

FME Objects Tutorial v7

Safe Software Inc. makes no warranty either expressed or implied, including, but not limited to, any implied warranties of merchantability or fitness for a particular purpose regarding these materials, and makes such materials available solely on an "as-is" basis.

In no event shall Safe Software Inc. be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of purchase or use of these materials. The sole and exclusive liability of Safe Software Inc., regardless of the form or action, shall not exceed the purchase price of the materials described herein.

This manual describes the functionality and use of the software at the time of publication. The software described herein and the descriptions themselves, are subject to change without notice.

Copyright

© 2000–2007 Safe Software Inc. All rights are reserved.

Revisions

Every effort has been made to ensure the accuracy of this document. Safe Software Inc. regrets any errors and omissions that may occur and would appreciate being informed of any errors found. Safe Software Inc. will correct any such errors and omissions in a subsequent version, as feasible. Please contact us at:

Safe Software Inc.

Fax: 604-501-9965

docs@safe.com

www.safe.com

Safe Software Inc. assumes no responsibility for any errors in this document or their consequences, and reserves the right to make improvements and changes to this document without notice.

Trademarks

FME is a registered trademark of Safe Software Inc. All brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Document Information

Document Name: FME Objects Tutorial

Version: July 2007

Contents

Introduction.....	7
Getting Started	7
Setting up the Visual Studio 2005 Workspace.....	9
Adding a Reference to FMEObjectsDotNet.dll	10
Chapter 1	13
Working with IFMEOSession	13
In this chapter	13
Objective	13
Setting up the Application GUI	13
Testing your changes	17
Adding an Exit Handler to the Application	18
Testing your changes	18
Creating an Instance of IFMEOSession	18
Testing your changes	20
Retrieving the IFMEOGGeometryTools	20
Testing your changes	21
Turning on FME Objects Logging.....	21
Testing your changes	22
Using IFMEODialog to Display About Dialog.....	22
Testing your changes	24
Chapter 2	26
Reading Features From a Source Dataset.....	26
In this chapter	26
Objective	26
Using IFMEODialog to Specify Source Dataset	26
Testing your changes	28
Reading Data Features Through IFMEOREader	28
Testing your changes	30
Logging Features to the Log File	30
Testing your changes	30

Chapter 3	32
Grouping Features by Feature Type.....	32
In this chapter	32
Objective	32
Placing a ListView on the Feature Type Tab	32
Testing your changes	35
Using the FeatureType property of IFMEOFeature	35
Testing your changes	38
Displaying the Results Inside ListView	39
Testing your changes	41
Disabling and Enabling the TabControl	41
Testing your changes	42
Chapter 4	43
Grouping Features by Geometry.....	43
In this chapter	43
Objective	43
Placing a ListView on the Geometry Tab	44
Testing your changes	45
Using the GeometryClass Property of IFMEOGeometry	45
Testing your changes	47
Displaying the Results Inside ListView	48
Testing your changes	49
Chapter 5	51
Retrieving Coordinate System Information	51
In this chapter	51
Objective	51
Placing a ListView and ComboBox on the Coord Sys Tab	51
Testing your changes	53
Populating the Entries of the Feature Type ComboBox	53
Testing your changes	55
Getting the Parameters of a Coordinate System.....	56
Testing your changes	58
Chapter 6	59
Retrieving Format Information	59

In this chapter	59
Objective	59
Placing a ListView on the Format Info Tab.....	59
Testing your changes	61
Getting the Parameters of a Format	61
Testing your changes	62
Chapter 7	63
Retrieving Schema Information	63
In this chapter	63
Objective	63
Placing a ListView and Schema Feature ComboBox on the Schema Info Tab	63
Testing your changes	65
Reading in Schema Features from a Dataset	65
Testing your changes	67
Populating the Entries of the Schema Feature ComboBox.....	68
Testing your changes	69
Displaying the Metadata of Schema Features.....	70
Testing your changes	71
Chapter 8	72
Saving Features to a Different Format.....	72
In this chapter	72
Objective	72
Using IFMEODialog to Specify the Destination Dataset	72
Testing your changes	76
Chapter 9	77
Rendering Features	77
In this chapter	77
Objective	77
Updating the Main Form to include a Rendering Panel.....	77
Testing your changes	78
Manipulating Geometry Coordinates.....	79
Testing your changes	81
Creating the RenderingVisitor Class	81
Using the Visitor Design Pattern to Facilitate Rendering of Feature Geometries.....	84

Testing your changes	90
Displaying Feature Geometries on the Rendering Panel	90
Testing your changes	94
Chapter 10	96
Advanced Topics	96
In this chapter	96
Objective	96
Using Separate Threads for Reading/Writing Datasets	96
Testing your changes	99
Implement Column Click Sorting on ListView Components	100
Testing your changes	105

Introduction

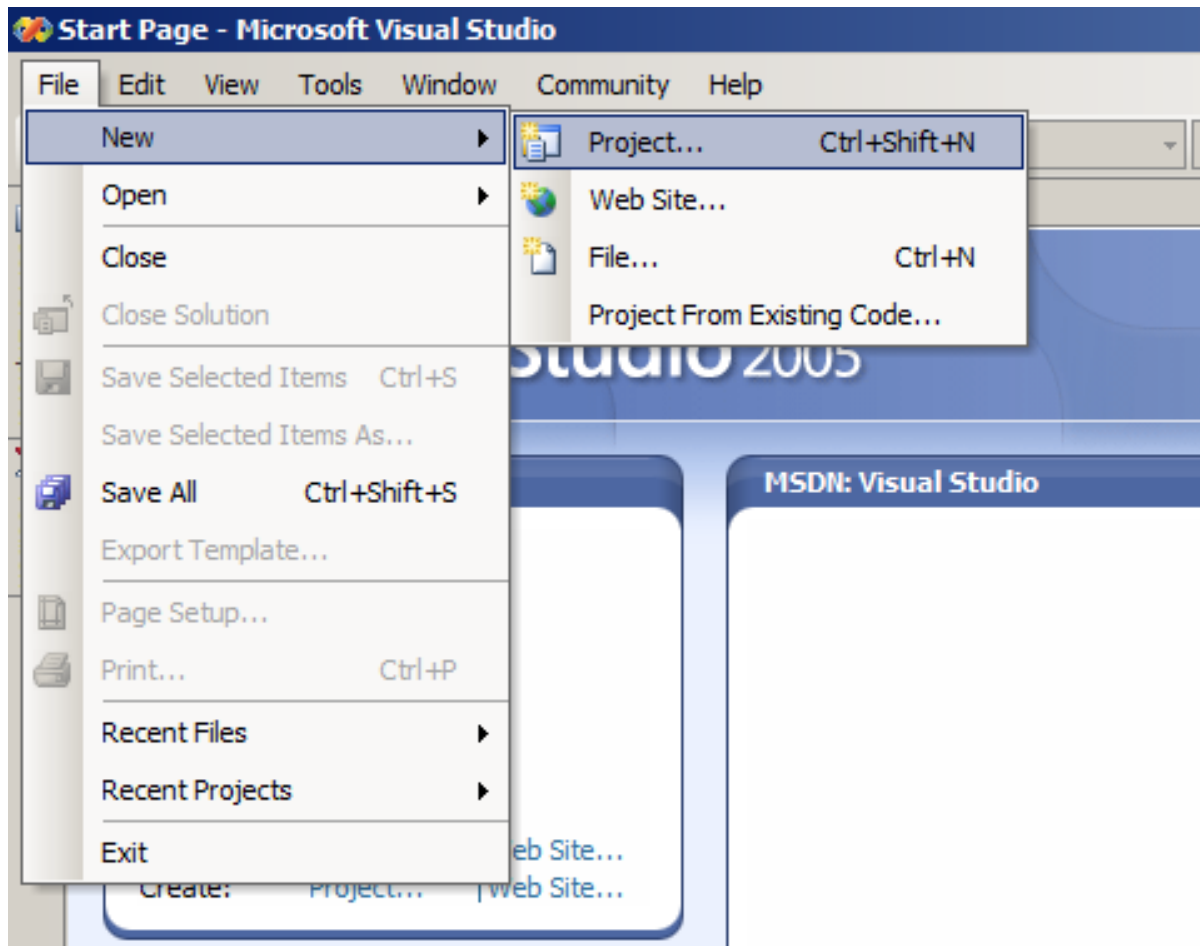
In this tutorial, you will use the C# language and MS Visual Studio 2005 to build a sample application that demonstrates the different usage of FME Objects. A strong background in C# is not required, but the tutorial assumes that the reader has some programming experience in either C++ or Java. Over the course of this tutorial, you will be working within two main views of MS Visual Studio 2005. You will use the Designer view to layout the GUI components, and the Code view for writing the source code that implement the functionality of the sample application.

This tutorial uses an accumulative approach to extend the same application from chapter to chapter (the work done is dependent on previous chapters). Therefore, each chapter can be seen as a checkpoint in the application. For the best learning experience, you are encouraged to use your own solution and build it from chapter to chapter. However, full source code solutions are available for each chapter. Hence, if you would like to start off a new chapter with a known working solution from the previous chapter, you have the ability to do so.

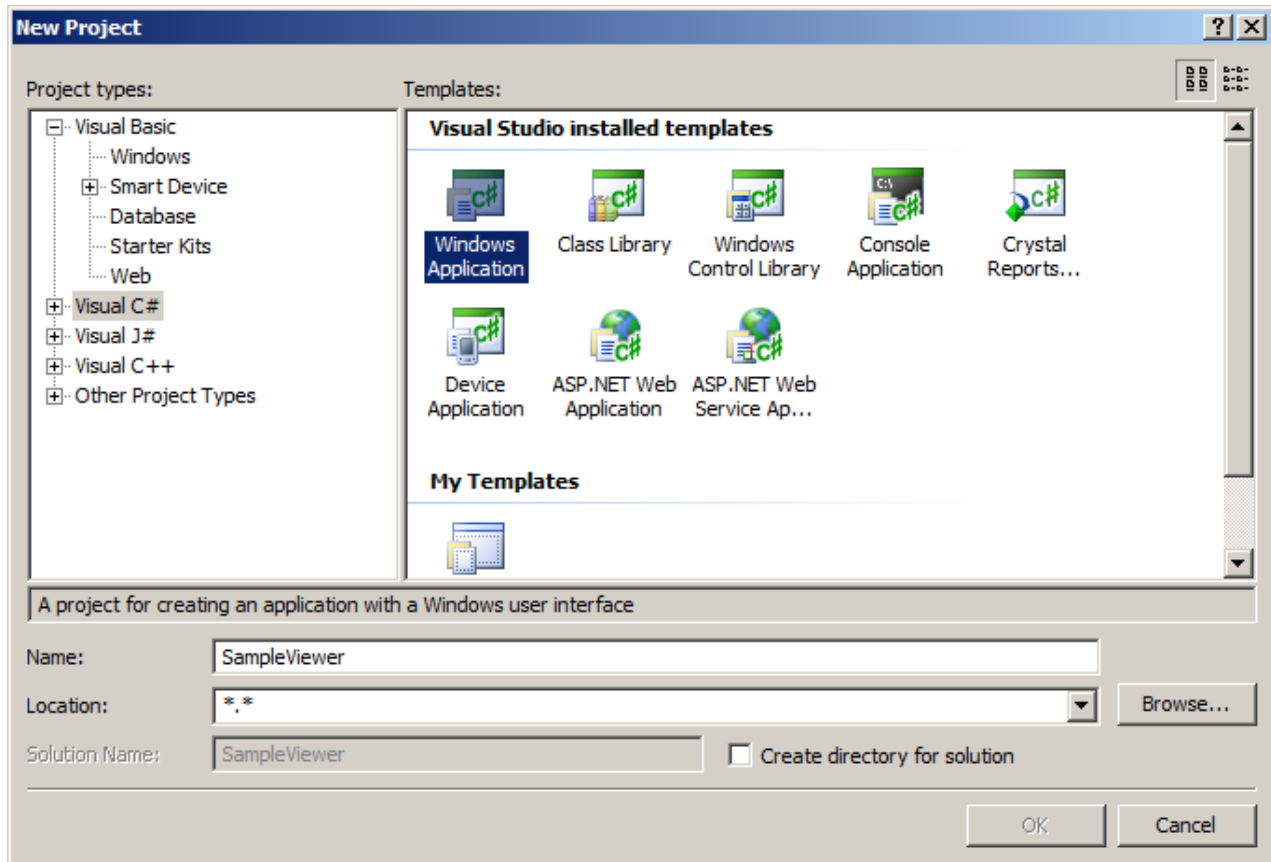
Getting Started

To begin, start up MS Visual Studio 2005.

1. Create a new project by selecting Project from the File > New menu.



2. In the New Project dialog that appears, select C# Windows Application as the project type.
3. Enter "SampleViewer" as the name, and save the project to a convenient location of your choice. Click OK.

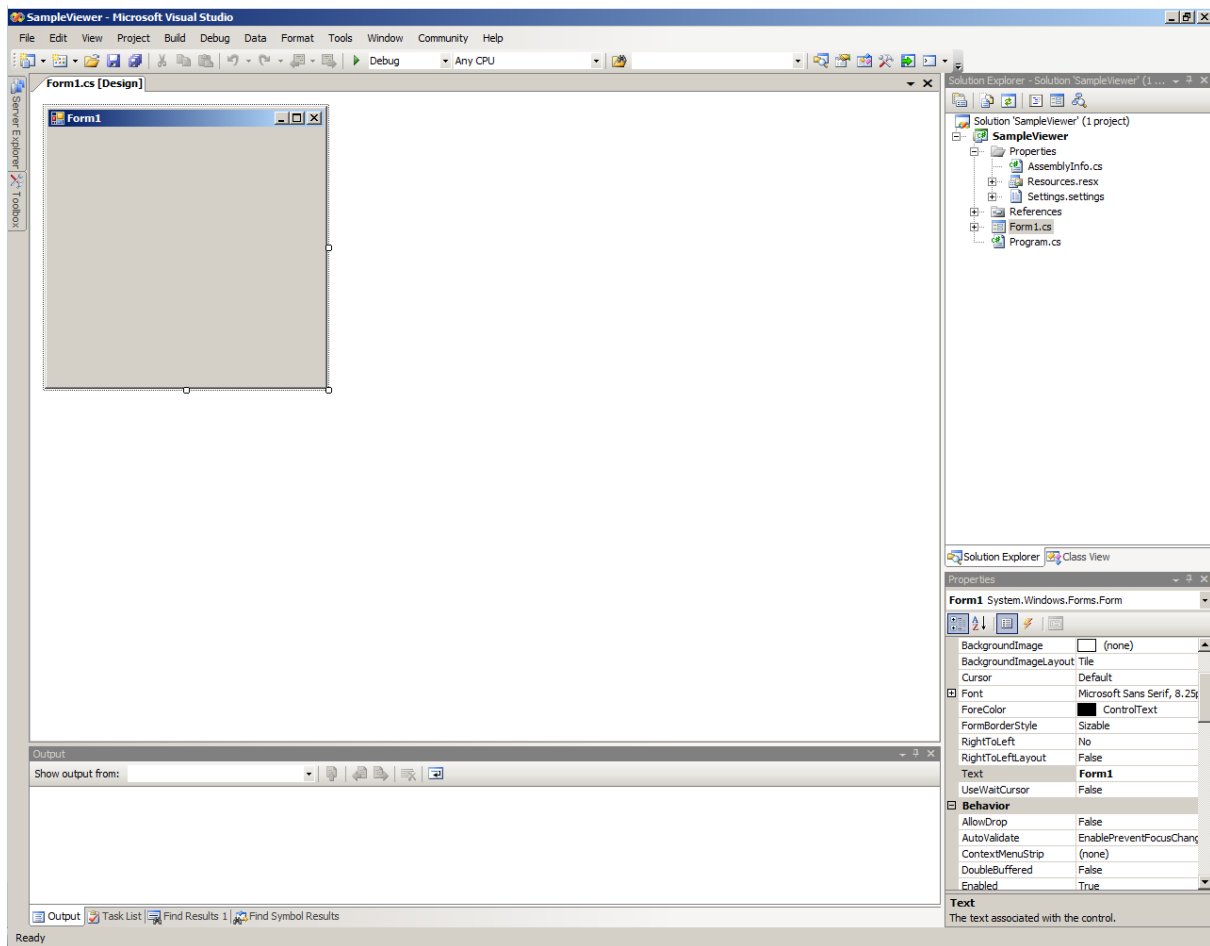


4. After clicking OK, Visual Studio 2005 creates an application framework using the selected template.

Setting up the Visual Studio 2005 Workspace

Before proceeding further, take the time now to set up the workspace configuration that will be used throughout the tutorial.

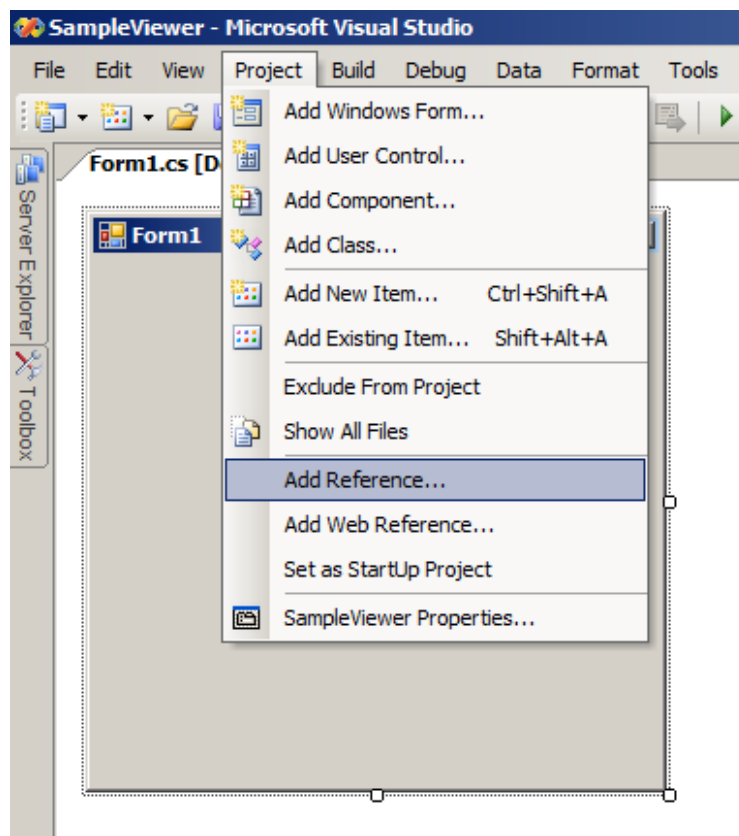
1. Make the solution explorer visible by selecting Solution Explorer from the View menu.
2. Make the properties window visible by selecting Properties Window from the View menu.
3. Make the output window visible by selecting Output from the View menu.
4. Make the task list window visible by selecting Task List from the View menu.
5. Make the find results window visible by selecting Find Results 1 from the View > Find Results menu.
6. Make the find symbols window visible by selecting Find Symbol Results from the View > Find Results menu.
7. Your workspace should now look like the following:



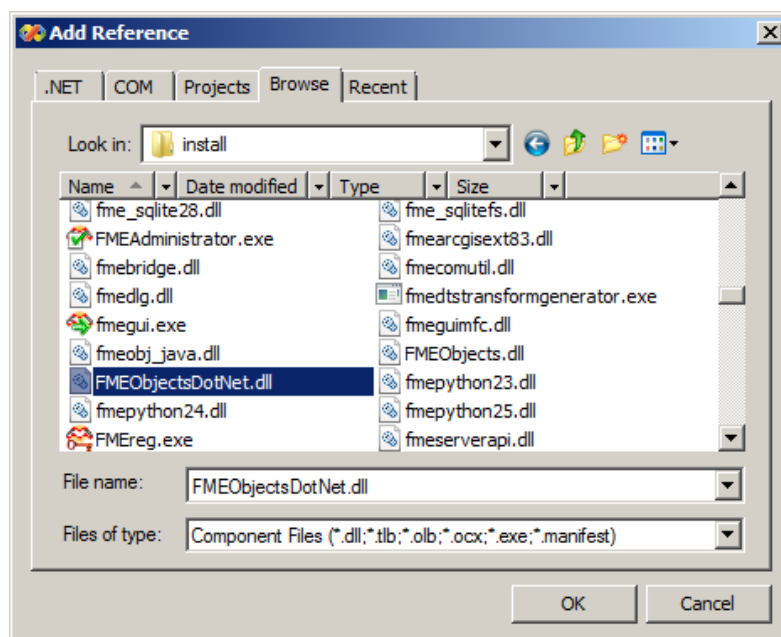
Adding a Reference to FMEObjectsDotNet.dll

Add a reference to the FMEObjectsDotNet.dll assembly so the application can use the FME Objects API.

1. Select Add Reference... from the Project menu.



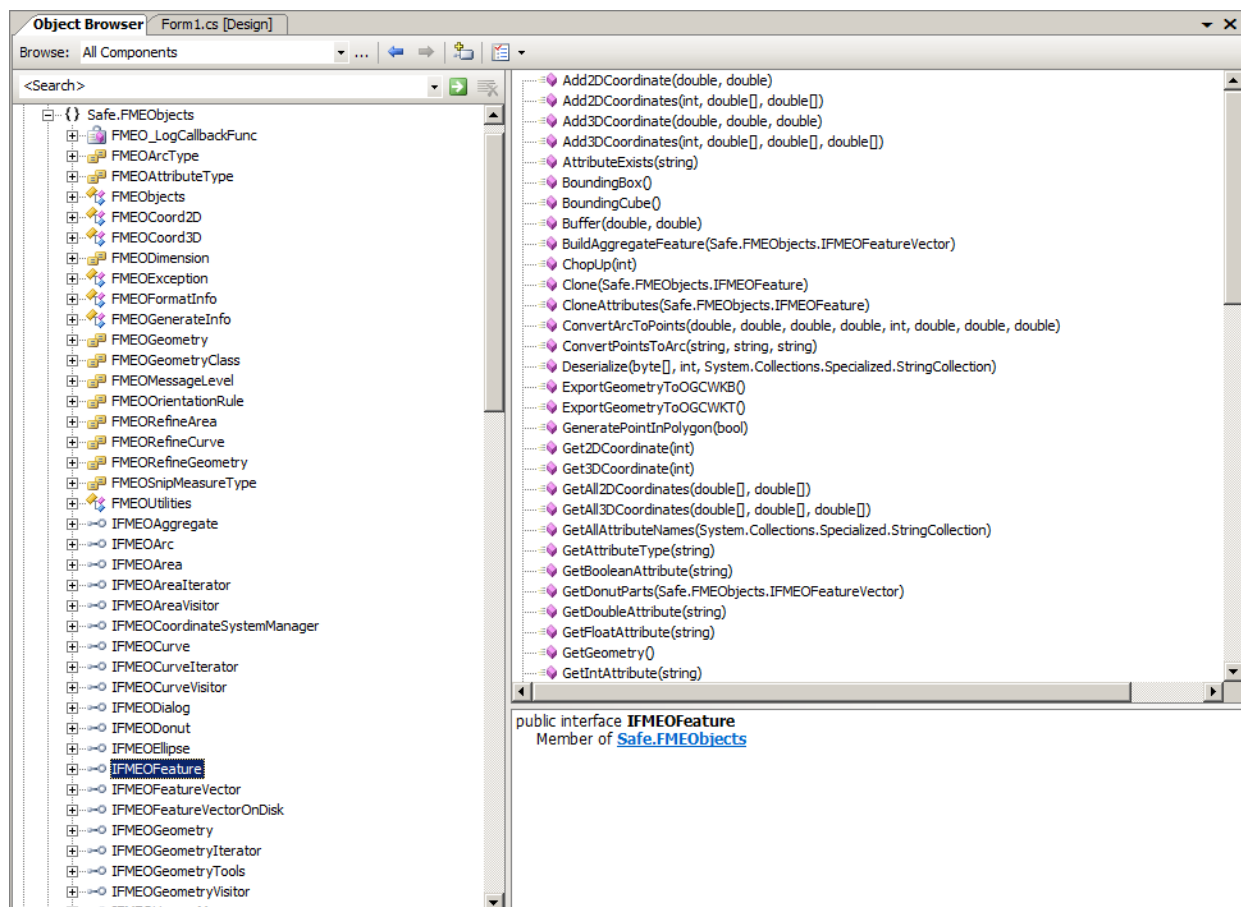
2. In the Add Reference dialog that appears, click the Browse tag and locate the FME installation folder.
3. Select FMEObjectsDotNet.dll and click OK to close the dialog.



