Design Patterns

Model View Presenter

http://i.msdn.microsoft.com/Global/Images/clear.gif Contents

[Following the MVP](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S1)   
[Making the First Test Pass](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S2)   
[Filling the DropDownList](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S3)   
[Implementing the View Interface](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S4)   
[What's Next?](http://msdn.microsoft.com/en-us/magazine/cc188690.aspx#S5)

As UI-creation technologies such as ASP.NET and Windows® Forms become more and more powerful, it's common practice to let the UI layer do more than it should. Without a clear separation of responsibilities, the UI layer can often become a catch-all for logic that really belongs in other layers of the application. One design pattern, the Model View Presenter (MVP) pattern, is especially well suited to solving this problem. In order to illustrate my point, I will build a display screen that follows the MVP pattern for customers in the Northwind database.

Why is it bad to have lots of logic in the UI layer? The code in the UI layer of an application is very difficult to test without either running the application manually or maintaining ugly UI runner scripts that automate the execution of UI components. While this is a big problem in itself, an even bigger problem is the reams of code that are duplicated between common views in an application. It can often be hard to see good candidates for refactoring when the logic to perform a specific business function is copied among different pieces in the UI layer. The MVP design pattern makes it much easier to factor logic and code out of the UI layer for more streamlined, reusable code that's easier to test.

**Figure 1** shows the main layers that make up the sample application. Notice that there are separate packages for UI and presentation. You might have expected them to be the same, but actually the UI layer of a project should consist only of the various UI elements—forms and controls. In a Web Forms project this is typically a collection of ASP.NET Web Forms, user controls, and server controls. In Windows Forms, it is a collection of Windows Forms, user controls, and third-party libraries. This extra layer is what keeps the display and the logic separate. In the presentation layer you have the objects that actually implement the behavior for the UI—things like validation display, collection input from the UI, and so forth.

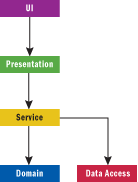


Figure 1**Application Architecture**

**Following the MVP**

As you can see in **Figure 2**, the UI for this project is pretty standard. When the page loads, the screen will display a dropdown box filled with all of the customers in the Northwind database. If you select a customer from the dropdown list, the page will update to display the information for that customer. By following the MVP design pattern you can factor behaviors out of the UI and into their own classes. **Figure 3** shows a class diagram that indicates the association between the different classes that are involved.

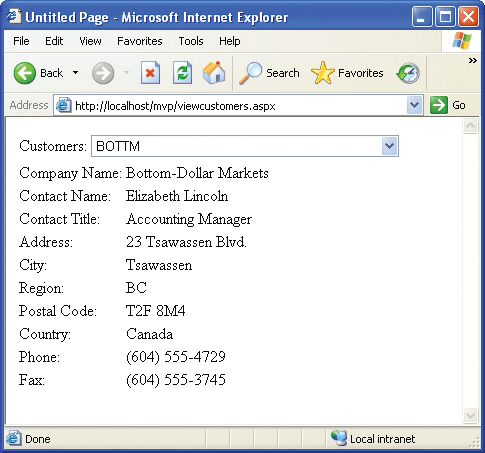


Figure 2**Customer Information**

It's important to note that the presenter has no knowledge of the actual UI layer of the application. It knows it can talk to an interface, but it does not know or care what the implementation of that interface is. This promotes reuse of presenters between disparate UI technologies.

I am going to use Test Driven Development (TDD) to build the functionality of the customer screen. **Figure 4** shows the details for the first test I will use to describe the behavior I expect to observe on page load. TDD lets me focus on one problem at a time, write just enough code to make the test pass, and then carry on. In this test I am making use of a mock object framework called NMock2 that allows me to build mock implementations of interfaces.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 4 The First Test

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl39_ctl00_ctl00_code');" \o "Copy Code)

[Test]

public void ShouldLoadListOfCustomersOnInitialize()

{

mockery = new Mockery();

ICustomerTask mockCustomerTask = mockery.NewMock<ICustomerTask>();

IViewCustomerView mockViewCustomerView =

mockery.NewMock<IViewCustomerView>();

ILookupList mockCustomerLookupList = mockery.NewMock<ILookupList>();

ViewCustomerPresenter presenter =

new ViewCustomerPresenter(mockViewCustomerView,

mockCustomerTask);

ILookupCollection mockLookupCollection =

mockery.NewMock<ILookupCollection>();

