Integrating Visual C# Code with Dynamic Languages and COM Components

Lab 15: Integrating Visual C# Code with Dynamic Languages and COM Components

Exercise 1: Integrating Code Written by Using a Dynamic Language into a Visual C# Application

Task 1: Examine the Python and Ruby code

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Using Notepad, open the Shuffler.py file in the E:\Labfiles\Lab 15\Python folder:
   a. Using Windows® Explorer, move to the **E:\Labfiles\Lab 15\Python** folder.
   b. Right-click **Shuffler.py**, and then click **Open**.
   c. In the **Windows** dialog box, click **Select a program from a list of installed programs**, and then click **OK**.
   d. In the **Open with** dialog box, click **Notepad**, and then click **OK**.

   In Notepad, examine the Python code.

   The Shuffler.py file contains a Python class called **Shuffler** that provides a method called **Shuffle**. The **Shuffle** method takes a parameter called *data* that contains a collection of items. The **Shuffle** method implements the Fisher-Yates-Durstenfeld algorithm to randomly shuffle the items in the **data** collection.

The Python class also exposes a function called **CreateShuffler** that creates a new instance of the **Shuffler** class. You will use this method from Microsoft Visual C#® to create a **Shuffler** object.

4. Close Notepad.
5. Using Notepad, open the Trapezoid.rb file in the E:\Labfiles\Lab 15\Ruby folder:
   a. Using Windows Explorer, move to the **E:\Labfiles\Lab 15\Ruby** folder.
   b. Right-click **Trapezoid.rb**, and then click **Open**.
   c. In the **Windows** dialog box, click **Select a program from a list of installed programs**, and then click **OK**.
   d. In the **Open with** dialog box, click **Notepad**, and then click **OK**.
6. In Notepad, examine the Ruby code.

   The Trapezoid.rb file contains a Ruby class called **Trapezoid** that models simple trapezoids. The constructor expects the angle of the lower-left vertex, the length of the base, the length of the top, and the height of the trapezoid. The lengths of the remaining sides and angles are calculated.

Note: The **Trapezoid** class models a subset of possible trapezoids. The length of the base must be greater than the length of the top, and the specified vertex must be an acute angle.

The lengths of the sides, the angles of each vertex, and the height are exposed as properties.

The **to_s** method returns a string representation of the trapezoid.

Note: The **to_s** method is the Ruby equivalent of the **ToString** method in the Microsoft .NET Framework. The Ruby binder in the dynamic language runtime (DLR) automatically translates a call to the **ToString** method on a Ruby object to a call to the **to_s** method.

The **area** method calculates the area of the trapezoid.

The Ruby file also provides a function called **CreateTrapezoid** that creates a new instance of the **Trapezoid** class.

7. Close Notepad.

Task 2: Open the starter project

- Open the DynamicLanguageInterop solution in the E:\Labfiles\Lab 15\Starter \DynamicLanguageInterop folder:

  a. On the **File** menu, point to **Open**, and then click Project/Solution.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 15\Starter \DynamicLanguageInterop** folder.

  c. Click **DynamicLanguageInterop.sln**, and then click **Open**.

Task 3: Create a Python object and call Python methods

1. Examine the InteropTestWindow.xaml file:

   - In Solution Explorer, expand the **DynamicLanguageInterop** project, and then double-click the **InteropTestWindow.xaml** file.

   This window contains two tabs, labeled **Python Test** and **Ruby Test**.

   The **Python Test** tab enables you to type values into the **Data** box and specify whether this is text or numeric data. When you click **Shuffle**, the data will be packaged up into an array and passed to the **Shuffle** method of a Python **Shuffler** object. The shuffled data will be displayed in the **Shuffled Data** box.

   The functionality to create the Python object and call the **Shuffle** method has not yet been implemented; you will do this in this task.

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronPython runtime.

Assembly	Path
IronPython	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.dll
IronPython.Modules	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.Modules.dll
Microsoft.Dynamic	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Dynamic.dll
Microsoft.Scripting	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Scripting.dll

a. In Solution Explorer, right-click **References**, and then click **Add Reference**.

b. In the **Add Reference** dialog box, click **Browse**.

c. Move to the **C:\Program Files\IronPython 2.6 for .NET 4.0** folder.

d. Select **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Dynamic.dll**, and **Microsoft.Scripting.dll**, and then click **OK**.

3. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the task list is displaying **User Tasks**, in the drop-down list box click **Comments**.

4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task. This task is located near the top of the InteropTestWindow.xaml.cs file. This is the code behind the InteropTestWindow window.

5. After the comment, add **using** statements to bring the **IronPython.Hosting** and **Microsoft.Scripting.Hosting** namespaces into scope.