4. Select Code from the View menu to switch to the Code view.
5. After adding the reference, you will need to expose the functionality of the namespaces that you wish to access. Add a new namespace "Safe.FMEObjects" to Form1.cs.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Safe.FMEObjects;
```

All the FME Objects related classes and interfaces are located in the Safe.FMEObjects namespace and all of the FME Objects related read-only properties are located in the Safe.FMEObjects.Constants namespace. The actual namespace names can be viewed either through the Object Browser (Ctrl+Alt+j) or using the FME Objects .NET API help system.



Chapter 1

Working with IFMEOSession

In this chapter

Setting up the application GUI
Adding an exit handler to the application
Creating an instance of IFMEOSession
Retrieving the IFMEOGGeometryTools
Turning on FME Objects logging
Using IFMEODialog to display the About dialog

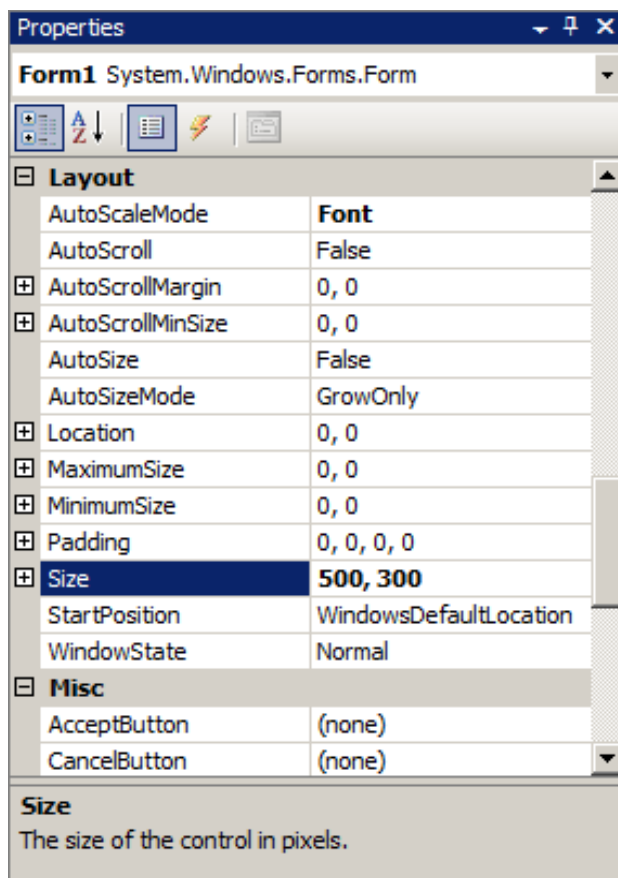
Objective

In this chapter, you will setup the initial GUI for the sample application. You will also begin to write FME Objects .NET code and be introduced to the IFMEOSession and IFMEODialog classes.

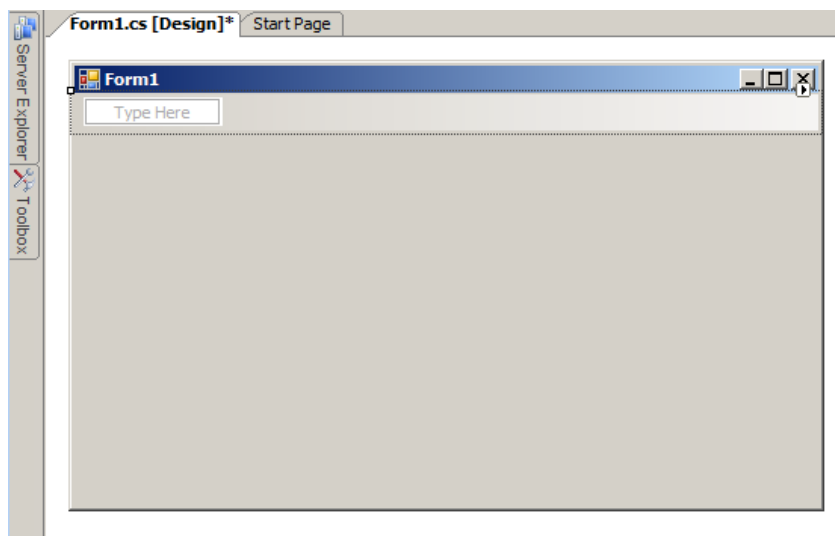
Setting up the Application GUI

In this exercise, you will use Windows Forms to design the GUI layout of the sample application.

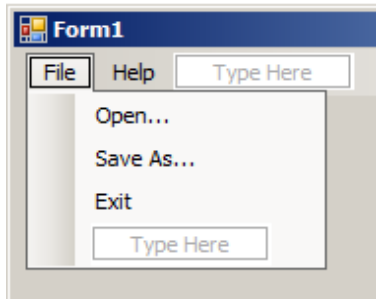
1. Open your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the getting_started folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view by selecting Designer from the View menu.
3. Select the main window form, and inside its properties display, set Size to be 500,300. (This is just a guideline, the exact size is not important)



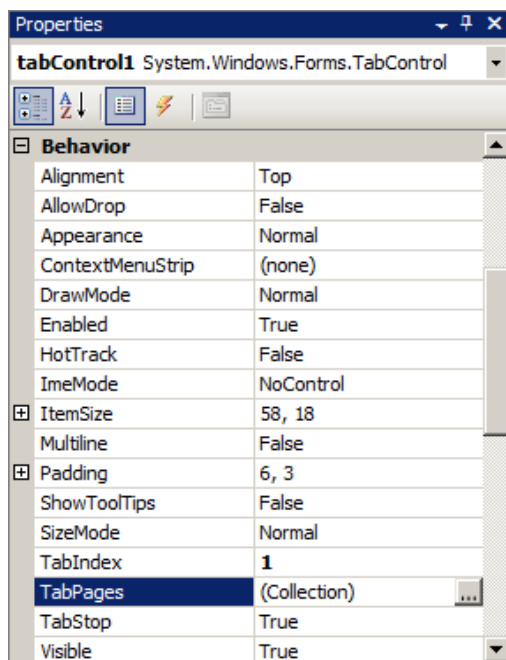
4. Bring up the Toolbox by selecting Toolbox from the View menu. Select MenuStrip from the Menus & Toolbars section of the Toolbox and drag an instance of it to the main window form.
5. Form1 should now look as follows:



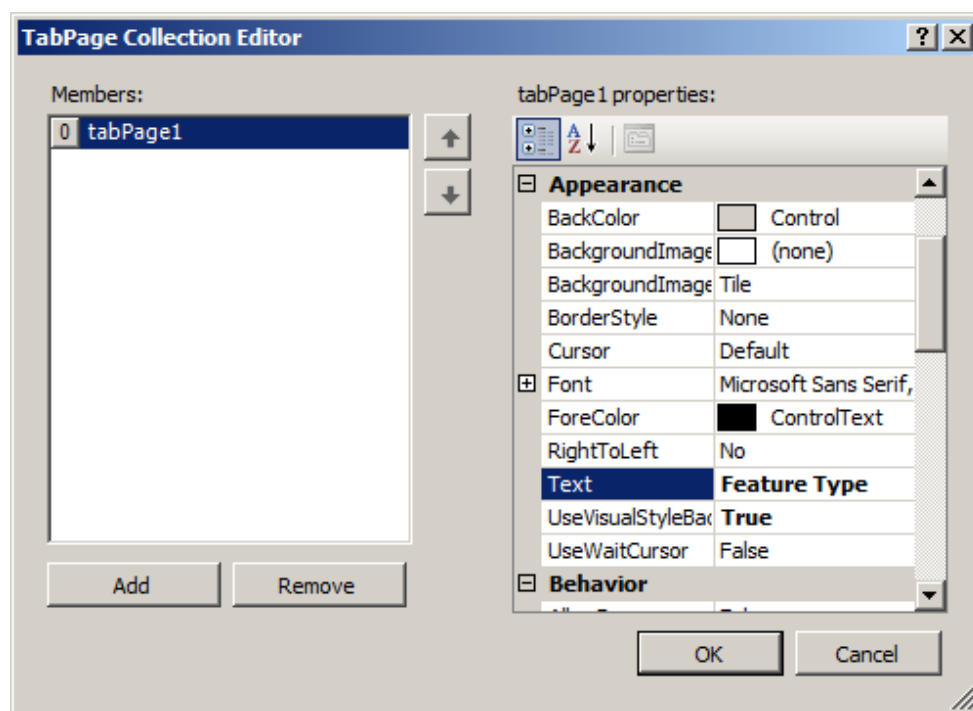
6. Add a new menu item by clicking on "Type Here", and type in the text "File".
7. Repeat the above step to add in the sub-items "Open...", "Save As...", and "Exit" under the File menu. Also, add a new menu "Help", and add the sub-item "About...".



8. Next, we add a TabControl to the application. From the Toolbox, drag an instance of TabControl on to Form1. The exact size is not important, and it can be readjusted later.
9. Under its properties, find Behavior > TabPages, and click on the browse button.

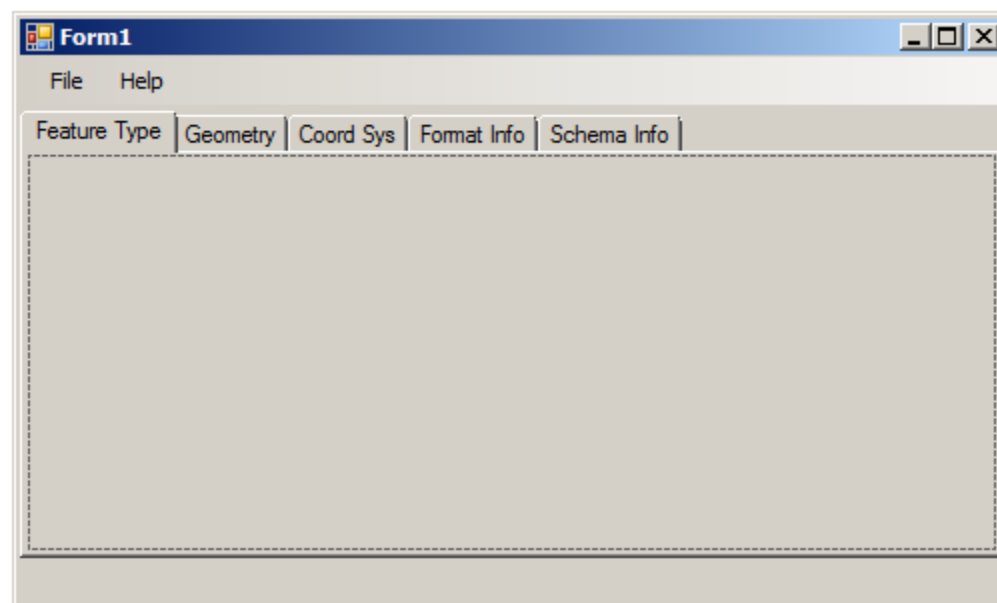


10. On the TabPage Collection Editor dialog, add a new tab by clicking on Add. Under its properties, set Appearance > Text to "Feature Type".



11. Repeat the step above to add tabs for "Geometry", "Coord Sys", "Format Info", and "Schema Info".

12. Now, adjust the size of the TabControl to look like the following:



13. Add a status bar to the application. From the Toolbox, select StatusStrip, and drag an instance of it to Form1.

14. Click on the left hand side of the new status bar and a pop-up menu should appear. Select StatusLabel from the list; this will create a label on the status bar where text can be displayed. Under its properties, set Appearance > Text to initially be empty.



15. Switch to the Code view and define the following method inside Form1.cs:

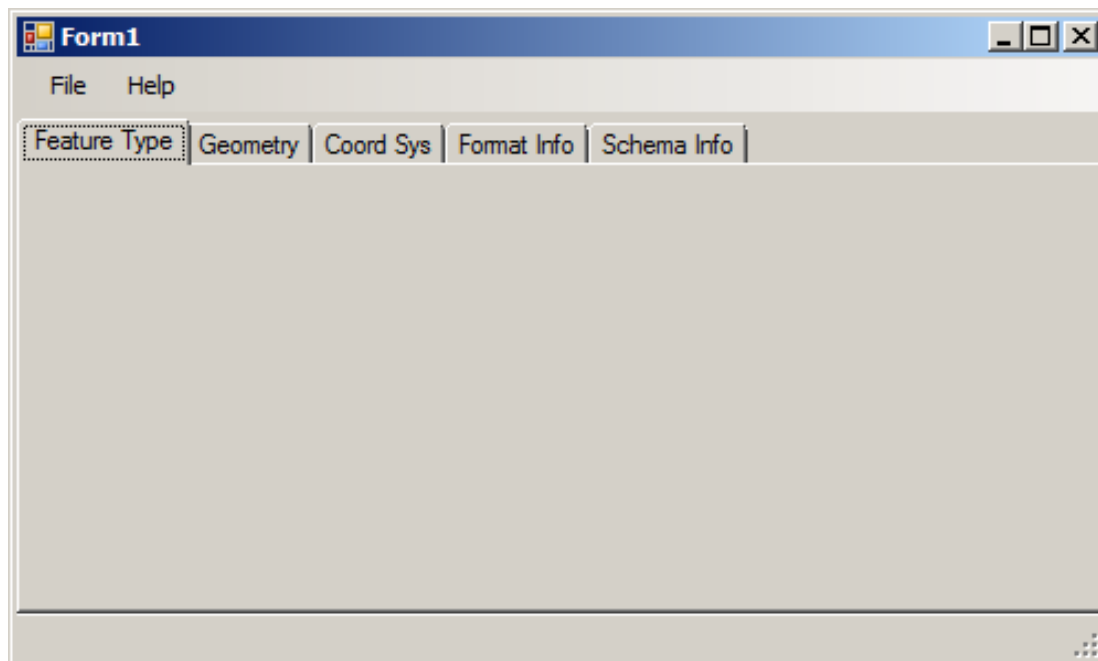
```
private void updateStatusBar(string pText)
{
    toolStripStatusLabel1.Text = pText;
    statusStrip1.Refresh();
}
```

This method will be used later in the tutorial for updating information to the status bar.

Testing your changes

Test your changes by building and running the application. Select Build Solution from the Build menu. Visual Studio 2005 indicates a successful build by showing "Build: 1 succeeded, 0 failed, 0 skipped" inside the Output window. If compilation is successful, proceed to run the application by selecting Start from the Debug menu.

The application should look as follows:



Close the application by clicking the "X" button at the top right-hand corner.


Adding an Exit Handler to the Application

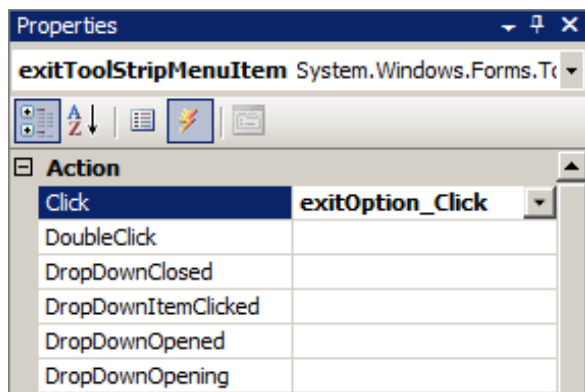
In this exercise, you will add a handler that lets a user exit the sample application by selecting Exit from the File menu.

1. Declare a new event handler by typing the following into Form1.cs:

```
private void exitOption_Click(object sender, System.EventArgs e)
{
    this.Dispose();
    Application.Exit();
}
```

This method will close the application by calling on Application.Exit().

2. Now, assign the new handler to the corresponding event. Switch to the Designer view. On the sample application, select Exit from the File menu. At the top of its Properties box, click on the Events button  to see the list of event handlers assigned to the Exit sub-item.
3. Under Misc > Click, select the exitOption_click event handler. The application will now respond when a user selects Exit from the File menu.



Testing your changes

Test your changes by building and running the application. The application should close when you select Exit from the File menu.

Creating an Instance of IFMEOSession

FME Objects .NET is a cross language compliant application programming interface (API). As a result, you should be able to develop an FME Objects application in any .NET language of your liking.

In FME Objects .NET, all classes except for the FMEObjects class are instantiated using the new keyword, while all non-geometry based interfaces are instantiated using the creation methods of IFMEOSession. IFMEOSession is created using a static method called CreateSession() on the FMEObjects class. It's important to understand that the IFMEOSession interface provides the methods needed to instantiate all the other FME Objects.

In object destruction, classes and interfaces are also treated differently. All the classes are completely managed by the .NET runtime which will take care of freeing resources. However, all the FME Objects .NET interfaces also implement the commonly used IDisposable interface. This means each interface will have a Dispose() method to free resources. It is strongly recommended that Dispose() be called when an interface object is no longer required. This will minimize the usage of memory and ensure that any resources such as files and database connections are immediately made available to other objects when they are no longer needed.

The first step is to create an instance of IFMEOSession inside the application.

1. At the top of Form1.cs, add a new heading where you will be declaring your member variables. Declare a new private member variable named m_fmeSession of type IFMEOSession and set its initial reference to null.

```
private IFMEOSession m_fmeSession = null;
```

2. Inside the class constructor, create an instance of IFMEOSession using FMEObjects.CreateSession(), and initialize the new session with null.

```
public Form1()
{
    InitializeComponent();

    m_fmeSession = FMEObjects.CreateSession();
    m_fmeSession.Init(null);
}
```

3. When the application exits, the session needs to be destroyed to free up resources. Switch to the Form1.Designer.cs file and add the following code to the Dispose method:

```
protected override void Dispose( bool disposing )
{
    if( disposing && (components != null))
    {
        components.Dispose();
    }

    // Every FME Objects application MUST at the very least dispose the
    // session to ensure clean up of all resources. For applications
    // that require efficient use of memory, Dispose() is available on
    // all FME Objects interfaces and should be called immediately after
    // they are no longer needed to immediately clean up resources.
    if (m_fmeSession != null)
    {
        m_fmeSession.Dispose();
        m_fmeSession = null;
    }
}
```

```

        base.Dispose( disposing );
    }

```

Testing your changes

Test your changes by verifying that your application still compiles successfully.

Retrieving the IFMEOGGeometryTools

As mentioned above, the IFMEOSession instantiates all of the non-geometry based interfaces through its creation methods. IFMEOGGeometryTools is the other key component in FME Objects .NET. It is primarily used to create and manipulate geometry objects (See Chapter 4 for more information on Geometry objects).

1. Create a variable of type IFMEOGGeometryTools and set its initial reference to null.

```
private IFMEOGGeometryTools m_fmeGeometryTools = null;
```

2. Inside the class constructor, FMEObjects.CreateSession(), and initialize the new session with null.

```
public Form1()
{
    InitializeComponent();

    ...

    m_fmeGeometryTools = m_fmeSession.GeometryTools();
}

```

3. When the application exits, the geometry tools needs to be destroyed to free up resources. Switch to the Form1.Designer.cs file and add the following code to the Dispose method:

```
protected override void Dispose( bool disposing )
{
    if( disposing && (components != null))
    {
        components.Dispose();
    }

    // Every FME Objects application MUST at the very least dispose the
    // session to ensure clean up of all resources. For applications
    // that require efficient use of memory, Dispose() is available on
    // all FME Objects interfaces and should be called immediately after
    // they are no longer needed to immediately clean up resources.

    if (m_fmeGeometryTools != null)
    {
        m_fmeGeometryTools.Dispose();
        m_fmeGeometryTools = null;
    }

    ...
}

```

Testing your changes

Test your changes by verifying that your application still compiles successfully.

Turning on FME Objects Logging

The IFMEOLogFile object allows applications to log messages and features to a text file. This is useful for testing and debugging applications, both during development and after they've been deployed.

1. Inside Form1.cs, declare a new member variable named `m_fmeLogFile` of type `IFMEOLogFile` and set its initial reference to null.
2. Inside the class constructor, add the following code:

```
public Form1()
{
    InitializeComponent();

    m_fmeSession = FMEObjects.CreateSession();
    m_fmeSession.Init(null);

    m_fmeGeometryTools = m_fmeSession.GetGeometryTools();

    m_fmeLogFile = m_fmeSession.LogFile();
    m_fmeLogFile.SetFileName(@"**\logfile.log", false);
}
```

The `SetFileName` method of `IFMEOLogFile` allows you to specify the path to your output log file. Set the path to a convenient location of your choice (ie. `m_fmeLogFile.SetFileName(@"C:\temp\logfile.log",false)`).

3. As with the previous exercise, you must ensure that the `IFMEOLogFile` object is disposed when the application terminates. Switch to the `Form1.Designer.cs` file and add the following code to the `Dispose` method:

```
protected override void Dispose( bool disposing )
{
    if( disposing && (components != null))
    {
        ...
    }

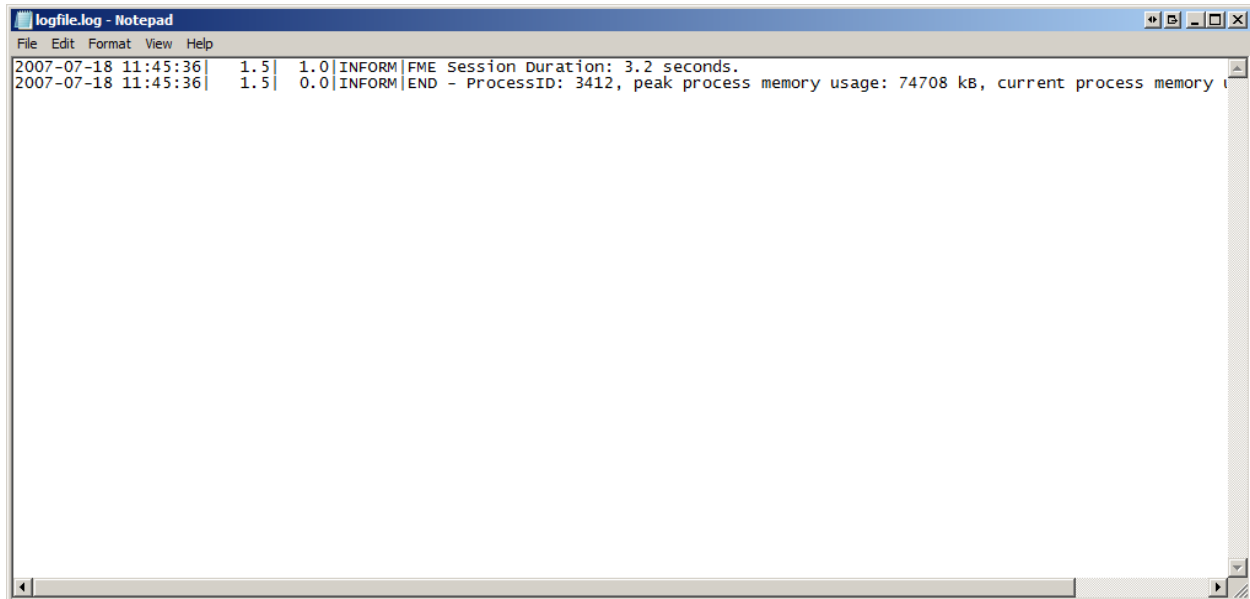
    // Every FME Objects application MUST at the very least dispose the
    // session to ensure clean up of all resources. For applications
    // that require efficient use of memory, Dispose() is available on
    // all FME Objects interfaces and should be called immediately after
    // they are no longer needed to immediately clean up resources.

    if (m_fmeLogFile != null)
    {
        m_fmeLogFile.Dispose();
        m_fmeLogFile = null;
    }

    ...
}
```

Testing your changes

Test your changes by building and running the application. Open the log file and you should see recorded information about the version of your FME Suite.



Using IFMEODialog to Display About Dialog

The IFMEODialog object provides access to the FME dialogs. IFMEODialog works closely with IFMEOREader and IFMEOWriter to encapsulate all the format specific information that readers and writers require during initialization.

In this exercise, you will use IFMEODialog to bring up the standard FME About dialog. This dialog should be displayed when a user selects About from the Help menu.

1. Inside Form1.cs, declare a new member variable named `m_fmeDialog` of type `IFMEODialog` and set its initial reference to null.
2. Inside the class constructor, add the following code:

```
public Form1()
{
    InitializeComponent();

    ...

    m_fmeLogFile = m_fmeSession.LogFile();
    m_fmeLogFile.SetFileName(@"**\logfile.log", false);

    m_fmeDialog = m_fmeSession.CreateDialogBox();
    m_fmeDialog.SetParentWindow(this.Handle);
}
```

- As with the IFMEOLogFile object, you must ensure that the IFMEODialog object is disposed when the application exits. Switch to the Form1.Desginer.cs file and add the following code to the Dispose method:

```
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        ...

        // Every FME Objects application MUST at the very least dispose the
        // session to ensure clean up of all resources. For applications
        // that require efficient use of memory, Dispose() is available on
        // all FME Objects interfaces and should be called immediately after
        // they are no longer needed to immediately clean up resources.

        if (m_fmeLogFile != null)
        {
            ...
        }

        if (m_fmeDialog != null)
        {
            m_fmeDialog.Dispose();
            m_fmeDialog = null;
        }

        ...
    }
}
```


- Define a new event handler named aboutOption_Click as shown:

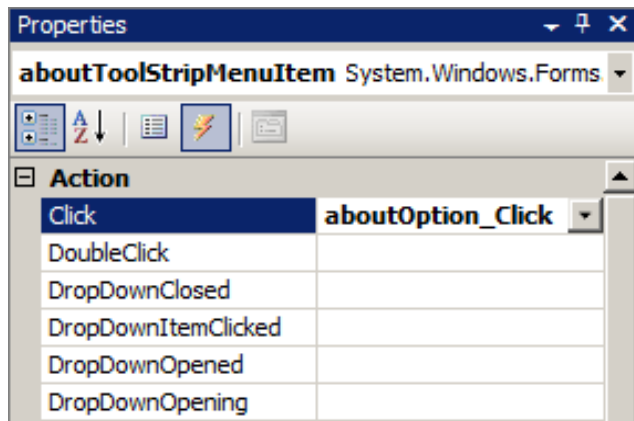
```
private void aboutOption_Click(object sender, System.EventArgs e)
{
    try
    {
        m_fmeDialog.About("Sample Application");
    }

    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage,
                                      FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace,
                                      FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(),
                                      FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}
```

Notice that the IFMEODialog object was created through the session object. In addition, a try/catch block was used for handling the FMEOException. FMEOException is a custom exception class used for encapsulating all FME Objects .NET related runtime errors. In the code shown above, any FMEOExceptions thrown will result in its error message, stack trace, and error number to be recorded inside the log file.

5. Switch to the Designer view. On the sample application, select About from the Help menu. At the top of its Properties box, click on the Events button  to see its list of events.
6. Assign the aboutOption_click handler to the Click event. The application will now respond when a user selects About from the Help menu.



Testing your changes

Build the solution and run the application. Select About from the Help menu. The FME About dialog should now appear.



Chapter 2

Reading Features From a Source Dataset

In this chapter

Using IFMEODialog to prompt for the source dataset
Reading data features through IFMEOREader
Logging features to the log file

Objective

In the previous chapter, you saw how to use IFMEODialog to display the About dialog box. In this chapter, you will invoke another FME dialog for allowing a user to specify the source dataset. The user-specified parameters from the dialog will be used to create an IFMEOREader for reading in data features from the dataset. In addition, each feature will be logged to the output file through IFMEOLogFile.

Using IFMEODialog to Specify Source Dataset

The FME Source Prompt dialog allows a user to input a source format and a source dataset.

In this exercise, you will use IFMEODialog to bring up the FME Source Prompt dialog. This dialog should be displayed when a user selects the Open... from the File menu.

1. Open your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter1 folder. By default, the standard solution will output to the log file FMEObjectsTutorial_logfile.txt under C:\temp.
2. Open Form1.cs from the Solution Explorer. Add System.Collections.Specialized to the list of namespaces being used.

```
using System.Collections.Specialized;
```

3. Declare a private member variable named m_dataInfo of type FMEOFormatInfo and a private member variable named m_createDirectives of type StringCollection. Set their initial reference to null.

```
private FMEOFormatInfo m_dataInfo = null;  
private StringCollection m_createDirectives = null;
```

4. Inside the Form1 class constructor, instantiate the two new private members:


```
m_dataInfo = new FMEOFormatInfo();  
m_createDirectives = new StringCollection();
```

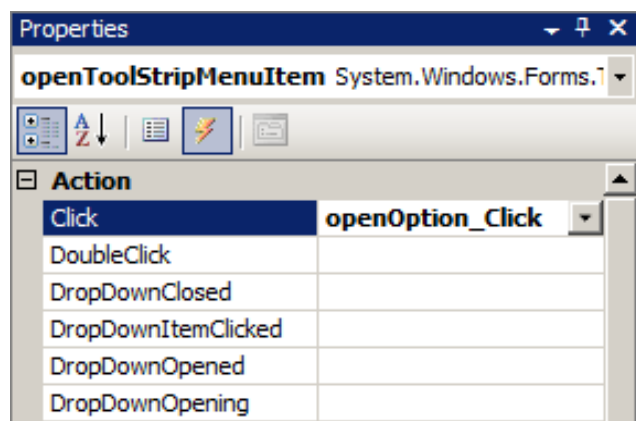
5. Define a new event handler named `openOption_Click` as shown:

```
private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            // Add more code later
        }
    }
    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(),
                                      FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}
```

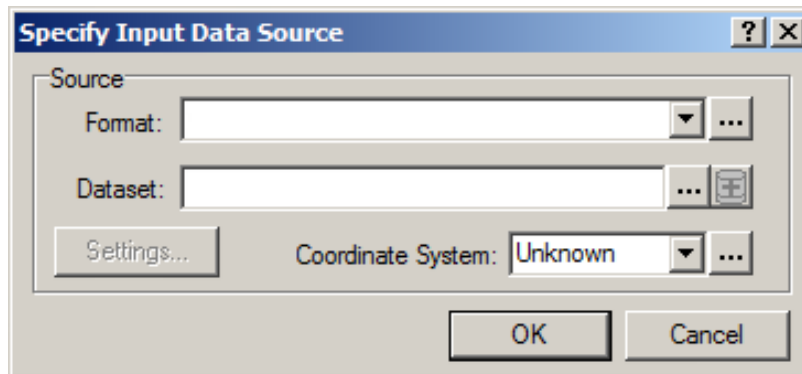
As you can see, the `SourcePrompt` method of `IFMEODialog` is used to display the FME Source Prompt dialog, and the information returned is stored back inside `m_dataInfo` and `m_createDirectives`.

- Now, attach the new handler to the corresponding event. Switch to the Designer view. On the sample application, click on File, and then select the Open menu item. At the top of its Properties box, click on the Events button  to see its list of events.
- Assign the `openOption_click` handler to the Click event. The application will now respond when a user selects Open... from the File menu.



Testing your changes

Build the solution and run the application. Select Open... from the File menu. The FME Source Prompt dialog should appear.



Click Cancel to dismiss the dialog.

Reading Data Features Through IFMEORReader

In the previous exercise, you used IFMEODialog to prompt the user for a description of the source dataset. In this exercise, you will use the information provided to create an IFMEORReader object that reads data features from the source dataset.

The IFMEORReader object provides a generic way to read from a source dataset. IFMEORReader has a built-in cache to improve performance when a data source is accessed multiple times. Applications can read data from multiple source datasets in parallel by opening multiple IFMEORReader objects. The IFMEORReader object also provides access to schema information (which you will see later in the tutorial).

1. Inside the openOption_Click method, add the following code to the conditional check:

```
if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
{
    // Add IFMEORReader code here
    // Update status bar
    updateStatusBar("Reading from source...");

    // If we are here, then the user did not hit Cancel. Now let's
    // create a reader to read in features. Caching is enabled for
    // the reader to improve performance on subsequent re-reads.
    IFMEORReader fmeReader = m_fmeSession.CreateReader(m_dataInfo.Format, false,
                                                         m_createDirectives);

    // Open the reader
    StringCollection openParams = new StringCollection();
    fmeReader.Open(m_dataInfo.Dataset, openParams);

    int featureCount = 0;
```

```

    // Now, read in the data features
    readDataFeatures(fmeReader, ref featureCount);

    // Update status bar with final feature count
    updateStatusBar("Total features: " + featureCount.ToString());

    // Clean up reader
    fmeReader.Close();
    fmeReader.Dispose();
    fmeReader = null;
}

```

By inserting the code inside the conditional check, the application does not create an IFMEORReader object when a user selects Cancel.

Notice that the IFMEORReader is created through the session object. The directives and format information specified by the user through the source prompt dialog are used to configure the IFMEORReader object.

2. Now, declare the helper method readDataFeatures.