Expect.Once.On(mockCustomerTask).Method(

"GetCustomerList").Will(Return.Value(mockLookupCollection));

Expect.Once.On(mockViewCustomerView).GetProperty(

"CustomerList").Will(Return.Value(mockCustomerLookupList));

Expect.Once.On(mockLookupCollection).Method(

"BindTo").With(mockCustomerLookupList);

presenter.Initialize();

}

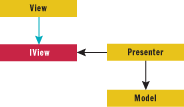


Figure 3**MVP Class Diagram**

In my MVP implementation, I have decided that the presenter is going to take as a dependency the view it is going to work with. It is always good to create objects in a state that enables them to do their work immediately. In this application, the presentation layer is dependent on the service layer to actually invoke the domain functionality. Because of this requirement it also makes sense to construct a presenter with an interface to a service class that it can talk to. This ensures that once a presenter is constructed, it is ready to do all of the work it needs to do. I start off by creating two specific mocks: one for the service layer and one for the view that the presenter will work with.

Why mocks? A rule of unit testing is to isolate the test as much as possible to focus on one specific object. In this test I am only interested in the expected behavior of the presenter. At this point I don't care about the actual implementation of the view interface or service interface; I trust the contracts defined by those interfaces and set the mocks to behave accordingly. This ensures that I focus my test solely around the behavior I expect of the presenter, not of any of its dependencies. The behavior I expect the presenter to exhibit after its initialize method is invoked is as follows.

First, the presenter should make one call to the GetCustomerList method on the ICustomerTask service layer object (mocked in the test). Notice that with the use of NMock I can simulate behavior of the mock. In the case of the service layer, I want it to return a mock ILookupCollection to the presenter. Then, after the presenter retrieves the ILookupCollection from the service layer, it should invoke the collection's BindTo method and pass the method an implementation of an ILookupList. By using the NMockExpect.Once method I can be sure that the test will fail if the presenter does not call the method once and only once.

After writing that test I am in a completely non-compilable state. I'm going to do the simplest thing possible to get the test to pass.

**Making the First Test Pass**

One of the advantages of writing the test first is that I now have a blueprint (the test) I can follow to get the test to compile and eventually pass. The first test has two interfaces that do not yet exist. These interfaces are the first prerequisites for getting the code to compile correctly. I'll start with the code for IViewCustomerView:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl08_code');" \o "Copy Code)

public interface IViewCustomerView

{

ILookupList CustomerList { get; }

}

This interface exposes one property that returns an ILookupList interface implementation. I don't yet have an ILookupList interface or even an implementer, for that matter. For the purpose of getting this test to pass I don't need an explicit implementor, so I can proceed to create the ILookupList interface:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl09_code');" \o "Copy Code)

public interface ILookupList { }

At this point the ILookupList interface looks pretty useless. My goal is to get the test to compile and pass, and these interfaces satisfy the test's requirements. It's time to shift focus to the object that I'm actually testing—the ViewCustomerPresenter. This class does not exist yet, but looking at the test you can glean two important facts about it: it has a constructor that requires both a view and service implementation as dependencies, and it has a void Initialize method. The code in **Figure 5** shows how to compile the test.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 5 Compiling the Test

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl42_ctl00_ctl00_code');" \o "Copy Code)

public class ViewCustomerPresenter

{

private readonly IViewCustomerView view;

private readonly ICustomerTask task;

public ViewCustomerPresenter(

IViewCustomerView view, ICustomerTask task)

{

this.view = view;

this.task = task;

}

public void Initialize()

{

throw new NotImplementedException();

}

}

Remember that a presenter requires all of its dependencies in order to do its work meaningfully; this is why the view and service are passed in. I have not implemented the initialize method so if I run the test I get a NotImplementedException.

As mentioned already, I am not coding the presenter blindly; I already know, from looking at the test, what behavior the presenter should exhibit once the initialize method is called. The implementation of that behavior is as follows:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl12_code');" \o "Copy Code)

public void Initialize()

{

task.GetCustomerList().BindTo(view.CustomerList);

}

In the source code accompanying this article there is a complete implementation of the GetCustomerList method in the CustomerTask class (which implements the ICustomerTask interface). From the perspective of implementing and testing the presenter, though, I don't need to know whether there is a working implementation yet. It is this level of abstraction that allows me to plow through with the testing of the presenter class. The first test is now in a state that will compile and run. This proves that when the Initialize method on the presenter is called, it will interact with its dependencies in the manner that I specified in the test, and ultimately when the concrete implementations of those dependencies are injected into the presenter, I can be sure that the resulting view (the ASPX page) will be filled with a list of Customers.

