Lab 2: Creating a Business Module

# Purpose

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

In this lab, you will use the Smart Client Development guidance package to add a business module to you smart client application solution. You will add code to the module that updates the user interface of the shell to expose features available in the module.

After completing this lab, you will be able to:

* + Understand the components of a business module
	+ Add a new business module to a smart client application using the guidance package.
	+ Update the shell with user interface elements from the module

# 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: Business and Foundational Modules

Developers can use the Smart Client Software Factory to develop smart client applications composed of independent, but collaborating, modules. Productivity increases because each developer (or team) is able to concentrate on a specific task. For example, developers of service components work with only the business logic in the component; they do not have to be concerned with background issues, such as threading and security or the appearance and behavior of the application.

Applications built with the Smart Client Software Factory have a design that delineates the difference between writing shell logic (requiring expertise in Windows Forms, user controls, and appearance and behavior), infrastructure components (built only once per application or re-used from other applications), and business logic (the user interfaces, logic, entities, and service agents of the specific application). This means that the application architecture supports developers and teams that have different concerns during the creation of an application. Figure 1 illustrates an example of a smart client application architecture that supports concerns of different teams and individuals with different expertise within a team.

* 1. 
	2. Figure 1
	3. Separation of concerns with a Composite UI Application Block application

Modules are distinct deployment units of a Composite UI Application Block application. You use modules to encapsulate a set of concerns of your application and deploy them to different users or applications. The Smart Client Software Factory makes a distinction between two different types of modules: business modules and foundational modules.

## Business Modules

A business module represents the conventional Composite UI Application Block module. It has at least one **WorkItem** (specifically, a **ControlledWorkItem**) and contains business logic elements. Typically, it includes some combination of services, views, presenters, and business entities.

The Smart Client Software Factory reference implementation contains examples of business modules. In the Branch Client reference implementation, the **BasicAccounts.Module**, **BranchSystems.Module** and **CreditCardAccounts.Module** projects are business modules. These modules encapsulate the logic for the different types of back-end financial systems. For example, the **BranchSystems.Module** module contains business logic to manage a queue of customers. This logic includes determining the user interface elements to display based upon the current user’s role.

When you use the Smart Client Software Factory to create a business module, the **Add Business Module** recipe creates both a **Module** class and a **ModuleController** class. As with all Composite UI Application Block modules, the **Module** class derives from the **ModuleInit** class. In the **Load** method of the **Module** class, the recipe generates code to create a new **ControlledWorkItem** and add it to the application **WorkItem** hierarchy, as shown in the following code.

* 1. C#
	2. public override void Load()
	3. {
	4. base.Load();
	5. ControlledWorkItem<ModuleController> workItem =
	6. \_rootWorkItem.WorkItems.AddNew<ControlledWorkItem<ModuleController>>();
	7. workItem.Controller.Run();
	8. }

## Foundational Modules

A foundational module is a module that either provides services to the shell and other modules, provides a layout, or both. It does not implement a use case or contain a **WorkItem**.

In the Branch Client reference implementation, the **BranchSystems.Layout** and **Infrastructure.Module** projects are foundational modules. The **BranchSystems.Layout** module does not contain a **WorkItem** (there is no **ModuleController** class). Instead, it contains a view to provide the layout of the shell form’s user interface. The **Infrastructure.Module** module provides services (for example, caching) to other modules in the application.

In a subsequent lab, you will add a foundational module to your application.

## Module Loading and Initialization

The Composite UI Application Block provides a service to load modules when the application starts. It uses a catalog file to determine the modules to load. The default name for this file is ProfileCatalog.xml. The following XML shows the contents of the ProfileCatalog.xml file for the Disconnected Service Agent (with CAB) QuickStart.

* 1. XML
	2. <SolutionProfile xmlns="http://schemas.microsoft.com/pag/cab-profile/2.0">
	3. <Section Name="Services">
	4. <Modules>
	5. **<ModuleInfo AssemblyFile="Infrastructure.Module.dll" />**
	6. </Modules>
	7. </Section>
	8. <Section Name="Apps">
	9. <Dependencies>
	10. <Dependency Name="Services" />
	11. </Dependencies>
	12. <Modules>
	13. **<ModuleInfo AssemblyFile="QuickStart.RestaurantModule.dll" />**
	14. </Modules>
	15. </Section>
	16. </SolutionProfile>

When the application block loads a module, it uses reflection to determine if the module includes a class that implements the **IModule** interface, and executes the **Load** method of that class. You implement the IModule interface (typically by deriving from **ModuleInit**) to initialize and run the module's **WorkItems**.

