A Complete Example of VisualAPL Application Development

This document and associated Visual Studio 2008 solution provides a start-to-finish demonstration of VisualAPL application development. Although the application system developed is very simple, the example can be extended to the development of other application systems.

A. End user description of the requirements
   Provide a system to calculate the curtate expectation of life of an individual based on their current age and a selected mortality table.

B. Application system architecture
   Divide the application system into modules for GUI, business rules (calculations) and data.

C. Implementation decisions
   GUI: Use C# WPF for the GUI which is the most recent generation of GUI development tools in Microsoft Visual Studio 2008. Other GUI development options are available, including Windows Forms (the previous generation of Visual Studio GUI development tools), legacy □wi forms supported by VisualAPL, HTML (possibly with JavaScript or Ajax) ASP/ASPX forms, .pdf Forms, etc. Check out these VisualAPL forum threads for additional information:
   - http://forum.apl2000.com/viewtopic.php?t=471
   - http://forum.apl2000.com/viewtopic.php?t=446

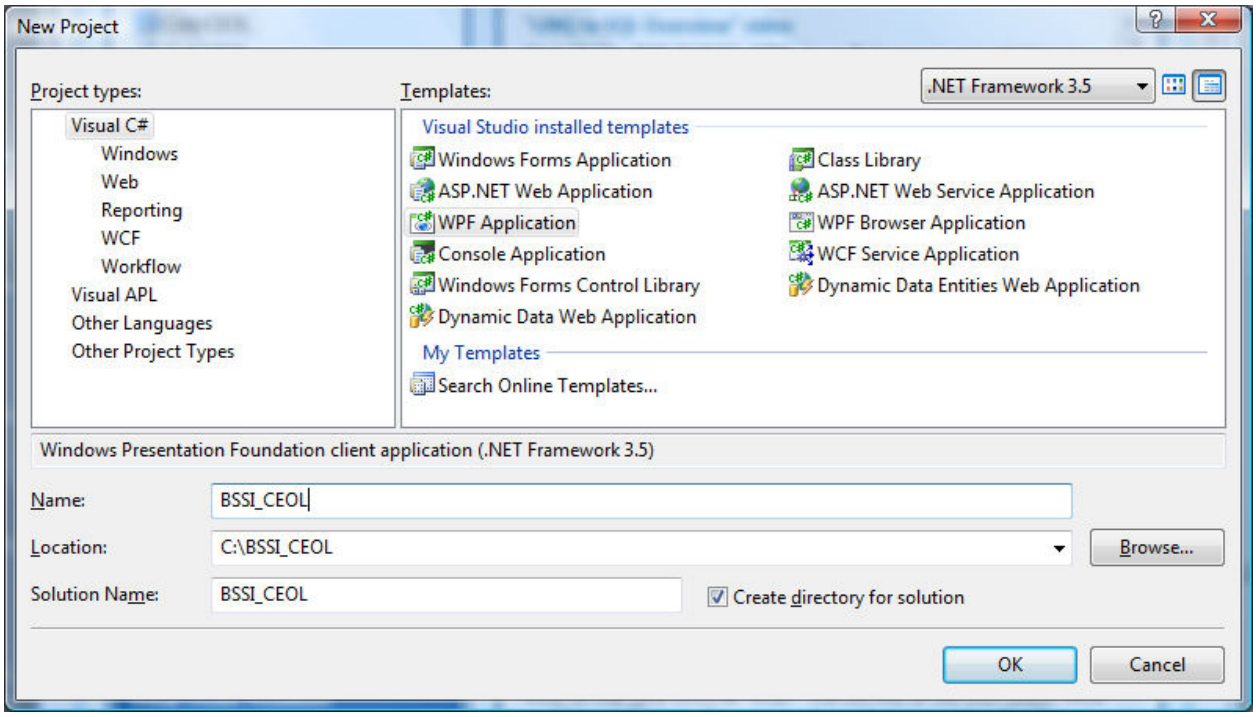
   Calculations: Use VisualAPL for the business rules and calculations. Other options for developing the business rules and calculations module might be C#, APL+Win running behind APLNext WebServices, VB.Net, etc.

   Data: Use a Microsoft Excel workbook for the “static” data source which is convenient because the Society of Actuaries (North America) provides mortality tables in this format. Other options for the data module might be Microsoft SQL Server, IBM DB2, Oracle, Microsoft Access, APL component files, etc.

D. Prepare a Microsoft .msi deployment project to support local installation.

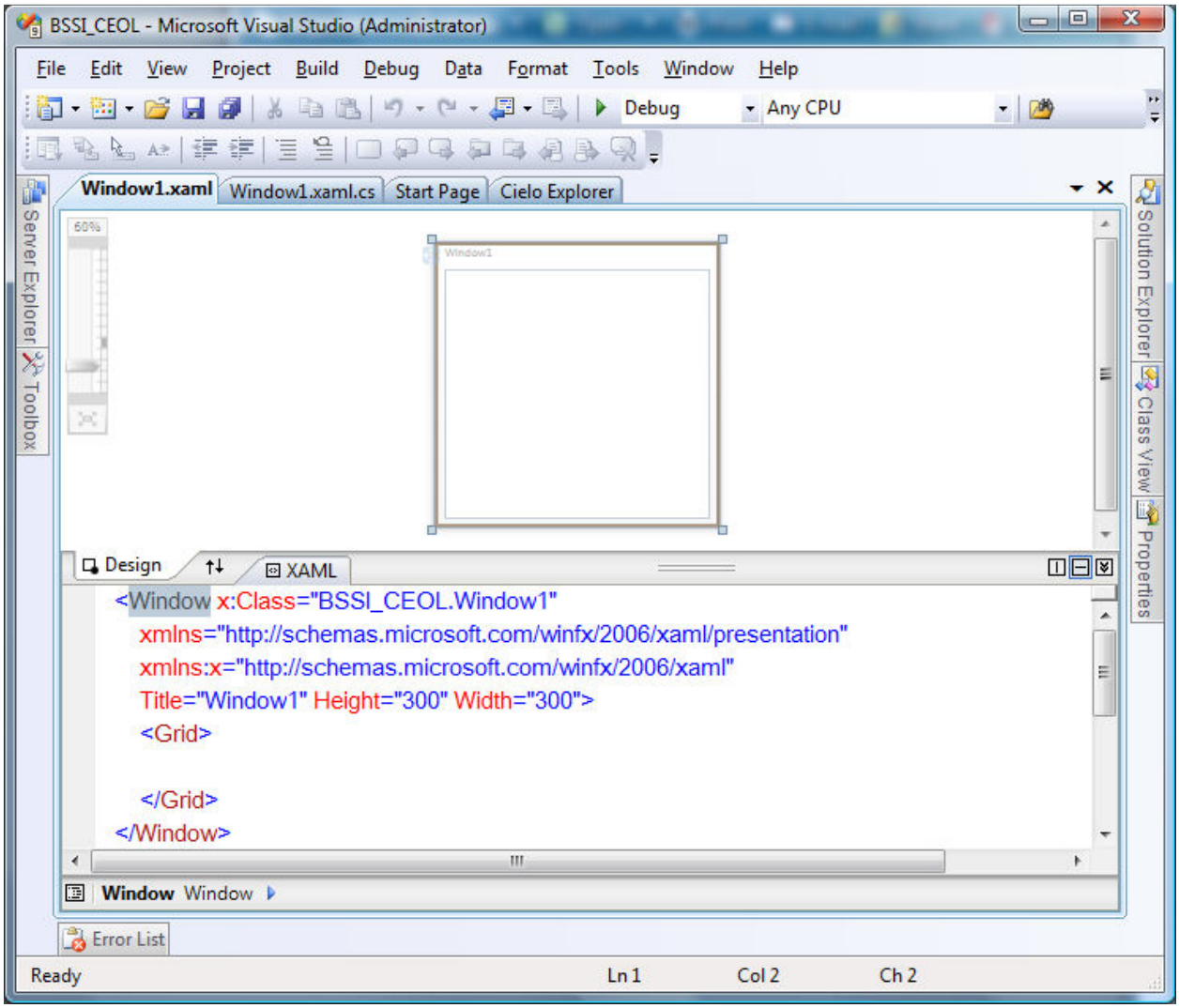
Create the Visual Studio 2008 Solution

Use the Visual Studio 2008 “File > New > Project > Visual C# > WPF Application” menu item to create a new Visual Studio 2008 solution. A C# WPF Application project type is chosen because WPF (Windows Presentation Foundation) will be used for the application system’s GUI (graphical user interface). As the solution development progresses, a VisualAPL class library project will be added to this solution to support the application system’s calculations and business rules.

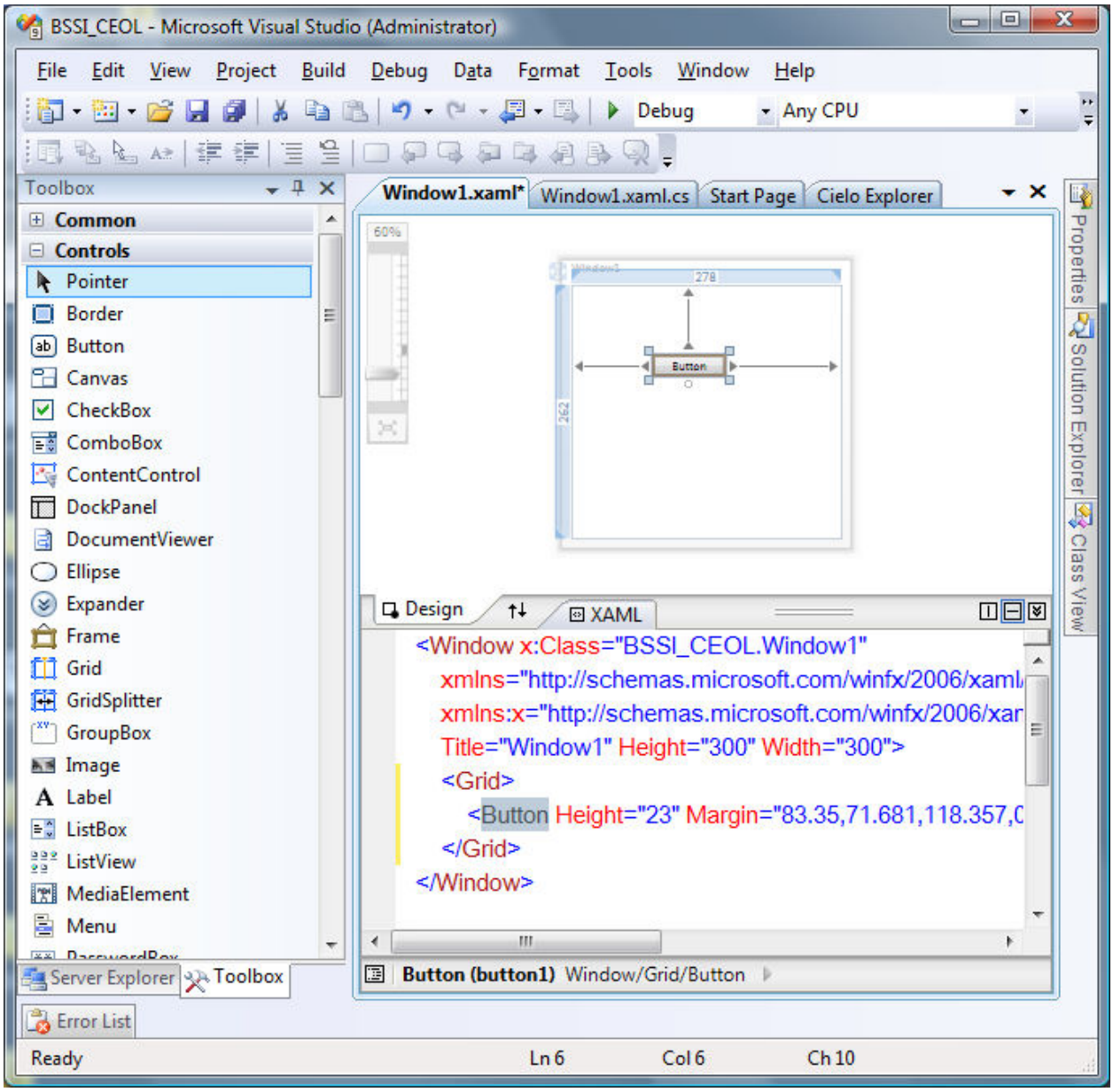


Create the WPF GUI

Visual Studio 2008 displays the default XAML file, Window1.xaml, describing the GUI and renders this XAML to the window above the XAML text. The code-behind-file, Window1.xaml.cs, will contain the event handlers for events defined for the GUI controls included in the XAML. Notice that the <Window …> XAML code block encloses the entire GUI specification, ending (in proper XML syntax) with the </Window> XAML code line. Also notice that the <Window …> XAML code line is displayed with separate lines for the x:Class, xmlns, (Title, Height and Width) property value specifications. These could have been displayed on a single line if desired. Additional property value specifications could be added to the <Window…> XAML code line, e.g. FontSize="16", which would apply to the entire GUI unless modified in a subordinate control of the GUI.

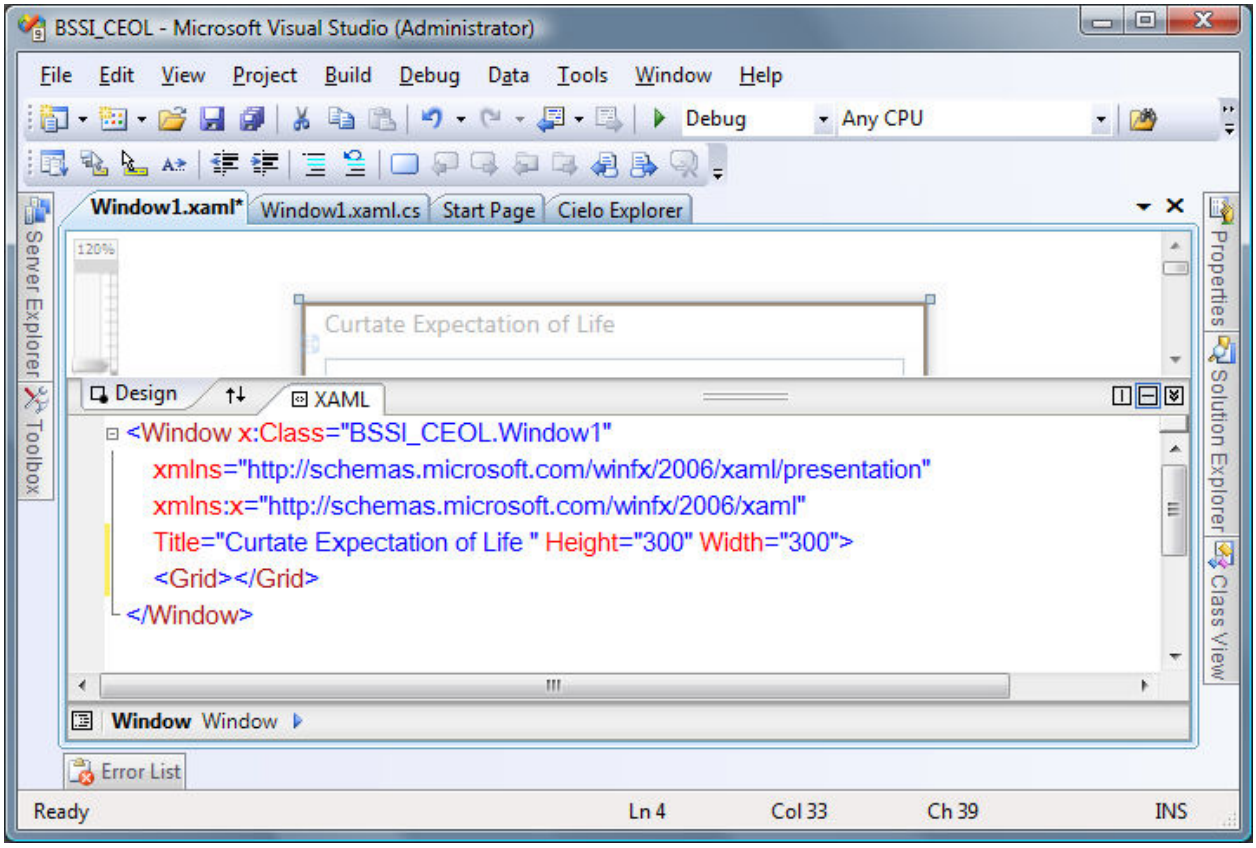


The WPF GUI can be programmed using the “Drag and Drop” method using the Visual Studio 2008 “Toolbox”. The resulting GUI employs fixed positioning of the controls and numerous other Microsoft-designed defaults.

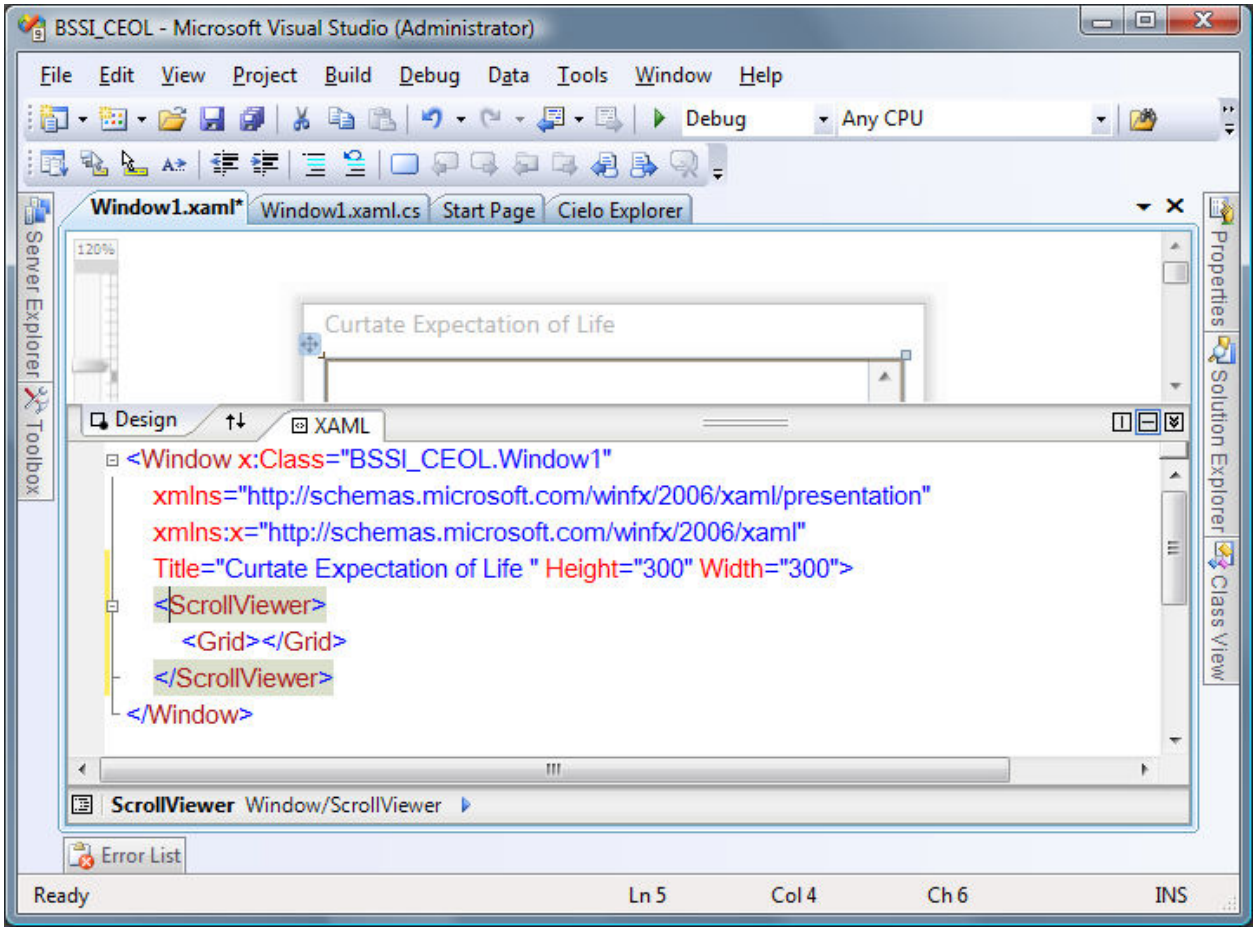


Alternatively, the programmer can type in the XAML statements. Typing XAML is illustrated in the development of the GUI for this solution. As the XAML is edited, Visual Studio 2008 updates the rendering of the GUI on the screen. Typing errors, if any, are noted by Visual Studio 2008 under the Error List tab.

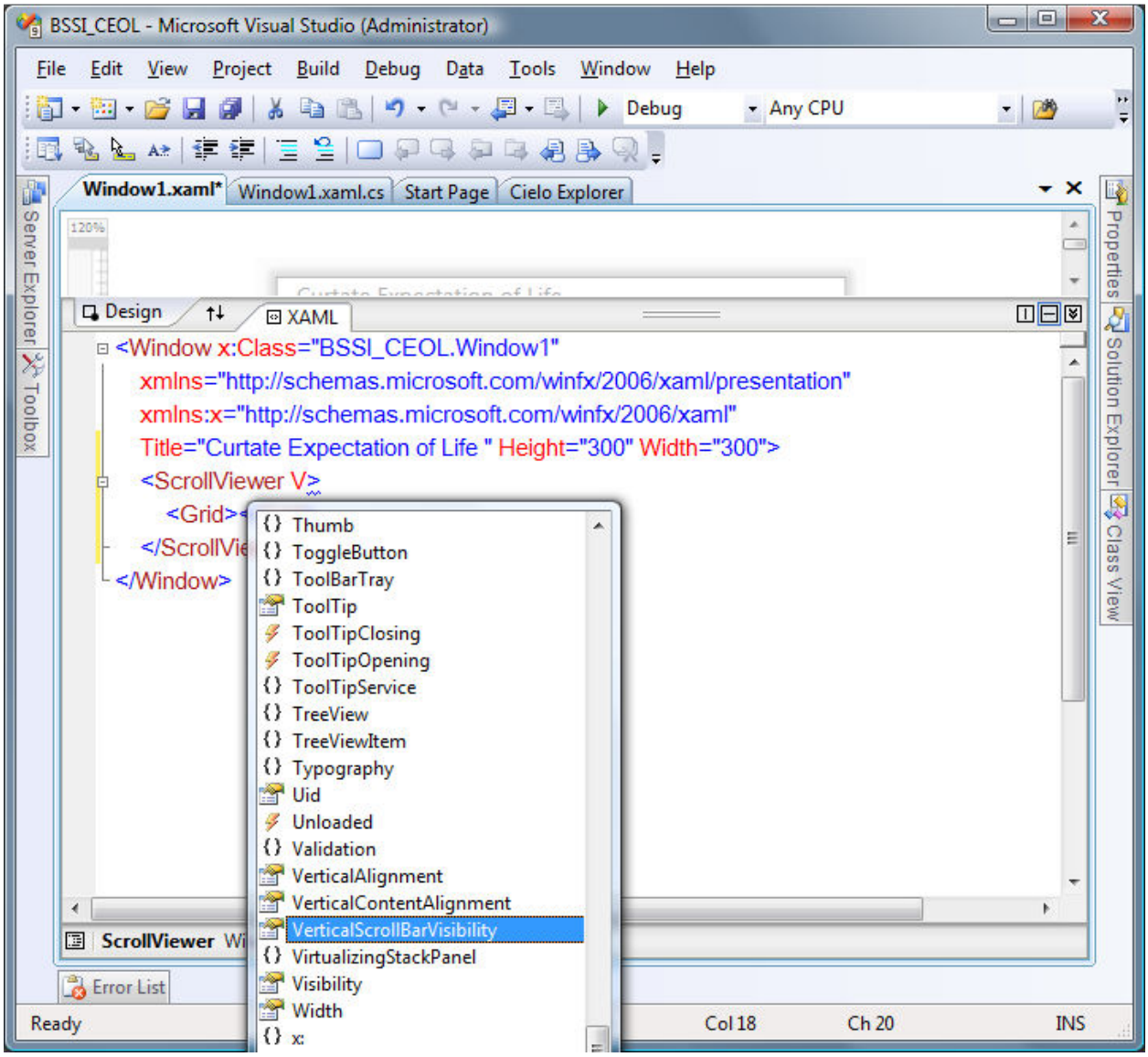
Edit the Title property of the topmost “Window” control of the GUI to “Curtate Expectation of Life”:



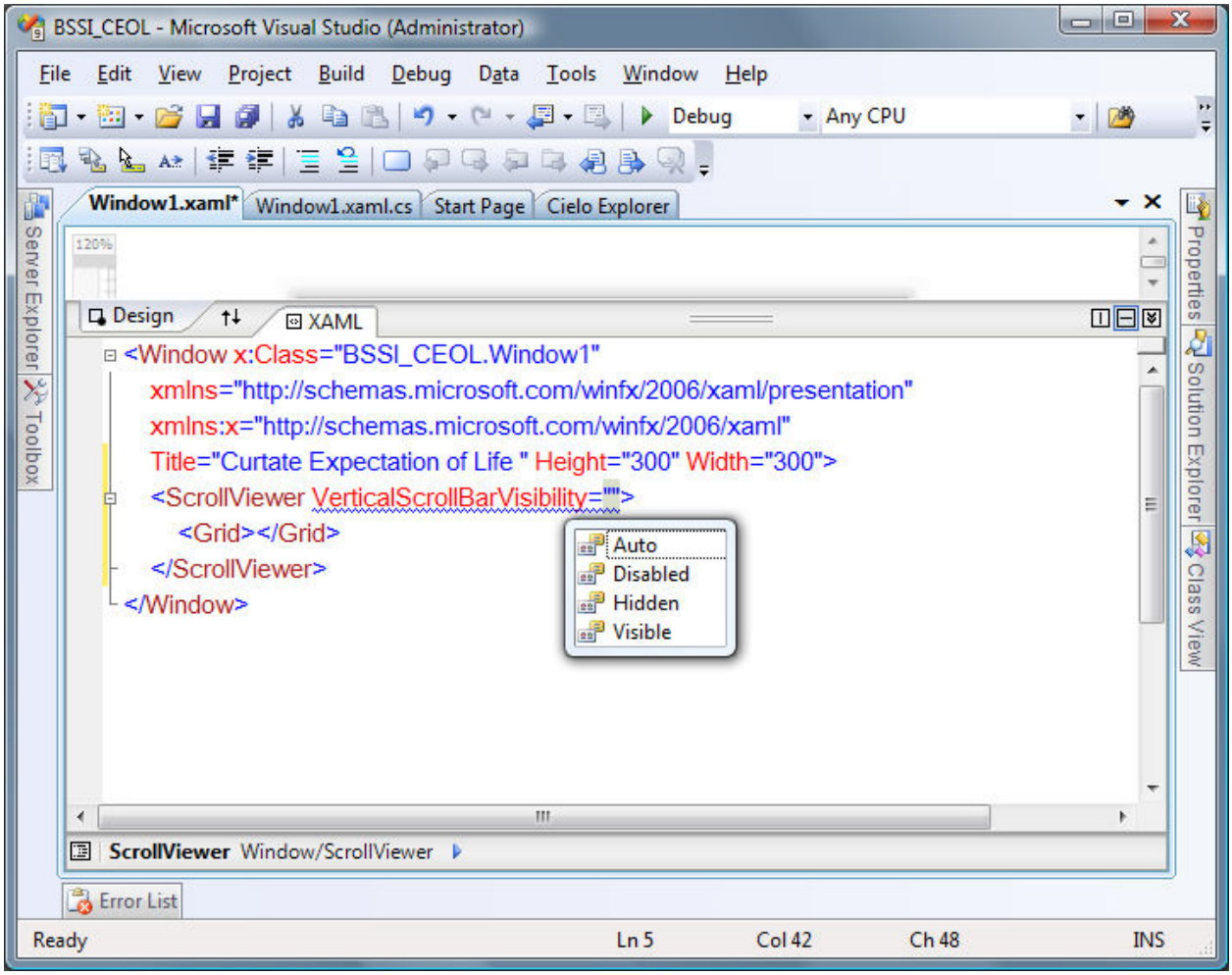
Because the entire GUI is to be resizable add a ScrollViewer control which will contain the Grid control:



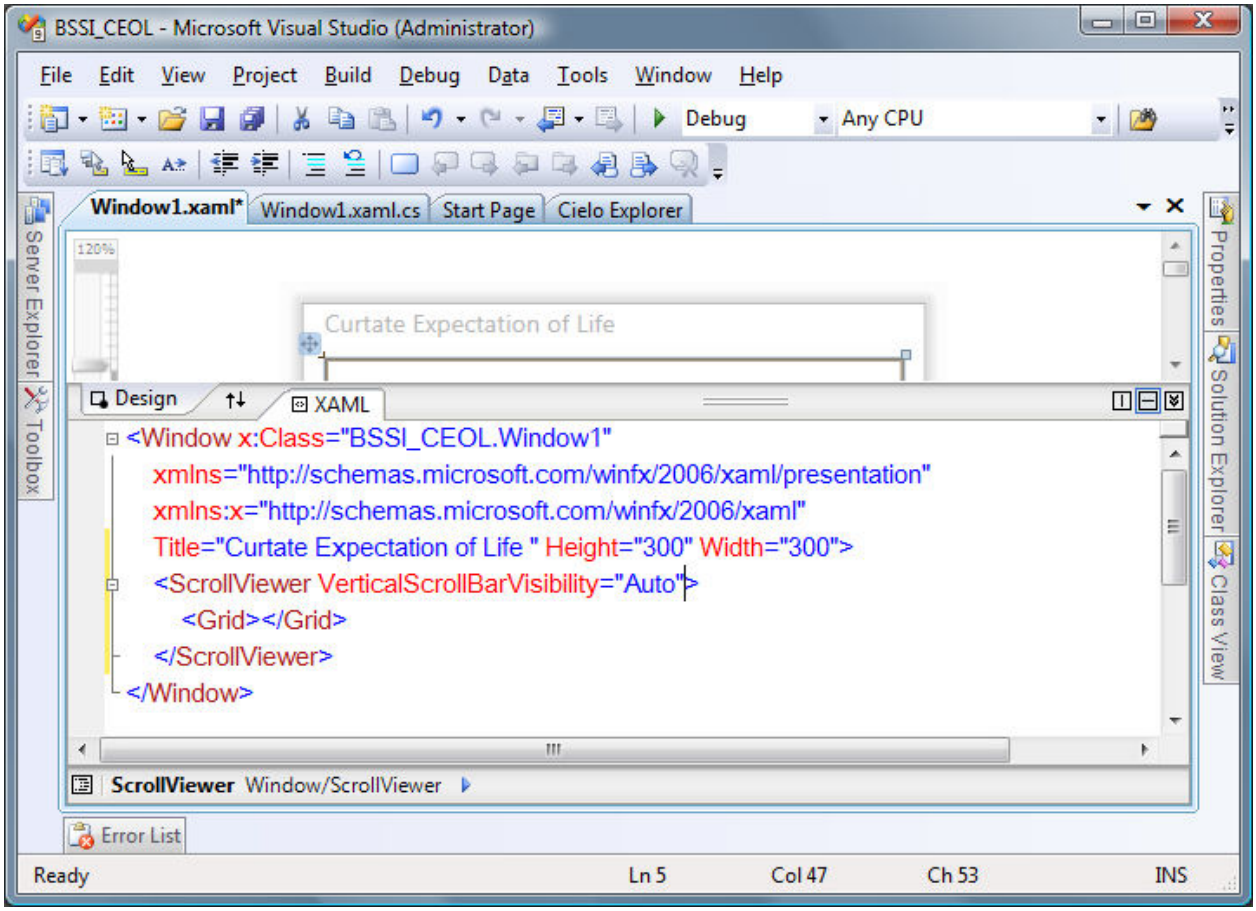
When typing XAML, adding control property values or event handlers to the XAML is done by positioning the cursor in the XAML immediately after the name of a control, a previously-set property value or an event handler specification, and pressing the SpaceBar key. Visual Studio 2008 Intellisense will present a list of the available properties and events. In this case the programmer sets the value of the VerticalScrollBarVisibility property for the ScrollViewer control:



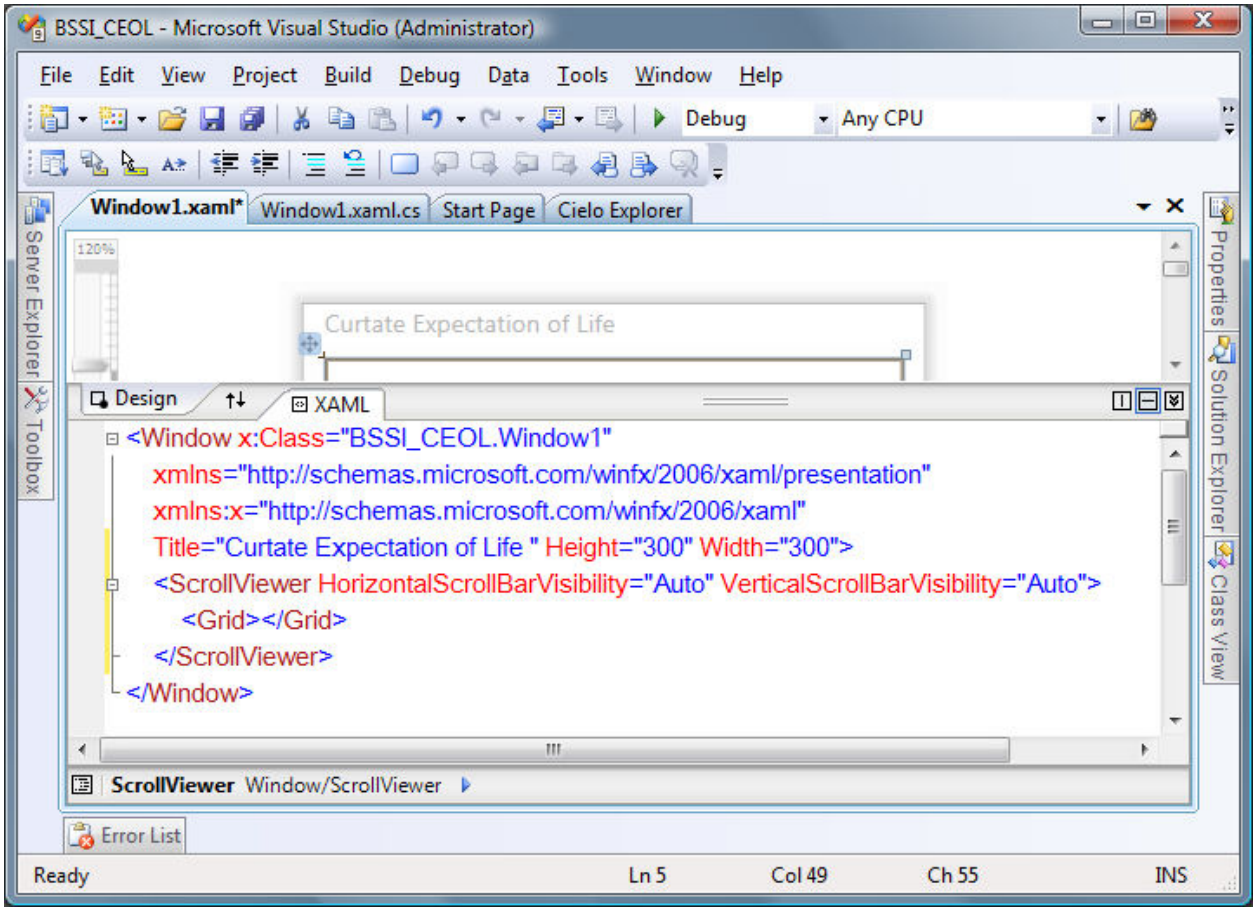
When the property name has been selected from the Intellisense list, if applicable, Intellisense will present the list of possible property values:



Now the ScrollViewer control's VerticalScrollBarVisibility property value has been set to Auto:

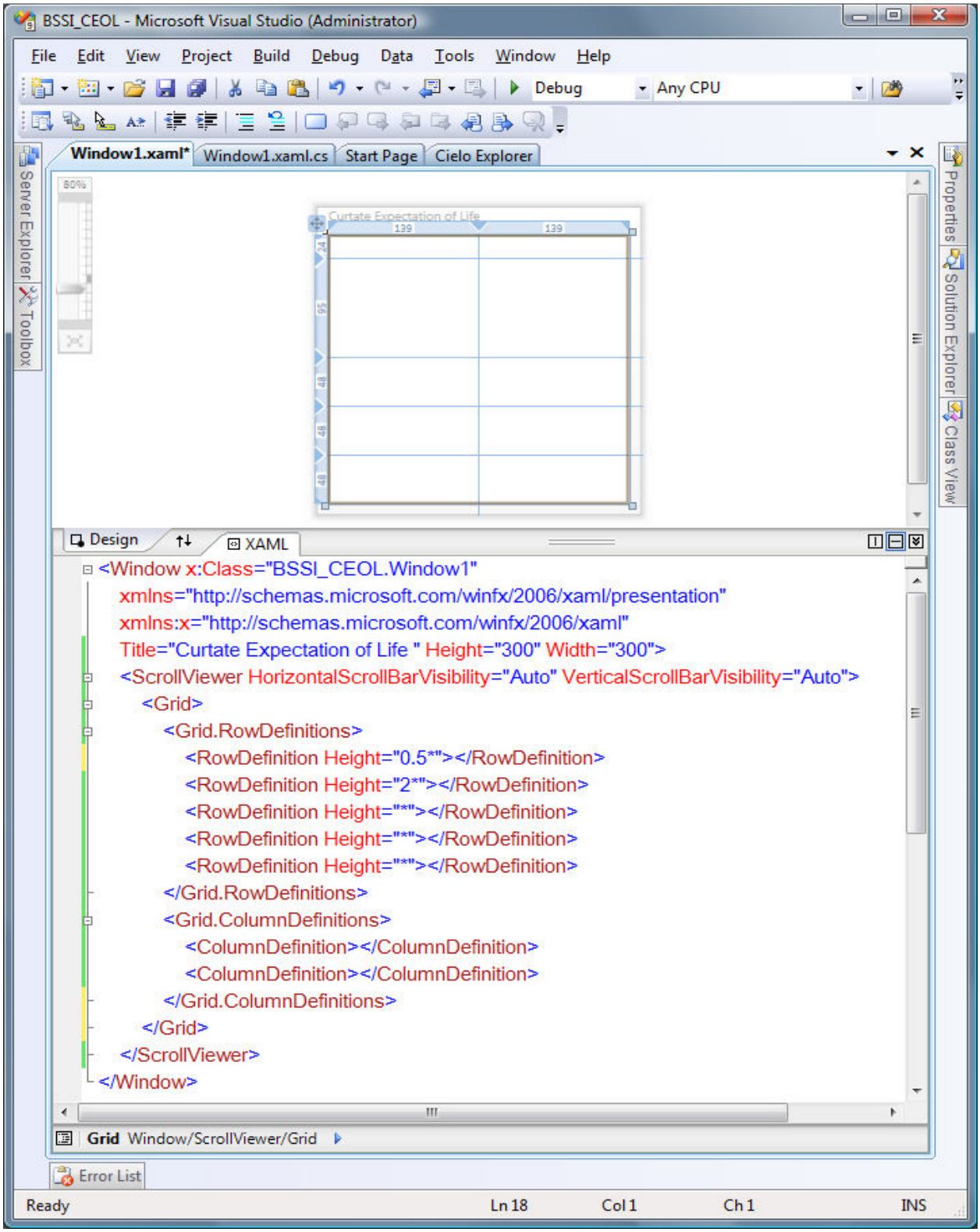


Similarly the programmer has set the ScrollViewer's HorizontalScrollBarVisibility property value to Auto:

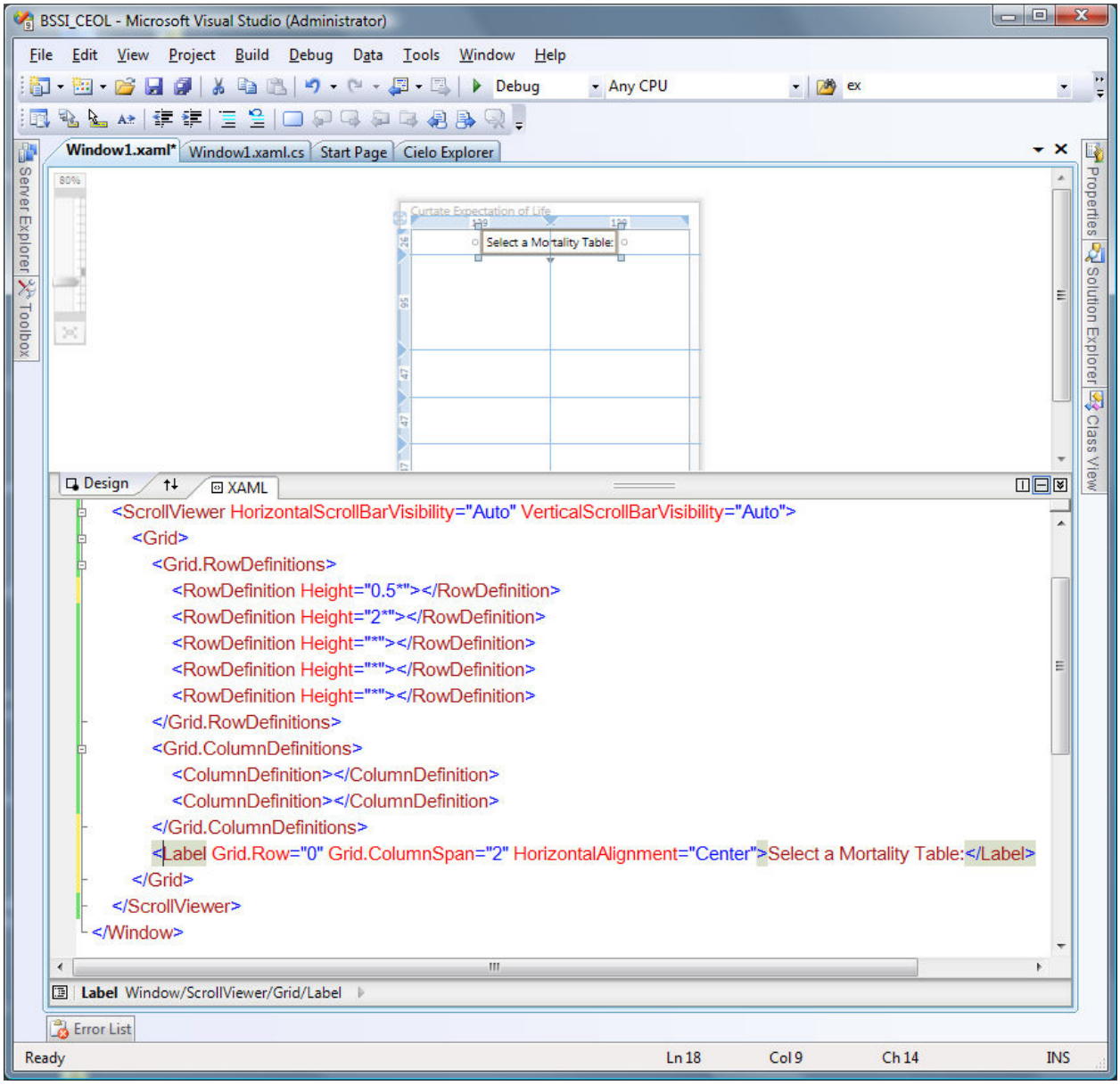


To contain the other controls for this GUI, the WPF Grid control will be used. There are many other WPF container controls. A definitive WPF reference is MacDonald, Matthew: *Pro WPF in C# 2008* Apress Publishing (http://www.apress.com/book/catalog?category=38).

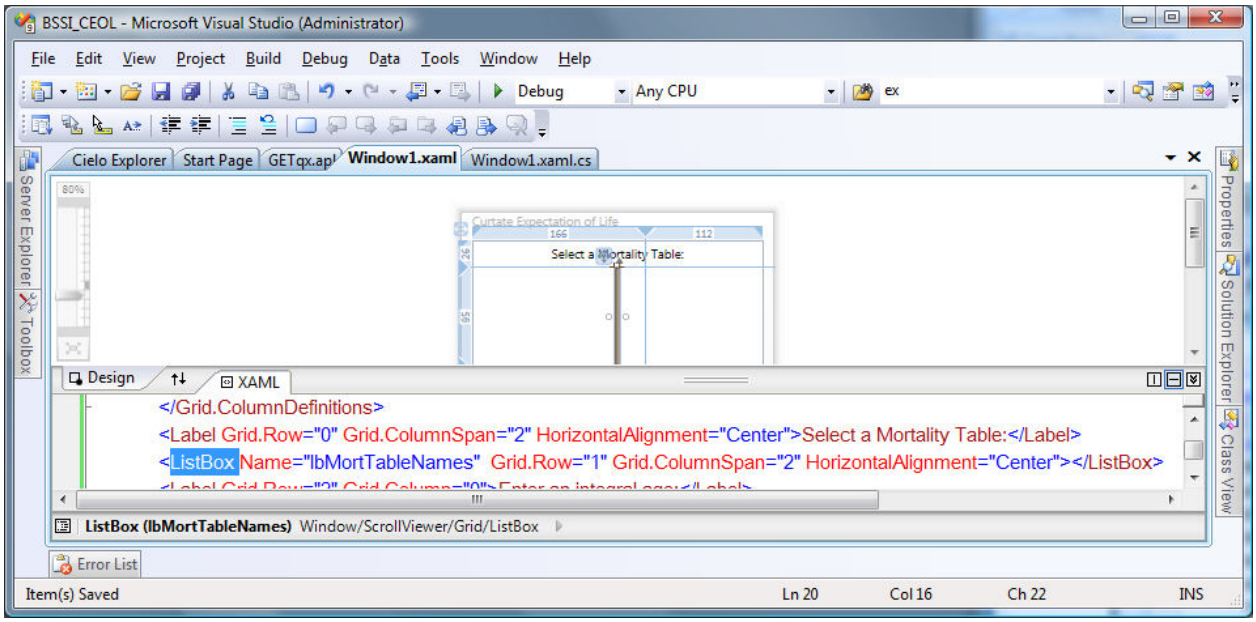
The Grid control XAML specification has been expanded to have five rows with proportional heights and two columns:



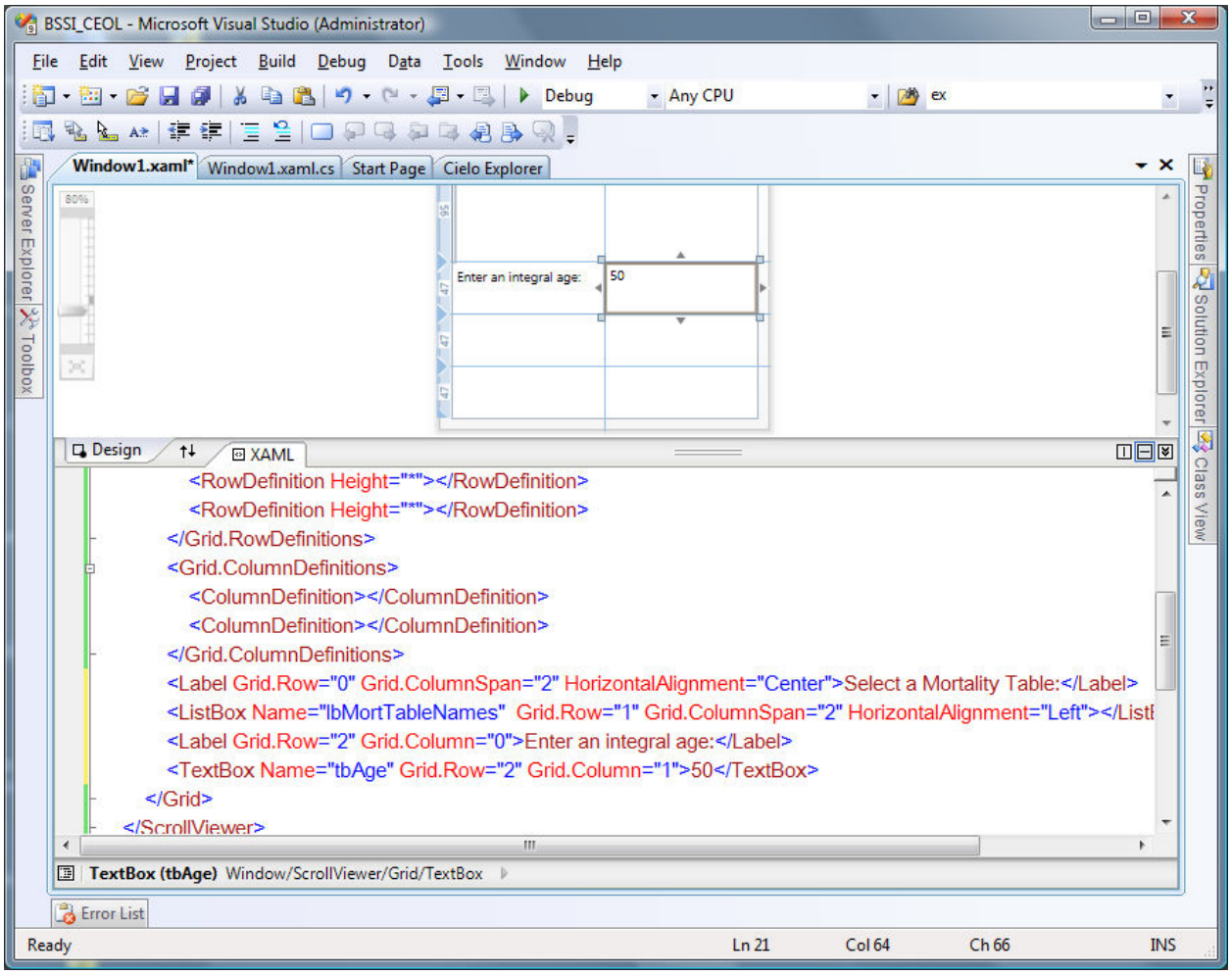
A Label control, spanning both columns of the Grid control is centered in row #0 of the Grid control with content “Select a Mortality Table”:



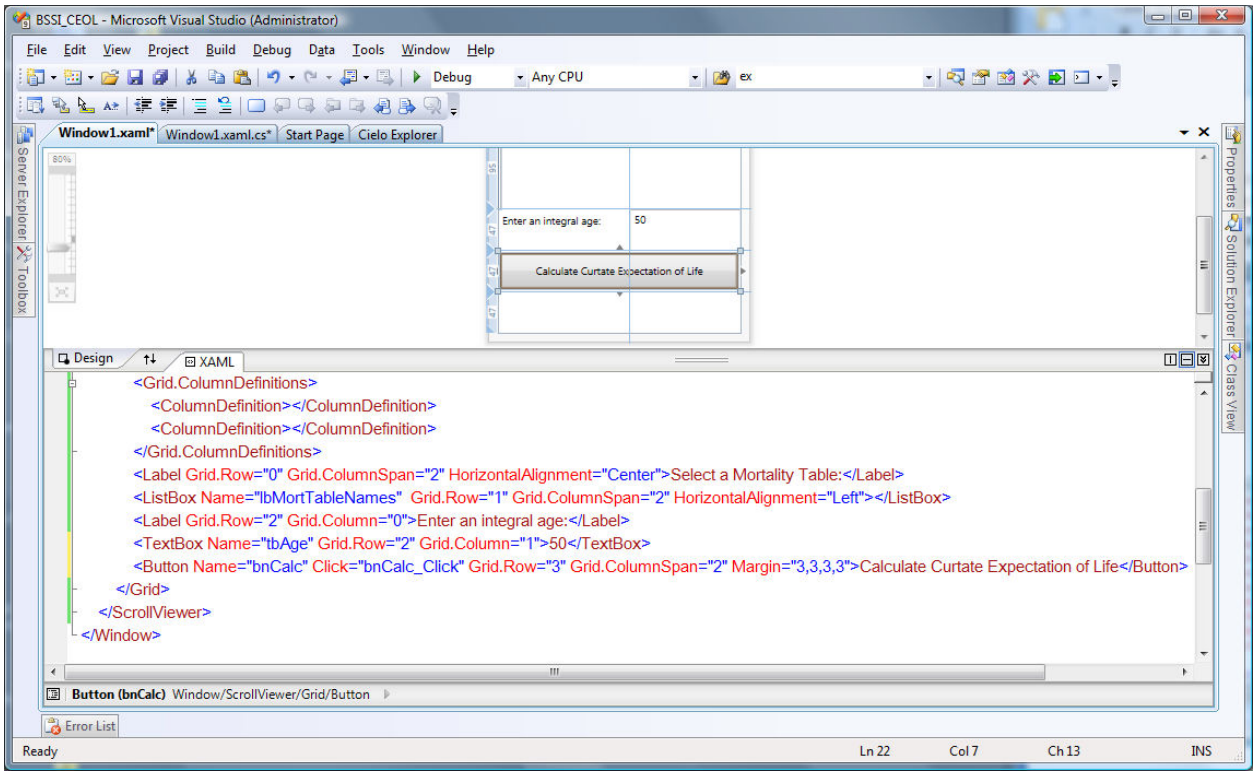
A ListBox control, spanning both columns of the Grid control is centered in row #1 of the Grid control. The name property value has been specified so that the code-behind file “Window1.xaml.cs” can fill the ListBox with the mortality table names:



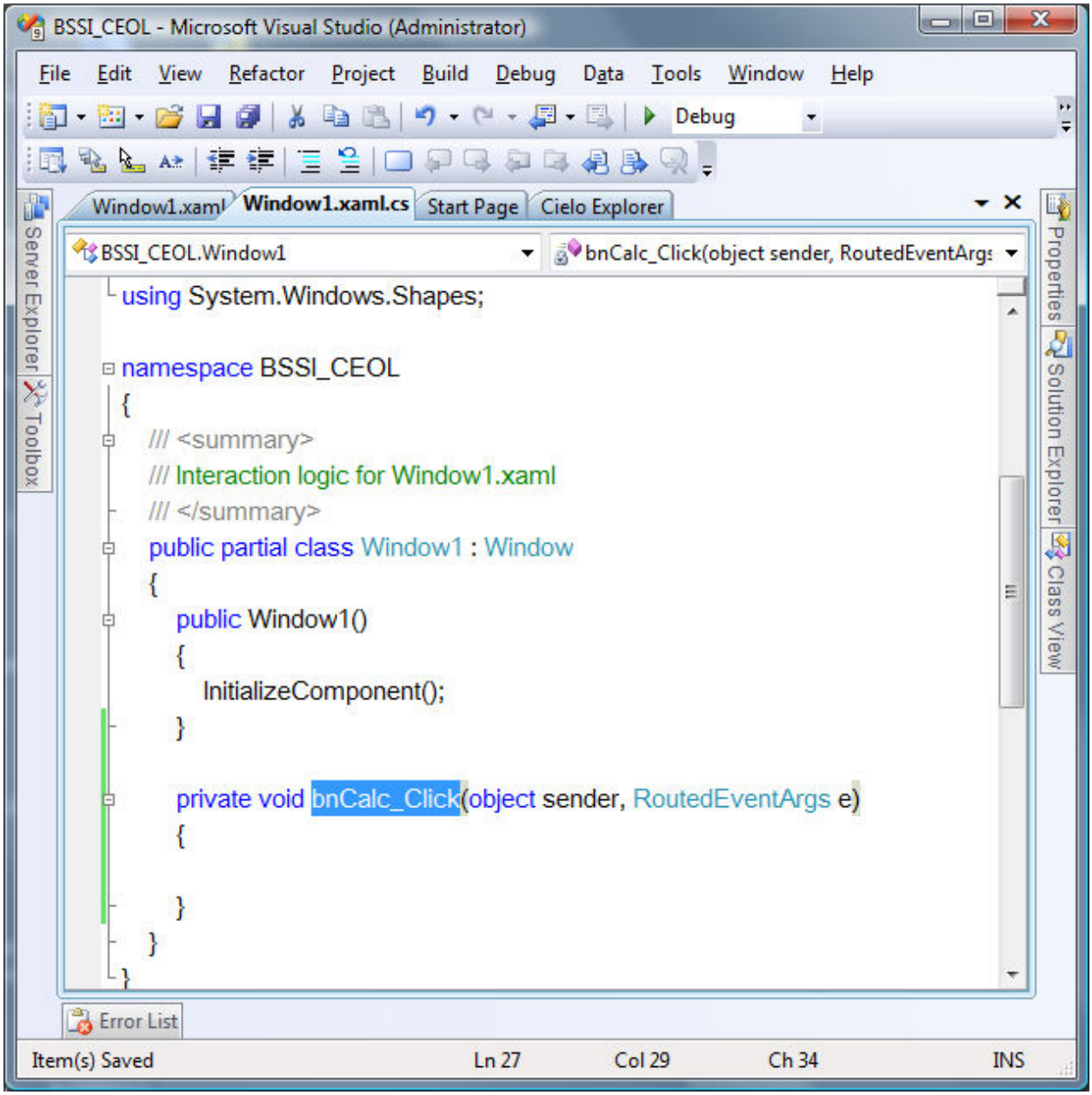
A Label control is placed in Grid Row #2 and Grid Column #0 with content “Enter an integral age:”. A TextBox control is placed in Grid Row #2 and Grid Column #1 with default content (Text property value) “50”. The TextBox control has the Name property specified so that the code-behind file “Window1.xaml.cs” can access the user entry from the Text property of this control:



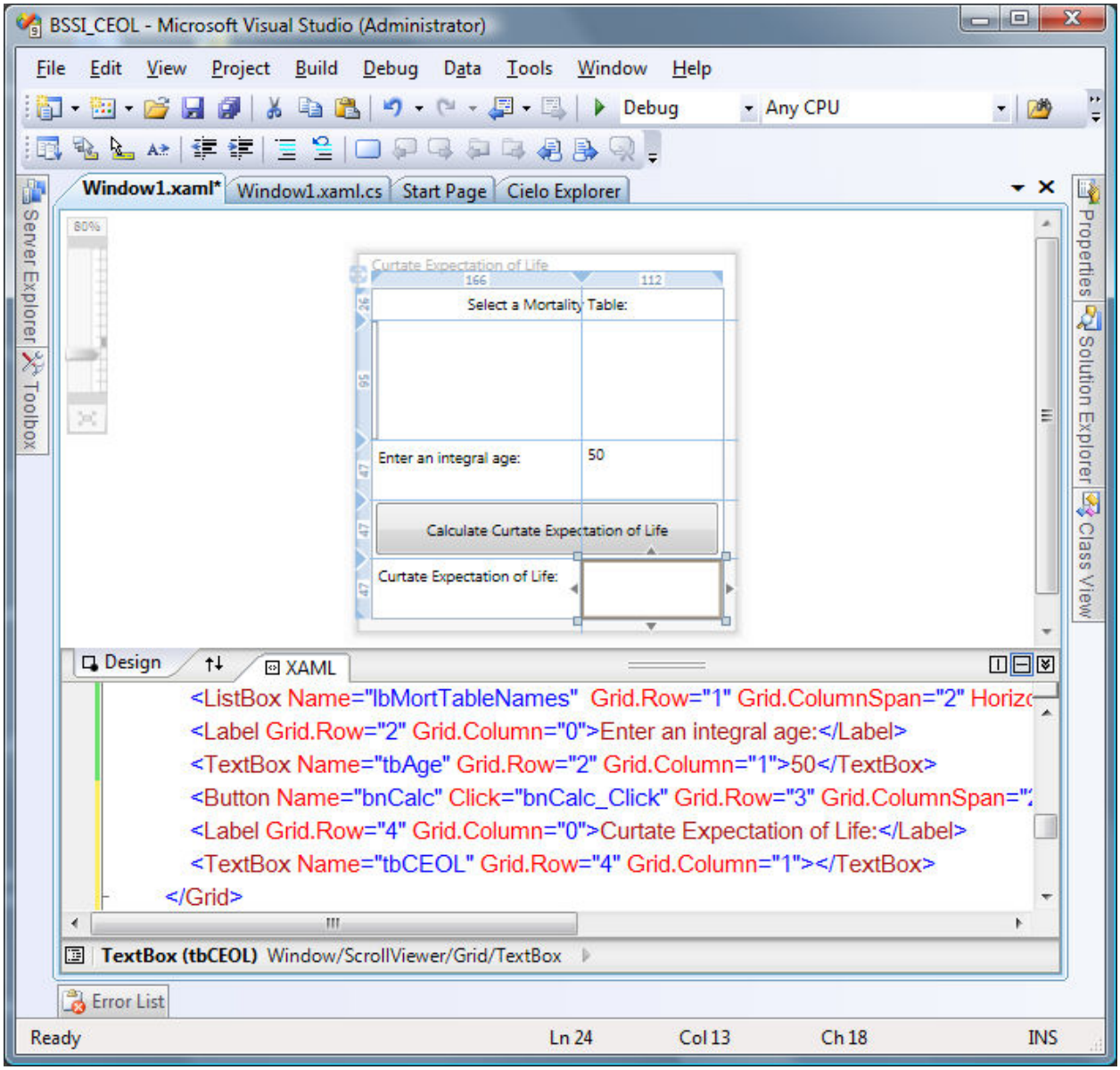
A Button control is placed in Grid Row #3 spanning two Grid Columns with content “Calculate…”. The Button control name property value is set so that the specified Click event handler function name is mnemonic:



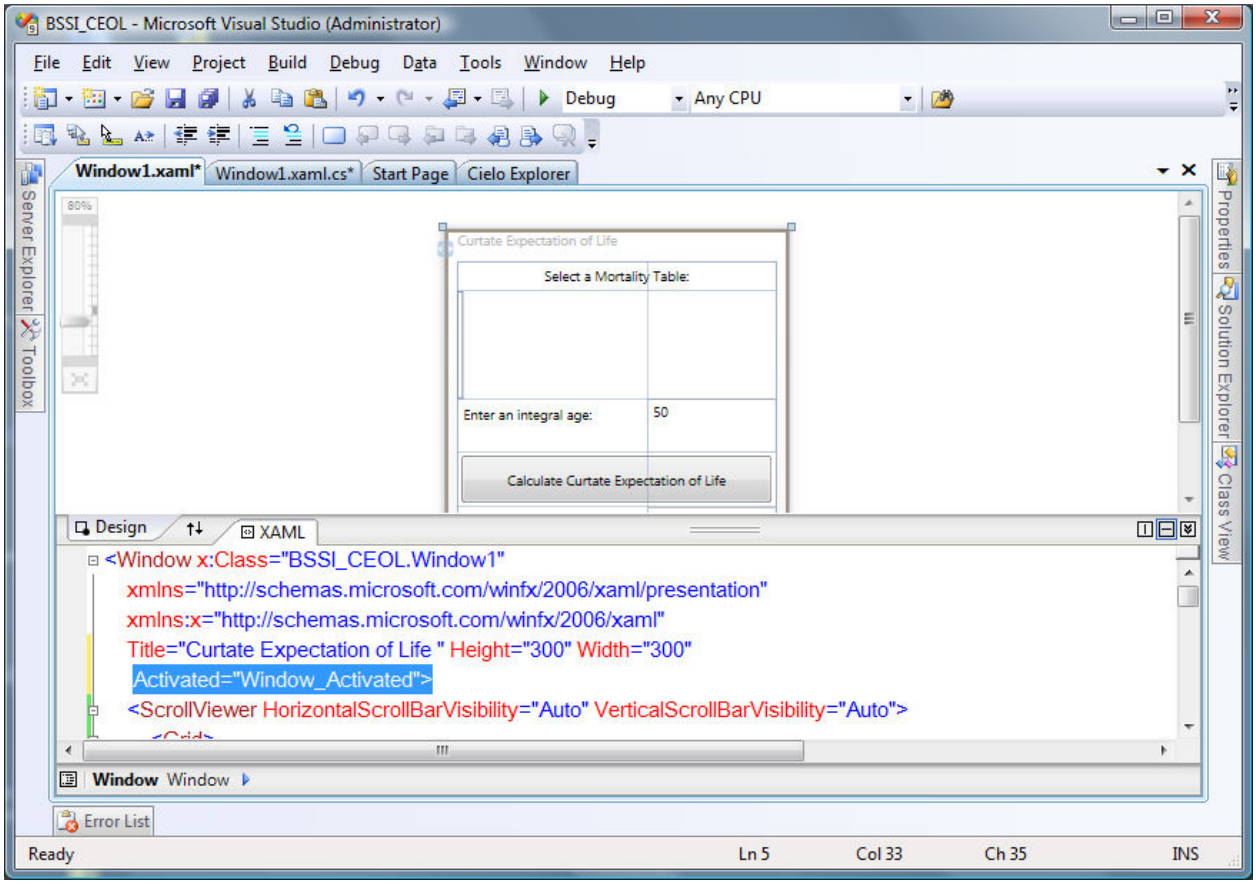
Visual Studio 2008 automatically creates an event handler function in the code-behind file "Window1.xaml.cs" when the Click event handler function name is specified in the XAML:



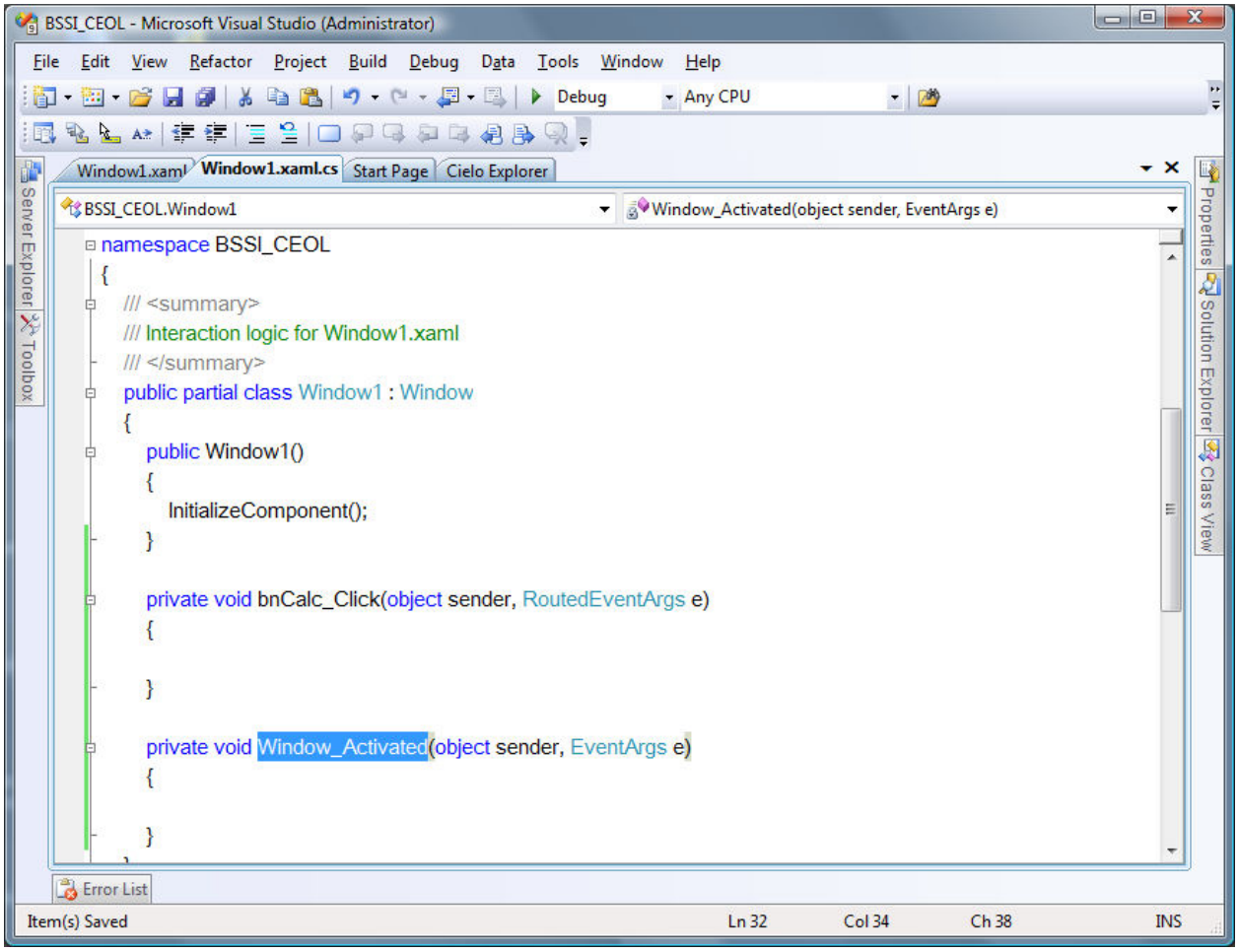
A Label control is placed in Grid Row #4 and Grid Column #0 with content “Curtate Expectation of Life:”. A TextBox control is placed in Grid Row #4 and Grid Column #1 with no default content. The TextBox control has the Name property specified so that the code-behind file “Window1.xaml.cs” can set the Text property with the value calculated by this solution:



The ListBox elements (mortality table names) will be filled when the WPF Window control is activated, so the Activated event handler's function name is specified in the XAML:



Visual Studio 2008 automatically creates an event handler function in the code-behind file "Window1.xaml.cs" when the Activate event handler function name is specified in the XAML:



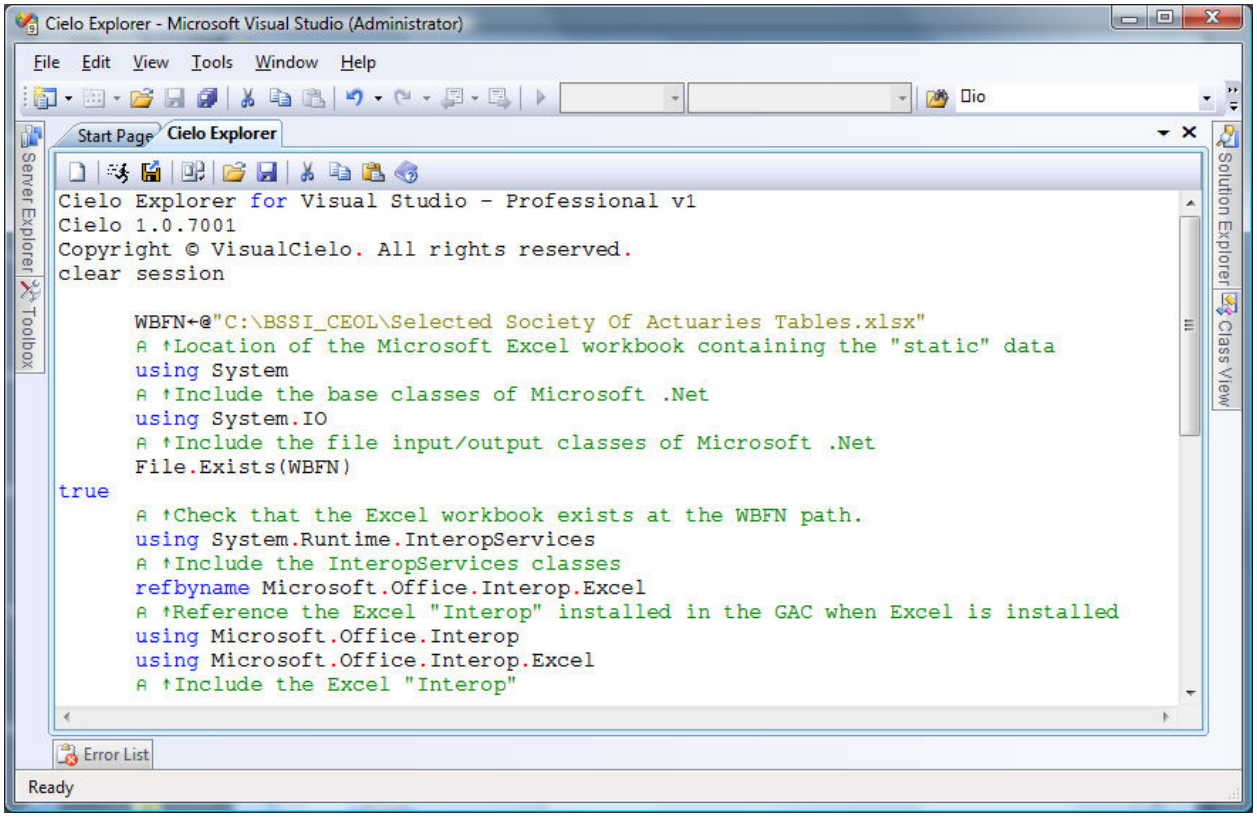
Remember to frequently use the Visual Studio 2008 "File > Save All" to update the solution files on disk.

Create the VisualAPL Application System Business Rules (Calculations)

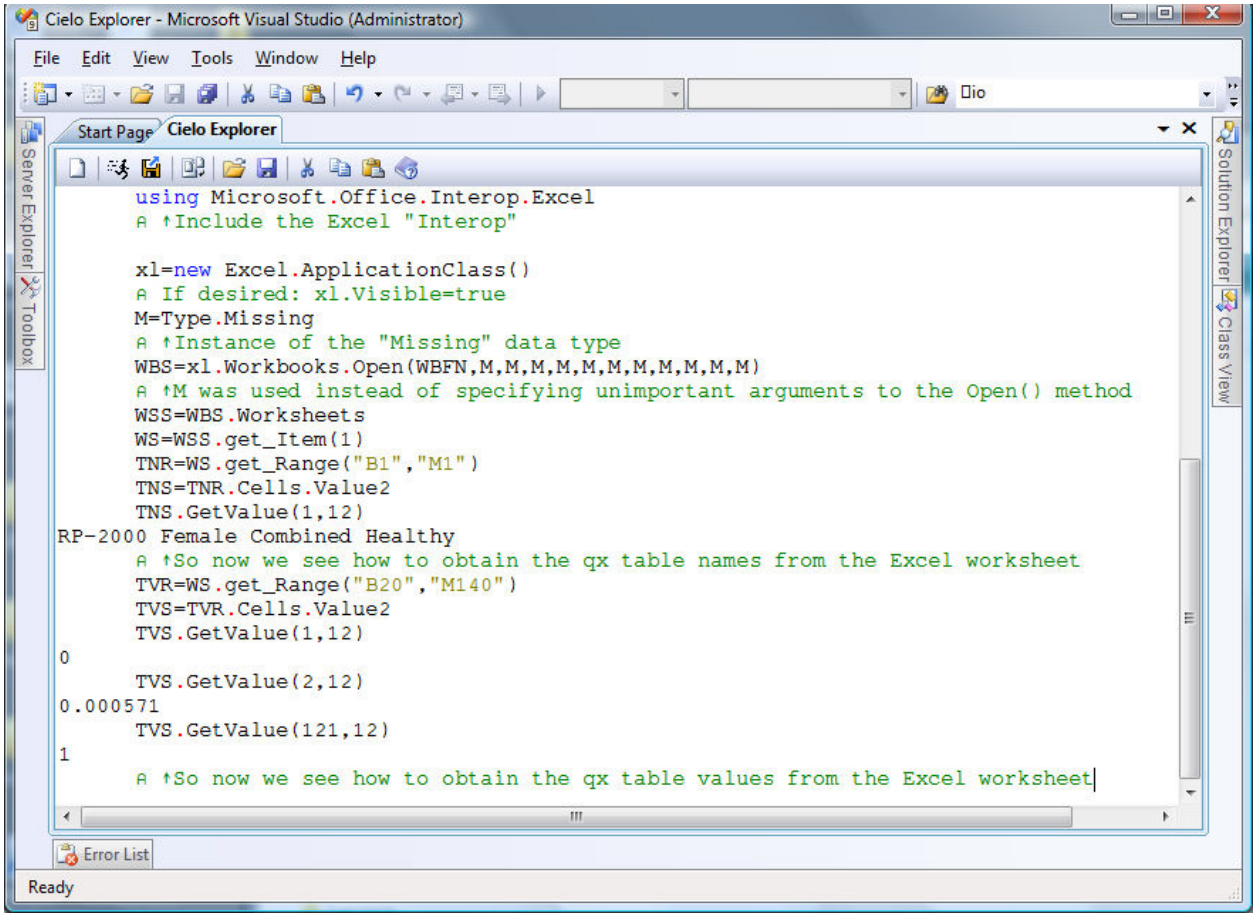
The “static” data are tables of actuarial qx (probability of death within the year for a person aged x) are obtained from the Society of Actuaries (North America). For purposes of this application system they are stored in a worksheet of a Microsoft Excel workbook. Other data storage methodologies are possible for this “static” data. Using a Microsoft Excel workbook in this project provides an opportunity to illustrate accessing Microsoft Excel from VisualAPL. Microsoft Excel is currently a Win32-based Microsoft product, so using the Microsoft-provided “Interop” means that from .Net Excel is accessed as an ActiveX object.

To understand the concept of accessing Microsoft Excel from VisualAPL, the Cielo Explorer immediate-mode, inter-active session is ideal. Using the Cielo Explorer makes it easy to see results and correct errors immediately. In the Cielo Explorer session, the applicable .Net references are first established so that the appropriate .Net tools are available to access Excel from the session.

It may be necessary to download the Microsoft Excel “primary interop assemblies” to the programmer’s machine. See http://support.microsoft.com/kb/328912/ for details.

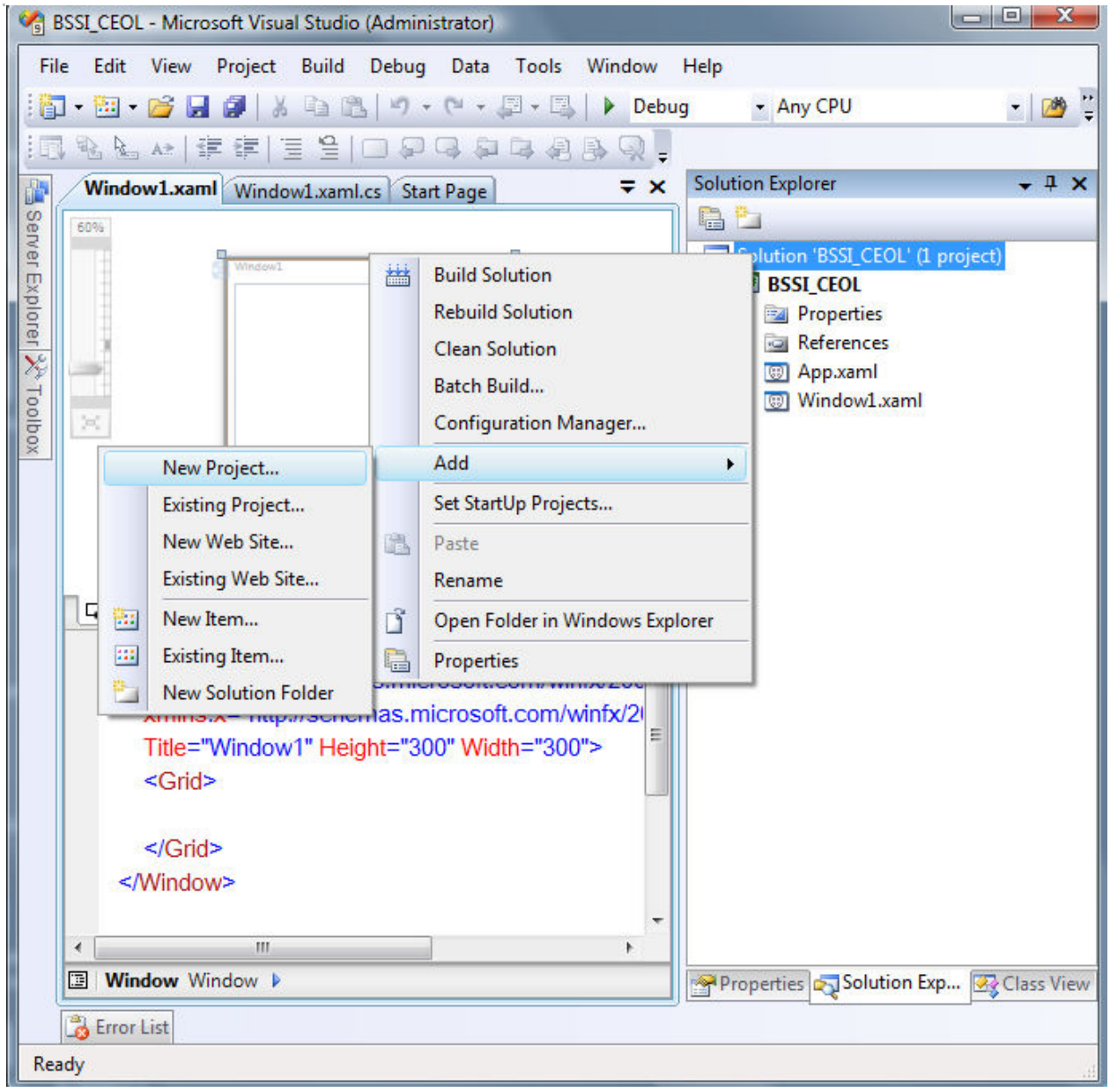


In the same Cielo Explorer session, the “Interop” tools are used access the qx table name and values:

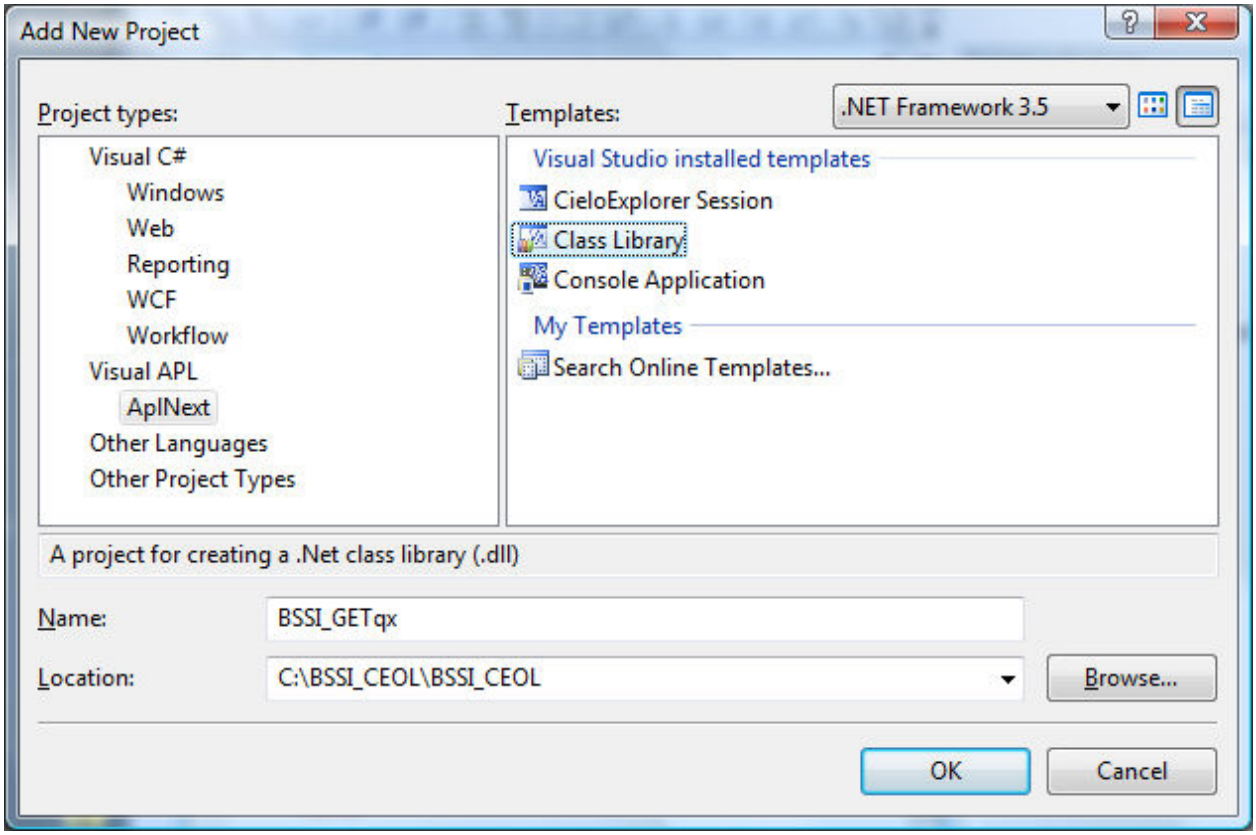


Now that the Cielo Explorer session has enabled the programmer to determine the method to access Microsoft Excel, a VisualAPL .Net assembly is created. Since access to the Excel workbook for the “static” data is going to be part of the application system, a VisualAPL class library project (a VisualAPL .Net assembly) is added to the Visual Studio solution. This .Net assembly will contain the application-specific APL functions to access Excel, obtain the actuarial qx table names and values and perform the necessary calculations for the solution.

