Lab 9: Managing the Lifetime of Objects and Controlling Resources

Exercise 1: Implementing the IDisposable Interface

Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010** .
3. Import the code snippets from the E:\Labfiles\Lab 9\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, click **Add**.
   c. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 9\Snippets** folder, and then click **Select Folder**.
   d. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module9 solution in the E:\Labfiles\Lab 9\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9\Ex1\Starter** folder, click **Module9.sln**, and then click **Open**.

Task 2: Create the ILoggingMeasuringDevice interface

In this task, you will develop the **ILoggingMeasuringDevice** interface. You will develop this new interface, which inherits from the existing **IMeasuringDevice** interface, rather than editing the existing interface to ensure compatibility with existing code.

1. Review the task list:

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

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

2. Open the ILoggingMeasuringDevice.cs file:

    - In Solution Explorer, double-click **ILoggingMeasuringDevice.cs**.

3. Remove the TODO comment and declare an interface named **ILoggingMeasuringDevice**. The interface must be accessible from code in different assemblies.

    Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface ILoggingMeasuringDevice
    {
    }
}
```

4. Modify the interface to inherit from the **IMeasuringDevice** interface.

    Your code should resemble the following code example.

```
public interface ILoggingMeasuringDevice : IMeasuringDevice
```

5. Add a method named **GetLoggingFile** that returns a **string** value to the interface. The method should take no parameters. The purpose of this method is to return the file name of the logging file used by the device. Add an XML comment that summarizes the purpose of the method.

    Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface ILoggingMeasuringDevice : IMeasuringDevice
    {
        /// <summary>
        /// Returns the file name of the logging file for the device.
        /// </summary>
        /// <returns>The file name for the logging file.</returns>
        string GetLoggingFile();
    }
}
```

6. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Modify the MeasureDataDevice class to implement the ILoggingMeasuringDevice interface

In this task, you will modify the existing **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface. You will add code to enable logging and modify existing methods to use the logging functionality.

1. Open the MeasureDataDevice.cs file:

   - In Solution Explorer, double-click **MeasureDataDevice.cs**.

2. Remove the comment **TODO: Modify this class to implement the ILoggingMeasuringDevice interface instead of the IMeasuringDevice interface** above the **MeasureDataDevice** class. Modify the **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface instead of the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    // TODO: Modify this class to implement the IDisposable interface.

    public abstract class MeasureDataDevice : ILoggingMeasuringDevice
    {
        ...
    }

}
```

3. In the task list, locate the comment **TODO: Add fields necessary to support logging.** Double-click this item to go to the relevant line in the MeasureDataDevice.cs file.

4. Remove the TODO comment and add a **string** field named **loggingFileName**. This field must be accessible to classes that inherit from this class. This field will store the file name and path for the log file.

   Your code should resemble the following code example.

```
...
protected DeviceType measurementType;
protected string loggingFileName;
// TODO: Add methods to implement the ILoggingMeasuringDevice
interface.

...
```

5. Add a **TextWriter** field named **loggingFileWriter**. This field should only be accessible to code in this class. You will use this object to write to a file.

   Your code should resemble the following code example.

```
...

protected string loggingFileName;
private TextWriter loggingFileWriter;

// TODO: Add methods to implement the ILoggingMeasuringDevice
interface.

...
```

6. In the task list, locate the comment **TODO: Add methods to implement the ILoggingMeasuringDevice interface.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.
7. Remove the TODO comment and add the **GetLoggingFile** method defined in the **ILoggingMeasuringDevice** interface. The method should take no parameters and return the value in the **loggingFileName** field.

   Your code should resemble the following code example.

```
...

private TextWriter loggingFileWriter;

public string GetLoggingFile()
{
    return loggingFileName;
}

...
```

8. In the task list, locate the comment **TODO: Add code to open a logging file and write an initial entry**. Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.

9. Remove the TODO comment and add the following code to instantiate the **loggingFileWriter** field. You can either type this code manually, or you can use the Mod9InstantiateLoggingFileWriter code snippet.

