Lab 3: Creating a View

# Purpose

Estimated time to complete this lab: **25 minutes**.

In this lab, you will learn how to use the Smart Client Development guidance package to create a view in a business module and show the view in the shell form.

After completing this lab, you will be able to:

* + Add a new Composite UI Application Block view to a smart client application using the guidance package.
  + Implement the Model-View-Presenter pattern in smart client applications.

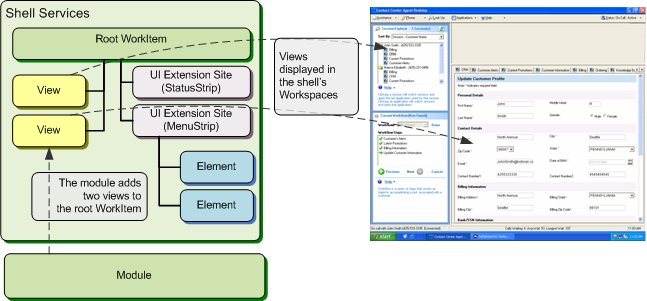
# Preparation

Before proceeding with this lab, you must install and configure the prerequisite software. For more information, see the topic Start Here.

Open the solution for the previous lab (either the one that you created or the end solution for that lab.)

# Background

Modules can also contain visual representations of data, such as a control, a Windows Form, or a wizard page. These visual representations are referred to as views or **SmartParts**. As an example, a module that manages information from a Customer Relationship Management (CRM) system has a view that displays a customer's contact information. The module loads this view during module initialization. When the application executes the business logic that requires the customer contact information to be displayed, the application loads the customer contact information view into the shell. The following figure illustrates a module that adds two views to the root **WorkItem** and later displays the views in the shell.

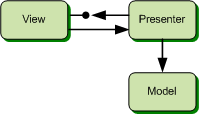
* 1. 
  2. Figure 1
  3. Views defined in module displayed in shell workspaces

### Model-View-Presenter (MVP) Pattern

The forms in a smart client application frequently contain various controls, handle user events, and contain logic to alter the controls in response to these events. Writing this code in the form class makes the class complex and difficult to test. In addition, it is difficult to share code between forms that require the same behavior.

The solution is to separate the responsibilities for the visual display and the event handling behavior into different classes. A view class manages the controls on the form, and it forwards events to a presenter class. The presenter contains the logic to respond to the events, and in turn, manipulate the state of the view. The presenter class uses the model (frequently, this is application state that is represented by business entities) to determine how to respond to the events.

This solution separates the responsibilities and also allows you to test the behavior without using the user interface. Figure 2 illustrates the logical view of the pattern.

* 1. 
  2. Figure 2
  3. MVP pattern logical view

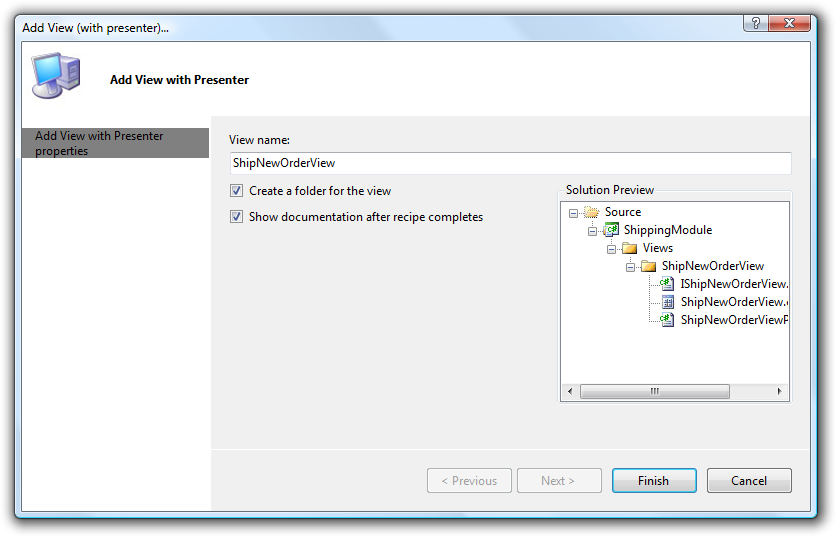
The model holds the business data, such as business entities. The model is unaware of the presenter that changes its state. The view holds a reference to its presenter and delegates to the presenter the handling of all user events (no business logic is implemented in the view). The presenter does not reference the class that implements the view; instead, it references an interface for the view (**IView**). With this, you can easily substitute one view implementation with another for the same presenter. One application of this feature is to test your presenter with a view implementation that does not have a user interface.