   Your code should resemble the following code example.

```
// TODO: Add Namespaces containing IronPython and IronRuby runtime
support and interop types
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
...
```

6. In the **InteropTestWindow** class, examine the string constants near the start of the class. In particular, note the **pythonLibPath** and **pythonCode** strings.

   The **pythonLibPath** constant specifies the folder where the Python libraries are installed. The **Shuffler** class makes use of a Python library called **random** that is located in this folder.

   The **pythonCode** constant specifies the name and location of the Python script that contains the **Shuffler** class.

7. In the task list, locate the **TODO: Create an instance of the Python runtime, and add a reference to the folder holding the "random" module** task, and then double-click this task. This task is located in the **ShuffleData** method.

   The **shuffle_Click** method calls the **ShuffleData** method when the user clicks the **Shuffle Data** button. The **shuffle_Click** method gathers the user input

from the form and parses it into an array of objects. It then passes this array to the **ShuffleData** method. The purpose of the **ShuffleData** method is to create a Python **Shuffler** Python object and then call the **Shuffle** method by using the array as a parameter. When the **ShuffleData** method finishes, the **shuffle_Click** method displays the shuffled data in the Windows Presentation Foundation (WPF) window.

8. After the TODO comment, add code that performs the following tasks:

   a. Create a **ScriptEngine** object called **pythonEngine** by using the static **CreateEngine** method of the **Python** class.

   b. Obtain a reference to the search paths that the Python runtime uses; call the **GetSearchPaths** method of the **pythonEngine** object and store the result in an **ICollection<string>** collection object called **paths**.

   c. Add the path that is specified in the **pythonLibPath** string to the **paths** collection.

   d. Set the search paths that the **pythonEngine** object uses to the **paths** collection; use the **SetSearchPaths** method.

   Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    // TODO: Create an instance of the Python runtime, and add a
reference to the folder holding the "random" module
    // The Python script references this module
    ScriptEngine pythonEngine = Python.CreateEngine();
    ICollection<string> paths = pythonEngine.GetSearchPaths();
    paths.Add(pythonLibPath);
    pythonEngine.SetSearchPaths(paths);
    ...
}
```

9. After the comment **TODO: Run the script and create an instance of the Shuffler class by using the CreateShuffler method in the script**, add code that performs the following tasks:

   a. Create a **dynamic** object called **pythonScript**. Initialize this object with the value that is returned by calling the **ExecuteFile** method of the **pythonEngine** object. Specify the **pythonCode** constant as the parameter to this method.

This statement causes the Python runtime to load the Shuffler.py script. The **pythonScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.

b. Create another **dynamic** object called **pythonShuffler**. Call the **CreateShuffler** method of the **pythonScript** object and store the result in the **pythonShuffler** object.

This statement invokes the **CreateShuffler** function in the Python script. This function creates an instance of the **Shuffler** class and returns it. The **pythonShuffler** object then holds a reference to this object.

Note: The pythonScript variable is a **dynamic** object, so Microsoft IntelliSense® does not display the **CreateShuffler** method (or any other methods or properties).

Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    ...
    // TODO: Run the script and create an instance of the Shuffler
class by using the CreateShuffler method in the script
    dynamic pythonScript = pythonEngine.ExecuteFile(pythonCode);
    dynamic pythonShuffler = pythonScript.CreateShuffler();
    ...
}
```

10. After the comment **TODO: Shuffle the data**, add code that calls the **Shuffle** method of the **pythonShuffler** object. Pass the **data** array as the parameter to the **Shuffle** method.

This statement runs the **Shuffle** method in the Python object. The DLR marshals the **data** array into a Python collection and then invokes the **Shuffle** method. When the method completes, the DLR unmarshals the shuffled collection back into the **data** array.

Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    ...
    // TODO: Shuffle the data.
    pythonShuffler.Shuffle(data);
}
```

11. Build the application and correct any errors:
    - On the **Build** menu, click **Build Solution**.

Task 4: Test the Python code

1. Run the application:
    - On the **Debug** menu, click **Start Without Debugging**.
2. In the Dynamic Language Interop Tests window, on the **Python Test** tab, in the **Data** box, type some random words that are separated by spaces.
3. Click the **Text** option button, and then click **Shuffle**. Verify that the shuffled version of the data appears in the **Shuffled Data** box.
4. Click **Shuffle** again. The data should be shuffled again and appear in a different sequence.
5. Replace the text in the **Data** box with integer values, click **Integer**, and then click **Shuffle**. Verify that the numeric data is shuffled.
6. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

Task 5: Create a Ruby object and call Ruby methods

1. Examine the **Ruby Test** tab in the InteropTestWindow.xaml file.

   The **Ruby Test** tab enables you to specify the dimensions of a trapezoid (the angle of the first vertex, the length of the base, the length of the top, and the height) by using a series of slider controls. When you click the **Visualize** button, the application will create an instance of the Ruby **Trapezoid** class and display a graphical representation in the canvas in the lower part of the window. The dimensions and area of the trapezoid will be displayed in the text block that is to the right.

   The functionality to create the Ruby object and calculate its area and dimensions has not yet been implemented; you will do this in this task.

   a. In Solution Explorer, double-click the **InteropTestWindow.xaml** file.

   b. Click the **Ruby Test** tab.

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronRuby runtime.

Assembly	Path
IronRuby	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.dll
IronRuby.Libraries	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.Libraries.dll

   a. In Solution Explorer, right-click **References**, and then click **Add Reference**.

   b. In the **Add Reference** dialog box, click **Browse**.

   c. Move to the **C:\Program Files\IronRuby 1.0v4\bin** folder.

   d. Select **IronRuby.dll** and **IronRuby.Libraries.dll**, and then click **OK**.

3. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task.

5. Add a **using** statement to bring the **IronRuby** namespace into scope.

   Your code should resemble the following code example.

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
using IronRuby;
...
```

6. In the **InteropTestWindow** class, examine the **rubyCode** string constant near the start of the class.

   The **rubyCode** constant specifies the name and location of the Ruby script that contains the **Trapezoid** class.

7. In the task list, locate the **TODO: Retrieve the values specified by the user. These values are used to create the trapezoid** task, and then double-click this task. This task is located in the **visualize_Click** method. This method is called when the user clicks the **Visualize** button, after the user has specified the data for the trapezoid.

8. After the TODO comment, add code that performs the following tasks:

   a. Create an integer variable called vertexAInDegrees. Initialize this variable with the value of the **vertexA** slider control.

Hint: Use the **Value** property of a slider control to read the value. This value is returned as a **Double** value, so use a cast to convert it to an integer. This cast is safe because the slider controls are configured to return integer values in a small range, so no data will be lost.

   b. Create an integer variable called lengthSideAB. Initialize this variable with the value of the **sideAB** slider control.

   c. Create an integer variable called lengthSideCD. Initialize this variable with the value of the **sideCD** slider control.

   d. Create an integer variable called heightOfTrapezoid. Initialize this variable with the value of the **height** slider control.

   Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        // TODO: Retrieve the values specified by the user. These
values are used to create the trapezoid.
        int vertexAInDegrees = (int)vertexA.Value;
        int lengthSideAB = (int)sideAB.Value;
        int lengthSideCD = (int)sideCD.Value;
        int heightOfTrapezoid = (int)height.Value;
        ...
    }
    ...
}
```

9. After the comment **TODO: Call the CreateTrapezoid method and build a trapezoid object**, add a statement that creates a dynamic variable called trapezoid and initializes it with the value that the **CreateTrapezoid** method returns. Pass the variables vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid as arguments to the **CreateTrapezoid** method.

   You will implement the **CreateTrapezoid** method in a later step. This method will create an instance of the Ruby **Trapezoid** class by using the specified data and return it.

Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Call the CreateTrapezoid method and build a trapezoid
object.
       dynamic trapezoid = CreateTrapezoid(vertexAInDegrees,
           lengthSideAB, lengthSideCD, heightOfTrapezoid);
        ...
    }
    ...
}
```

10. After the comment **TODO: Display the lengths of each side, the internal angles, and the area of the trapezoid**, add a statement that calls the **DisplayStatistics** method. Pass the **trapezoid** object and the **trapezoidStatistics** text block as parameters to this method.

    You will implement the **DisplayStatistics** method in a later step. This method will call the **to_s** and **area** methods of the Ruby **Trapezoid** class and display the results in the **trapezoidStatistics** text block on the right of the **Ruby Test** tab in the WPF window.

    Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Display the lengths of each side, the internal
angles, and the area of the trapezoid.
        DisplayStatistics(trapezoid, this.trapezoidStatistics);
        ...
    }
    ...
}
```

11. After the comment **TODO: Display a graphical representation of the trapezoid**, add a statement that calls the **RenderTrapezoid** method. Pass the **trapezoid** object and the **trapezoidCanvas** canvas control as parameters to this method.

The **RenderTrapezoid** method is already complete. This method queries the properties of the Ruby **Trapezoid** object and uses them to draw a representation of the trapezoid on the canvas in the lower part of the window.

Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Display a graphical representation of the trapezoid.
        RenderTrapezoid(trapezoid, this.trapezoidCanvas);
    }
    ...
}
```

12. In the task list, locate the **TODO: Create an instance of the Ruby runtime** task, and then double-click this task. This task is located in the **CreateTrapezoid** method.