**Filling the DropDownList**

So far I have been dealing mainly with interfaces to abstract the actual implementation details away, allowing the focus to be on the presenter. It is now time to build some of the plumbing that will ultimately allow the presenter to populate a list on a Web page in a way that can be tested. The key to making this work is the interaction that will occur in the BindTo method of the LookupCollection class. If you take a look at the implementation of the LookupCollection class in **Figure 6**, you will notice that it implements the ILookupCollection interface. The source code for the article has the accompanying tests that were used to build up the functionality of the LookupCollection class.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 6 The LookupCollection Class

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl44_ctl00_ctl00_code');" \o "Copy Code)

public class LookupCollection : ILookupCollection

{

private IList<ILookupDTO> items;

public LookupCollection(IEnumerable<ILookupDTO> items)

{

this.items = new List<ILookupDTO>(items);

}

public int Count { get { return items.Count; } }

public void BindTo(ILookupList list)

{

list.Clear();

foreach (ILookupDTO dto in items) list.Add(dto);

}

}

The implementation of the BindTo method is of particular interest. Notice that in this method the collection iterates through its own private list of ILookupDTO implementations. An ILookupDTO is an interface that caters well to binding to comboboxes in the UI layer:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl15_code');" \o "Copy Code)

public interface ILookupDTO

{

string Value { get; }

string Text { get; }

}

**Figure 7** shows the code that tests the lookup collection's BindTo method, which will help explain the expected interaction between a LookupCollection and an ILookupList. The last line is of particular interest. In this test I expect that before attempting to add items to the list, the LookupCollection will invoke the Clear method on the ILookupList implementation. I then expect Add to be called on an ILookupList 10 times, and as an argument to the Add method, the LookupCollection will pass in an object that implements the ILookupDTO interface. To make this actually work with a control that lives in a Web project (such as a dropdown listbox) you will need to create an implementation of ILookupList that knows how to work with controls in a Web project.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 7 A Test that Describes Behavior

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl46_ctl00_ctl00_code');" \o "Copy Code)

[Test]

public void ShouldBeAbleToBindToLookupList()

{

IList<ILookupDTO> dtos = new IList;

ILookupList mockLookupList = mockery.NewMock<ILookupList>();

Expect.Once.On(mockLookupList).Method("Clear");

for (int i = 0; i < 10; i++)

{

SimpleLookupDTO dto =

new SimpleLookupDTO(i.ToString(),i.ToString());

dtos.Add(dto);

Expect.Once.On(mockLookupList).Method("Add").With(dto);

}

new LookupCollection(dtos).BindTo(mockLookupList);

}

The source code that accompanies this article contains a project named MVP.Web.Controls. This project contains any Web-specific controls or classes that I chose to create to complete the solution. Why did I place the code in this project and not in the APP\_CODE directory or the Web project itself? Testability. Anything that lives in the Web project is difficult to test directly without either running the application manually or automating the UI using some sort of testing robot. The MVP pattern allows me to think at a higher level of abstraction and test implementations of the core interfaces (ILookupList and ILookupCollection) without manually running the application. I am going to add a new class, a WebLookupList control, to the Web.Controls project. **Figure 8** shows the first test for this class.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 8 First Test for WebLookupList Control

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl47_ctl00_ctl00_code');" \o "Copy Code)

[Test]

public void ShouldAddItemToUnderlyingList()

{

ListControl webList = new DropDownList();

ILookupList list = new WebLookupList(webList);

SimpleLookupDTO dto = new SimpleLookupDTO("1","1");

list.Add(dto);

Assert.AreEqual(1, webList.Items.Count);

Assert.AreEqual(dto.Value, webList.Items[0].Value);

Assert.AreEqual(dto.Text, webList.Items[0].Text);

}

Some things stand out in the test that is shown in **Figure 8**. The test project clearly needs a reference to the System.Web library so that it can instantiate DropDownList Web controls. Looking at the test, you should see that the WebLookupList class will implement the ILookupList interface. It is also going to take a ListControl as a dependency. Two of the most common ListControl implementations in the System.Web.UI.WebControls namespace are the DropDownList and the ListBox classes. A key feature of the test in **Figure 8** is the fact that I am ensuring that a WebLookupList correctly updates the state of an actual Web ListControl to which it is delegating responsibility. **Figure 9** shows the class diagram for the classes involved in the WebLookupList implementation. I can satisfy the requirements of the first test for the WebLookupList control with the code in **Figure 10**.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 10 WebLookupList Control

