Programming Review of Visual Basic.NET for the Laboratory Automation Industry

Robert Harkness,[1] Malcolm Crook,[1] and David Povey[2]*
[1]Process Analysis & Automation, Farnborough, Hampshire, UK
[2]University of Surrey, Guildford, Surrey, UK

Keywords:
visual basic,
.NET,
object oriented,
laboratory
automation

The latest version of Microsoft Visual Basic (2005) is built upon the Microsoft .NET Framework. It has finally become a fully-fledged Object Oriented Language with all the associated features one would come to expect. It allows Visual Basic programmers to tackle much larger applications, through improved scalability and reusability. This article discusses the new features using code examples to real applications in the Laboratory Automation environment. (JALA 2007;12:25–32)

Introduction

Microsoft Visual Basic (VB) has proved to be a popular programming language, especially in business programming.[1] Take up by end users within the Laboratory Automation industry appears to be fairly high. This is because of the familiarity many people have with VB for Applications (VBA) through using other Microsoft products such as Excel and Word. The easy to use syntax enables rapid application development (RAD) that has made it suitable for creating small programs that are commonly required in the laboratory.

The latest incarnation of VB is known as VB 2005. It is a programming language that uses the Microsoft .NET Framework version 2.0, a new form of Windows API that is much more extensive and Object Oriented. This has been a massive change for VB since it was introduced back in 2002 under the name VB.NET. This is because the Framework is substantially more advanced in terms of objects and components compared to COM, on which VB was previously based.

This article will attempt to address the key areas on how VB has changed and how it can still be used in the Laboratory Automation industry for creating specifically desktop Windows Applications.

*Correspondence: Robert Harkness, B.Sc., Process Analysis and Automation, Laboratory Automation, Falcon House, Fernhill Road, Farnborough, Hampshire United Kingdom; Phone: +44.1252.554062; E-mail: rob.harkness@paa.co.uk

1535-5535/$32.00

doi:10.1016/j.jala.2006.10.014

Language Discussion

Object Orientation

VB.NET represents a significant shift from an essentially procedural-based language (VB6) to one that demands an understanding of Object Oriented (OO)[2] programming concepts. Although it is still possible to create applications for the laboratory that hardly make any use of objects, a thorough understanding of the .NET Framework is essential to fully exploit the language.

VB4 was the first version that allowed use of classes in applications, a major improvement at the time. Many people then proclaimed that VB was an OO language. If the standard definition of an OO language was followed, a key requirement of an OO language is Inheritance; however, this was not available in VB4. Inheritance is now a key feature of VB.NET. All of the other OO features, such as Abstraction, Polymorphism, and Encapsulation are

present, as they were in the previous versions of VB6 although they are implemented slightly differently as shown in the examples below.

In Code Example 1—Abstraction and Inheritance, there are two classes *ThermoReader* and *TecanReader*. Both are inheriting from the base class *PlateReader* which has a *Read* method. The Base class is an abstraction of a Plate Reader, the ability to write a class that represents something.

```
Public MustInherit Class PlateReader
   Public MustOverride Sub Read()
End Class
Public Class ThermoReader
   Inherits PlateReader
   Public Overrides Sub Read()
      'Code here to run this reader
   End Sub
End Class
Public Class TecanReader
   Inherits PlateReader
   Public Overrides Sub Read()
      'Code here to run this reader
   End Sub
End Class
```

Code Example 1. Abstraction and Inheritance.

Notice in this example how the base class is constructed. The keyword *MustOverride* is used for the method *Read* in the *PlateReader* base class. When *ThermoReader* inherits the base class, it must implement its own version of the method *Read* as indicated by the *Overrides* keyword preceding it.

As both classes are inheriting from the same base class, a method can be created that can run either type of reader, irrespective of the fact that each object is created from a different class, a concept known as Polymorphism.

```
Dim objThermo As New ThermoReader
Dim objTecan As New TecanReader

RunReader objThermo
RunReader objTecan

Public Sub RunReader (ByVal Reader As PlateReader)
   Reader.Read()
End Sub
```

Code Example 3. Polymorphism.

