

Validation Application Block Hands-On Lab for Enterprise Library

This walkthrough should act as your guide to learn about the Validation Application Block and practice how to leverage its capabilities in various application contexts.

There are 14 exercises (labs) in the package. The first 11 labs deal with a Windows® Forms data processing application that takes the information entered by the user to populate and process Customer and Address business objects. The Validation Application Block is used to validate the created business objects before processing them in gradually more sophisticated ways. Starting with Lab 7, the Windows Forms validation–integration feature is used to directly validate the input for the form's controls. Labs 8 through 11 deal with the extensibility of the application block. Lab 12 shows how to use the ASP.NET validation-integration feature of the application block to validate the ASP.NET control's values, using a Web forms version of the simple data entry application from the previous labs. For Lab 13, the ASP.NET application works as a front-end for a Windows Communication Foundation (WCF) service while the WCF validation–integration feature of the application block is used to declaratively validate the service parameters on the server side. Finally, lab 14 performs validation operations on objects and uses attributes to specify validation rules for a simple WPF application.

There are two ways you can complete this lab set: you can manually complete it from start-to-finish or you can use the starter solutions to complete only the labs you want to. By using the provided starter solutions, you can complete any of the labs in the order you prefer.

Estimated completion time for all the labs is three and a half hours.

This hands-on lab includes the following labs:

- [Lab 1: Adding Validation](#)
- [Lab 2: Consuming Validation Results](#)
- [Lab 3: Validating Object Graphs](#)
- [Lab 4: Using Custom Message Templates for Validation Failures](#)
- [Lab 5: Specifying Validation Rules Through Configuration](#)
- [Lab 6: Using Rule Sets](#)
- [Lab 7: Integrating with Windows Forms](#)
- [Lab 8: Implementing Self Validation](#)
- [Lab 9: Implementing a Custom Validator](#)
- [Lab 10: Using a Custom Validator Through Configuration](#)
- [Lab 11: Implementing a Custom Validator with Design-Time Support](#)

- [Lab 12: Integrating with ASP.NET](#)
- [Lab 13: Integrating with WCF](#)
- [Lab 14: WPF Integration](#)

All of the Hands-On Labs use a simplified approach to object generation through Unity and the Enterprise Library container. The recommended approach when developing applications is to generate instances of Enterprise Library objects using dependency injection to inject instances of the required objects into your application classes, and thereby obtain all of the advantages that this technique offers.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the Hands-On Labs examples use the simpler approach for resolving Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation installed with the Enterprise Library, or available on MSDN® at <http://msdn.microsoft.com/entlib/>.

Authors

This Hands-On Lab was produced by the following individuals:

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.), Nicolas Botto (Digit Factory)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

Lab 1: Adding Validation

Estimated time to complete this lab: **15 minutes**

Each Lab folder contains a folder named Before and a folder named After. Each Before folder contains source code files as they are before performing the edits and additions for that lab. Each After folder contains the source code edits and additions for that lab.

Purpose

In this lab, you will practice performing validation operations on objects and using attributes to specify validation rules for a simple data-entry application. For information about the use of attributes to specify validation rules, see "Key Scenarios" in the Validation Application Block documentation available at [http://msdn.microsoft.com/en-us/library/ff664741\(v=PandP.50\).aspx](http://msdn.microsoft.com/en-us/library/ff664741(v=PandP.50).aspx).

Preparation

Open the solution file from Lab01\Before\ValidationHOL.sln.

The solution consists of the following two projects:

- **ValidationHOL.BusinessLogic**. This project contains two simple domain classes, **Customer** and **Address**
- **ValidationHOL**. This project implements a simple data-entry Windows Forms application that creates instances of the domain classes using the values entered in the input fields and performs some processing.

If you build and run the ValidationHOL project without any modifications, leave the fields empty, and then click **Accept**; no validation will be performed. Because no validation attributes are set at this point, the application simply attempts to process the invalid data.

Procedures

This lab includes the following tasks:

- Task 1: Adding Validation Attributes to the Business Classes
- Task 2: Invoking Validation on Instances of a Business Class

Validation specified for a type is not automatically enforced; validation must be explicitly invoked by the client code.

Task 1: Adding Validation Attributes to the Business Classes

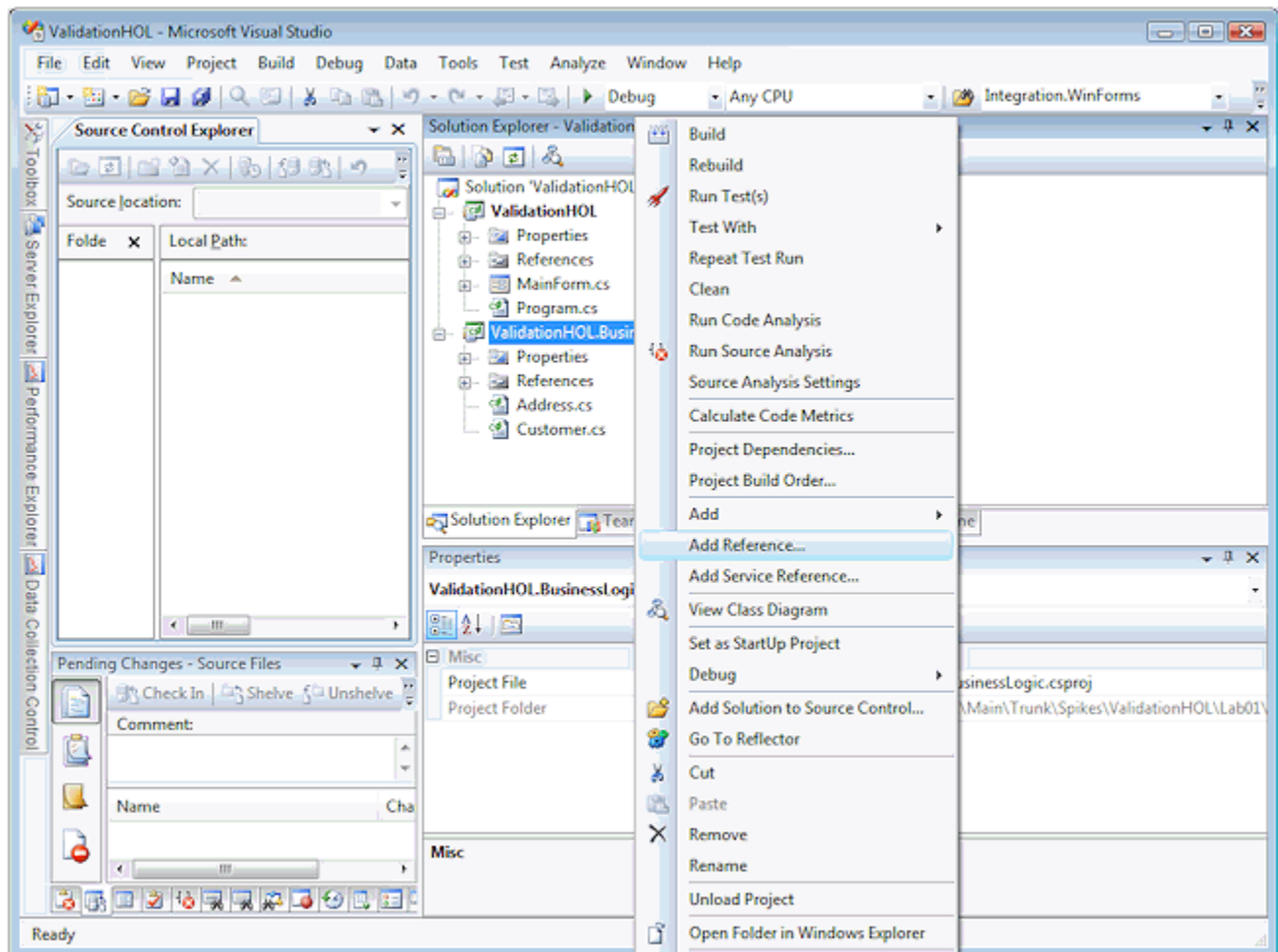
Attributes can be used to specify validation rules. These validation attributes can be used with classes and public fields, readable properties, and non-void methods with no arguments. Attributes for each built-in validation rule are available without modification to the application block. For information about the available validators, see the topic "Using the Validation Block Validators" in the online

documentation for Enterprise Library at [http://msdn.microsoft.com/en-us/library/ff664694\(v=PandP.50\).aspx](http://msdn.microsoft.com/en-us/library/ff664694(v=PandP.50).aspx).

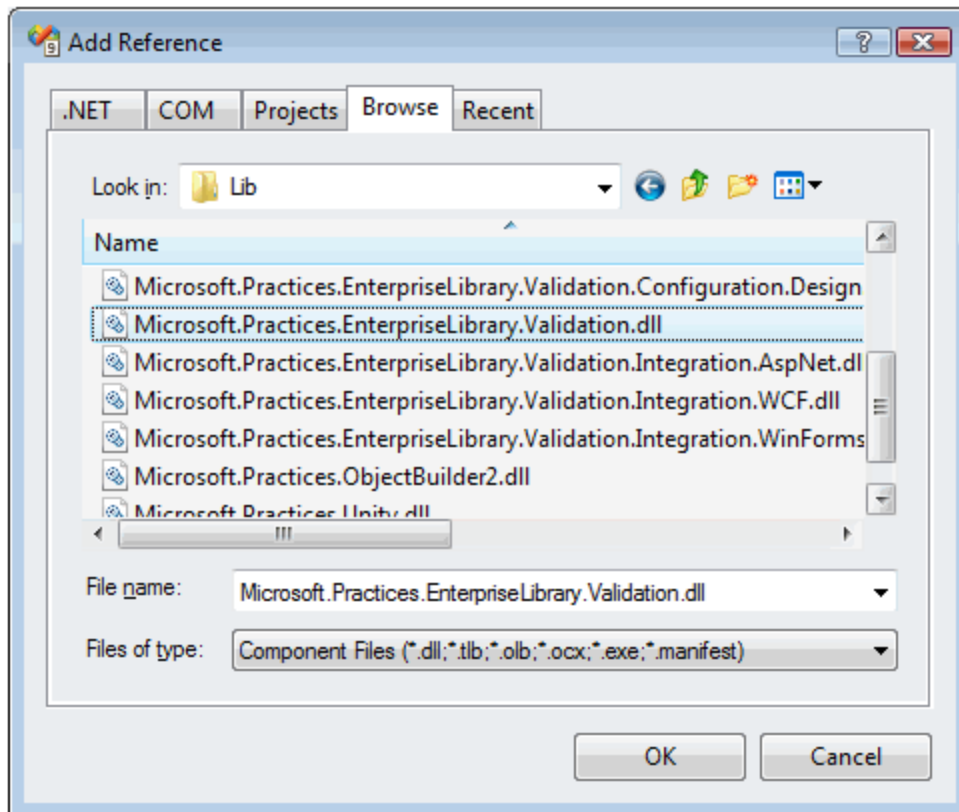
In this lab, you will validate the domain classes' properties for minimum and maximum lengths, except for the **SSN** and **ZipCode** properties from the **Customer** and **Address** classes, respectively, where regular expressions will be used to validate the values.

To add validation attributes

1. Add a reference to the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** assembly from the Lib folder of the examples. To do this, right-click the **ValidationHOL.BusinessLogic** project in Solution Explorer, and then click **Add Reference**.



Click the **Browse** tab, navigate to the Lib folder in the lab's main folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** file, and then click **OK**.



2. Repeat steps 1 and 2 to add a reference to the assembly **System.ComponentModel.DataAnnotations** to the ValidationHOL.BusinessLogic project.
3. Add **using** directives to the Customer.cs file in the ValidationHOL.BusinessLogic project to make the necessary types available without full name qualification.

```
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

4. Add validation attributes to the **Customer** class's properties.

```
public class Customer
{
    [StringLengthValidator(1, 25)]
    public string FirstName { get; set; }
    [StringLengthValidator(1, 25)]
    public string LastName { get; set; }
    [RegexValidator(@"^\d\d\d-\d\d-\d\d\d\d$")]
    public string SSN { get; set; }
    public Address Address { get; set; }
}
```

Typically, there are several parameters for each validation attribute. For information about each attribute, see the topic "Using the Validation Block Validators" in the online documentation for Enterprise Library at [http://msdn.microsoft.com/en-us/library/ff664694\(v=PandP.50\).aspx](http://msdn.microsoft.com/en-us/library/ff664694(v=PandP.50).aspx).

5. Add a **using** directive to the Address.cs file to make the necessary validator attribute types available without full name qualification.

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

6. Add validation attributes to the **Address** class's properties.

```
public class Address
{
    [StringLengthValidator(1, 50)]
    public string StreetAddress { get; set; }
    [ValidatorComposition(CompositionType.And)]
    [StringLengthValidator(1, 30)]
    [ContainsCharactersValidator("sea", ContainsCharacters.All)]
    public string City { get; set; }
    [StringLengthValidator(2, 2)]
    public string State { get; set; }
    [RegexValidator(@"^\d{5}$")]
    public string ZipCode { get; set; }
}
```

Note: There are multiple validator attributes and the additional **ValidatorComposition** attribute for the **City** property. When multiple validation attributes are specified for a class member, a *composite* validator is created, and the **ValidatorComposition** attribute determines whether it will be an *and* composite validator or an *or* composite validator. The former succeeds only when all the validators succeed; the latter succeeds if at least one of the validators succeeds. Also, the **ContainsCharactersValidator** causes city names that do not contain all of the specified letters 's', 'e', and 'a' to be deemed invalid.

Task 2: Invoking Validation on Instances of a Business Class

Validation must be explicitly invoked through code unless some kind of mechanism, such as the Policy Injection Application Block, is used to intercept method calls.

The result of invoking validation is an instance of the **ValidationResults** class, which can indicate whether the validation succeeded. If it did not succeed, it will contain a collection of **ValidationResult** instances for each failed validation rule.

To invoke validation

1. Add reference to each of the following assemblies located in the **Lib** folder to the ValidationHOL project:
 - Microsoft.Practices.EnterpriseLibrary.Validation.dll
 - Microsoft.Practices.EnterpriseLibrary.Common.dll
 - Microsoft.Practices.ServiceLocation.dll
2. Add **using** directives to the MainForm.cs file to make the necessary types available without full name qualification. Be sure to open the file in the code view.

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
```

```
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

3. Add a member variable for the validator object to the MainForm.cs class.

```
private Validator<Customer> customerValidator;
```

4. Update the **MainForm_Load** method in the **MainForm** class to create the validator object by adding the flowing highlighted code.

```
private void MainForm_Load(object sender, EventArgs e)
{
    ValidatorFactory valFactory =
        EnterpriseLibraryContainer.Current.GetInstance<ValidatorFactory>();
    customerValidator = valFactory.CreateValidator<Customer>();
}
```

5. Update the **acceptButton_Click** method in the **MainForm** class to invoke validation and show an error message if validation fails by adding the flowing highlighted code.

```
private void acceptButton_Click(object sender, EventArgs e)
{
    Customer customer = new Customer
    {
        FirstName = firstNameTextBox.Text,
        LastName = lastNameTextBox.Text,
        SSN = ssnTextBox.Text,
        Address = new Address
        {
            StreetAddress = streetAddressTextBox.Text,
            City = cityTextBox.Text,
            State = stateComboBox.Text,
            ZipCode = zipCodeTextBox.Text
        }
    };

    ValidationResult results = customerValidator.Validate(customer);

    if (!results.IsValid)
    {
        MessageBox.Show(
            this,
            "Customer is not valid",
            "Error",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
        return;
    }

    MessageBox.Show(
        this,
        "Processing customer '" + customer.FirstName + "'",
        "Working",
```

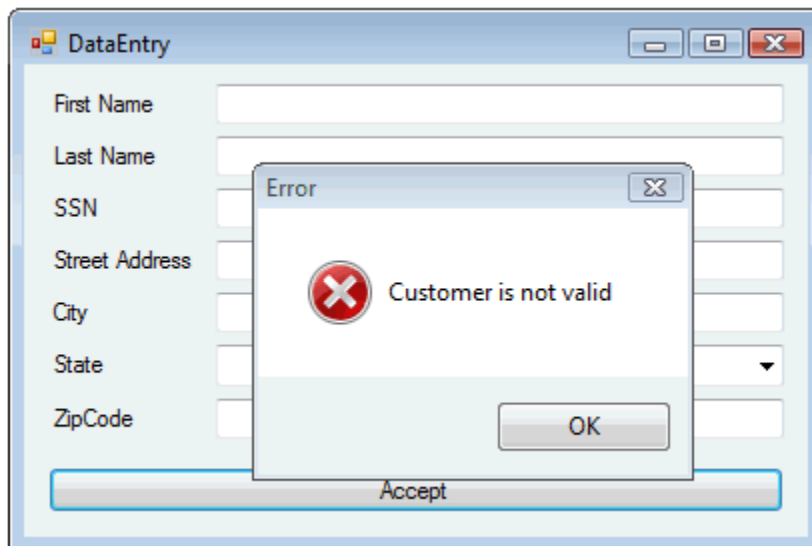
```
        MessageBoxButtons.OK,  
        MessageBoxIcon.Information);  
    }
```

Verification

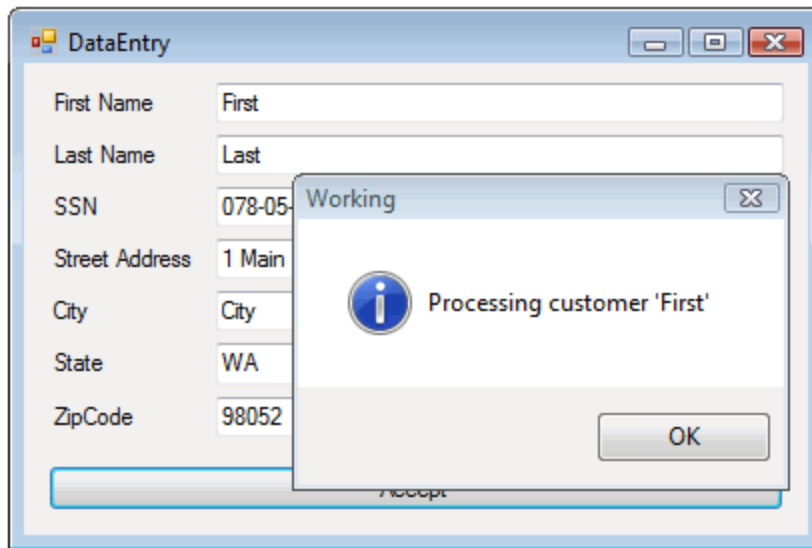
In this section, you will verify that the specified validation rules are actually used when performing the validation operation.

To validate that the validation attributes were properly added

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names that are longer than 25 characters), and then click **Accept**. The error message is illustrated here.



3. Enter valid values for the fields, and then click **Accept**. For the SSN, enter a value such as **111-11-1111**. The information message box is illustrated here.



You can also compare the lab solution folder to the contents of the Lab1\After folder. To compare the code used for the validation code that was added, see the ValidationHOL.BusinessLogic folder.

Lab 2: Consuming Validation Results

Estimated time to complete this lab: **5 minutes**

Purpose

In this lab, you will examine the results of a validation operation to retrieve more detailed information about validation failures.

Preparation

Continue working on the solution from Lab 1 or open the solution file from Lab02\Before\ValidationHOL.sln.

Procedures

This lab includes the following task:

- Task 1: Iterating Over the **ValidationResult** Elements in a Failed Validation Result to Obtain Details of the Failed Validation

Task 1: Iterating Over the ValidationResult Elements in a Failed Validation Result to Obtain Details of the Failed Validation

The information provided by the message in the error dialog box can be expanded by gathering and including information about the individual failures. Each **ValidationResult** element in a non-valid **ValidationResults** object represents the failure of an individual rule and includes, among other pieces of

information, a message that indicates the nature of the failure and a key that can be used to identify the failed rule. This information can be included in the displayed error message.

To gather the validation results

1. Add **using** directives to the **MainForm.cs** file for the types that will be used to build the message.

```
using System.Text;
using System.Globalization;
```

2. Update the **acceptButton_Click** method in the **MainForm** class to build an error text combining all the **ValidationResult** elements by adding the following highlighted code.

```
private void acceptButton_Click(object sender, EventArgs e)
{
    Customer customer = new Customer
    {
        FirstName = firstNameTextBox.Text,
        LastName = lastNameTextBox.Text,
        SSN = ssnTextBox.Text,
        Address = new Address
        {
            StreetAddress = streetAddressTextBox.Text,
            City = cityTextBox.Text,
            State = stateComboBox.Text,
            ZipCode = zipCodeTextBox.Text
        }
    };

    ValidationResult results = customerValidator.Validate(customer);

    if (!results.IsValid)
    {
        StringBuilder builder = new StringBuilder();
        builder.AppendLine("Customer is not valid:");
        foreach (ValidationResult result in results)
        {
            builder.AppendLine(
                string.Format(
                    CultureInfo.CurrentCulture,
                    "{0}: {1}",
                    result.Key,
                    result.Message));
        }
        MessageBox.Show(
            this,
            builder.ToString(),
            "Error",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
        return;
    }
}
```

```

    }

    MessageBox.Show(
        this,
        "Processing customer '" + customer.FirstName + "'",
        "Working",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}

```

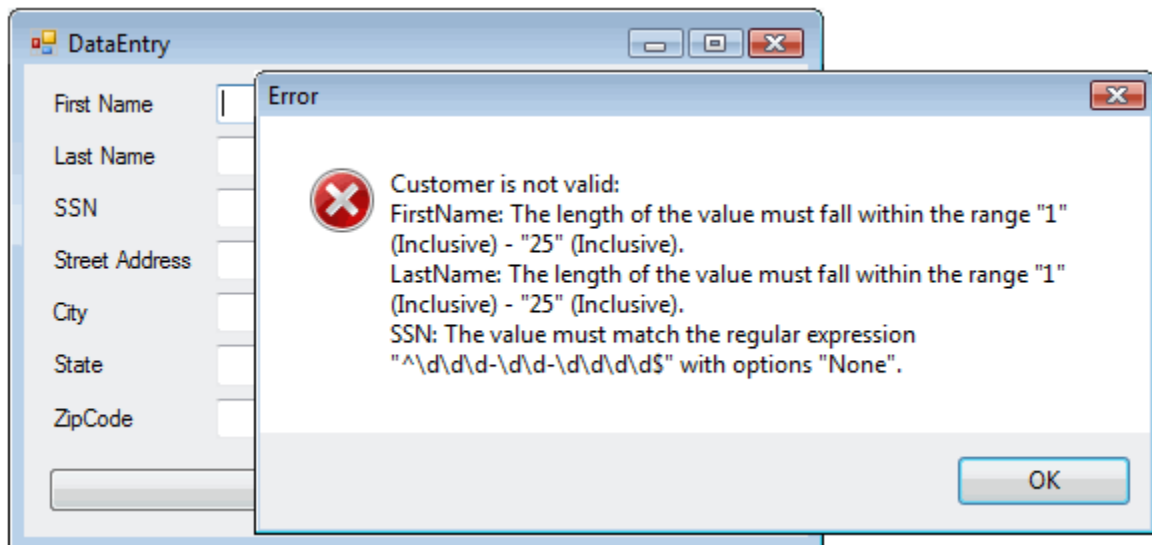
This code retrieves the **Key** and **Message** properties of each **ValidationResult** element used to create the error message.

Verification

In this section, you will verify that the error message now includes the messages of each validation rule.

To validate that the ValidationResult elements are properly retrieved and error messages displayed

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names that are longer than 25 characters), and then click **Accept**. The error message illustrated here should display.



Note: Only error messages for the customer are displayed. In fact, specifying valid values only for the first three fields (corresponding to the properties in the **Customer** class) is enough to satisfy the validation, even though the properties in the **Address** field also have validation attributes. Validation of nested objects does not occur by default, so it must be explicitly indicated. The next lab shows how to perform nested-object validation.

You can also compare the lab solution folder to the contents of the Lab2\After folder. To see the business logic code that was added for this lab, see the ValidationHOL.BusinessLogic folder.

Lab 3: Validating Object Graphs

Estimated time to complete this lab: **5 minutes**

Purpose

In this lab, you will practice using the **ObjectValidator** to validate object graphs. It must be noted that the **ObjectValidator** uses the validation rules specified for the static type of the object it validates, regardless of the actual type of the validated property.

Preparation

Continue working on the solution from Lab 2 or open the solution file from Lab03\Before\ValidationHOL.sln.

Procedures

This lab includes the following task:

- Task 1: Adding the **[ObjectValidator]** Attribute to the **Address** Property in the **Customer** Class

Task 1: Adding the **[ObjectValidator]** Attribute to the Address Property in the Customer Class

Add the attribute, as shown in the following code example.

