Lab 5: Creating a Foundational 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 foundational module to you smart client application solution. You will also create a new global service that will simulate a real printing service.

After completing this lab, you will be able to:

* + Understand the components of a foundational module
	+ Add a new foundational module to a smart client application using the guidance package.
	+ Create a global service

# 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: 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. An example of a foundational module is a module that contains infrastructure services, like caching or logging services. Since a foundational module does not contain a WorkItem, all the services created within a foundational module are registered as global services with the root WorkItem.

Exercise 1: Creating a New Foundational Module

In this exercise you create a foundational module which will provide a printing service for all the modules in the application. The ShippingModule module will consume it to print packing slips and shipping labels for sales orders.

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

In this task you will add a foundation module to your application. The guidance package includes the Visual Studio template named **Add** **Foundational 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 Foundational Module (C#)**. The **Add New Project** dialog box appears with the **Add Foundational Module (C#)** template selected.
	2. Enter **PrintingModule** as the **Name** and set the **Location** to the **Source** folder of the solution.
		1. Click **OK**. The guidance package displays the **Add Foundational Module** wizard.
		2. 
		3. Figure
		4. Add Foundational 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. Deselect the option **Create a unit test project for this module.** If you select this option, the recipe will create a test project for the module with test classes for your module components. In this lab you will not write unit tests for the module.
	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 **PrintingModule**.
		1. 
		2. Figure
		3. PrintingModule module in Solution Explorer

The root folder of the project contains a class, **Module**. 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 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 global services.
1.

The recipe also adds the following XML entry for the module to the profile catalog of the application (defined in the file Shell\ProfileCatalog.xml). 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. **<ModuleInfo AssemblyFile="PrintingModule.dll" />**
	5. </Modules>

The Composite UI Application Block loads modules in the order they are specified in the profile catalog file, unless you explicitly define inter-module dependencies. Thus, the ShippingModule module will be loaded before the PrintingModule. This might lead to problems if the ShippingModule instantiates a service during module initialization that has a dependency in a component that belongs to the PrintingService module. To avoid this problem, you can specify module dependencies by arranging modules into sections and establishing dependencies between the sections. When you create a smart client solution, two sections are defined by default:

* + **Services**. Include in this section modules that expose global services to other modules. Typically you will include foundational modules in this section.
	+ **Apps**. Include in this section modules that might have dependencies on services published by other modules. Typically you will include business modules in this section.

In the next step you will move the PrintingModule module to the **Services** section so that the Composite UI Application Block loads it before the ShippingModule module.

* 1. Move the <ModuleInfo> element for the PrintingModule module into the **Services** section, as shown in the following code.
		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. **<ModuleInfo AssemblyFile="PrintingModule.dll" />**
		7. </Modules>
		8. </Section>
		9. <Section Name="Apps">
		10. <Dependencies>
		11. <Dependency Name="Services" />
		12. </Dependencies>
		13. <Modules>
		14. <ModuleInfo AssemblyFile="ShippingModule.dll" />
		15. </Modules>
		16. </Section>
		17. </SolutionProfile>
	2. **Note:** You can also define dependencies for a particular module in code using the **[ModuleDependency]** attribute. For more information, see Creating a Module in the Smart Client Software Factory help.

# Task 2. Move Business Entities to Infrastructure.Interface project

In the next task (Task 3) you will add a printing service to the PrintingModule module. This service will simulate the printing of a sales order, thus its methods will receive an **Order** entity as a parameter. Since the **Order** class is defined in the ShippingModule project, you would need to add a reference to it in the PrintingModule project. To avoid having a reference to the ShippingModule project in the PrintingModule project, in this task you will move the Order and OrderLineItem entities to the Infrastructure.Interface project.

* 1. In Solution Explorer, select the Order.cs and OrderLineItem.cs files in the **BusinessEntities** folder of the **ShippingModule** project, cut them, and paste them in the **BusinessEntities** folder in the **Infrastructure.Interface** project.
		1. Even though you moved the source code files to the Infrastructure.Interface project, the classes´ namespace is still AdventureWorks.ShippingModule.BusinessEntities. In the following steps you will rename the namespace to AdventureWorks.Infrastructure.Interface.BusinessEntities.
	2. Go to the **Edit** menu, point to **Find and Replace** and then select **Quick Replace**.
	3. Set **AdventureWorks.ShippingModule.BusinessEntities** in the **Find what** field.
	4. Set **AdventureWorks.Infrastructure.Interface.BusinessEntities** in the **Replace with** field.
	5. In the **Look in** drop down list, set **Entire Solution**.
	6. Click **Replace All**. 13 occurrences should be replaced.

# Task 3. Implement the printing service

In this task you will create a simple printing service that will be consumed by the **ShipNewOrderView** view.