```

private void readDataFeatures(IFMEORReader fmeReader, ref int featureCount)
{
    // Initialize counters
    int numFeatures = 0;

    // Now, read all the different features
    IFMEORFeature fmeFeature = m_fmeSession.CreateFeature();
    while (fmeReader.Read(fmeFeature))
    {
        // Increment feature count
        numFeatures++;
    }
    fmeFeature.Dispose();

    featureCount = numFeatures;
}

```

The Read method of IFMEORReader returns true as long as there are more features to read, otherwise, it returns false. Notice that each of the data features is read in as an IFMEORFeature object. The IFMEORFeature object provides a generic spatial data representation consisting of a set of attributes and an optional geometry with an associated coordinate system. The IFMEORFeature object can also be used to process nonspatial data.

3. If the application was reading a large dataset, it may take an extended period of time to read in all the features. To make the application more user-friendly, you can update the status bar over at periodic intervals. Declare a new private integer constant named m_kUpdateInterval and set its value to 500.

```
private const int m_kUpdateInterval = 500;
```

4. Inside readDataFeatures, update the code as follows:

```

private void readDataFeatures(IFMEORReader fmeReader, ref int featureCount)
{
    // Initialize counters
    int numFeatures = 0;

```

```

// Now, read all the different features
IFMEOWFeature fmeFeature = m_fmeSession.CreateFeature();
while (fmeReader.Read(fmeFeature))
{
    // Increment feature count
    numFeatures++;

    // Check if we need to update status bar
    if ( (numFeatures % m_kUpdateInterval) == 0 )
    {
        updateStatusBar("Read " + numFeatures.ToString() + " features...");
    }
}
fmeFeature.Dispose();

featureCount = numFeatures;
}

```

Now, the status bar will be updated for every 500 features that are read.

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open building_footprints.xdr (with Facet as format), and check the status bar to verify the number of features read.

Logging Features to the Log File

In the first chapter, you created an IFMEOWLogFile object and set its path to the output log file. In this exercise, you will use the LogFeature method of IFMEOWLogFile to log the input features to the output log file.

1. Inside the readDataFeatures method, update the code inside the read loop as shown:

```

// Now, read all the different features
IFMEOWFeature fmeFeature = m_fmeSession.CreateFeature();
while (fmeReader.Read(fmeFeature))
{
    // Log the feature to the log file
    m_fmeLogFile.LogFeature(fmeFeature, FMEOMessageLevel.Inform, -1);

    // Increment feature count
    numFeatures++;

    // Check if we need to update status bar
    if ( (numFeatures % m_kUpdateInterval) == 0 )
    {
        ...
    }
}

```

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open schoolRegions.gml (with GML - Geography Markup Language as format), and after reading has been complete, open the output log file to examine the logged features.

```

logfile.log - Notepad
File Edit Format View Help

2007-07-18 15:41:58 2.8 2.2 INFORM Module 'GML' API version matches current core version (3.2 20070516)
2007-07-18 15:41:58 2.8 0.0 INFORM The uri-map document 'file:///C:/FME/install/xml/urimap/gml_urimap.xml' is being used
2007-07-18 15:41:58 2.8 0.0 INFORM Using Dynamic Reader $Revision: 33857 $ ( $Date: 2007-02-08 23:17:48 -0800 (Thu, 08
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/Users/RSchimpel/Desktop/schoolRegions.xsd' ...
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/Users/RSchimpel/Desktop/schoolRegions.xsd --importing--> http://schemas.o
2007-07-18 15:41:58 2.8 0.0 INFORM URI 'http://schemas.opengis.net/gml/2.1.2/feature.xsd' mapped to 'file:///C:/FME/in
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/feature.x
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/feature.xsd --including--> geometr
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometry
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometryBasic2d.xsd --including-->
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometry
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometryBasic3d.xsd --including-->
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/measures
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/measures.xsd --including--> units
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/units.xs
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/units.xsd --including--> dictionar
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/dictionar
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/dictionary.xsd --including--> gml
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/gmlBase
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/gmlBase.xsd --including--> basicT
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/basicType
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/gmlBase.xsd --importing--> http://
2007-07-18 15:41:58 2.8 0.0 INFORM URI 'http://schemas.opengis.net/gml/3.1.0/xlink/xlinks.xsd' mapped to 'file:///C:/F
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/xlink/xlinks
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/feature.xsd --including--> tempor
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/temporal
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/temporal.xsd --including--> gmlBa
2007-07-18 15:41:58 2.8 0.0 INFORM schema document already parsed once, ignoring: gmlBase.xsd
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/feature.xsd --including--> geometr
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometry
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometryAggregates.xsd --including
2007-07-18 15:41:58 2.8 0.0 INFORM Parsing schema document 'file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometry
2007-07-18 15:41:58 2.8 0.0 INFORM file:///C:/FME/install/xml/schemas/gml/3.1.1/base/geometryPrimitives.xsd --including
2007-07-18 15:41:58 2.8 0.0 INFORM schema document already parsed once, ignoring: geometryBasic2d.xsd
2007-07-18 15:41:59 4.0 1.2 INFORM FME Configuration: Source coordinate system for reader R_1[GML] set to 'TX83-CF' as
2007-07-18 15:41:59 4.0 0.0 INFORM Coordinate System 'TX83-CF' parameters: CS_NAME='TX83-CF' DESC_NM='NAD83 Texas State
2007-07-18 15:41:59 4.0 0.0 INFORM Coordinate System 'TX83-CF' as OGC well known text: PROJCS["NAD83 Texas State Plane
2007-07-18 15:41:59 4.0 0.0 STATS Storing feature(s) to FME feature store file 'C:\Temp\logfile_log.ffs'
2007-07-18 15:41:59 4.0 0.0 INFORM ++++++
2007-07-18 15:41:59 4.0 0.0 INFORM Feature Type: 'school_districts'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'district_code' has value '227-901'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'fme_geometry' has value 'fme_polygon'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'fme_type' has value 'fme_area'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'gml_geometry_property' has value 'polygonProperty'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'name' has value 'Austin ISD'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'short_name' has value 'Austin'
2007-07-18 15:41:59 4.0 0.0 INFORM Attribute(string): 'trustee_code' has value '3'

```

Chapter 3

Grouping Features by Feature Type

In this chapter

Placing a ListView on the FeatureType tab
Using the FeatureType property of IFMEOFeature
Displaying the results inside ListView
Disabling and enabling the TabControl

Objective

In the previous chapter, you used IFMEOReader for reading in data features from a source dataset. Each of the data features was read in as an IFMEOFeature object.

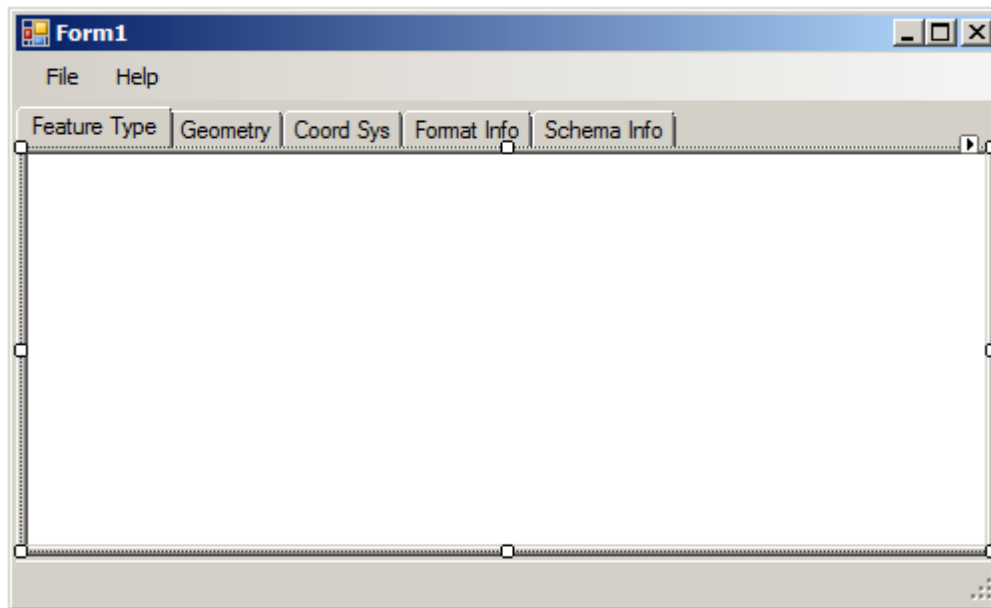
FME Objects data features are identified according to a well-defined classification scheme (ie. Theme, Layer, Level, Table, Class, etc). This classification scheme is also known as the feature type of a data feature.

In this chapter, you will learn to categorize features by their feature type.

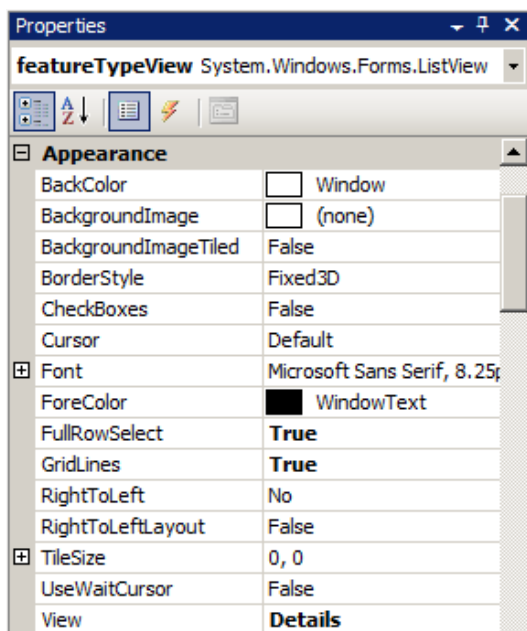
Placing a ListView on the Feature Type Tab

In this exercise, you will categorize features by their feature type. To show the results, you will use a Windows Form ListView to display the information in the form of a table.

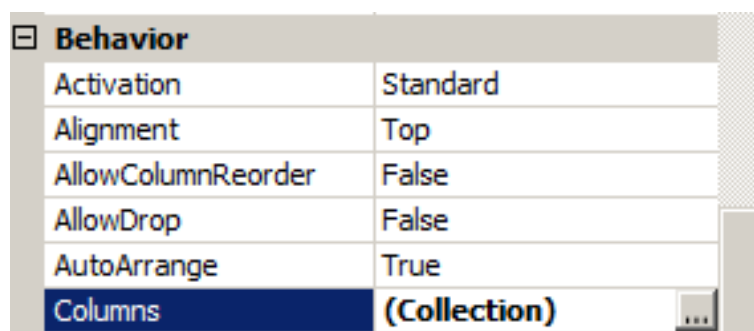
1. Open your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter2 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Click on the Feature Type tab to be in focus.
3. From the Toolbox, drag an instance of ListView to the sample application. On its properties, change its name from listView1 to featureTypeView. Adjust the size to look like the following:



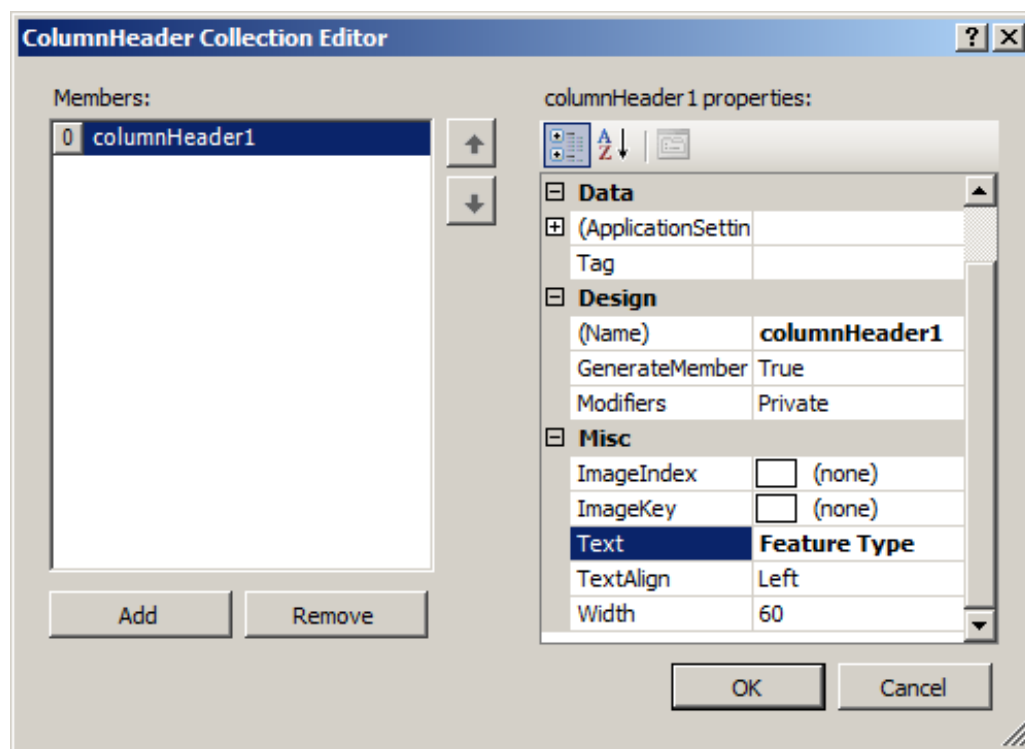
4. With featureTypeView selected, go to its Properties box. Set FullRowSelect and GridLines to True, as well as the View to be Details and MultiSelect to False.



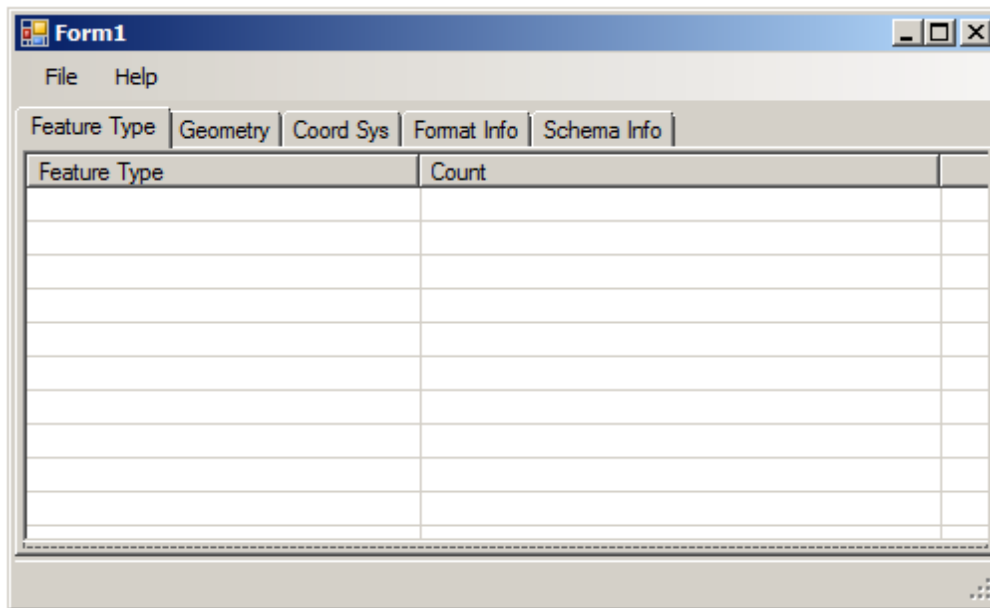
5. Find the Columns attribute and click on the browse button to bring up the.ColumnHeader Collection Editor dialog.



- Click on Add to add a new column. Under its properties, set Misc > Text to Feature Type.



- Repeat the step above to add another column Count. Then, click Ok to close the dialog.



Testing your changes

Build the solution and run the application. The ListView component should appear under the Feature Type tab.

Using the FeatureType property of IFMEOFeature

When dealing with datasets that have multiple feature types, one may find it useful to know the number of features that fall under each feature type category. In this exercise, you will learn to retrieve the feature type of each feature through the FeatureType property. Then, the information will be displayed in a table, where each row will show the number of features that were tallied for each feature type of the dataset.

1. Inside Form1.cs, declare a new private member variable named `m_featureTypeDictionary` of type of `SortedList<string, IFMEOFeatureVectorOnDisk>`. `m_featureTypeDictionary` will be used as a dictionary to map a feature type name to a vector of corresponding features that have that same feature type. Set the initial reference to null.

```
private SortedList<string, IFMEOFeatureVectorOnDisk> m_featureTypeDictionary = null;
```

2. Inside the class constructor, initialize `m_featureTypeDictionary` with a new instance of `SortedList<string, IFMEOFeatureVectorOnDisk>`.

```
m_featureTypeDictionary = new SortedList<string, IFMEOFeatureVectorOnDisk>();
```

3. Declare a new private integer constant `m_kMaxFeatureInMemory`, and set its value to 500. This is the constant that will be used as the upper limit for features stored in memory by `IFMEOFeatureVectorOnDisk`.

```
private const int m_kMaxFeatureInMemory = 500;
```

4. Declare a new method named `insertIntoFeatureTypeDictionary` as follows:

```
private void insertIntoFeatureTypeDictionary(IFMEOFeature pFeature)
{
    string currFeatureType = pFeature.FeatureType;

    // First, we check for where to put the feature inside m_featureTypeDictionary

    // Check to see if an entry for currFeatureType already exist
    if (!m_featureTypeDictionary.Contains(currFeatureType))
    {
        // We must create a new entry to the m_featureTypeDictionary
        IFMEOFeatureVectorOnDisk newVectorOnDisk =
            m_fmSession.CreateFeatureVectorOnDisk(m_kMaxFeatureInMemory);

        // Add the new entry to the dictionary
        m_featureTypeDictionary.Add(currFeatureType, newVectorOnDisk);
    }

    // Now, we put the feature into the proper IFMEOFeatureVectorOnDisk
    IFMEOFeatureVectorOnDisk currVectorOnDisk = m_featureTypeDictionary[currFeatureType];
    currVectorOnDisk.Append(pFeature);
}
```

This method takes in an `IFMEOFeature` as a parameter, and it examines its feature type through the `FeatureType` property. Then, using the feature type as the key, the method retrieves the corresponding `IFMEOFeatureVectorOnDisk` inside `m_featureTypeDictionary`, and appends the feature into the vector (stored for later use). If the key entry does not exist, it creates a new `IFMEOFeatureVectorOnDisk`, and a new map entry is entered inside `m_featureTypeDictionary`.

`IFMEOFeatureVectorOnDisk` is used instead of `IFMEOFeatureVector` so that the application can handle very large data volumes. For such datasets, it is not a good approach to hold all the features in memory (which is what `IFMEOFeatureVector` will do). A better approach is to use `IFMEOFeatureVectorOnDisk`, which has an upper limit on the number of features to retain in memory. Once the limit is exceeded, the remaining features are stored on disk.

5. Declare a new method named `disposeFeatureTypeDictionaryEntries` as follows:

```
private void disposeFeatureTypeDictionaryEntries()
{
    IEnumerable<KeyValuePair<string, IFMEOFeatureVectorOnDisk>> iterator =
        m_featureTypeDictionary.GetEnumerator();

    // Iterate through each entry m_featureTypeDictionary, and clear each
    // IFMEOFeatureVectorOnDisk
    while (iterator.MoveNext())
    {
        string currFeatureType = iterator.Current.Key;

        IFMEOFeatureVectorOnDisk currVectorOnDisk = iterator.Current.Value;
```

```

        // Calling Clear of IFMEFeatureVectorOnDisk should also dispose each of the
        // features that it contains
        currVectorOnDisk.Clear();
        currVectorOnDisk.Dispose();
        currVectorOnDisk = null;
    }

    // Finally, call Clear of m_featureTypeDictionary
    m_featureTypeDictionary.Clear();
}

```

This method is used to dispose all the features that are stored inside the `m_featureTypeDictionary`.

6. Declare a wrapper to call `disposeFeatureTypeDictionaryEntries` as shown below. This method will be extended in the future chapters.

```

private void resetAllDictionaries()
{
    // Dispose all the features inside of m_featureTypeDictionary
    disposeFeatureTypeDictionaryEntries();
}

```

7. Now, with all the methods defined, you can make the appropriate calls inside the `openOption_Click` and `readDataFeatures`. Update `openOption_Click` as follows:

```

private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            // Disposes and clears entries inside m_featureTypeDictionary
            resetAllDictionaries();

            ...

            // Clean up reader
            fmeReader.Close();
            fmeReader.Dispose();
            fmeReader = null;
        }
    }
    catch (FMEOException ex)
    {
        ...
    }
}

```

Before reading in the new dataset, the method ensures that all existing features stored from the previous dataset have been disposed by calling `resetAllDictionaries`.

8. Update `readDataFeatures` as follows:

```

private void readDataFeatures(IFMEORReader fmeReader, ref int featureCount)
{
    ...

    // Now, read all the different features
    IFMEFeature fmeFeature = m_fmeSession.CreateFeature();
}

```

```

while (fmeReader.Read(fmeFeature))
{
    // Log the feature to the log file
    ...

    // For every feature, we will put it into m_featureTypeDictionary
    insertIntoFeatureTypeDictionary(fmeFeature);

    ...

    // Check if we need to update status bar
    if ( (numFeatures % m_kUpdateInterval) == 0 )
    {
        ...
    }

    // Create a new feature for the next read
    fmeFeature = m_fmeSession.CreateFeature();
}
...
}

```

For every feature that is read in from the new dataset, it is added to `m_featureTypeDictionary` by calling `insertIntoFeatureTypeDictionary`.

Notice that at the end of each iteration, a new `IFMEOFeature` object is created. This is done because `IFMEOFeatureVectorOnDisk` takes ownership of the feature that it appends (done inside `insertIntoFeatureTypeDictionary`). Hence, for every iteration, you must create a new `IFMEOFeature` object to be used for storing the next input feature.

9. Finally, switch to the `Form1.Designer.cs` file and add a call to `resetAllDictionaries` inside the `Dispose` method. This will ensure that all the features are disposed when the application exits.

```

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        ...
    }

    // Disposes and clears entries inside m_featureTypeDictionary
    resetAllDictionaries();

    if (m_fmeLogFile != null)
    {
        ...
    }

    ...
}

```

Testing your changes

Build the solution and verify that the application still successfully compiles.

Displaying the Results Inside ListView

In the previous exercise, you processed every feature being read in and stored it inside `m_featureTypeDictionary`. In this exercise, you will display the results inside the `ListView` component of the Feature Type tab.

1. Inside `Form1.cs`, declare a new enumeration to the class as follows:

```
private enum eTabPanels
{
    FeatureTypeTab = 0,
    GeometryTab,
    CoordSysTab,
    FormatInfoTab,
    SchemaInfoTab
}
```

The enumeration is used to denote that `FeatureTypeTab` is representing the Feature Type tab with an index of 0 (since it is the first tab), and the `Geometry` tab with an index of 1. The same idea applies to the remaining tabs.

2. Declare a method named `refreshCurrentTab` that will be called to refresh the contents of the tab in focus:

```
private void refreshCurrentTab()
{
    eTabPanels selectedPanel = (eTabPanels) tabControl1.SelectedIndex;

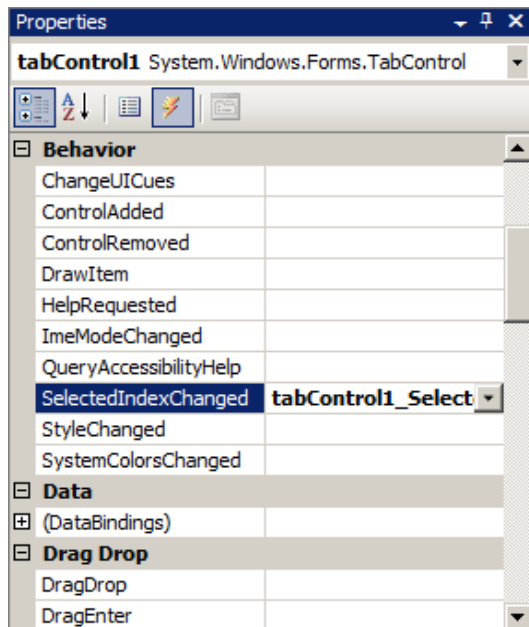
    // Update the corresponding tab
    switch(selectedPanel)
    {
        case eTabPanels.FeatureTypeTab:
            updateFeatureTypeTab();
            break;
        default:
            break;
    }
}
```

This method uses a switch statement to determine the appropriate tab to refresh.

3. Declare a new event handler that will be called whenever a user selects a different tab in focus. This event handler will call `refreshCurrentTab` to refresh the proper tab.

```
private void tabControl1_SelectedIndexChanged(object sender, System.EventArgs e)
{
    refreshCurrentTab();
}
```

4. Assign the new event handler to `tabControl1`. Switch to the Designer view and select `tabControl1` (the component with the nested tabs). Assign `tabControl1_SelectedIndexChanged` to the `SelectedIndexChanged` event.



5. You would also like to refresh the current tab whenever a new dataset has been read in. Inside the `openOption_Click` handler, add the call to `refreshCurrentTab` as follows:

```
if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
{
    ...

    // Clean up reader
    fmeReader.Close();
    fmeReader.Dispose();
    fmeReader = null;

    // Refresh the current tab
    refreshCurrentTab();
}
```

6. Finally, declare the `updateFeatureTypeTab` method that will be used to update the contents of the Feature Type tab.

```
private void updateFeatureTypeTab()
{
    // Clear the current entries in the featureTypeView
    featureTypeView.Items.Clear();

    IEnumerator<KeyValuePair<string, IFMEOfeatureVectorOnDisk>> iterator =
        m_featureTypeDictionary.GetEnumerator();
    while (iterator.MoveNext())
    {
        string currFeatureType = iterator.Current.Key;
        IFMEOfeatureVectorOnDisk currVectorOnDisk =
            m_featureTypeDictionary[currFeatureType];

        // Create a new subItem list - it will be a row of data containing
        // the feature type and its associated count
        string [] subItemList = {currFeatureType, currVectorOnDisk.Count.ToString()};
    }
}
```



```

        ListViewItem newItem = new ListViewItem(subItemList,-1);

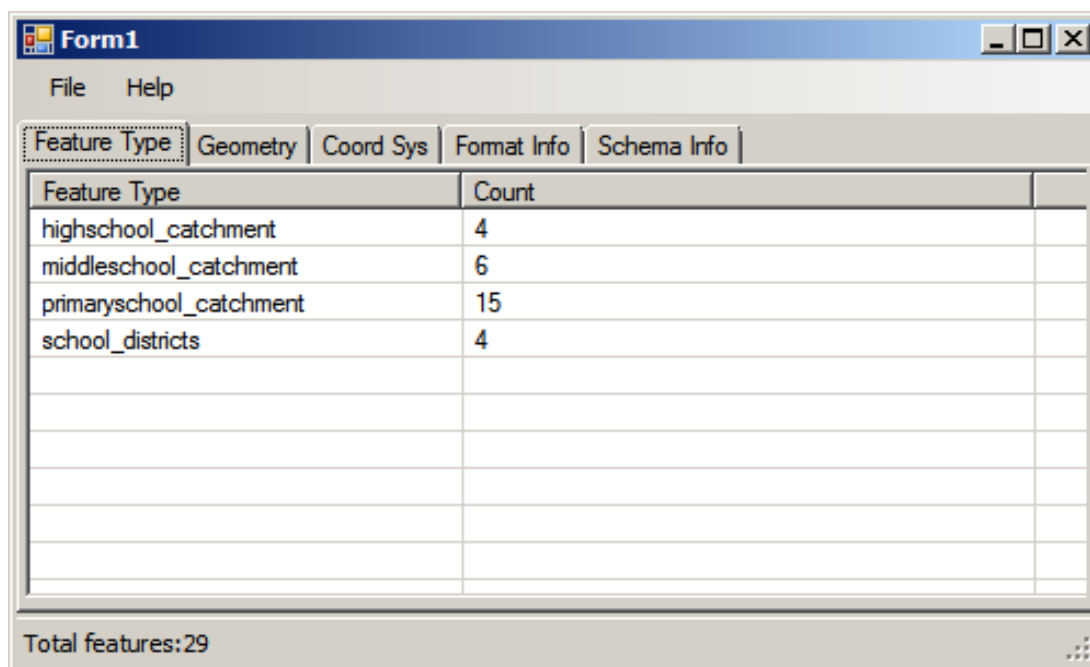
        // Add the item to featureTypeView
        featureTypeView.Items.Add(newItem);
    }
}

```

The `updateFeatureTypeTab` method iterates through each entry of `m_featureTypeDictionary`. For each feature type, it examines the `Count` property of its corresponding `IFMEOfeatureVectorOnDisk` (which reveals the number of features the vector contains), and displays the information inside a new row of the `ListView` component (`featureTypeView`).

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open `schoolRegions.gml` (with GML - Geography Markup Language as format), and then select the Feature Type tab to see the populated entries of the table.



Disabling and Enabling the TabControl

In this exercise, you will make a small change to the sample application to prevent unexpected behavior. For safety measures, the switching of the tab panels should be disabled until a dataset has been read in.

1. Inside the class constructor, update the code as follows:

```

public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    ...

    m_featureTypeDictionary = new SortedList();

    // Disable the tab panels
    tabControl1.Enabled = false;
}

```

Now, the tab control is disabled at the start of the application.

2. Inside the `openOption_Click` handler, add the code as follows:

```

private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        ...