13. At the start of this method, remove the statement that throws the **NotImplementedException** exception. After the comment, add a statement that creates a **ScriptRuntime** object called **rubyRuntime**. Initialize the rubyRuntime variable with the value that the static **CreateRuntime** method of the **Ruby** class returns.

    Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    // TODO: Create an instance of the Ruby runtime.
    ScriptRuntime rubyRuntime = Ruby.CreateRuntime();

    ...
}
```

14. After the comment **TODO: Run the Ruby script that defines the Trapezoid class**, add a statement that creates a **dynamic** object called **rubyScript**. Initialize the rubyScript variable with the value that the **UseFile** method of the **rubyRuntime** object returns. Pass the **rubyCode** constant as the parameter to the **UseFile** method.

    This statement causes the Ruby runtime to load the Trapezoid.rb script. The **rubyScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.

Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Run the Ruby script that defines the Trapezoid class.
    dynamic rubyScript = rubyRuntime.UseFile(rubyCode);
    ...
}
```

15. After the comment **TODO: Call the CreateTrapezoid method in the Ruby script to create a trapezoid object**, add a statement that creates a **dynamic** object called **rubyTrapezoid**. Initialize the rubyTrapezoid variable with the value that the **CreateTrapezoid** method of the **rubyScript** object returns. Pass the vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid variables as parameters to the **CreateTrapezoid** method.

    This statement invokes the **CreateTrapezoid** function in the Ruby script. The DLR marshals the arguments that are specified and passes them as parameters to the **CreateTrapezoid** function. This function creates an instance of the **Trapezoid** class and returns it. The **rubyTrapezoid** object then holds a reference to this object.

Note: The rubyScript variable is a **dynamic** object, so IntelliSense does not display the **CreateTrapezoid** method.

Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Call the CreateTrapezoid method in the Ruby script to
create a trapezoid object.
    dynamic rubyTrapezoid = rubyScript.CreateTrapezoid(
        vertexAInDegrees, lengthSideAB, lengthSideCD,
        heightOfTrapezoid);
    ...
}
```

16. After the comment **TODO: Return the trapezoid object**, add a statement that returns the value in the rubyTrapezoid variable.

    Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Return the trapezoid object.

    return rubyTrapezoid;

}
```

17. In the task list, locate the **TODO: Use a StringBuilder object to construct a string holding the details of the trapezoid** task, and then double-click this task. This task is located in the **DisplayStatistics** method.

18. After the comment, add a statement that creates a new **StringBuilder** object called **builder**.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    // TODO: Use a StringBuilder object to construct a string holding
the details of the trapezoid.

    StringBuilder builder = new StringBuilder();

    ...
}
```

19. After the comment **TODO: Call the to_s method of the trapezoid object to return the details of the trapezoid as a string**, add a statement that calls the **ToString** method of the trapezoid variable and appends the result to the end of the **builder** object.

    The DLR automatically converts the **ToString** method call into a call to the **to_s** method in the Ruby object. The **to_s** method constructs a Ruby string, which is unmarshaled into a .NET Framework string.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...

    // TODO: Call the to_s method of the trapezoid object to return
the details of the trapezoid as a string.
```

```
    // Note: The ToString method invokes to_s.
    builder.Append(trapezoid.ToString());
    ...
}
```

20. After the comment **TODO: Calculate the area of the trapezoid object by using the area method of the trapezoid class**, add code that calls the **area** method of the trapezoid variable, converts the result into a string, and appends this string to the end of the **builder** object.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...
    // TODO: Calculate the area of the trapezoid object by using the
area method of the trapezoid class
    // and append it to the string holding the details of the
trapezoid
    builder.Append(String.Format("\nArea:\t\t{0}",
        trapezoid.area().ToString()));
    ...
}
```

21. After the comment **TODO: Display the details of the trapezoid in the TextBlock control**, add a statement that sets the **Text** property of the **trapezoidStatistics** control to the string that is constructed by the **builder** object.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...
    // TODO: Display the details of the trapezoid in the TextBlock
control
    trapezoidStatistics.Text = builder.ToString();
}
```

22. Build the application and correct any errors:

    - On the **Build** menu, click **Build Solution**.

Task 6: Test the Ruby code

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. In the Dynamic Language Interop Tests window, click the **Ruby Test** tab.
3. Set the **Vertex A** slider to **75**, set the **Length of Base** slider to **200**, set the **Length of Top** slider to **100**, set the **Height** slider to **150**, and then click **Visualize**.