* 1. In Solution Explorer, right-click the **Services** folder in the **PrintingModule** project, point to **Add** and then click **New Item**.
	2. From the **Templates** window, select **Interface** and set the name to **IPrintingService.**
	3. Click **Add**.
	4. Add the following **using** statement to the file. You will use it to refer to the Order and OrderLineItem classes.
		1. C#
		2. using AdventureWorks.Infrastructure.Interface.BusinessEntities;
	5. Change the interface signature to make it public:
		1. C#
		2. public interface IPrintingService
	6. Add a method named **PrintPackingSlip** to the interface body. This method will be used to print the packing slips for an order.
		1. C#
		2. void PrintPackingSlip(Order order);
	7. Add a method named **PrintShippingLabels** to the interface body. This method will be used to print the shipping labes for an order.
		1. C#
		2. void PrintShippingLabels(Order order);
	8. In Solution Explorer, right-click the **Services** folder in the **PrintingModule** project, point to **Add** and then click **Class**.
	9. Set the class name to **LocalPrintingService** and then click **Add.**
	10. Add the following **using** statements at the top of the file:
		1. C#
		2. using System.Windows.Forms;
		3. using AdventureWorks.Infrastructure.Interface.BusinessEntities;
	11. Change the class signature to make it public and to implement the **IPrintingService** interface:
		1. C#
		2. public class LocalPrintingService : IPrintingService
	12. Implement the **PrintPackingSlip** method. In this method, display a message box as demonstrated in the following code.
		1. C#
		2. public void PrintPackingSlip(Order order)
		3. {
		4. MessageBox.Show(String.Format("Packing slip for Order # {0} sent to printer...", order.OrderId.ToString()));
		5. }
	13. Implement the **PrintShippingLabels** method. In this method, display a message box as shown in the following code.
		1. C#
		2. public void PrintShippingLabels(Order order)
		3. {
		4. MessageBox.Show(String.Format("Shipping labels for Order # {0} sent to printer...", order.OrderId.ToString()));
		5. }

# Task 4. Register the Printing Service as a Global Service

In this task, you will register the printing service as a global service. Instead of manually registering the service with a WorkItem as you did in Lab 4 – Creating a Service, you will use the **[Service]** attribute to have the Composite UI Application Block automatically register the service with the root WorkItem when the application starts.

* 1. Open the LocalPrintingService.cs file, located in the Services folder of the PrintingModule project.
	2. Add the following **using** statement at the top of the file:
		1. C#
		2. using Microsoft.Practices.CompositeUI;
	3. Decorate the class with the [**Service**] attribute and specify the **IPrintingService** type as the registration key, as shown in the following code.
		1. C#
		2. **[Service(typeof(IPrintingService))]**
		3. public class LocalPrintingService : IPrintingService
1. 1. **Note:** For more information about the [**Service**] attribute, see Adding Services in the Smart Client Software Factory help.

# Task 5. Consume the Printing Service from the Presenter

In this task you will update the presenter to consume the printing service.

* 1. Add a reference tothe **PrintingModule** project in the **ShippingModule** project. To do this, right-click the ShippingModule project node in Solution Explorer and select **Add Reference**. From the **Projects** tab, select **PrintingModule** and click **OK**.
		1. **Note:** By adding a reference to the PrintingModule project in the ShippingModule project, you are tightly coupling the modules. You can decouple the modules by using the Module Interface Separation pattern. For more information, see Module Interface Separation in the Smart Client Software Factory help.
	2. Open the file Views\ShipNewOrderView\ShipNewOrderViewPresenter.cs, located in the ShippingModule project.
	3. Add the following **using** statement at the top of the file:
		1. C#
		2. using AdventureWorks.PrintingModule.Services;
	4. Add the following member variable declaration to hold a reference to a **IPrintingService** instance:
		1. C#
		2. private IPrintingService \_printingService;
	5. Add an additional parameter to the constructor to receive an instance of the printing service using dependency injection, and store the instance reference in the \_printingService member variable for further use. To do this, replace the constructor with the following code:
		1. C#
		2. public ShipNewOrderViewPresenter
		3. (
		4. [ServiceDependency] ISalesOrderService orderService,
		5. [ServiceDependency] IEntityTranslatorService translator,
		6. **[ServiceDependency] IPrintingService printingService**
		7. )
		8. {
		9. \_translator = translator;
		10. \_orderService = orderService;
		11. \_order = translator.Translate<Order>(\_orderService.GetNextOrder());
		12. **\_printingService = printingService;**
		13. }

When the presenter is created, ObjectBuilder locates an implementation of the **IPrintingService** interface, and passes it to the constructor.

* 1. Modify **PrintPackingSlip** and **PrintShippingLabels** methods to consume the **LocalPrintingService**. To do this, replace the **PrintPackingSlip** method with the following code:
		1. C#
		2. internal void PrintPackingSlip()
		3. {
		4. if (!Validate())
		5. {
		6. View.ShowMessage("Please enter all the box numbers before performing this operation.");
		7. return;
		8. }
		9. \_printingService.PrintPackingSlip(\_order);
		10. }
	2. Replace the **PrintShippingLabels** method with the following code:
		1. C#
		2. internal void PrintShippingLabels()
		3. {
		4. if (!Validate())
		5. {
		6. View.ShowMessage("Please enter all the box numbers before performing this operation.");
		7. return;
		8. }
		9. \_printingService.PrintShippingLabels(\_order);
		10. }

# Task 6. Compile and run the solution

In this task you will verify that you correctly added the printing service to the root WorkItem and that the presenter is consuming it.

* 1. If you created unit tests for the ShipNewOrderViewPresenter class in Lab 4 – Creating a Service, you will get compilation errors because the OnViewReadyBindsViewToOrder test is outdated. To avoid these errors, please exclude the file Views\ShipNewOrderViewPresenterFixture.cs from the ShippingModule.Tests project. Updating the test is out of the scope of this lab.
	2. Build and run the application.
	3. Click the **Ship Order** button to open the **ShipNewOrderView** displaying an order.
	4. Change the box number for the displayed order.
	5. Click the **Print Shipping Labels** and **Print Packing Slip** buttons to invoke the printing service. A message box will appear.
		1. 
		2. Figure
		3. Print Packing Slip message box displayed by the printing service.
	6. Close the application.

To check the finished solution, open the solution file **CS\Developer\05-CreatingFoundationalModule\AdventureWorksCycles.sln**.