        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            ...

            // Clean up reader
            fmeReader.Close();
            fmeReader.Dispose();
            fmeReader = null;

            // Enable the tab panels
            tabControl1.Enabled = true;

            // Refresh the current tab
            refreshCurrentTab();
        }
        ...
    }
    catch (FMEOException ex)
    {
        ...
    }
}

```

The tab control is now enabled after a dataset has been read in.

Testing your changes

Build the solution and run the application. You should not be able to select any tab until the first dataset has been read in.

Chapter 4

Grouping Features by Geometry

In this chapter

Placing a ListView on the Geometry tab
Using the GeometryClass property of IFMEOGGeometry
Displaying the results inside ListView

Objective

A Feature is associated with zero or one IFMEOGGeometry objects. An IFMEOGGeometry object encapsulates all of geometry data for a given Feature. Some examples of the data stored include coordinates, orientation, and dimension (2D vs. 3D).

There are a number of subclasses of IFMEOGGeometry to better represent the various geometries encountered in the GIS world. Some of major geometry categories currently included in FME Objects are:

- Curves
- Areas
- Collections

As the name suggests, geometries that fall under the Curves category are used to represent curves or portions of curves. Some basic curve segments include IFMEOArc and IFMEOLine and more complex curves can be generated by appending simple curve segments into an IFMEOPath.

As well, geometries that are closed (eg. The coordinates for the first and last point are the same) typically fall under the Areas category. Some simple areas include IFMEOEllipse and IFMEOPolygon and these can be combined into a IFMEODonut to represent complicated closed figures.

The last major geometry category, Collections, involves geometries that can singly represent a number of other geometries. A good example of this is the IFMEOAggregate that can append any type of geometry to itself. As well, for more specialized requirements, IFMEOMultiArea, IFMEOMultiCurve, IFMEOMultiPoint and IFMEOMultiText are available;

these geometries are used to combine a number of a given type of geometry together (eg. IFMEOMultiArea is used to combine a number of areas together.)

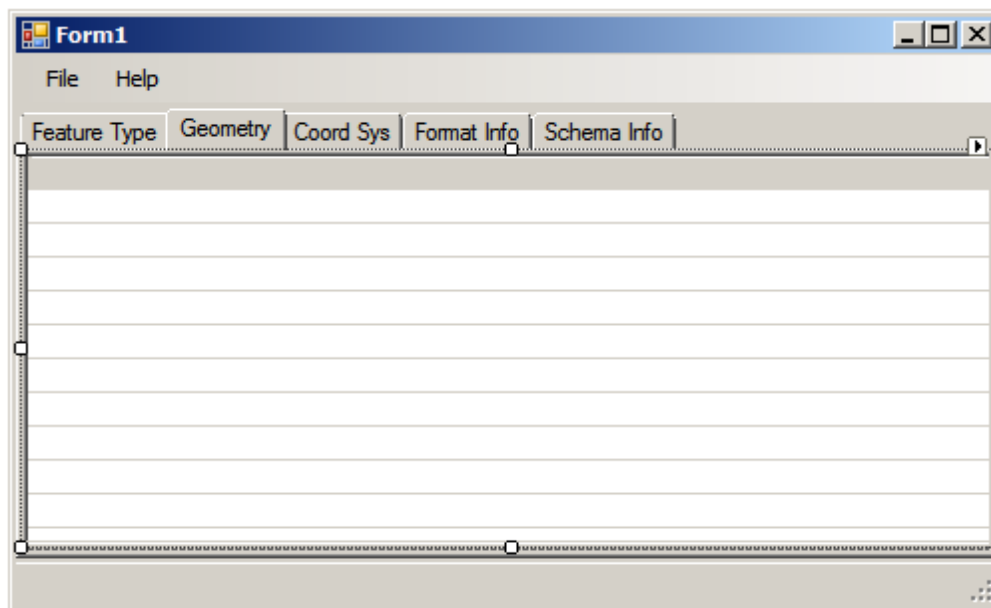
Besides these major geometry categories, there are also a number of other geometries which are common and useful. The IFMEOPoint and IFMEOText geometries are used to represent point and text objects respectively. As well, the IFMEONull object represents those situations where the given feature does not have any geometry information.

In this chapter, you will learn to categorize geometries by using the GeometryClass property.

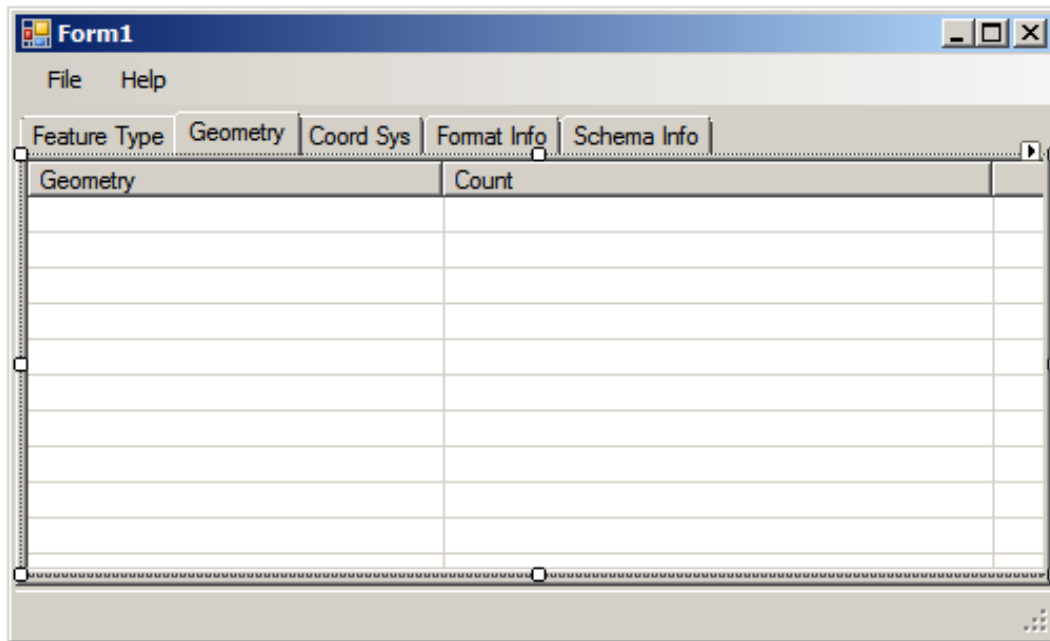
Placing a ListView on the Geometry Tab

In this exercise, you will place a ListView component on the Geometry tab.

1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter3 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Select the Geometry tab to be in focus.
3. From the Toolbox, drag an instance of ListView to the sample application. On its properties, change its name from listView1 to geometryView. Set FullRowSelect and GridLines to True, as well as the View to be Details and MultiSelect to False.
4. Adjust the size of the ListView to look like the following:



- Find the Behavior > Columns attribute and click on the browse button to bring up the ColumnHeader Collection Editor dialog.
- Click on Add to add a new column. Under its properties, set Misc > Text property to be Geometry.
- Repeat the step above to add another column Count. Then, click Ok to close the dialog.



Testing your changes

Build the solution and run the application. The ListView component should now appear under the Geometry tab.

Using the GeometryClass Property of IFMEOGGeometry

- Declare a new private member variable named `m_geometryDictionary` of type of `SortedList<FMEOGGeometryClass, IFMEOFeatureVectorOnDisk>`. `m_geometryDictionary` will be used as a dictionary to map a fme type value to a vector of corresponding features that have that same fme type. Set the initial reference to null.

```
private SortedList<FMEOGGeometryClass, IFMEOFeatureVectorOnDisk> m_geometryDictionary = null;
```

- Inside the class constructor, initialize `m_geometryDictionary` with a new instance of `SortedList<FMEOGGeometryClass, IFMEOFeatureVectorOnDisk>`.

```
m_geometryDictionary = new SortedList<FMEOGGeometryClass, IFMEOFeatureVectorOnDisk>();
```

3. Declare a new method named `insertIntoGeometryDictionary` as follows:

```
private void insertIntoGeometryDictionary(IFMEOFeature pFeature)
{
    // We assume that all features that enter this function actually have
    // geometries - This will be enforced by readDataFeatures()

    // Get the geometry of the current feature
    IFMEOGeometry currGeometry = pFeature.GetGeometry();

    // Get the type of the geometry - This is referred to as the GeometryClass
    FMEOGeometryClass currGeomClass = currGeometry.GeometryClass;

    // Next, we check for where to put the feature inside m_geometryDictionary

    // Check to see if an entry for currGeomClass already exist
    if (!m_geometryDictionary.ContainsKey(currGeomClass))
    {
        // We must create a new entry to the m_geometryDictionary
        IFMEOFeatureVectorOnDisk newVectorOnDisk =
            m_fmeSession.CreateFeatureVectorOnDisk(m_kMaxFeatureInMemory);

        // Add the new entry to the dictionary
        m_geometryDictionary.Add(currGeomClass, newVectorOnDisk);
    }

    // An entry exist, so we get its corresponding IFMEOFeatureVectorOnDisk
    // and add the new feature to it
    IFMEOFeatureVectorOnDisk currVectorOnDisk = m_geometryDictionary[currGeomClass];
    currVectorOnDisk.Append(pFeature);
}
```

As you may have noticed, this method is very similar to the `insertIntoFeatureTypeDictionary` method that was defined in the previous chapter. The major difference is that entries are now added to `m_geometryDictionary` instead. In addition, the geometry is first retrieved and then the `GeometryClass` property is used for retrieving the value.

4. Declare a new method named `disposeGeometryDictionaryEntries` as follows:

```
private void disposeGeometryDictionaryEntries()
{
    IEnumerator<KeyValuePair<FMEOGeometryClass, IFMEOFeatureVectorOnDisk>> iterator =
        m_geometryDictionary.GetEnumerator();

    // Iterate through each entry m_geometryDictionary, and clear each
    // IFMEOFeatureVectorOnDisk
    while (iterator.MoveNext())
    {
        FMEOGeometryClass currGeomClass = iterator.Current.Key;
        IFMEOFeatureVectorOnDisk currVectorOnDisk = iterator.Current.Value;

        // Calling Clear of IFMEOFeatureVectorOnDisk should also dispose each of the features
        // that it contains
        currVectorOnDisk.Clear();
        currVectorOnDisk.Dispose();
        currVectorOnDisk = null;
    }

    // Finally, call Clear of m_geometryDictionary
    m_geometryDictionary.Clear();
}
```

This method is used to dispose all the features that are stored inside the `m_GeometryDictionary`.

- Now, add the call to `disposeGeometryDictionaryEntries` inside `resetAllDictionaries`.

```
private void resetAllDictionaries()
{
    // Dispose all the features inside m_featureTypeDictionary
    disposeFeatureTypeDictionaryEntries();

    // Dispose all the features inside m_geometryDictionary
    disposeGeometryDictionaryEntries();
}
```

- Now, with all the methods defined, you can make the appropriate call inside the `readDataFeatures`. Update `readDataFeatures` as follows:

```
private void readDataFeatures(IFMEORReader fmeReader, ref int featureCount)
{
    ...

    // Now, read all the different features
    IFMEOFeature fmeFeature = m_fmeSession.CreateFeature();
    while (fmeReader.Read(fmeFeature))
    {
        // Log the feature to the log file
        m_fmeLogFile.LogFeature(fmeFeature, FMEOMessageLevel.Inform, -1);

        // Attempt to retrieve the geometry
        IFMEOGeometry featureGeometry = fmeFeature.GetGeometry();

        // If that feature has a geometry
        if (featureGeometry != null)
        {
            IFMEOFeature fmeFeatureCopy = m_fmeSession.CreateFeature();
            fmeFeature.Clone(fmeFeatureCopy);

            insertIntoGeometryDictionary(fmeFeatureCopy);

            featureGeometry.Dispose();
        }

        // For every feature, we will put it into m_featureTypeDictionary
        insertIntoFeatureTypeDictionary(fmeFeature);
    }
    ...
}
```

Note that we have the check to ensure that only those features with geometries get added to the geometry dictionary. As well, we must create a second copy of the feature through the `Clone` method before passing it into `insertIntoGeometryDictionary`. This is because the `IFMEOFeatureVectorOnDisk` will take ownership of the feature that it appends (done inside `insertIntoGeometryDictionary`).

Testing your changes

Build the solution and verify that the application still successfully compiles.

Displaying the Results Inside ListView

In the previous exercise, you processed every feature being read in and stored it inside `m_geometryDictionary`. In this exercise, you will display the results inside the `ListView` component of the Geometry tab.

1. Inside `Form1.cs`, add a new case statement for updating the contents of the Geometry tab:

```
private void refreshCurrentTab()
{
    eTabPanels selectedPanel = (eTabPanels) tabControl1.SelectedIndex;

    // Update the corresponding tab
    switch(selectedPanel)
    {
        case eTabPanels.FeatureTypeTab:
            updateFeatureTypeTab();
            break;
        case eTabPanels.GeometryTab:
            updateGeometryTab();
            break;
        default:
            break;
    }
}
```

2. Now, declare the `updateGeometryTab` method that will be used to update the contents of the Geometry tab.

```
private void updateGeometryTab()
{
    // Clear the current entries in the geometryView
    geometryView.Items.Clear();

    // Put all the possible geometry classes into a sorted list so that we
    // can iterate through each one
    SortedList<FMEOGGeometryClass, string> geometryColl =
        new SortedList<FMEOGGeometryClass, string>();

    geometryColl.Add(FMEOGGeometryClass.IFMEONull, "Null");
    geometryColl.Add(FMEOGGeometryClass.IFMEODonut, "Donut");
    geometryColl.Add(FMEOGGeometryClass.IFMEOEllipse, "Ellipse");
    geometryColl.Add(FMEOGGeometryClass.IFMEOPolygon, "Polygon");
    geometryColl.Add(FMEOGGeometryClass.IFMEOPath, "Path");
    geometryColl.Add(FMEOGGeometryClass.IFMEOArc, "Arc");
    geometryColl.Add(FMEOGGeometryClass.IFMEOLine, "Line");
    geometryColl.Add(FMEOGGeometryClass.IFMEOPoint, "Point");
    geometryColl.Add(FMEOGGeometryClass.IFMEOText, "Text");
    geometryColl.Add(FMEOGGeometryClass.IFMEOAggregate, "Aggregate");
    geometryColl.Add(FMEOGGeometryClass.IFMEOMultiArea, "MultiArea");
    geometryColl.Add(FMEOGGeometryClass.IFMEOMultiCurve, "MultiCurve");
    geometryColl.Add(FMEOGGeometryClass.IFMEOMultiPoint, "MultiPoint");
    geometryColl.Add(FMEOGGeometryClass.IFMEOMultiText, "MultiText");

    // Get an iterator for the sorted list
    IEnumerable<KeyValuePair<FMEOGGeometryClass, string>> iterator =
        geometryColl.GetEnumerator();
```



```

while (iterator.MoveNext())
{
    // Create a new subItem list - it will be a row of data containing
    // the geometry class and its associated count
    string [] subItemList = new string[2];
    subItemList[0] = iterator.Current.Value;

    FMEOGeometryClass currGeomClass = iterator.Current.Key;

    // Check if the entry for currGeomClass exist inside m_geometryDictionary. If the entry
    // does not exist, it means that there wasn't any feature that had that geometry class
    // - in that case, we can display its count to be zero
    if (m_geometryDictionary.ContainsKey(currGeomClass))
    {
        IFMEOFeatureVectorOnDisk currVectorOnDisk = m_geometryDictionary[currGeomClass];
        subItemList[1] = currVectorOnDisk.Count.ToString();
    }
    else
    {
        subItemList[1] = "0";
    }

    // Create a new item from the subItem
    ListViewItem newItem = new ListViewItem(subItemList,-1);

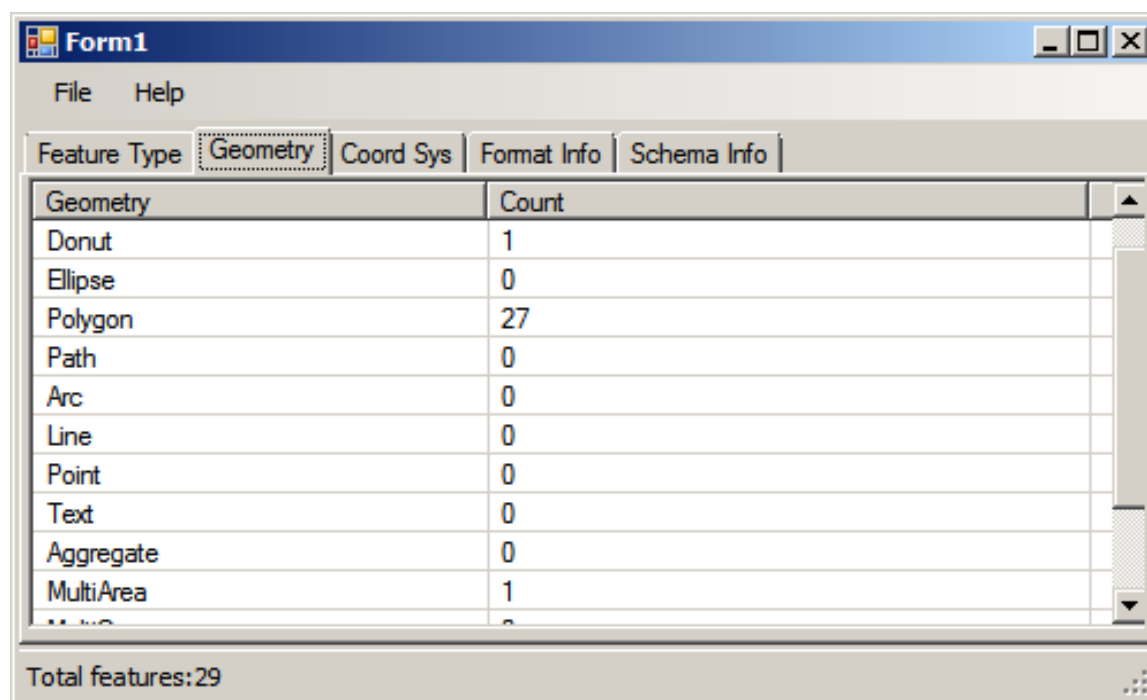
    // Add the item to geometryView
    geometryView.Items.Add(newItem);
}
}

```

A StringCollection is constructed at the beginning of the method to hold all the possible values of the GeometryClass property. Inside the loop, each of the values is used as keys to the m_geometryDictionary. If the entry was not found, it means that the dataset does not contain any features with that geometry class. Otherwise, the Count property of IFMEOVectorOnDisk is used to determine the correct number of features.

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open schoolRegions.gml (with GML - Geography Markup Language as format), and then select the FME Type tab to see the populated entries of the table.



The screenshot shows a window titled "Form1" with a menu bar containing "File" and "Help". Below the menu bar is a tabbed interface with four tabs: "Feature Type", "Geometry", "Coord Sys", "Format Info", and "Schema Info". The "Geometry" tab is currently selected. The main area of the window displays a table with two columns: "Geometry" and "Count". The table lists various geometry types and their corresponding counts. A vertical scrollbar is visible on the right side of the table. At the bottom of the window, a status bar displays the text "Total features:29".

Geometry	Count
Donut	1
Ellipse	0
Polygon	27
Path	0
Arc	0
Line	0
Point	0
Text	0
Aggregate	0
MultiArea	1

Total features:29

Chapter 5

Retrieving Coordinate System Information

In this chapter

Placing a ListView and ComboBox on the Coord Sys tab
Populating the entries of the ComboBox
Getting the parameters of a coordinate system

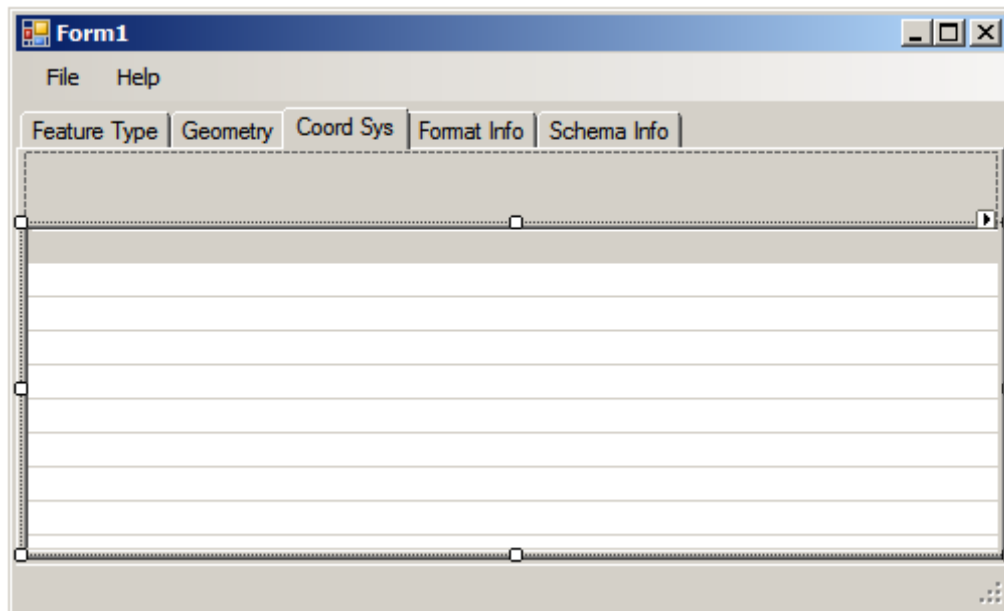
Objective

All FME Objects feature geometries are associated with a coordinate system. In this chapter, you will learn to retrieve the name of the coordinate system associated with a particular feature. Then, using IFMEOCoordinateSystemManager, you will be able to extract the specific parameters of a coordinate system, and display the information inside a ListView component.

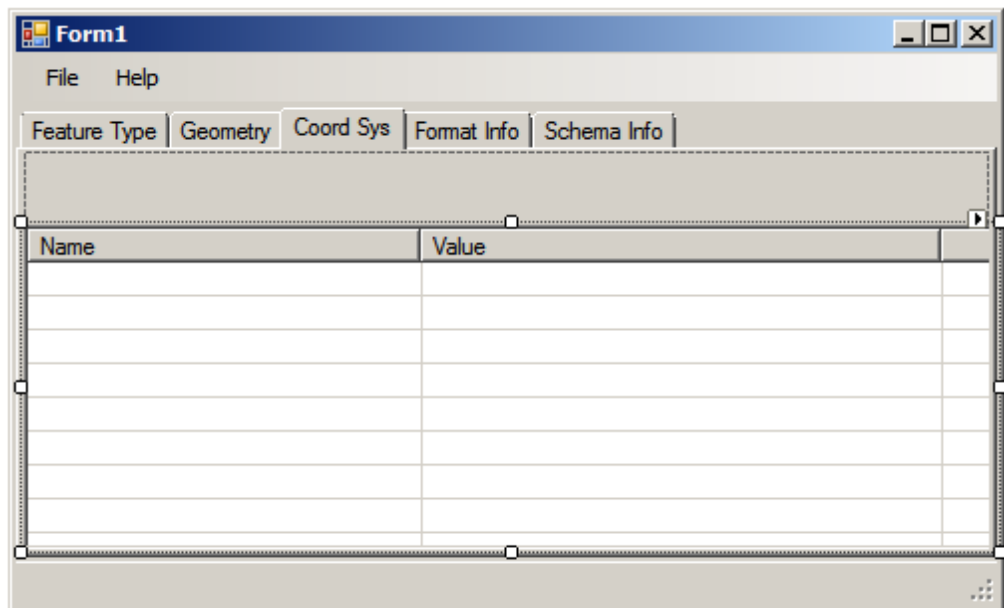
Placing a ListView and ComboBox on the Coord Sys Tab

In this exercise, you will place a ListView component on the Coord Sys tab (similar to Chapter 3). In addition, you will also place a ComboBox component which will be populated with the different feature types of the dataset.

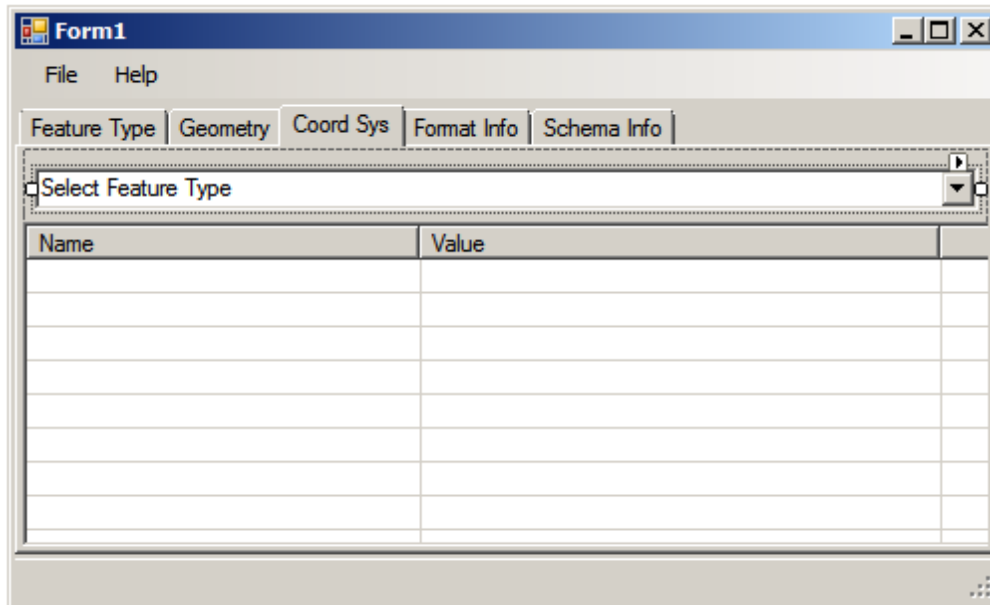
1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter4 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Select the Coord Sys tab to be in focus.
3. From the Toolbox, drag an instance of ListView to the sample application. On its properties, change its name from listView1 to coordSysView. Set FullRowSelect and GridLines to True, as well as the View to be Details and MultiSelect to False.
4. Adjust the size of the ListView to look like the following:



- Find the Columns attribute and click on the browse button to bring up the ColumnHeader Collection Editor dialog.
- Click on Add to add a new column. Under its properties, set Misc > Text to Name.
- Repeat the step above to add another column Value. Then, click Ok to close the dialog.



8. With the Coord Sys tab still in focus, drag an instance of ComboBox from the Toolbox to the sample application. Rename it to featureTypeComboBox, and set its text property to be Select Feature Type.
9. Adjust the size of the ComboBox to look like the following:



Testing your changes

Build the solution and run the application. The ListView and ComboBox components should appear under the Coord Sys tab.

Populating the Entries of the Feature Type ComboBox

In this exercise, you will populate the entries of featureTypeComboBox with the different feature types represented in the dataset. Most formats are limited to one coordinate system per dataset, however some allow different coordinate system per individual feature types. Hence, using a drop box is a good means for allowing a user to navigate between different coordinate systems present.

1. The featureTypeComboBox should be populated with the current feature type entries any time that the Coord Sys tab is in focus. Inside the refreshCurrentTab method, add a new case statement as follows:

```
private void refreshCurrentTab()
{
    eTabPanels selectedPanel = (eTabPanels) tabControl1.SelectedIndex;
```

```

// Update the corresponding tab
switch(selectedPanel)
{
    case eTabPanels.FeatureTypeTab:
        updateFeatureTypeTab();
        break;
    case eTabPanels.GeometryTab:
        updateGeometryTab();
        break;
    case eTabPanels.CoordSysTab:
        updateCoordSysTab();
        break;
    default:
        break;
}
}

```

- Now, declare the updateCoordSysTab method as follows:

```

private void updateCoordSysTab()
{
    // Each feature type may have its own coordinate system, so we will provide an
    // entry for each feature type in the drop box for the user to select

    // Clear the current entries of the featureTypeDropBox
    featureTypeComboBox.Items.Clear();
    featureTypeComboBox.Enabled = true;

    IEnumerator<KeyValuePair<string, IFMEFeatureVectorOnDisk>> iterator =
        m_featureTypeDictionary.GetEnumerator();

    // Add an item to the featureTypeDropBox for every entry of m_featureTypeDictionary
    while (iterator.MoveNext())
    {
        string currFeatureType = iterator.Current.Key;
        featureTypeComboBox.Items.Add(currFeatureType);

        // If this is a homogenous dataset, we will disable the
        // dropbox and have the single feature type to be selected
        if (m_featureTypeDictionary.Count == 1)
        {
            featureTypeComboBox.Enabled = false;
            featureTypeComboBox.SelectedItem = currFeatureType;
        }
    }
}

```

The method iterates through each key of m_featureTypeDictionary (which represents the feature type) and adds it as a new entry to the featureTypeComboBox. If there is only one feature type in the dataset, the featureTypeComboBox is disabled, and the single feature type is in focus by default.

- Inside the openOption_Click event handler, insert the following code to reset the text of the featureTypeComboBox and the items inside coordSysView every time a new dataset is read in.