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl48_ctl00_ctl00_code');" \o "Copy Code)

public class WebLookupList : ILookupList

{

private ListControl underlyingList;

public WebLookupList(ListControl underlyingList)

{

this.underlyingList = underlyingList;

}

public void Add(ILookupDTO dto)

{

underlyingList.Items.Add(new ListItem(dto.Text, dto.Value));

}

}

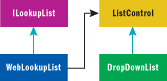


Figure 9**WebLookupList Class**

Remember, one of the keys to MVP is the separation of layers introduced by the creation of a view interface. The presenter doesn't know what implementation of a view, and respectively an ILookupList, it will be talking to; it just knows that it will be able to call any of the methods defined by those interfaces. Ultimately, the WebLookupList class is a class that wraps and delegates to an underlying ListControl (base class for some of the ListControls defined in the System.Web.UI.WebControls project). With that code now in place, I can compile and run the WebLookupList control test which should pass. I can add one more test for the WebLookupList that tests the actual behavior of the clear method:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl23_code');" \o "Copy Code)

[Test]

public void ShouldClearUnderlyingList()

{

ListControl webList = new DropDownList();

ILookupList list = new WebLookupList(webList);

webList.Items.Add(new ListItem("1", "1"));

list.Clear();

Assert.AreEqual(0, webList.Items.Count);

}

Again I am testing that the WebLookupList class will actually change the state of the underlying ListControl (DropDownList) when its own methods are invoked. The WebLookupList is now feature complete for the purposes of populating a DropDownList in a Web Form. It is now time for me to wire everything together and get the Web page's dropdown filled with a list of customers.

**Implementing the View Interface**

Because I am building a Web Forms front end, it makes sense that the implementer for the IViewCustomerView interface would be a Web Form or a user control. For the purpose of this column, I'll make it a Web Form. The general appearance of the page has already been created, as you saw in **Figure 2**. Now I need only to implement the view interface. Switching to the codebehind for the ViewCustomers.aspx page, I can add the following code, indicating that the page is required to implement the IViewCustomersView interface:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl24_code');" \o "Copy Code)

public partial class ViewCustomers : Page,IViewCustomerView

If you look at the code sample, you'll notice that the Web project and the Presentation are two completely different assemblies. Also, the Presentation project has no reference whatsoever to the Web.UI project, further maintaining the layer of separation. On the other hand, the Web.UI project has to have a reference to the Presentation project, as that is where the View interface and the presenter live.

By choosing to implement the IViewCustomerView interface, our Web page now has a responsibility to implement any methods or properties defined by that interface. Currently there is only one property on the IViewCustomerView interface and that is a getter that returns any implementation of an ILookupList interface. I added a reference to the Web.Controls project so that I can instantiate a WebLookupListControl. I did this because the WebLookupListControl implements the ILookupList interface and it knows how to delegate to actual WebControls that live in ASP.NET. Taking a look at the ASPX for the ViewCustomer page, you will see that the list of customers is simply an asp:DropDownList control:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl25_code');" \o "Copy Code)

<td>Customers:</td>

<td><asp:DropDownList id="customerDropDownList" AutoPostBack="true"

runat="server" Width="308px"></asp:DropDownList></td>

</tr>

With this already in place, I can quickly continue to implement the code required to satisfy the implementation of the IViewCustomerView interface:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26_code');" \o "Copy Code)

public ILookupList CustomerList

{

get { return new WebLookupList(this.customerDropDownList);}

}

I now need to invoke the Initialize method on the presenter that will trigger it to actually do some work. To do that, the view needs to be able to instantiate the presenter so that methods on it can be invoked. If you look back to the presenter, you'll remember that it requires both a view and a service that it will work with. The ICustomerTask interface represents an interface that lives in the service layer of the application. Service layers are typically responsible for orchestrating the interaction between domain objects and converting the results of those interactions into Data Transfer Objects (DTOs) that are then passed from the service layer to the presentation layer and then to the UI layer. There is a problem, however—I have stipulated that the presenter needs to be constructed with both the view and service implementations.