Exercise 1: Implementing the User Interface

In this exercise, you will use the guidance package to create a view for the ShippingModule module. This view will be the interface presented to the user to ship a new order. The implementation will follow the MVP pattern. Later you will add code to the ShippingModule to display this view when the user clicks the **Ship Order** button in the shell.

# Task 1. Add a new view (with presenter)

In this task you will use recipe named **Add View (with presenter)** to create classes that implement the Model-View-Presenter pattern.

* 1. In Solution Explorer, right-click the **Views** folder of the **ShippingModule** project, point to **Smart Client Software Factory**, and then click **Add** **View (with presenter)**.
     1. **Note:** The **Views** folder was created by the **Add Business Module** recipe. You are not required to create your views in this folder. It is a recommendation for organizational purposes.
  2. The **Add View (with presenter)** recipe launches a wizard. Enter **ShipNewOrderView** in the **View Name** field and select the **Create a folder for the view** option.
     1. 
     2. Figure 3
     3. Add View (with presenter) wizard page
     4. If **Create a folder for the view** is selected, the recipe will create a folder and place the new items in this folder; otherwise, the new items are placed in the selected folder in solution explorer.
     5. If you want to see a summary of the recipe actions and suggested next steps after the recipe completes, select the **Show documentation after recipe completes** check box.
  3. Click **Finish**. The recipe generates the folder, classes and interface for the implementation. Figure 4 illustrates the new items as they appear in Solution Explorer.
     1. 
     2. Figure 4
     3. ShipNewOrderView in Solution Explorer

The recipe generates the following items:

* + **A view interface class**. This is an empty interface definition for the view. You will modify this interface to define the public interface to the view. (The presenter class uses this interface to communicate with the view.)
  + **A view implementation user control**. This class derives from **UserControl** and has the **[SmartPart]** attribute. This is required to support the inversion of control functionality, which will automatically associate service dependencies, controllers, and so on. The user control also implements the view interface and contains a reference to its presenter. You will modify this class to call the presenter for user-interface actions that affect other views or business logic. The user control implementation is split into three files using a partial class:
    - ShipNewOrderView.cs. This file contains user code. Use this file to implement the view logic. If you run the recipe again for the view, the code in this file will not be removed.
    - ShipNewOrderView.Designer.cs. This file contains the designer code for the user control. Typically you do not edit the content of this file manually; instead, you use the Visual Studio designer.
    - ShipNewOrderView.GeneratedCode.cs. This file contains generated code to have a new instance of the presenter injected when the user control is created. The content of this file is regenerated when you run the recipe again for the view; therefore you should not edit it.
  + **A presenter class for the view**. This class extends the **Presenter** base class defined in **Infrastructure.Interface** project and contains the business logic for the view. You will modify this class to update the view for your business logic. The implementation of the class is split into two files using a partial class:
    - ShipNewOrderViewPresenter.cs. Use this file to implement the presenter logic. If you run the recipe again for the view, the code in this file will not be lost.
    - ShipNewOrderViewPresenter.GeneratedCode.cs. This file contains generated code which gets regenerated every time you run the recipe for this view; therefore you should not edit it.
  1. **Note:** The Presenter base class contains generic code for presenters. It includes a reference to a view and a reference to the **WorkItem** that contains the view.