```

private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        ...
    }
}

```

```

if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
{
    ...

    // Clean up reader
    fmeReader.Close();
    fmeReader.Dispose();
    fmeReader = null;

    // Reset GUI components
    this.featureTypeComboBox.Text = "Select Feature Type";
    coordSysView.Items.Clear();

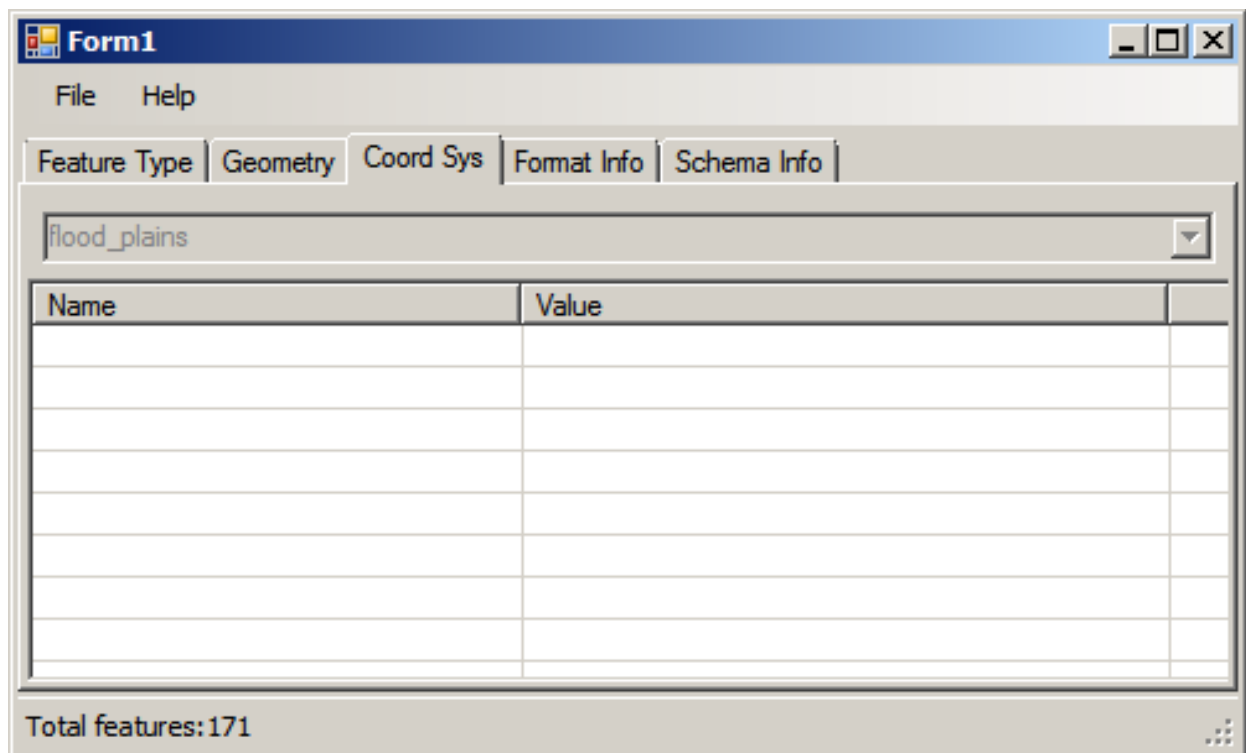
    // Enable the tab panels
    tabControl1.Enabled = true;

    // Refresh the current tab
    refreshCurrentTab();
}

```

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open flood_plains.mif (with MapInfo MIF/MID as the format) and select the Coord Sys tab. Notice that the dropbox is disabled for this dataset because only 1 unique feature type was found.



Getting the Parameters of a Coordinate System

The IFMEOCordinateSystemManager provides the ability to retrieve the specific parameters associated with a coordinate system (ie. projection, datum, ellipsoid, and units). In this exercise, you will obtain an instance of IFMEOCordinateSystemManager through IFMEOSession, and use it to display the coordinate system parameters on the screen.

1. Inside Form1.cs, define a new event handler named featureTypeComboBox_SelectedIndexChanged. This handler will be called whenever a user selects an item from the featureTypeComboBox.

```
private void featureTypeComboBox_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // Clear the current entries in the coordSysView
    coordSysView.Items.Clear();

    // Get the feature type that the user selected
    string selectedFeatureType = (string) featureTypeComboBox.SelectedItem;

    // Use selectedFeatureType to get its corresponding IFMEOFeatureVectorOnDisk
    // inside m_featureTypeDictionary
    IFMEOFeatureVectorOnDisk currVectorOnDisk = m_featureTypeDictionary[selectedFeatureType];

    // At this point, the client owns a copy of this feature
    IFMEOFeature selectedFeature = currVectorOnDisk.GetAt(0);

    string coordSystemName = selectedFeature.CoordinateSystem;

    // Dispose the feature since we owned it
    selectedFeature.Dispose();

    if (coordSystemName.Length != 0)
    {
        // Add code later
    }
}
```

This method retrieves the name of the feature type selected from featureTypeComboBox, and uses that name as the key to get the corresponding IFMEOFeatureVectorOnDisk from m_featureTypeDictionary. Since all features of a feature type have the same coordinate system, you can use any one of the features to examine its coordinate information. The code above uses the feature that is at the head of IFMEOFeatureVectorOnDisk.

Notice that the GetAt method of IFMEOFeatureVectorOnDisk gives the caller ownership of the IFMEOFeature that is returned. Therefore, one must dispose the feature accordingly when it is no longer needed.

2. With the name of the coordinate system, you can use IFMEOCordinateSystemManager to retrieve the specific parameters. Inside the conditional check of the event handler, insert the code as follows:

```
if (coordSystemName.Length != 0)
{
    StringCollection coordParams = new StringCollection();

    // Get the parameters associated with the coordinate system
```



```

IFMEOCordinateSystemManager coordManager = m_fmeSession.CoordinateSystemManager();
coordManager.GetCoordinateSystem(coordSystemName, coordParams);

// Iterate through every name-value pair inside coordParams
for (int counter = 0; (counter+1) < coordParams.Count; counter += 2)
{
    string newKey = coordParams[counter];
    string newValue = coordParams[counter+1];

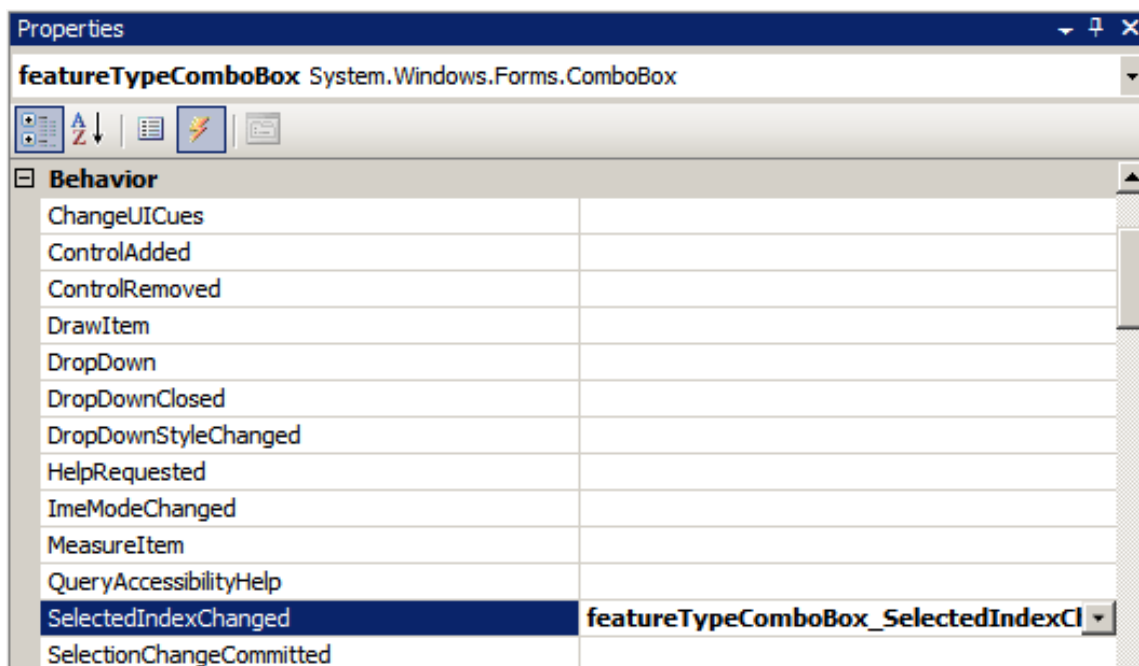
    // Create a new subItem list - a row containing the coord sys parameter name
    // and its associated value
    string [] subItems = {newKey,newValue};
    ListViewItem newItem = new ListViewItem(subItems,-1);

    // Add the new item to coordSysView
    coordSysView.Items.Add(newItem);
}
}

```

An instance of `IFMEOCordinateSystemManager` can be obtained through the `IFMEOSession` object. The `GetCoordinateSystem` method of `IFMEOCordinateSystemManager` gets the parameters associated with the coordinate system name, and stores the information inside the `StringCollection`. Since the parameter name and value are stored as consecutive entries inside the `StringCollection`, a loop is used to create a new `ListViewItem` for each name/value pair, and the information is added to `coordSysView`.

- Now, the new handler must be attached to the correct event. Switch to the Designer view, and select the `ComboBox` component (`featureTypeComboBox`) from the `Coord Sys` tab. Under `Events`, assign `featureTypeComboBox_SelectedIndexChanged` to the `SelectedIndexChanged` event.



Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open flood_plains.mif and select the Coord Sys tab. You should see the coordinate system parameters populate inside the ListView component.

The screenshot shows a Windows-style application window titled "Form1". It has a menu bar with "File" and "Help". Below the menu bar are five tabs: "Feature Type", "Geometry", "Coord Sys", "Format Info", and "Schema Info". The "Coord Sys" tab is currently selected. Below the tabs is a text box containing "flood_plains". Underneath the text box is a table with two columns: "Name" and "Value". The table contains the following data:

Name	Value
DESC_NM	NAD83 Texas State Planes, Central Zone, US Foot
SOURCE	Calculated from TX83-C by Mentor Software
DT_NAME	NAD83
LOCATION	
PARM14	0
ZERO_X	0.001
HGT_LAT	0
HGT_ZZ	0
PARM15	0

At the bottom of the window, there is a status bar that says "Total features: 171".

Chapter 6

Retrieving Format Information

In this chapter

Placing a ListView on the Format Info tab
Getting the parameters of a format

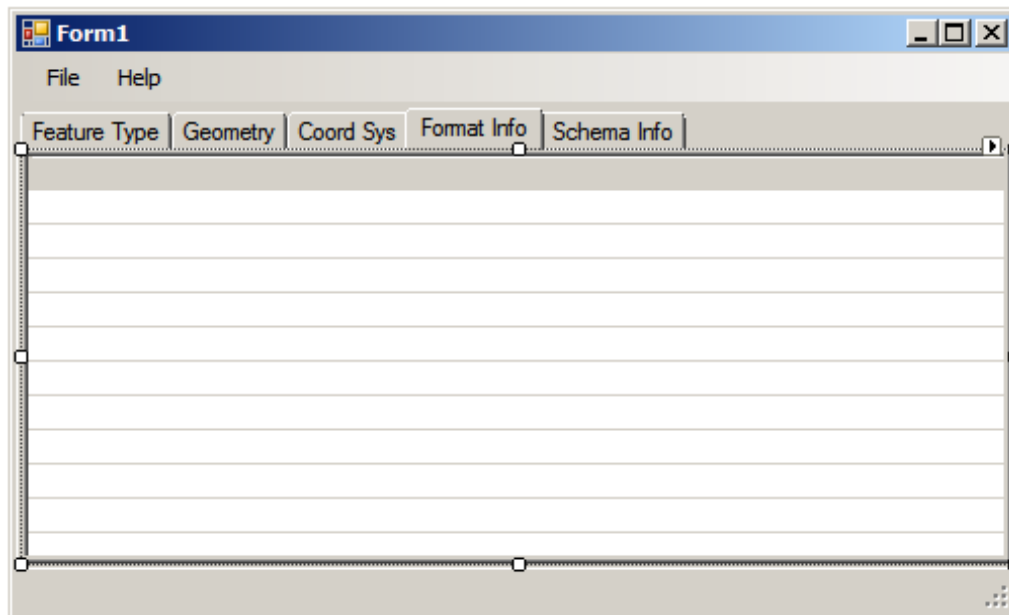
Objective

From the Source Data dialog prompt (when you select Open... from the File menu), you can click on the browse button beside Format to bring up the Formats Gallery. From the Formats Gallery, you can view specific information about a format such as its description, short name, and extension. IFMEODialog also has a method named GetFormatInfoEx that can return additional information about a format such as its associated Delphi/Windows filter and generic translation ability. In this chapter, you will learn to use the GetFormatInfoEx method to retrieve the detailed information for a format.

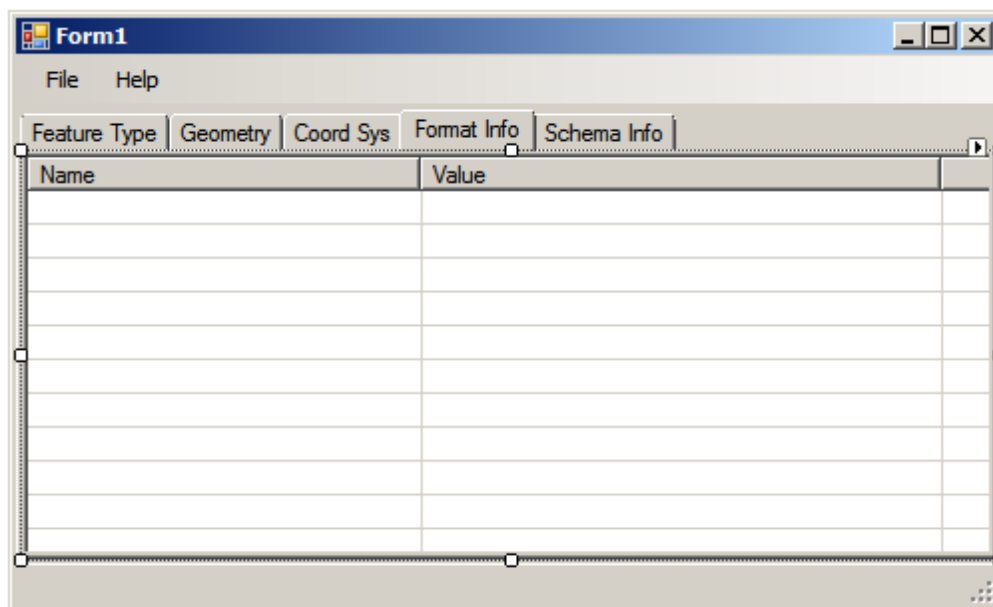
Placing a ListView on the Format Info Tab

In this exercise, you will place a ListView component on the Format Info tab.

1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter5 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Select the Format Info tab to be in focus.
3. From the Toolbox, drag an instance of ListView to the sample application. On its properties, change its name from listView1 to formatInfoView. Set FullRowSelect and GridLines to True, as well as the View to be Details and MultiSelect to False.
4. Adjust the size of the ListView to look like the following:



- Find the Columns attribute and click on the browse button to bring up the ColumnHeader Collection Editor dialog.
- Click on Add to add a new column. Under its properties, set Misc > Text property to be Name.
- Repeat the step above to add another column Value. Then, click Ok to close the dialog.



Testing your changes

Build the solution and run the application. The ListView component should now appear under the Format Info tab.

Getting the Parameters of a Format

In this exercise, you will use the GetFormatInfoEx method of IFMEODialog to retrieve format parameters.

Inside the event handler openOption_Click, you may recall that an instance of IFMEODialog was created for displaying the Source Data dialog. Once a user has selected a format and dataset from the dialog box, you will save the format parameters into a StringCollection such that the information can be accessed for later use.

1. Inside the refreshCurrentTab method of Form1.cs, add a new case statement as follows for updating the Format Info tab:

```
private void refreshCurrentTab()
{
    eTabPanels selectedPanel = (eTabPanels) tabControl1.SelectedIndex;

    // Update the corresponding tab
    switch(selectedPanel)
    {
        case eTabPanels.FeatureTypeTab:
            updateFeatureTypeTab();
            break;
        case eTabPanels.GeometryTab:
            updateGeometryTab();
            break;
        case eTabPanels.CoordSysTab:
            updateCoordSysTab();
            break;
        case eTabPanels.FormatInfoTab:
            updateFormatTab();
            break;
        default:
            break;
    }
}
```

2. Now, define the new event handler updateFormatTab which will be used to update the contents of the Format Info tab.

```
private void updateFormatTab()
{
    // Clear the current entries in the formatInfoView
    formatInfoView.Items.Clear();

    StringCollection formatParams = new StringCollection();
    // Retrieve the format parameters through the dialog
    m_fmeDialog.GetFormatInfoEx(m_dataInfo.Format, formatParams);

    // Iterate through every name-value pair inside m_formatParams
    for (int counter = 0; (counter+1) < formatParams.Count; counter += 2)
    {
        string currParam = formatParams[counter];
```

```

string currValue = formatParams[counter+1];

// Create a new subItem list - a row containing the format sys param
// and its associated value
string [] subItems = {currParam,currValue};
ListViewItem newItem = new ListViewItem(subItems,-1);

// Add the new item to formatInfoView
formatInfoView.Items.Add(newItem);
}
}

```

As you can see, the GetFormatInfoEx method of IFMEODialog takes in a format identifier as an argument and stores the output format parameters inside a StringCollection.

The name/value pair of each format parameter is stored as consecutive entries inside the StringCollection, so a loop is used to create a new ListViewItem for each pair, and the information is added to formatInfoView.

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open schoolRegions.gml (with GML - Geography Markup Language as format) and select the Format Info tab. You should see 8 entries of parameter name/value pairs inside the ListView component.

Name	Value
FORMAT_LONG_NAME	GML - Geography Markup Language
DATASET_TYPE	FILE
INPUT_OUTPUT	BOTH
FILE_EXTENSIONS	GML Files(*.gml;*.xml;*.gml.gz) *.gml;*.xml;*.gml.gz ...
COORD_SYSTEM_AWARE	YES
SOURCE_SETTINGS	YES
DESTINATION_SETTINGS	NO
AUTOMATED_TRANSLATION	BOTH

Total features:29

Chapter 7

Retrieving Schema Information

In this chapter

Placing a ListView and schema feature ComboBox on the Schema Info tab

Reading in schema features from a dataset

Populating the entries of the schema feature ComboBox

Displaying the metadata of schema features

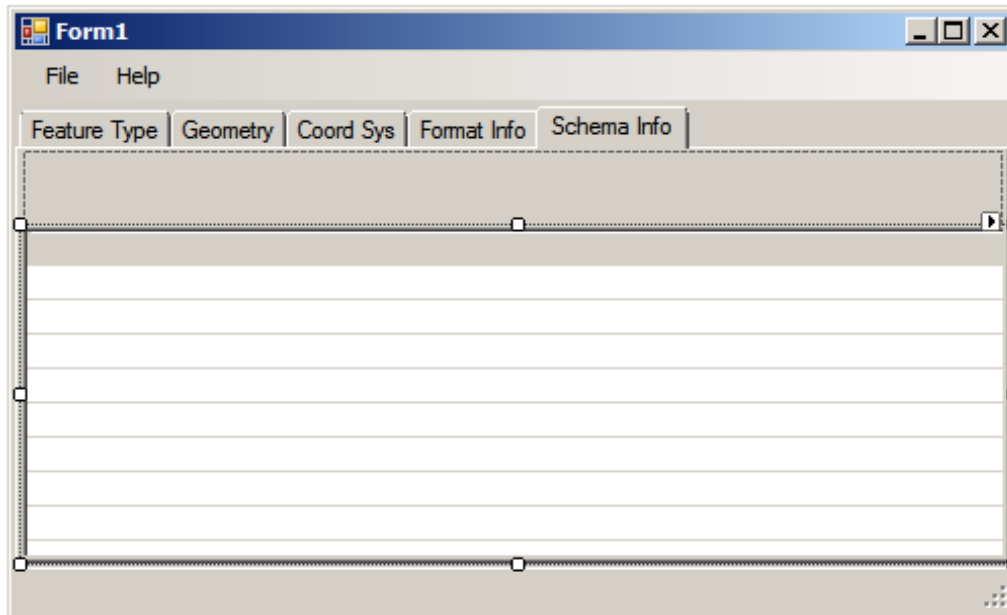
Objective

Schema features are useful when you need to understand the structure of input data or when you want to define a new structure or change the existing structure of data being written out. Schema features provide metadata on geometry, attribution, and coordinate system. In this chapter, you will be working with schema features to display the metadata of each feature type.

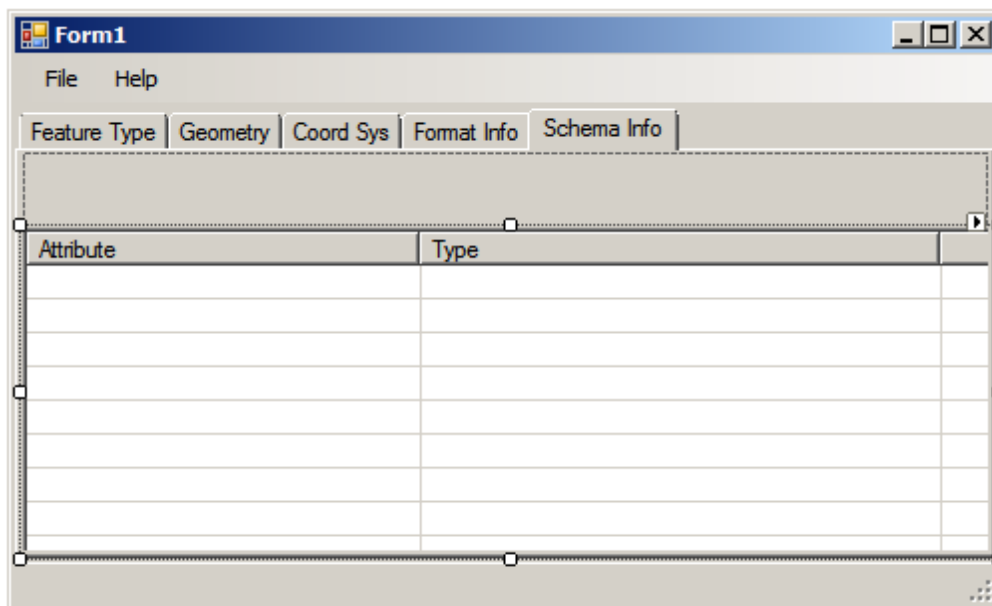
Placing a ListView and Schema Feature ComboBox on the Schema Info Tab

In this exercise, you will place a ListView component on the Schema Info tab. In addition, you will also place a ComboBox component which will be populated with the different schema features of the dataset.

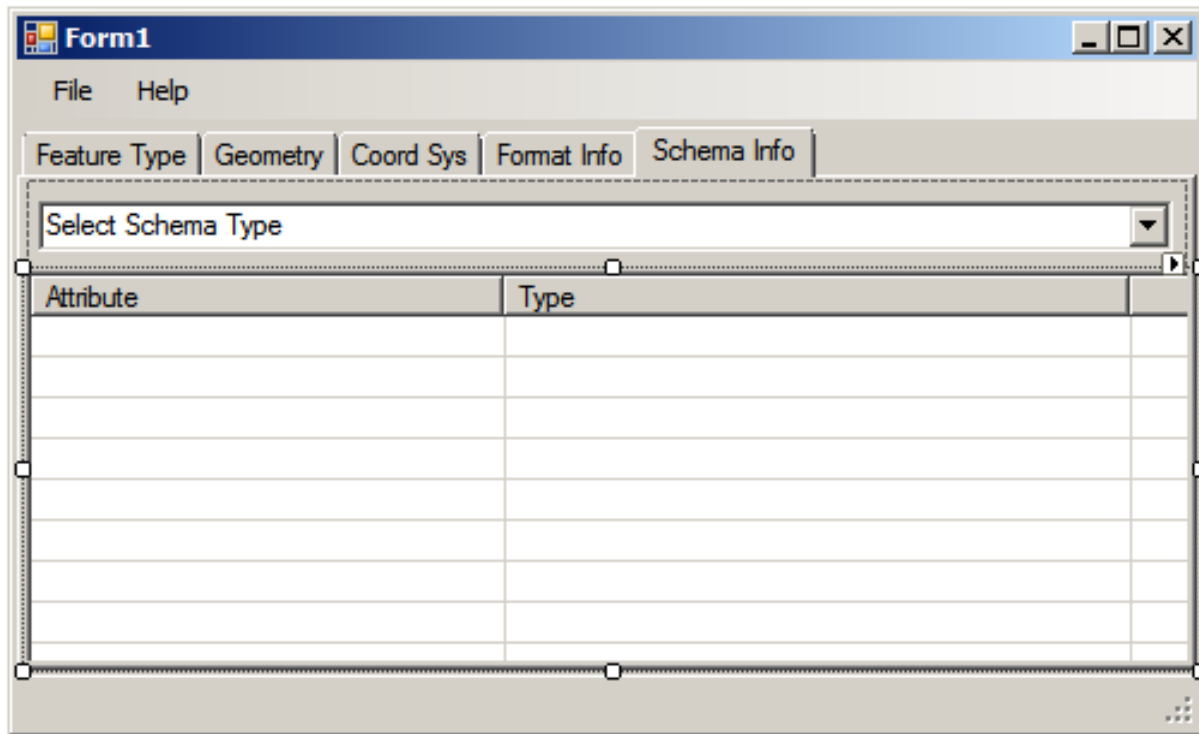
1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter6 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Select the Schema Info tab to be in focus.
3. From the Toolbox, click on ListView and drag out an instance of it to the sample application. On the properties box, rename it from listView1 to schemaInfoView. Set FullRowSelect and GridLines to True, as well as the View to be Details and MultiSelect to False.
4. Adjust the size of the ListView to look like the following:



- Find the Columns attribute and click on the browse button to bring up the ColumnHeader Collection Editor dialog.
- Click on Add to add a new column. Under its properties, set Misc > Text property to be Attribute.
- Repeat the step above to add another column Type. Then, click Ok to close the dialog.



8. With the Schema Info tab still in focus, drag out an instance of ComboBox from the Toolbox to the sample application. On its properties, rename it to schemaFeatureComboBox, and set its text property to be Select Schema Feature.
9. Adjust the size of the ComboBox to look like the following:



Testing your changes

Build the solution and run the application. The ListView and ComboBox components should now appear under the Schema Info tab.

Reading in Schema Features from a Dataset

In this exercise, you will modify the `openOption_Click` handler to read in schema features from a dataset (instead of just the data features). Schema features are read in the same way as data features, but the `ReadSchema` method of `IFMEORReader` is used instead. The schema features will be stored inside a dictionary for future access.

1. Inside `Form1.cs`, declare a new member private variable named `m_schemaDictionary` of type `SortedList<string, IFMEOFeature>` and set its initial reference to `null`.