```
// New code to check the logging file is not already open.
// If it is already open then write a log message.
// If not, open the logging file.
if (loggingFileWriter == null)
{
    // Check if the logging file exists - if not create it.
    if (!File.Exists(loggingFileName))
    {
        loggingFileWriter = File.CreateText(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Created");
        loggingFileWriter.WriteLine("Collecting Started");
    }
    else
    {
        loggingFileWriter = new StreamWriter(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Opened");
        loggingFileWriter.WriteLine("Collecting Started");
    }
}
else
{
    loggingFileWriter.WriteLine
        ("Log file status checked - Already open");
    loggingFileWriter.WriteLine("Collecting Started");
}
```

The code checks whether the **loggingFileWriter** object has already been instantiated. If it has not, the code instantiates it by checking whether the file specified by the **loggingFileName** field already exists. If the file exists, the code opens the file; if it does not, the code creates a new file:

- To use the code snippet, delete the TODO comment, type **Mod9InstantiateLoggingFileWriter** and then press the TAB key.

10. In the task list, locate the comment **TODO: Add code to write a message to the log file.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.

11. Remove the TODO comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.

Your code should resemble the following code example.

```
public void StopCollecting()
{
    if (controller != null)
    {
        controller.StopDevice();
        controller = null;
    }

    // New code to write to the log.
    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine("Collecting Stopped.");
    }
}
```

12. In the task list, locate the comment **TODO: Add code to log each time a measurement is taken.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.

13. Remove the TODO comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.

    Your code should resemble the following code example.

```
while (controller != null)
{
    System.Threading.Thread.Sleep(timer.Next(1000, 5000));
    dataCaptured[x] = controller != null ?
        controller.TakeMeasurement() : dataCaptured[x];
    mostRecentMeasure = dataCaptured[x];
    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine
            ("Measurement Taken: {0}", mostRecentMeasure.ToString());
    }
    x++;
    if (x == 10)
    {
        x = 0;
    }
}
```

14. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Modify the MeasureDataDevice class to implement the IDisposable interface

In this task, you will modify the existing **MeasureDataDevice** class to implement the **IDisposable** interface. You will add code to ensure that the **TextWriter** object that writes messages to the log file is properly closed when an instance of the **MeasureDataDevice** class is disposed of.

1. At the top of the **MeasureDataDevice** class, remove the comment **TODO: Modify this class to implement the IDisposable interface**, and then modify the **MeasureDataDevice** class to implement the **IDisposable** interface in addition to the **ILoggingMeasuringDevice** interface.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{

    public abstract class MeasureDataDevice
        : ILoggingMeasuringDevice, IDisposable

    {
        ...
    }

}
```

2. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IDisposable** interface:
   - Right-click **IDisposable**, point to **Implement Interface**, and then click **Implement Interface**.
3. Move to the end of the **MeasureDataDevice** class. After the **Dispose** method added by the Implement Interface Wizard, add an overloaded **virtual void Dispose** method that implements the dispose pattern. This method should take a Boolean parameter called **disposing** and perform the following tasks:

   a. Check that the *disposing* parameter is set to **true**. If it is not, finish without disposing of anything.

   b. If the **loggingFileWriter** object is not null, write the message "Object disposed" to the logging file, flush the contents of the **loggingFileWriter** object, close it, and set the loggingFileWriter variable to **null**.

   Your code should resemble the following code example.