The actual instantiation of the presenter is going to take place in the codebehind for the Web page. This is a problem, because the UI project has no reference to the service layer project. The presentation project does, however, have a reference to the service layer project. This allows me to solve the problem by adding an overloaded constructor to the ViewCustomerPresenterClass:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl27_code');" \o "Copy Code)

public ViewCustomerPresenter(IViewCustomerView view) :

this(view, new CustomerTask()) {}

This new constructor satisfies the presenter's requirement for implementations of both the view and the service, while also maintaining the separation of the UI layer from the service layer. It is now fairly trivial to finish off the code for the codebehind:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl28_code');" \o "Copy Code)

protected override void OnInit(EventArgs e)

{

base.OnInit(e);

presenter = new ViewCustomerPresenter(this);

}

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack) presenter.Initialize();

}

Notice the key to the instantiation of the presenter is the fact that I am making use of the newly created overload for the constructor, and the Web Form passes itself in as an object that implements the view interface!

With the code for the codebehind implemented, I can now build and run the application. The DropDownList on the Web page is now filled with a list of customer names without the need for any databinding code in the codebehind. Plus, scores of tests have been run on all the pieces that ultimately work together, ensuring that the presentation layer architecture will behave as expected.

I'm going to wrap up my discussion of MVP by showing what is required to display customer information for a customer selected in the DropDownList. Once again, I start by writing a test that describes the behavior I hope to observe (see **Figure 11**).

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 11 One Last Test

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl55_ctl00_ctl00_code');" \o "Copy Code)

[Test]

public void ShouldDisplayCustomerDetails()

{

SimpleLookupDTO lookupDTO = new SimpleLookupDTO("1","JPBOO");

CustomerDTO dto = new CustomerDTO("BLAH", "BLAHCOMPNAME",

"BLAHCONTACTNAME", "BLAHCONTACTTILE", "ADDRESS", "CITY",

"REGION", "POSTALCODE", Country.CANADA, "4444444", "4444444");

Expect.Once.On(mockViewCustomerView).GetProperty(

"CustomerList").Will(Return.Value(mockCustomerLookupList));

Expect.Once.On(mockCustomerLookupList).GetProperty(

"SelectedItem").Will(Return.Value(lookupDTO));

Expect.Once.On(mockCustomerTask).Method(

"GetDetailsForCustomer").With(1).Will(Return.Value(dto));

Expect.Once.On(mockViewCustomerView).SetProperty(

"CompanyName").To(dto.CompanyName);

Expect.Once.On(mockViewCustomerView).SetProperty(

"ContactName").To(dto.ContactName);

Expect.Once.On(mockViewCustomerView).SetProperty(

"ContactTitle").To(dto.ContactTitle);

Expect.Once.On(mockViewCustomerView).SetProperty(

"Address").To(dto.Address);

Expect.Once.On(mockViewCustomerView).SetProperty(

"City").To(dto.City);

Expect.Once.On(mockViewCustomerView).SetProperty(

"Region").To(dto.Region);

Expect.Once.On(mockViewCustomerView).SetProperty(

"PostalCode").To(dto.PostalCode);

Expect.Once.On(mockViewCustomerView).SetProperty(

"Country").To(dto.CountryOfResidence.Name);

Expect.Once.On(mockViewCustomerView).SetProperty(

"Phone").To(dto.Phone);

Expect.Once.On(mockViewCustomerView).SetProperty("Fax").To(dto.Fax);

presenter.DisplayCustomerDetails();

}

As before, I am taking advantage of the NMock library to create mocks of the task and view interfaces. This particular test verifies the behavior of the presenter by asking the service layer for a DTO representing a particular customer. Once the presenter retrieves the DTO from the service layer, it will update properties on the view directly, thus eliminating the need for the view to have any knowledge of how to correctly display the information from the object. For brevity I am not going to discuss the implementation of the SelectedItem property on the WebLookupList control; instead I will leave it to you to examine the source code to see the implementation details. What this test really demonstrates is the interaction that occurs between the presenter and the view once the presenter retrieves a CustomerDTO from the service layer. If I attempt to run the test now, I will be in a big failure state as a lot of the properties don't yet exist on the view interface. So I'll go ahead and add the necessary members to the IViewCustomerView interface, as you see in **Figure 12**.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 12 Completing the IVewCustomerView Interface

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl56_ctl00_ctl00_code');" \o "Copy Code)

public interface IViewCustomerView

{

ILookupList CustomerList{get;}

string CompanyName{set;}

string ContactName{set;}

string ContactTitle{set;}

string Address{set;}

string City{set;}

string Region{set;}

string PostalCode{set;}

string Country{set;}

string Phone{set;}

string Fax{set;}

}

Immediately after adding these interface members, my Web Form complains because it is no longer fulfilling the contract of the interface, so I have to go back to the codebehind for my Web Form and implement the remaining members. As stated before, the entire markup for the Web page has already been created, as have the table cells which have been marked with the "runat=server" attribute and are named according to the information that should be displayed in them. This makes the resulting code to implement the interface members very trivial:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl33_code');" \o "Copy Code)

public string CompanyName

{

set { this.companyNameLabel.InnerText = value; }

}

public string ContactName

{

set { this.contactNameLabel.InnerText = value; }

}

...