```
private SortedList<string, IFMEOFeature> m_schemaDictionary = null;
```

2. Inside the class constructor, initialize `m_schemaDictionary` with a new instance of `SortedList<string, IFMEOFeature>`.

```

public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
    ...

    m_featureTypeDictionary = new SortedList<string, IFMEOFeatureVectorOnDisk>();
    m_geometryDictionary = new SortedList<FMEOGeometryClass, IFMEOFeatureVectorOnDisk>();
    m_schemaDictionary = new SortedList<string, IFMEOFeature>();

    // Disable the tab panels
    tabControl1.Enabled = false;
}

```

3. Declare a new method named `readSchemaFeatures` that will read schema features from a dataset:

```

private void readSchemaFeatures(IFMEOReader fmeReader, ref int schemaFeatureCount)
{
    // Initialize counters
    int numSchemaFeatures = 0;

    // Now, read all the different features
    IFMEOFeature fmeSchemaFeature = m_fmeSession.CreateFeature();
    while (fmeReader.ReadSchema(fmeSchemaFeature))
    {
        // Increment feature count
        numSchemaFeatures++;

        string featureType = fmeSchemaFeature.FeatureType;

        // Add the schema feature to the dictionary
        m_schemaDictionary.Add(featureType, fmeSchemaFeature);

        // Create a new feature for the next read
        fmeSchemaFeature = m_fmeSession.CreateFeature();
    }
    // Dispose the last schema feature which was not added
    fmeSchemaFeature.Dispose();

    schemaFeatureCount = numSchemaFeatures;
}

```

Since a schema feature provides the data model for one particular feature type, the feature type is used as the key to `m_schemaDictionary`.

4. Inside the `openOption_Click` event handler, update the code to call `readSchemaFeatures`:

```

private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            ...

```

```

        // Open the reader
        StringCollection openParams = new StringCollection();
        fmeReader.Open(m_dataInfo.Dataset, openParams);

        int schemaFeatureCount = 0;
        // Now, read in the schema features
        readSchemaFeatures(fmeReader, ref schemaFeatureCount);

        int featureCount = 0;
        // Now, read in the data features
        readDataFeatures(fmeReader, ref featureCount);

        ...
    }

```

5. The schema features stored from the existing dataset should also be disposed when a new dataset has been read in. Declare a new method named `disposeSchemaDictionaryEntries` as follows:

```

private void disposeSchemaDictionaryEntries()
{
    IEnumerator<KeyValuePair<string, IFMEFeature>> iterator =
                                                m_schemaDictionary.GetEnumerator();

    // Iterate through each entry m_schemaDictionary, and dispose each
    // schema feature
    while (iterator.MoveNext())
    {
        string currKey = iterator.Current.Key;
        IFMEFeature currFeature = m_schemaDictionary[currKey];
        currFeature.Dispose();
    }

    // Finally, call Clear of m_schemaDictionary
    m_schemaDictionary.Clear();
}

```

6. Finally, inside the method `resetAllDictionaries`, insert the code as shown:

```

private void resetAllDictionaries()
{
    // Dispose all the features inside m_featureTypeDictionary
    disposeFeatureTypeDictionaryEntries();

    // Dispose all the features inside m_geometryDictionary
    disposeFmeTypeDictionaryEntries();

    // Dispose all the schema features inside m_schemaDictionary
    disposeSchemaDictionaryEntries();
}

```

Testing your changes

Build the solution and run the application. Open a dataset, and the application should still exhibit the same behavior. In the next exercise, you will be able to verify that schema features were read in correctly.

Populating the Entries of the Schema Feature ComboBox

In this exercise, you will populate the entries of schemaFeatureComboBox with all the different feature types of the dataset.

1. The entries inside the schemaFeatureComboBox should be refreshed any time that the Schema Info tab is in focus. Inside the refreshCurrentTab method, add a new case statement as follows:

```
private void refreshCurrentTab()
{
    eTabPanels selectedPanel = (eTabPanels) tabControl1.SelectedIndex;

    // Update the corresponding tab
    switch(selectedPanel)
    {
        ...
        case eTabPanels.FormatInfoTab:
            updateFormatTab();
            break;
        case eTabPanels.SchemaInfoTab:
            updateSchemaInfoTab();
            break;
        default:
            break;
    }
}
```

2. Now, declare the updateSchemaInfoTab as follows:

```
private void updateSchemaInfoTab()
{
    // Clear the current entries of the schemaFeatureComboBox
    schemaFeatureComboBox.Items.Clear();
    schemaFeatureComboBox.Enabled = true;

    IEnumerator<KeyValuePair<string, IFMEObject>> iterator =
                                                m_schemaDictionary.GetEnumerator();

    // Add an item to the schemaFeatureComboBox for every entry of m_schemaDictionary
    while (iterator.MoveNext())
    {
        string currKey = iterator.Current.Key;
        schemaFeatureComboBox.Items.Add(currKey);

        // If there is only one schema feature, we will disable the
        // dropdown and have the single schema feature to be selected
        if (m_schemaDictionary.Count == 1)
        {
            schemaFeatureComboBox.Enabled = false;
            schemaFeatureComboBox.SelectedItem = currKey;
        }
    }
}
```

The method iterates through each key of m_schemaDictionary (which represents the feature type) and adds it as a new entry to the schemaFeatureComboBox. If there is only one schema feature in the dataset, the schemaFeatureComboBox is disabled, and the single schema feature is in focus by default.

3. Inside the `openOption_Click` event handler, insert the following code to reset the text of the `schemaFeatureComboBox` and the items of `schemaInfoView` every time a new dataset is read in.

```
private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        ...

        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            ...

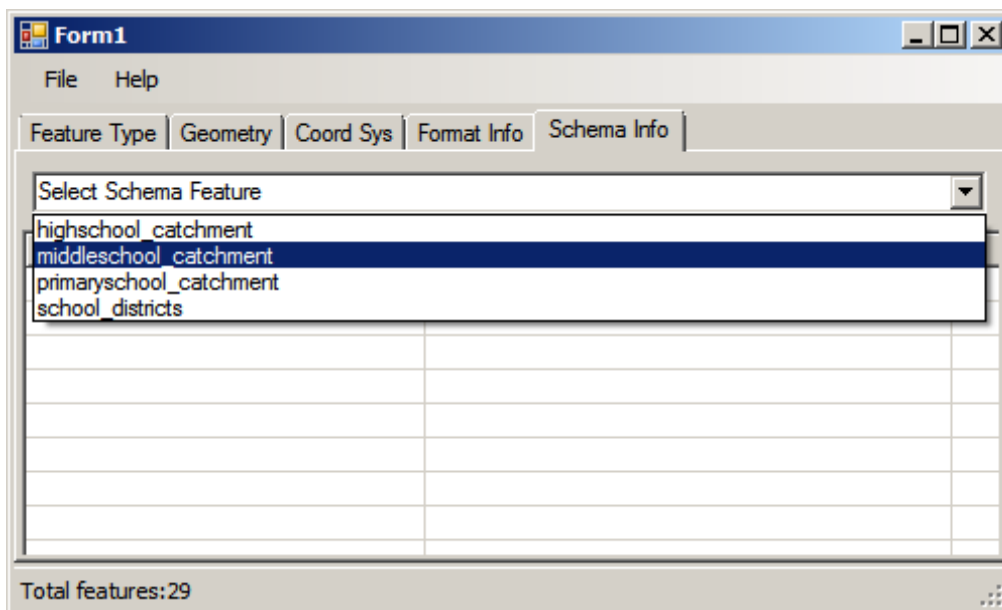
            // Reset GUI components
            this.featureTypeComboBox.Text = "Select Feature Type";
            this.schemaFeatureComboBox.Text = "Select Schema Feature";
            coordSysView.Items.Clear();
            schemaInfoView.Items.Clear();

            // Enable the tab panels
            tabControl1.Enabled = true;

            // Refresh the current tab
            refreshCurrentTab();
        }
        ...
    }
}
```

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open `schoolRegions.gml` (with GML - Geography Markup Language as format) and select the Schema Info tab. The dropbox should contain the feature types of all the schema features that were read in.



Displaying the Metadata of Schema Features

In this exercise, you will retrieve and display the metadata contained within each schema feature. As with data features, the list of attribute names can be obtained through the `GetAllAttributeNames` method of `IFMEOWFeature`. Similarly, the value associated with the attribute name can be retrieved through the `GetStringAttribute` method of `IFMEOWFeature`. However, whereas the values of data features reveal attribute information, values of schema features reveal attribute metadata. In other words, calling `GetStringAttribute` with an attribute name on a schema feature will return you the type of that attribute as a string (ie. "number(2,0)").

1. Inside `Form1.cs`, define a new event handler named `schemaFeatureComboBox_SelectedIndexChanged`. This handler will be called whenever a user selects a new item from the `schemaFeatureComboBox`, and will be responsible for displaying the metadata of the selected schema feature.

```
private void schemaFeatureComboBox_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // Clear the schemaInfoView of current entries
    schemaInfoView.Items.Clear();

    string selectedFeatureType = (string) schemaFeatureComboBox.SelectedItem;

    // Get the list of attributes for the selected feature type
    IFMEOWFeature schemaFeature = m_schemaDictionary[selectedFeatureType];
    StringCollection attrList = new StringCollection();
    schemaFeature.GetAllAttributeNames(attrList);

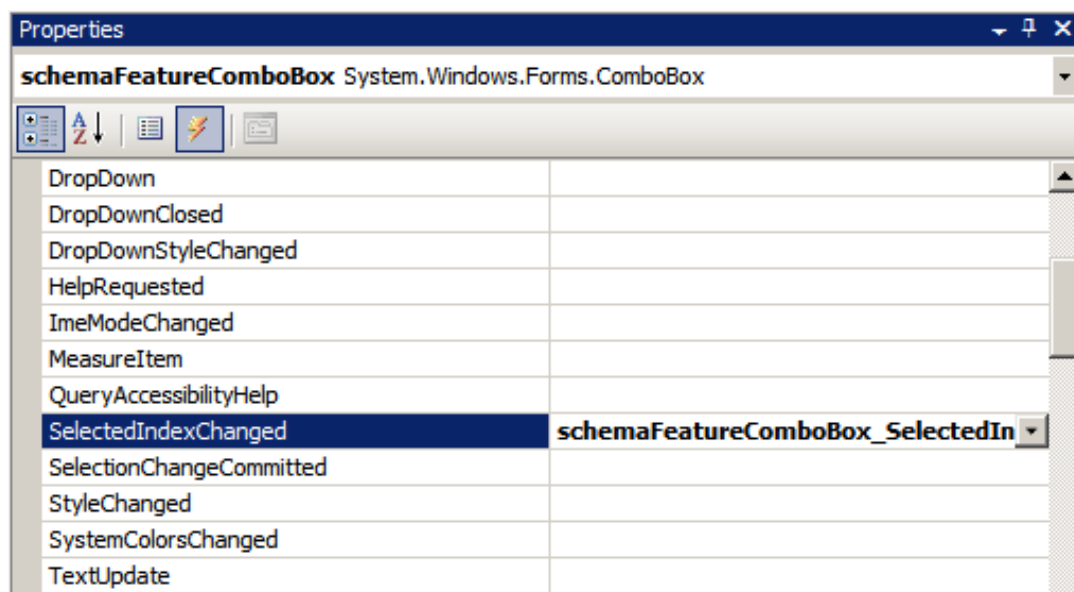
    // Iterate through each attribute name and get their type
    foreach (string currAttr in attrList)
    {
        string currAttrType = schemaFeature.GetStringAttribute(currAttr);

        // Create a new subItem list - a row containing the attribute name
        // and its type
        string [] subItems = {currAttr, currAttrType};
        ListViewItem newItem = new ListViewItem(subItems, -1);

        // Add the new item to formatInfoView
        schemaInfoView.Items.Add(newItem);
    }
}
```

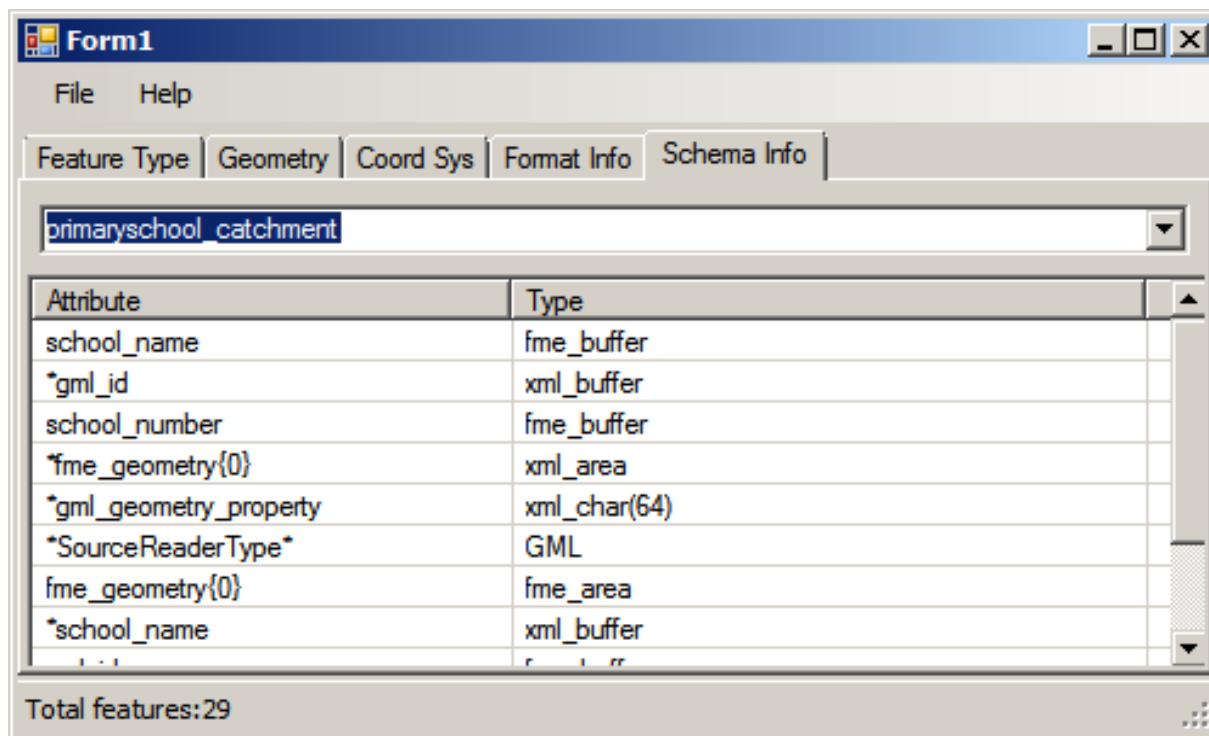
The method uses the name of the selected feature type as the key to obtain the corresponding schema feature inside `m_schemaDictionary`. Then a loop is used to iterate through each attribute of the schema feature, and its metadata is retrieved and displayed inside `schemaInfoView`.

2. Now, the new handler must be attached to the correct event. Switch to the Designer view, and select the `ComboBox` component (`schemaFeatureComboBox`) from the Schema Info tab. Under Events, assign `schemaFeatureComboBox_SelectedIndexChanged` to the `SelectedIndexChanged` event.



Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open `schoolRegions.gml` (with GML - Geography Markup Language as format) and select the Schema Info tab. Select a feature type from the dropbox and the metadata for the schema feature should be displayed.



Chapter 8

Saving Features to a Different Format

In this chapter

Using IFMEODialog to specify the destination dataset

Writing schema and data features through IFMEOWriter

Objective

Similar to reading datasets, you can use IFMEODialog to display an output data destination prompt for allowing the user to specify the format and location of the destination dataset. Then, that information can be used to configure an IFMEOWriter object for writing out the data features.

Using IFMEODialog to Specify the Destination Dataset

1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter7 folder.
2. Open Form1.cs from the Solution Explorer. Define a new event handler named saveAsOption_Click as shown. This method will be called when the user selects Save As... from the File menu.

```
private void saveAsOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.DestinationPrompt("", "", m_dataInfo, m_createDirectives))
        {
        }
    }

    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(),
                                     FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}
```



```
    }
}
```

Notice that the DestinationPrompt method of IFMEODialog is used instead of SourcePrompt to display the output destination data prompt to the user.

3. Declare a helper method named writeSchemaFeatures as follows:

```
private void writeSchemaFeatures(IFMEOWriter fmeWriter, ref int schemaFeatureCount)
{
    schemaFeatureCount = m_schemaDictionary.Count;

    IEnumerable<KeyValuePair<string, IFMEOWFeature>> schemaIterator =
        m_schemaDictionary.GetEnumerator();

    // Write out all schema features to the destination dataset
    while (schemaIterator.MoveNext())
    {
        string currKey = schemaIterator.Current.Key;
        IFMEOWFeature currSchemaFeature = m_schemaDictionary[currKey];

        // Write out the schema feature
        fmeWriter.AddSchema(currSchemaFeature);
    }
}
```

Writing features is a two-step process. First, the schema for each feature type must be supplied to the writer by using the AddSchema method of IFMEOWriter. In chapter 7, you may recall that all the schema features were already stored inside m_schemaDictionary. The code above iterates through each schema feature, and passes it to the writer using AddSchema.

4. Now, declare the helper method named writeDataFeatures that will actually write out the data features:

```
private void writeDataFeatures(IFMEOWriter fmeWriter, ref int featureCount)
{
    // We know that we have all the features stored inside IFMEOWFeatureVectorOnDisk
    // within entries of m_featureTypeDictionary. So we can just loop through each
    // entry inside m_featureTypeDictionary, grab its corresponding IFMEOWFeatureVectorOnDisk,
    // and write all those features inside that IFMEOWFeatureVectorOnDisk

    int numFeatures = 0;

    IEnumerable<KeyValuePair<string, IFMEOWFeatureVectorOnDisk>> featureIterator =
        m_featureTypeDictionary.GetEnumerator();

    while (featureIterator.MoveNext())
    {
        string currFeatureType = featureIterator.Current.Key;
        IFMEOWFeatureVectorOnDisk currVecOnDisk = m_featureTypeDictionary[currFeatureType];

        for (int counter = 0; counter < currVecOnDisk.Count; counter++)
        {
            // At this point, we take ownership of a copy of the feature on disk
            IFMEOWFeature currFeature = currVecOnDisk.GetAt(counter);

            // Write out the feature
            fmeWriter.Write(currFeature);

            // Increment feature count
            numFeatures++;
        }
    }
}
```

```

        // Check if we need to update status bar
        if ( (numFeatures % m_kUpdateInterval) == 0 )
        {
            updateStatusBar("Wrote " + numFeatures.ToString() + " features...");
        }

        // Dispose the feature since we owned it
        currFeature.Dispose();
    }
}

featureCount = numFeatures;
}

```

You may recall from chapter 3 that all the data features are already stored inside `m_featureTypeDictionary`, which maps a feature type to its corresponding `IFMEOWFeatureVectorOnDisk`. The code above writes out each feature of the dataset inside a nested loop. The outer loop iterates through each feature type of `m_featureTypeDictionary` and retrieves its `IFMEOWFeatureVectorOnDisk`. Then the inner loop is used to iterate through each feature of the `IFMEOWFeatureVectorOnDisk`, and the feature is written through the `Write` method of `IFMEOWriter`.

5. Finally, update the `saveAsOption_Click` handler to create the `IFMEOWriter` object, and to make the appropriate calls to `writeSchemaFeatures` and `writeDataFeatures`.

```

private void saveAsOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.DestinationPrompt("", "", m_dataInfo, m_createDirectives))
        {
            // Update the status bar
            updateStatusBar("Writing to destination...");

            // Create the writer
            IFMEOWriter fmeWriter = m_fmeSession.CreateWriter(m_dataInfo.Format,
                                                            m_createDirectives);

            // Open the writer
            StringCollection openParams = new StringCollection();
            fmeWriter.Open(m_dataInfo.Dataset, openParams);

            int schemaFeatureCount = 0;
            // Now, write out the schema features
            writeSchemaFeatures(fmeWriter, ref schemaFeatureCount);

            int featureCount = 0;
            // Now, write out the data features
            writeDataFeatures(fmeWriter, ref featureCount);

            // Update status bar with final feature count
            updateStatusBar("Total features written: " + featureCount.ToString());

            // Clean up the writer
            fmeWriter.Close();
            fmeWriter.Dispose();
            fmeWriter = null;
        }
    }
    ...
}

```

6. It only makes sense to allow a user to write out a dataset when an existing dataset has already been read. Rename the Save As... option under the File menu to saveAsOption. Inside the class constructor, insert the code as shown to disable the Save As... option at the beginning of the application.

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    ...

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
    ...

    // Disable the tab panels and Save As option
    tabControl1.Enabled = false;
    saveAsToolStripMenuItem.Enabled = false;
}
```


7. The Save As... option should be enabled when the first dataset has been read in. Inside openOption_Click, insert the code as shown:

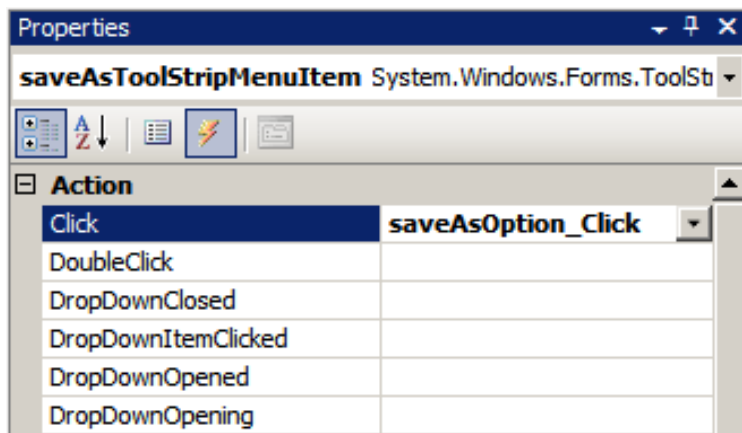
```
private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        ...

        if (m_fmeDialog.SourcePrompt("", "", sourceInfo, createDirectives))
        {
            ...

            // Enable the tab panels
            tabControl1.Enabled = true;
            saveAsToolStripMenuItem.Enabled = true;

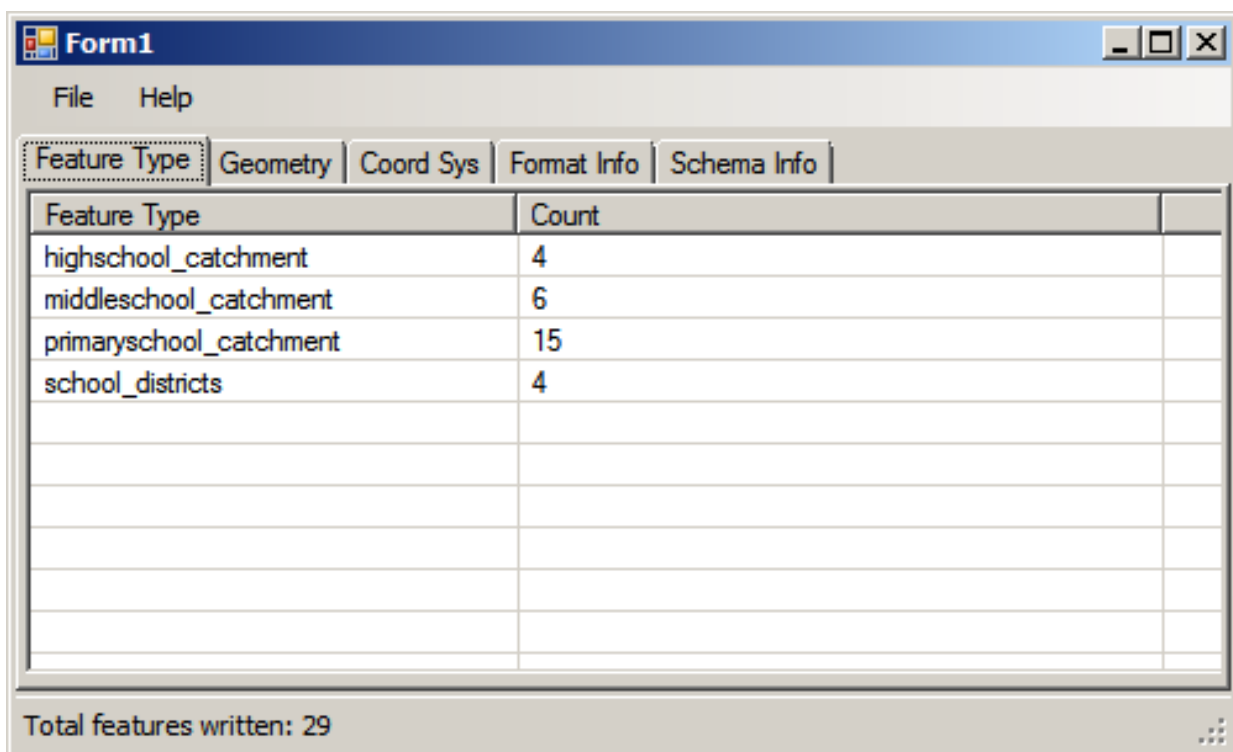
            ...
        }
    }
}
```

8. Now, attach the new handler to the corresponding event. Switch to the Designer view. On the sample application, click on File, and then select Save As..., and click on the Events button  to see its list of events. Assign the saveAsOption_click handler to the Click event. The application will now respond when a user selects Save As.... From the File menu.



Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open `schoolRegions.gml` (with GML - Geography Markup Language as format). Then select `Save As...` from the File menu. Specify a format and destination on the dialog and press `Ok` to begin writing.



Chapter 9

Rendering Features

In this chapter

Updating the Main Form to include a Rendering Panel

Manipulating Geometry Coordinates

Creating the RenderingVisitor Class

Using the Visitor Design Pattern to Facilitate Rendering of Feature Geometries

Displaying Feature Geometries on the Rendering Panel

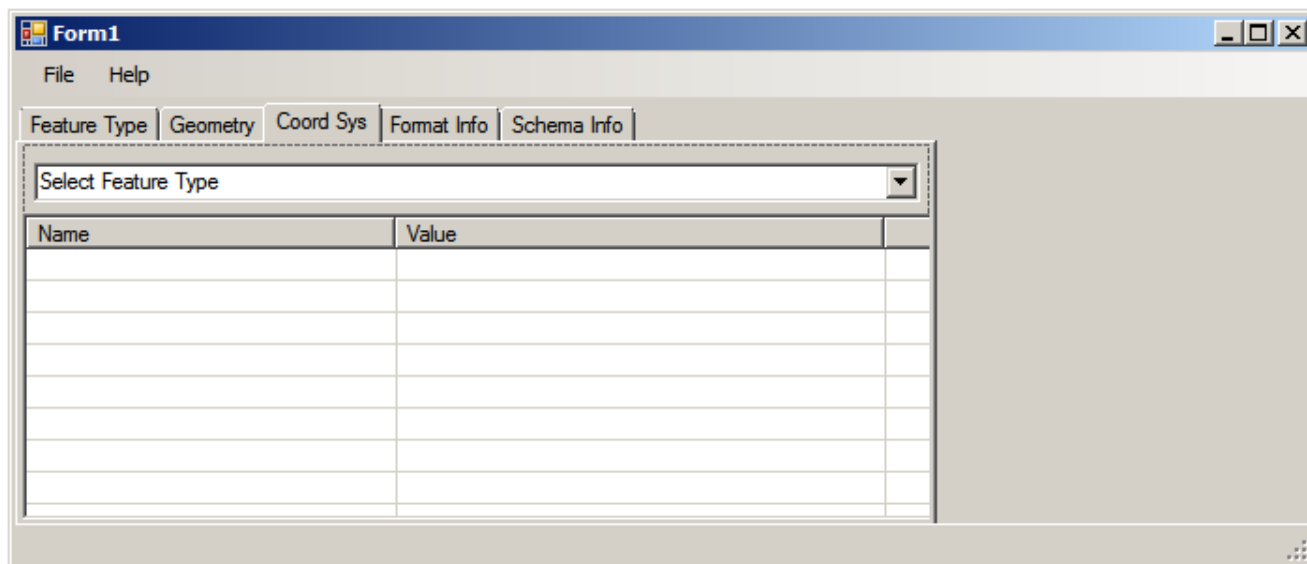
Objective

As shown earlier in Chapter 4, for a given IFMEOFeature, it is possible to have a IFMOGeometry associated with that feature. This chapter will take advantage of several different constructs in the FME Objects API to facilitate rendering of these geometries; in particular, the Visitor Design Pattern will be used to perform the actual rendering.