```
public class Customer
{
    [StringLengthValidator(1, 25)]
    public string FirstName { get; set; }
    [StringLengthValidator(1, 25)]
    public string LastName { get; set; }
    [RegexValidator(@"^\d\d\d-\d\d-\d\d\d\d$")]
    public string SSN { get; set; }
    [ObjectValidator]
    public Address Address { get; set; }
}
```

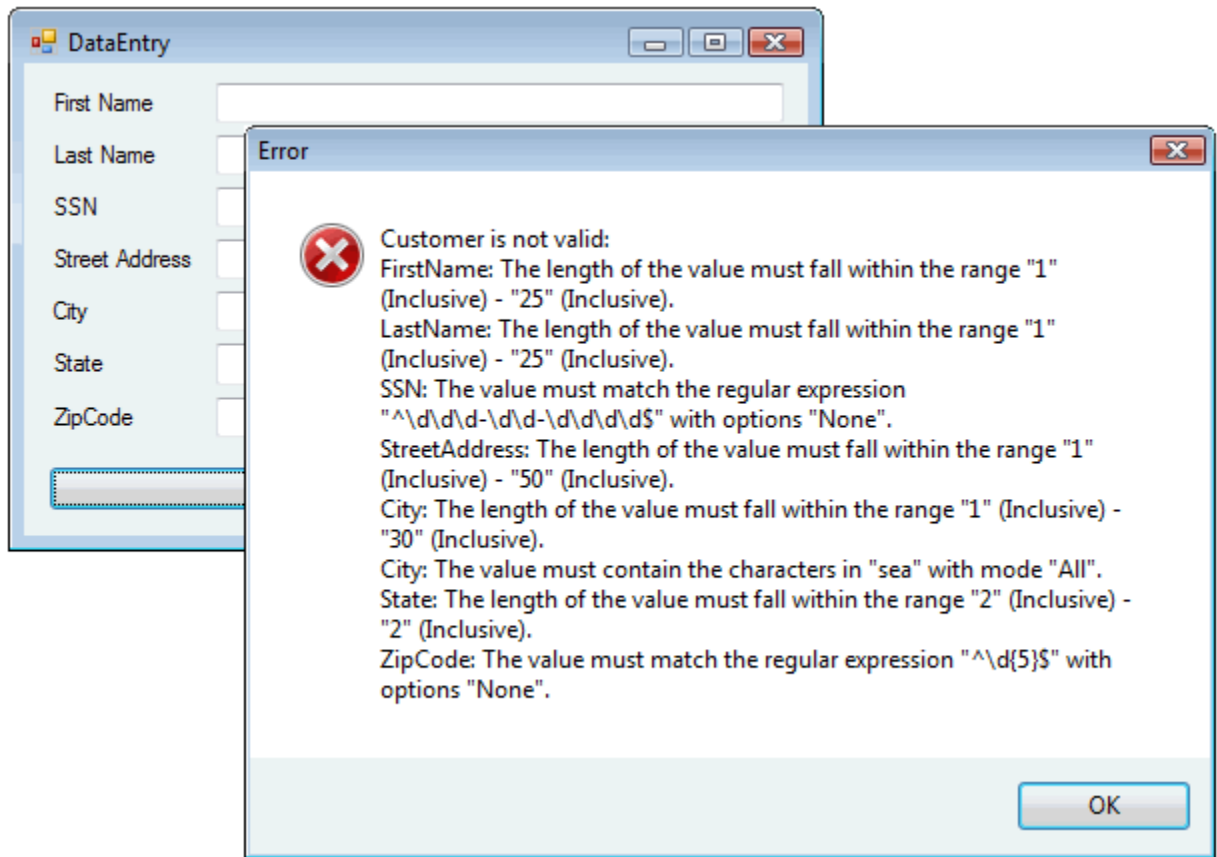
Verification

In this section, you will verify that the error message now includes the messages of each validation rule, including the rules associated with the **Address** class.

To validate that the address fields are validated according to the specified rules

1. Build and run the ValidationHOL project.

2. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names longer than 25 characters), and then click **Accept**. The error message illustrated here should display.



You can also compare the lab solution folder to the contents of the Lab3\\After folder. To see the business logic code that was added for this lab, see the ValidationHOL.BusinessLogic folder.

Lab 4: Using Custom Message Templates for Validation Failures

Estimated time to complete this lab: **10 minutes**

Purpose

In this lab, you practice overriding the generic, default validation message templates using custom localized resources. Literal message template overrides are also possible using a similar procedure.

Message template overrides, either literal or resource-based, are used to create validation failure messages. Typically, validators provide values that can be used in the templates, such as the minimum and maximum length in a **StringLengthValidator**. For information about the format items supplied by each validator, check the validator's reference.

Preparation

Continue working on the solution from Lab 3 or open the solution file from Lab04\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

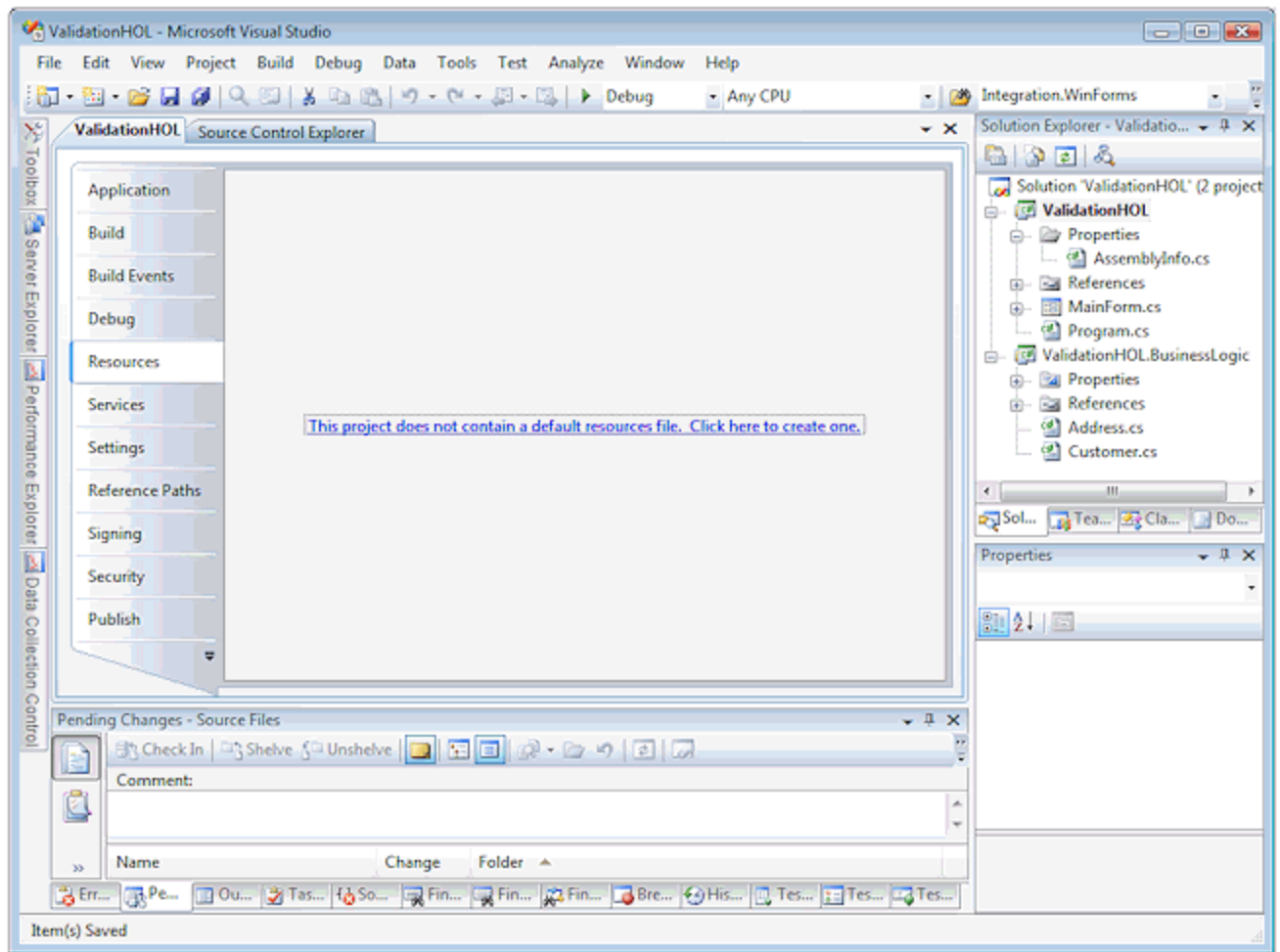
- Task 1: Creating a Resources File
 - Task 2: Adding Resource Strings for the Validation Messages
 - Task 3: Updating the Validation Attributes to Use the Resources
-

Task 1: Creating a Resources File

This lab uses the default resources file, but you could use any resources file.

To create the resources file

1. In Solution Explorer, right-click the **ValidationHOL.BusinessLogic** project, and then click **Properties**.
2. Click the **Resources** tab, and then click the link that offers to create a default resources file.



Task 2: Adding Resource Strings for the Validation Messages

Add the following strings in the resource file editor.

Name	Value
FirstNameMessage	The first name must be between {3} and {5} characters long.
LastNameMessage	The last name must be between {3} and {5} characters long.
SSNMessage	The social security number (SSN) must use the format ###-##-####.

Note: {3} and {5} in the strings in the preceding table represent the lower and upper bounds, respectively, for a **StringLengthValidator**.

Task 3: Updating the Validation Attributes to Use the Resources

Set the **MessageTemplateResourceType** and **MessageTemplateResourceName** named parameters for the validator attributes with customized message templates on the **Customer** class.

To update the attributes

1. Add a **using** directive to the Customer.cs file to make the default resources type available without full name qualification. Be sure to open the file in the code view.