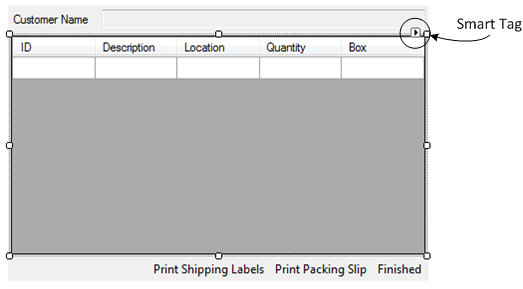
# Task 2. Create business entities

In this task you will create two business entities that will represent an order and its detail items. You will use these entities throughout the application.

* 1. Create a folder named **BusinessEntities** in the ShippingModule project.
  2. Add a new class file named Order.cs to the folder you have just created. To do this, right-click the BusinessEntities folder, point to **Add** and select **Class**. In the **Add New Item** dialog, set the item name to **Order** and click **Add**.
  3. Replace the empty class definition with the following code:
     1. C#
     2. public enum OrderState
     3. {
     4. New,
     5. Submitted
     6. }
     7. public class Order
     8. {
     9. private List<OrderLineItem> \_lineItems = new List<OrderLineItem>();
     10. private string \_customerName;
     11. private int \_orderId;
     12. private OrderState \_state = OrderState.New;
     13. public List<OrderLineItem> LineItems
     14. {
     15. get { return \_lineItems; }
     16. set { \_lineItems = value; }
     17. }
     18. public OrderState State
     19. {
     20. get { return \_state; }
     21. set { \_state = value; }
     22. }
     23. public string CustomerName
     24. {
     25. get { return \_customerName; }
     26. set { \_customerName = value; }
     27. }
     28. public int OrderId
     29. {
     30. get { return \_orderId; }
     31. set { \_orderId = value; }
     32. }
     33. }
  4. Add a new class file named OrderLineItem.cs to the BusinessEntities folder.
  5. Use the following code to define the class:
     1. C#
     2. public class OrderLineItem
     3. {
     4. private int \_productId;
     5. private string \_description;
     6. private short \_quantity;
     7. private string \_location;
     8. private int \_boxNumber;
     9. public int BoxNumber
     10. {
     11. get { return \_boxNumber; }
     12. set { \_boxNumber = value; }
     13. }
     14. public int ProductId
     15. {
     16. get { return \_productId; }
     17. set { \_productId = value; }
     18. }
     19. public string Description
     20. {
     21. get { return \_description; }
     22. set { \_description = value; }
     23. }
     24. public short Quantity
     25. {
     26. get { return \_quantity; }
     27. set { \_quantity = value; }
     28. }
     29. public string Location
     30. {
     31. get { return \_location; }
     32. set { \_location = value; }
     33. }
     34. }
  6. Build the ShippingModule project.
  7. **Note:** Storing your business entities in a folder named BusinessEntities is just a recommendation for organizational purposes.

# Task 3. Add controls to the design surface

In this task you will drag controls onto the view surface. These controls will display customer name, order details, and buttons to print shipping labels, print a packing slip, and to finish the order. Figure 5 illustrates the layout of the **ShipNewOrderView**.