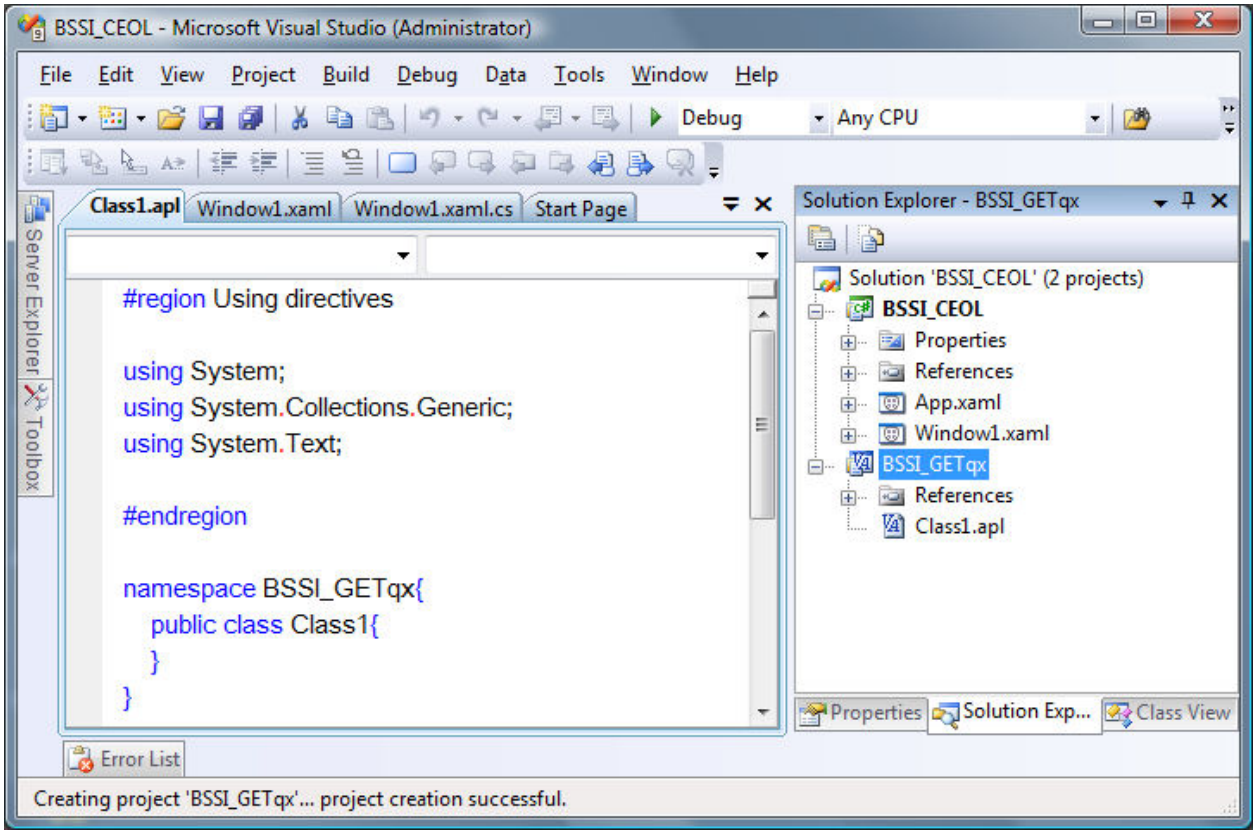
From the Visual Studio 2008 Solution Explorer, right click the “Solution ‘BSSI_CEOL’ (1 project)” node and select “Add > New Project”:


BSSI_CEOL - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Window Help
Debug
Any CPU
Window1.xaml
Window1.xaml.cs
Start Page
Solution Explorer
Solution 'BSSI_CEOL' (1 project)
BSSI_CEOL
Properties
References
App.xaml
Window1.xaml
Build Solution
Rebuild Solution
Clean Solution
Batch Build...
Configuration Manager...
Add
Set StartUp Projects...
Paste
Rename
Open Folder in Windows Explorer
Properties
New Project...
Existing Project...
New Web Site...
Existing Web Site...
New Item...
Existing Item...
New Solution Folder
Title="Window1" Height="300" Width="300">
<Grid>
</Grid>
</Window>
Window Window
Properties
Solution Exp...
Class View
Error List
Ready

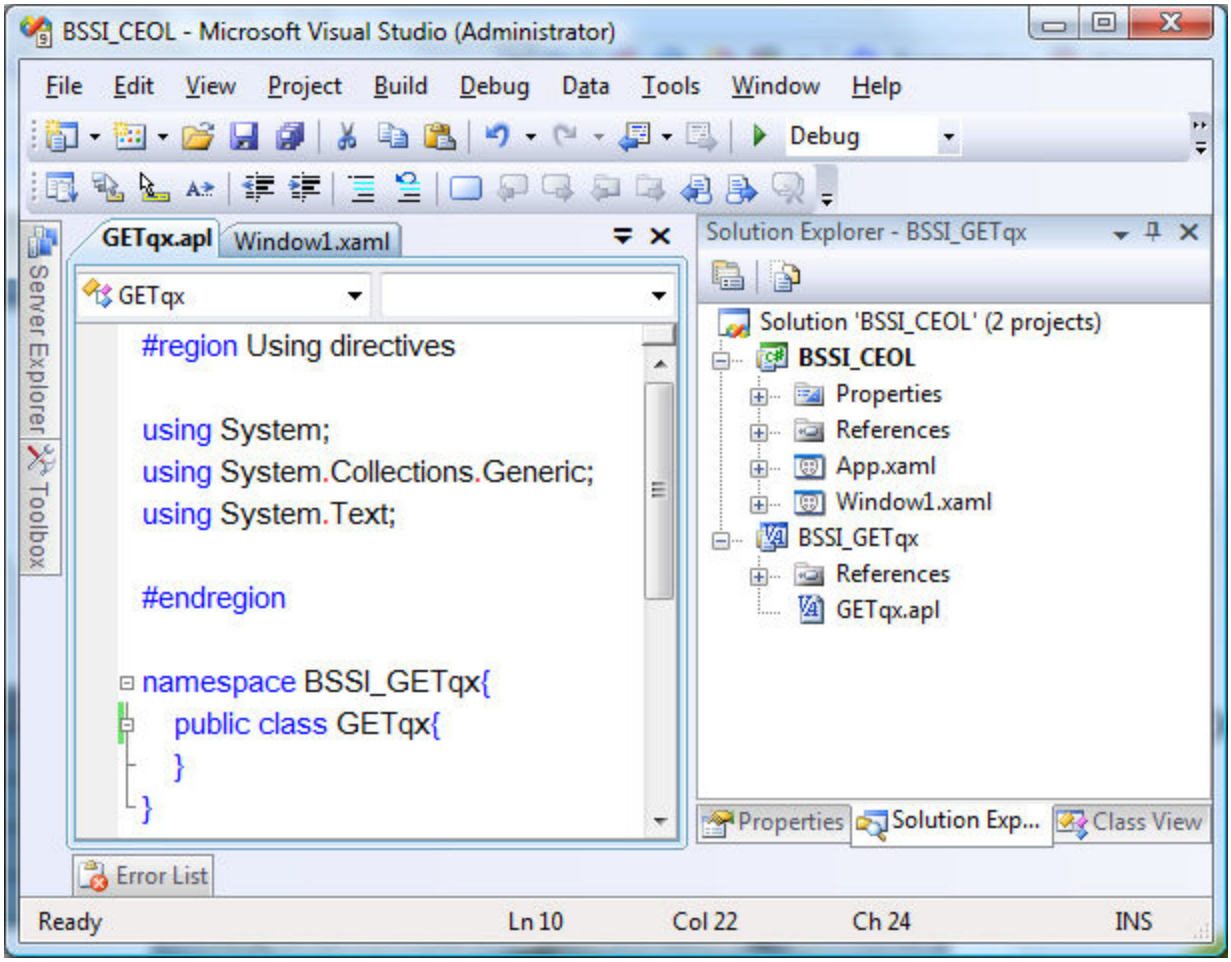
For the project type select “VisualAPL > Class Library” and enter the project name appropriately:



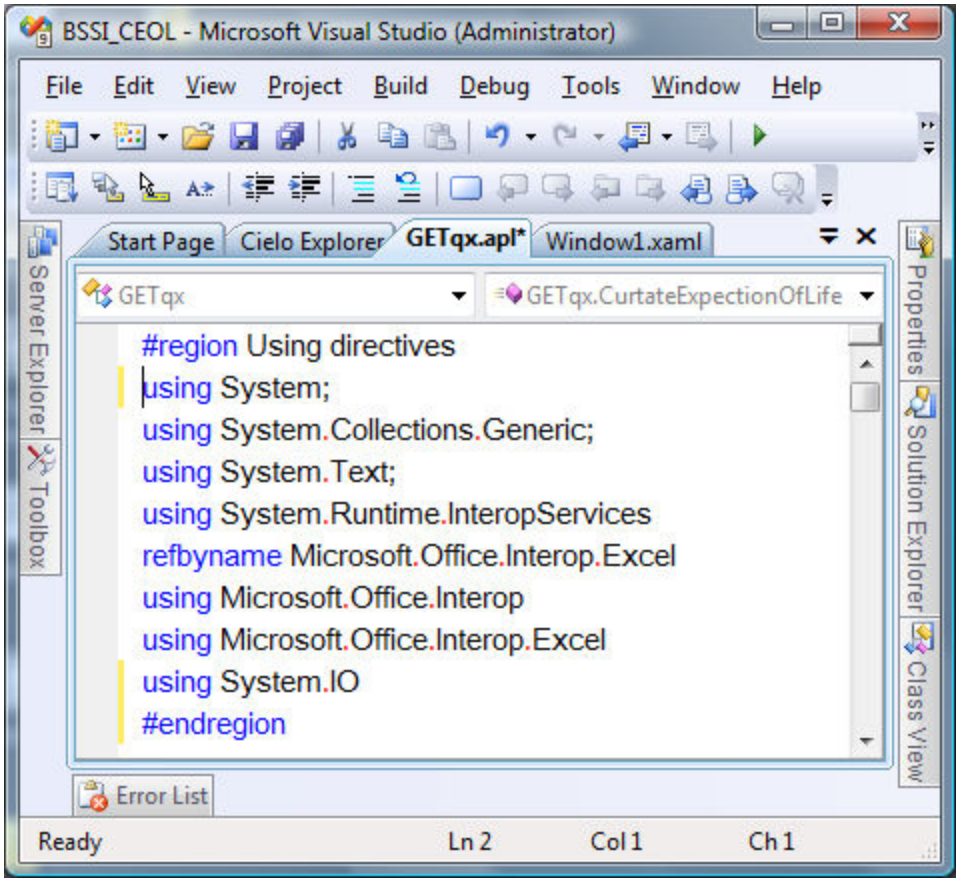
The VisualAPL class library will be added to the solution:



Edit the class name from “Class1” to “GETqx”. Instructions for doing this are available at: http://forum.apl2000.com/viewtopic.php?t=453



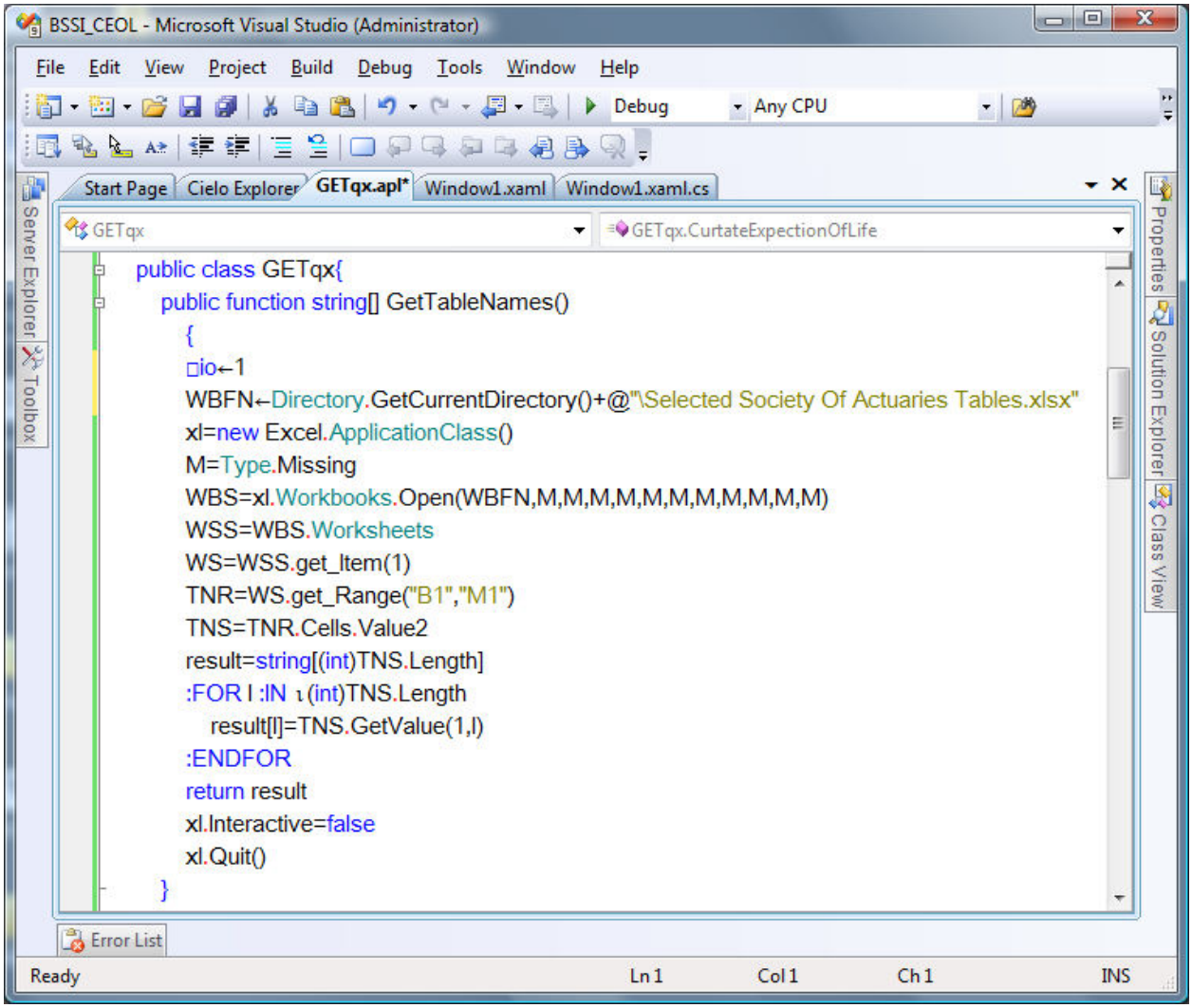
From the Cielo Explorer session copy the using directives to the VisualAPL GETqx class library project:



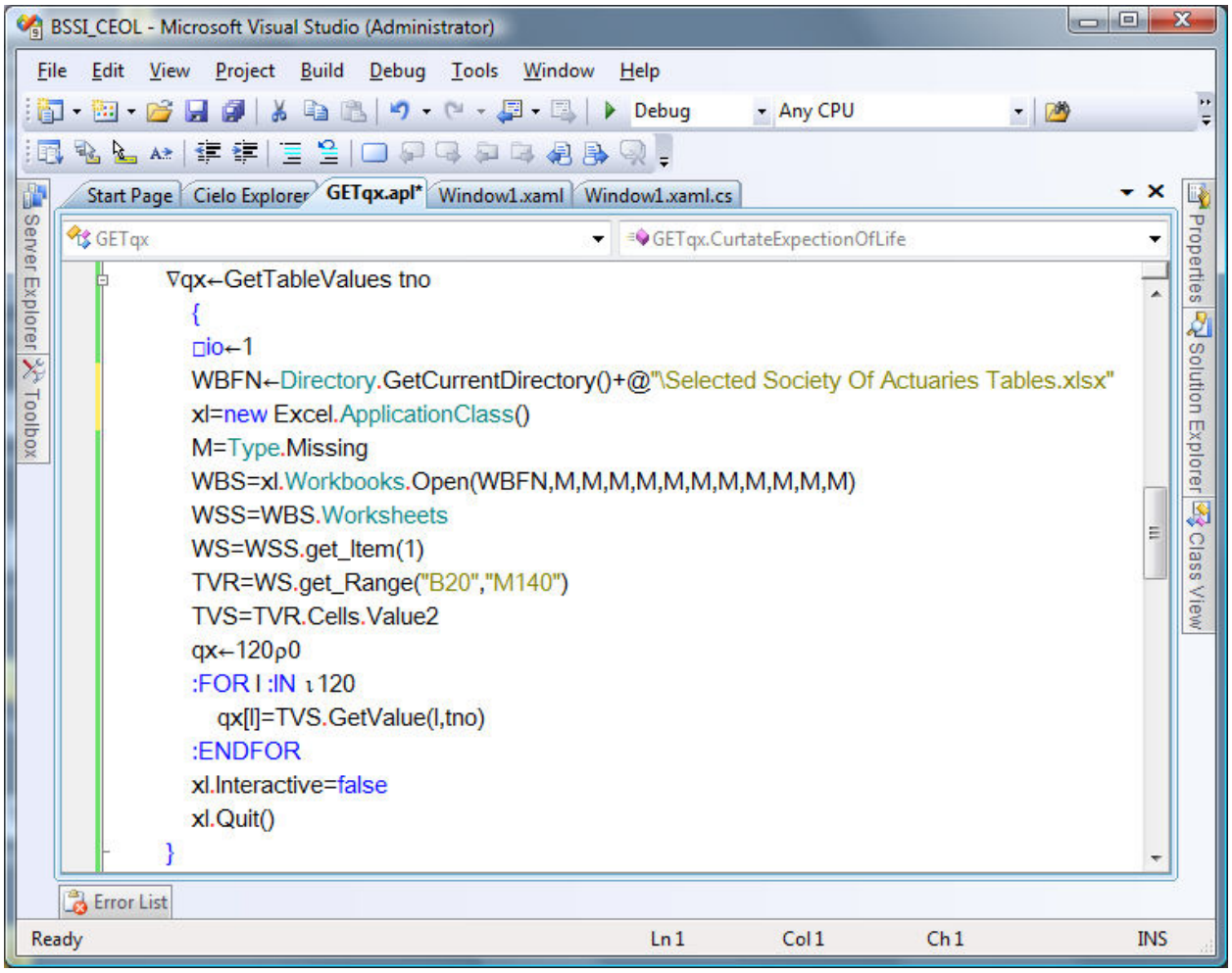
s

Add the GetTableNames() VisualAPL function to theGETqx class. Most of the statements in this function can be copied from the Cielo Explorer session. Because this function will be called by the C# WPF GUI class, a Visual Studio function signature, the public keyword and a strong data type result are used.

Notice that the “WBFN” workbook file name variable now assumes that the Excel workbook is in the current application subdirectory. When the solution is being debugged, the current subdirectory is “…BSSI_CEOL\bin\Debug” of the C# WPF GUI project, because that is the “StartUp Project” in the Visual Studio 2008 Solution Explorer for the solution.

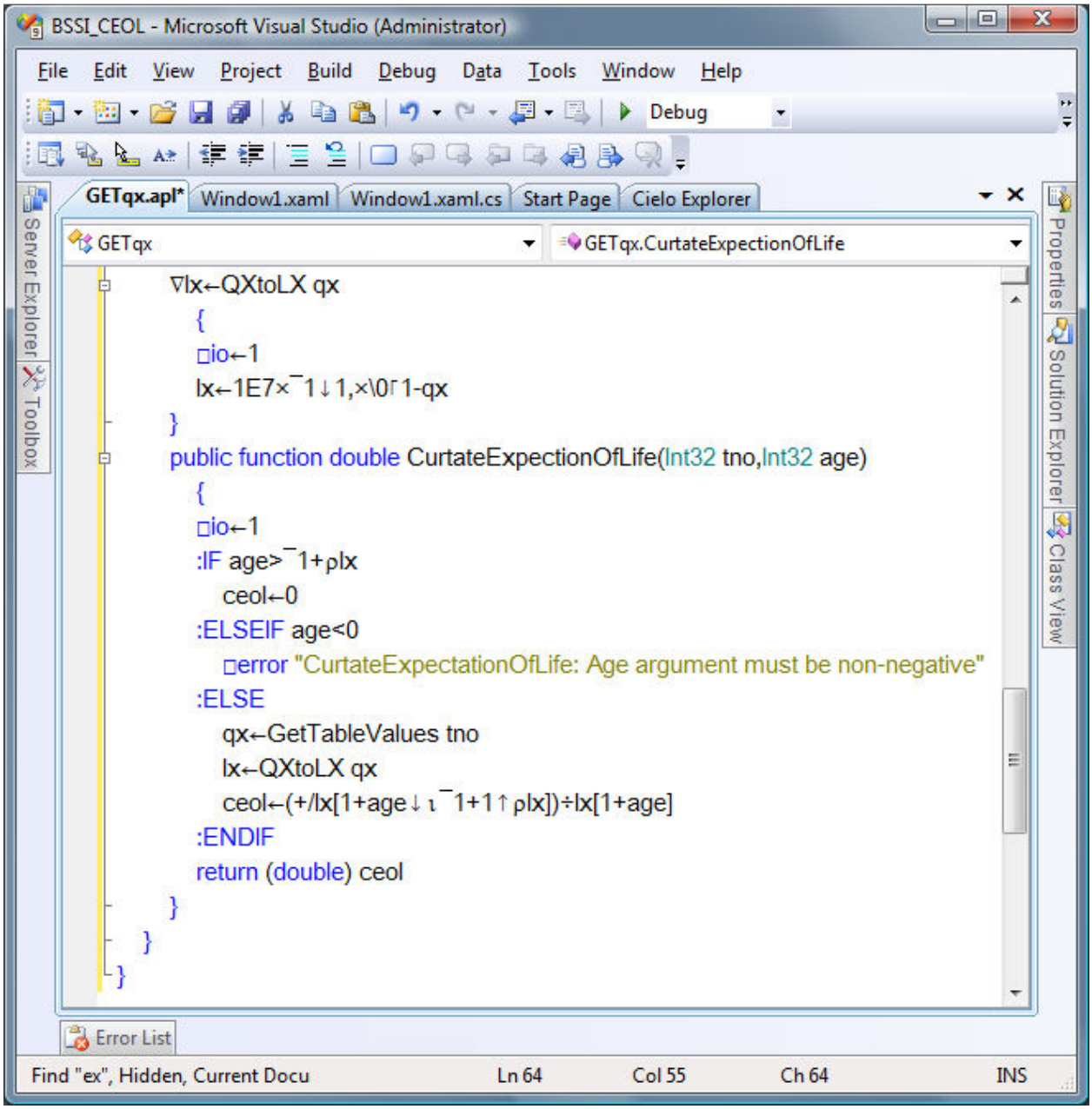


Add the GetTableValues VisualAPL function to theGETqx class. Most of the statements in this function can be copied from the Cielo Explorer session. Because this function will be called by other VisualAPL functions in this class it does not have to be public, it can use the traditional APL function signature and it does not require strong data types.