```
using ValidationHOL.BusinessLogic.Properties;
```

2. Update the attributes for the **FirstName**, **LastName**, and **SSN** properties in the **Customer** class as show by the following highlighted code. The class should look like the following code fragment.

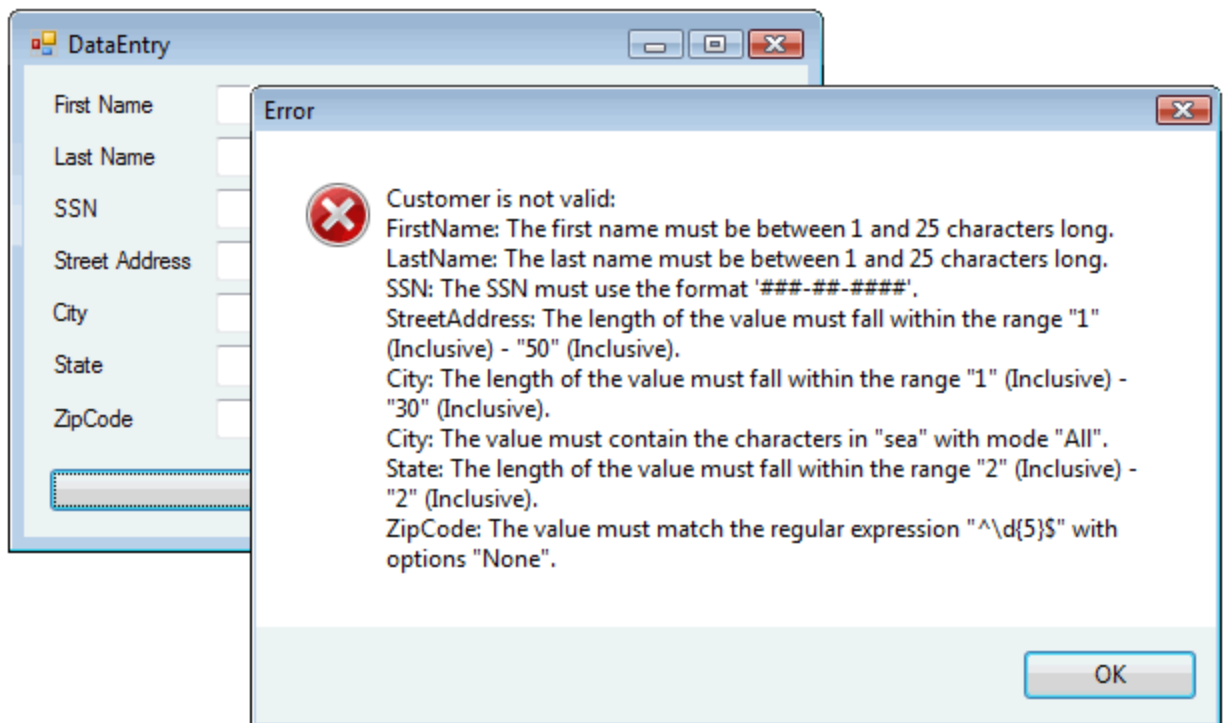
```
public class Customer
{
    [StringLengthValidator(1, 25,
        MessageTemplateResourceType = typeof(Resources),
        MessageTemplateName = "FirstNameMessage")]
    public string FirstName { get; set; }
    [StringLengthValidator(1, 25,
        MessageTemplateResourceType = typeof(Resources),
        MessageTemplateName = "LastNameMessage")]
    public string LastName { get; set; }
    [RegexValidator(@"^\d\d\d-\d\d-\d\d\d\d$",
        MessageTemplateResourceType = typeof(Resources),
        MessageTemplateName = "SSNMessage")]
    public string SSN { get; set; }
    [ObjectValidator]
    public Address Address { get; set; }
}
```

Verification

In this section, you will verify that the error message includes the overridden messages of each validation rule.

To validate that the custom message templates are used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names that are longer than 25 characters), and then click **Accept**. The error message illustrated here should display.



You can also compare the lab solution folder to the contents of the Lab4\\After folder. To see the business logic code that was added for this lab, see the ValidationHOL.BusinessLogic folder.

Lab 5: Specifying Validation Rules Through Configuration

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will practice using configuration files instead of attributes to specify validation rules.

Preparation

Continue working on the solution from Lab 4 and remove the validation attributes from the **Customer** and **Address** classes or open the solution file from Lab05\\Before\\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Creating a Configuration File
- Task 2: Building the Solution
- Task 3: Adding Validation Configuration to the Configuration File Using the Configuration Editor

- Task 4: Adding Validation Configuration for the **Customer** Class
 - Task 5: Adding Validation Configuration for the **Address** Class
-

Task 1: Creating a Configuration File

Add an empty application configuration file item to the **ValidationHOL** project.

To create a configuration file

1. In Solution Explorer, right-click the **ValidationHOL** project, point to **Add**, and then click **New Item**.
 2. In the templates list, click **Application Configuration File**, and then click **Add**.
-

For information about the .NET Framework configuration files, see [Configuration Files](#) on MSDN.

Task 2: Building the Solution

Right-click on the root node for the solution in Solution Explorer and click **Rebuild Solution**.

Task 3: Adding Validation Configuration to the Configuration File Using the Configuration Editor

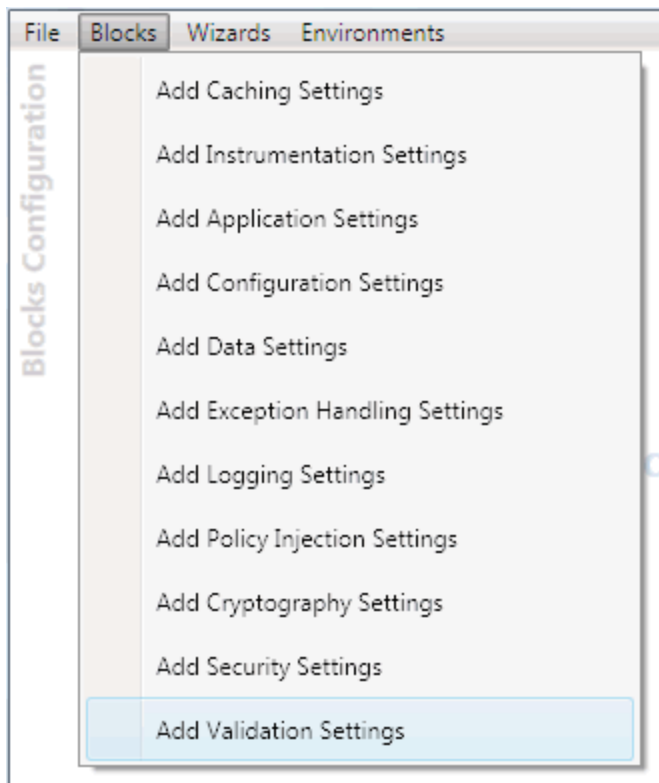
These instructions describe how to use the copy of the stand-alone configuration editor bundled with this lab, but any version of the editor can be used as long as it is compatible with the lab's binaries.

To add the required configuration information

1. From the Lib folder, start the configuration editor (EntLibConfig.exe).

Select the version of the configuration editor you require. There are versions specifically aimed at the 32-bit (x86) platform and versions compiled for any platform. For each of these platforms, there is a separate version for the 3.5 and 4.0 versions of the .NET Framework.

2. To open the App.config file for the ValidationHOL project created in the previous task, click **Open...** on the **File** menu, navigate to the location where the file resides, select the file, and then click **Open**.
3. Add the validation configuration section. On the **Blocks** menu select **Add Validation Settings**.

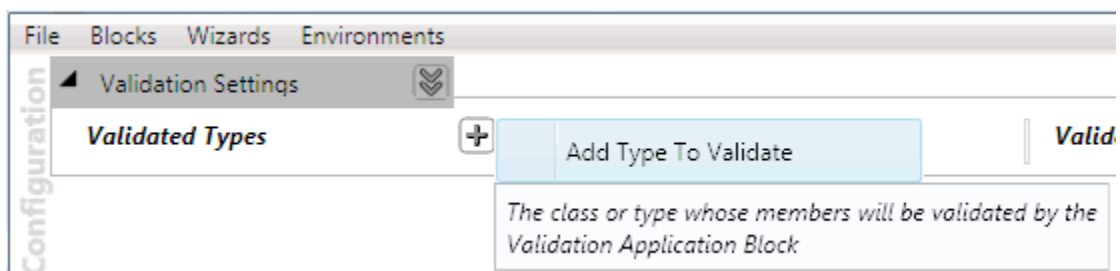


Task 4: Adding Validation Configuration for the Customer Class

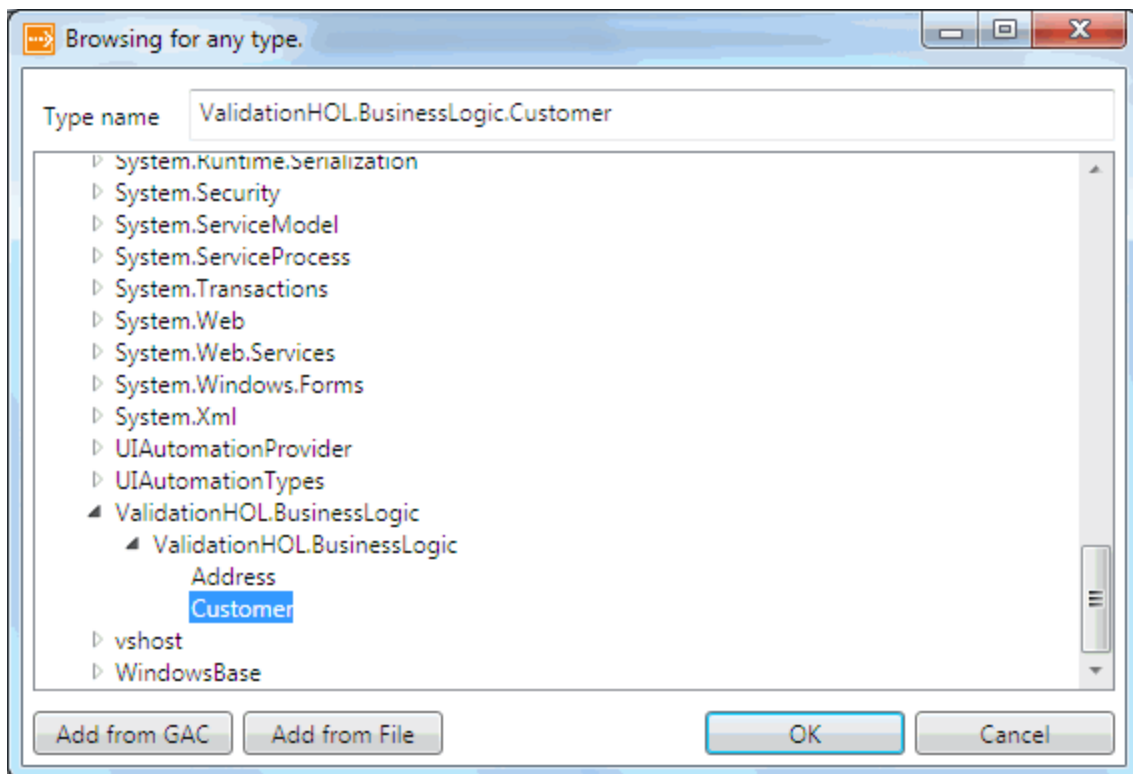
For each type requiring validation, you must add an entry to the validation block's configuration.

To add the required configuration information

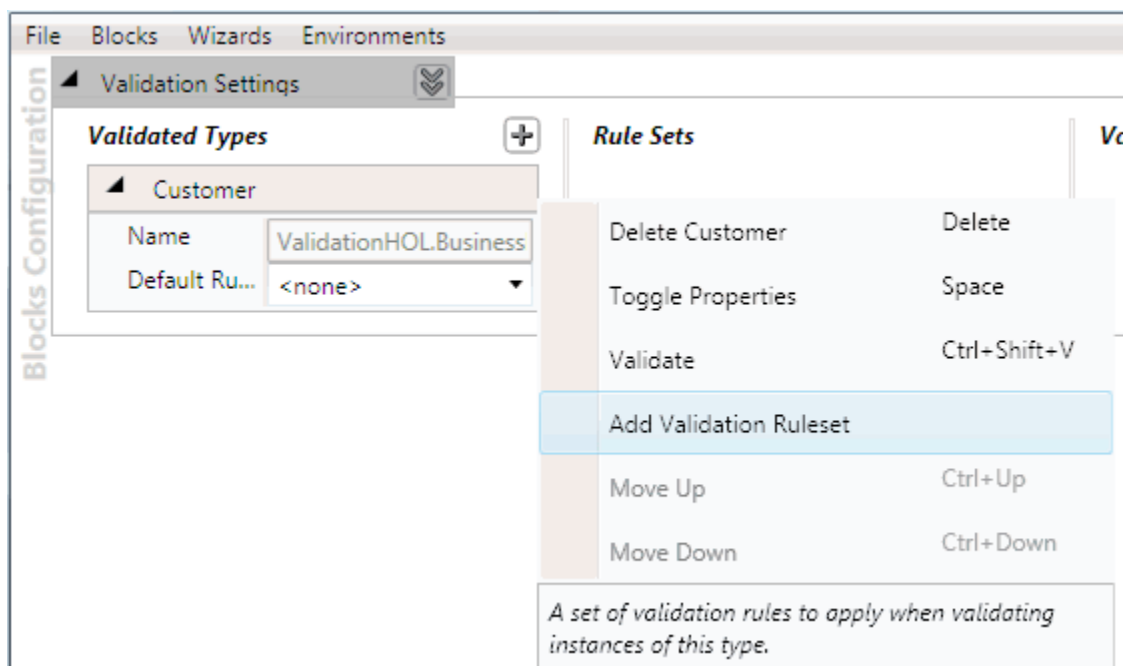
1. Add a new entry for the **Customer** type in the validation section. Click the plus sign icon to the right of **Validated Types**, and select **Add Type To Validate** to open the type selector dialog box.



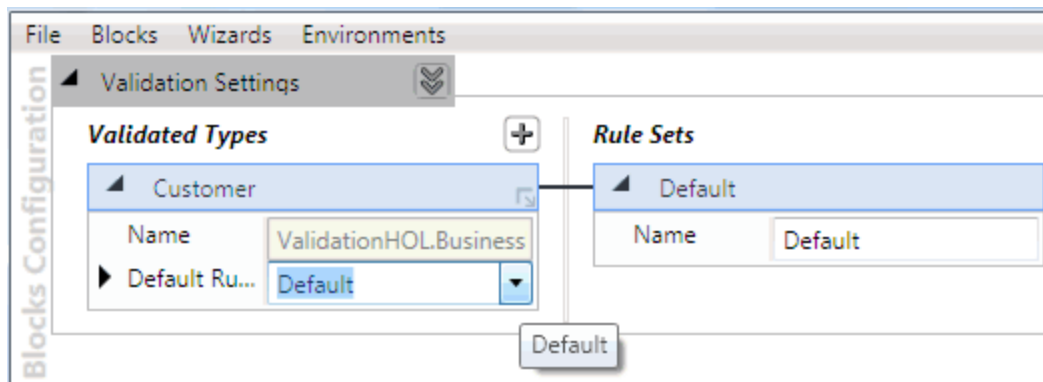
2. In the type selector dialog box select the type to validate, which in this case requires loading the assembly where it resides first (this is only necessary if it is not already loaded). Load the **ValidationHOL.BusinessLogic.dll** file. To do this, click **Add from File**, navigate to the output folder for the ValidationHOL.BusinessLogic project (typically ValidationHOL.BusinessLogic\bin\Debug), click the **ValidationHOL.BusinessLogic.dll** file, and then click **Open**. Now the **Customer** type will be available in the types tree. Locate the type or its name in the **Filter** box to filter the Types tree. Click the **Customer** type, and then click **OK**.



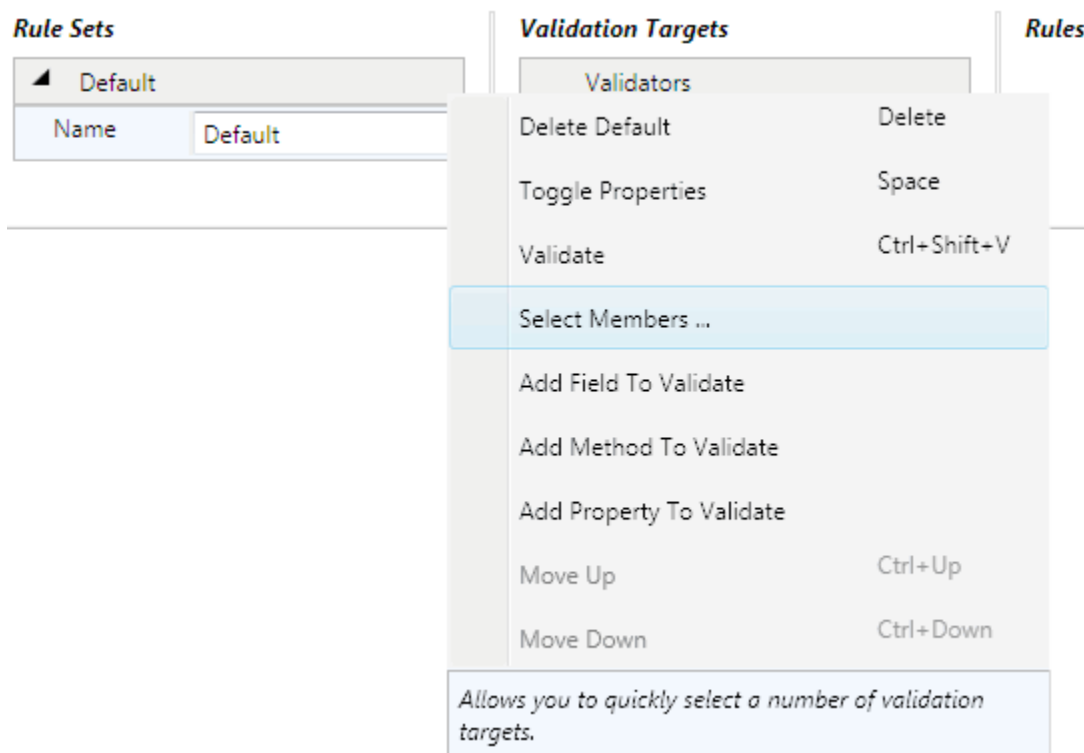
3. Add a new rule set for the **Customer** type and make it the default for the type. Right-click the **Customer** node, and click **Add Validation Ruleset**.



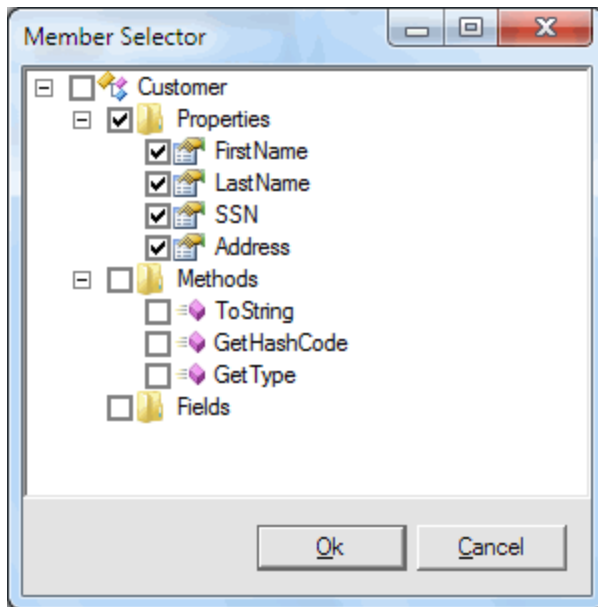
4. (Optional) Change the **Rule Set** name. In the **Rule Sets** pane type a name in the **Name** property edit box, such as **Default**.
5. Select the rule set for the **Customer** type. On the **Customer** property grid set the rule set to the **Default** rule set you just added by selecting the rule set from the drop down list.



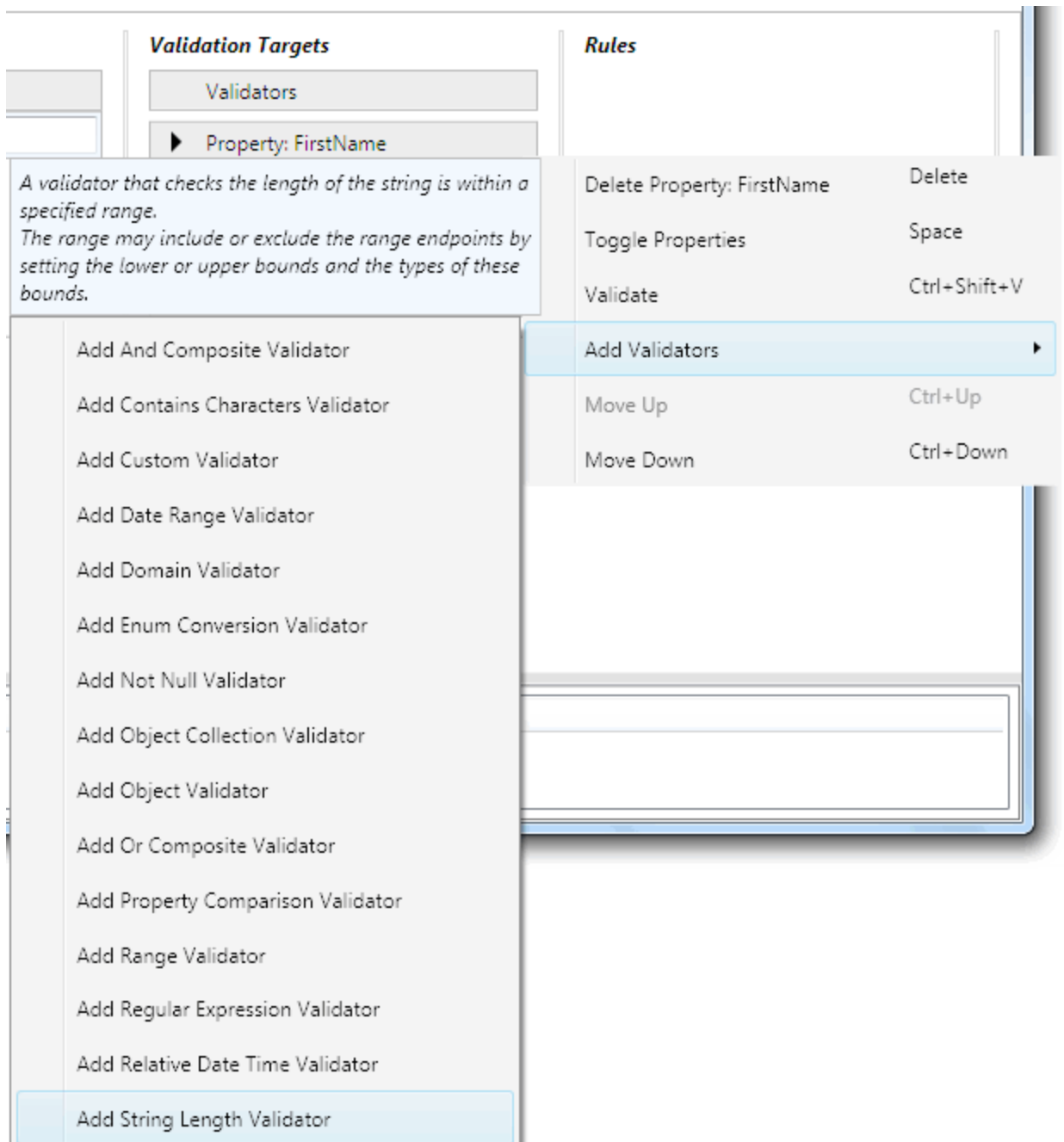
6. Select the **Customer** type's members to be validated. You can either add them one at a time or you can add them all at the same time by using the **Select members** feature. To add them all at the same time, right-click the **Default** node, and click **Select Members**.



Select the check boxes for all the properties, and then click **OK**.



7. Set up a **String Length Validator** for the **FirstName** property. Right click the **FirstName** node, select **Add Validators** and in the context menu click **Add String Length Validator**.



- Set the newly added **String Length Validator** properties. In the **Rules** pane **String Length Validator** node set **Lower Bound** to **1**, set **Lower Bound Type** to **Inclusive**, set **Upper Bound** to **25**, and set **Upper Bound Type** to **Inclusive**.

Validation Targets

Validators

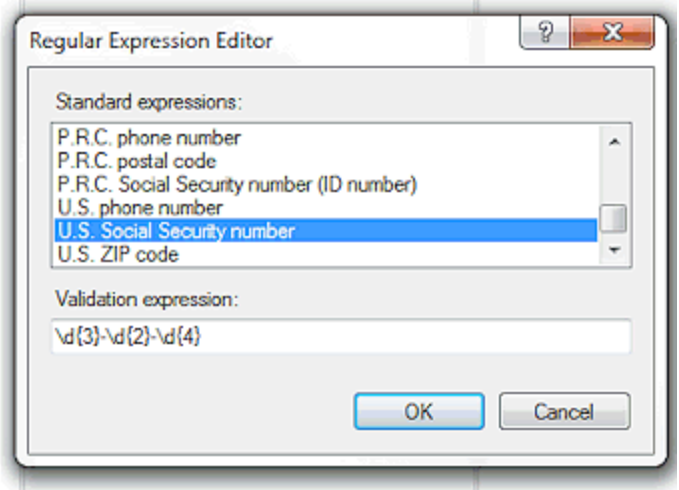
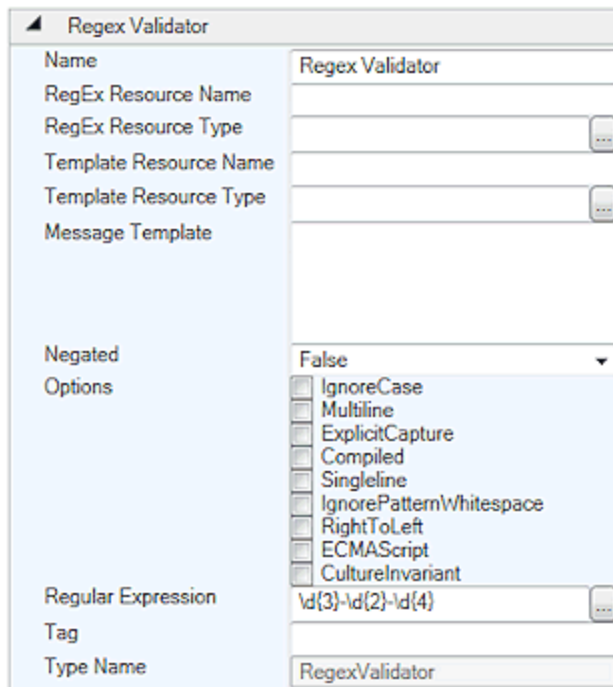
► Property: **FirstName**

Rules

String Length Validator

Name	String Length Validator
Template Resource Name	
Template Resource Type	...
Lower Bound	1
Lower Bound Type	Inclusive
Message Template	
Negated	False
Tag	
Type Name	StringLengthValidator
Upper Bound	25
Upper Bound Type	Inclusive

9. Set up a **String Length Validator** for the **LastName** property. Set **Lower Bound** to **1**, set **Lower Bound Type** to **Inclusive**, set **Upper Bound** to **25**, and set **Upper Bound Type** to **Inclusive**.
10. Set up a **Regular Expression Validator** for the **SSN** property. To specify the regular expression, click the ellipses (...) button in the **Regular Expression** property to open the Regular Expression Editor dialog. Select **U.S Social Security number** and click **OK**.



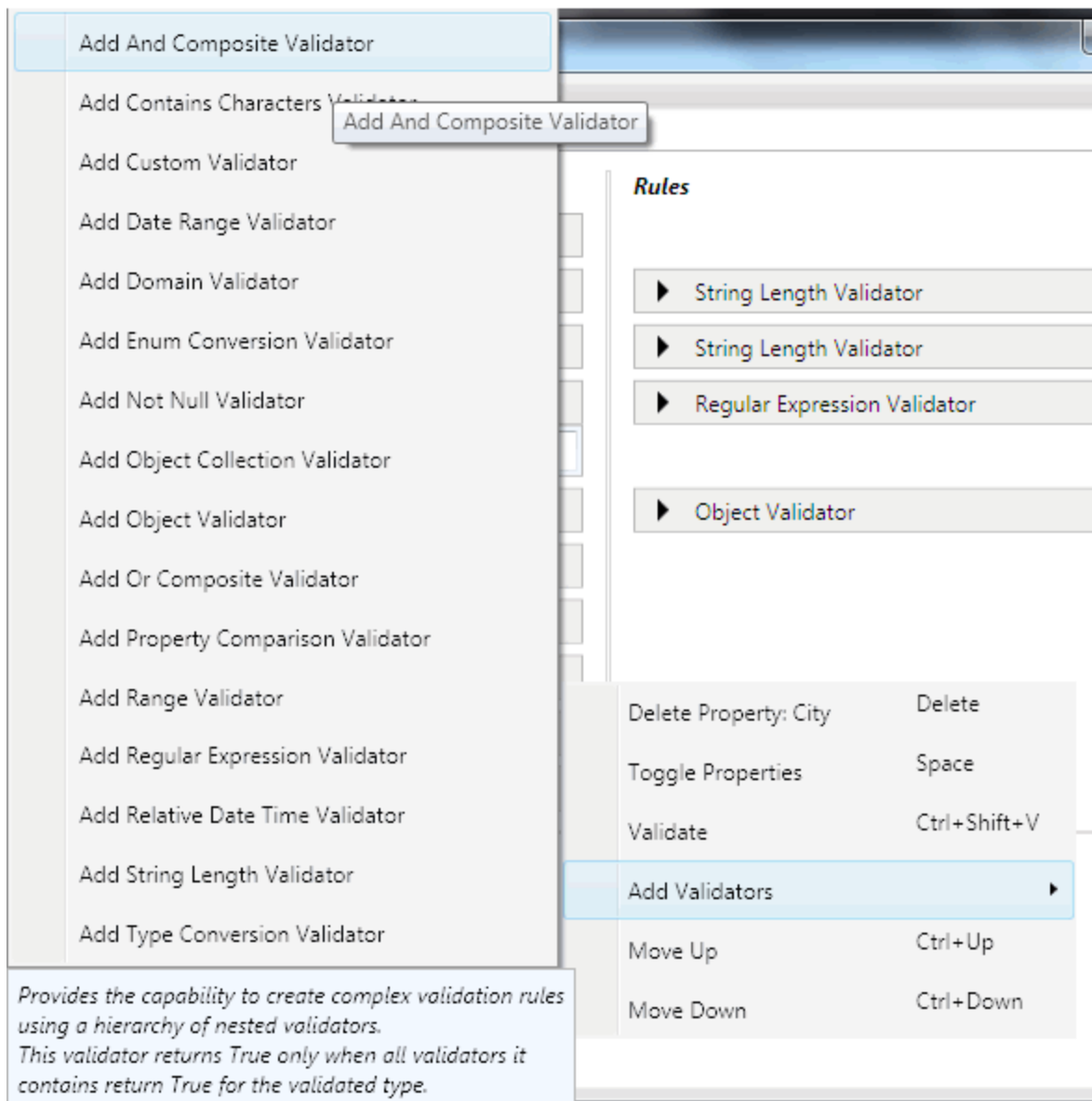
11. Set up an **Object Validator** for the **Address** property. No configuration is required for this validator.

Task 5: Adding Validation Configuration for the Address Class

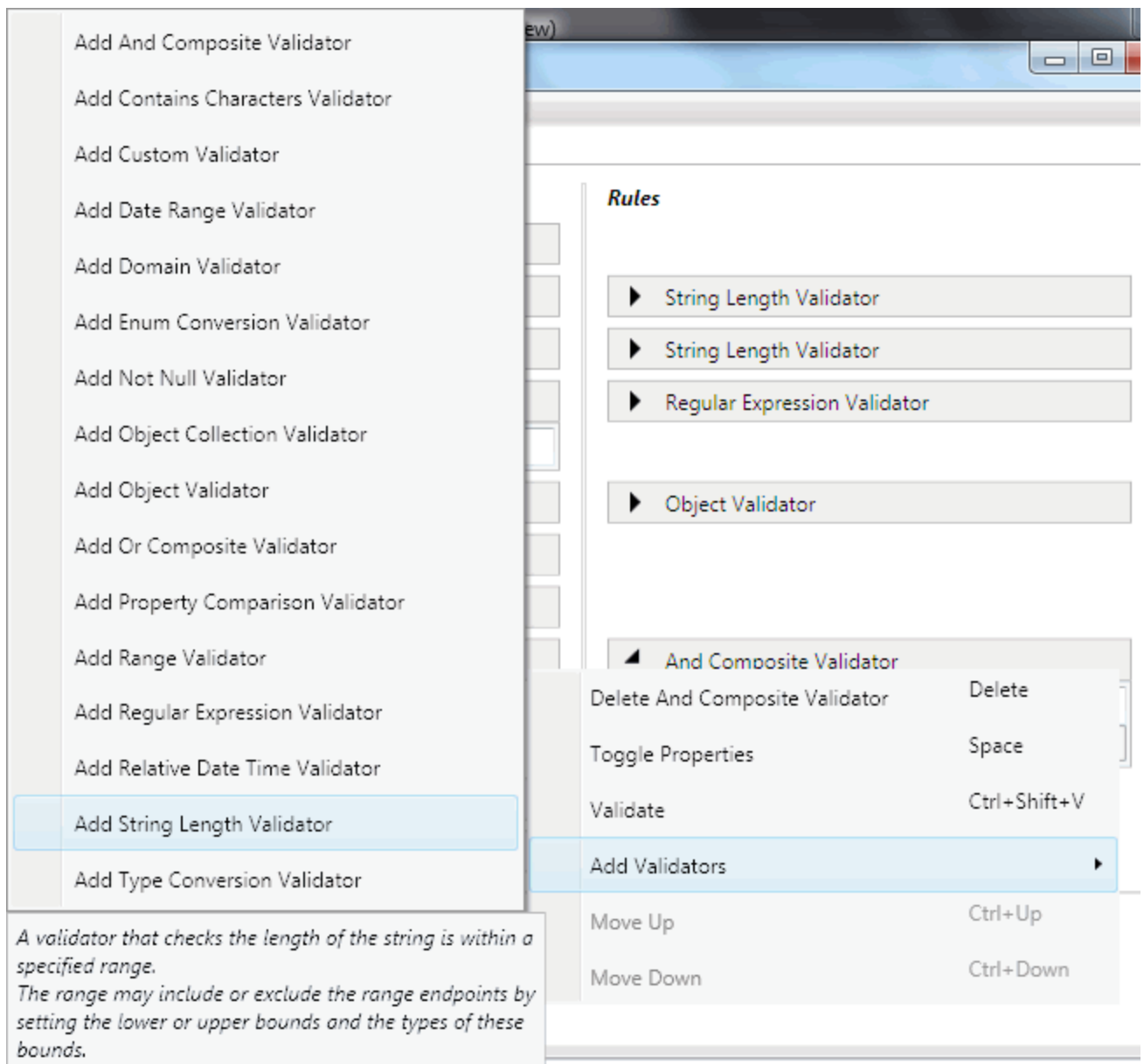
To add validation for a class, you must add type information, rule information, and validator information to the configuration file.

To add the required configuration information

1. Add the **Address** type to **Validated Types** by clicking the plus sign icon again in the **Validation Settings** section.
2. Add a new rule set for the **Address** type. Rename it to **Default** and make it the default rule set for the **Address** type.
3. Add entries for the **Address** type's members.
4. Set up an **And Composite Validator** for the **City** property using the same approach used to add validators to the **Customer** properties.



5. Add a **String Length Validator** and a **Contains Characters Validator** to the **And Composite Validator**.



6. Set **Lower Bound** to **1**, set **Lower Bound Type** to **Inclusive**, set **Upper Bound** to **30**, set **Upper Bound Type** to **Inclusive** for the **String Length Validator**. Set the **Character Set** to **sea**, and set **Contains Characters** to **All** for the **Contains Characters Validator**
7. Set up a **String Length Validator** for the **State** property. Set **Lower Bound** to **2**, set **Lower Bound Type** to **Inclusive**, and set **Upper Bound** to **2**, set **Upper Bound Type** to **Inclusive**.
8. Set up a **String Length Validator** for the **StreetAddress** property. Set **Lower Bound** to **1**, set **Lower Bound Type** to **Inclusive**, set **Upper Bound** to **50**, and set **Lower Bound Type** to **Inclusive**.
9. Set up a **Regular Expression Validator** for the **ZipCode** property. Select **U.S Zip Code** in Regular Expression Editor dialog.

10. Save the configuration file. On the **File** menu, click **Save** (Optionally click **Save As** to save to a different file name or location.)
11. Close the editor.

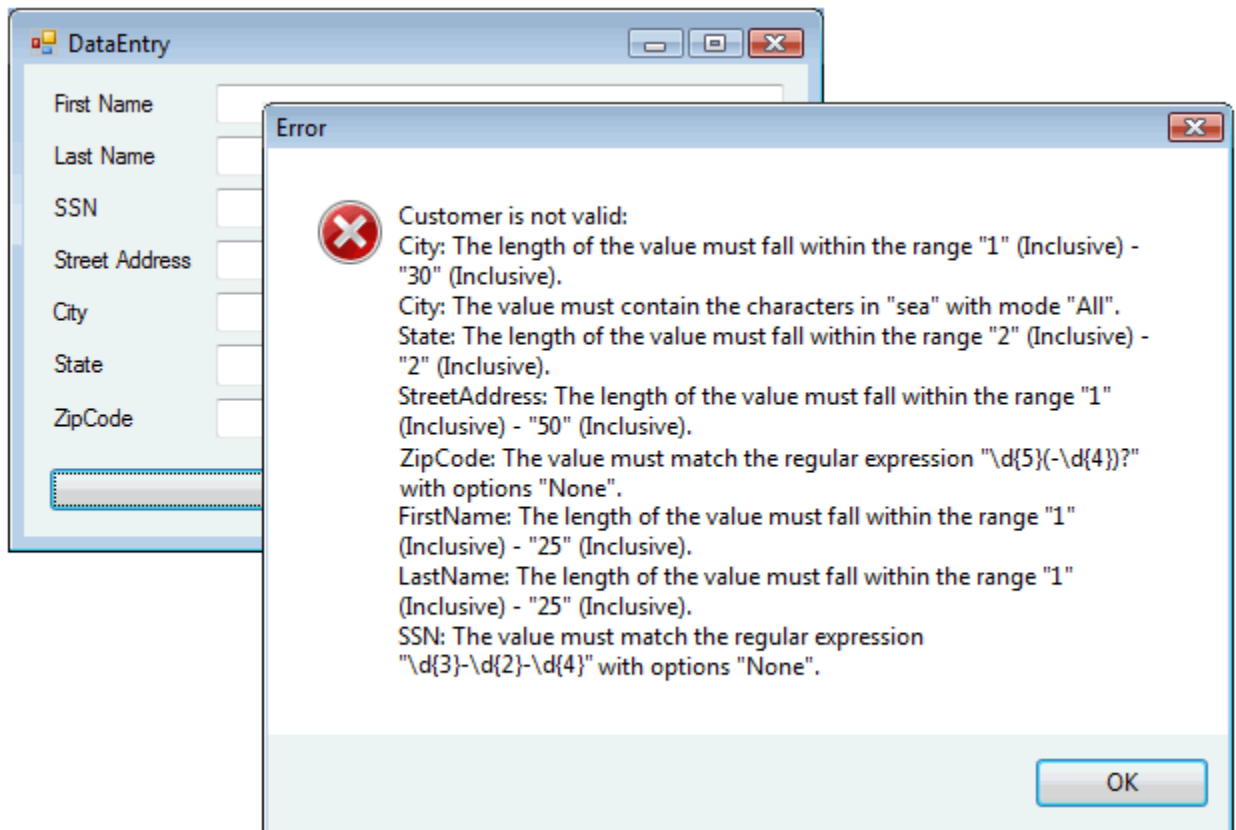
It is not strictly necessary to add the **AndCompositeValidator** in the **City** property. You can add several validator nodes to a property, which results in all the validators being grouped in an implicit **AndCompositeValidator**.

Verification

In this section, you will verify that the rules specified in the configuration file are used to validate the objects.

To validate that rules in the configuration file are used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names longer than 25 characters), and then click **Accept**. The following error message should display.



You can also compare the lab solution folder to the contents of the Lab5\\After folder.

Lab 6: Using Rule Sets

Estimated time to complete this lab: **10 minutes**

Purpose

In this lab, you will practice using multiple rule sets to validate the use of different rules in different contexts. You will add an additional rule set for the **Customer** class in the configuration file and add a check box in the user interface (UI) to indicate whether the customer should be validated using the default rule set, specified in the previous lab, or the new alternative one. Both **Name** and **Text** for the check box will be modified.

Multiple rule sets can be used with validation attributes by using the **Ruleset** named parameter, and rule sets can contain rules specified through attributes and through the configuration file, as long as the same rule set names are used in both locations.

Preparation

Continue working on the solution from Lab 5 or open the solution file from Lab06\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Adding and Configuring a New, Non-Default Rule Set in the Configuration File
- Task 2: Adding a Check Box to the UI to Choose Which Rule Set to Use
- Task 3: Updating the **MainForm** Class to Use the Value of the New Check Box to Determine Which Rule Set to Use to Validate the **Customer** Instance

Task 1: Adding and Configuring a New, Non-Default Rule Set in the Configuration File

Adding a new rule set named Alternative to the configuration file is done in the same way as the first one that was added in the previous lab. However, this new rule set will not be made the default for its type.

To add and configure the new rule set

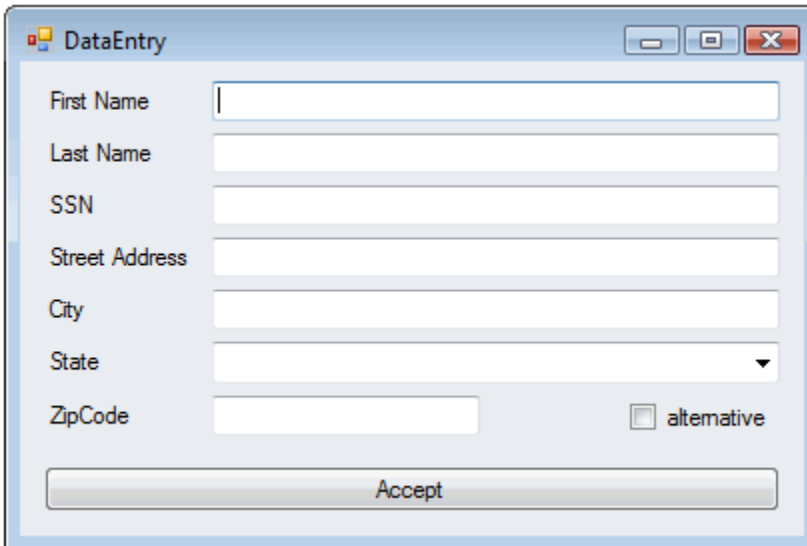
1. Open the configuration file for the ValidationHOL project using the configuration tool from the Lib folder.
2. Add a new rule set to the **Customer** class, as in the previous lab, and rename it to **Alternative**.
3. Add entries for the **FirstName** and **LastName** properties. You can choose the members as in the previous lab or just manually add the properties. To manually add the properties, right-

click the **Alternative** node, click **Add Property To Validate**, and then update the name as appropriate.

4. Set up a **String Length Validator** for the **First Name** property. Set **Lower Bound** to **1**, set **LowerBoundType** to **Inclusive**, and set **Upper Bound** to **20**, set **Upper Bound Type** to **Inclusive**.
5. Set up a **Domain Validator** for the **LastName** property. To do this, right-click **LastName**, point to **Add Validators**, and then click **Add Domain Validator**. Click the **Domain Validator** node, click the **Domain** property plus sign icon to add a **Domain Configuration Element**. Type an entry for the **Domain Configuration Element**. Add entries for **Smith**, **Jones**, and **Doe** by clicking the plus sign icon once for each entry and updating each entry added. This validator will allow only the specified elements as last names.
6. Save and close the configuration.

Task 2: Adding a Check Box to the UI to Choose Which Rule Set to Use

Add a new check box to the **MainForm**. Name the new check box **alternativeValidation** and label the check box "alternative".

The image shows a Windows-style application window titled "DataEntry". It contains several text input fields: "First Name", "Last Name", "SSN", "Street Address", "City", "State" (which is a dropdown menu), and "ZipCode". To the right of the "ZipCode" field is a checkbox with the label "alternative". At the bottom of the form is a large button labeled "Accept".

Task 3: Update the MainForm Class to Use Value of the New Checkbox to Determine Which Rule Set to Use to Validate the Customer Instance

Now you can add an additional validator that will apply the Alternative rule set.

To use the Additional rule set

1. Add a member variable to the **MainForm** class to store the validator for the alternate rule set, and update the **MainForm_Load** method to create both validators.