* 1. 
  2. Figure 5
  3. ShipNewOrderView layout
  4. Double click in **ShipNewOrderView.cs** to open the designer window.
  5. Go to the Properties window and set the following property value for the **UserControl**:
     + Size = 422, 278
  6. Open the Toolbox and drag a **Label** control onto the upper-left corner of the design surface. Open the **Properties** window for the label and set the following property values:
     + Text = “Customer Name:”
     + Location = 3, 7
  7. Drag a **TextBox** from the **General** tab of the Toolbox and set the following properties:
     + Anchor = Top, Left, Right
     + Location = 94, 4
     + Width = 325
     + ReadOnly = True
  8. In the Properties window for the text box, click the plus sign next to **(DataBindings)**, and then click **Add Project Data Source** in the **Text** drop-down list box. This will launch the Data Source Configuration Wizard.
     1. **Note:** In this text box, we will show the customer’s name. To accomplish this task, we create a **BindingSource** associated with the **Order** business entity.
  9. Select **Object** as the data source, and then click **Next**.
  10. In the object browser, navigate through the **AdventureWorks.ShippingModule.BusinessEntities** node in the **ShippingModule** assembly, and then select **Order**.
  11. Click **Finish**. An object browser will be shown for the **Text** binding property. Select the **CustomerName** field of the **Order** class.
  12. Drag a **DataGridView** from the **Data** tab of the Toolbox onto the design surface. You will use the DataGridView to display the order details.
  13. Point to the smart tag, and then click **Add Project Data Source** in the **Choose Data Source** drop-down list box to launch the Data Source Configuration Wizard.
      1. 
      2. Figure 6
      3. Grid View’s Smart Tag
  14. Select **Object** as the data source, and then click **Next**.
  15. In the object browser, navigate through the **AdventureWorks.ShippingModule.BusinessEntities** node in the **ShippingModule** assembly and select **OrderLineItem**.
  16. Click **Finish**. A BindingSource will automatically be created.
  17. Open the smart tag again and click **Edit Columns** to open the **Edit Columns** dialog box.
  18. Sort the columns and set the **HeaderText**, as shown in figure 6 and then click **OK**.
  19. Set the following properties for the **DataGridView**:
      + Anchor = Top, Bottom, Left, Right
      + Location = 3, 30
      + Size = 416, 220
      + AutoSizeColumnsMode = Fill
      + RowHeadersVisible = False
  20. Add a **ToolStrip** at the bottom of the UserControl where **Print Shipping Label**, **Print Packing Slip**, and **Finished** buttons will be placed. To do this, in the Toolbox, double-click the **ToolStrip** button in the **Menus & Toolbars** section.
  21. Set the following properties (you can use the smart tag for the first three):
      + Dock = Bottom
      + GripStyle = Hidden
      + RenderMode = System
      + RightToLeft = Yes
  22. Click on the ellipsis of the **Items** property to open the Items Collection Editor.
  23. Add three buttons and set this property for all of them:
      + DisplayStyle = Text
  24. Set the following properties for the corresponding button:
      + 1st button: Text = “Finished”; Name = FinishedButton
      + 2nd button: Text = “Print Packing Slip”; Name = PrintPackingSlipButton
      + 3rd button: Text = “Print Shipping Labels”; Name = PrintShippingLabelButton
  25. Click **OK** and save the changes.

# Task 3. Explore the view code

In this task you will explore the view code to understand how the view collaborates with the presenter.

* 1. In Solution Explorer, right-click the file ShipNewOrderView.GeneratedCode.cs and select View Code. This file is located in the Views\ShipNewOrderView folder of the ShippingModule project.
     1. The view contains a public property for the presenter with the **[CreateNew]** attribute applied to it so that ObjectBuilder injects a new instance of the **ShipNewOrderViewPresenter** class when the view is created:
     2. C#
     3. [CreateNew]
     4. public ShipNewOrderViewPresenter Presenter
     5. {
     6. set
     7. {
     8. \_presenter = value;
     9. \_presenter.View = this;
     10. }
     11. }
     12. In this property, the presenter injected by ObjectBuilder is stored in the local variable \_presenter, and the view is assigned to the presenter. The \_presenter variable is declared in the ShipNewOrderView.Designer.cs file.
     13. The view also overrides the **OnLoad** method of the UserControl class to notify the presenter that the view is ready, in the code of the ShipNewOrderView.cs file:
     14. C#
     15. protected override void OnLoad(EventArgs e)
     16. {
     17. \_presenter.OnViewReady();
     18. base.OnLoad(e);
     19. }
     20. Calling the **OnViewReady** method of the presenter in the **OnLoad** method allows the presenter to execute business logic when the view is loaded.

# Task 4. Implement the view logic

You now have a user interface that displays controls but does not respond to user input. In this section, you will implement event handlers for the controls’ events (by using method calls to the presenter) and public methods in the view that the presenter will call. In the following exercise, you will be guided to implement the presenter logic.

It is the view's responsibility to display a particular order to the user, so you will add a method named **BindToOrder** to the **IShipNewOrderView** interface. This method will be in charge of showing the order details in the view.