Exercise 1: Creating a New Business Module

In this exercise, you will create a business module for your application. This module contains the business logic to ship an order. The module exposes the feature to the user through an icon that appears in the shell’s launch bar (the **ToolStrip** control).

# Task 1. Use the Guidance Package to add a new Business Module

In this task you will add a business module to your application. The guidance package includes the Visual Studio template named **Add** **Business Module (C#)**. This template unfolds a new class library project for the module.

* 1. In Solution Explorer, right-click the **Source** solution folder, point to **Smart Client Software Factory**, and then click **Add Business Module (C#)**. The **Add New Project** dialog box appears with the **Add Business Module (C#)** template selected.
	2. Enter **ShippingModule** as the **Name** and set the **Location** to the **Source** folder of the solution.
		1. Click **OK**. The guidance package displays the **Add Business Module** wizard.
		2. Figure 2
		3. Add Business Module Wizard
	3. Deselect the option **Create an interface library for this module**. If you select this option, the recipe will create an additional project to contain the elements that provide the public interface to the assembly. For more information about separating the interface from the implementation of modules, see the Module Interface Separation pattern in the Smart Client Software Factory documentation.
	4. Select the option **Create a unit test project for this module** to have the recipe create a test project for the module with test classes for your module components. You will implement unit tests in a subsequent lab.
	5. Optionally, select the option **Show documentation after recipe completes** if you want to see a summary of the recipe actions and suggested next steps after the recipe completes.
	6. Click **Finish**. The guidance package will generate a new class library project named **ShippingModule**.
		1. 
		2. Figure 3
		3. Shipping Module in Solution Explorer

The root folder of the project contains two classes, **Module** and **ModuleController**. The **Module** class derives from the Composite UI Application Block class **ModuleInit**. The Composite UI Application Block calls the **Load** method of this class on startup. The **Load** method contains code to create and run a new **WorkItem**. This WorkItem is the module’s main WorkItem.

The **ModuleController** class contains methods that you can modify to customize the behavior of the module on startup. For example, you can add services or display user-interface items. (In the next task you will modify some of these methods.)

The project also contains the following folders:

* + **Constants**. This folder contains four classes named **CommandNames**, **EventTopicNames**, **UIExtensionSiteNames**, and **WorkspaceNames**. You can modify these classes to define module-specific identifiers for your commands, event topics, **UIExtensionSites**, and **WorkSpaces**.
	+ **Services**. You use this folder to store the implementation of business services.
	+ **Views**. You use this folder to store views.
1.

The recipe also adds the following XML entry for the module to the profile catalog of the application. This means that the Composite UI Application Block will load the module at application initialization time.

* 1. XML
	2. <Modules>
	3. <ModuleInfo AssemblyFile="ShippingModule.dll" />
	4. </Modules>

# Task 2. Add the launch point for the module

In this task you will add code in the module that updates the user interface of the shell. The module adds a button to the launch bar to expose functionality in the shipping module to the user.

## Background: UI Extension Sites

Most Windows-based applications use common user interface elements throughout an application. The Composite UI Application supports the development of shared **UIElements** by shell developers that module developers can access. For example, menus, toolbars, status bars, progress bars, sidebars, action panes, the notification tray, and so on. Shell developers are responsible for the creation of these components, and module developers can create instances of the component and access their properties. To allow modules to extend a shell user interface element, shell developers register the shell element as a **UIExtensionSite**. (A **UIExtensionSite** is identified by a string.) After a **UIExtensionSite** is registered, other parts of the application can access and extend that **UIExtensionSite**.

After a module is loaded, it can add elements to the shell's UI Extension Sites. For example, a module can add menu items to the shell's **MenuStrip** to expose the module's features. If the entry for that module is removed from the catalog, the module will not be loaded when the application starts. The application will continue to function, but the menu items for that module will not appear. illustrates a module that adds menu items to the shell **MenuStrip**.

* 1. 
	2. Figure 4
	3. Menu items added by module

## Procedure

The first step is to add a string constant for the title and an icon to the module.