```
private Validator<Customer> customerValidator;  
private Validator<Customer> alternativeValidator;
```

```

private void MainForm_Load(object sender, EventArgs e)
{
    ValidatorFactory valFactory =

EnterpriseLibraryContainer.Current.GetInstance<ValidatorFactory>();
    customerValidator = valFactory.CreateValidator<Customer>();
    alternativeValidator =
    valFactory.CreateValidator<Customer>("Alternative");
}

```

2. Update the **acceptButton_Click** method in the **MainForm** class to test the **Checked** property in the new check box to determine which rule set to use when validating the **Customer** instance.

```

private void acceptButton_Click(object sender, EventArgs e)
{
    Customer customer = new Customer
    {
        FirstName = firstNameTextBox.Text,
        LastName = lastNameTextBox.Text,
        SSN = ssnTextBox.Text,
        Address = new Address
        {
            StreetAddress = streetAddressTextBox.Text,
            City = cityTextBox.Text,
            State = stateComboBox.Text,
            ZipCode = zipCodeTextBox.Text
        }
    };

    ValidationResults results =
        this.alternativeValidation.Checked
        ? alternativeValidator.Validate(customer)
        : customerValidator.Validate(customer);
    if (!results.IsValid)
    {
        StringBuilder builder = new StringBuilder();
        builder.AppendLine("Customer is not valid:");
        foreach (ValidationResult result in results)
        {
            builder.AppendLine(
                string.Format(
                    CultureInfo.CurrentCulture,
                    "{0}: {1}",
                    result.Key,
                    result.Message));
        }
        MessageBox.Show(
            this,
            builder.ToString(),

```

```

        "Error",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    return;
}

MessageBox.Show(
    this,
    "Processing customer '" + customer.FirstName + "'",
    "Working",
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
}

```

Verification

In this section, you will verify that the different rule sets are used, depending on the status of the **alternative** check box.

To validate that the custom message templates are used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank or add long or invalid values, and then click **Accept**. The error messages displayed in the message box should correspond to the default rule set.
3. Select the **alternative** check box and repeat the previous step. The error messages displayed in the message box should correspond to the **Alternative** rule set.

You can also compare the lab solution folder to the contents of the Lab6\After folder.

Lab 7: Integrating Windows Forms

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will practice integrating the Validation Application Block into a form using the native validation primitives of Windows Forms. Unlike in the previous labs, this mechanism is used to validate a control's values, not instances of the types for which validation has been specified.

For an introduction to Windows Forms user input validation, see [User Input Validation in Windows Forms](#) on MSDN. For information about the Windows Forms integration feature of the Validation Application Block, and the **ValidationProvider** in particular, see [Integrating with Windows Forms](#) on MSDN. For information about the extender provider mechanism used to implement the validation integration, see [ExtenderProvider Interface](#) on MSDN.

Preparation

Open the solution file from Lab07\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

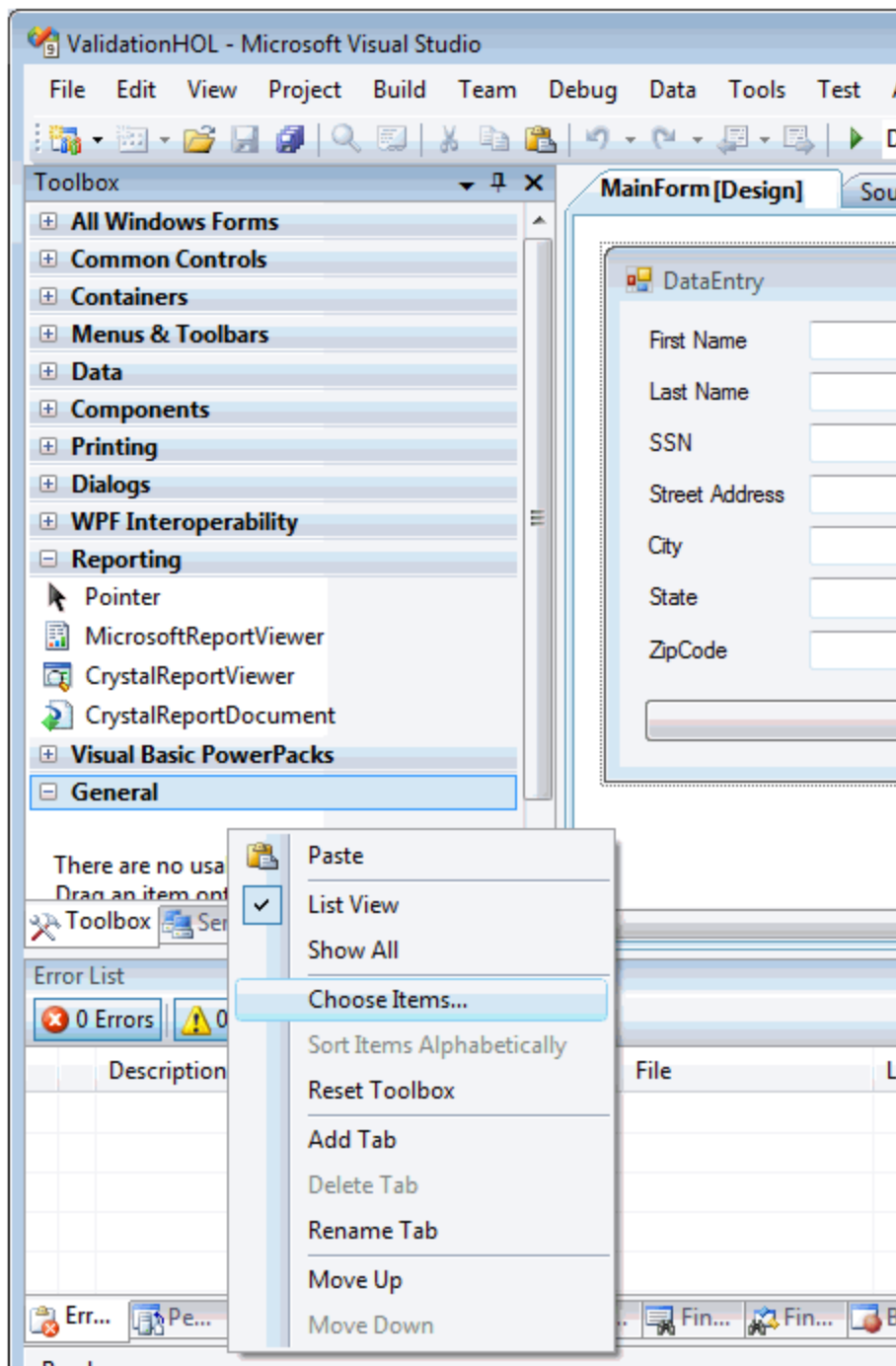
- Task 1: Adding the **ValidationProvider** to the Windows Forms Toolbox
 - Task 2: Adding an **ErrorProvider** to the Form
 - Task 3: Adding Validation Support for the **Customer** Fields
 - Task 4: Adding Validation Support for the **Address** Fields
-

Task 1: Adding the ValidationProvider to the Windows Forms Toolbox

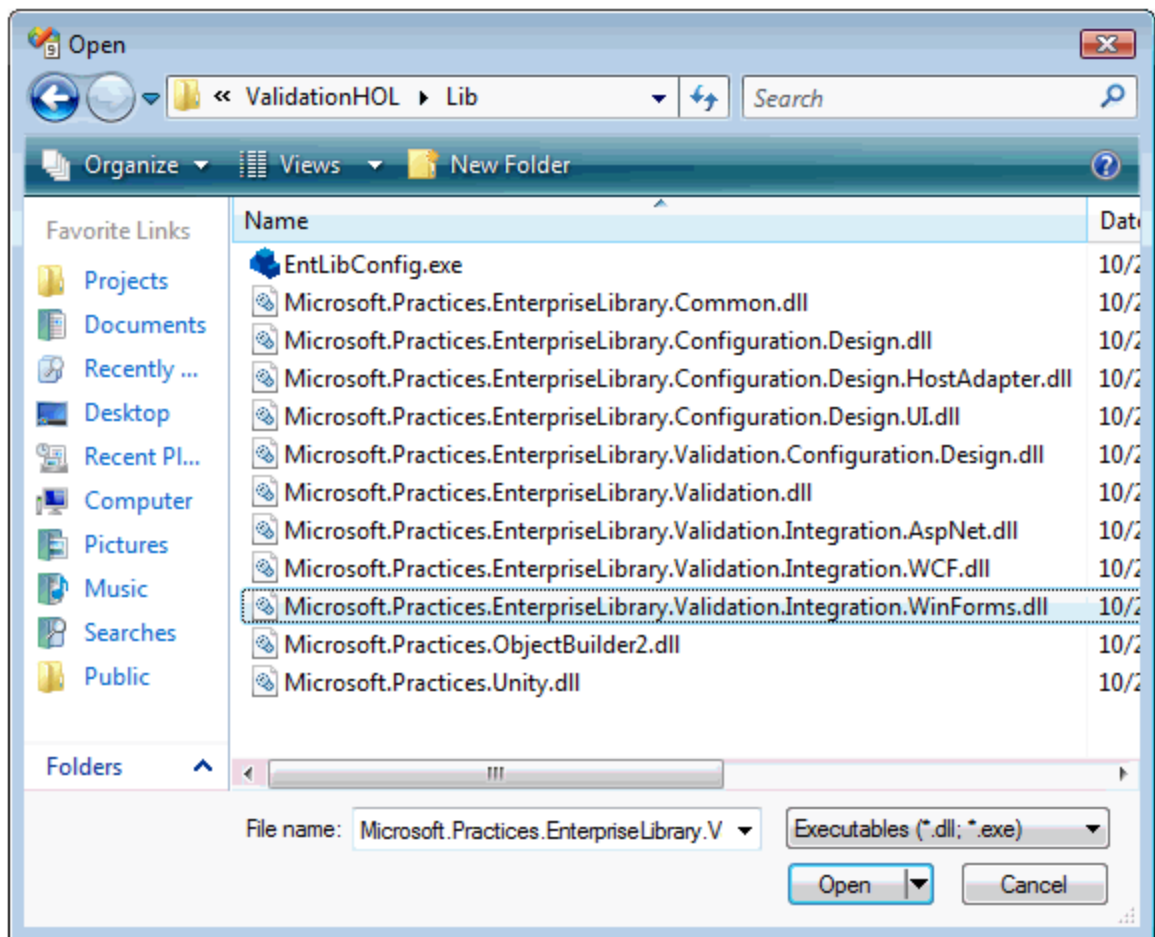
Adding an item to the toolbox makes it available when using a designer. This step is necessary only once per instance of Visual Studio®, unless it is manually removed.

To add the ValidationProvider to the toolbox

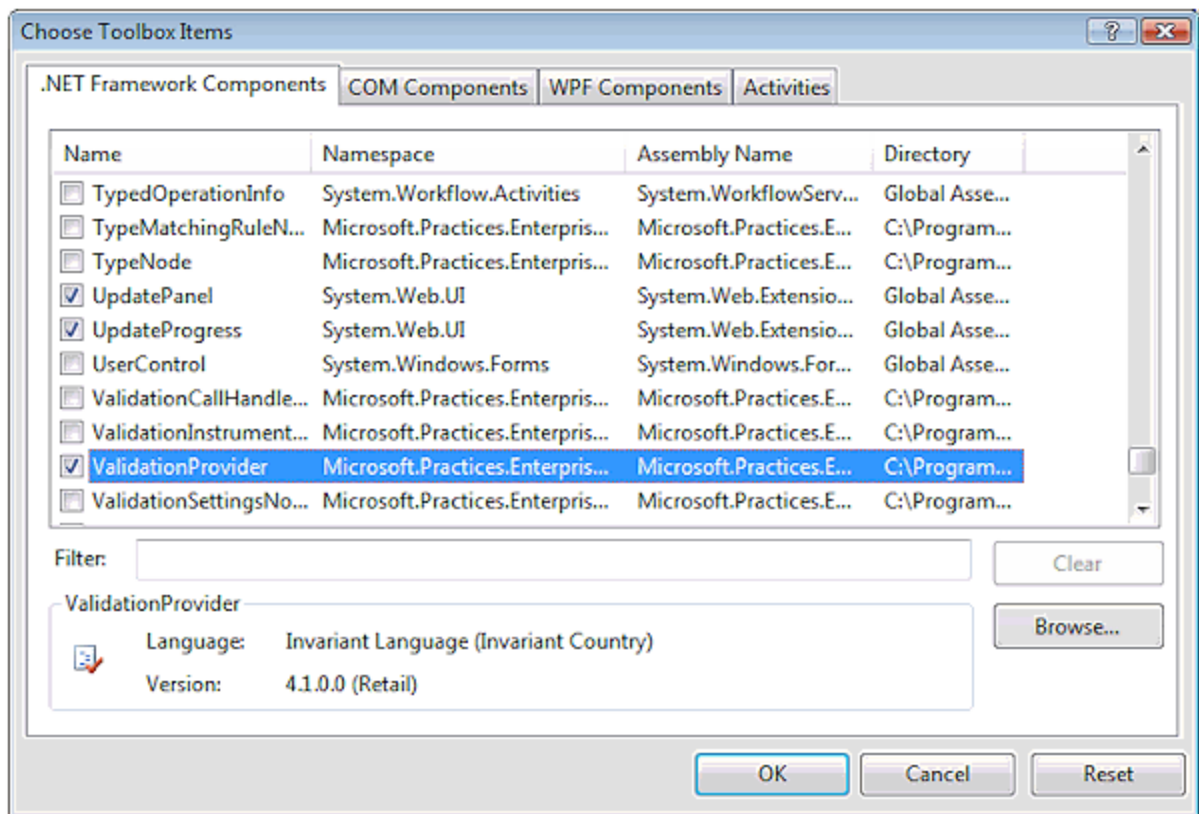
1. In Design mode, open the MainForm.cs file.
2. Open the **Toolbox** tool window.
3. Right-click the tool window surface, and then click **Choose Items**. This operation may take a few minutes to complete.



4. Click the **Browse** button, navigate to the Lib folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms.dll** assembly, and then click **Open**.



5. Select the check box for **ValidationProvider**, and then click **OK**.



Task 2: Adding an ErrorProvider to the Form

This task is not strictly necessary for validation integration to work. However, the standard **ErrorProvider** component is typically used together with the **ValidationProvider** to provide feedback to the user when the user input is invalid.

To add an ErrorProvider to the form

1. Expand the **Components** category in the **Toolbox**.
2. Click the **ErrorProvider** item and drag it to the designer surface.

Task 3: Adding Validation Support for the Customer Fields

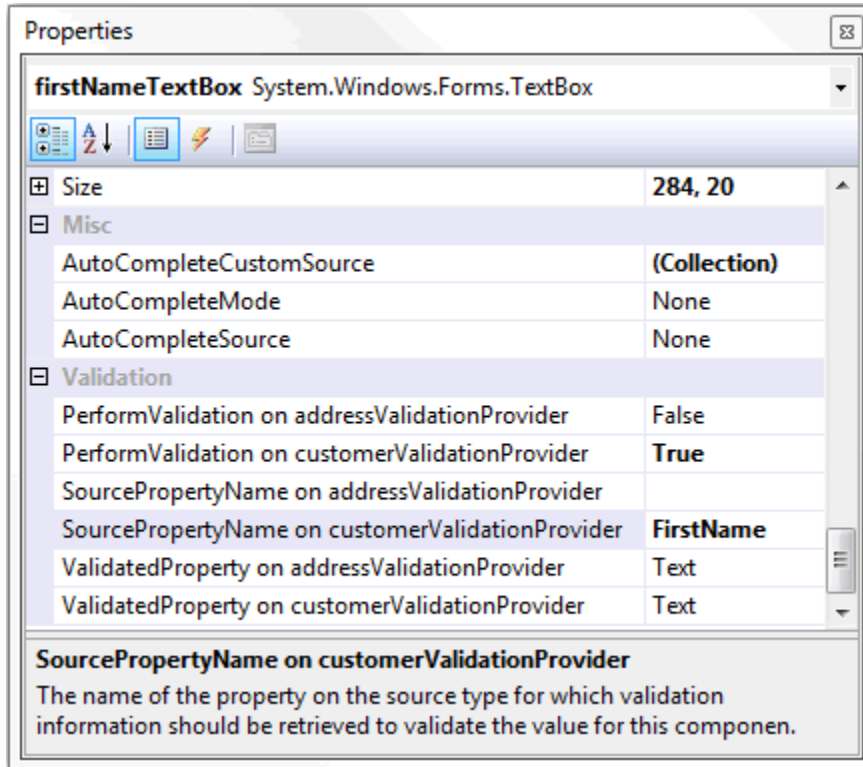
A validation provider is required for each type for which validation rules have been specified. The validation provider extends each control in the form with new properties that specify whether validation needs to be performed on that control using the rules specified for the provider's type and which property should be used as the source of the validation rules.

To add validation support for the customer fields

1. In the **Toolbox**, expand the **General** category.
2. Click the **ValidationProvider** item and drag it to the designer surface.
3. In the **Properties** tool window, change the provider's name to **customerValidationProvider**, set **SourceTypeName** to "ValidationHOL.BusinessLogic.Customer,

ValidationHOL.BusinessLogic" (without the quotation marks), and then set **ErrorProvider** to **errorProvider1**.

4. Click the **First Name** box. In the **Validation** section of the Visual Studio Properties window set **Performs validation on customerValidationProvider** to **True** and set **SourcePropertyName** on **customerValidationProvider** to **FirstName**.



5. Click the **Last Name** box. Set **Performs validation on customerValidationProvider** to **True** and set **SourcePropertyName** on **customerValidationProvider** to **LastName**.
6. Click the **SSN** box. Set **Performs validation on customerValidationProvider** to **True** and set **SourcePropertyName** on **customerValidationProvider** to **SSN**.

Task 4: Adding Validation Support for the Address Fields

A new validation provider is required to validate controls using the rules specified for the properties in the **Address** type. You must be careful to specify the extended properties in the controls that require validation provided by the appropriate validation provider.

To add validation support for the address fields

1. Click the **ValidationProvider** item and drag it to the designer surface.
2. In the **Properties** tool window, change the provider's name to **addressValidationProvider**, set **SourceType** to **"ValidationHOL.BusinessLogic.Address, ValidationHOL.BusinessLogic"** (not including the quotation marks) and set **ErrorProvider** to **errorProvider1**.

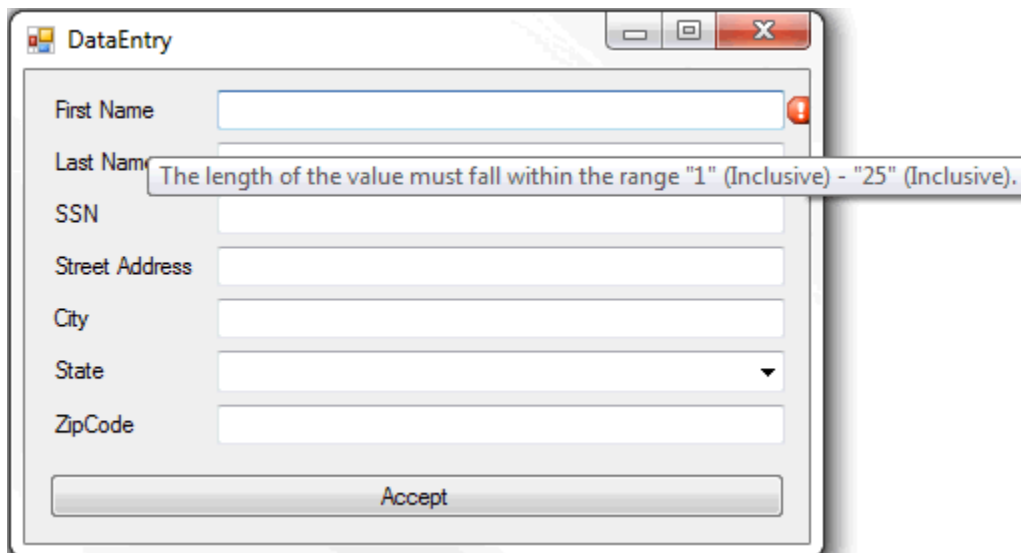
3. Click the **Street Address** box. In the **Validation** section of the Visual Studio Properties window set **Performs validation on addressValidationProvider** to **True** and set **SourcePropertyName on addressValidationProvider** to **StreetAddress**.
4. Click the **City** box. Set **Performs validation on addressValidationProvider** to **True** and set **SourcePropertyName** to **City**.
5. Click the **State** combo box. Set **Performs validation on addressValidationProvider** to **True** and set **SourcePropertyName on addressValidationProvider** to **State**.
6. Click the **Zip Code** box. Set **Performs validation on addressValidationProvider** to **True** and set **SourcePropertyName on addressValidationProvider** to **ZipCode**.

Verification

In this section, you will verify that validation on the individual fields is performed when the standard Windows Forms **Validating** event is triggered.

To validate that user input is validated

1. Build and run the ValidationHOL project.
2. In the form, leave a field blank or add long or invalid values (such as a social security number that does not match the ###-##-#### format or names that are longer than 25 characters), and then click the **Accept** button. The focus returns to the invalid field and a tooltip, added by the standard **ErrorProvider** will be displayed next to the field.



3. Due to the presence of the validation providers, you must stop the application by clicking the Stop button in Visual Studio.

Lab 8: Implementing Self Validation

Estimated time to complete this lab: **15 minutes**

Purpose

In this lab, you will practice implementing self-validation methods. Self-validation can be used to perform arbitrary validation logic and complement the use of validators. For information about self-validation methods, see [Using Self Validation](#) on MSDN.

In this lab, a self-validation method will be used to validate a customer's social security number more thoroughly. The format of a social security number is well known and can be easily represented with a regular expression, but there are additional checks that can be performed on a social security number to determine whether it is valid. For more information, see http://ssa-custhelp.ssa.gov/cgi-bin/ssa.cfg/php/enduser/std_adp.php?p_faqid=425.

Preparation

Continue working on the solution from Lab 7 or open the solution file from Lab08\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Removing the Configuration for the Validation of the **SSN** Property From the Configuration File
- Task 2: Implementing a Self-Validation Method to Validate the **SSN** Property from the **Customer** Class

Task 1: Removing the Configuration for the Validation of the SSN Property from the Configuration File

You must first remove the configuration for the validation of the **SSN** property from the configuration file before you can implement a self-validation method to validate the **SSN** property.

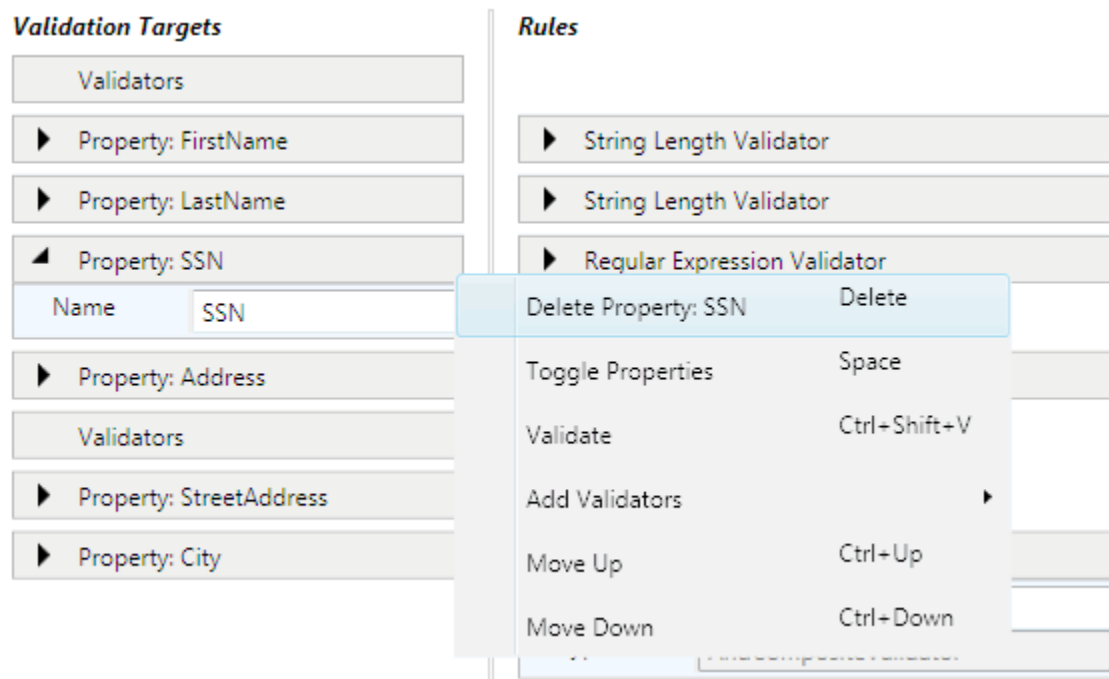
To remove the configuration for the validation of the SSN property

1. In the Lib folder, start the configuration editor (EntLibConfig.exe).

Select the version of the configuration editor you require. There are versions specifically aimed at the 32-bit (x86) platform and versions compiled for any platform. For each of these platforms, there is a separate version for the 3.5 and 4.0 versions of the .NET Framework.

2. In the ValidationHOL project, open the App.config file.
3. Locate the node for the **SSN** property. The path navigate to this node in the configuration tool is Validation Settings | Validated Types | [Customer] | Rule Sets | [Default] | ValidationTargets | [Property: SSN].

- Remove the node. To do this, right-click the **Property: SSN** node, and then click **Delete Property SSN** (or click on the node then click the keyboard **Delete** key).



- Save the configuration file and close the editor.

Task 2: Implementing a Self-Validation Method to Validate the SSN Property from the Customer Class

After you remove the configuration for the validation of the **SSN** property from the configuration file, you can implement a self-validation method to validate the **SSN** property.

To implement a self-validation method

- Add a reference to the Validation assembly to the ValidationHOL.BusinessLogic project. To do this, right-click the ValidationHOL.BusinessLogic project in Solution Explorer and then click **Add Reference**. Click the **Browse** tab, navigate to the Lib folder in the lab's main folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** file, and then click **OK**.
- Add a **using** directive to the Customer.cs file to make the required types available without full name qualification. Be sure to open the file in the code view.

