end with the word “adapter”. So, if our PersonAdapter class uses the Adapter design pattern to adapt PersonRows to Person objects, does the PersonTableAdapter also use the Adapter design pattern? The answer is yes. The PersonTableAdapter is adapting the Person database table in to DataRows and vice versa. Design patterns are everywhere. Cool, huh?

* Go to **PersonRepository.cs**

public IList<IPerson> GetAll()

{

IList<IPerson> returnValues = new List<IPerson>();

using (PersonTableAdapter adapter = new PersonTableAdapter())

{

var personData = adapter.GetData();

if (personData.Count > 0)

{

new PersonAdapter().Adapt(

personData, returnValues);

}

}

return returnValues;

}

* Add the above code to **PersonRepository.cs**

Before we run the unit tests, let’s review the code for our **PersonRepository\_GetAll()** unit test.

[TestMethod]

public void PersonRepository\_GetAll()

{

DeleteAllPersonsFromDatabase();

CreateTestPersonsInDatabase();

IPersonRepository repository = new PersonRepository();

IList<IPerson> results = repository.GetAll();

Assert.IsNotNull(results, "Repository returned null results.");

Assert.AreNotEqual<int>(results.Count, 0, "Result count was zero.");

}

Looking at the code for the test, we first empty the database of Person records and then we create some records in the database. This gives us some test data to work with. Then we call in to the repository to retrieve the data and then we check that got back some kind of non-empty result.

What’s missing? We don’t actually check that the data we get back from the repository matches what we put in to the database. We probably should add those checks, huh?

[TestMethod]

public void PersonRepository\_GetAll()

{

DeleteAllPersonsFromDatabase();

LabsDatabaseDataSet dataset = CreateTestPersonsInDatabase();

IPersonRepository repository = new PersonRepository();

IList<IPerson> results = repository.GetAll();

Assert.IsNotNull(results, "Repository returned null results.");

Assert.AreNotEqual<int>(results.Count, 0, "Result count was zero.");

UnitTestUtility.AssertAreEqual(dataset.Person, results);

}

* Add the highlighted code to the **PersonRepository\_GetAll()** unit test method

Change the CreateTestPersonsInDatabase() method to return the dataset that it creates.

private LabsDatabaseDataSet CreateTestPersonsInDatabase()

{

LabsDatabaseDataSet dataset = new LabsDatabaseDataSet();

dataset.Person.AddPersonRow("FN1", "LN1", "ln1@email.com", "status1");

dataset.Person.AddPersonRow("FN2", "LN2", "ln2@email.com", "status2");

dataset.Person.AddPersonRow("FN3", "LN3", "ln3@email.com", "status3");

using (PersonTableAdapter adapter = new PersonTableAdapter())

{

adapter.Update(dataset);

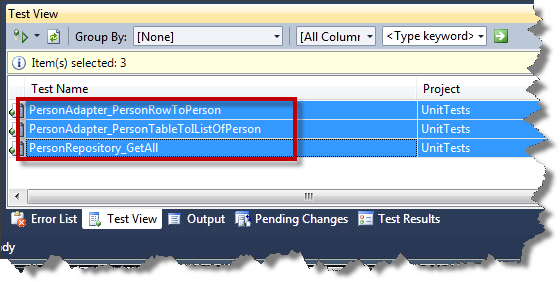
}

return dataset;

}

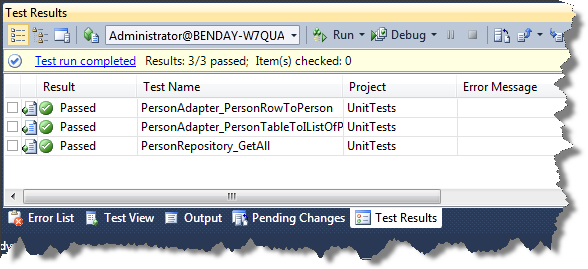
* Add the highlighted code to the **CreateTestPersonsInDatabase()** method

Go ahead and run all the unit tests.



* Go to the **Test View** window
* Run all the unit tests

The unit tests should all pass.



## Refactor the Windows User Interface to Use the Repository

Ok. The repository is implemented and the unit tests pass. The only thing left to do is refactor the Windows Forms user interface to use the repository.

private void LoadPersonsAndPopulateGrid()

{

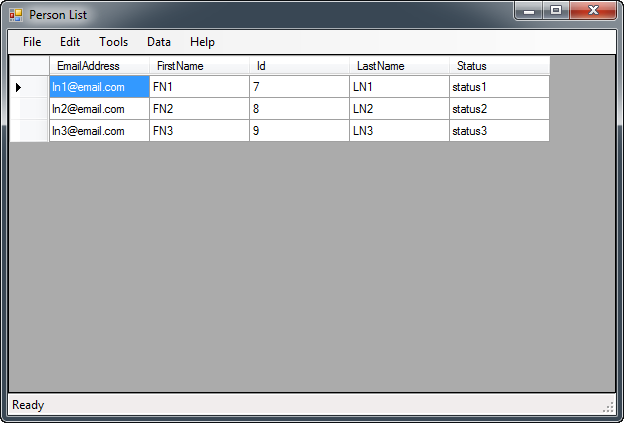
IPersonRepository repository = new PersonRepository();

IList<IPerson> personData = repository.GetAll();

m\_grid.DataSource = personData;

}

* Change the **LoadPersonAndPopulateGrid()** method in **PersonListView.cs** to look like the code above
* Compile the code
* Run the Windows Forms user interface project



You’ve successfully refactored your application to use the Repository pattern for data access.