* 1. In Solution Explorer, right-click the **ShippingModule** project and then click **Properties**.
	2. Click the **Resources** tab.
	3. Click **This project does not contain a default resources file. Click here to create one**. A resources file will be created in the Properties folder of the module.
	4. Add a new string constant named **ModuleTitle** and set its value to **Ship Order**.
		1. 
		2. Figure 5
		3. ModuleTitle string resource
	5. Click the **Add Resource** drop-down list box and then click **Add Existing File**.
	6. Browse to the Assets\icons folder, click **ShippingIcon.bmp**, and then click **Open**.
		1. 
		2. Figure 6
		3. Shipping module icon in the resource editor
	7. Click **Save**.
		1. Next you will add a **ToolStripButton** with module’s icon to the shell’s **ToolStrip**. You do this by adding a new **ToolStripButton** to the **LaunchBar** **Extension Site**.
	8. Open the file ModuleController.cs in the **ShippingModule** project and add the following **using** statements before the class signature:
		1. C#
		2. using System.Drawing;
		3. using AdventureWorks.ShippingModule.Constants;
	9. Replace the **ExtendToolStrip** method with the following code:
		1. C#
		2. private void ExtendToolStrip()
		3. {
		4. AddLaunchBarButton(CommandNames.ShipOrder,Properties.Resources.ShippingIcon, Properties.Resources.ModuleTitle);
		5. }
		6. After the module is loaded, the **Run** method of the **ModuleController** class is invoked. The Smart Client Software Factory generates code in the **Run** method that calls the **ExtendToolStrip** method. You will use this method to add the launch point for the module to the shell.
	10. Create a method that creates a new **ToolStripButton** with the specified icon and text and adds it to the shell’s launch bar. To do this, paste the following code below the **ExtendToolStrip** method:
		1. C#
		2. private void AddLaunchBarButton(string commandName, Image shippingIcon, string shipOrderText)
		3. {
		4. ToolStripButton element = new ToolStripButton();
		5. element.TextImageRelation = TextImageRelation.ImageAboveText;
		6. element.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;
		7. element.ImageScaling = ToolStripItemImageScaling.None;
		8. element.Image = shippingIcon;
		9. element.Text = shipOrderText;
		10. element.ToolTipText = shipOrderText;
		11. WorkItem.UIExtensionSites[Constants.UIExtensionSiteNames.LaunchBar].Add(element);
		12. }
		13. This code creates a new **ToolStripButton** and sets some visual properties (**TextImageRelation**, **DisplayStyle**, and **ImageScaling**.) It also sets the module’s icon as the button’s image and defines the **Text** and **ToolTipText** property values.
		14. Finally, it adds the button to the **LaunchBar** extension site of the **WorkItem**. (In Lab 1 you added code to the **AfterShellCreated** method of the **ShellApplication** class to register the **LaunchBar** extension site.)

# Task 3. Associate a command and command handler with the launch bar icon

Applications will often contain more than one control that invokes the same method. For example, a menu item and toolbar item may both run the **OpenFile** method. The Composite UI Application Block uses the concept of commands to enable you to write one event handler that is associated with more than one **UIElement**, and associate one **UIElement** with multiple command handlers.

In this task, you will add code to display a view in the shell when the user clicks on the ShippingModule button.

* 1. Add the following constant declaration to the CommandNames.cs file. This file is located in the **Constants** folder of the **ShippingModule** project:
		1. C#
		2. public const string ShipOrder = "ShipOrder";
	2. Add the code to associate the **Click** event of the **ToolStripButton** with the **ShipOrder** command. To do this, add the following code to the end of the **AddLaunchBarButton** method in the ModuleController class:
		1. C#
		2. WorkItem.Commands[commandName].AddInvoker(element, "Click");
		3. When this code executes, the Composite UI Application Block registers the command with the “Click” event. This means that when a user clicks on the **ToolStripButton**, the Composite UI Application Block calls methods that are registered as command handlers for the **ShipOrder** command (you will register a command handler in the next step.)
		4. The full **AddLaunchBarButton** method should appear as follows.
		5. C#
		6. private void AddLaunchBarButton(string commandName, Image shippingIcon, string shipOrderText)
		7. {
		8. ToolStripButton element = new ToolStripButton();
		9. element.TextImageRelation = TextImageRelation.ImageAboveText;
		10. element.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;
		11. element.ImageScaling = ToolStripItemImageScaling.None;
		12. element.Image = shippingIcon;
		13. element.Text = shipOrderText;
		14. element.ToolTipText = shipOrderText;
		15. WorkItem.UIExtensionSites[Constants.UIExtensionSiteNames.LaunchBar].Add(element);
		16. **WorkItem.Commands[commandName].AddInvoker(element, "Click");**
		17. }
	3. To add a command handler for the ShipOrder command, paste the following code into the **ModuleController** class:
		1. C#
		2. [CommandHandler(CommandNames.ShipOrder)]
		3. public void OnShowOrder(object sender, EventArgs e)
		4. {
		5. MessageBox.Show("Ship Order invoked");
		6. }
		7. The **CommandHandler** attribute indicates to the Composite UI Application Block that this method is to be invoked whenever the **ShipOrder** command is executed. The code in this method shows a message indicating that the user has invoked the command. In the next lab, you will update the code to display a view.