```
using System;
using System.Text.RegularExpressions;
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

- Add the **[HasSelfValidation]** attribute to the **Customer** class.

```
[HasSelfValidation]
public class Customer
{
```



```
...  
}
```

4. Implement a self-validation method in the **Customer** class with the **[SelfValidation]** attribute. Self-validation methods must be void and have a single **ValidationResults** parameter, and they should populate the parameter with **ValidationResult** instances to indicate validation failures.

```
static Regex ssnCaptureRegex =  
    new Regex(@"^(?<area>\d{3})-(?<group>\d{2})-(?<serial>\d{4})$");  
  
[SelfValidation]  
public void ValidateSSN(ValidationResults validationResults)  
{  
    // Validation logic according to  
    // http://ssa-custhelp.ssa.gov/cgi  
    // -bin/ssa.cfg/php/enduser/std_adp.php?p_faaid=425  
  
    Match match = ssnCaptureRegex.Match(this.SSN);  
    if (match.Success)  
    {  
        string area = match.Groups["area"].Value;  
        string group = match.Groups["group"].Value;  
        string serial = match.Groups["serial"].Value;  
  
        if (area == "666"  
            || string.Compare(area, "772", StringComparison.Ordinal) > 0)  
        {  
            validationResults.AddResult(  
                new ValidationResult(  
                    "Invalid area",  
                    this,  
                    "SSN",  
                    null,  
                    null));  
        }  
        else if (area == "000" || group == "00" || serial == "0000")  
        {  
            validationResults.AddResult(  
                new ValidationResult(  
                    "SSN elements cannot be all '0'",  
                    this,  
                    "SSN",  
                    null,  
                    null));  
        }  
    }  
    else  
    {  
        validationResults.AddResult(  

```

```

        new ValidationResult(
            "Must match the pattern '###-##-####'",
            this,
            "SSN",
            null,
            null));
    }
}

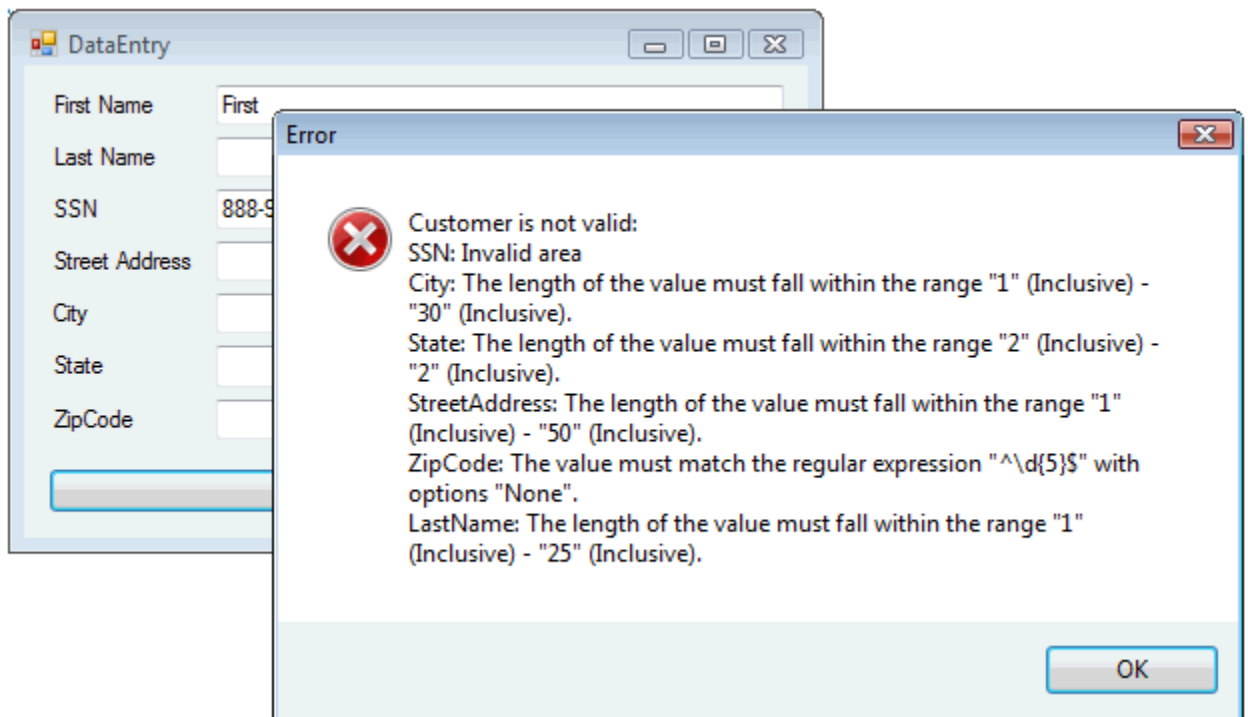
```

Verification

In this section, you will verify that the custom validation performed by the self-validation method is used to validate the object.

To validate that the self validation method is used to validate the object

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank, except for the **First Name**, **Last Name**, and the **SSN** text boxes. A valid value for the **First Name** and **Last Name** are required to change focus and enter an SSN value. Use a value for the SSN such as 666-78-9999 and click **Accept** to trigger a specialized error message that cannot be easily specified using the standard validator used before. The following figure illustrates the error message.



The self-validation logic was not invoked to validate the **SSN** box as in the previous labs. Integration with Windows Forms' validation mechanism utilizes the validation rules associated with a business object, but no instance of the validated type is actually used. Self-validation relies on methods on the validated type, so the validation rules implemented by such methods cannot be used to validate user

input. The next lab shows how to encapsulate the validation logic in a custom validator, which can then be used to validate user input just as you would with the built-in validators used for other properties.

You can compare the lab solution folder to the contents of the Lab8\After folder.

Lab 9: Implementing a Custom Validator

Estimated time to complete this lab: **15 minutes**

Purpose

In this lab, you will practice implementing a new validator and the corresponding attribute for it, and then you will use it to validate a business object and to perform integration with the Windows Forms user input validation mechanism.

Preparation

Continue working on the solution from Lab 8 or open the solution file from Lab09\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

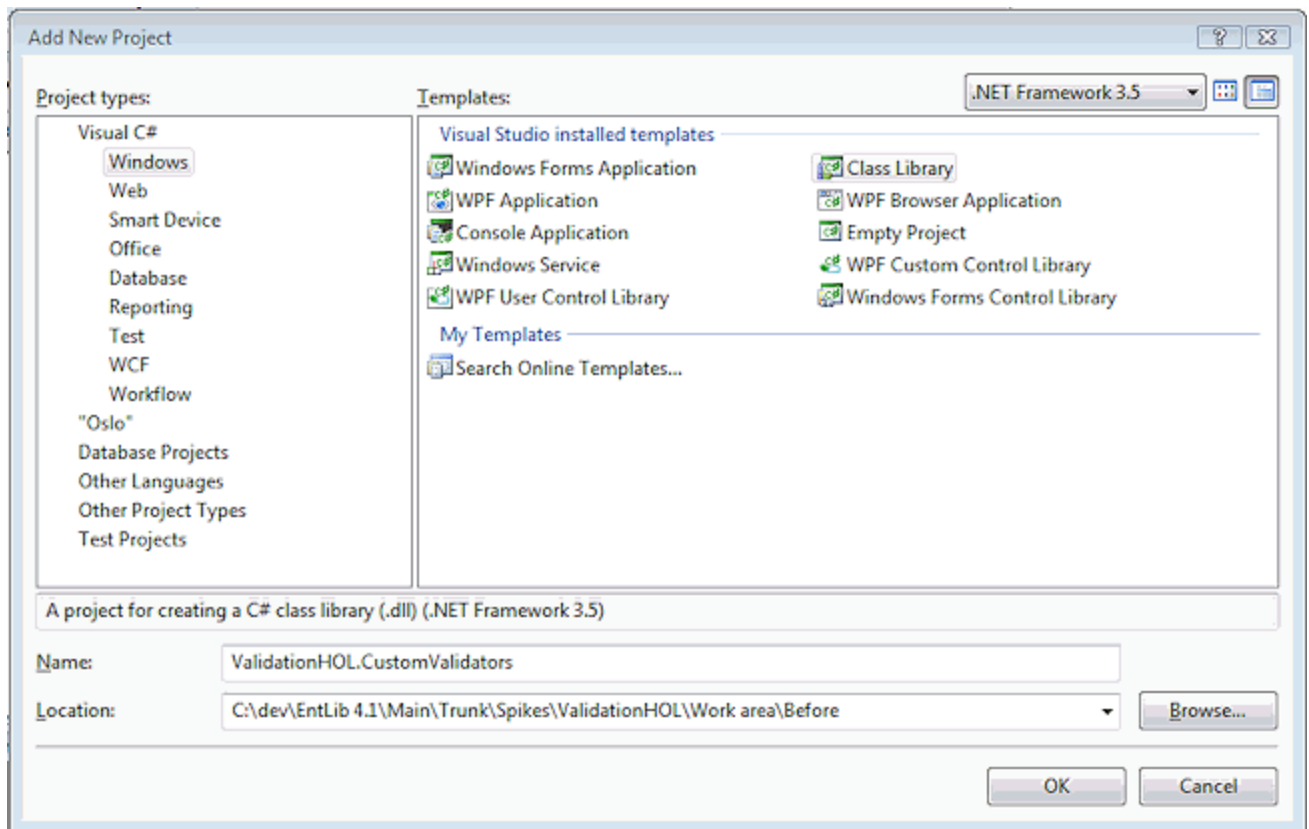
- Task 1: Creating a New Project for the Custom SSN Validator
- Task 2: Implementing the SSN Validator
- Task 3: Implementing the SSN Validator Attribute
- Task 4: Using the SSN Validator Instead of the Built-In One

Task 1: Creating a New Project for the Custom SSN Validator

The custom validator will reside in a separate project. This is not strictly necessary, but it is recommended to facilitate reuse.

To create a new project for the custom validator

1. Create the new project. To do this, right-click the solution node in the Solution Explorer, point to **Add**, and then click **New Project**. Click the **Class Library** template, enter **ValidationHOL.CustomValidators** as the new project name, and then click **OK**.



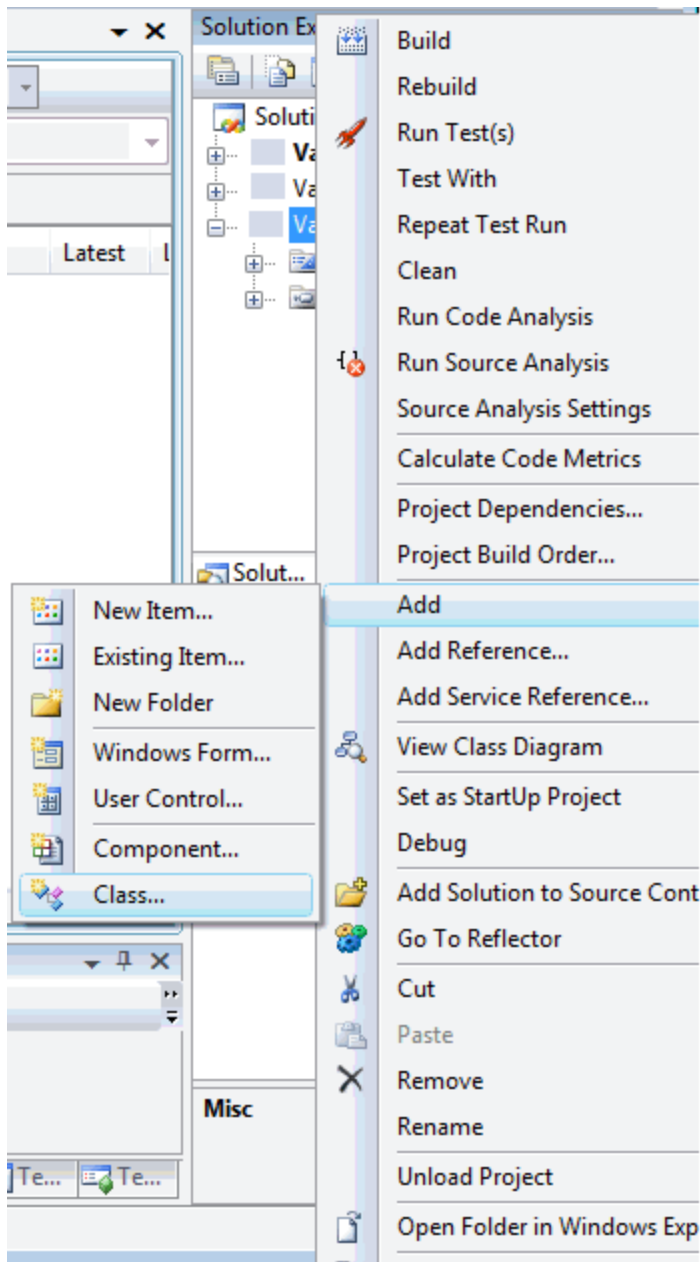
2. Add a reference to the Data Annotations assembly to the new class. This is required to use the **ValidatorAttribute** class. To do this, right-click the ValidationHOL.CustomValidators project in Solution Explorer and then click **Add Reference**. Click the **.NET** tab, click **System.ComponentModel.DataAnnotations**, and then click **OK**.
3. Add a reference to the Validation assembly to the ValidationHOL.CustomValidators project. To do this, right-click the ValidationHOL.BusinessLogic project in Solution Explorer and then click **Add Reference**. Click the **Browse** tab, navigate to the Lib folder in the lab's main folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** file, and then click **OK**.
4. Delete the Class1.cs file created by default in the new project.

Task 2: Implementing the SSN Validator

A validator is simply a class that inherits from the **Validator** class and overrides some specific methods, as shown in the example in this task.

To implement the SSN validator

1. Add a new **SSNValidator** class to the **ValidationHOL.CustomValidators** project. To do this, right-click the **ValidationHOL.CustomValidators** node in the Solution Explorer, point to **Add**, and then click **Class**. In the **New Class** dialog box, type **SSNValidator** in the **Name** box, and then click **Add**.



2. Add **using** directives to the SSNValidator.cs file to make the necessary types available without full name qualification.

```
using System.Text.RegularExpressions;
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

3. Implement the validator as shown in the following code example. This validator implements a validation logic that is slightly more complex than the one used for the self-validation method in previous labs; this validator can be configured to ignore the hyphens in the social security number. This validator defines two constructors and overrides the **DoValidate** method.

```
public class SSNValidator : Validator<string>
```

```

{
    public SSNValidator(string tag)
        : this(tag, false)
    {
    }

    public SSNValidator(string tag, bool ignoreHypens)
        : base(string.Empty, tag)
    {
        this.ignoreHypens = ignoreHypens;
    }

    static Regex ssnCaptureRegex =
        new Regex(@"^(?<area>\d{3})-(?<group>\d{2})-(?<serial>\d{4})$");
    static Regex ssnCaptureNoHypensRegex =
        new Regex(@"^(?<area>\d{3})(?<group>\d{2})(?<serial>\d{4})$");
    private bool ignoreHypens;

    protected override string DefaultMessageTemplate
    {
        get { throw new NotImplementedException(); }
    }

    protected override void DoValidate(
        string objectToValidate,
        object currentTarget,
        string key,
        ValidationResults validationResults)
    {
        // validation logic according to
        // http://ssa-custhelp.ssa.gov/cgi
        // -bin/ssa.cfg/php/enduser/std_adp.php?p_faaid=425

        Match match =
            (ignoreHypens ? ssnCaptureNoHypensRegex : ssnCaptureRegex)
                .Match(objectToValidate);
        if (match.Success)
        {
            string area = match.Groups["area"].Value;
            string group = match.Groups["group"].Value;
            string serial = match.Groups["serial"].Value;

            if (area == "666"
                || string.Compare(area, "772", StringComparison.Ordinal) > 0)
            {
                LogValidationResult(
                    validationResults,
                    "Invalid area",
                    currentTarget,
                    key);
            }
        }
    }
}

```

```

    }
    else if (area == "000" || group == "00" || serial == "0000")
    {
        LogValidationResult(
            validationResults,
            "SSN elements cannot be all '0'",
            currentTarget,
            key);
    }
}
else
{
    LogValidationResult(
        validationResults,
        this.ignoreHypens
        ? "Must be 9 digits"
        : "Must match the pattern '###-##-####'",
        currentTarget,
        key);
}
}
}
}

```

Task 3: Implementing the SSN Validator Attribute

The **validator** attribute enables attribute-driven validation specification. These attributes are usually quite simple.

To implement the SSN validator attribute

1. Add a new class named **SSNValidatorAttribute** to the ValidationHOL.CustomValidators project.
2. Add **using** directives to the SSNValidatorAttribute.cs file to make the necessary types available without full name qualification.

```

using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;

```

3. Implement the validator attribute, as shown in the following example. This implementation does not expose the configuration knobs available in the represented validator; however, adding such configuration support is a simple exercise.

```

public class SSNValidatorAttribute : ValidatorAttribute
{
    protected override Validator DoCreateValidator(Type targetType)
    {
        return new SSNValidator(this.Tag);
    }
}

```

Task 4: Using the SSN Validator Instead of the Built-In One

The new **[SSNValidator]** can be used to annotate properties like the **[RegexValidator]** that was used in previous labs. Additionally, attribute-driven validation will be used for the **SSN** property—but only for the **SSN** property—the rest of the validation rules in this lab continue to be specified through configuration.

To use the SSN validator through attributes

1. Add a reference to the ValidationHOL.CustomValidators project in the ValidationHOL.BusinessLogic project. To do this, right-click the **ValidationHOL.BusinessLogic** node in the Solution Explorer, click **Add Reference**, select the **Projects** tab, click **ValidationHOL.CustomValidators**, and then click **OK**.
2. Open the Customer.cs file.
3. Add **using** directives to the Customer.cs file to make the necessary types available without full name qualification.

```
using ValidationHOL.CustomValidators;
```

4. Remove the **[HasSelfValidation]** attribute, the **ssnCaptureRegex** field, and the **ValidateSSN()** self-validation method.
5. Update the validator attribute on the **SSN** property.

```
public class Customer
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    [SSNValidator]
    public string SSN { get; set; }
    public Address Address { get; set; }
}
```

Verification

In this section, you will verify that the custom validator is used when performing the validation operation, including Windows Forms–integrated user input validation.

To validate that the custom SSN validator is used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank, except for the **First Name**, **Last Name**, and the **SSN** text boxes. A valid value for the **First Name** and **Last Name** are required to change focus and enter an SSN value. Use a value for the SSN such as 666-78-9999 and click **Accept** to trigger the integrated user input validation for the **SSN** field.

You can compare the lab solution folder to the contents of the Lab9\After folder.

Lab 10: Using a Custom Validator through Configuration

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will practice using a custom validator in a configuration file when it has no design-time support. This is a low-effort approach—only the validator class and, optionally, the validator attribute class are needed.

When a validator does not provide specialized design-time support, the XML element representing the validator in the configuration file is de-serialized as a "property bag," which contains all the attributes in the element. The validator must then implement a constructor that gets this property bag as its parameter.

Preparation

Continue working on the solution from Lab 9 or open the solution file from Lab10\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Adding the Necessary References to the **ValidationHOL.CustomValidators** Project
- Task 2: Updating the SSN Validator Class to be Used as a Custom Validator Through Configuration
- Task 3: Removing the Attribute Reference to the Validator from the **Customer** Class
- Task 4: Updating the Configuration File to Use the New Validator
- Task 5: Making the **ValidationHOL.CustomValidators** Available When the Application Runs

Task 1: Adding the Necessary References to the ValidationHOL.CustomValidators Project

You must add references to the **System.Configuration** assembly and to the **Microsoft.Practices.EnterpriseLibrary.Common.dll** assembly.

To add the references

1. Add a reference to the **System.Configuration** assembly. To do this, right-click the **ValidationHOL.CustomValidators** node in the Solution Explorer, click **Add Reference**, make sure the **.NET** tab is selected, locate and select the entry with **System.Configuration** as its component name, and then click **OK**.
2. From the Lib folder of the ValidationHOL.BusinessLogic project, add a reference to the **Microsoft.Practices.EnterpriseLibrary.Common.dll** assembly. To do this, right-click the **ValidationHOL.CustomValidators** node in Solution Explorer, click **Add Reference**, click

Browse, navigate to the Lib folder in the HOL main folder, click the **Microsoft.Practices.EnterpriseLibrary.Common.dll** file, and then click **OK**.

Task 2: Updating the SSN Validator Class to be Used As a Custom Validator Through Configuration

Implement a **SSNValidator** class constructor with a single parameter in order to create custom validators through configuration.

To update the class

1. Add **using** directives to the **SSNValidator.cs** file to make the necessary types available without full name qualification.