   Verify that a representation of the trapezoid is displayed in the canvas in the lower half of the window and the statistics for the trapezoid appear in the text block that is to the right. The area of the trapezoid should be 22,500.
4. Experiment with different values for the slider controls, and then click **Visualize**. If you specify values that are outside the range for the set of trapezoids that the **Trapezoid** class can model, a message box should be displayed to indicate the problem. This error message is raised by the constructor in the **Trapezoid** class. The DLR catches the error and converts it into a .NET Framework **Exception** object. The **visualize_Click** method caches this exception and displays the error in a message box.
5. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

Exercise 2: Using a COM Component from a Visual C# Application

Task 1: Examine the data files

1. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder, and then verify that this folder contains the following three text files:
   - 298K.txt
   - 318K.txt
   - 338K.txt
2. Using Notepad, open the 298K.txt file:
   - In Windows Explorer, right-click the **298K.txt** file, and then click **Open**.

   This file contains results from the deflection tests for steel girders that were subjected to various pressures at a temperature of 298 Kelvin. The number on

a line by itself at the top of the file is the temperature at which the tests were performed (298). The remaining lines contain pairs of numbers; the numbers in each pair are separated by a comma. These numbers are the pressure applied, which is measured in kiloNewtons (kN), and the deflection of the girder, which is measured in millimeters.

3. Close Notepad.
4. Using Notepad, open the 318K.txt file.

   This file is in the same format as the 298K.txt file. It contains the results of deflection tests that were performed at a temperature of 318 Kelvin. Notice that the final few lines do not contain any deflection data because the test was halted at a force of 1,000 kN.

5. Close Notepad.
6. Using Notepad, open the 338K.txt file.

   This file is similar to the other two. It contains the results of deflection tests that were performed at a temperature of 338 Kelvin. The test was halted at a force of 800 kN.

7. Close Notepad.

Task 2: Open the starter project and examine the StressData type

1. Using Visual Studio, open the GenerateGraph solution in the E:\Labfiles\Lab 15\Starter\GenerateGraph folder:
   a. On the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 15\Starter\GenerateGraph** folder.
   c. Click **GenerateGraph.sln**, and then click **Open**.
2. Open the StressData.cs file:
   - In Solution Explorer, double-click **StressData.cs**.

   The **StressData** type acts as a container for the stress data for a given temperature. It contains the following public properties:

   - **Temperature**. This is a short value that records the temperature of the test.
   - **Data**. This is a **Dictionary** collection that holds the data. The stress value is used as the key into the dictionary, and the item data is the deflection.

The **StressData** class also overrides the **ToString** method, which returns a formatted string that lists the stress test data that is stored in the object.

Task 3: Examine the GraphWindow test harness

1. Open the GraphWindow.xaml file:

   - In Solution Explorer, double-click **GraphWindow.xaml**.

   This window provides a simple test harness for reading the data from the data files and invoking Microsoft Office Excel® to generate a graph by using this data.

   When users click **Get Data**, they are prompted for the data file to load. The file is read into a new **StressData** object, and the contents of the file are displayed in the **TreeView** control that occupies the main part of the window. A user can click **Get Data** multiple times and load multiple files; they will all be read in and displayed. The **StressData** objects are stored in a **List** collection that is held in a private field in the **GraphWindow** class and is called **graphData**. This code has already been written for you.

   When a user clicks **Graph**, the data in the **graphData** collection will be used to generate an Office Excel graph. The information in each **StressData** object will be transferred to an Office Excel worksheet, and a line graph will then be generated to show the stress data for each temperature. A user can quickly examine this graph and spot any trends in the failure of girders.

2. Open the GraphWindow.xaml.cs code file:

   - In Solution Explorer, expand **GraphWindow.xaml**, and then double-click **GraphWindow.xaml.cs**.

3. Locate the **populateFromFile** method.

   This method uses a **StreamReader** object to read and parse the stress data from a file that is specified as a parameter, and it populates a **StressData** object that is also specified as a parameter.

   This method is complete.

4. Locate the **displayData** method.

   This method takes a populated **StressData** object and displays the items in this object in the **TreeView** control in the window.

   This method is also complete.

5. Locate the **getData_Click** method.

This method runs when the user clicks the **Get Data** button. It uses an **OpenFileDialog** object to prompt the user for the name of a data file and then passes the file name together with a new **StressData** object to the **populateFromFile** method. It then adds the populated **StressData** object to the **graphData** collection before it calls the **displayData** method to add the data to the **TreeView** control in the window.

This method is complete.