# Task 4. Compile and run the solution

In this task you will verify that you correctly added the module to the solution and that the module can be launched from the Shell’s launch bar.

* 1. Build and run the application.
		1. You will see a form with the **ShippingModule** button in the ToolStrip located on the left. Click the button to simulate the module being shown. In the next lab, you will implement and display a view.
		2. 
		3. Figure 7
		4. Module button displayed in shell form
	2. Close the application.

# Task 5. Reflection

You can use the Smart Client Software Factory at different times in the application development life cycle. In this exercise, you learned how to add a module to your application. This is an activity that is typically repeated during the course of developing your application. In the subsequent exercises, you will use additional recipes to further extend your application.

The code generated by the Add Business Module recipe requires that the solution has code constructs produced by the **Smart Client Application** template. For example, the generated module project includes a reference to the **Infrastructure.Interface** project in the same solution. In the next exercise you will modify the module’s code to use elements in the **Infrastructure.Interface** project.

Exercise 2: Writing reusable code for module controllers

The launch bar in the shell is a UI extension site where modules can add **ToolStripButtons**. In the previous exercise, you created a method in the **ShippingModule** module with the logic to add a **ToolStripButton** to the launch bar. If you create a second module and want to add a **ToolStripButton** to the shell for that module, you must duplicate the **AddLaunchBarButton** method in that module. In this exercise, you will update the WorkItemController base class to expose reusable code for adding module launch points to the shell. You will then update the **ModuleController** class of the **ShippingModule** to consume its functionality.

# Task 1. Update the WorkItemController base class

* 1. Open the file WorkItemController.cs located in the root of the Infrastructure.Interface project.
	2. Add the following **using** statements:
		1. C#
		2. using System.Drawing;
		3. using AdventureWorks.Infrastructure.Interface.Constants;
	3. Paste the following code in the class body:
		1. C#
		2. protected void RegisterLaunchPoint(string text, Image icon, string commandName)
		3. {
		4. ToolStripButton button = new ToolStripButton();
		5. button.TextImageRelation = TextImageRelation.ImageAboveText;
		6. button.DisplayStyle = ToolStripItemDisplayStyle.ImageAndText;
		7. button.ImageScaling = ToolStripItemImageScaling.None;
		8. button.Image = icon;
		9. button.Text = text;
		10. button.ToolTipText = text;
		11.
		12. WorkItem.Commands[commandName].AddInvoker(button, "Click");
		13. WorkItem.UIExtensionSites[UIExtensionSiteNames.LaunchBar].Add(button);
		14. }
		15. This method is nearly the same as **AddLaunchBarButton** that you created in the previous exercise.
	4. The **RegisterLaunchPoint** method requires the containing project to have a reference to the System.Drawing assembly. To add it, in Solution Explorer right-click the **Infrastructure.Interface** project and select **Add Reference**. In the **.NET** tab**,** select the **System.Drawing** assembly and click **OK**.

# Task 2. Refactor the Shipping Module

In this task, you will perform changes to the shipping module to enable you toreuse the piece of code you added in the previous task and to follow the suggested design for this solution.

* 1. Open th**e** ModuleController.cs file in the **ShippingModule** project.
	2. Remove the definition of the **AddLaunchBarButton** method.
	3. Replace the **ExtendToolStrip** method with the following code:
		1. C#
		2. private void ExtendToolStrip()
		3. {
		4. RegisterLaunchPoint(Properties.Resources.ModuleTitle, Properties.Resources.ShippingIcon, CommandNames.ShipOrder);
		5. }
		6. This method now uses the base class method **RegisterLaunchPoint** to add the **ToolStripButton** to the launch bar of the shell.
	4. Remove the following **using** statement you added in previous exercise, you need it no longer:
		1. C#
		2. using System.Drawing;
1.

# Task 3. Compile and run the solution

In this task you will verify that you perform the refactoring correctly.

* 1. Build and run the application.
		1. You will see a form with the **ShippingModule** button in the ToolStrip located on the left. Click the button to simulate the module being shown. In the next lab, you will implement a view.
		2. 
		3. Figure 8
		4. Message displayed from module command handler
	2. Close the application.

To check the finished solution open the solution file **CS\Developer\02-CreatingBusinessModule \AdventureWorksCycles.sln**.