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // Check that the log file is closed; if it is not closed, log
        // a message and close it.
        if (loggingFileWriter != null)
        {
            loggingFileWriter.WriteLine("Object Disposed");
            loggingFileWriter.Flush();
            loggingFileWriter.Close();
            loggingFileWriter = null;
        }
    }
}
```

4. Locate the **Dispose** method, which takes no parameters, and then remove the default method body inserted by Visual Studio, which throws a **NotImplementedException** exception. Add statements that call the overloaded **Dispose** method and specify true as the parameter, and then suppress finalization for the current object.

   Your code should resemble the following code example.

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

5. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

   At the end of this task, the **MeasuringDataDevice** class should resemble the following code example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using DeviceControl;
using System.IO;

namespace MeasuringDevice
{
    public abstract class MeasureDataDevice
        : ILoggingMeasuringDevice, IDisposable
    {
        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into a metric value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to metric units.</returns>
        public abstract decimal MetricValue();

        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into an imperial value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to imperial units.</returns>
        public abstract decimal ImperialValue();

        /// <summary>
        /// Starts the measuring device.
        /// </summary>
        public void StartCollecting()
        {
            controller =
                DeviceController.StartDevice(measurementType);

            // New code to check the logging file is not already open.
            // If it is already open then write a log message.
            // If not, open the logging file.
            if (loggingFileWriter == null)
            {
                // Check whether the logging file exists -
                // if not create it.
                if (!File.Exists(loggingFileName))
                {
                    loggingFileWriter =
                        File.CreateText(loggingFileName);
                    loggingFileWriter.WriteLine
                        ("Log file status checked - Created");
                    loggingFileWriter.WriteLine("Collecting Started");
                }
                else
                {
                    loggingFileWriter =
                        new StreamWriter(loggingFileName);
                    loggingFileWriter.WriteLine
                        ("Log file status checked - Opened");
                    loggingFileWriter.WriteLine("Collecting Started");
                }
            }
            else
            {
                loggingFileWriter.WriteLine
                    ("Log file status checked - Already open");
                loggingFileWriter.WriteLine("Collecting Started");
            }
            GetMeasurements();
        }

        /// <summary>
        /// Stops the measuring device.
        /// </summary>
        public void StopCollecting()
        {
            if (controller != null)
            {
                controller.StopDevice();
                controller = null;
            }

            // New code to write to the log.
            if (loggingFileWriter != null)
            {
                loggingFileWriter.WriteLine("Collecting Stopped");
            }
        }

        /// <summary>
        /// Enables access to the raw data from the device in whatever
        /// units are native to the device.
        /// </summary>
        /// <returns>The raw data from the device in native
        /// format.</returns>
        public int[] GetRawData()
        {
            return dataCaptured;
        }

        private void GetMeasurements()
        {
            dataCaptured = new int[10];
            System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
            {
                int x = 0;
                Random timer = new Random();

                while (controller != null)
                {
                    System.Threading.Thread.Sleep
                        (timer.Next(1000, 5000));
                    dataCaptured[x] = controller != null
                        ? controller.TakeMeasurement()
                        : dataCaptured[x];
                    mostRecentMeasure = dataCaptured[x];

                    if (loggingFileWriter != null)
                    {
                        loggingFileWriter.WriteLine
                            ("Measurement Taken: {0}",
                            mostRecentMeasure.ToString());
                    }

                    x++;
                    if (x == 10)
                    {
                        x = 0;
                    }
                }
            });
        }

        protected Units unitsToUse;
        protected int[] dataCaptured;
        protected int mostRecentMeasure;
        protected DeviceController controller;
        protected DeviceType measurementType;

        // New fields and method to implement the logging
        // functionality.

        protected string loggingFileName;
        private TextWriter loggingFileWriter;

        /// <summary>
        /// Returns the file name of the logging file for the device.
        /// </summary>
        /// <returns>The file name of the logging file.</returns>
        public string GetLoggingFile()
        {
            return loggingFileName;
        }

        // New methods to implement the IDisposable interface.

        /// <summary>
        /// Dispose method required for the IDispose interface.
        /// </summary>
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {

            if (disposing)
            {
                // Check that the log file is closed; if it is not
                // closed, log a message and close it.

                if (loggingFileWriter != null)
                {
                    loggingFileWriter.WriteLine("Object Disposed");
                    loggingFileWriter.Flush();
                    loggingFileWriter.Close();
                    loggingFileWriter = null;
                }

            }

        }
    }
}
```

Task 5: Modify the MeasureMassDevice class to use logging

In this task, you will modify the existing **MeasureMassDevice** class to set the **loggingFileName** field when the class is instantiated.

1. Open the MeasureMassDevice.cs file:
   - In Solution Explorer, double-click **MeasureMassDevice.cs**.
2. In the **MeasureMassDevice** class, remove the comment **TODO: Modify the constructor to set the log filename based on a string parameter**, and then modify the constructor to take a *string* parameter called *logFileName*. In the body of the constructor, set the **loggingFileName** field to the *logFileName* parameter. You should also update the XML comments for the constructor to describe the new parameter.

   Your code should resemble the following code example.

```
/// <summary>
/// Construct a new instance of the MeasureMassDevice class.
/// </summary>
/// <param name="DeviceUnits">Specifies the units used natively by the
/// device.</param>
/// <param name="LogFileName">Specifies the required file name used
/// for logging in the class.</param>
public MeasureMassDevice(Units deviceUnits, string logFileName)
{
    unitsToUse = deviceUnits;
    measurementType = DeviceType.MASS;
    loggingFileName = logFileName;
}
```

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

Exercise 2: Managing Resources Used by an Object

Task 1: Open the starter project

- Open the Module9 solution from the E:\Labfiles\Lab 9\Ex2\Starter folder. This solution contains the completed code from Exercise 1 and skeleton code for Exercise 2:
    a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
    b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9\Ex2\Starter** folder, click **Module9.sln**, and then click **Open**.

Task 2: Test the logging functionality by using the test harness

1. Run the Exercise2 Test Harness application:
    - On the **Debug** menu, click **Start Debugging**.
2. Click **Get Measurements**. This action causes the application to pause for 20 seconds while some measurements data is generated and then display this data. This pause is necessary because the application waits for measurement data from the emulated device.