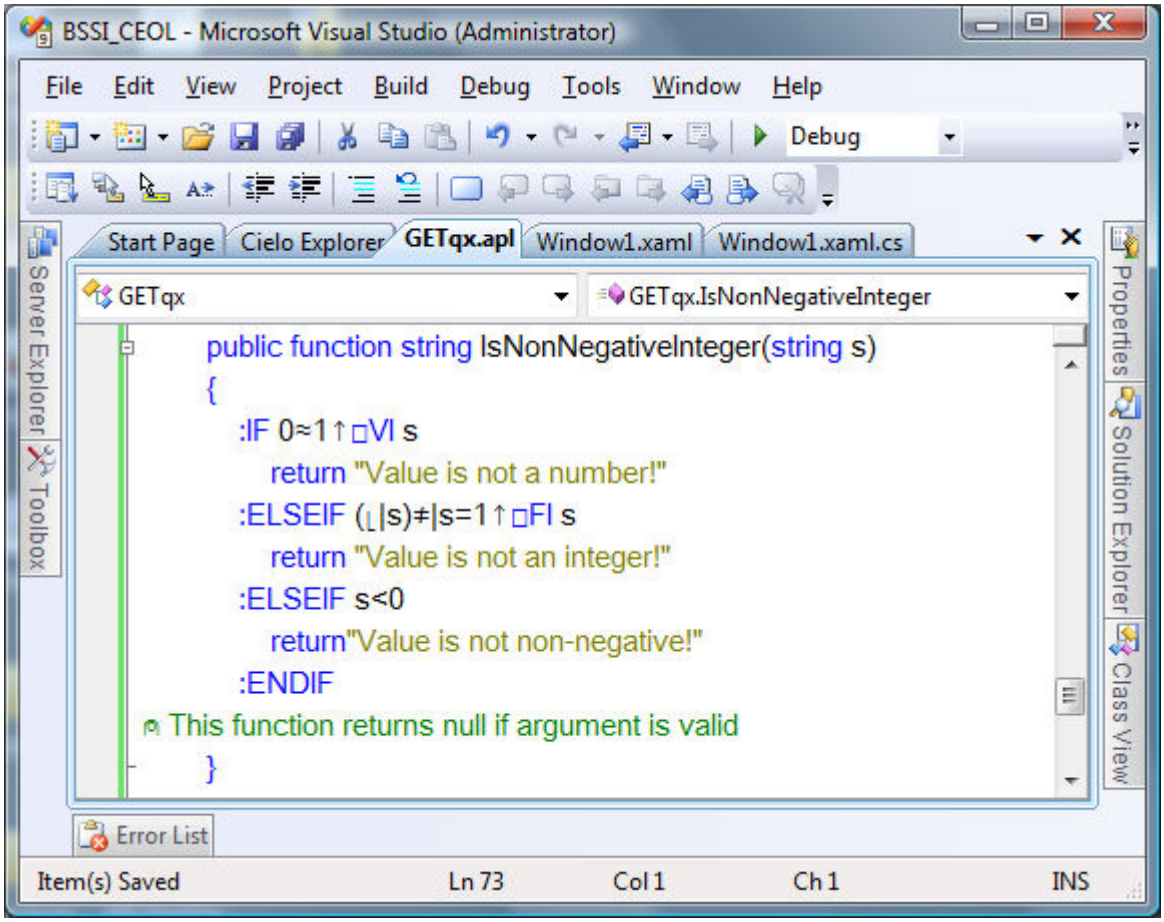


Closing Microsoft Excel (.Quit() method) from .Net is non-trivial. Refer to http://support.microsoft.com/kb/317109 for details. For simplicity of illustration, the screen captures in this document do not illustrate these techniques; however the associated Visual Studio 2008 solution files provided with this document incorporate the Microsoft-suggested methodology.

Add the QXtoLX and CurtateExpectationOfLife VisualAPL functions to theGETqx class. Because the QXtoLX function will be called by other VisualAPL functions in this class it does not have to be public, it can use the traditional APL function signature and it does not require strong data types. Because the CurtateExpectationofLife function will be called by the C# WPF GUI class, a Visual Studio function signature, the public keyword and strong data type arguments and result are used.



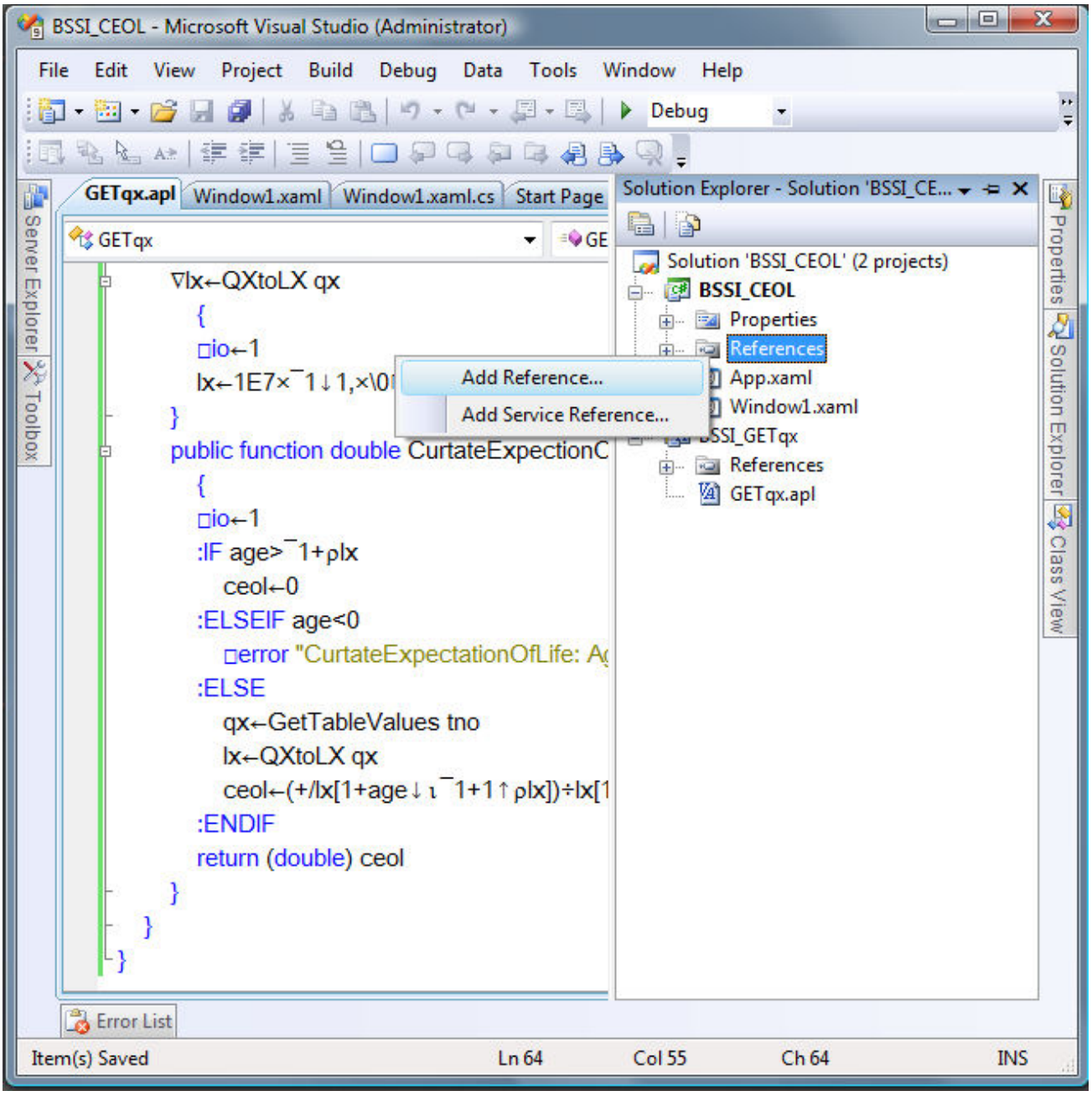
Add the IsNonNegativeInteger VisualAPL function which will be convenient to validate the user entry in the tbAge TextBox in the C# WPF GUI.



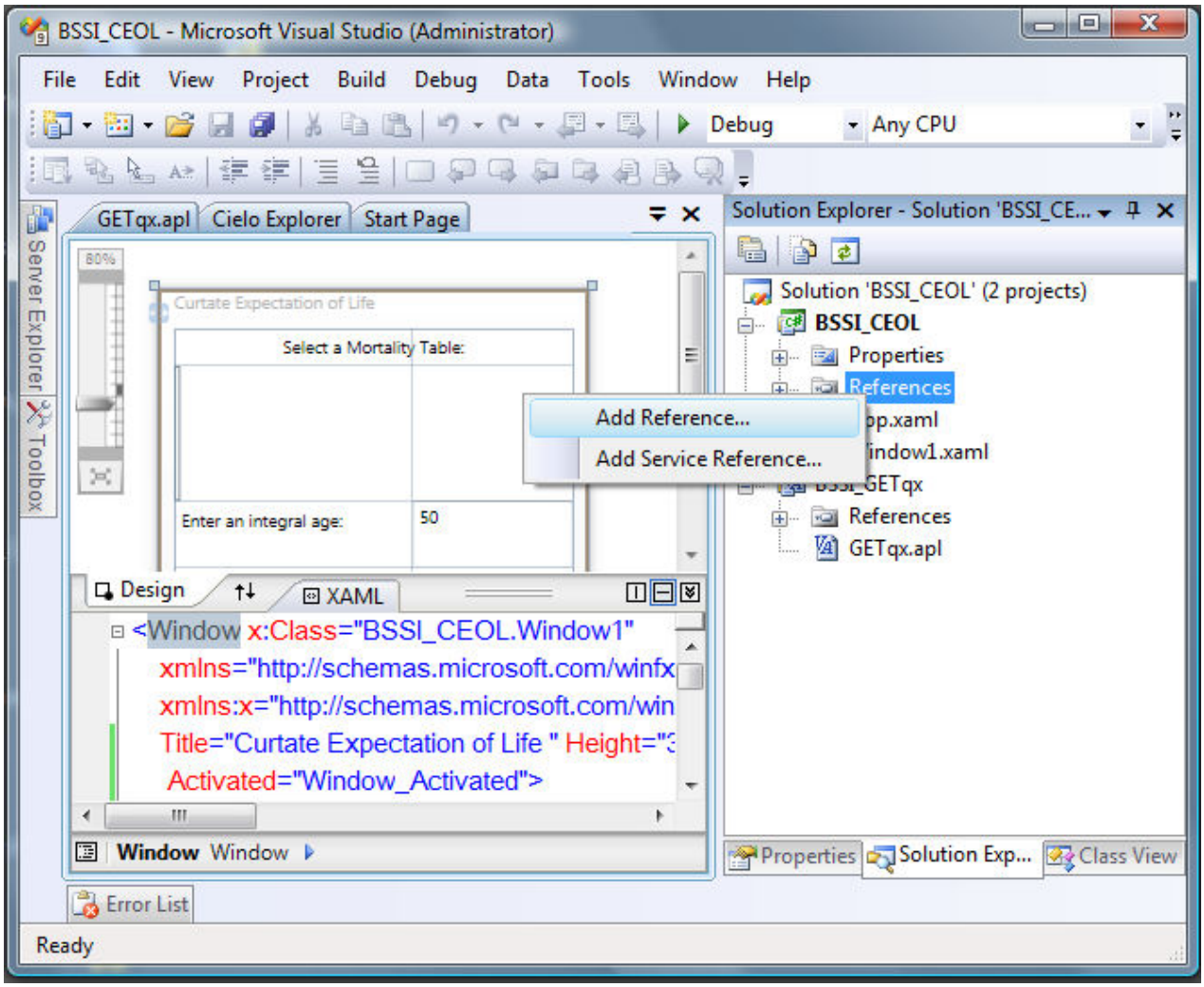
Remember to use the Visual Studio 2008 “File > Save All” to update the solution files on disk.

Connect the GUI to the Calculations/Business Rules

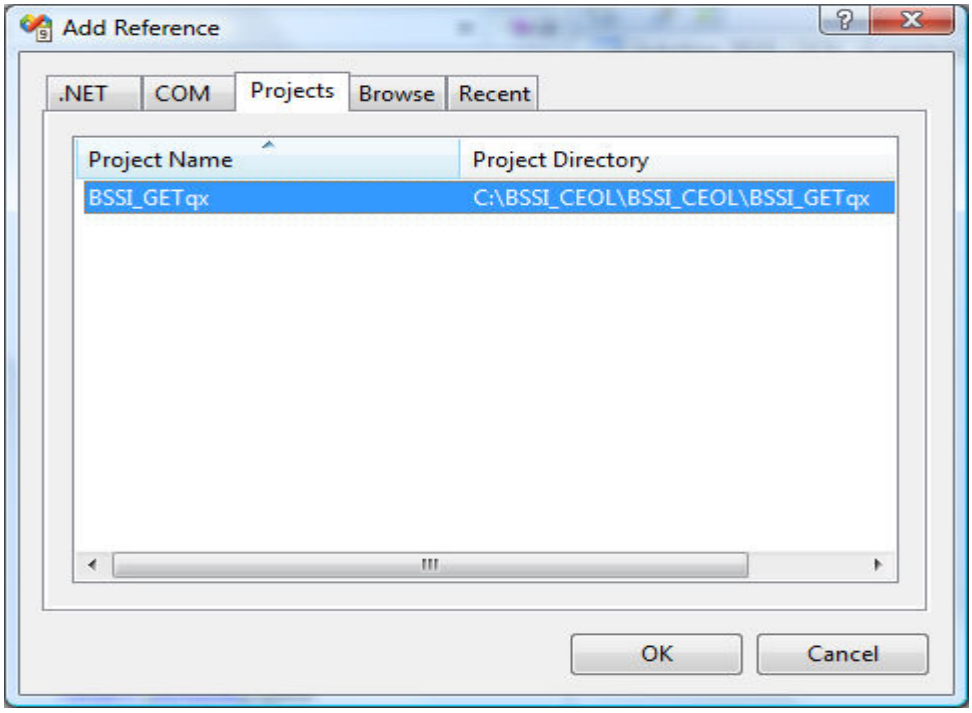
So that the C# GUI project can use the functions in the VisualAPL class, use the Visual Studio 2008 Solution Explorer to right click the References node of the C# WPF project and select “Add reference…”.



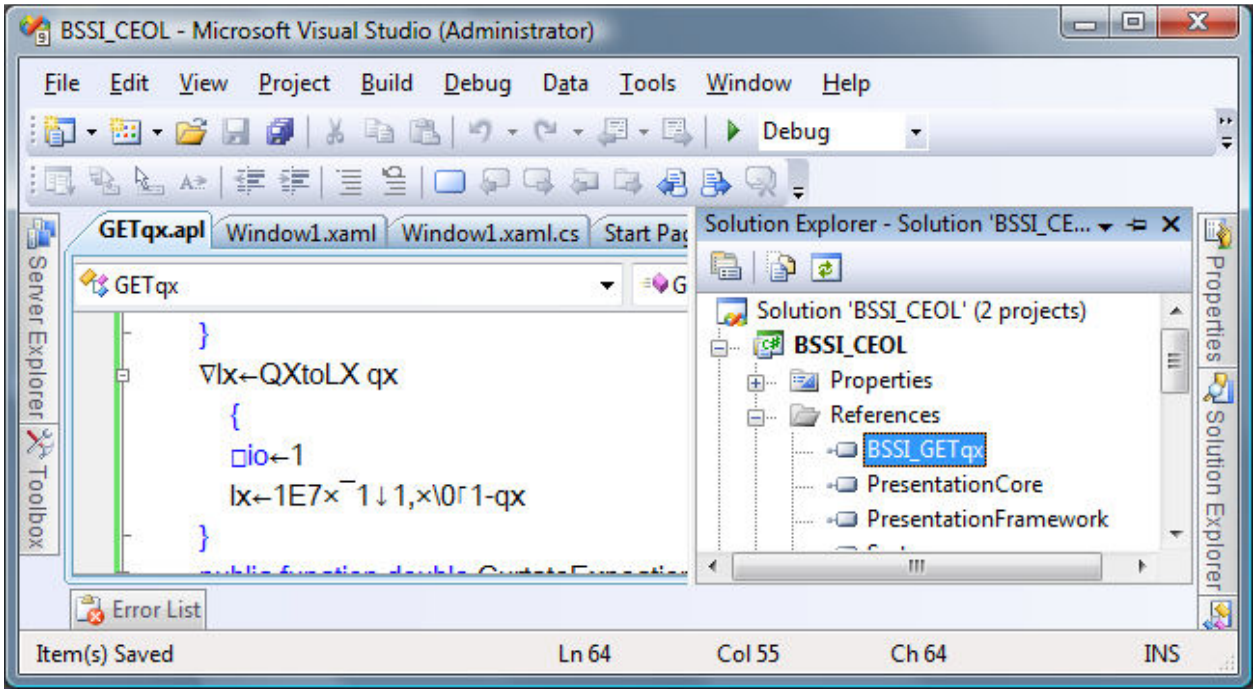
To connect the VisualAPL calculation functions to the C# WPF GUI, the C# WPF GUI “BSSI_CEOL” project must have a reference to the VisualAPL “BSSI_GETqx” project. In the Visual Studio 2008 Solution Explorer, right click the References node of the C# WPF GUI project and select “Add Reference…”:



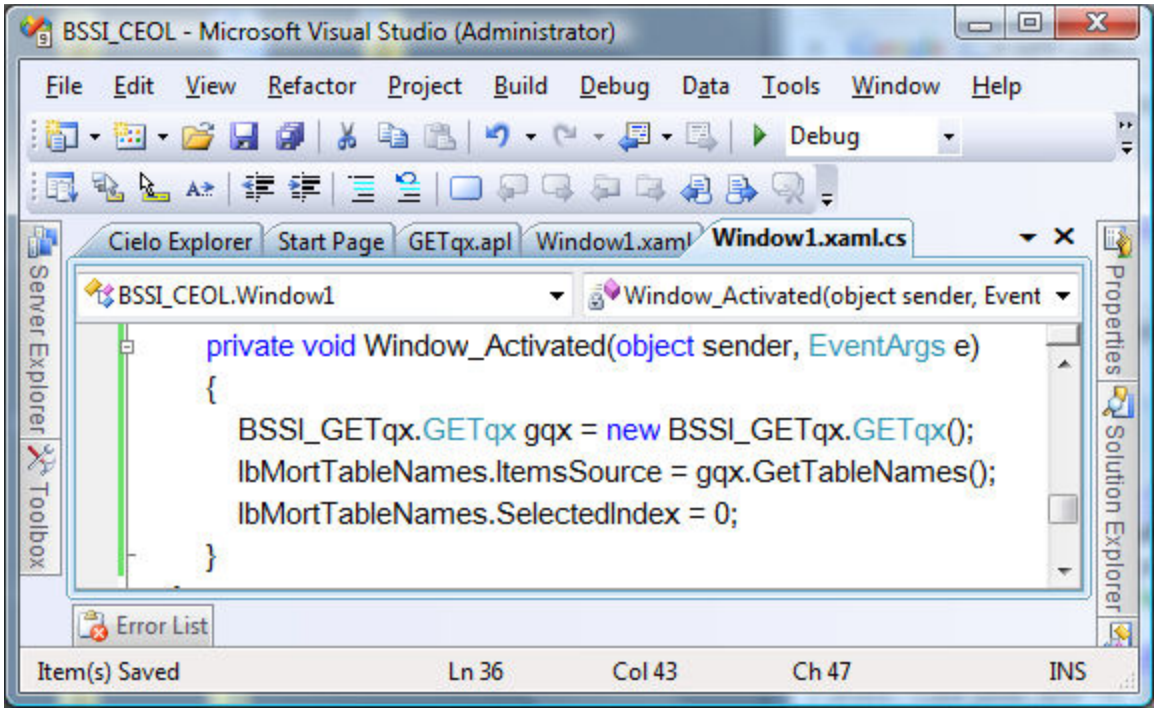
Visual Studio 2008 will display the “Add Reference” dialog. To complete the new reference, click the“Projects” tab, select the VisualAPL class library project “BSSI_GETqx” and click the “OK” button.



The reference to the VisualAPL class is now included in the C# WPF GUI project:

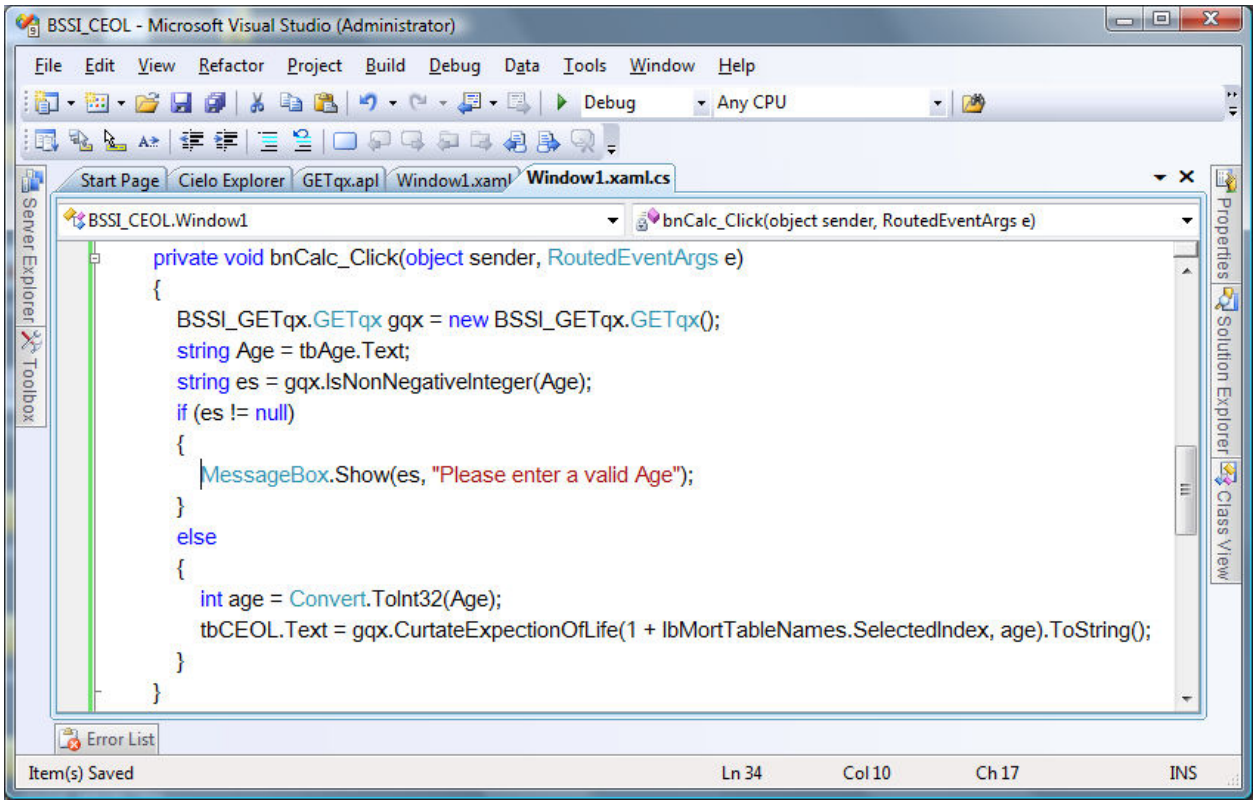


The code-behind file “Window1.xaml.cs” file contains the C# event handler “Window_Activate” function where the mortality table names are to be loaded into the WPF ListBox control in the GUI. An instance “gqx” of the VisualAPL BSSI_GETqx class is created in the Window_Activate function. The VisualAPL “GetTableNames()” function in the BSSI_GETqx class “gqx.GETqx()” obtains the mortality table names data. There are many ways to fill the items in a WPF ListBox control, including “WPF binding”, looping to set each individual binding, or using the “ItemsSource” property of the”lbMortTableNames” illustrated below. Finally the first mortality table name (index origin 0 in C#) is selected as the default mortality table.



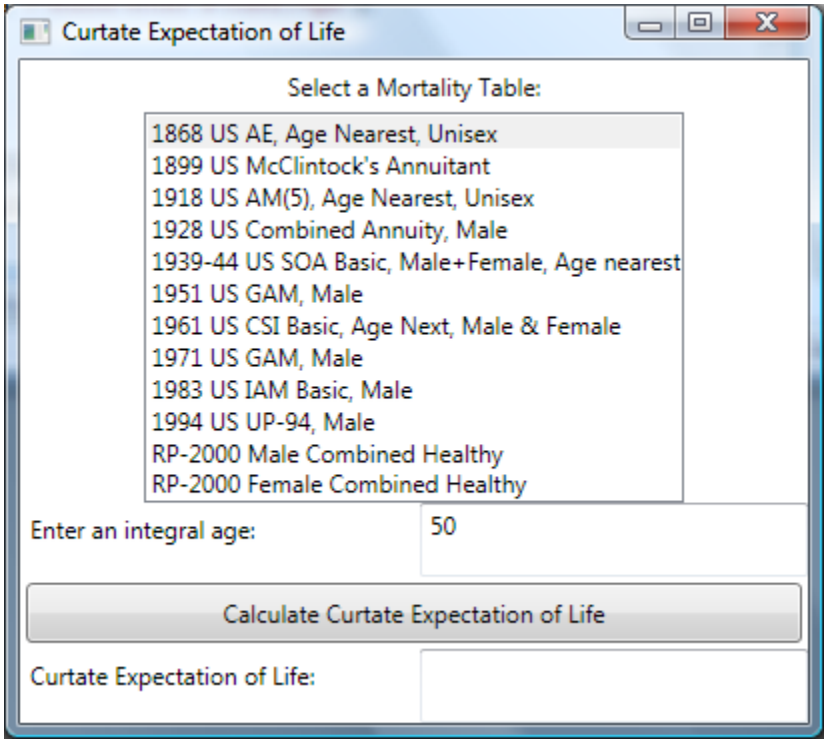
The code-behind file “Window1.xaml.cs” file contains the C# event handler “bnCalc_Click” function where:

- The user-entered age (as a string) in the tbAge.Text TextBox property is accessed
- The user-entered age (as a string) is validated using the VisualAPL IsNonNegativeInteger function
- The user-entered age is converted to an integer
- The user-selected mortality table index in the lbMortTableNames ListBox is accessed
- The user entered values are passed as arguments to the VisualAPL CurtateExpectationOfLife() and the result is displayed in the tbCEOL.Text TextBox property

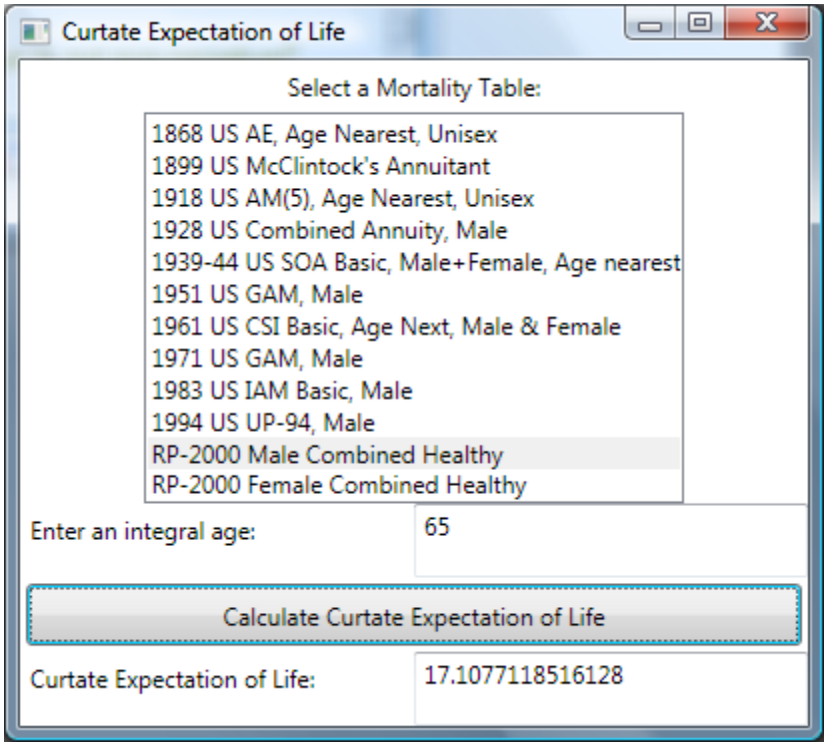


Test (Debug) the Completed Solution

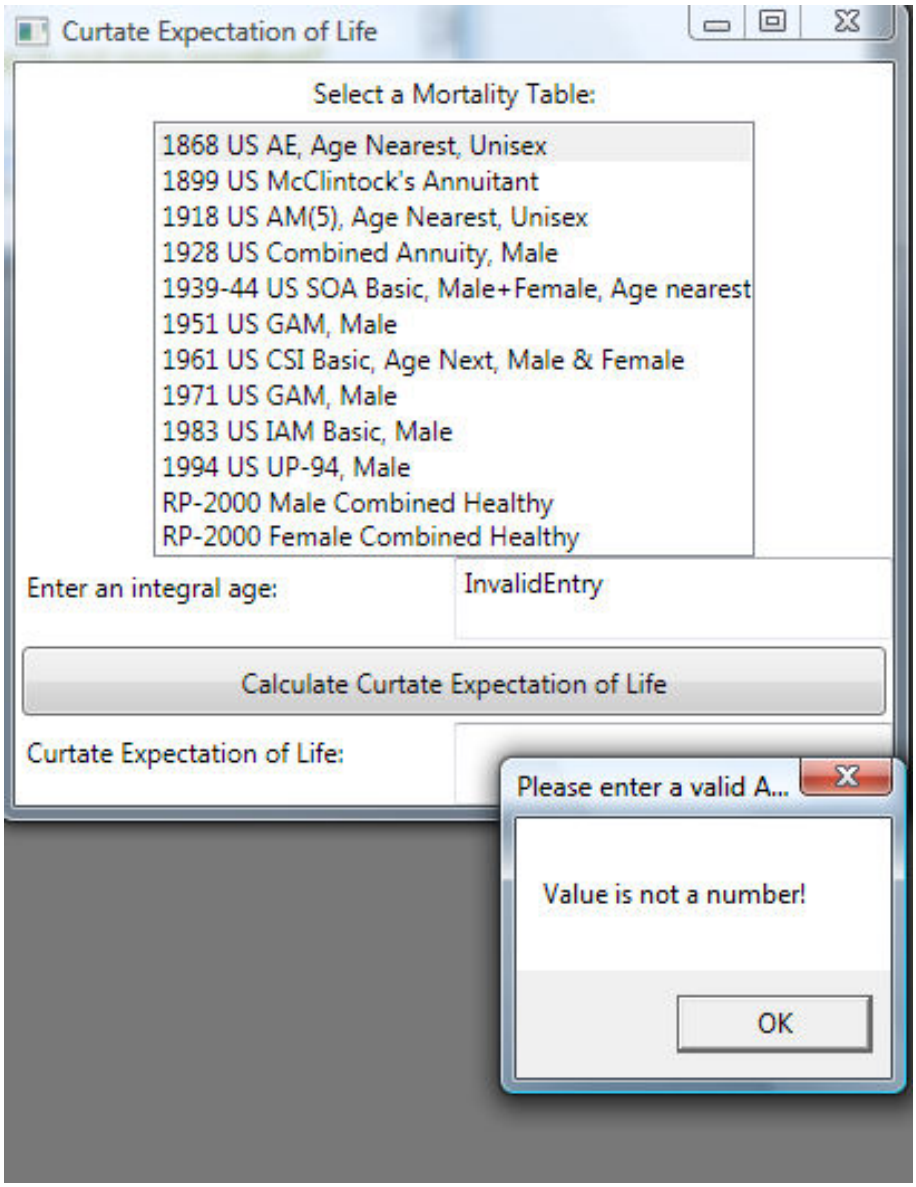
The Visual Studio 2008 “Debug > Start Debugging” (F5) is used to test the application system solution. Visual Studio 2008 will compile the C# WPF GUI and the VisualAPL class library and then display the GUI ready for user input. Compile errors and warnings, if any, will be displayed in the Errors/Warnings/Messages tabs in the Visual Studio 2008 programming window.