* 1. In the Views\ShipNewOrderView folder of the ShippingModule project, open the IShipNewOrderView.cs file.
  2. Paste the following code inside the interface definition:
     1. C#
     2. void BindToOrder(AdventureWorks.ShippingModule.BusinessEntities.Order order);
     3. This method will be called from the presenter whenever an order has to be displayed to the user.
  3. Open the code view for the file ShipNewOrderView.cs. Right-click **IShipNewOrderView** (in the first line of the class definition), point to **Implement Interface**, and then click **Implement Interface** to let Visual Studio automatically generate the interface’s method skeletons.
  4. Add the following **using** statement at the top of the file.
     1. C#
     2. using AdventureWorks.ShippingModule.BusinessEntities;
  5. In the **IShipNewOrderView Members** code region, replace the **BindToOrder** method with the following code.
     1. C#
     2. public void BindToOrder(AdventureWorks.ShippingModule.BusinessEntities.Order order)
     3. {
     4. orderBindingSource.Clear();
     5. orderBindingSource.Add(order);
     6. orderLineItemBindingSource.Clear();
     7. foreach (OrderLineItem item in order.LineItems)
     8. {
     9. orderLineItemBindingSource.Add(item);
     10. }
     11. }
     12. This method clears the **orderBindingSource** and adds the new order. The binding source is used as the data source for the **Customer Name** TextBox’s **Text** property. Then it clears the binding source which contains line items and adds all the items of the new order. This means the new line items will be shown in the **DataGridView**.

In the next steps you will implement a method that will display a message box. The presenter will invoke this method to display a message to the user.

* 1. In the Views\ShipNewOrderView folder of the ShippingModule project, open the IShipNewOrderView.cs file and add the following method to the interface body:
     1. C#
     2. void ShowMessage(string message);
     3. This method will be called from the presenter whenever a message has to be displayed to the user.
  2. Open the code view for the file ShipNewOrderView.cs and paste the following code in the **IShipNewOrderView Members** code region:
     1. C#
     2. public void ShowMessage(string message)
     3. {
     4. MessageBox.Show(message);
     5. }

In the following steps, you will write code in the view to respond to user clicks on the bottom **ToolStripButtons**. In the view, you will call the presenter to perform each of the required actions.

* 1. Edit the view in the Designer. Double-click the button **PrintShippingLabelButton** to create an empty **Click** event handler for it.
  2. Add the following highlighted code to the event handler to call the presenter:
     1. C#
     2. private void PrintShippingLabelButton\_Click(object sender, EventArgs e)
     3. {
     4. **\_presenter.PrintShippingLabels();**
     5. }
     6. When the user clicks the Print Shipping Label button, the view calls the presenter to perform the labels printing.
  3. The **PrintShippingLabels** method is not yet implemented in the presenter. If you build the solution, you will get compile time errors. Right-click the method call or press SHIFT+ALT+F10 and select **Generate Method Stub** in the contextual menu to have the method definition created automatically by Visual Studio 2005, in the presenter class.
  4. Go back to the Designer and double-click the **PrintPackingSlipButton** button.
  5. Use the following code to implement the **PrintPackingSlipButton\_Click** method:
     1. C#
     2. private void PrintPackingSlipButton\_Click(object sender, EventArgs e)
     3. {
     4. **\_presenter.PrintPackingSlip();**
     5. }
  6. Generate a method stub for the **PrintPackingSlip** method as you did with PrintShippingLabels method.
  7. In the Designer, double-click the button **FinishedButton**.
  8. Replace the **FinishedButton\_Click** with the following code:
     1. C#
     2. private void FinishedButton\_Click(object sender, EventArgs e)
     3. {
     4. **\_presenter.Submit();**
     5. }
     6. When the user clicks the **Finished** button, the view tells the presenter to submit the order.
  9. Generate method stub for **Submit**.

# Task 5. Show the view in the shell

In this task you will write the code to display the view when the user clicks the **Ship Order** button in the launch bar.