Updating the Main Form to include a Rendering Panel

In this exercise, you will update the main form to include an area for the rendered geometries.

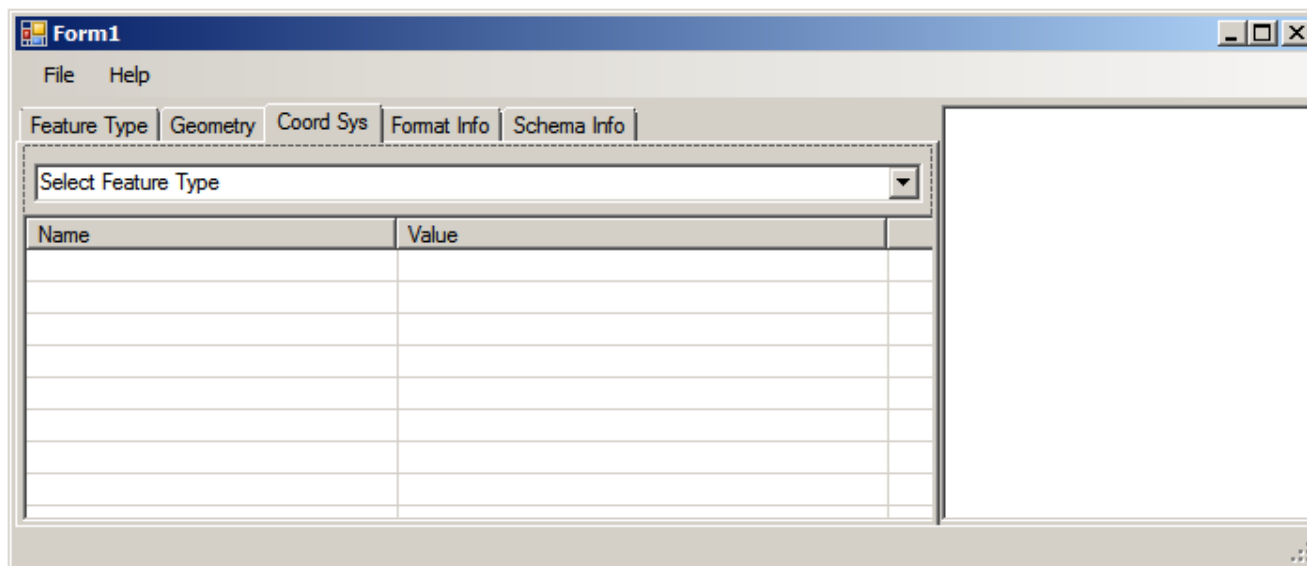
1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter8 folder.
2. Open Form1.cs from the Solution Explorer. Switch to the Designer view. Select the entire form to be in focus.
3. Select the main window form, and inside its properties display, increase the Size to 700,300. (This is just a guideline, the exact size is not important) The form should now look like the following:



The screenshot shows a window titled 'Form1' with a menu bar containing 'File' and 'Help'. Below the menu bar are five tabs: 'Feature Type', 'Geometry', 'Coord Sys', 'Format Info', and 'Schema Info'. The 'Feature Type' tab is selected. Inside this tab, there is a dropdown menu labeled 'Select Feature Type' with a downward arrow. Below the dropdown is a table with two columns: 'Name' and 'Value'. The table is currently empty.

Name	Value
------	-------

4. From the Toolbox, click on Panel and drag out an instance of it to the sample application. On the properties box, rename it from panel1 to renderingPanel. Set BorderStyle to be Fixed3D and BackColor to be White (Located under the Web tab). The form should now look like the following:



The screenshot shows the same 'Form1' window, but now with a large white rectangular area added to the right side of the 'Feature Type' tab. This area is the 'renderingPanel' mentioned in the instructions. The rest of the interface, including the menu bar, tabs, dropdown menu, and table, remains the same as in the previous screenshot.

Name	Value
------	-------

Testing your changes

Build the solution and run the application. The form should now be resized and the rendering panel should now appear on the right-hand side of form.

Manipulating Geometry Coordinates

In this exercise, you will be creating functions that will manipulate the coordinates of the various feature geometries. This is necessary so we can map the given coordinate (which could theoretically be in any coordinate system) to its respective screen coordinate.

1. Declare the helper method named `generateBoundingBox` that will be used to generate the bounding box for the geometries associated with a given set of features:

```
private IFMEORectangle generateBoundingBox(IFMEOFeatureVectorOnDisk features)
{
    // We iterate through the collection of features, keeping track of those coordinates
    // that are the most extreme values (eg. Largest X value, etc). We then use these
    // values to generate the bounding box for the collection of features.

    double minX = System.Double.MaxValue;
    double maxX = System.Double.MinValue;

    double minY = System.Double.MaxValue;
    double maxY = System.Double.MinValue;

    // Used for retrieving the bounding box for the individual geometries
    IFMEOPoint minPoint = m_fmeGeometryTools.CreatePoint();
    IFMEOPoint maxPoint = m_fmeGeometryTools.CreatePoint();

    for (int i = 0; i < features.Count; i++)
    {
        // More code to add later...
    }

    // Dispose of the points
    minPoint.Dispose();
    maxPoint.Dispose();
    minPoint = null;
    maxPoint = null;

    // If the bounding box is valid
    if ((minX != System.Double.MaxValue) && (minY != System.Double.MaxValue) &&
        (maxX != System.Double.MinValue) && (maxY != System.Double.MinValue))
    {
        // Return it
        return m_fmeSession.CreateRectangle(minX, maxX, minY, maxY);
    }
    else
    {
        return null;
    }
}
```

The method iterates through each feature of the given `IFMEOFeatureVectorOnDisk` and uses it to generate a `IFMEORectangle` that represents the bounding box that will encompass all of the features. If the bounding box could not be successfully created for the given feature vector, null will be returned.

2. Inside the for-loop of the `generateBoundingBox` method, add the following code:

```
private IFMEORectangle generateBoundingBox(IFMEOFeatureVectorOnDisk features)
{
    ...

    for (int i = 0; i < features.Count; i++)
    {
```

```

// At this point, we take ownership of a copy of the feature on disk
IFMEOFeature currFeature = features.GetAt(i);

// Attempt to retrieve the feature on the current feature
IFMEOGGeometry currGeometry = currFeature.GetGeometry();

// If there is a geometry present on the given feature
if (currGeometry != null)
{
    // Retrieve the bounds from the current geometry
    currGeometry.Bounds(minPoint, maxPoint);

    // If we have the smallest X value (and the coordinate is valid)
    if ((minPoint.X != FMEOUTilities.NotANumber) && (minPoint.X < minX))
    {
        minX = minPoint.X;
    }
    // If we have the smallest Y value (and the coordinate is valid)
    if ((minPoint.Y != FMEOUTilities.NotANumber) && (minPoint.Y < minY))
    {
        minY = minPoint.Y;
    }
    // If we have the largest X value (and the coordinate is valid)
    if ((maxPoint.X != FMEOUTilities.NotANumber) && (maxPoint.X > maxX))
    {
        maxX = maxPoint.X;
    }
    // If we have the largest Y value (and the coordinate is valid)
    if ((maxPoint.Y != FMEOUTilities.NotANumber) && (maxPoint.Y > maxY))
    {
        maxY = maxPoint.Y;
    }

    // Dispose the geometry
    currGeometry.Dispose();
}

// Dispose the feature since we owned it
currFeature.Dispose();
}

...
}

```

The given for-loop is the core portion of the generateBoundingBox method; It iterates through all of the features in the given IFMEOFeatureVectorOnDisk and retrieves the bounding box for the geometry associated with that feature (if one exists.) By comparing the various values of the geometry bounding box with the respective max or min value, the end result is that the bounding box of the entire set of features will be generated.

3. Now, declare the helper method named calculateScalingFactors that will generate the appropriate scaling factor for the coordinates of the geometries:

```

private void calculateScalingFactors(IFMEORectangle featureSetBoundingBox,
    int canvasHeight,
    int canvasWidth,
    ref double xScaleFactor,
    ref double yScaleFactor)
{
    // We use the bounding box for the given feature set as well as the dimensions of
    // the canvas to determine the scaling factors necessary to map the current coordinates
    // to screen coordinates.
}

```



```

double rangeX = featureSetBoundingBox.MaxX - featureSetBoundingBox.MinX;
double rangeY = featureSetBoundingBox.MaxY - featureSetBoundingBox.MinY;
// Compensating for zero-width ranges

// If both ranges are zero-width
if ((rangeX == 0) && (rangeY == 0))
{
    featureSetBoundingBox.MinX = featureSetBoundingBox.MinX - 0.5;
    featureSetBoundingBox.MinY = featureSetBoundingBox.MinY - 0.5;
    featureSetBoundingBox.MaxX = featureSetBoundingBox.MaxX + 0.5;
    featureSetBoundingBox.MaxY = featureSetBoundingBox.MaxY + 0.5;
    rangeX = 1;
    rangeY = 1;
}
// If only the X range is zero-width
else if (rangeX == 0)
{
    rangeX = (rangeY * 5) / 100;
    featureSetBoundingBox.MinX = featureSetBoundingBox.MinX - (rangeX / 2);
    featureSetBoundingBox.MaxX = featureSetBoundingBox.MaxX + (rangeX / 2);
}
// If only the Y range is zero-width
else if (rangeY == 0)
{
    rangeY = (rangeX * 5) / 100;
    featureSetBoundingBox.MinY = featureSetBoundingBox.MinY - (rangeY / 2);
    featureSetBoundingBox.MaxY = featureSetBoundingBox.MaxY + (rangeY / 2);
}

xScaleFactor = (canvasWidth / rangeX);
yScaleFactor = (canvasHeight / rangeY);
}

```

This method takes in the bounding box for a particular set of features as well as the dimensions for the canvas where those features are to be rendered on; using those parameters, it calculates what the scaling factors should be so that the features can be properly rendered.

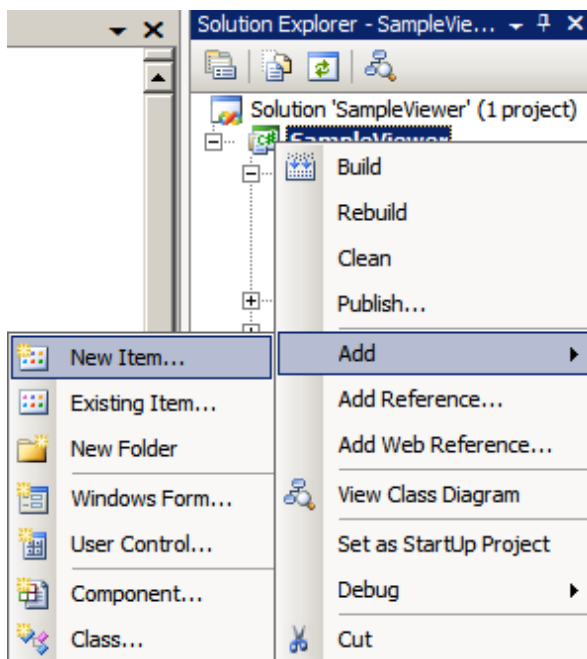
Testing your changes

Build the solution and verify that the application still successfully compiles.

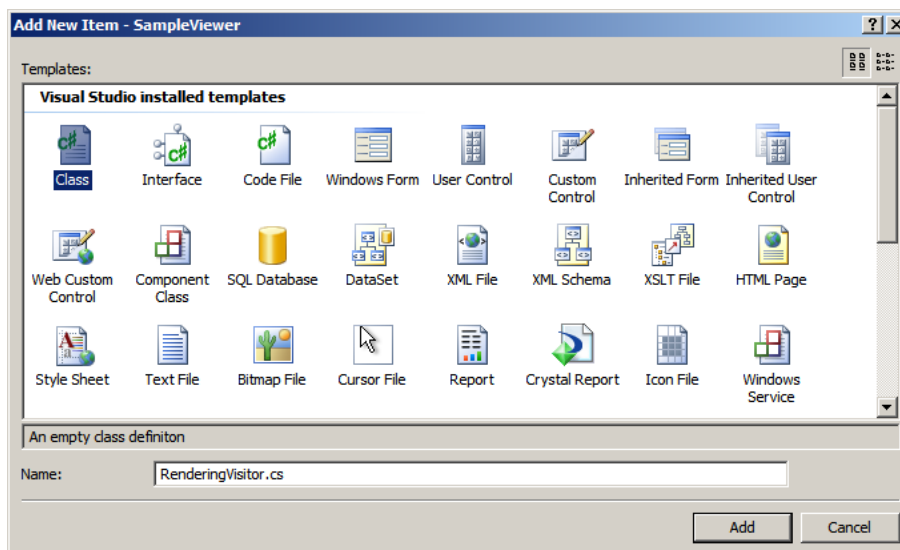
Creating the RenderingVisitor Class

In this exercise, we will be creating a new class, `RenderingVisitor.cs`, and providing it with basic methods and data members; As the name suggests, this class will be the workhorse that will actually renders the geometries. This class will be expanded upon in further sections.

1. Right-click on the `SampleViewer` project in the Solution Explorer. A pop-up window should now appear. Navigate to `Add > New Item...` and double-click on it.



2. The Add New Item dialog should appear. Select the Class template and change the name of the file to RenderingVisitor.cs then click the Add button.



3. RenderingVisitor.cs should now appear on screen. Add the following lines to the list of namespace uses:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using Safe.FMEObjects;
```

4. Modify the RenderingVisitor class so that it is subclass of IFMEOGGeometryVisitor.

```
class RenderingVisitor : IFMEOGGeometryVisitor
```

5. At the top of RenderingVisitor.cs, add a new heading where you will be declaring your member variables. Declare the private member variables m_graphics which will be used to render the graphics onto the rendering panel. As well, declare the private variable m_featureSetBoundingBox which represents the bounding box for a feature collection. Initialize both of them to null.

```
private Graphics m_graphics = null;
private IFMEORectangle m_featureSetBoundingBox = null;
```

6. As well, add the following private member variables used to hold the amount to scale the coordinates of the geometry and initialize them to 1.

```
private Graphics m_graphics = null;
private IFMEORectangle m_featureSetBoundingBox = null;
private double m_xScaleFactor = 1;
private double m_yScaleFactor = 1;
```

7. Add a constructor for the RenderingVisitor class and define it to be the following:

```
public RenderingVisitor(Graphics graphics,
                        IFMEORectangle featureSetBoundingBox,
                        double xScaleFactor,
                        double yScaleFactor)
{
    m_graphics = graphics;
    m_featureSetBoundingBox = featureSetBoundingBox;
    m_xScaleFactor = xScaleFactor;
    m_yScaleFactor = yScaleFactor;
}
```

8. As well, add the following public property Version which is used to state which version of the API that the RenderingVisitor was created under:

```
public int Version
{
    get
    {
        return **;
    }
}
```

This facility is used to ensure that subclasses made under previous versions of the FME Objects .NET API are updated to reflect changes in the API. The ** present in the property are used to denote the appropriate version number; at the time of this document's release, the version was 1.

9. Finally, add the following private helper function ConvertToScreenCoords that will take the given coordinates and convert them to the respective screen coordinates:

```
private void ConvertToScreenCoords(ref double x, ref double y)
{
    x = (x - m_featureSetBoundingBox.MinX) * m_xScaleFactor;
    y = (y - m_featureSetBoundingBox.MinY) * m_yScaleFactor;
}
```

Using the Visitor Design Pattern to Facilitate Rendering of Feature Geometries

In this exercise, we will be taking advantage of the Visitor Design Pattern to allow us to generically render all of the geometries; in particular, since `RenderingVisitor` implements the `IFMEOGGeometryVisitor` interface, `RenderingVisitor` has a capability to “visit” any particular geometry through the `Visit*` series methods (eg. `VisitArc` for visiting arcs, etc.)

However, since the `Visit*` methods require one to know what type of geometry there are working with, this would typically require a large amount of downcasting. This is where `IFMEOGGeometry`’s `AcceptGeometryVisitor` method comes in play; through polymorphism, it ensures that the appropriate `Visit*` method is called on the geometry without any of the downcasting that previously would have been required.

Together, the `IFMEOGGeometry`’s `AcceptGeometryVisitor` method and the `IFMEOGGeometryVisitor`’s `Visit*` series of functions form the Visitor Design Pattern, which can be used in a multitude of situations. Note that the `IFMEOAreaVisitor`, `IFMEOSimpleAreaVisitor`, `IFMEOCurveVisitor` and `IFMEOSegmentVisitor` are also available for more specialized operations.

1. In the `RenderingVisitor.cs` file, add a new heading for constants and add the following definitions:

```
private static readonly Pen m_kPen = Pens.Black;
private static readonly Brush m_kBrush = Brushes.Black;
private static readonly Font m_kFont = new Font(new FontFamily("Times New Roman"), 6);
```

2. Add the `VisitNull` method which will be used to render `IFMEONull` geometries:

```
public bool VisitNull(IFMEONull fmeNull)
{
    // No rendering to perform; Return true
    return true;
}
```

Since a `IFMEONull` is used to represent a “Null” geometry, which is just an empty geometry, there is no rendering work required.

3. Add the `VisitPoint` method which will be used to render `IFMEOPoint` geometries:

```
public bool VisitPoint(IFMEOPoint point)
{
    // Extract the X and Y coordinates from the point
    double x = point.X;
    double y = point.Y;

    // Convert the coordinates to screen coordinates
    ConvertToScreenCoords(ref x, ref y);

    // Render the point on the rendering panel
    m_graphics.FillEllipse(m_kBrush, (float)x, (float)y, 1, 1);

    // Rendering complete; Return true
    return true;
}
```

4. Add the VisitText method which will render IFMEOText geometries:

```
public bool VisitText(IFMEOText text)
{
    // If there is text to render
    if (text.TextString.Length > 0)
    {
        // Get the text's object location; Note that we own the returned IFMEOPoint
        IFMEOPoint point = text.LocationAsPoint;
        double x = point.X;
        double y = point.Y;

        // Convert the coordinates to screen coordinates
        ConvertToScreenCoords(ref x, ref y);

        // Render the text string on the rendering panel
        m_graphics.DrawString(text.TextString, m_kFont, m_kBrush, (float)x, (float)y);

        // Dispose of the given point
        point.Dispose();
    }

    // Rendering complete; Return true
    return true;
}
```

5. Add the VisitLine method which will render IFMEOLine geometries:

```
public bool VisitLine(IFMEOLine line)
{
    // If there are points on the line to render
    if (line.NoOfPoints > 0)
    {
        // Used to retrieve the coordinates from the line
        double[] x = new double[line.NoOfPoints];
        double[] y = new double[line.NoOfPoints];

        // Retrieve the points from the line
        line.GetPointsXY(x, y);

        // Convert the coordinates to screen coordinates
        for (int i = 0; i < line.NoOfPoints; i++)
        {
            ConvertToScreenCoords(ref x[i], ref y[i]);
        }

        // If there is only one point
        if (line.NoOfPoints == 1)
        {
            // Render the one point (similar to VisitPoint)
            m_graphics.FillEllipse(m_kBrush, (float)x[0], (float)y[0], 1, 1);
        }
        else
        {
            // Go through all of the coordinates
            for (int i = 0; (i+1) < line.NoOfPoints; i++)
            {
                // Render each line segment
                m_graphics.DrawLine(m_kPen, (float)x[i], (float)y[i],
                                     (float)x[i+1], (float)y[i+1]);
            }
        }
    }

    // Rendering complete; Return true
    return true;
}
```

6. Add the VisitArc method which will render IFMEOArc geometries:

```
public bool VisitArc(IFMEOArc arc)
{
    // Get a line representation of the arc
    IFMEOLine line = arc.GetAsLine();

    // Render the line representation
    line.AcceptGeometryVisitor(this);

    // Rendering complete; Return true
    return true;
}
```

The reason that a line representation of the arc is being used instead of the actual arc is that the System.Drawing.Graphics namespace describes arcs differently than the FME Objects .NET API does. As such, it would be simpler in this case to render a line representation of the arc.

7. Add the VisitPath method which will render IFMEOPath geometries:

```
public bool VisitPath(IFMEOPath path)
{
    // If there segments associated with this path
    if (path.NoOfParts > 0)
    {
        // Get a IFMEOSegmentIterator for this path
        IFMEOSegmentIterator iterator = path.GetIterator();

        // Go through all of the segments associated with this path
        while (iterator.Next() == true)
        {
            // Get the current segment - We do not own the returned segment
            // and as such, it does not need to be disposed
            IFMEOSegment currSegment = iterator.GetPart();

            // Render that segment
            currSegment.AcceptGeometryVisitor(this);
        }

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}
```

8. Add the VisitPolygon method which will render IFMEOPolygon geometries:

```
public bool VisitPolygon(IFMEOPolygon polygon)
{
    // Get a curve representing the boundary of the polygon - We do not
    // own the returned curve and as such, it does not need to be disposed
    IFMEOCurve curve = polygon.GetBoundaryAsCurve();

    // Render the curve representation
    curve.AcceptGeometryVisitor(this);

    // Rendering complete; Return true
    return true;
}
```

9. Add the VisitEllipse method which will render IFMEOEllipse geometries:

```
public bool VisitEllipse(IFMEOEllipse ellipse)
{
    // Get the arc representing the boundary of the ellipse - We do not
    // own the returned arc and as such, it does not need to be disposed
    IFMEOArc arc = ellipse.GetBoundaryAsArc();

    // Render the boundary arc
    arc.AcceptGeometryVisitor(this);

    // Rendering complete; Return true
    return true;
}
```

10. Add the VisitDonut method which will render IFMEODonut geometries:

```
public bool VisitDonut(IFMEODonut donut)
{
    // Retrieve the outer boundary of the donut - We do not own the returned
    // simple area and as such, it does not need to be disposed
    IFMEOSimpleArea outerBoundary = donut.OuterBoundaryAsSimpleArea;

    // Render the outer boundary
    outerBoundary.AcceptGeometryVisitor(this);

    // If there are inner boundaries associated with this donut
    if (donut.NoOfInnerBoundaries > 0)
    {
        // Get a IFMEOSimpleAreaIterator for this donut
        IFMEOSimpleAreaIterator iterator = donut.GetIterator();

        // Go through all of the inner boundaries
        while (iterator.Next() == true)
        {
            // Get the current simple area
            IFMEOSimpleArea currSimpleArea = iterator.GetPart();

            // Render that simple area
            currSimpleArea.AcceptGeometryVisitor(this);
        }

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}
```

11. Add the VisitMultiArea method which will render IFMEOMultiArea geometries:

```
public bool VisitMultiArea(IFMEOMultiArea multiArea)
{
    // If there are areas associated with this multi-area
    if (multiArea.NoOfParts > 0)
    {
        // Get a IFMEOAreaIterator for this multi-area
        IFMEOAreaIterator iterator = multiArea.GetIterator();

        // Go through all of the areas associated with this multi-area
        while (iterator.Next() == true)
        {
```

```

        // Get the current area - We do not own the returned area
        // and as such, it does not need to be disposed
        IFMEOArea currArea = iterator.GetPart();

        // Render that area
        currArea.AcceptGeometryVisitor(this);
    }

    // Dispose of the iterator
    iterator.Dispose();
}

// Rendering complete; Return true
return true;
}

```

12. Add the VisitMultiCurve method which will render IFMEOMultiCurve geometries:

```

public bool VisitMultiCurve(IFMEOMultiCurve multiCurve)
{
    // If there are curves associated with this multi-curve
    if (multiCurve.NoOfParts > 0)
    {
        // Get a IFMEOCurveIterator for this multi-curve
        IFMEOCurveIterator iterator = multiCurve.GetIterator();

        // Go through all of the curves associated with this multi-curve
        while (iterator.Next() == true)
        {
            // Get the current curve - We do not own the returned curve
            // and as such, it does not need to be disposed
            IFMEOCurve currCurve = iterator.GetPart();

            // Render that curve
            currCurve.AcceptGeometryVisitor(this);
        }

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}

```

13. Add the VisitMultiPoint method which will render IFMEOMultiPoint geometries:

```

public bool VisitMultiPoint(IFMEOMultiPoint multiPoint)
{
    // If there are curves associated with this multi-point
    if (multiPoint.NoOfParts > 0)
    {
        // Get a IFMEOPointIterator for this multi-point
        IFMEOPointIterator iterator = multiPoint.GetIterator();

        // Go through all of the points associated with this multi-point
        while (iterator.Next() == true)
        {
            // Get the current point - We do not own the returned point
            // and as such, it does not need to be disposed
            IFMEOPoint currPoint = iterator.GetPart();

            // Render that point
            currPoint.AcceptGeometryVisitor(this);
        }
    }
}

```



```

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}

```

14. Add the VisitMultiText method which will render IFMEOMultiText geometries:

```

public bool VisitMultiText(IFMEOMultiText multiText)
{
    // If there are text objects associated with this multi-text
    if (multiText.NoOfParts > 0)
    {
        // Get a IFMEOTextIterator for this multi-text
        IFMEOTextIterator iterator = multiText.GetIterator();

        // Go through all of the text objects associated with this multi-text
        while (iterator.Next() == true)
        {
            // Get the current text - We do not own the returned text
            // and as such, it does not need to be disposed
            IFMEOText currText = iterator.GetPart();

            // Render that text object
            currText.AcceptGeometryVisitor(this);
        }

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}

```

15. Finally, add the VisitAggregate method which will render IFMEOAggregate geometries:

```

public bool VisitAggregate(IFMEOAggregate aggregate)
{
    // If there are geometries associated with this aggregate
    if (aggregate.NoOfParts > 0)
    {
        // Get a IFMEOGeometryIterator for this aggregate
        IFMEOGeometryIterator iterator = aggregate.GetIterator();

        // Go through all of the geometries associated with this aggregate
        while (iterator.Next() == true)
        {
            // Get the current geometry - We do not own the returned geometry
            // and as such, it does not need to be disposed
            IFMEOGeometry currGeometry = iterator.GetPart();

            // Render that geometry
            currGeometry.AcceptGeometryVisitor(this);
        }

        // Dispose of the iterator
        iterator.Dispose();
    }

    // Rendering complete; Return true
    return true;
}

```

Testing your changes

Build the solution and verify that the application still successfully compiles.

Displaying Feature Geometries on the Rendering Panel

In this exercise, you will update the Rendering Panel to include a Paint handler; this event handler will, depending on what tab is currently selected, extract the appropriate set of features and will then, through the RenderingVisitor, proceed to render them.

1. Switch to the Form1.cs file and define the private helper function GetFeaturesToRender as follows:

```
public IFMEOfeatureVectorOnDisk GetFeaturesToRender()
{
    // The set of features to render
    IFMEOfeatureVectorOnDisk featuresToRender = null;

    // Depending on the current tab
    switch ((eTabPanels)tabControll1.SelectedIndex)
    {
        case (eTabPanels.FeatureTypeTab):
        {
            // The selected feature type - Note there should be
            // at most 1 selected (since MultiSelect is set to False)
            ListView.SelectedListViewItemCollection selected = featureTypeView.SelectedItems;

            // If there is an item selected
            if (selected.Count > 0)
            {
                // Get the name of the feature type
                string featureType = selected[0].SubItems[0].Text;

                if ((featureType != null) &&
                    (m_featureTypeDictionary.ContainsKey(featureType) == true))
                {
                    // Get the IFMEOfeatureVectorOnDisk associated with that feature type
                    featuresToRender = m_featureTypeDictionary[featureType];
                }
            }

            break;
        }
        case (eTabPanels.GeometryTab):
        {
            // The selected feature type - Note there should be
            // at most 1 selected (since MultiSelect is set to False)
            ListView.SelectedListViewItemCollection selected = geometryView.SelectedItems;