Clicking the “Calculate…” button causes the application system to perform the required calculations when valid entries are provided by the user:



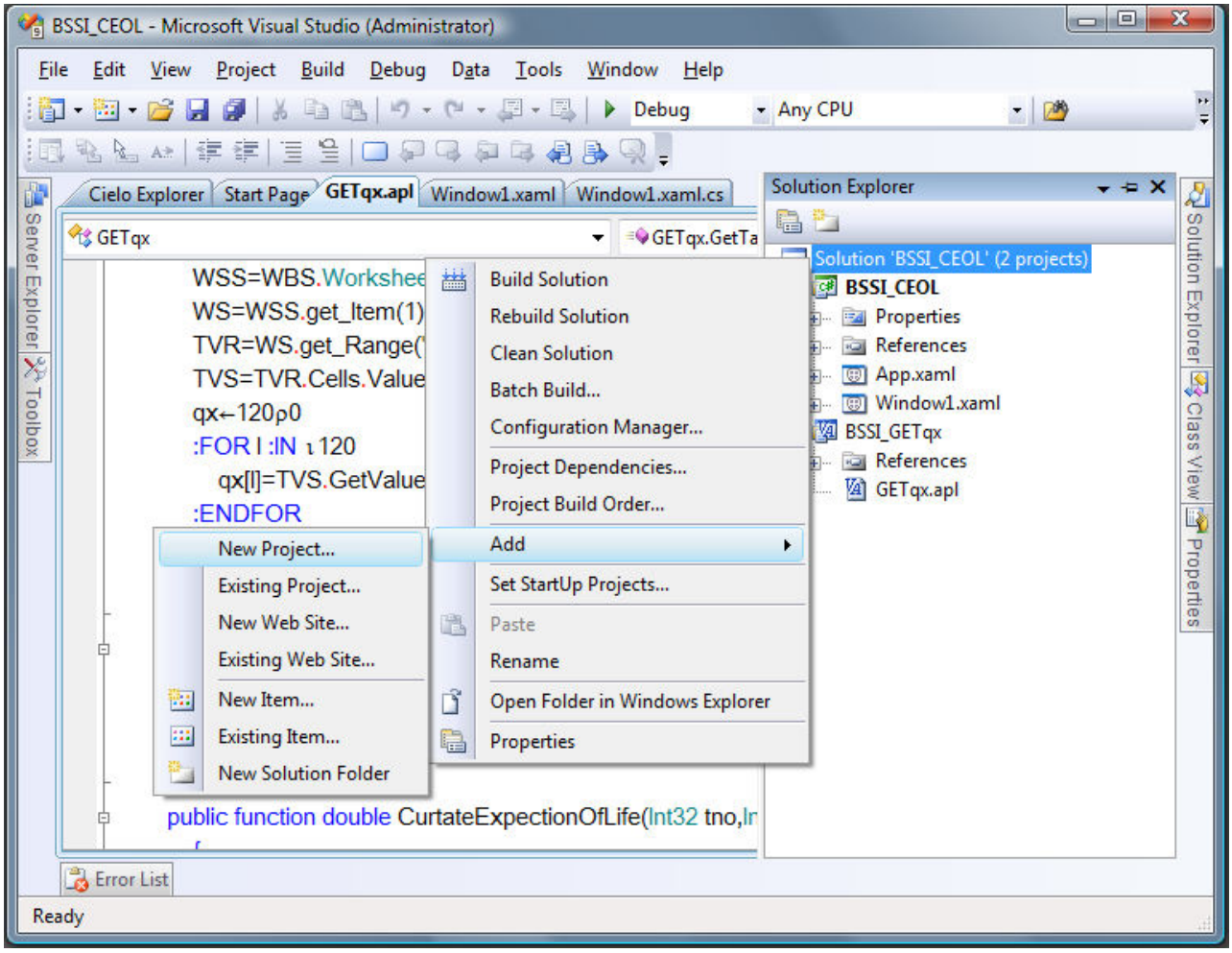
Clicking the “Calculate…” button causes the application system to display the MessageBox error message when invalid entries are provided by the user:



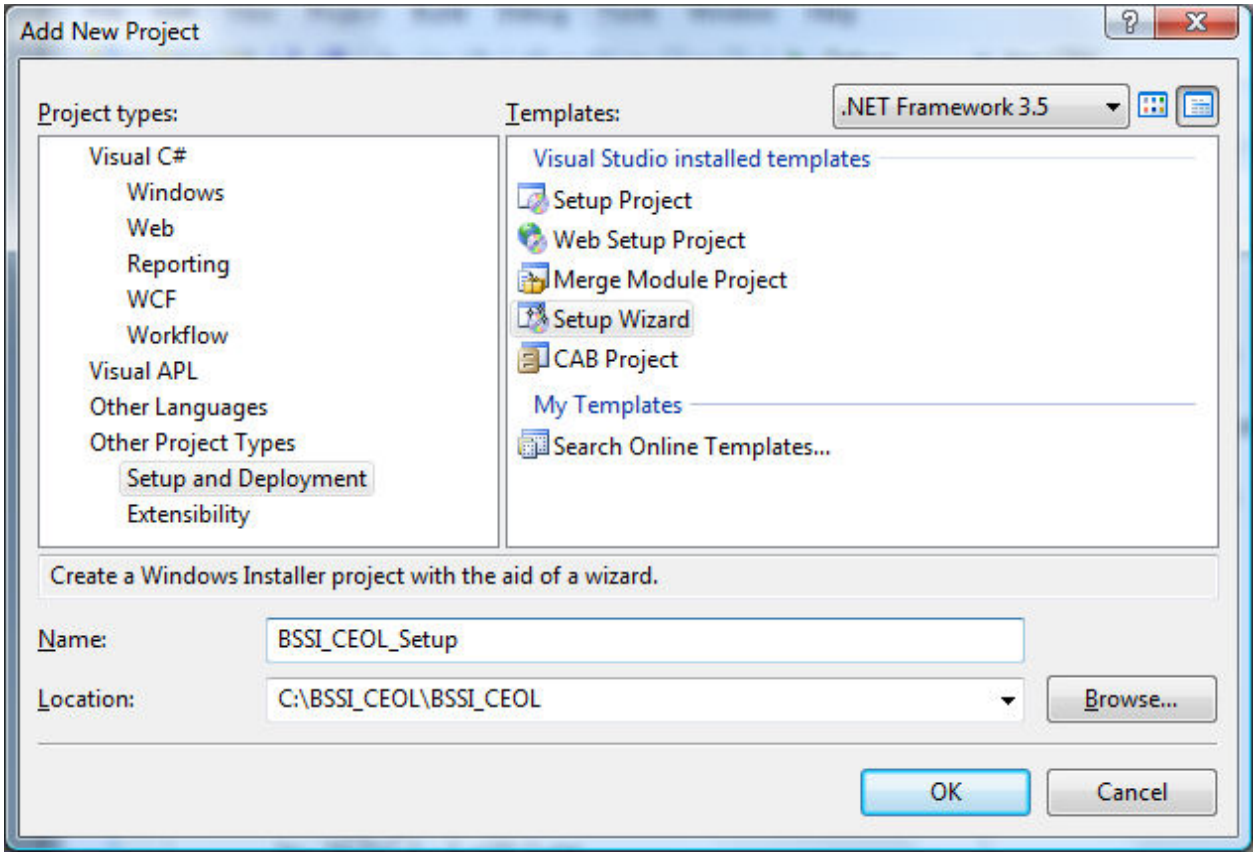
Click the [X] icon in the upper right corner of the WPF GUI or use the Visual Studio 2008 “Debug > Stop Debugging” (Shift+F5) to end the debugging session.

Create the Deployment (.msi) Project

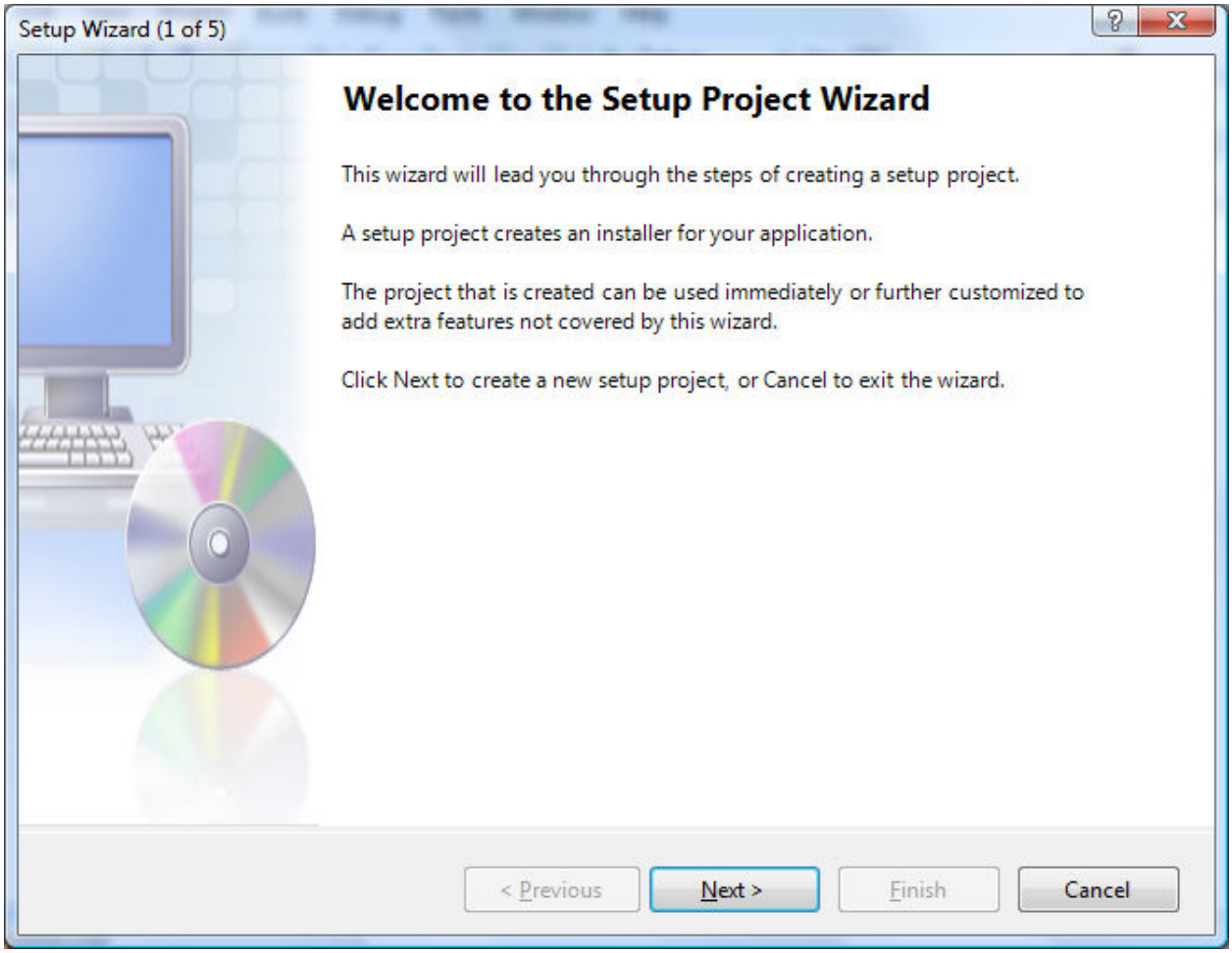
To deploy the completed application a Microsoft ‘installer’ project is added to the solution. From the Microsoft Visual Studio 2008 Solution Explorer, right click the “Solution ‘BSSI_CEOL’ (2 projects)” node and select “Add > New Project…”:



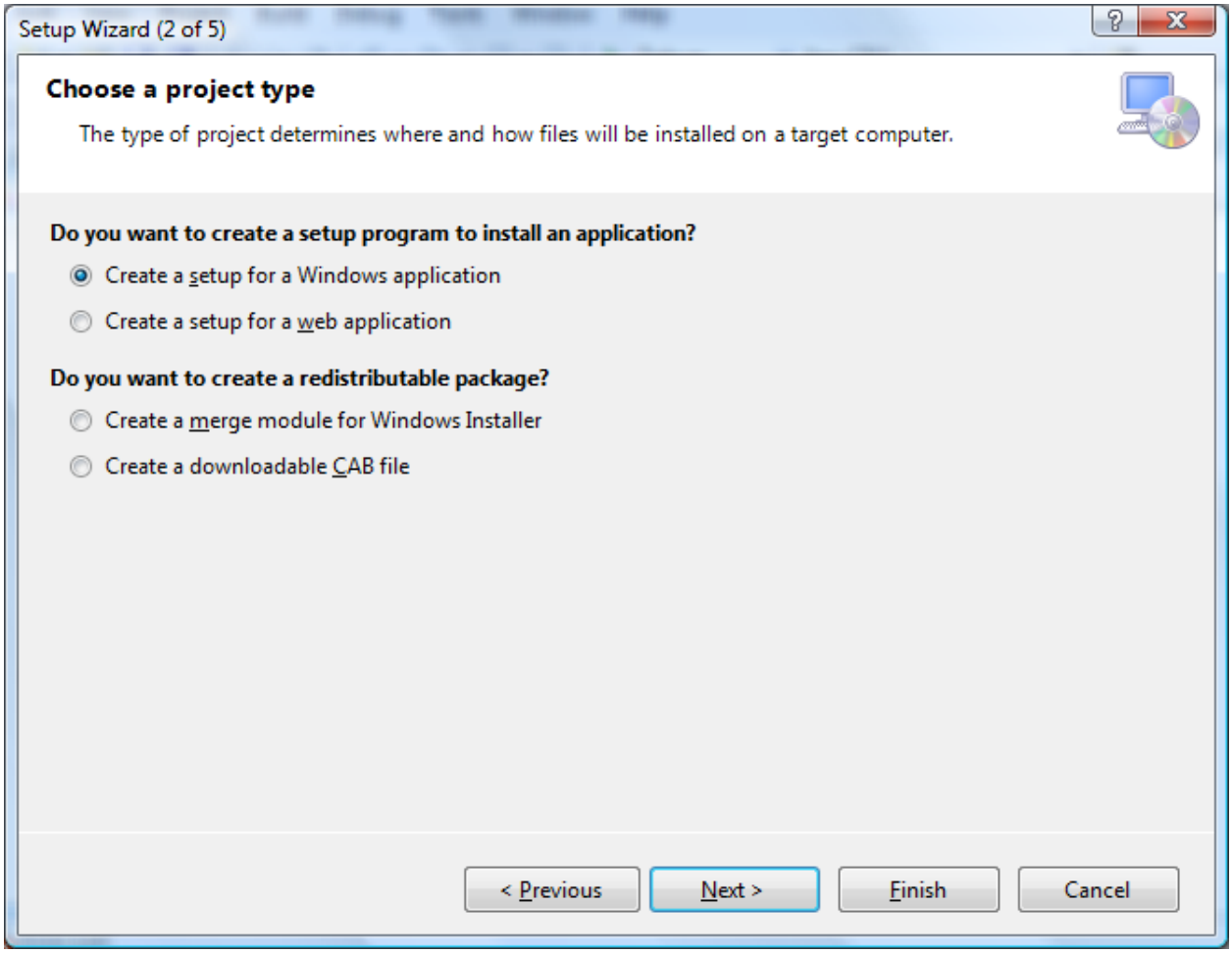
In the Microsoft Visual Studio 2008 “Add New Project” dialog, select “Other Project Types > Setup and Deployment > Setup Wizard” and enter an appropriate name for the .msi project:



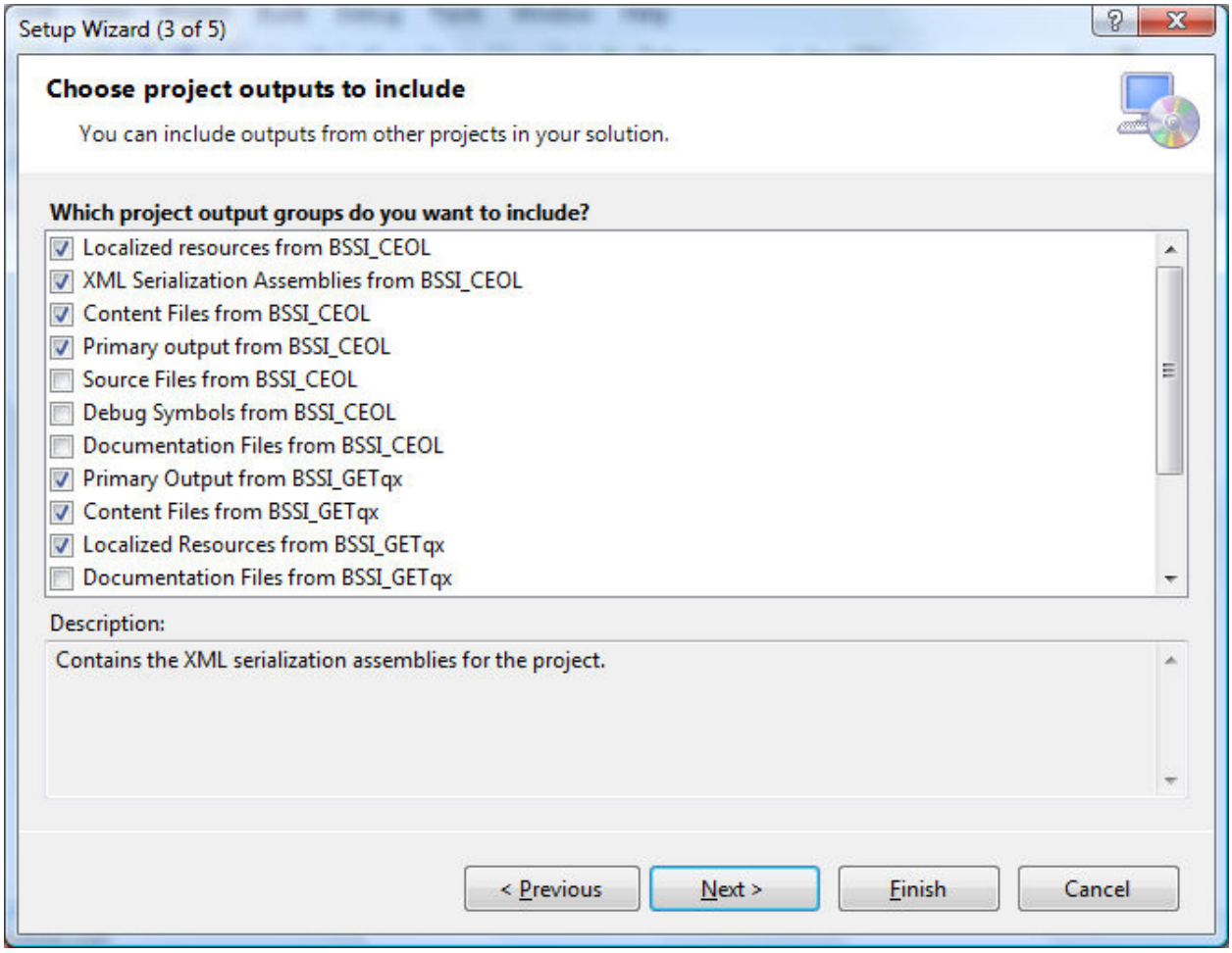
Click the “Next” button to continue creating the deployment project:



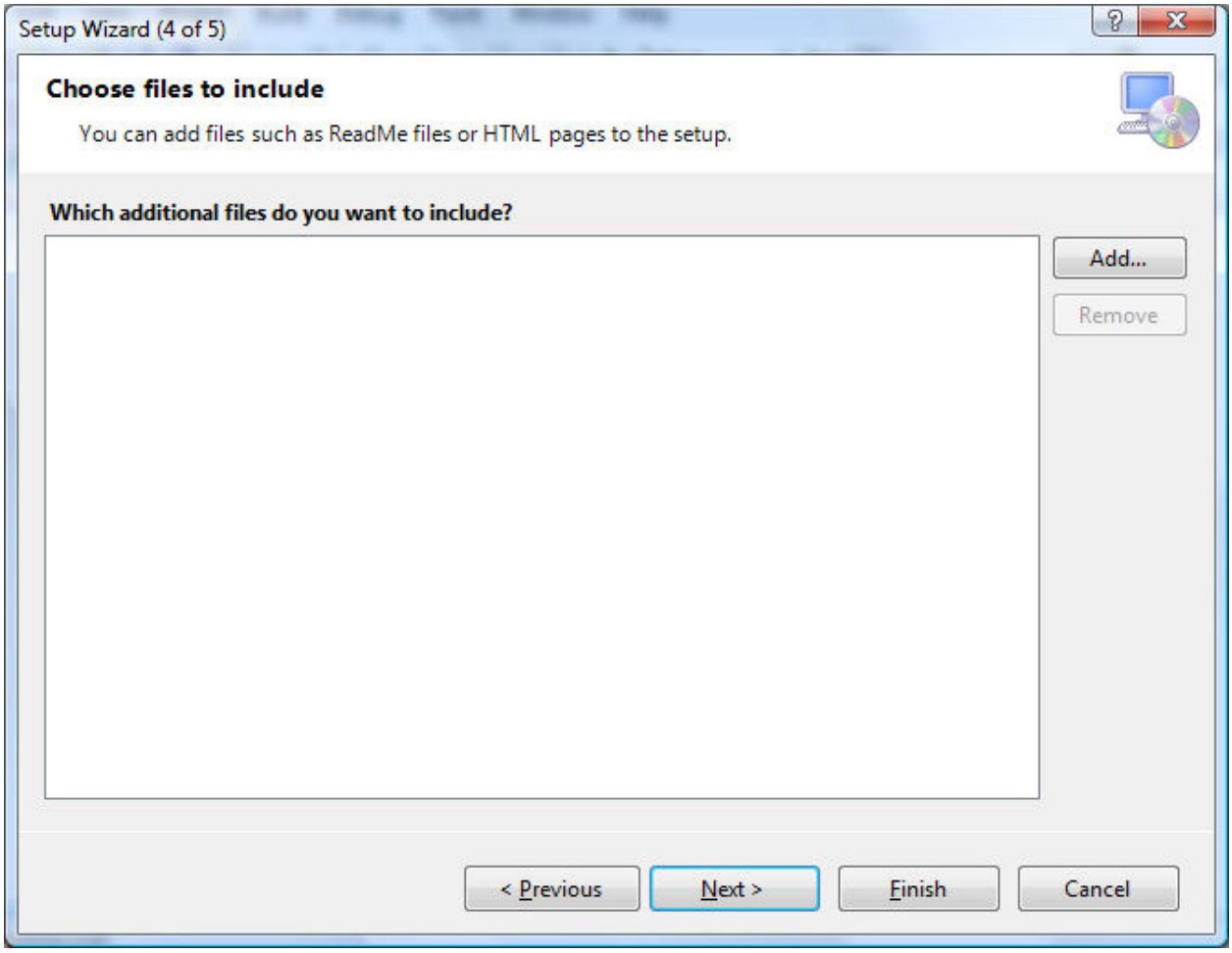
Select “create a setup for a Windows application” and click the “Next” button.