* 1. Open the ModuleController.cs file in the ShippingModule project.
  2. Replace the body of the **OnShowOrder** method with the following code:
     1. C#
     2. [CommandHandler(CommandNames.ShipOrder)]
     3. public void OnShowOrder(object sender, EventArgs e)
     4. {
     5. **ShowViewInWorkspace<ShipNewOrderView>(WorkspaceNames.PrimaryWorkspace);**
     6. }
     7. When the user clicks the **Ship Order** button in the shell, the ShipOrder command is executed by the Composite UI Application Block. Thus, the OnShowOrder method is invoked. The ShowViewInWorkspace method is implemented in the WorkItemController. This method creates a new **ShipNewOrderView** instance and adds it to the **SmartParts** collection of the **WorkItem**. Then, it shows the view in the workspace supplied as a parameter (in this case, the primary workspace in the Shell).

# Task 6. Compile and run the solution

In this task you will verify that you correctly implemented the ShipNewOrderView view and that it appears in the Shell when you click the **Ship Order** button.

* 1. If you try to build the solution, you will get compilation errors. The errors are caused because the mock view (**MockShipNewOrderView**) that was created by the Add View recipe does not implement the new methods that you added to the IShipNewOrderView interface. To remove these errors, you need to implement the methods. To do this, open the file Views\ShipNewOrderViewPresenterFixture.cs in the ShippingModule.Tests project, right-click **IShipNewOrderView** (in the first line of the MockShipNewOrderView class definition), point to **Implement Interface**, and then click **Implement Interface** to let Visual Studio automatically generate the interface’s method skeletons.
  2. Build and run the application.
     1. You will see the same MDI interface that you saw in the previous lab. Click the **Ship Order** button to open the ShipNewOrderView view. You will see empty controls that show no information because the presenter logic is not yet implemented. If you click any of the three buttons in the ToolStrip of the view, an exception will be thrown because the operations in the presenter are not implemented yet. In the next exercise, you will implement the presenter logic.
     2. You can click the **Ship Order** buttons multiple times to create new instances of the ShipNewOrderView view, showing different orders with no title. In exercise 3, you will implement the **ISmartPartInfoProvider** interface to provide the title of the view.
     3. 
     4. Figure 7
     5. ShipNewOrder view displayed in shell
  3. Close the application.

Exercise 2: Implementing the Presenter Logic

In this exercise, you will implement the logic for the presenter. A presenter contains code to handle user events and to update the state of the view.

* 1. **Note**: Typically you do not write extensive business logic in the presenter; instead, you delegate it to other components named services. This approach lets you consume the business logic encapsulated in services from other presenters and facilitates testing because you can test the presentation logic and the business logic separately. In this exercise though, you will perform simple business logic tasks in the presenter to keep the exercise focused on views and presenters. You will replace this business logic with a service in a subsequent lab.

# Task 1. Show the next order when the view loads

When the **ShipNewOrderView** loads, it must display the next order ready to be shipped. This business logic will be implemented in the **OnViewReady** method of the presenter. The **OnViewReady** method is called by the view when it is loaded. You will modify this method to bind the next available order to the view.

* 1. In the Views\ShipNewOrderView folder, open the ShipNewOrderViewPresenter.cs file and add the following **using** statement at the top of the file.
     1. C#
     2. using AdventureWorks.ShippingModule.BusinessEntities;
  2. Paste the following code to declare a local variable to hold a reference to an Order instance and to define the constructor.
     1. C#
     2. private Order \_order;
     3. public ShipNewOrderViewPresenter()
     4. {
     5. \_order = CreateOrder();
     6. }
     7. The \_order variable will be used to store current order information. In the constructor, you create an order by calling the **CreateOrder** method (which you will implement in the next step) and store it in the \_order variable for further use.
  3. Use the following code to define the **CreateOrder** method. This method basically creates a new order with a correlative identifier, fills it with a few order line items, and returns it. In a subsequent lab you will encapsulate this logic in a service.
     1. C#
     2. private static int nextOrderNumber = 1;
     3. private static Order CreateOrder()
     4. {
     5. Order order = new Order();
     6. order.OrderId = nextOrderNumber++;
     7. order.CustomerName = "Hance, Jim";
     8. for (int i = 1; i <= 3; i++)
     9. {
     10. OrderLineItem item = new OrderLineItem();
     11. item.Location = "One Microsoft Way, Redmond, WA, US";
     12. item.ProductId = i;
     13. item.Quantity = (short)i;
     14. item.BoxNumber = 0;
     15. item.Description = String.Format("Item {0} description", i);
     16. order.LineItems.Add(item);
     17. }
     18. return order;
     19. }
  4. The **OnViewReady** method is called by the view when it loads. Add the following highlighted line to the method body to bind the view to the order when it is loaded:
     1. C#
     2. public override void OnViewReady()
     3. {
     4. **View.BindToOrder(\_order);**
     5. base.OnViewReady();
     6. }