6. Locate the **generateGraph_Click** method.

   This method runs when the user clicks the **Generate** button. It prompts the user for the name of an Office Excel workbook to create. It will then create this new workbook and copy the data in the **graphData** collection into a worksheet in this workbook before it generates a graph.

   This method is not complete. You will add the missing functionality and complete the **transferDataToExcelSheet** and **generateExcelChart** helper methods that this code will use.

Task 4: Copy data to an Office Excel worksheet

1. Add a reference to the Microsoft Excel 12.0 Object Library to the application. This is the COM object library that implements the Office Excel object model:
   a. In Solution Explorer, right-click **References**, and then click **Add Reference**.
   b. In the **Add Reference** dialog box, click the **COM** tab.
   c. In the list of COM components, scroll down and click **Microsoft Excel 12.0 Object Library**, and then click **OK**.
2. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
3. In the task list, locate the **TODO: Add the Microsoft.Office.Interop.Excel namespace** task, and then double-click this task. This task is located near the top of the GraphWindow.xaml.cs file.
4. Bring the **Microsoft.Office.Interop.Excel** namespace into scope, and give it an alias of **Excel**. This alias helps you to distinguish items in this namespace and

avoid name clashes without having to specify the full namespace in ambiguous object references.

Your code should resemble the following code example.

```
// TODO: Add the Microsoft.Office.Interop.Excel namespace.
using Excel = Microsoft.Office.Interop.Excel;
```

5. Locate the **transferDataToExcelSheet** method.

   The **generateGraph_Click** method will call this method. It takes three parameters:

   - An **Excel.Worksheet** object called **excelWS**. This object is a reference to the Office Excel worksheet that you will copy the data to.
   - An **Excel.Range** object called **dataRange**. This is an output parameter. You will use this object to indicate the area of the worksheet that contains the data after it has been copied.
   - A **List<StressData>** object called **excelData**. This is a collection of **StressData** objects that contain the data that you will copy to the Office Excel worksheet.

   This method returns **true** if it successfully copies the data to the Office Excel worksheet and **false** if an exception occurs.

6. In the **transferDataToExcelSheet** method, after the comment **TODO: Copy the data for the applied stresses to the first column in the worksheet**, add code that performs the following tasks:

   a. Declare an integer variable called rowNum and initialize it to **1**.

   b. Declare an integer variable called colNum and initialize it to **1**.

   c. Set the value of the cell at location rowNum, colNum in the **excelWS** worksheet object to the text "**Applied Stress**".

Hint: You can use the **Cells** property to read and write a cell in an **Excel** worksheet object. This property acts like a two-dimensional array.

   d. Use a **foreach** loop to iterate through the keys in the first **StressData** object in the **excelData** collection.

Hint: Remember that the **StressData** object contains a **Dictionary** property called **Data**, and the key values in this dictionary are the applied stresses for the test (100, 200, 300, up to 1,500 kN). You can use the **Keys** property of a **Dictionary** object to obtain a collection of keys that you can iterate through.

e. In the body of the **foreach** loop, increment the rowNum variable, and store the value of each key found in the cell at location rowNum, colNum in the **excelWS** worksheet object.

Your code should resemble the following code example.

```
private bool transferDataToExcelSheet(Excel.Worksheet excelWS,
    out Excel.Range dataRange, List<StressData> excelData)
{
    try
    {
        // TODO: Copy the data for the applied stresses to the first
column in the worksheet.
        // This should be a list of values: 100, 200, 300, ..., 1500
        // Each set of data in the list in the graphData object uses
the same set of stresses.
        int rowNum = 1;
        int colNum = 1;
        excelWS.Cells[rowNum, colNum] = "Applied Stress";
        foreach (short appliedStress in excelData[0].Data.Keys)
        {
            rowNum++;
            excelWS.Cells[rowNum, colNum] = appliedStress;
        }
        ...
    }
}
```

7. Locate the comment **TODO: Give each column a header that specifies the temperature**.

   This comment is located in a **foreach** loop that iterates over each item in the **excelData** collection. These items are **StressData** objects, and each **StressData** object contains the data for the tests for a given temperature. When complete, the code in this **foreach** loop will copy the data for each **StressData** object to a new column in the **excelWS** worksheet object, and each column will have a header that specifies the temperature.

8. After the comment, add code that performs the following tasks:

   a. Increment the colNum variable so that it refers to the next column in the worksheet.

b. Set the rowNum variable to **1**.

c. Retrieve the temperature from the **deflectionData StressData** object, format it as a string with the letter "K" appended to the end (for Kelvin), and store this string in the cell at location rowNum, colNum in the **excelWS** worksheet object.