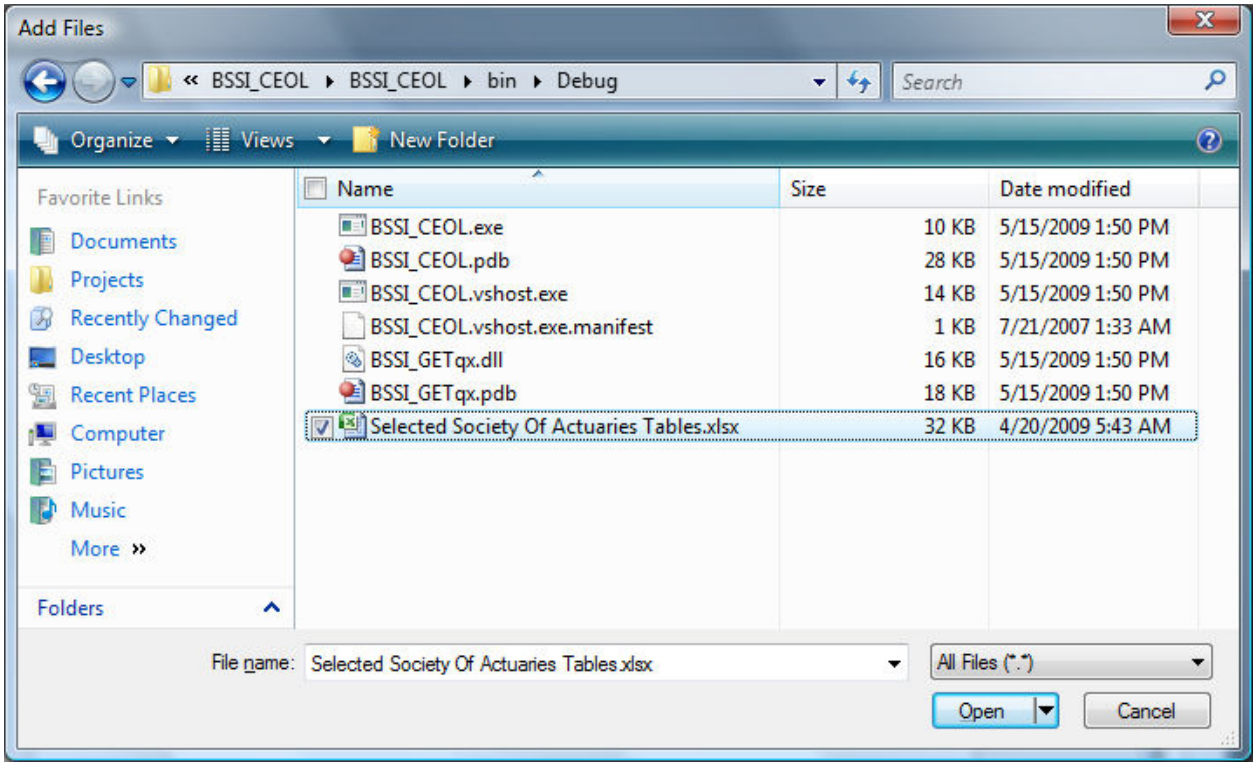
Select the files to include in the .msi installer. Generally the source files should not be included. Click the "Next" button to continue with the wizard:



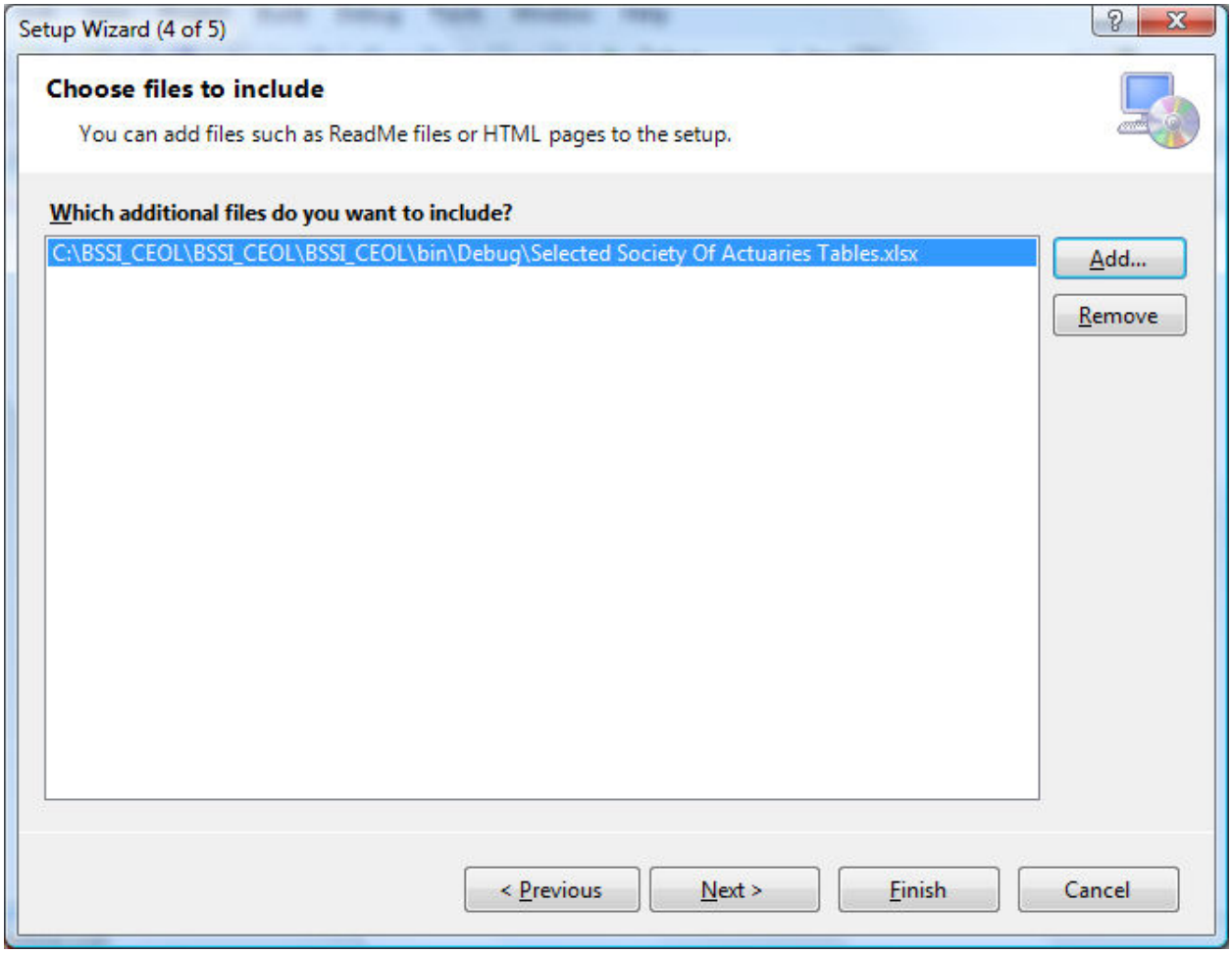
Because the “Selected Society Of Actuaries Tables.xlsx” Excel workbook is part of the solution, click the “Add…” button to include it in the .msi project:



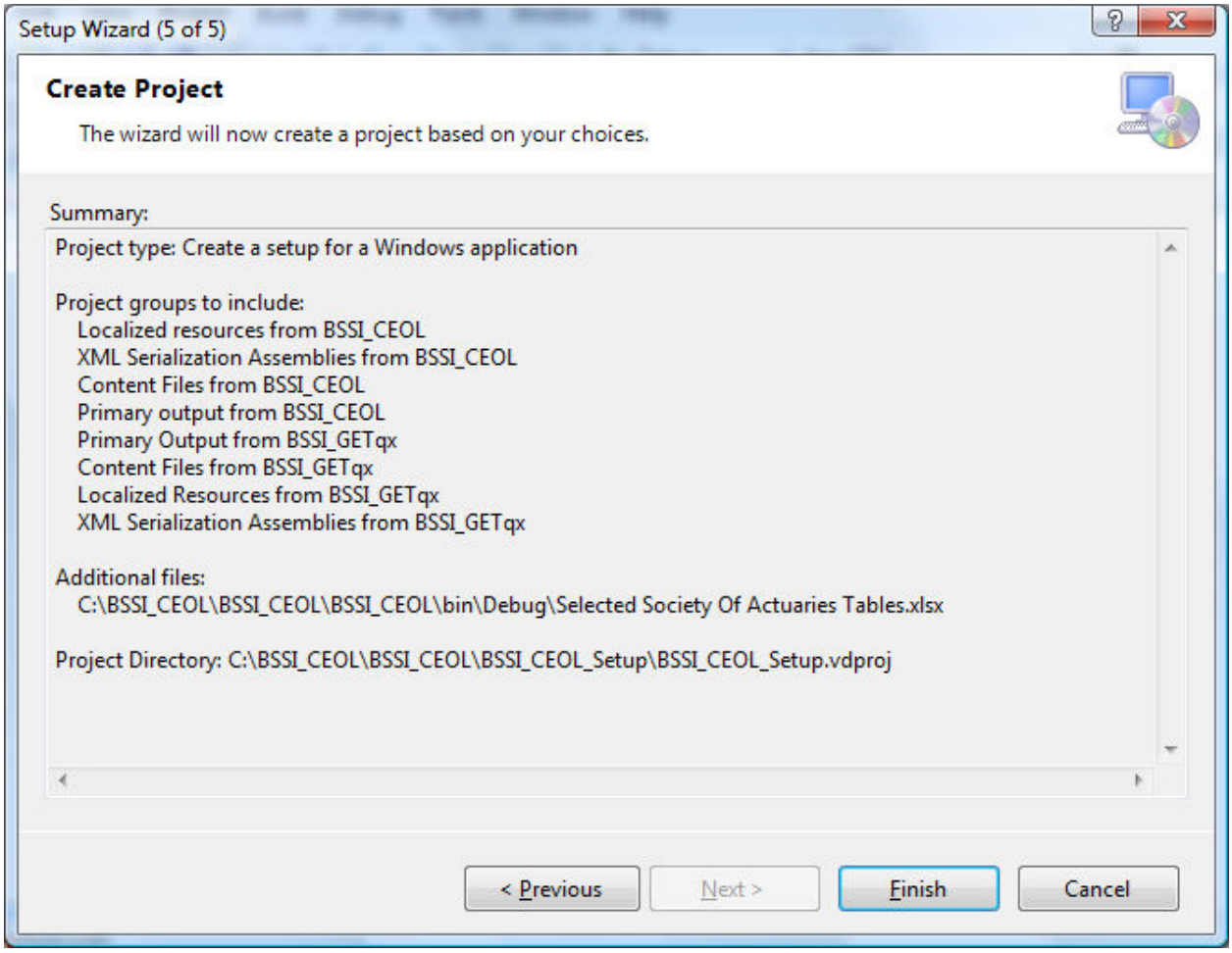
Browse to the Excel workbook location, select the file and click the “Open” button to include it in the .msi project:



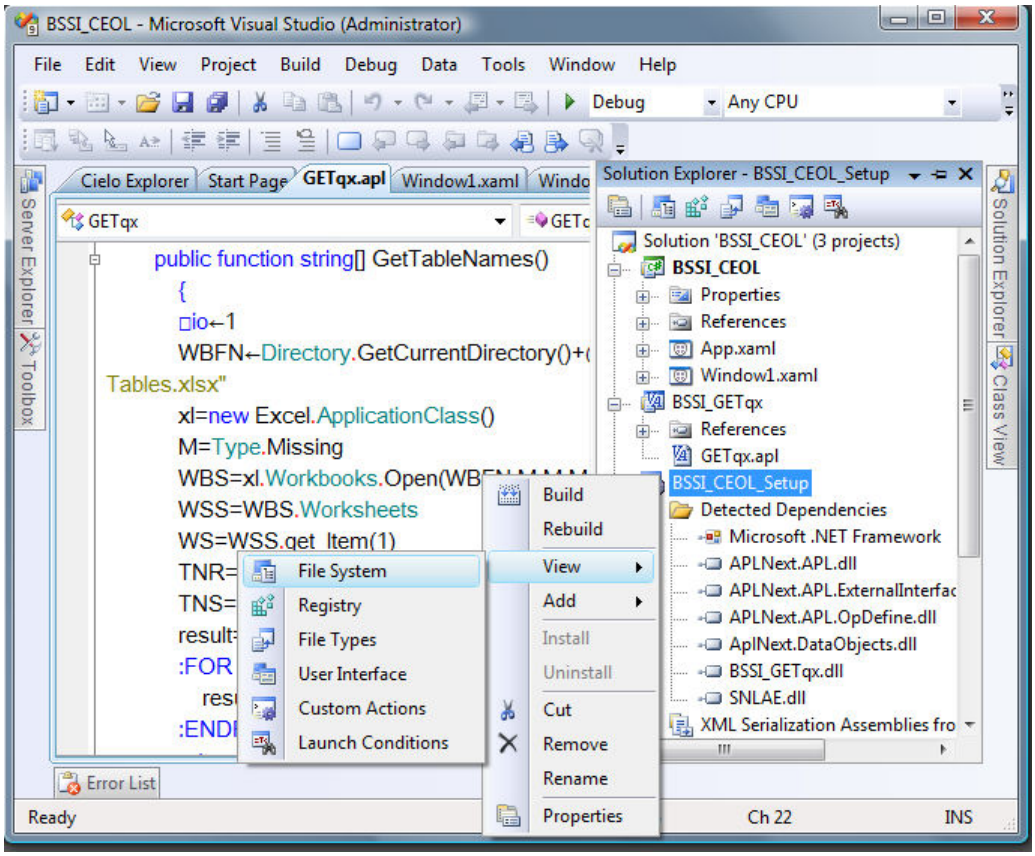
Click the “Next” button to continue with the .msi wizard:



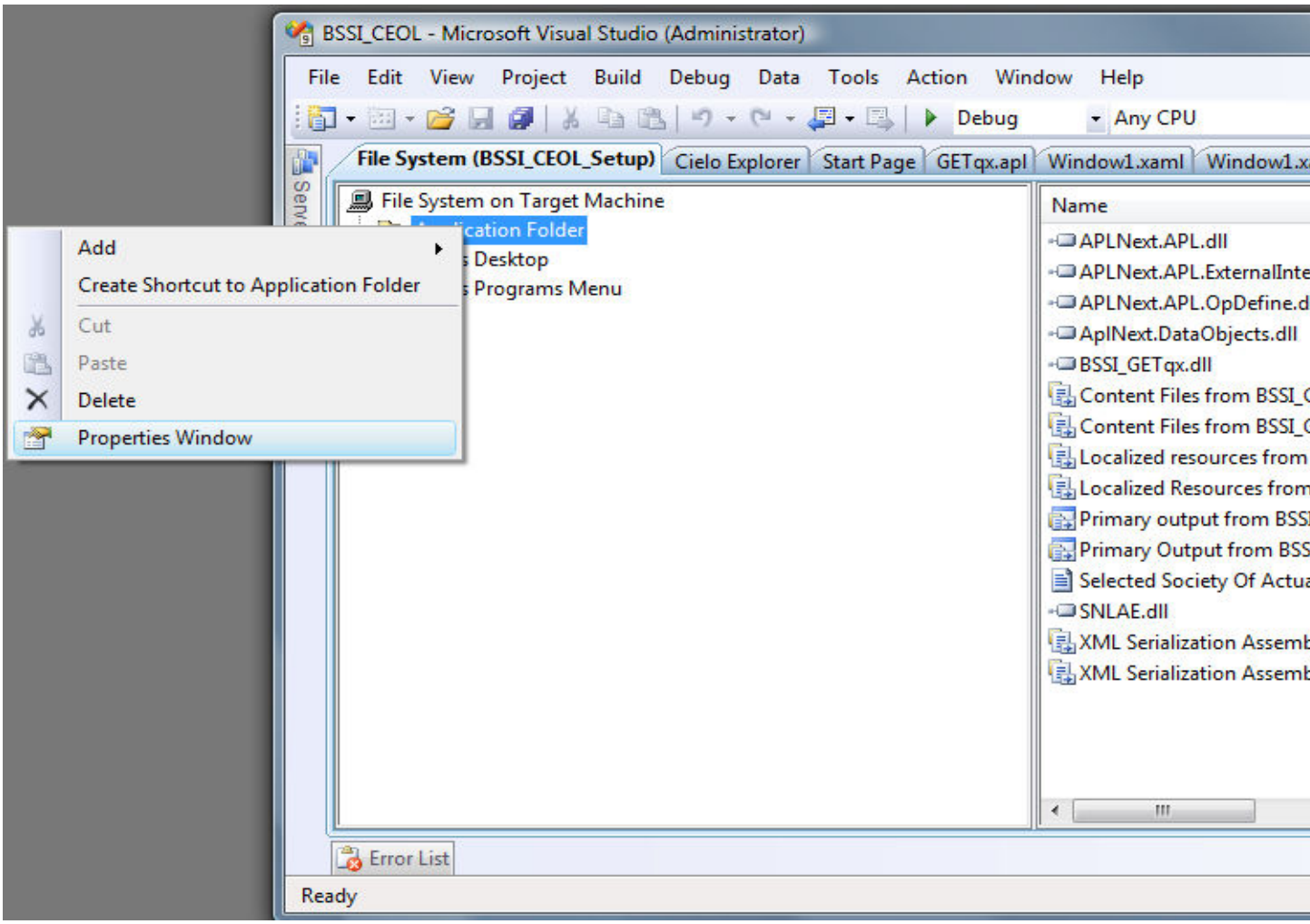
Review the .msi project summary and if it is acceptable, click the “Finish” button to complete the .msi wizard:



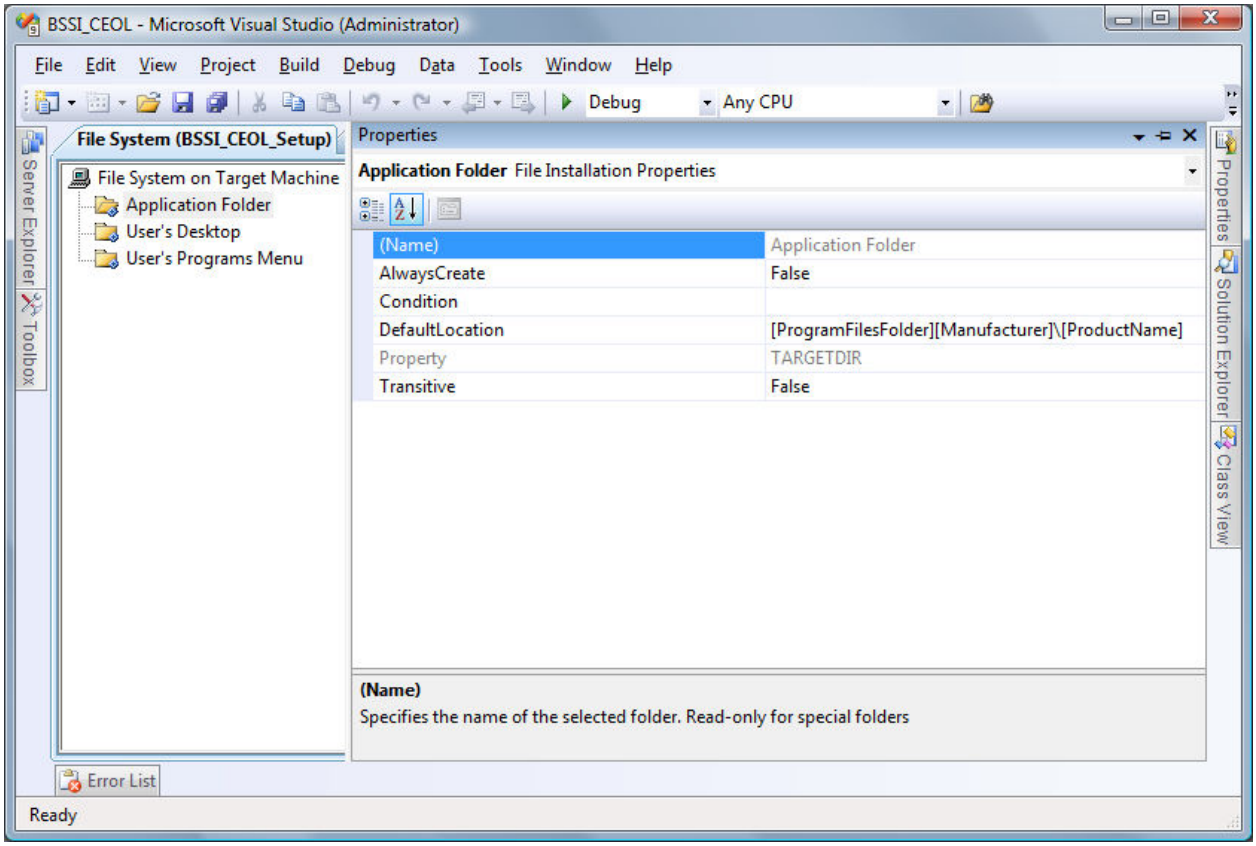
There may be some properties of the .msi project which should be modified which are not included in the Setup Wizard. To access these propertied, in the Visual Studio 2008 Solution Explorer, right click on the setup project node and select “View > File System”:



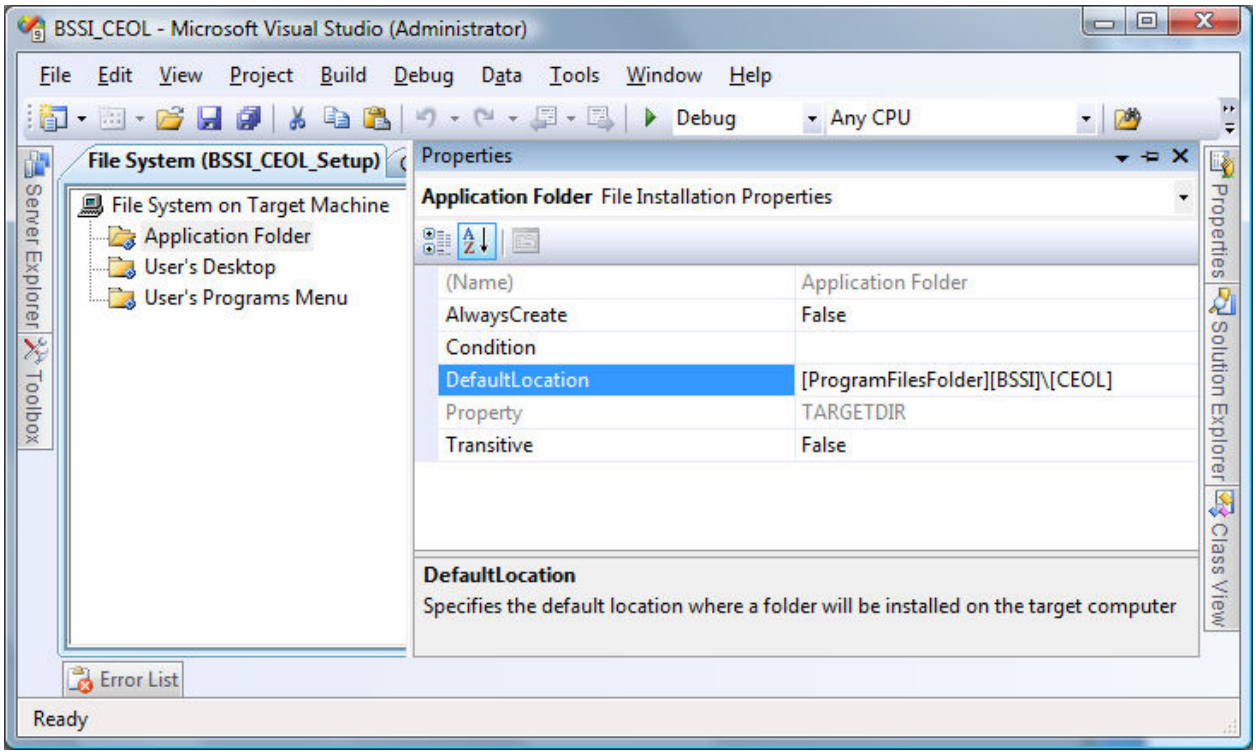
Right click the “Application Folder” node below the “File System on Target Machine” node and select the “Properties Window”:



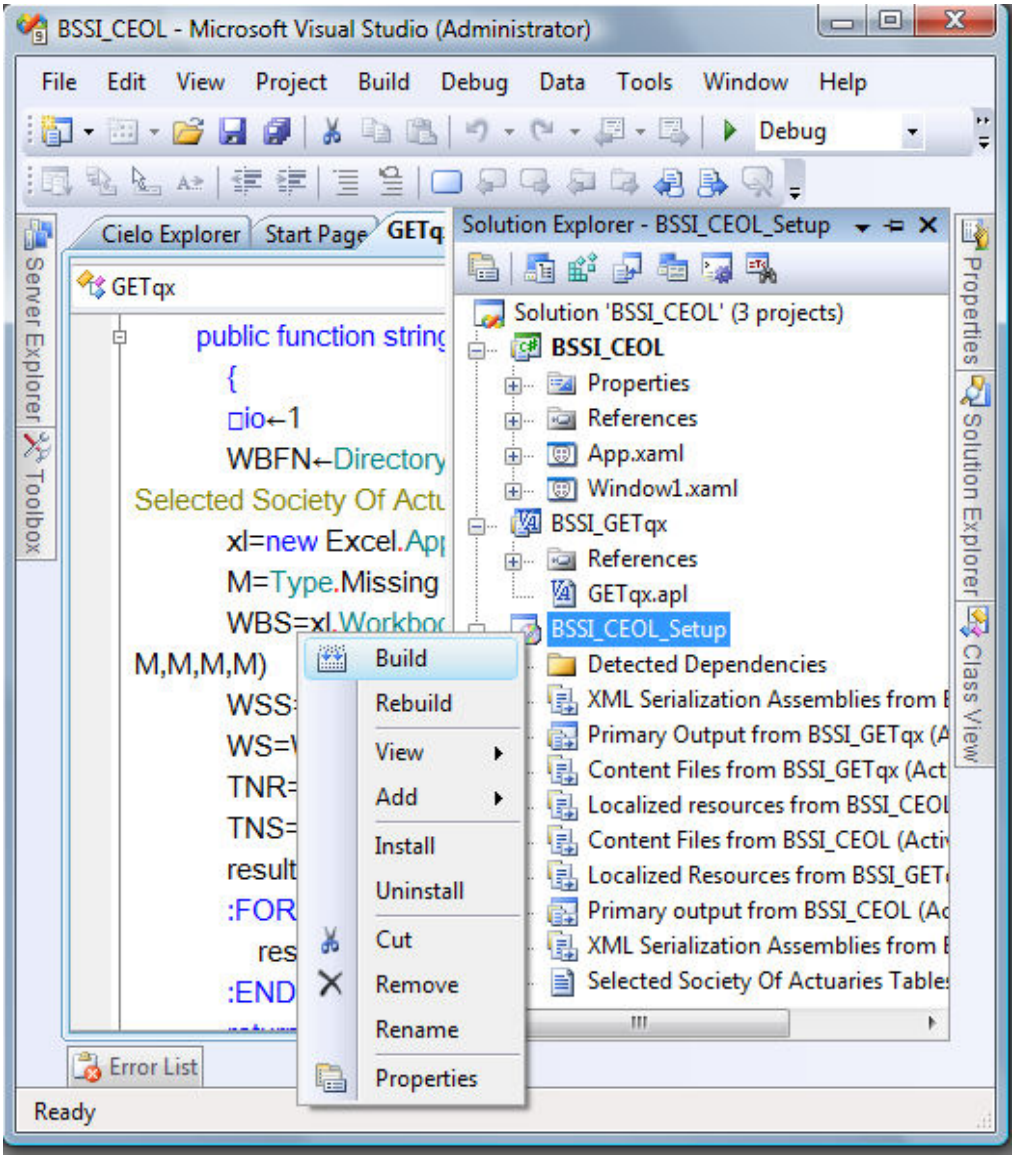
The “[ProgramFilesFolder]”, “[Manufacturer]” and “[ProductName]” properties may be edited:



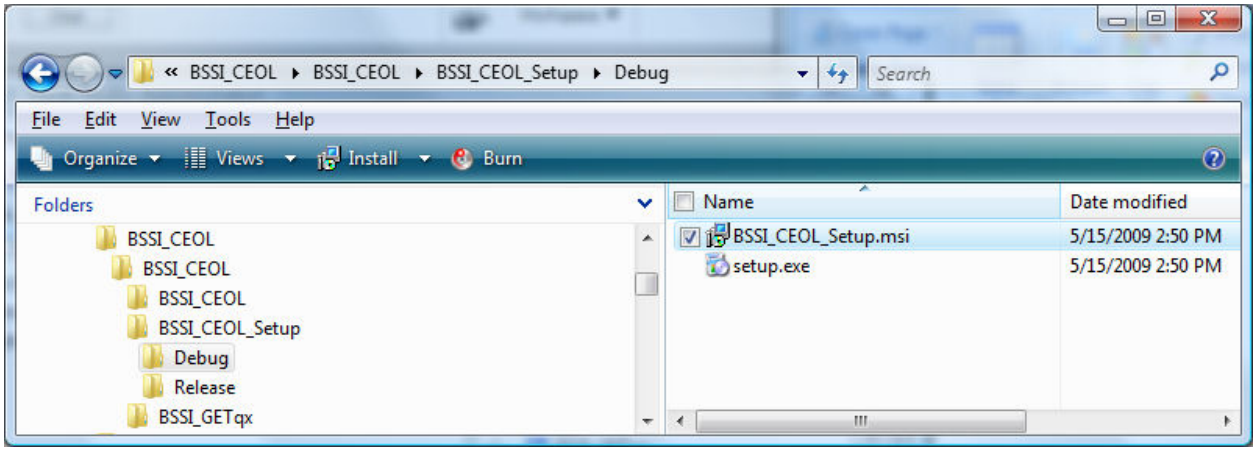
The modified properties will be incorporated into the .msi installer when the programmer builds the setup project from Microsoft Visual Studio. In the example below the “manufacturer” and “product name” have been modified.