```
using System.Collections.Specialized;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Validation.Configuration;
```

2. Add the **ConfigurationElementType** attribute to the **SSNValidator** class so it can be used as a custom validator specified in a configuration file.

```
[ConfigurationElementType(typeof(CustomValidatorData))]
```

3. Add a constructor with a single **NameValueCollection** parameter. This constructor is used to create custom validators through configuration; it passes the values for the attributes in the corresponding XML element.

```
public SSNValidator(NameValueCollection attributes)
    : base(string.Empty, string.Empty)
{
    string ignoreHypensValue = attributes.Get("ignoreHypens");

    if (ignoreHypensValue != null)
    {
        this.ignoreHypens = ignoreHypensValue == "true";
    }
}
```

4. Build the solution. This step is necessary so that the validator type can be picked up with the configuration editor.

Task 3: Removing the Attribute Reference to the Validator from the Customer Class

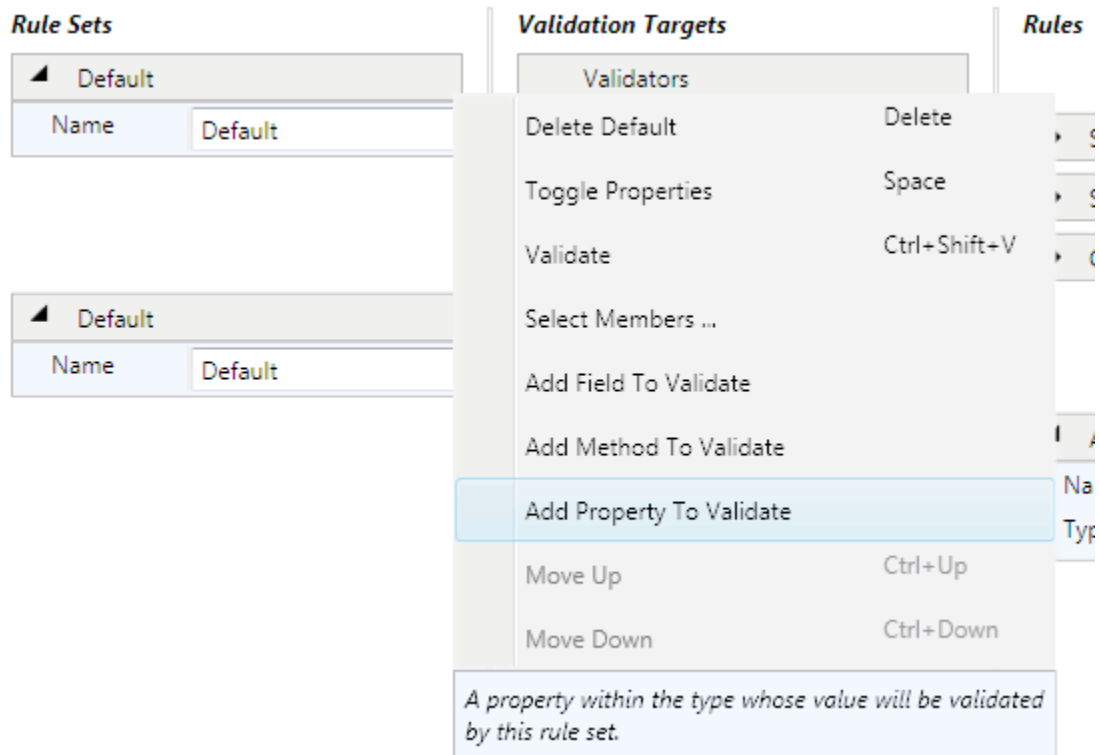
Remove the **[SSNValidator]** attribute for the **SSN** property in the **Customer** class in the **ValidationHOL.BusinessLogic** project.

Task 4: Updating the Configuration File to Use the New Validator

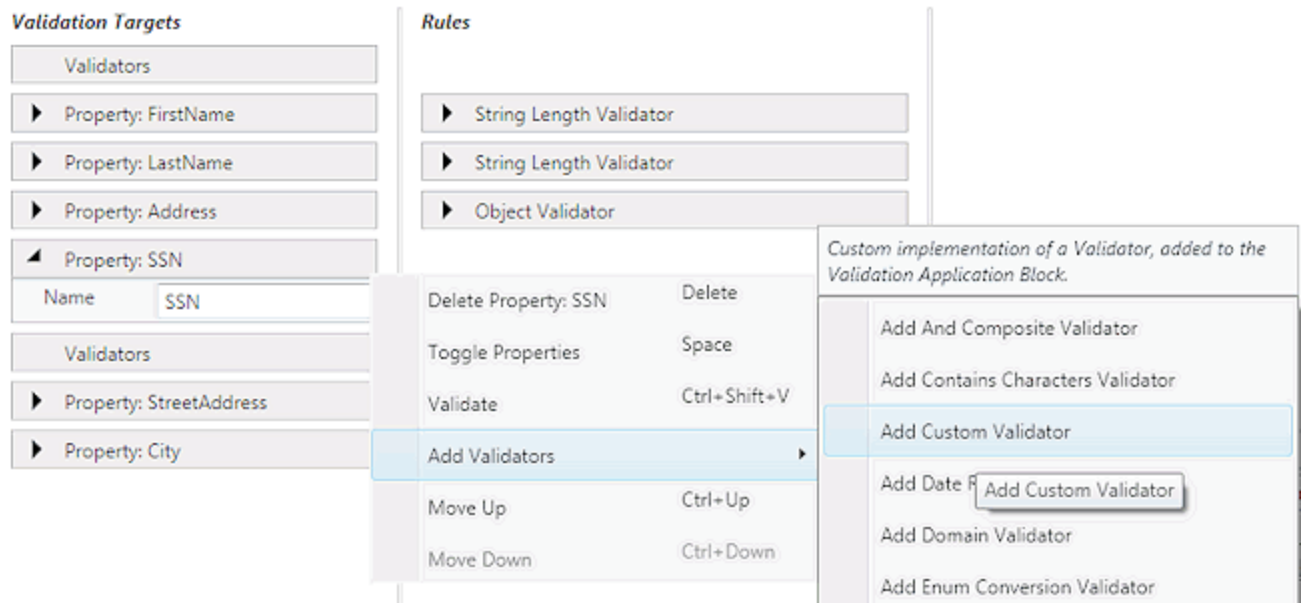
Add the validator information to the configuration file.

To add the required configuration information

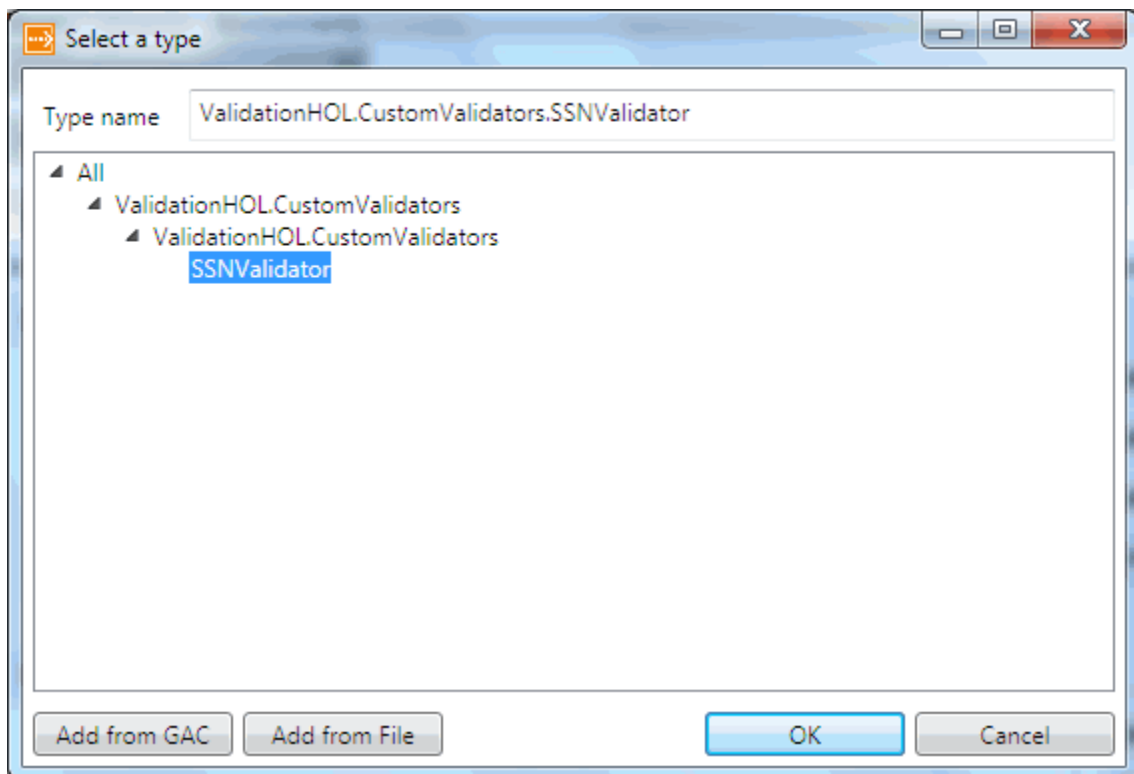
1. Start the configuration editor (EntLibConfig.exe) by selecting the appropriate version for your environment from the Lib folder of the Hands On Labs. Do *not* use the Visual Studio integrated editor.
2. Click **Open...** on the **File** menu, navigate to the folder containing the ValidationHOL project (exercises\Lab10\Before\ValidationHOL), select the App.config file, and then click **OK**.
3. Locate the node for the **Default** rule set for the **Customer** type.
4. Right-click the **Default** node, and click **Add Property To Validate**.



5. Enter **SSN** for the **Name** property on the new node.
6. Add a new custom validator for the **SSN** property. Right-click the **Property: SSN** node, click **Add Validators** and then click **Add Custom Validator**.



7. The type selector dialog box is displayed. Click the **Add from File** button to navigate to the output folder (typically ValidationHOL.CustomValidators \bin\Debug), and load the ValidationHOL.CustomValidators.dll file. Select the **SSNValidator** type from in the types tree, and then click **OK**.



A type will be available if it inherits from the **Validator** class and if it has the proper **[ConfigurationElementType]** attribute.

8. Save the file and close the editor.

Task 5: Making the ValidationHOL.CustomValidators Available When the Application Runs

The custom validator is now referenced only through configuration, so the assembly it belongs to will not be copied by the build process to the application project's output folder. One way to make the assembly available is to add a reference to the ValidationHOL.CustomValidators project in the ValidationHOL project. To do this, right-click the **ValidationHOL** project in Solution Explorer, click **Add Reference**, click **Projects**, click the **ValidationHOL.CustomValidators** entry in the list, and then click **OK**.

Verification

In this section, you will verify that the specified custom validation rules are used when performing the validation operation.

To validate that the custom SSN validator is used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank, except for the **First Name**, **Last Name**, and the **SSN** text boxes. A valid value for the **First Name** and **Last Name** are required to change focus and enter an SSN value. Use a value for the SSN such as 666-78-9999 and click **Accept** to trigger the custom user input validation for the **SSN** field.

You can compare the lab solution folder to the contents of the Lab10\After folder.

Lab 11: Implementing a Custom Validator with Design-Time Support

Estimated time to complete this lab: **10 minutes**

Purpose

In this lab, you will practice providing custom design-time support for a custom validator, instead of relying on the generic, property-bag-supported approach from the previous lab.

Custom design-time support has the following benefits:

- **More complex configuration is possible, and can be validated at deserialization time.** The generic approach saves all *unknown* elements in a property bag of strings, while custom configuration objects will enforce type safety and will validate the configuration at deserialization time.
- **The design-time experience is improved.** Users of the configuration tool do not need to know in advance either the keys that are relevant for a run-time object nor their appropriate values.

The downside of providing custom design-time support is that it takes more effort to implement.

For information about creating custom configuration objects, see [ConfigurationElement Class](#) on MSDN.

Preparation

Continue working on the solution from Lab 10 or open the solution file from Lab11\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Creating a Custom Configuration Object for the Validator
- Task 2: Updating the **SSNValidator** Class to Use the New Configuration Object
- Task 3: Updating the Configuration File to Use the New Design-Time Support

Task 1: Creating a Custom Configuration Object for the Validator

The custom configuration object for the **SSNValidator** will be used to deserialize the XML element representing the validator in a configuration file, and it must declare the relevant configuration properties for the validator.

To create a custom configuration object for the SSN validator

1. In the ValidationHOL.CustomValidators project, create a **Configuration** folder. To do this, right-click the **ValidationHOL.CustomValidators** node in Solution Explorer, point to **Add**, click **New Folder**, type **Configuration** as the new name for the folder, and then press ENTER.
2. In the Configuration folder, create a new class named **SSNValidatorData**. To do this, right-click the **Configuration** node in Solution Explorer, point to **Add**, click **Class**, enter **SSNValidatorData** for the new class name, and then click **Add**.
3. Add **using** directives to the SSNValidatorData.cs file to make the necessary types available without full name qualification.

```
using System.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration.Design;
using ValidationHOL.CustomValidators.Properties;
```

4. Add a resources file named Resources.resx to the **Properties** folder of the ValidationHOL.CustomValidators project. Do not add these resources to the default resources file. Add the following name/value pairs to the new Resources.resx file.

Name	Value
IgnoreHyphensDescription	Ignore hyphens when validating?
IgnoreHyphensName	Ignore Hyphens
SSNValidatorDescription	Validates a SSN
SSNValidatorName	SSN Validator

5. Update the **SSNValidatorData** class to be public and inherit from **ValueValidatorData**. Add the **ResourceDescription** and **ResourceDisplayName** attributes to the class.

```
[ResourceDescription(typeof(Resources), "SSNValidatorDescription")]
[ResourceDisplayName(typeof(Resources), "SSNValidatorName")]
public class SSNValidatorData : ValueValidatorData
{
    ...
}
```

6. Add a Boolean property named **IgnoreHyphens** to support strongly-typed configuration deserialization. Add the **ResourceDescription** and **ResourceDisplayName** attributes to it to specify the name and description that will appear in the configuration tools.

```
[ConfigurationProperty("ignoreHyphens")]
[ResourceDescription(typeof(Resources), "IgnoreHyphensDescription")]
[ResourceDisplayName(typeof(Resources), "IgnoreHyphensName")]
public bool IgnoreHyphens
{
    get
    {
        return (bool)this["ignoreHypens"];
    }
    set
    {
        this["ignoreHypens"] = value;
    }
}
```

The **ResourceDescription** attribute specifies the description for a property that can be loaded from a string assembly resource instead of hardcoded string.

The **ResourceDisplayName** attribute specifies the name for a property that can be loaded from a string assembly resource instead of hardcoded string.

7. Add constructors to the new class. The zero-arguments constructor is required because it is used by the .NET Framework configuration subsystem during deserialization of a configuration file; any other convenience constructor is optional.

```

public SSNValidatorData()
{
}

public SSNValidatorData(string name)
    : base(name, typeof(SSNValidator))
{
}

```

8. Override the inherited virtual **DoCreateValidator** method to create the actual validator during the validator-creation step.

```

protected override Validator DoCreateValidator(Type targetType)
{
    return new SSNValidator(this.Tag, this.IgnoreHypens);
}

```

9. Save the updated file.

Task 2: Updating the SSNValidator Class to Use the New Configuration Object

For the configuration subsystem to de-serialize the XML element for the validator using the new configuration object, the **[ConfigurationElementType]** attribute must be updated.

To update the SSNValidator class

1. Update the **[ConfigurationElementType]** attribute for the **SSNValidator** to use the new **SSNValidatorData** configuration object.

```

[ConfigurationElementType(typeof(SSNValidatorData))]
public class SSNValidator : Validator<string>
{
    ...
}

```

2. Save the updated file.

Task 3: Updating the Configuration File to Use the New Design-Time Support

Use the configuration editor to remove the current validator for the **SSN** property and add a new SSN validator.

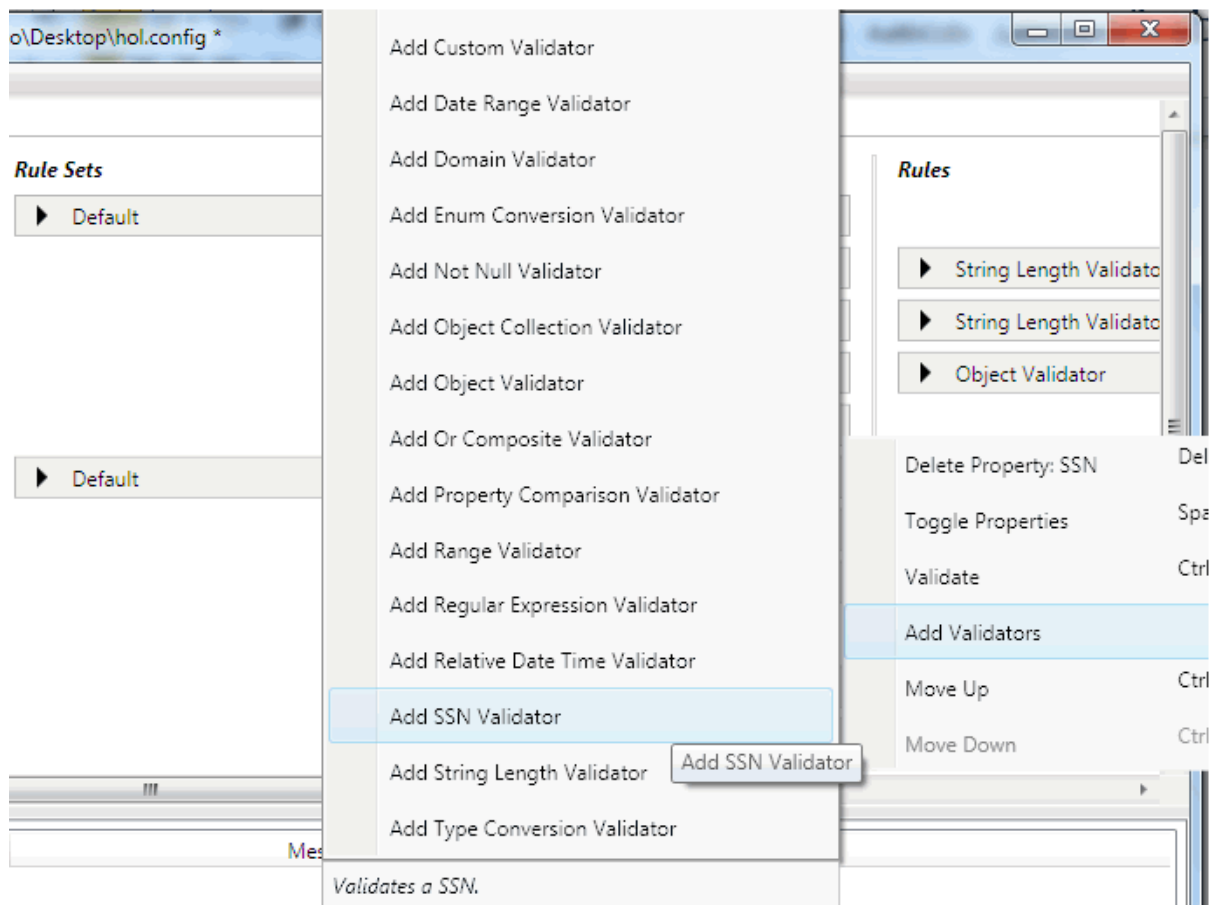
To update the SSNValidator class

1. For the configuration tools to load a custom provider, the provider class must be present in the same folder as the configuration tool. Copy the file ValidationHOL.CustomValidators.dll from the bin\Debug folder of the ValidationHOL.CustomValidators folder into the Lib folder of the Hands On Labs (which contains the Enterprise Library assemblies and the configuration tools).

2. Start the configuration editor (EntLibConfig) by selecting the appropriate version for your environment from the Lib folder of the Hands On Labs.

You must use the standalone configuration tool. The custom design time support will not appear in the Visual Studio integrated editor until you compile it successfully, then close Visual Studio and reopen it.

3. In the ValidationHOL project, open the App.config file.
4. Locate the node for the **SSN** property in the **Default** rule set for the **Customer** type. In the configuration tool use the following path Validation Settings | Validated Types | [Customer] | Rule Sets | [Default] | ValidationTargets | [Property: SSN].
5. Remove the current validator for the **SSN** property (as performed in lab 8).
6. Add a new SSN validator for the **SSN** property. To do this, right-click the **Property: SSN** node, point to **Add Validators**, and then click **SSN Validator**.



7. Save the file and close the editor.

Verification

In this section, you will verify that the specified custom validation rules are used when performing the validation operation. The application's run-time behavior will not change compared to the previous lab.

To validate that the custom SSN validator is used

1. Build and run the ValidationHOL project.
2. In the form, leave the fields blank, except for the **First Name**, **Last Name**, and the **SSN** text boxes. A valid value for the **First Name** and **Last Name** are required to change focus and enter an SSN value. Use a value for the SSN such as 666-78-9999 and click **Accept** to trigger the custom user input validation for the **SSN** field.

You can compare the lab solution folder to the contents of the Lab11\After folder.

Lab 12: Integrating with ASP.NET

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will practice integrating the Validation Application Block into a Web form using the native validation primitives of ASP.NET. For information about this feature, see [Integrating with ASP.NET](#) on MSDN. For an introduction to the ASP.NET validation mechanism, see [BaseValidator Class](#) on MSDN.

The Web application used in this lab is similar to the Windows Forms application used in the previous labs. As in previous labs, information about a customer is entered through input fields and some simple processing is performed on this information if it is considered valid.

Preparation

Open the solution file from Lab12\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

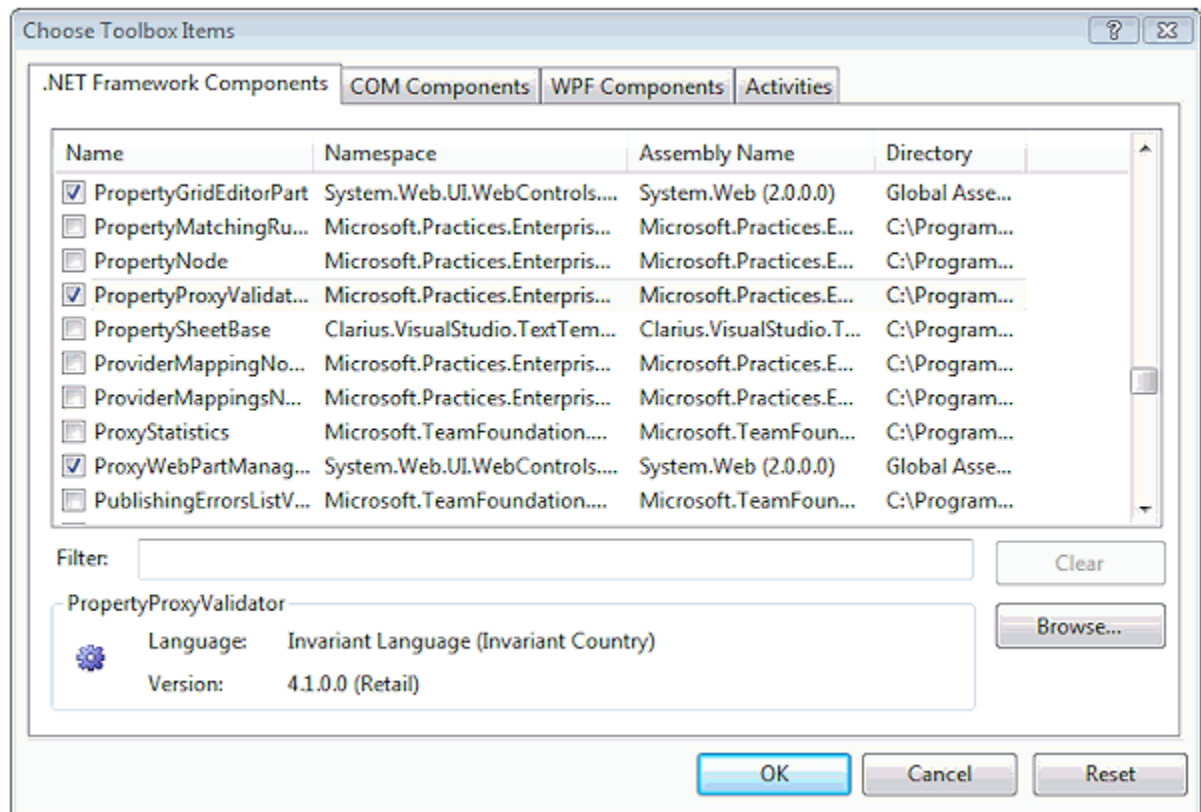
- Task 1: Adding the **PropertyProxyValidator** to the Toolbox
- Task 2: Adding Validators for the Controls in the Default.aspx File

Task 1: Adding the PropertyProxyValidator to the Toolbox

The **PropertyProxyValidator** extends the ASP.NET **BaseValidator** and works as a bridge to the Enterprise Library Validation Application Block.

To add the validator to the Toolbox

1. From the ValidationHOL.Web project in Design mode, open the Default.aspx file.
2. Open the **Toolbox**.
3. Right-click on the tool window surface, and then click **Choose Items**. This operation may take a few minutes.
4. Click **Browse**, navigate to the Lib folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.Integration.AspNet.dll** file, and then click **Add**.
5. Check the entry for **PropertyProxyValidator**, and then click **OK**.



Task 2: Adding Validators for the Controls in the Default.aspx File

For each field that is subject to validation, you must add a **PropertyProxyValidator** that targets the field.

To add the required validators

1. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **First Name** box. In the **Properties** tool window, set the following values:
 - **(ID)** = **FirstNameValidator**
 - **PropertyName** = **FirstName**
 - **SourceTypeName** = **ValidationHOL.BusinessLogic.Customer**,
ValidationHOL.BusinessLogic

- **ControlToValidate = FirstNameTextBox**
2. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **Last Name** box. In the **Properties** tool window, set the following values:
 - **(ID) = LastNameValidator**
 - **PropertyName = LastName**
 - **SourceTypeName = ValidationHOL.BusinessLogic.Customer, ValidationHOL.BusinessLogic**
 - **ControlToValidate = LastNameTextBox**
 3. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **SSN** box. In the **Properties** tool window, set the following values:
 - **(ID) = SSNValidator**
 - **PropertyName = SSN**
 - **SourceTypeName = ValidationHOL.BusinessLogic.Customer, ValidationHOL.BusinessLogic**
 - **ControlToValidate = SSNTextBox**
 4. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **Street Address** box. In the **Properties** tool window, set the following values:
 - **(ID) = StreetAddressValidator**
 - **PropertyName = StreetAddress**
 - **SourceTypeName = ValidationHOL.BusinessLogic.Address, ValidationHOL.BusinessLogic**
 - **ControlToValidate = StreetAddressTextBox**
 5. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **City** box. In the **Properties** tool window, set the following values:
 - **(ID) = CityValidator**
 - **PropertyName = City**
 - **SourceTypeName = ValidationHOL.BusinessLogic.Address, ValidationHOL.BusinessLogic**
 - **ControlToValidate = CityTextBox**
 6. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **State** drop-down list. In the **Properties** tool window, set the following values:
 - **(ID) = StateValidator**
 - **PropertyName = State**

- **SourceTypeName** = ValidationHOL.BusinessLogic.Address, ValidationHOL.BusinessLogic
 - **ControlToValidate** = StateDropDownList
7. Drop a **PropertyProxyValidator** from the **Toolbox** next to the **Zip Code** box. In the **Properties** tool window, set the following values:
- **(ID)** = ZipCodeValidator
 - **PropertyName** = ZipCode
 - **SourceTypeName** = ValidationHOL.BusinessLogic.Address, ValidationHOL.BusinessLogic
 - **ControlToValidate** = ZipCodeTextBox
-

Verification

In this section, you will verify that the custom validation rules specified for the business objects are applied to the fields in the Web form.

To validate that the validation rules are applied

1. Build and run the Web application.
2. Input invalid values to some of the fields in the Web form, and then click **Do Work**. An error message appears next to the fields with invalid data and the page will be considered not valid.

The screenshot shows a Windows Internet Explorer browser window with the address bar displaying `http://localhost:11013/default.aspx`. The page contains a form with the following fields and validation messages:

Field	Value	Validation Message
First Name	First	
Last Name	Last	
SSN	222222222	Must match the pattern '###-##-####'
Street Address		The length of the value must fall within the range "1" (Inclusive) - "50" (Inclusive).
City		The length of the value must fall within the range "1" (Inclusive) - "30" (Inclusive).
State	AL	
Zip Code	888	The value must match the regular expression " <code>^\d{5}\$</code> " with options "None".

At the bottom of the form is a button labeled "Do work". The browser's status bar at the bottom indicates "Internet | Protected Mode: On" and a zoom level of "100%".

You can compare the lab solution folder to the contents of the Lab12\After folder.

Lab 13: Integrating with WCF

Estimated time to complete this lab: **20 minutes**

Purpose

In this lab, you will practice adding a custom Windows Communication Foundation (WCF) behavior that validates the parameters in a Web service call. For information about this feature, see [Integrating with WCF](#) on MSDN.

The Web service in this Lab uses a **Customer** object as one of its parameters, and the Web service client is a modified version of the Web application from Lab 12. The Web application no longer references the **ValidationHOL.BusinessLogic** project; instead, it includes client code generated from the service with the Svcutil.exe tool.

Preparation

Open the solution file from Lab13\Before\ValidationHOL.sln.

Procedures

This lab includes the following tasks:

- Task 1: Updating the Service Interface
- Task 2: Adding the Validation Behavior to the Service Through Configuration
- Task 3: Regenerating the Client-side Types
- Task 4: Updating the Client Code to Handle the Fault Exception

Task 1: Updating the Service Interface

The service interface will be updated for two reasons: to declare the **ValidationFault** as a fault contract and to add validation specifications to one of its operation contract's parameters.

To update the service interface

1. Add references to the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** and **Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF.dll** assemblies (located in the Lib folder) to the ValidationHOL.Service project.
2. Add a **using** directive to the **ICustomerService.cs** file to make the necessary types available without full name qualification.