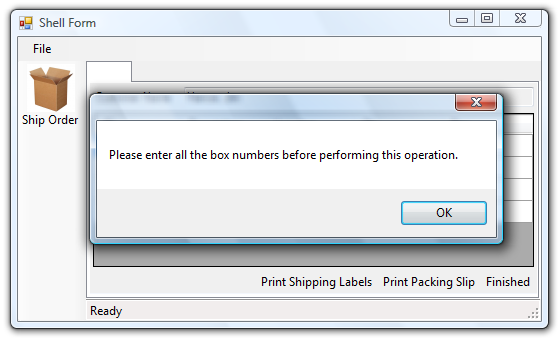
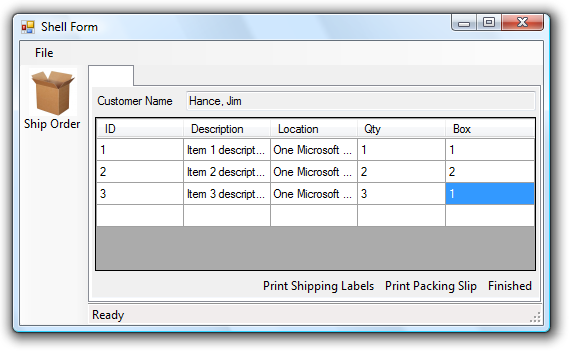
# Task 2. Submit the order

In this task you will write code in the **Submit** method to submit an order. The view calls the **Submit** method when the **Finished** button is clicked.

* 1. Open the ShipNewOrderViewPresenter.cs file.
  2. Replace the **Submit** method with the following code:
     1. C#
     2. public void Submit()
     3. {
     4. if (!Validate())
     5. {
     6. View.ShowMessage("Please enter all the box numbers before performing this operation.");
     7. return;
     8. }
     9. \_order.State = OrderState.Submitted;
     10. OnCloseView();
     11. }
     12. This method validates the order by calling the **Validate** method (which you will implement in the next step) and if validation does not succeed, it tells the view to display an error message by invoking the **ShowMessage** method. If the validation succeeds, it changes the state of the order to **Submitted** and closes the view by calling the OnCloseView method. The OnCloseView method is defined in the **Presenter** base class.
  3. Paste the following code to define the Validate method.
     1. C#
     2. // Note: This validation logic is not a real-world example but is sufficient
     3. // to force the user to enter box numbers in the view.
     4. private bool Validate()
     5. {
     6. foreach (OrderLineItem lineItem in \_order.LineItems)
     7. {
     8. if (lineItem.BoxNumber < 1)
     9. return false;
     10. }
     11. return true;
     12. }

# Task 3. Compile and run the solution

In this task you will verify that you correctly implemented the presenter logic.

* 1. Build and run the application.
     1. The MDI interface will appear. Click the **Ship Order** button to open the ShipNewOrderView displaying an order.
  2. Press the **Finished** button. A message box will appear asking you to enter all the box numbers before submitting the order. Click **OK**.
     1. 
     2. Figure 8
     3. Error message displayed in the view
  3. Enter some numeric values for the Box column, and then click the **Finished** button to submit the order. The ShipNewOrderView view will close.
     1. **Note:** You have to press the Tab key after entering numeric values in the Box column. Otherwise, the validation will fail.
     2. 
     3. Figure 9
     4. Order information displayed in view
  4. Close the application.