Note that the measurement data is logged to the E:\Labfiles\Lab 9 \LogFile.txt file by default.

3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.
4. Using Notepad, open the LogFile.txt file in the E:\Labfiles\Lab 9 folder:
   a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.
   b. In Notepad, on the **File** menu, click **Open**.
   c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.
5. Review the contents of the LogFile.txt file.

   The file is empty. Although the application has retrieved values from the emulated device and written them to the log file, the **TextWriter** object caches data in memory and writes to the underlying file system when it is either flushed or closed. When you closed the application, you disposed of the **TextWriter** object without flushing its in-memory cache to the log file, which is why the file is empty.
6. Close Notepad:
   - On the **File** menu, click **Exit**.
7. Run the Exercise2 Test Harness application again, click **Get Measurements**, and then wait for the data to appear:
   a. In Visual Studio, on the **Debug** menu, click **Start Debugging**.
   b. In the Exercise 2 window, click **Get Measurements**.
8. After the application populates the text boxes with data from the emulated device, click **Get Measurements** again.

   The application will throw an unhandled **IOException** exception. The exception is thrown because each time you click **Get Measurements**, you create a new instance of the **MeasureMassDevice** class. Each instance of the **MeasureMassDevice** class creates its own instance of the **TextWriter** class to log measurements. The test harness does not currently dispose of the **MeasureMassDevice** objects after the code run by the **Get Measurements** button completes. This means that the object is not closed and therefore retains its lock on the log file. When you attempt to create a second instance of the **MeasureMassDevice** class that uses the same log file, this instance cannot access the file because it is still in use by the first instance.

9. Stop the Exercise 2 application:

   - On the **Debug** menu, click **Stop Debugging**.

Task 3: Modify the test harness to dispose of objects correctly

1. In Visual Studio, open the MainWindow.xaml.cs file in the Exercise2 Test Harness project:

   - In Solution Explorer, expand the **Exercise2 Test Harness** project, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.

2. In the **createInstance_Click** method, remove the **TODO: Modify this method** comment in the MainWindow.xaml.cs file. Modify the **createInstance_Click** method to ensure that the **device** field is disposed of when the method completes by using a **using** block.

   Your code should resemble the following code example.

```
private void createInstance_Click(object sender, RoutedEventArgs e)
{
    using (MeasureMassDevice device =
        new MeasureMassDevice(Units.Metric,
            @"E:\Labfiles\Lab 9\LogFile.txt"))
    {
        device.StartCollecting();
        loggingFileNameBox.Text = device.GetLoggingFile();
        System.Threading.Thread.Sleep(20000);
        metricValueBox.Text = device.MetricValue().ToString();
        imperialValueBox.Text = device.ImperialValue().ToString();
        rawDataValues.ItemsSource = device.GetRawData();
        device.StopCollecting();
    }
}
```

3. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Verify that the object is disposed of correctly

1. Run the Exercise2 Test Harness application:

   - On the **Debug** menu, click **Start Debugging**.

2. Click **Get Measurements**, and then wait until the data appears.

3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.

4. Open Notepad and examine the log file:

   a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.

   b. In Notepad, on the **File** menu, click **Open**.

   c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.

5. Review the contents of the log file.

   The file now contains the values displayed on the form and status messages generated when the file is opened and closed. When the code for the **Get Measurements** button completes, it now disposes of the **MeasureMassDevice** instance, which forces the **TextWriter** object to flush its in-memory cache to the file, and then closes the **TextWriter**.

6. Close Notepad:

   - On the **File** menu, click **Exit**.

7. Run the Exercise2 Test Harness application again:

   - On the **Debug** menu, click **Start Debugging**.

8. Click **Get Measurements**, and then wait for the data to appear.

9. In the Exercise 2 window, click **Get Measurements** again. The application will pause for another 20 seconds.

   This time, the application does not throw an exception. This is because the resources are properly disposed of each time you click **Get Measurements**. When you close the **TextWriter** object, you release the lock on the file, and a new instance of the **TextWriter** class can now use the same log file without throwing an exception.

10. Open Notepad and examine the log file:

    a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.

    b. In Notepad, on the **File** menu, click **Open**.

    c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.

11. Review the contents of the log file.

The file contains the most recent values displayed on the form.

12. Close Notepad:

    - On the **File** menu, click **Exit**.

13. Close the Exercise 2 window.
14. Close Visual Studio:

    - On the **File** menu, click **Exit**.