Your code should resemble the following code example.

```
foreach (StressData deflectionData in excelData)
{
    // TODO: Give each column a header that specifies the temperature.
    colNum++;
    rowNum = 1;
    excelWS.Cells[rowNum, colNum] = String.Format("{0}K",
        deflectionData.Temperature);
    ...
}
```

9. Locate the comment **TODO: Only copy the deflection value if it is not null**.

   This comment is located in a nested **foreach** loop that iterates over each value in a **StressData** object. Remember that not all stresses have a deflection value. Where this occurs, the data in the **StressData** object is null. The **if** statement detects whether the current deflection value is null.

10. After the comment, in the body of the **if** statement, add code that performs the following tasks:

    a. Increment the rowNum variable so that it refers to the next row in the worksheet.

    b. Copy the value of the deflection variable (that contains the deflection data) into the cell at location rowNum, colNum in the **excelWS** worksheet object.

    Your code should resemble the following code example.

```
// Copy the deflection data to this column in the worksheet
foreach (short? deflection in deflectionData.Data.Values)
{
    // TODO: Only copy the deflection value if it is not null
    if (deflection != null)
    {
        rowNum++;
        excelWS.Cells[rowNum, colNum] = deflection;
    }
}
```

11. Locate the comment **TODO: Specify the range of cells in the spreadsheet containing the data in the dataRange variable**.

    This comment is located after all of the **foreach** loops have completed and all of the data has been copied to the worksheet.

12. After the comment, add a statement that populates the dataRange variable with information about the set of cells that have been filled.

Hint: You can determine the boundaries of the filled area of an Office Excel worksheet by querying the **UsedRange** property. This property returns an **Excel.Range** object.

Your code should resemble the following code example.

```
...
// TODO: Specify the range of cells in the spreadsheet containing the
data in the dataRange variable.
dataRange = excelWS.UsedRange;
...
```

13. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**.

Task 5: Generate an Office Excel graph

1. Locate the **generateExcelChart** method.

   The **generateGraph_Click** method will call this method after the data has been copied to the Office Excel worksheet. It takes three parameters:

   - A **string** object called **fileName**. When the graph has been created, the method will save the Office Excel workbook to a file by using this file name.
   - An **Excel.Workbook** object called **excelWB**. This is a reference to the Office Excel workbook containing the Office Excel worksheet that contains the data to use for the graph.
   - An **Excel.Range** object called **dataRange**. This range specifies the location in the Office Excel worksheet that contains the data to use for the graph.

2. In the **generateExcelChart** method, after the comment **TODO: Generate a line graph based on the data in the dataRange range**, add code that performs the following tasks:

a. Add a new chart object to the Office Excel workbook, and store a reference to this chart object in an Excel.Chart variable called excelChart.

Hint: You can create a new chart by using the **Add** method of the **Charts** property of an Office Excel workbook object. The **Add** method takes no parameters and returns a reference to the chart object.

b. Call the **ChartWizard** method of the chart object to generate the chart. The following table lists the parameters that you should specify.

Parameter name	Value
Title	**"Applied Stress (kN) versus Deflection (mm)"**
Source	**dataRange**
Gallery	**Excel.XlChartType.xlLine**
PlotBy	**Excel.XlRowCol.xlColumns**
CategoryLabels	**1**
SeriesLabels	**1**
ValueTitle	**"Deflection"**
CategoryTitle	**"Applied Stress"**

Your code should resemble the following code example.

```
private static void generateExcelChart(string fileName, Excel.Workbook
    excelWB, Excel.Range dataRange)
{
    // TODO: Generate a line graph based on the data in the dataRange
range.
    Excel.Chart excelChart = excelWB.Charts.Add();
    excelChart.ChartWizard(
        Title: "Applied Stress (kN) versus Deflection (mm)",
        Source: dataRange, Gallery: Excel.XlChartType.xlLine,
        PlotBy: Excel.XlRowCol.xlColumns, CategoryLabels: 1,
        SeriesLabels: 1, ValueTitle: "Deflection",
        CategoryTitle: "Applied Stress");
        ...
}
```

3. After the comment **TODO: Save the Excel workbook**, add a statement that saves the Office Excel workbook by using the value in the *fileName* parameter.

Hint: Use the **SaveAs** method of the Office Excel Workbook object to save a workbook. This method takes a parameter called *Filename* that specifies the name of the file to use.