Exercise 3: Showing Tab Names Implementing ISmartPartInfoProvider

In this exercise, you will implement the **ISmartPartInfoProvider** in a view to provide information about it.

# Background: SmartParts and SmartPartInfo

SmartParts are the visual components of an application. You can create SmartParts by customizing a standard user control. To enable the consumer of a workspace to provide rich hinting and extra information about a SmartPart, workspaces use an implementation of **ISmartPartInfo** that the SmartPart author can add to their controls.

## The ISmartPartInfoProvider interface

The ISmartPartInfoProvider interface contains only one method named **GetSmartPartInfo**:

* 1. C#
  2. public interface ISmartPartInfoProvider
  3. {
  4. **ISmartPartInfo GetSmartPartInfo(Type smartPartInfoType);**
  5. }

The GetSmartPartInfo method returns the SmartPartInfo of the view, which provides rich hinting and extra information about it.

You can make your view implement the ISmartPartInfoProvider interface to indicate that it can provide a SmartPartInfo.

# Task 1. Implement the ISmartPartInfoProvider interface in the view

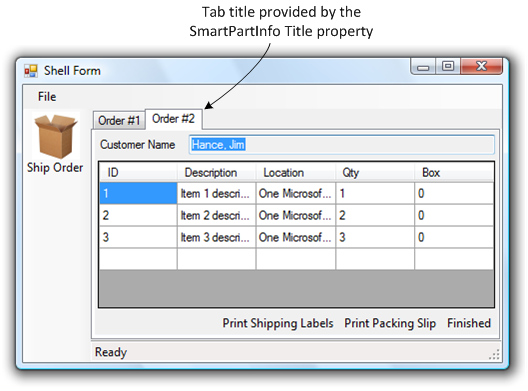
In this task you will modify the ShipNewOrderView view to implement the **ISmartPartInfoProvider** interface. The **SmartPartInfo** provided by the view will be used by the TabWorkspace in the shell to display the title of the view in the tab.

* 1. Open the ShipNewOrderView.cs file in code mode.
  2. Replace the class signature with the following code to indicate that the ShipNewOrderView view implements the **ISmartPartInfoProvider** interface:
     1. C#
     2. public partial class ShipNewOrderView : UserControl, IShipNewOrderView, **ISmartPartInfoProvider**
  3. Right-click ISmartPartInfoProvider, point to Implement Interface, and then click Implement Interface. Visual Studio will generate the GetSmartPartInfo method automatically.
  4. Replace the generated GetSmartPartInfo method with the following code:
     1. C#
     2. public ISmartPartInfo GetSmartPartInfo(Type smartPartInfoType)
     3. {
     4. return \_presenter.GetSmartPartInfo(smartPartInfoType);
     5. }
     6. Whenever a workspace asks for a SmartPartInfo to the view, the view will forward the request to the presenter, which will return a SmartPartInfo with current order information.
  5. Go to the ShipNewOrderViewPresenter.cs file.
  6. Add the following **using** statement before the class definition. You will use it to refer to the ISmartPartInfo interface.
     1. C#
     2. using Microsoft.Practices.CompositeUI.SmartParts;
  7. Define the GetSmartPartInfo method with the following code.
     1. C#
     2. public ISmartPartInfo GetSmartPartInfo(Type smartPartInfoType)
     3. {
     4. ISmartPartInfo result =
     5. (ISmartPartInfo)Activator.CreateInstance(smartPartInfoType);
     6. result.Title = String.Format("Order #{0}", \_order.OrderId);
     7. result.Description = "Select this to select and ship the next order from the queue.";
     8. return result;
     9. }
     10. In this method, you create a new SmartPartInfo and set the title and description using current order information.



# Task 2. Compile and run the solution

In this task you will verify that the view provides a valid ISmartPartInfo instance.

* 1. Build and run the application.
     1. The MDI interface will appear. Click the **Ship Order** button to open the **ShipNewOrderView** view displaying an order. The order number appears in the tab title.
     2. 
     3. Figure 10
     4. Order number in tab title
  2. Close the application.

To check the finished solution open the solution file **CS\Developer\03-CreatingView\AdventureWorksCycles.sln**.