            // If there is an item selected
            if (selected.Count > 0)
            {
                // Get the name of the geometry type
                string featureType = selected[0].SubItems[0].Text;

                FMEOGeometryClass geomClass;

                if (featureType == "Null") geomClass = FMEOGeometryClass.IFMEONull;
```

```

else if(featureType=="Donut") geomClass = FMEOGGeometryClass.IFMEODonut;
else if(featureType=="Ellipse") geomClass = FMEOGGeometryClass.IFMEOEllipse;
else if(featureType=="Polygon") geomClass = FMEOGGeometryClass.IFMEOPolygon;
else if(featureType=="Path") geomClass = FMEOGGeometryClass.IFMEOPath;
else if(featureType=="Arc") geomClass = FMEOGGeometryClass.IFMEOArc;
else if(featureType=="Line") geomClass = FMEOGGeometryClass.IFMEOLine;
else if(featureType=="Point") geomClass = FMEOGGeometryClass.IFMEOPoint;
else if(featureType=="Text") geomClass = FMEOGGeometryClass.IFMEOText;
else if(featureType=="Aggregate") geomClass = FMEOGGeometryClass.IFMEOAggregate;
else if(featureType=="MultiArea") geomClass = FMEOGGeometryClass.IFMEOMultiArea;
else if(featureType=="MultiCurve") geomClass = FMEOGGeometryClass.IFMEOMultiCurve;
else if(featureType=="MultiPoint") geomClass = FMEOGGeometryClass.IFMEOMultiPoint;
else if(featureType=="MultiText") geomClass = FMEOGGeometryClass.IFMEOMultiText;
else break;

if (m_geometryDictionary.ContainsKey(geomClass) == true)
{
    // Get the IFMEOFeatureVectorOnDisk associated with that feature type
    featuresToRender = m_geometryDictionary[geomClass];
}
break;
}
case (eTabPanels.CoordSysTab):
{
    // Get the text from the combo box
    string featureType = (string)featureTypeComboBox.SelectedItem;

    if ((featureType != null) &&
        (m_featureTypeDictionary.ContainsKey(featureType) == true))
    {
        // Get the IFMEOFeatureVectorOnDisk associated with that feature type
        featuresToRender = m_featureTypeDictionary[featureType];
    }

    break;
}
case (eTabPanels.FormatInfoTab):
{
    // There is no geometry associated with the format information; Break out
    break;
}
case (eTabPanels.SchemaInfoTab):
{
    // Get the text from the combo box
    string featureType = (string)schemaFeatureComboBox.SelectedItem;

    if ((featureType != null) &&
        (m_featureTypeDictionary.ContainsKey(featureType) == true))
    {
        // Get the IFMEOFeatureVectorOnDisk associated with that feature type
        featuresToRender = m_featureTypeDictionary[featureType];
    }

    break;
}
default:
{
    break;
}
}
// Return the features to be rendered
return featuresToRender;
}

```

- Next, add the `renderingPanel_Paint` event handler method as follows; This method will be used to render the given features whenever the rendering panel is repainted:

```
public void renderingPanel_Paint(object sender, System.Windows.Forms.PaintEventArgs e)
{
    // The features to be rendered
    IFMEOFeatureVectorOnDisk renderFeatures = GetFeaturesToRender();

    // If there are features to render
    if ((renderFeatures != null) && (renderFeatures.Count > 0))
    {
        // Generate the bounding box for the feature set
        IFMEORectangle featureSetBoundingBox = generateBoundingBox(renderFeatures);

        // If the bounding box was generated successfully
        if (featureSetBoundingBox != null)
        {
            // Calculate the appropriate scaling factors
            double xScaleFactor = 1;
            double yScaleFactor = 1;
            calculateScalingFactors(featureSetBoundingBox,
                                   renderingPanel.Height, renderingPanel.Width,
                                   ref xScaleFactor, ref yScaleFactor);

            // Create the RenderingVisitor
            RenderingVisitor visitor = new RenderingVisitor(e.Graphics, featureSetBoundingBox,
                                                            xScaleFactor, yScaleFactor);

            // Go through each of the features
            for (int i = 0; i < renderFeatures.Count; i++)
            {
                // Get the current feature - This copy should be
                // disposed once it is no longer needed
                IFMEOFeature currFeature = renderFeatures.GetAt(i);

                // Attempt to get the geometry for the current feature
                IFMEOGeometry currGeometry = currFeature.GetGeometry();

                // If the current feature actually has geometry
                if (currGeometry != null)
                {
                    // Render the geometry
                    currGeometry.AcceptGeometryVisitor(visitor);

                    // Dispose of the geometry
                    currGeometry.Dispose();
                }

                // Dispose of the feature copy
                currFeature.Dispose();
                currFeature = null;
            }

            // Dispose of the feature set bounding box
            featureSetBoundingBox.Dispose();
            featureSetBoundingBox = null;
        }
    }
}
```

- Define the following `featureTypeView_SelectedIndexChanged` event handler function:

```
public void featureTypeView_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // Force a repaint of the renderingPanel
    renderingPanel.Refresh();
}
```

This will be used to make the renderingPanel repaint when the selected index of the featureTypeView is changed, rendering the geometry of the newly-selected feature type.

4. Define the following geometryView_SelectedIndexChanged event handler function:

```
public void geometryView_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // Force a repaint of the renderingPanel
    renderingPanel.Refresh();
}
```

This will be used to make the renderingPanel repaint when the selected index of the geometryView is changed, rendering the geometry of the newly-selected type

5. Update the featureTypeComboBox_SelectedIndexChanged event handler to include the following lines:

```
private void featureTypeComboBox_SelectedIndexChanged(object sender, System.EventArgs e)
{
    ...

    // Dispose the feature since we owned it
    selectedFeature.Dispose();

    if (coordSystemName.Length != 0)
    {
        ...
    }

    // Force the rendering panel to repaint itself
    renderingPanel.Refresh();
}
```

This will be used to make the renderingPanel repaint when the selected index of the featureTypeComboBox is changed, rendering the geometry of the newly-selected feature type.

6. Update the schemaFeatureComboBox_SelectedIndexChanged event handler to include the following lines:


```
private void schemaFeatureComboBox_SelectedIndexChanged(object sender, System.EventArgs e)
{
    ...

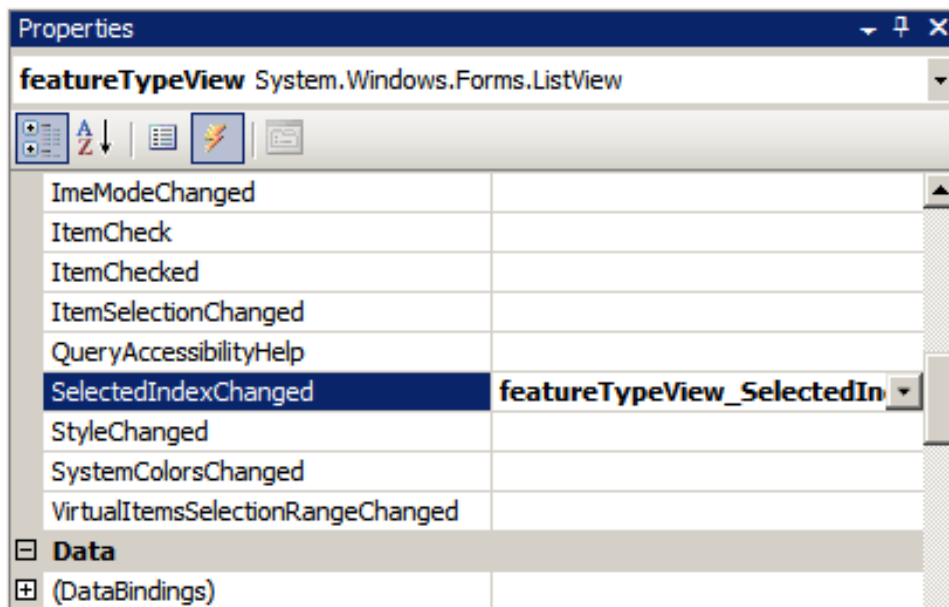
    // Dispose the feature since we owned it
    selectedFeature.Dispose();

    if (coordSystemName.Length != 0)
    {
        ...
    }

    // Force the rendering panel to repaint itself
    renderingPanel.Refresh();
}
```

This will be used to make the renderingPanel repaint when the selected index of the schemaFeatureComboBox is changed, rendering the geometry of the newly-selected feature type.

7. Switch to the Designer view and click on the Feature Type Tab. Click on the featureTypeView List View, navigate to the Properties panel and click on the Events button  to see its list of events. Assign the featureTypeView_SelectedIndexChanged handler to the SelectedIndexChanged event. The rendering panel will now repaint when the selected feature type changes.



8. Repeat the above step using the Geometry Tab's geometryView and the geometryView_SelectedIndexChanged method.
9. Finally, click on the renderingPanel on the form and again, navigate to the Properties panel. Switch to the list of events and assign the renderingPanel_Paint handler to the Paint event. This means that whenever the Rendering Panel is repainted, it will render the appropriate features.

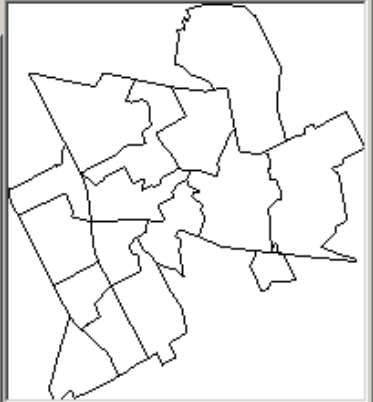
Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open schoolRegions.gml (with GML - Geography Markup Language as format). Observe that when you select a feature type or a type of geometry in any of the tabs, the set of features associated with it will be rendered on the Rendering Panel.

Form1 File Help

Feature Type Geometry Coord Sys Format Info Schema Info

Feature Type	Count
highschool_catchment	4
middleschool_catchment	6
primaryschool_catchment	15
school_districts	4



Total features: 29

Chapter 10

Advanced Topics

In this chapter

Using separate threads for reading/writing datasets

Implement column click sorting on ListView components (extra)

Objective

The purpose of this chapter is to assist you in making some further enhancements to the sample application.

Using Separate Threads for Reading/Writing Datasets

The sample application that you have developed in this tutorial has only one main thread of control. When working with small sized datasets (tens of thousands of features), this design is probably sufficient, as the waiting time for opening/writing datasets is not significant. However, for situations where the application needs to handle production sized datasets (hundreds of thousands of features), the user interface delay would become unacceptable. With only one thread of control, the user would not be able to do anything else with the application until the read/write operation completes.

A better design is to spawn a separate thread for the work of reading/writing datasets. With this approach, the main thread is still available for responding to user events (from the GUI) while the work of reading/writing datasets is taking place. Hence, the overall program response is significantly improved.

1. Open up your existing solution from the previous chapter. If you wish to start from a standard solution, open the SampleViewer.sln in the chapter9 folder.
2. Open up Form1.cs in the Code view. Add System.Threading to the list of namespaces being used.

```
using System.Threading;
```

3. Declare a new member named m_workerThread of type Thread. Set its initial reference to null.

```
private Thread m_workerThread = null;
```


4. Declare a new method named `enableOpenAndSave` as follows:

```
private void enableOpenAndSave(bool flag)
{
    openToolStripMenuItem.Enabled = flag;
    saveToolStripMenuItem.Enabled = flag;
}
```

This is a helper method used to enable/disable the Open.../Save As... options from the File menu. To prevent unexpected behavior, both the Open... and Save As... options should be disabled whenever a worker thread is active.

5. Navigate to the `openOption_Click` event handler. Select all the code within the `m_fmeDialog.SourcePrompt` conditional check. Cut and paste the code as a new method named `executeRead`. In addition, use the `enableOpenAndSave` method to replace the `saveAsToolStripMenuItem.Enabled = true` line.

```
private void executeRead()
{
    try
    {
        // Disposes and clears entries inside m_featureTypeDictionary
        resetAllDictionaries();

        // Add IFMEORReader code here
        // Update status bar
        updateStatusBar("Reading from source...");

        ...

        // Enable the tab panels
        tabControl1.Enabled = true;

        // Refresh the current tab
        refreshCurrentTab();

        enableOpenAndSave(true);
    }
    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(), FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}
```

6. Now, modify the `openOption_Click` event handler as follows:

```
private void openOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.SourcePrompt("", "", m_dataInfo, m_createDirectives))
        {
            // create a new thread to read
            enableOpenAndSave(false);
            m_workerThread = new Thread(new ThreadStart(executeRead));
        }
    }
}
```

```

        m_workerThread.Start();
    }
}
catch(FMEOException ex)
{
    // Log errors to log file
    m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
    m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
    m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(),
                                  FMEOMessageLevel.Error);

    // Now exit the application
    Application.Exit();
}
}

```

The biggest modification to this method is the creation of a new thread. The new thread is supplied with the method to execute (executeRead), and then the new thread is started.

7. Next, you will perform the same modification for creating a separate thread for writing datasets. Navigate to the saveAsOption_Click handler, and select all the code within the m_fmeDialog.DestinationPrompt conditional check. Cut and paste the code as a new method named executeWrite. In addition, add a call to enableOpenAndSave to enable the Open/Save As... options once writing is complete.

```

private void executeWrite()
{
    try
    {
        // Update the status bar
        updateStatusBar("Writing to destination...");

        // Create the writer
        IFMEOWriter fmeWriter = m_fmeSession.CreateWriter(m_outputDataInfo.Format,
                                                         m_createDirectives);

        ...

        // Clean up the writer
        fmeWriter.Close();
        fmeWriter.Dispose();
        fmeWriter = null;

        enableOpenAndSave(true);
    }

    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(), FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}

```

8. Now, modify the saveAsOption_Click event handler as follows:

```
private void saveAsOption_Click(object sender, System.EventArgs e)
{
    try
    {
        if (m_fmeDialog.DestinationPrompt("", "", m_outputDataInfo, m_createDirectives))
        {
            // Create a new thread to write
            enableOpenAndSave(false);
            m_workerThread = new Thread(new ThreadStart(executeWrite));
            m_workerThread.Start();
        }
    }

    catch(FMEOException ex)
    {
        // Log errors to log file
        m_fmeLogFile.LogMessageString(ex.FmeErrorMessage, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeStackTrace, FMEOMessageLevel.Error);
        m_fmeLogFile.LogMessageString(ex.FmeErrorNumber.ToString(),
                                      FMEOMessageLevel.Error);

        // Now exit the application
        Application.Exit();
    }
}
```

As with reading datasets, a new thread is also now created for writing datasets.

9. Switch to the Form1.Designer.cs file and update the Dispose method as follows:

```
protected override void Dispose( bool disposing )
{
    if(disposing && (components != null))
    {
        components.Dispose();
    }

    // If we have another existing worker thread, we need
    // to terminate it
    if (m_workerThread != null)
    {
        m_workerThread.Abort();
    }

    // Disposes and clears entries inside all of the dictionaries
    resetAllDictionaries();

    ...
}
```

If there is a worker thread still existing when the sample application needs to terminate, the worker thread should be terminated to prevent unexpected behavior.

Testing your changes

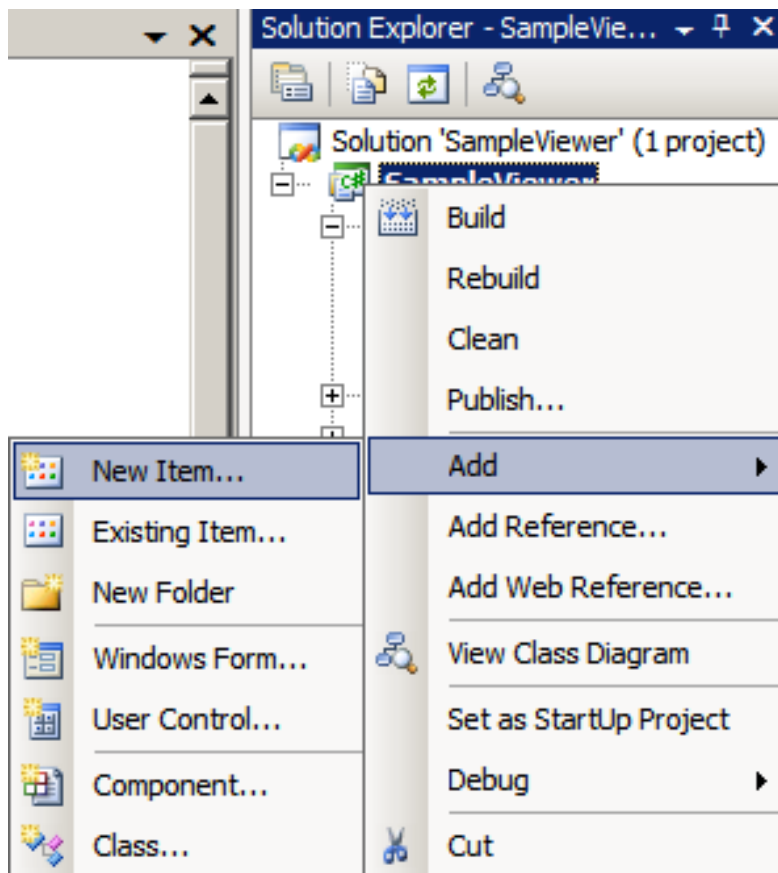
Build the solution and run the application. From the Austin Mini Dataset, open the water distribution drawings distribution_L_24.dwg through to distribution_L_29.dwg (with AutoCAD DWG/DXF as format). Observe that while the reading is taking place, you are still able to interact with the program (ie. minimizing and moving the application). The same behavior should exhibit when writing to a dataset.

Implement Column Click Sorting on ListView Components

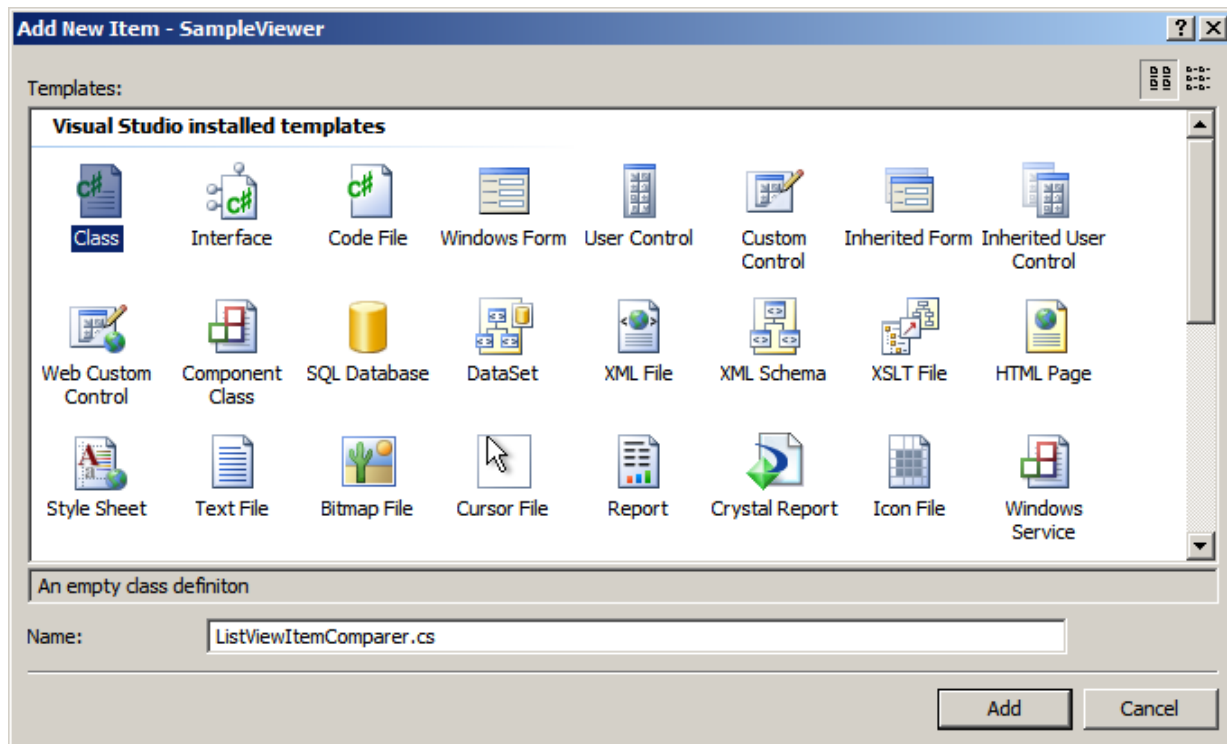
In the current sample application, you will notice that the items displayed inside each of the ListView components are not sorted. To make the application more user-friendly, the user should be given the ability to sort items by clicking on a particular column of the ListView component.

Each ListView component contains an IComparer member (named ListViewItemSorter) that controls the way that items are sorted in the control. In this exercise, you will create a new class that implements the IComparer interface, and you will use this class to set the sorting behavior of each ListView components in the application.

1. Add a new class to the project. Inside the Solution Explorer, right-click on the project file and select Add, and then New Item....



2. In the Add New Item dialog, ensure that the class template has been selected and enter ListViewItemComparer for the name.



3. Extend the class definition as shown:

```
using System;
using System.Collections;
using System.Windows.Forms;

namespace SampleViewer
{
    /// <summary>
    /// ListViewItemComparer is used by all the ListView components for sorting its entries
    /// </summary>
    public class ListViewItemComparer : IComparer
    {
        private int m_column;
        private bool m_integerSort;
        private SortOrder m_sortOrder;

        /// <summary>
        /// Default constructor - sorting in ascending order by column 0
        /// </summary>
        public ListViewItemComparer()
        {
            m_column = 0;
            m_sortOrder = SortOrder.Ascending;
            m_integerSort = false;
        }
    }
}
```

As mentioned earlier, the ListViewItemComparer class will implement the IComparer interface. The m_column member variable is used for storing the column (of the ListView component) of which the items are sorted by. The m_integerSort member variable determines whether or integer sorting will be used.

4. Declare 3 properties inside ListViewItemComparer, which will set/get the values of m_column, m_sortOrder, and m_integerSort.

```

/// <summary>
/// SortColumn
/// Get/set m_column
/// </summary>
public int SortColumn
{
    get
    {
        return m_column;
    }
    set
    {
        m_column = value;
    }
}

/// <summary>
/// IntegerSort
/// Get/set m_integerSort
/// </summary>
public bool IntegerSort
{
    get
    {
        return m_integerSort;
    }
    set
    {
        m_integerSort = value;
    }
}

/// <summary>
/// SortOrder Property
/// Get/set m_sortOrder
/// </summary>
public SortOrder SortOrder
{
    get
    {
        return m_sortOrder;
    }
    set
    {
        m_sortOrder = value;
    }
}

```

5. Declare the algorithms for sorting strings and integer as follows:

```

/// <summary>
/// StringCompare
/// String comparison between two strings
/// </summary>
private int StringCompare(object x, object y)
{
    int result = String.Compare(((ListViewItem) x).SubItems[m_column].Text,
                                ((ListViewItem) y).SubItems[m_column].Text);

    if (m_sortOrder == SortOrder.Ascending)
    {
        return result;
    }
}

```

```

    }
    else
    {
        return result*(-1);
    }
}

/// <summary>
/// IntegerCompare
/// Integer comparison between two integers
/// </summary>
private int IntegerCompare(object x, object y)
{
    int integerX = Convert.ToInt32(((ListViewItem) x).SubItems[m_column].Text);
    int integerY = Convert.ToInt32(((ListViewItem) y).SubItems[m_column].Text);

    int result = 0;

    if (integerX < integerY)
    {
        result = -1;
    }
    else if (integerX > integerY)
    {
        result = 1;
    }

    if (m_sortOrder == SortOrder.Ascending)
    {
        return result;
    }
    else
    {
        return result*(-1);
    }
}

```

6. Finally, to finish off the definition of `ListViewItemComparer`, we must implement the method `Compare` as required by the `IComparer` interface.

```

/// <summary>
/// Compare
/// Compares two objects and indicates whether the first one is less than,
/// equal to, or greater than the second object
/// </summary>
public int Compare(object x, object y)
{
    int result;

    // Check whether to use integer sorting or string sorting
    if (m_integerSort)
    {
        result = IntegerCompare(x,y);
    }
    else
    {
        result = StringCompare(x,y);
    }

    return result;
}

```

As you can see, the method performs a simple check to see whether or not we are using integer sorting, and then it determines the result using the appropriate sorting algorithm.

7. At this point, the `ListViewItemComparer` class is ready to be used. From the Solution Explorer, open `Form1.cs` in the Code view. Update the constructor as shown:

```
public Form1()
{
    ...

    // Disable the tab panels and save as option
    tabPanels.Enabled = false;
    saveAsOption.Enabled = false;

    // Set the ListView in each tab to use the default sort
    // (in ascending order by column 0)
    featureTypeView.ListViewItemSorter = new ListViewItemComparer();
    geometryView.ListViewItemSorter = new ListViewItemComparer();
    coordSysView.ListViewItemSorter = new ListViewItemComparer();
    formatInfoView.ListViewItemSorter = new ListViewItemComparer();
    schemaInfoView.ListViewItemSorter = new ListViewItemComparer();
}
```

Now, each of the `ListView` components has been initialized to sort items in the control by the first column.

8. Declare a new event handler named `columnClick`. This event handler will be called whenever a user selects a column on any of the `ListView` components.

```
private void columnClick(object sender, ColumnClickEventArgs e)
{
    ListView listView = (ListView) sender;

    // Retrieve the selected column from the ColumnHeaderCollection
    ListView.ColumnHeaderCollection columnList = listView.Columns;
    ColumnHeader selectedColumn = columnList[e.Column];

    // Denotes whether or not we will be using integer sorting
    bool integerSort = false;

    // All "Count" columns will need to use integer sorting instead of the
    // default string sorting
    if (selectedColumn.Text.Equals("Count"))
    {
        integerSort = true;
    }

    ListViewItemComparer itemSorter = (ListViewItemComparer) listView.ListViewItemSorter;

    // Set whether or not we will be doing integer sorting
    itemSorter.IntegerSort = integerSort;

    // Check to see if we are sorting by a different column or not
    if (itemSorter.SortColumn == e.Column)
    {
        // In this case, we swap the sorting order
        if (itemSorter.SortOrder == SortOrder.Ascending)
        {
            itemSorter.SortOrder = SortOrder.Descending;
        }
        else
    }
```



```

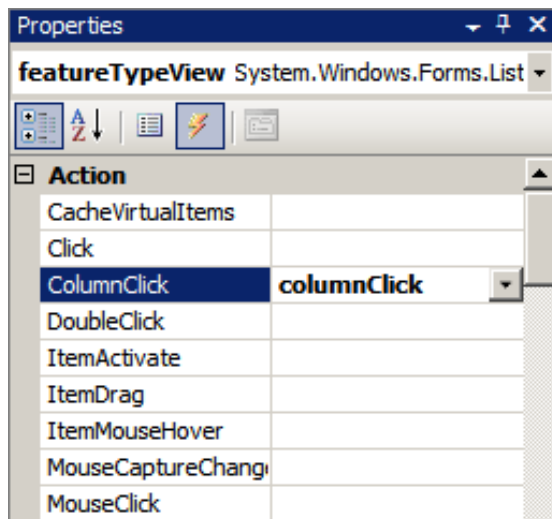
        {
            itemSorter.SortOrder = SortOrder.Ascending;
        }
    }
else
{
    // In this case, we set the new column to sort by, and
    // by default, we use ascending order
    itemSorter.SortColumn = e.Column;
    itemSorter.SortOrder = SortOrder.Ascending;
}

// Finally, sort the items using the updated sorting control
listView.Sort();
}

```

The event handler casts the sender parameter as a `ListView` type, and then it examines the name of the selected column to determine whether or not integer sorting should be used. In addition, if the same column has been selected consecutive times, the sorting order is swapped (from ascending to descending, and vice versa). Otherwise, the default ascending order is set.

9. The last step is to attach the `columnClick` handler to each of the `columnClick` events of the `ListView` components. As an example, select the `ListView` component inside the `Feature Type` tab and select `Events` under its properties. Assign the `columnClick` handler to the `columnClick` event.



10. Perform the same step to all other `ListView` components of the sample application.

Testing your changes

Build the solution and run the application. From the Austin Mini Dataset, open `schoolRegions.gml` (with GML - Geography Markup Language as format). Then click on any column of a `ListView` component. The items should be sorted accordingly.