Your code should resemble the following code example.

```
private static void generateExcelChart(string fileName, Excel.Workbook
    excelWB, Excel.Range dataRange)
{
    ...
    // TODO: Save the Excel workbook
    excelWB.SaveAs(Filename: fileName);
}
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 6: Complete the test harness

1. Return to the **generateGraph_Click** method.
2. After the comment **TODO: If the user specifies a valid file name, start Excel and create a new workbook and worksheet to hold the data**, add code to perform the following tasks:
   a. Create a new **Excel.Application** object called **excelApp**.
   b. Make the application visible on the user's desktop by setting the **Visible** property of the **excelApp** object to **true**. (By default, Office Excel will run in the background.)
   c. Set the **AlertBeforeOverwriting** property of the **excelApp** object to **false**. This ensures that the **SaveAs** method always saves the workbook.
   d. Set the **DisplayAlerts** property of the **excelApp** object to **false**.
   e. Create a new workbook, and store a reference to this workbook in an **Excel.Workbook** variable called excelWB.

Hint: You can create a new workbook by using the **Add** method of the **Workbooks** property of an **Excel.Application** object. This method takes no parameters and returns a reference to the new workbook.

f. Create a variable called excelWS of type **Excel.Worksheet** and set it as the active worksheet in the new workbook.

Hint: You can obtain a reference to the active worksheet in an Office Excel workbook by using the **ActiveSheet** property.

You code should resemble the following code example.

```
if (saveDialog.ShowDialog().Value)
{
    // TODO: If the user specifies a valid file name, start Excel
    // and create a new workbook and worksheet to hold the data

    excelApp = new Excel.Application();
    excelApp.Visible = true;
    excelApp.AlertBeforeOverwriting = false;
    excelApp.DisplayAlerts = false;
    Excel.Workbook excelWB = excelApp.Workbooks.Add();
    Excel.Worksheet excelWS = excelWB.ActiveSheet;

    ...
}
```

3. After the comment **TODO: Copy the data from the graphData variable to the new worksheet and generate a graph**, add code to perform the following tasks:

   a. Create an **Excel.Range** object called **dataRange** and initialize it to null.

   b. Call the **transferDataToExcelSheet** method, and pass the **excelWS** object the **dataRange** object, and the graphData variable as parameters. Note that the **dataRange** object should be an output parameter.

   c. If the value that the **transferDataToExcelSheet** method returns is **true**, call the **generateExcelChart** method. Pass the **FileName** property of the **SaveDialog** object, the **excelWB** object, and the **dataRange** object as parameters.

   Your code should resemble the following code example.

```
if (saveDialog.ShowDialog().Value)
{
    ...
    // TODO: Copy the data from the graphData variable to the new
Worksheet and generate a graph
    // The dataRange variable specifies the cells on the worksheet
that hold the data
    Excel.Range dataRange = null;
    if (transferDataToExcelSheet(excelWS, out dataRange,
                                 this.graphData))
    {
        generateExcelChart(saveDialog.FileName, excelWB, dataRange);
    }
}
```

4. At the end of the **generateGraph_Click** method, in the **finally** block, after the comment **TODO: Close Excel and release any resources**, add code to check whether the excelApp variable is null; if it is not, close the Office Excel application.

Hint: Use the **Quit** method of an **Excel.Application** object to close Office Excel.

Your code should resemble the following code example.

```
finally
{
    // TODO: Close Excel and release any resources
    if (excelApp != null)
    {
        excelApp.Quit();
    }
}
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 7: Test the application

1. Start the application in Debug mode:
   - On the **Debug** menu, click **Start Debugging**.
2. In the Graphing Data window, click **Get Data**.

3. In the **Graph Data** dialog box, click the **298K.txt** file, and then click **Open**.
4. In the Graphing Data window, in the tree view, expand the **Temperature: 298K** node. Verify that the data has been correctly loaded.
5. Repeat steps 2, 3, and 4 and load the data in the 318K.txt and 338K.txt files. Verify that the tree view lists the data from all three files.

Note: The **displayData** method displays the value –1 for any missing deflection data.

6. Click **Graph**.
7. In the **Graph Data** dialog box, accept the default file name, StressData.xlsx, for the name of the Office Excel workbook to be generated, and then click **Save**.

   You will see Office Excel start to run and your data copied across to a new worksheet. You will also briefly see the graph that is generated before the workbook is saved and Office Excel closes.
8. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder. Verify that this folder contains the Office Excel workbook StressData.xlsx.
9. Double-click the **StressData.xlsx** file to start Office Excel and open the workbook.

   The workbook should contain a chart that displays the stress test results by using the data and settings that you specified.
10. In Office Excel, click the **Sheet1** tab.

    This is the worksheet that your code generated. The first column contains the applied stress values, and the remaining three columns contain the deflections recorded at each of the three temperatures.
11. Close Office Excel.
12. Close the Graphing Data window.
13. Close Visual Studio:
    - On the **File** menu, click **Exit**.