```
using Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF;  
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

3. Specify the **ValidationFault** for the service operation as shown by the following highlighted code.

```
[OperationContract]  
[FaultContract(typeof(ValidationFault))]  
string ProcessCustomer(...)
```

4. Add validation for the **notes** parameter as shown by the following highlighted code. The **Customer** type itself already has validation definitions, so it is not necessary to annotate it with additional attributes; the **ValidationBehavior** uses validation definitions for the parameter type and any additional validation attributes present on the parameter itself.

```
string ProcessCustomer(  
    Customer customer,  
    [StringLengthValidator(1, 100,  
        MessageTemplate = "The notes must be 1 to 100 characters long.")]  
    string notes);  
}
```

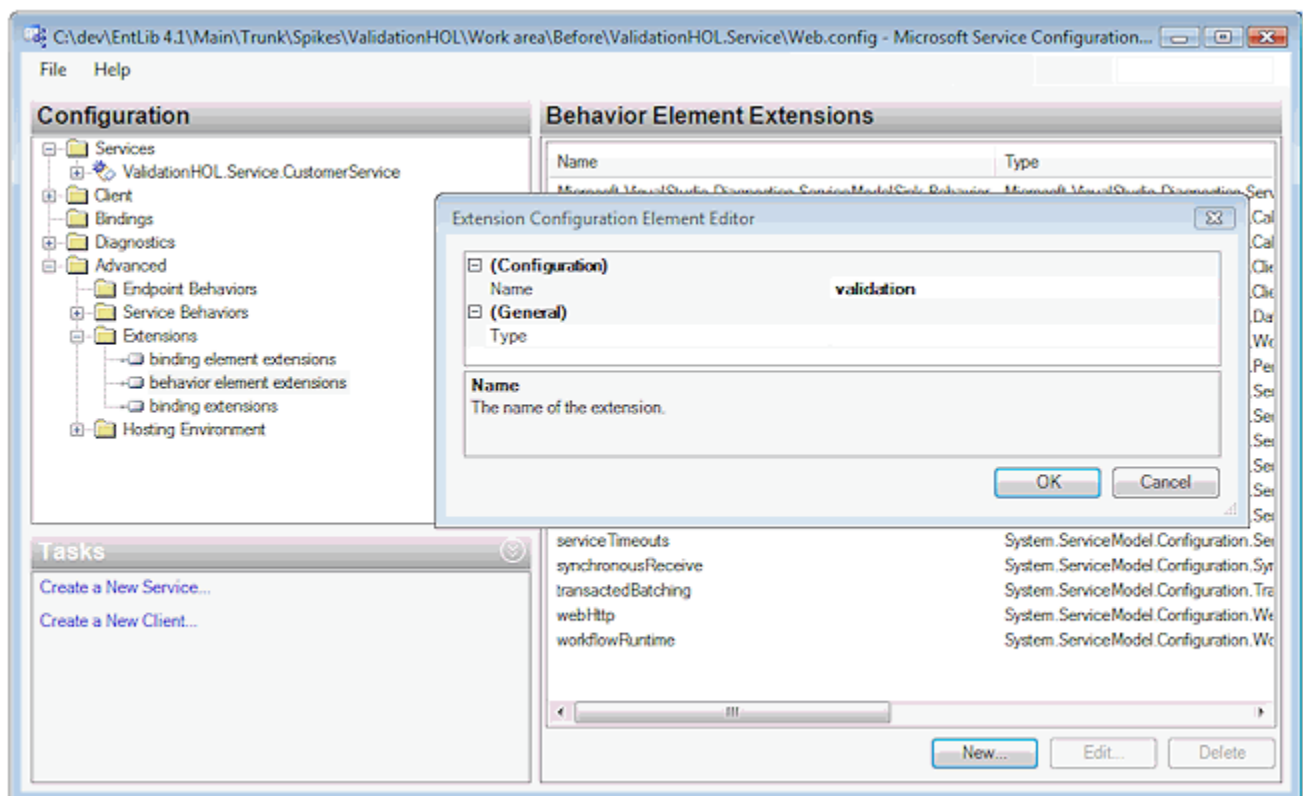
Task 2: Adding the Validation Behavior to the Service Through Configuration

There are several ways to apply the **ValidationBehavior** to a service: you can annotate it with the **[ValidationBehavior]** attribute or specify it through the configuration file. If you do the latter, you can manually edit the configuration file or use the SvcConfigEditor.exe tool. In this lab, you will use the tool.

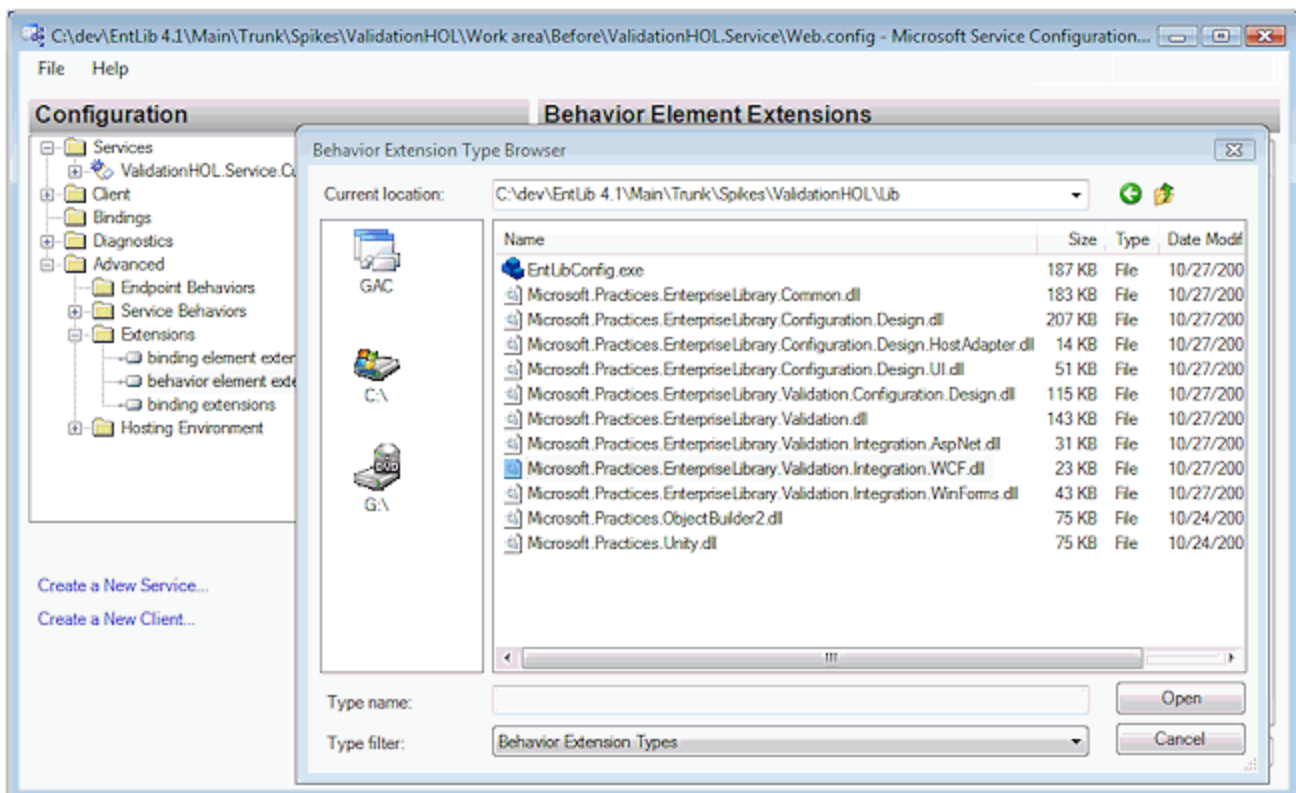
Using the **[ValidationBehavior]** attribute enables finer-grained configuration, which is sometimes necessary: with the attribute, a different rule set can be specified for each service operation, but the configuration file can target only complete endpoints.

To add the validation behavior using the SvcConfigEditor.exe tool

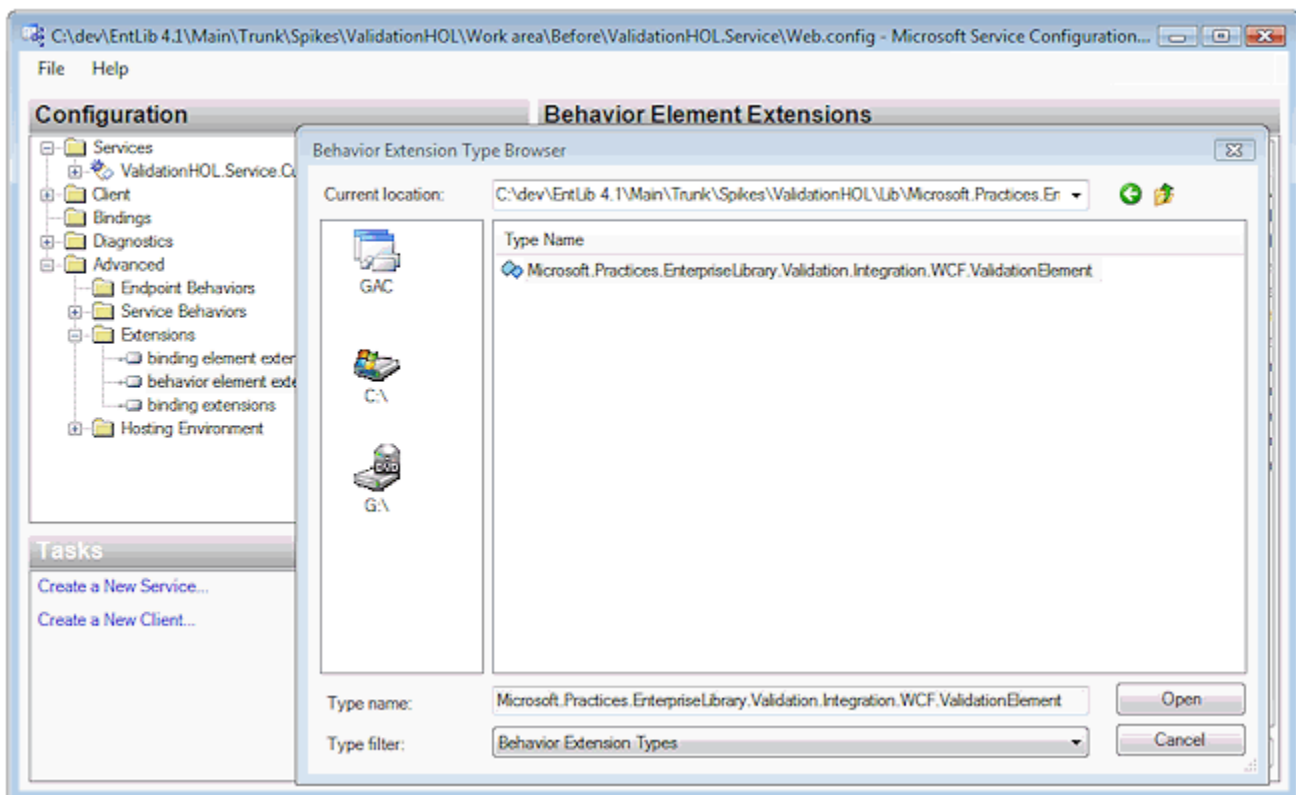
1. Launch the SvcConfigEditor.exe tool. The tool is typically located in the C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin, but you can start the tool from a Visual Studio Command Prompt.
2. Open the Web.config file from the ValidationHOL.Service project.
3. Declare the validation extension. Locate the behavior element extensions node in the **Configuration** tree, under **Advanced\Extensions**. Click the **New** button, set validation as the **Name**.



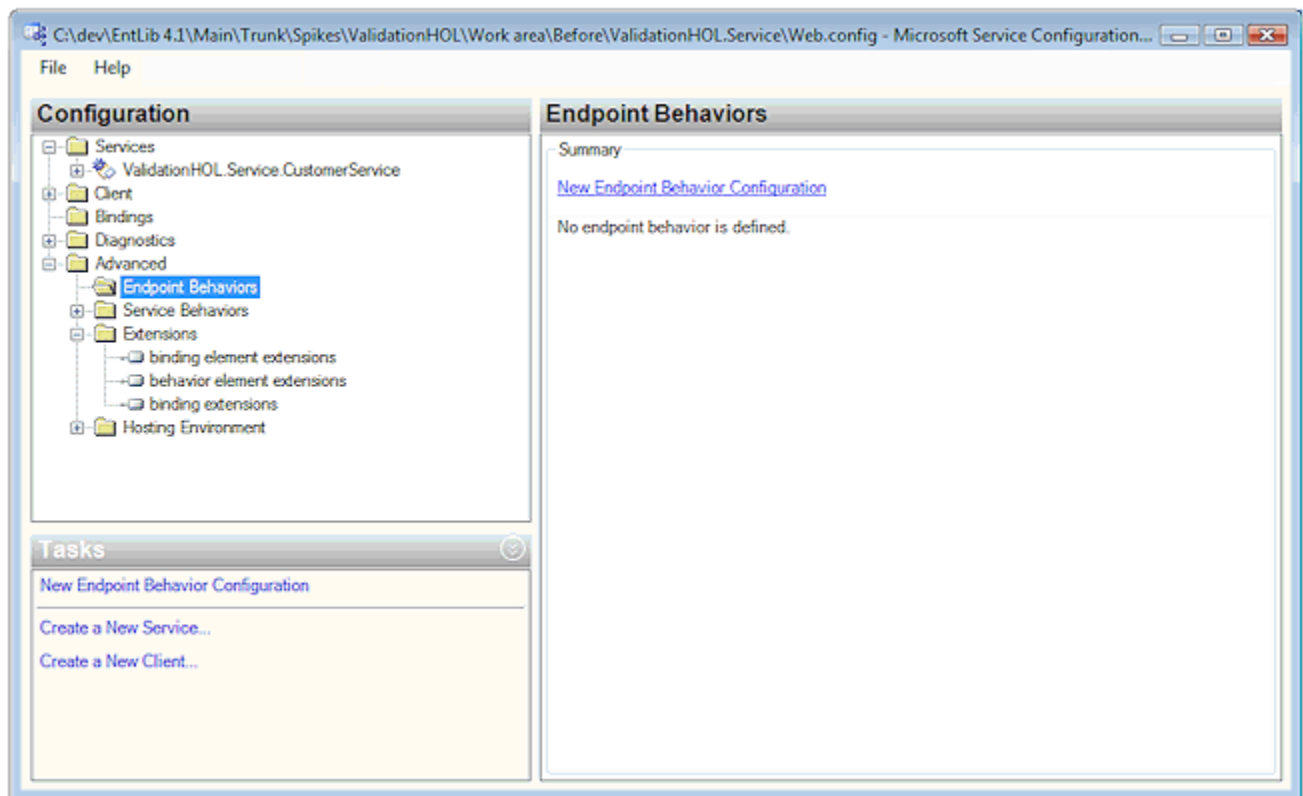
4. Click the **Type** property and click the ellipsis button (...) to launch the type picker. Locate the Lib folder in the browser, select the **Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF.dll** file, and then click **Open**.



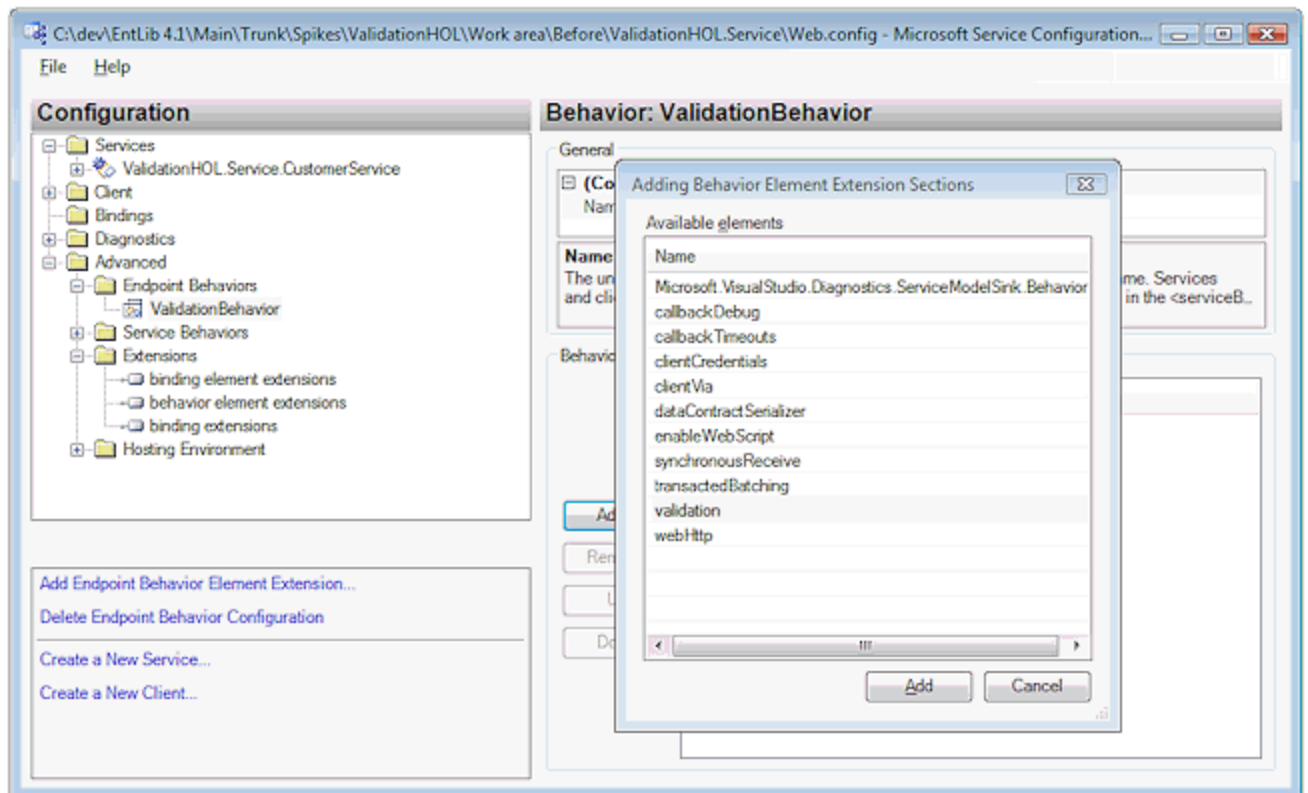
- Click the **Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF.ValidationElement** type, and then click **Open**.



6. Save the file. This is necessary to make the new extension available in the next step.
7. Define the validation endpoint behavior. Locate the **Endpoint Behaviors** node in the **Configuration** tree, and then click **New Endpoint Behavior Configuration**.