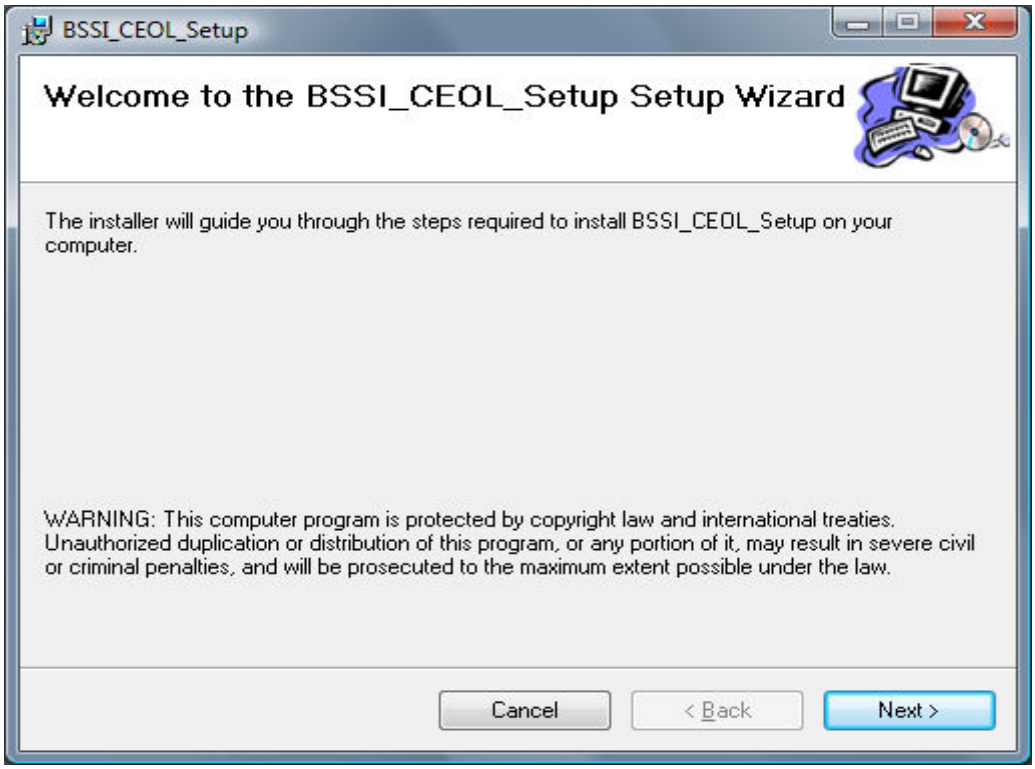
In the Visual Studio 2008 Solution Explorer, right click on the .msi project ("BSSI_CEOL_Setup") node and select "Build" to create the .msi installer for the solution:



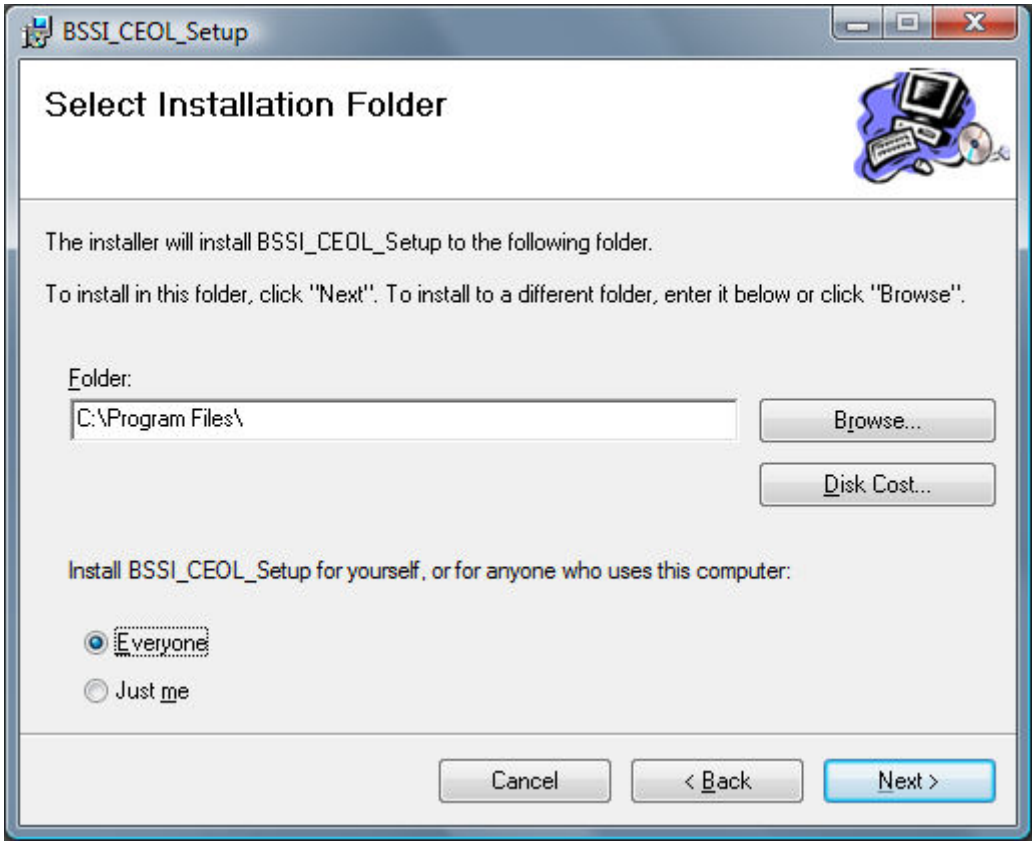
To test the solution .msi installer, close Microsoft Visual Studio 2008, use Windows Explorer to browse to the "\debug\" subdirectory of the .msi installer and double click the .msi installer:



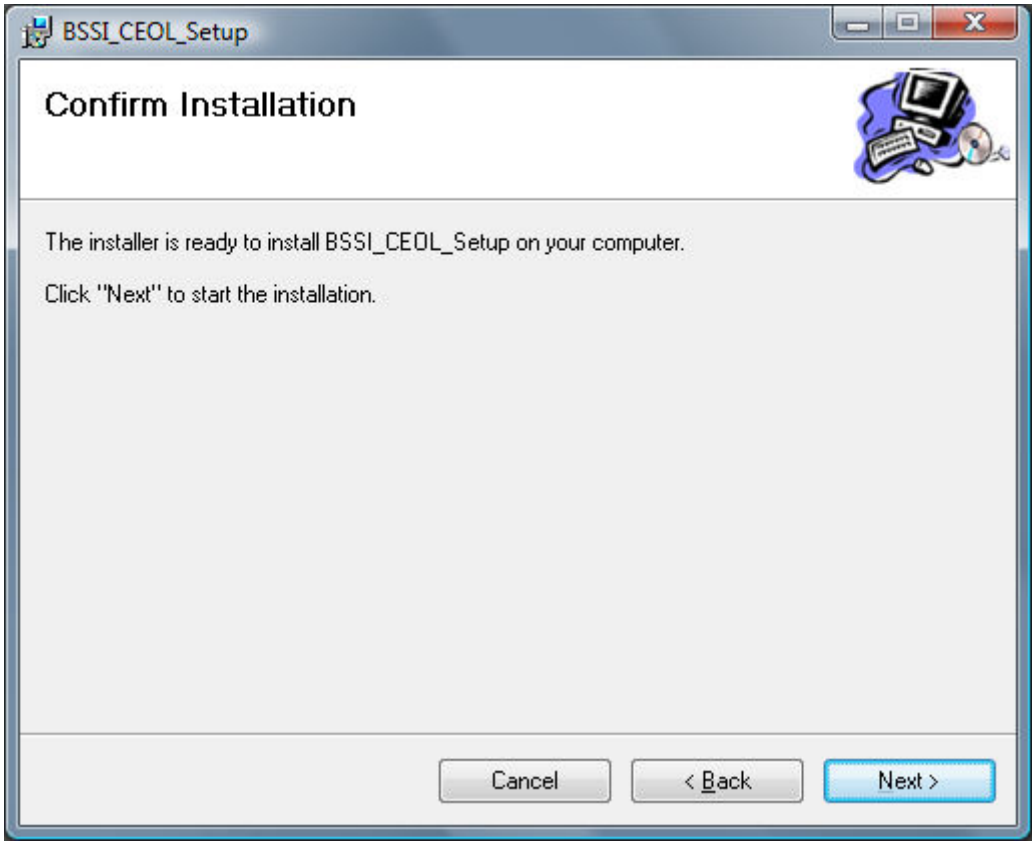
The .msi installer form will be presented – click the "Next" button:



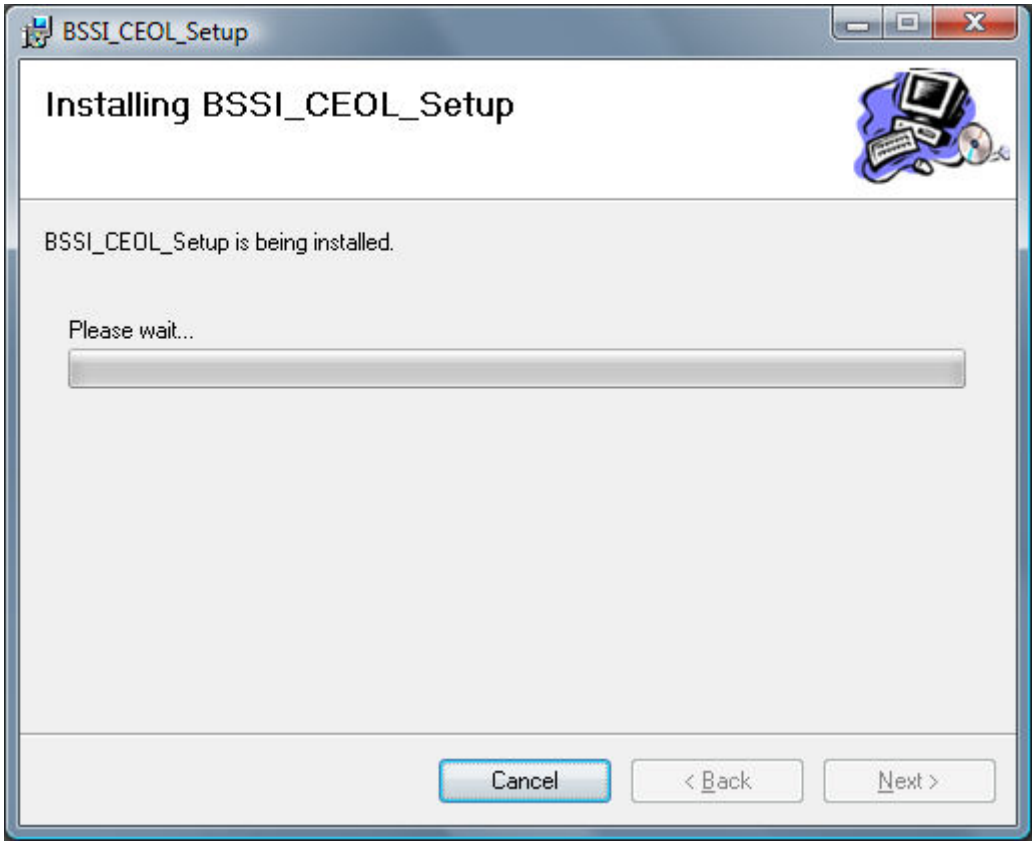
Select the installation folder and scope of the installation and click the “Next” button:



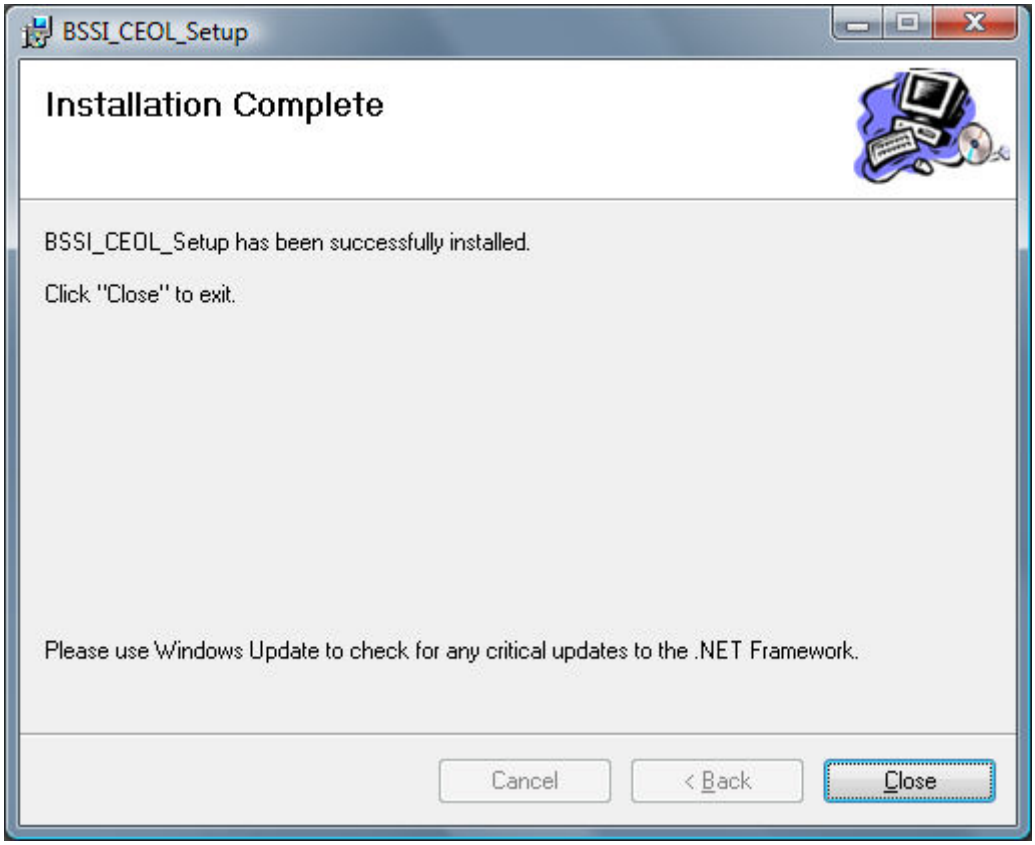
Click the “Next” button to continue:



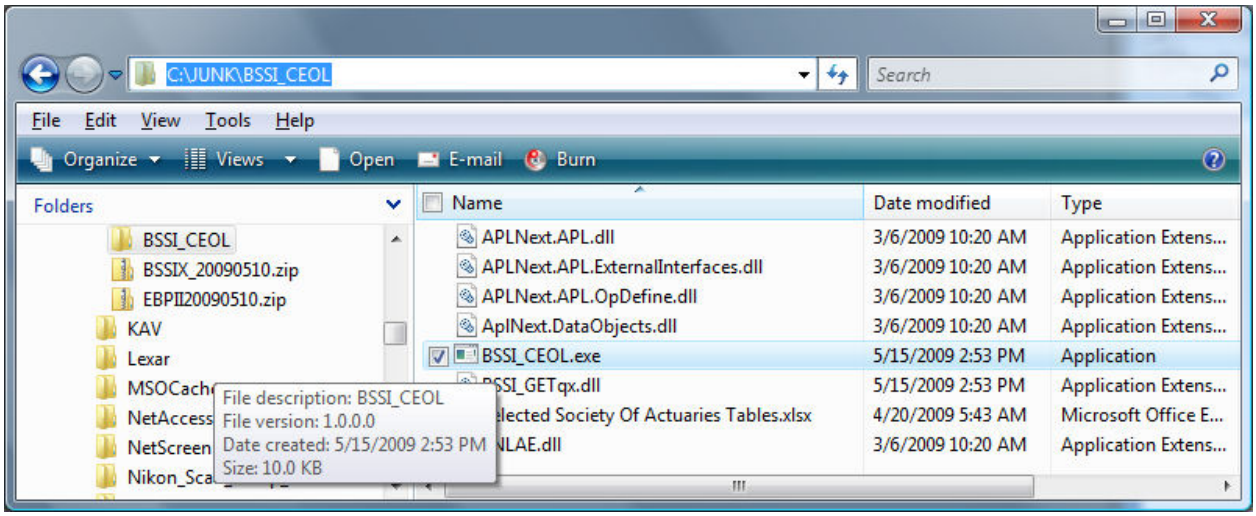
This form will be displayed while the installation is occurring:



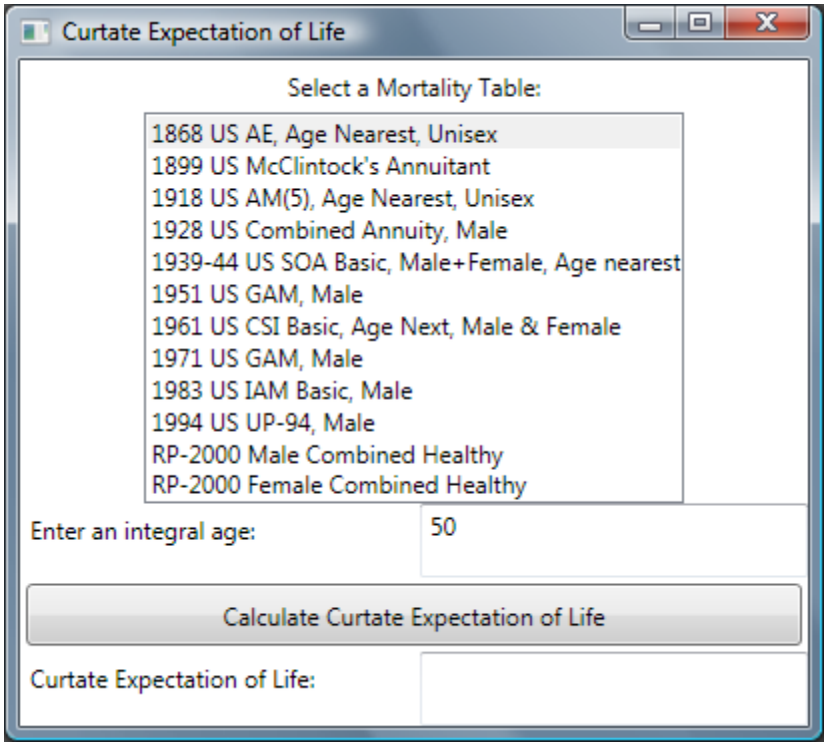
Click the “Close” button to complete the installation:



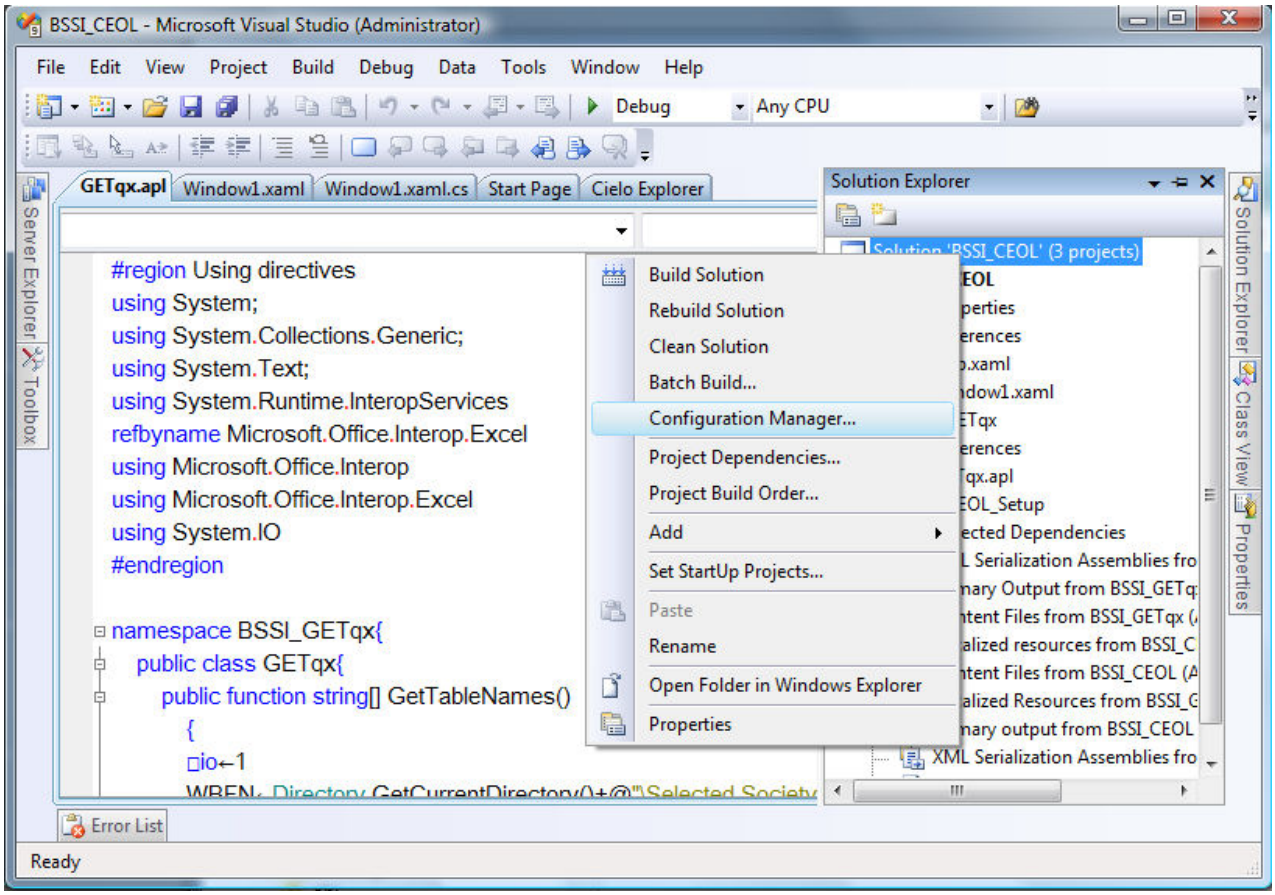
To test the application system solution as installed by the .msi, browse to the location where the application system was installed by the .msi installer and double click the application system .exe file ("BSSI_CEOL.exe"):



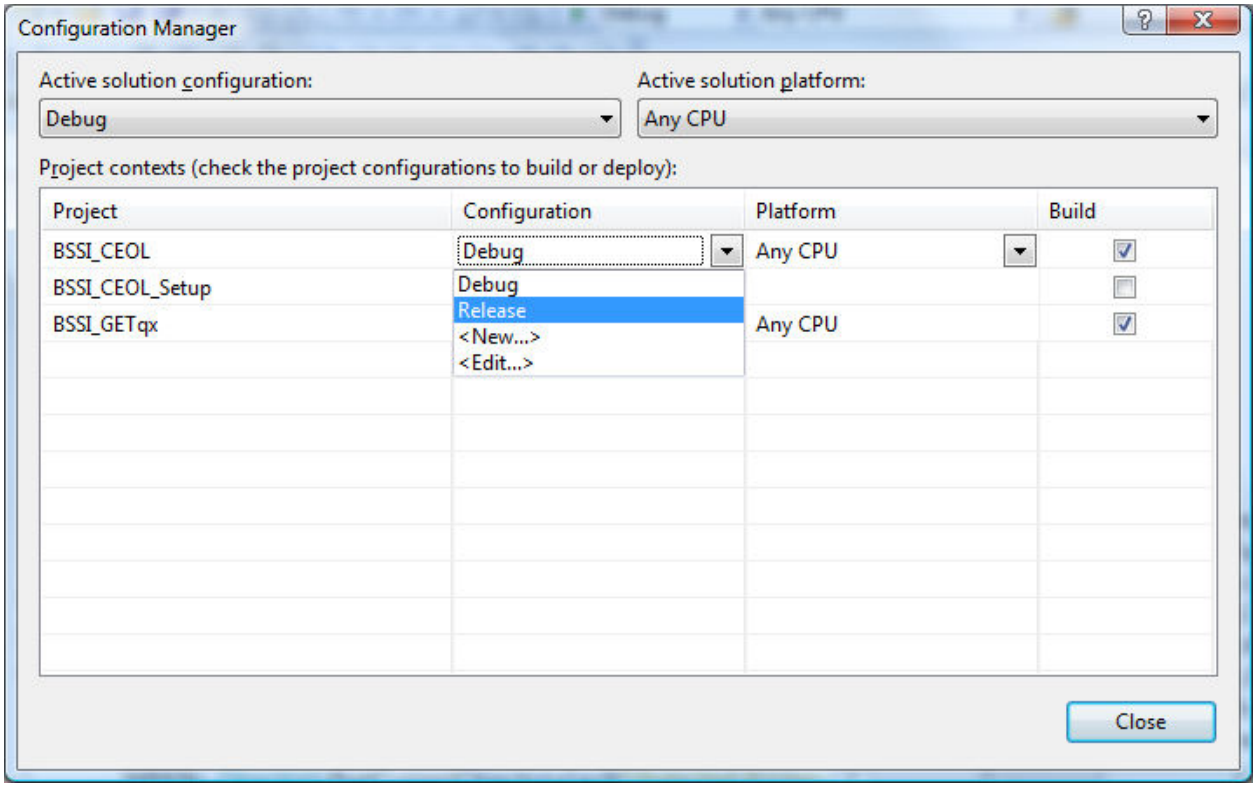
The application system main form will be presented and the application can be now be used:



The “Builds” which have been illustrated are configured (by default by Microsoft Visual Studio 2008) as “Debug builds”. This means that incorporated into the resulting compiled code will be debugging information useful to a Visual Studio 2008 programmer. The “Debug builds” are placed into the “…\bin\debug\” subdirectories. Eventually when the application system solution projects and the associated .msi installer project are thoroughly tested a production version of the compiled files should be prepared. This is controlled by the Visual Studio 2008 Configuration Manager. In the Visual Studio 2008 Solution Explorer, right click the “Solution…” node and select “Configuration Manager…”:



The “Configuration” option can be modified to “Release” when it is appropriate to build the solution files for a production environment:



It is also possible to deploy a solution as a 1-click installer published to a web server. This topic is not covered here.