With the setter properties implemented, there is just one thing left to do. I need a way to tell the presenter to display the information for the selected customer. Looking back at the test, you can see that the implementation of this behavior will live in the DisplayCustomerDetails method on the presenter. This method will not, however, take any arguments. When invoked, the presenter will turn back around to the view, pull from it any information it needs (which it will retrieve by using the ILookupList), and then use that information to retrieve the details about the customer in question. All that I need to do from a UI perspective is set the AutoPostBack property of the DropDownList to true, and I also need to add the following event handler hookup code to the OnInit method of the page:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl34_code');" \o "Copy Code)

protected override void OnInit(EventArgs e)

{

base.OnInit(e);

presenter = new ViewCustomerPresenter(this);

this.customerDropDownList.SelectedIndexChanged += delegate

{

presenter.DisplayCustomerDetails();

};

}

This event handler ensures that whenever a new customer is selected in the dropdown, the view will ask the presenter to display the details for that customer.

It is important to note that this is typical behavior. When a view asks a presenter to do something, it asks without giving any specific details, and it is up to the presenter to return to the view and get any information it needs using the view interface. **Figure 13** shows the code required to implement the required behavior in the presenter.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 13 Completing the Presenter

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl59_ctl00_ctl00_code');" \o "Copy Code)

public void DisplayCustomerDetails()

{

int? customerId = SelectedCustomerId;

if (customerId.HasValue)

{

CustomerDTO customer =

task.GetDetailsForCustomer(customerId.Value);

UpdateViewFrom(customer);

}

}

private int? SelectedCustomerId

{

get

{

string selectedId = view.CustomerList.SelectedItem.Value;

if (String.IsNullOrEmpty(selectedId)) return null;

int? id = null;

try

{

id = int.Parse(selectedId.Trim());

}

catch (FormatException) {}

return id;

}

}

private void UpdateViewFrom(CustomerDTO customer)

{

view.CompanyName = customer.CompanyName;

view.ContactName = customer.ContactName;

view.ContactTitle = customer.ContactTitle;

view.Address = customer.Address;

view.City = customer.City;

view.Region = customer.Region;

view.Country = customer.CountryOfResidence.Name;

view.Phone = customer.Phone;

view.Fax = customer.Fax;

view.PostalCode = customer.PostalCode;

}

Hopefully you now see the value of adding the presenter layer. It is the presenter's responsibility to attempt to retrieve an ID for a customer for whom it needs to display details. This is code that would normally have been performed in the codebehind, but is now inside a class that I can fully test and exercise outside of any presentation-layer technology.

In the event that the presenter can retrieve a valid customer ID from the view, it turns to the service layer and asks for a DTO that represents the details for the customer. Once the presenter has the DTO in hand, it updates the view with the information contained in the DTO. A key point to note is the simplicity of the view interface; aside from the ILookupList interface, the view interface consists entirely of string DataTypes. It is ultimately the responsibility of the presenter to correctly convert and format the information retrieved from the DTO so that it can actually be handed to the view as a string. Although not demonstrated in this example, the presenter would also be responsible for reading information from the view and converting it to the necessary types that the service layer would expect.

With all the pieces in place I can now run the application. When the page first loads, I get a list of customers and the first customer appears (not selected) in the DropDownList. If I select a customer, a postback occurs, the interaction between the view and the presenter takes place, and the Web page is updated with the related customer information.

**What's Next?**

The Model View Presenter design pattern is really just a fresh take on the Model View Controller pattern that many developers are already familiar with; the key distinction is that MVP truly separates the UI from the domain/service layer of the application. Although this example was fairly simple from a requirements perspective, it should help you abstract the interaction between a UI and the other layers of your applications. Also, you should now see ways that you can use these layers of indirection to make your application more testable with automation. As you delve deeper into the MVP pattern, I hope you'll find other ways to pull as much formatting and conditional logic out of your codebehinds and place them into testable view/presenter interaction models.