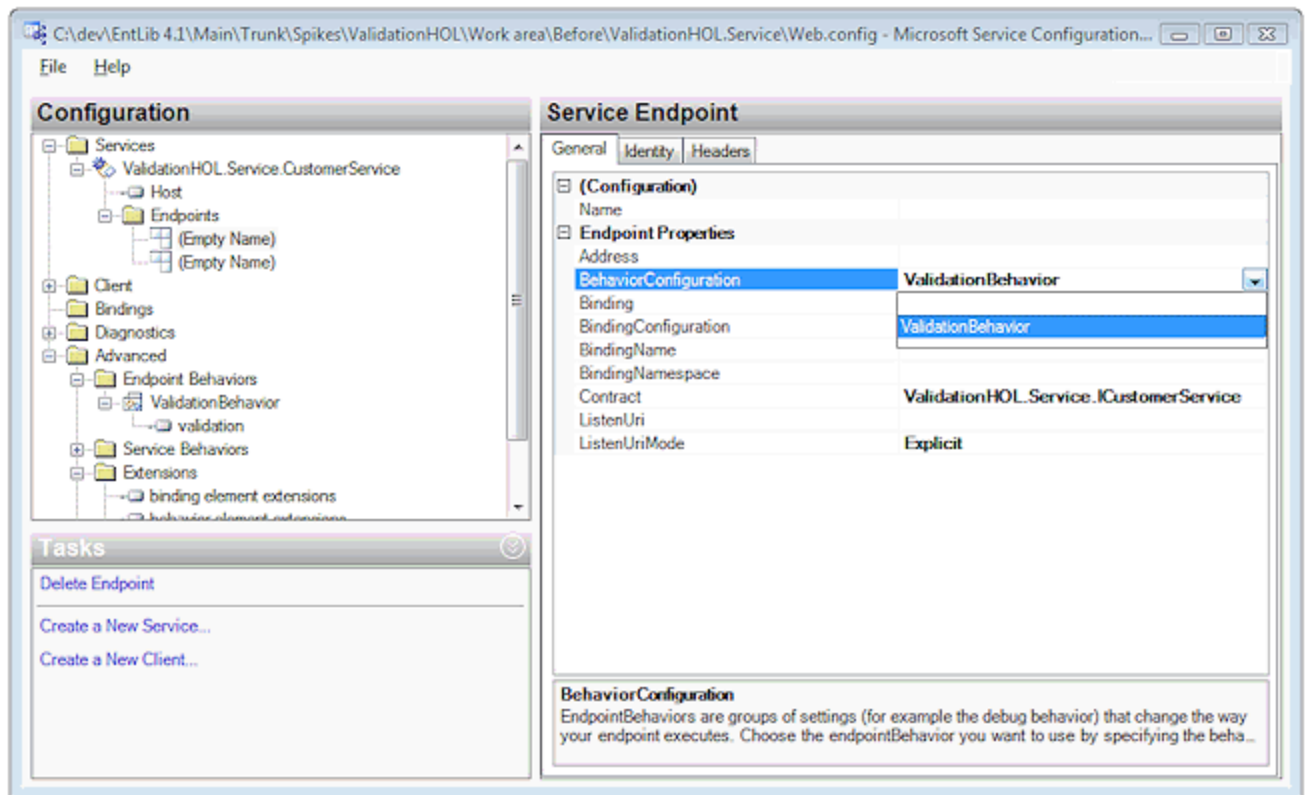


8. Set **ValidationBehavior** as the name for the new entry, and then click **Add**. Select the **validation** item, and then click **Add**.



This behavior extension can also be added to existing behavior configuration elements.

9. Specify the validation behavior for the service. Locate the endpoint for the **ICustomerService** interface—the first **(Empty Name)** entry in the list—and select the **ValidationBehavior** from the **Behavior Configuration** drop-down list box.



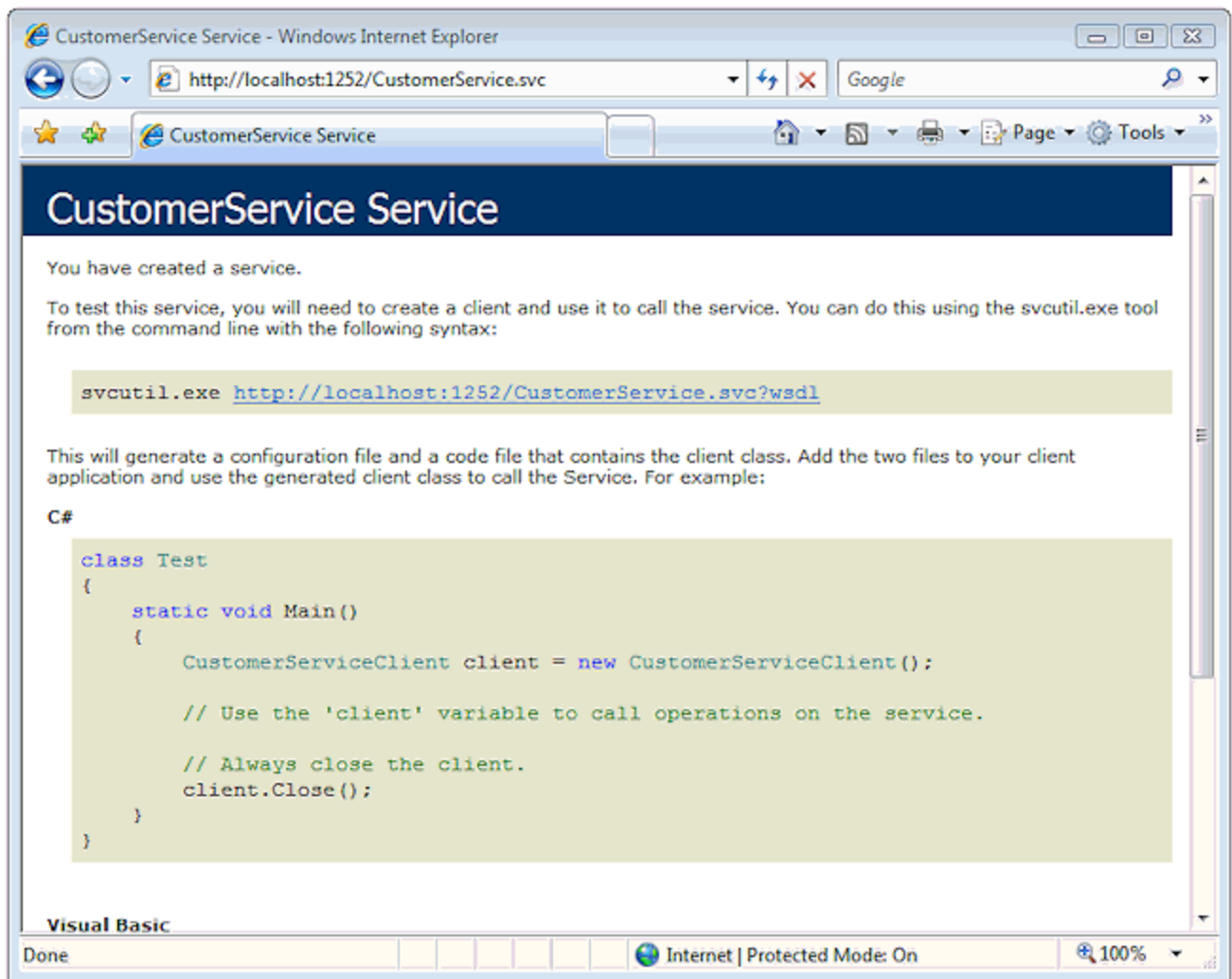
10. Save the file and close the tool.

Task 3: Regenerating the Client-side Types

This step is required because of the change in the **ICustomerService** service interface.

To regenerate the client-side types

1. Start the service. Right-click the **ValidationHOL.Service** project in Solution Explorer, click **Debug**, and then click **Start new instance**. This opens an instance of the default Web browser.
2. Locate the URL for the service's metadata exchange endpoint. In the directory listing shown in the new Web browser instance, click **CustomerService.svc**. The following figure illustrates the customer service test page. On the page, locate a sample invocation to the Svcutil.exe command to generate the client code and record the URL. In the screenshot below, it is <http://localhost:1252/CustomerService.svc?wsdl> but the port number will vary between computers.



3. Execute the **Svcutil.exe** command to generate an up-to-date version of the service client code. Open a Visual Studio Command Prompt window, change the current folder to the folder for the ValidationHOL.Web project, and then invoke the **svcutil.exe** command using the command line saved from the previous step.
4. Close the command window. Then, in Visual Studio, select **Stop Debugging** from the **Debug** menu.

Task 4: Updating the Client Code to Handle the Fault Exception

This change will make explicit use of the **ValidationFault** now available in the updated client code. This fault contains **ValidationDetail** objects, which hold a subset of the information available in a **ValidationResult** object.

To handle the new fault contract

1. Add a **using** directive to the FollowUp.aspx.cs file to make the necessary types available without full name qualification.

```
using System.ServiceModel;
```

```
using System.Text;
using Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF;
using
www.microsoft.com.practices.EnterpriseLibrary._2007._01.wcf.validation;
```

2. Update the **Page_Load** method to explicitly handle the **FaultException** for the **ValidationFault** by adding the following highlighted code.

```
protected void Page_Load(object sender, EventArgs e)
{
    using (CustomerServiceClient client = new CustomerServiceClient())
    {
        try
        {
            this.StatusLabel.Text = client.ProcessCustomer(
                (Customer)Context.Items["customer"],
                (string)Context.Items["notes"]);
        }
        catch (FaultException<ValidationFault> fex)
        {
            StringBuilder builder = new StringBuilder();
            builder.AppendLine("Validation error invoking service:<br/>");
            foreach (ValidationDetail result in fex.Detail.Details)
            {
                builder.AppendLine(
                    string.Format("{0}: {1}<br/>", result.Key, result.Message));
            }
            this.StatusLabel.Text = builder.ToString();
        }
        catch (Exception ex)
        {
            this.StatusLabel.Text = "Unknown error invoking service: "
                + ex.ToString() + "<br/>";
        }
    }
}
```

Verification

In this section, you will verify that a **FaultException** for the **ValidationFault** contract when the supplied information does not satisfy the validation criteria.

To validate that the ValidationFault is received

1. Build and run the application. Launch both the **ValidationHOL.Service** and **ValidationHOL.Web** projects.
2. Locate the Web browser instance for the Web application. In the form, leave the fields blank or add long or invalid values (such as a social security number that does not match the ###-

##-#### format or names longer than 25 characters), and then click **Do work**. The follow-up page will show a list of the validation failures retrieved from the validation fault.

3. Navigate back to the Web application's data entry page, enter valid values into the input fields, and then click **Do work**. A "Processed customer" message should display.

Validation occurs on the service side; there is no integration with the ASP.NET validation mechanism like there was with Lab 12.

You can compare the lab solution folder to the contents of the Lab13\After folder.

Lab 14: WPF Integration

Estimated time to complete this lab: **15 minutes**

Each Lab folder contains a folder named Before and a folder named After. Each Before folder contains source code files as they are before performing the edits and additions for that lab. Each After folder contains the source code edits and additions for that lab.

Purpose

In this lab, you will practice performing validation operations on objects and **using** attributes to specify validation rules for a simple WPF application.

Preparation

Open the solution file from Lab14\Before\ValidationHOL.sln.

The solution consists of the following two projects:

- **ValidationHOL.BusinessLogic**. This project contains one simple domain class, **Customer**
- **ValidationHOL.Wpf**. This project implements a simple WPF application that validates an instance of the domain class **Customer** using the values entered in the input fields and performs some processing.

If you build and run the ValidationHOL.Wpf project without any modifications, type data in the fields empty; no validation will be performed. Because no validation attributes are set at this point, the application simply attempts to process the invalid data.

Procedures

This lab includes the following tasks:

- Task 1: Adding Validation Attributes to the Business Classes
- Task 2: Adding WPF Binding Validation Rules

Validation specified for a type will be invoked by the WPF binding.

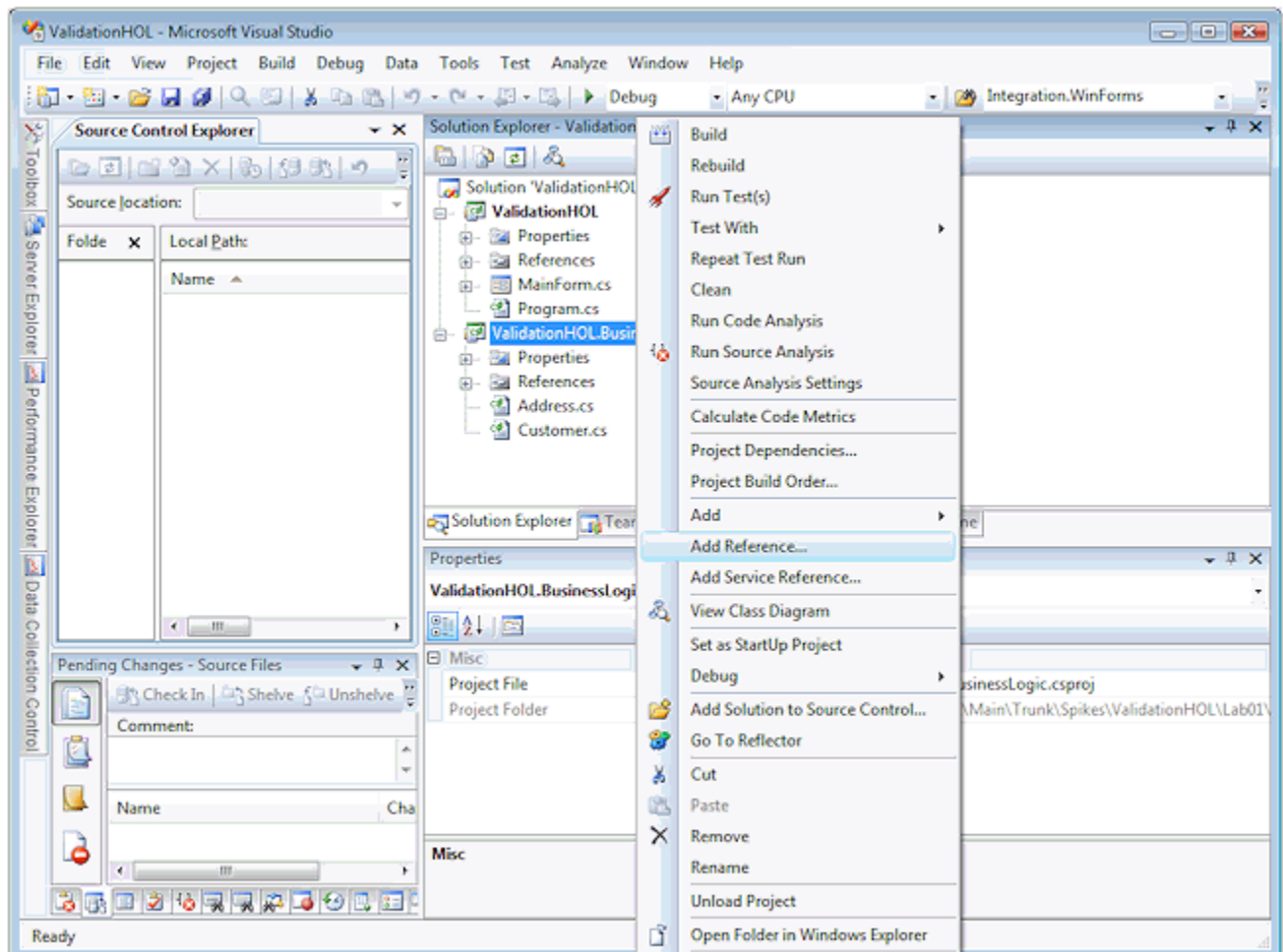
Task 1: Adding Validation Attributes to the Business Classes

Attributes can be used to specify validation rules. These validation attributes can be used with classes and public fields, readable properties, and non-void methods with no arguments. Attributes for each built-in validation rule are available without modification to the application block. For information about the available validators, see the topic "Using the Validation Block Validators" in the online documentation for Enterprise Library at [http://msdn.microsoft.com/en-us/library/ff664694\(v=PandP.50\).aspx](http://msdn.microsoft.com/en-us/library/ff664694(v=PandP.50).aspx).

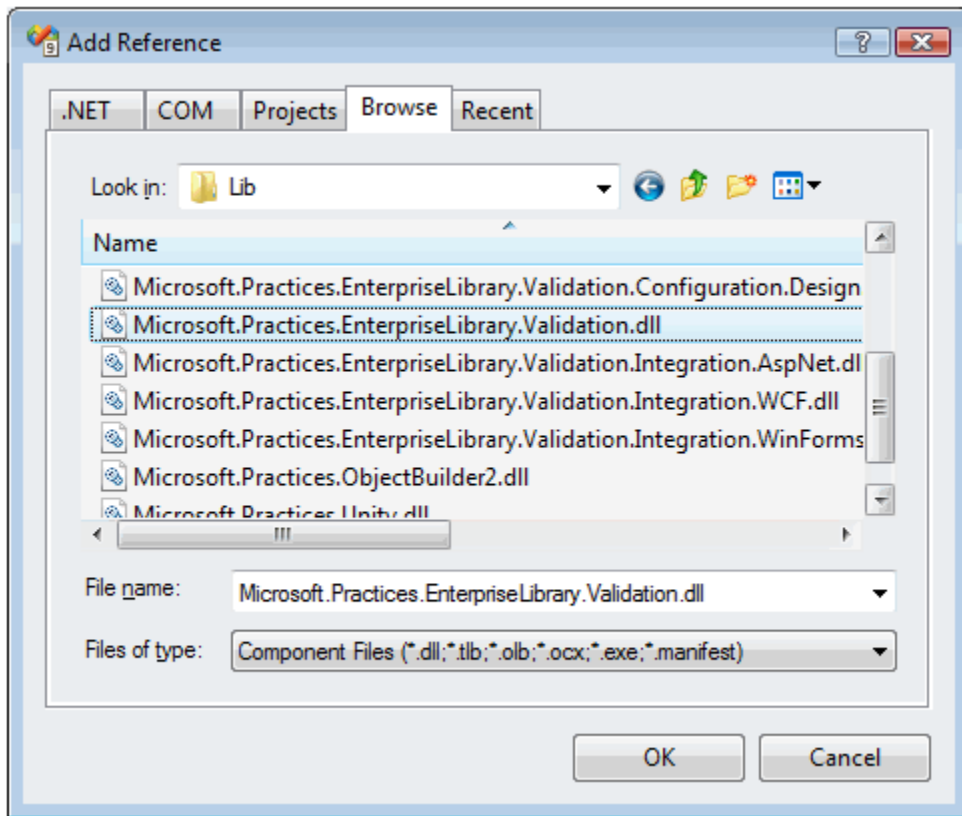
In this lab, you will verify the domain classes' properties for maximum lengths and valid domains. You will be using the Enterprise Library **StringLengthValidator** and **DomainValidator** and the **Required Validator** specified in the **System.ComponentModel.DataAnnotations**.

To add validation attributes

1. Add a reference to the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** assembly from the Lib folder of the examples to the ValidationHOL.BusinessLogic project. To do this, right-click the **ValidationHOL.BusinessLogic** project in Solution Explorer and then click **Add reference**.



2. Click the **Browse** tab, navigate to the Lib folder in the lab's main folder, click the **Microsoft.Practices.EnterpriseLibrary.Validation.dll** file, and then click **OK**.



3. Repeat steps 1 and 2 to add a reference to the assembly **System.ComponentModel.DataAnnotations** to the ValidationHOL.BusinessLogic project.
4. Add **using** directives to the Customer.cs file in the ValidationHOL.BusinessLogic project to make the necessary types available without full name qualification.

```
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;  
using System.ComponentModel.DataAnnotations;
```

5. Add validation attributes to the **Customer** class's properties as shown in the highlighted code below.

```
public class Customer  
{  
    [StringLengthValidator(5,  
        MessageTemplate="Name must be less than 5 characters.")]  
    [Required(ErrorMessage = "Name is required")]  
    public string Name { get; set; }  
  
    [Required(ErrorMessage = "Country is required")]  
    [DomainValidator("ARG", "ITA", "USA",  
        MessageTemplate = "Invalid country")]  
    public string Country { get; set; }  
}
```

Typically, there are several parameters for each validation attribute. For information about each attribute, see the topic "Using the Validation Block Validators" in the online documentation for Enterprise Library at [http://msdn.microsoft.com/en-us/library/ff664694\(v=PandP.50\).aspx](http://msdn.microsoft.com/en-us/library/ff664694(v=PandP.50).aspx).

Task 2: Adding WPF Binding Validation Rules

The WPF data binding model allows you to associate Validation Rules to the binding object. The Validation Rule will check that the value is correct.

Enterprise Library provides a Validation Rule allowing you to integrate the set of validators in the WPF application.

To associate the validation rule

1. Add the following directives to the Window1.xaml file to make the necessary types available without full name qualification. Be sure to open the file in the code view.

```
xmlns:vab="clr-  
namespace:Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WPF  
;assembly=Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WPF  
"
```

2. Add the following settings for the **TextBox.Text** properties of the **Name** and **Country** text boxes in the **Window1.xaml** file to associate the validation rule by adding the following highlighted code.

```
<TextBox x:Name="txtName" Height="20" Width="200" Grid.Column="1"  
Grid.Row="0" TextChanged="validate">  
  <TextBox.Text>  
    <Binding Path="Name" UpdateSourceTrigger="PropertyChanged">  
      <Binding.ValidationRules>  
        <vab:ValidatorRule ValidationSpecificationSource="All"  
          SourceType="{x:Type bl:Customer}"  
          SourcePropertyName="Name"/>  
      </Binding.ValidationRules>  
    </Binding>  
  </TextBox.Text>  
</TextBox>  
  
<TextBox x:Name="txtCountry" Height="20" Width="200" Grid.Column="1"  
Grid.Row="1" TextChanged="validate">  
  <TextBox.Text>  
    <Binding Path="Country" UpdateSourceTrigger="PropertyChanged">  
      <Binding.ValidationRules>  
        <vab:ValidatorRule ValidationSpecificationSource="All"  
          SourceType="{x:Type bl:Customer}"  
          SourcePropertyName="Country"/>  
      </Binding.ValidationRules>  
    </Binding>  
  </TextBox.Text>  
</TextBox>
```

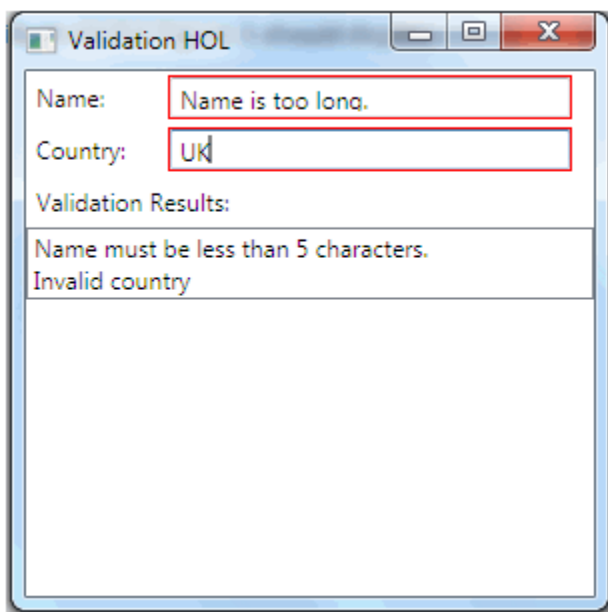
```
</TextBox.Text>  
</TextBox>
```

Verification

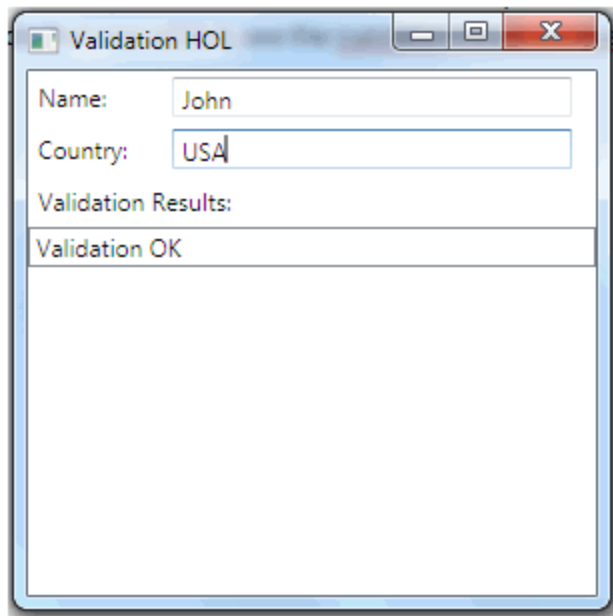
In this section, you will verify that the specified validation rules are actually used when performing the validation operation.

To validate that the validation attributes were properly added

1. Build and run the ValidationHOL.Wpf project.
2. In the form, enter text for the **Name** field with a length greater than 5 characters and for the **Country** field that is not in the following list “ARG”, “USA” or “ITA”.
The error message illustrated below should display.



3. Enter valid values for the fields for the **Name** and **Country** fields. For the **Name**, enter a value such as **John** and for the Country, enter a value such as **USA**.
The information illustrated in the following figure should display.



You can also compare the lab solution folder to the contents of the Lab14\After folder.

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

© 2010 Microsoft. All rights reserved.

Microsoft, MSDN, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.