The *ThermoReader* class can be extended to include a property that represents where generated data will be stored when a Read has completed even though this property is not in the base class (see Code Example 4—Encapsulation).

```
Public Class ThermoReader
   Inherits PlateReader
   Private mDataPath As String
   Public Overrides Sub Read()
      'Code here to run this reader
            'Saves data to path specfied in 'mDataPath'
   End Sub
   Public Property DataPath() As String
      Get
         Return mDataPath
      End Get
      Set(ByVal value As String)
         mDataPath = value
      End Set
   End Property
End Class
```

Code Example 4. Encapsulation.

Every object that is instantiated from the *ThermoReader* class will contain the property *DataPath* and the value of this property will always be contained within the object. Wherever the object is available, the value of the property *DataPath* will also be available to as it is encapsulated within the object.

Types

VB.NET is very flexible when working with types because it can behave as both a static and dynamic typed language.[3] Turn on *Option Strict* in VB.NET by entering the keyword at the top of each class or as a global setting for the project and it becomes statically typed. If this option is off, the language behaves as a dynamic typed language. This choice allows the programmer to decide what level of type checking he/she wants at compile-time. If *Option Strict* is turned on, type errors are highlighted at edit time. For most work, *Option Strict* should be on, to reduce type software bugs. However, static typing can be restrictive and there are times when dynamic typing makes life easier. An example is when working with COM (Component Object Model) on which VB6 is based. If *Option Strict* is turned on, late binding is not possible.

All of the types that were included in VB6, apart from *Variants* and *Currency,* are present in VB.NET although they are different in structure. For example, the *Integer* type is now 32 bit and not 16 bit, plus there is support for unsigned *Integers*.[4] Essentially, everything is an *Object,* so it is still possible to work with unknown types if required. This replaces *Variants* in the previous versions of VB. The problem with using *Object* for unknown types of data is that it is very prone to error as assumptions are made that the data is of a certain type. VB.NET has also introduced *Generics* to address this issue.[5] Code can be written to handle types before knowing what the type will be. The example below is a very simple class that behaves as a collection.

```
Public Class GenericCollection(Of itemType)
   Private mCol As New Collection

   Public Sub Add(ByVal item As itemType)
      mCol.Add(item)
   End Sub
   Default Public ReadOnly Property Item(ByVal Index As Integer) As itemType
      Get
         Return DirectCast(mCol(Index), itemType)
      End Get
   End Property
End Class
```

Code Example 6. Generic Collection.

Instead of referring to an explicit type, the placeholder *itemType* is used. When creating an instance of this class, *itemType* is set as a type as shown in the example below, that is, instantiating the class to handle Strings.

```
Dim colGen As New GenericCollection(Of String)
```

In Code Example 6—Generic Collection, the function *DirectCast* is used. It is strong typed and ensures that the data is or inherits from the expected type.[3] There is also the *CType* function that will convert data to a different type. This is weaker typed than *DirectCast* and useful for converting, for example, a String to a numeric type, although the runtime performance is slower.

Operating Systems

Because the language is based upon the Microsoft .NET framework, it is limited to creating applications that run on Windows. Web applications are a slight exception, in that they support cross-platform clients, but deployment does require a Windows Web server. The .NET framework is also available in a 64-bit variation.[6] All of the functionality in the 32-bit version of the framework is present; the difference lies behind the scenes in the Common Language Runtime. There is no real noticeable difference in working in both environments. More often than not, programs compiled in a 32-bit OS (Operating System) such as Windows XP can be run on a 64-bit OS. As well as desktop applications, programs can be written to run on a PDA or Mobile Phone by making use of the .NET Compact Framework, a subset of the main Framework.

GUI Development Tools

One of the major factors in take up of any language is how good the IDE (Integrated Design Environment) is. The IDE for VB.NET 2005 is used by all the languages that are part of Microsoft Visual Studio 2005 (Fig. 1), these are Visual C#, Visual C++ and Visual J#.[7]

The IDE has form designers for Windows Forms, traditional Windows applications, and Web Forms using ASP.NET pages. Designing and coding Windows Forms and Web Forms has been made as similar as possible so that both

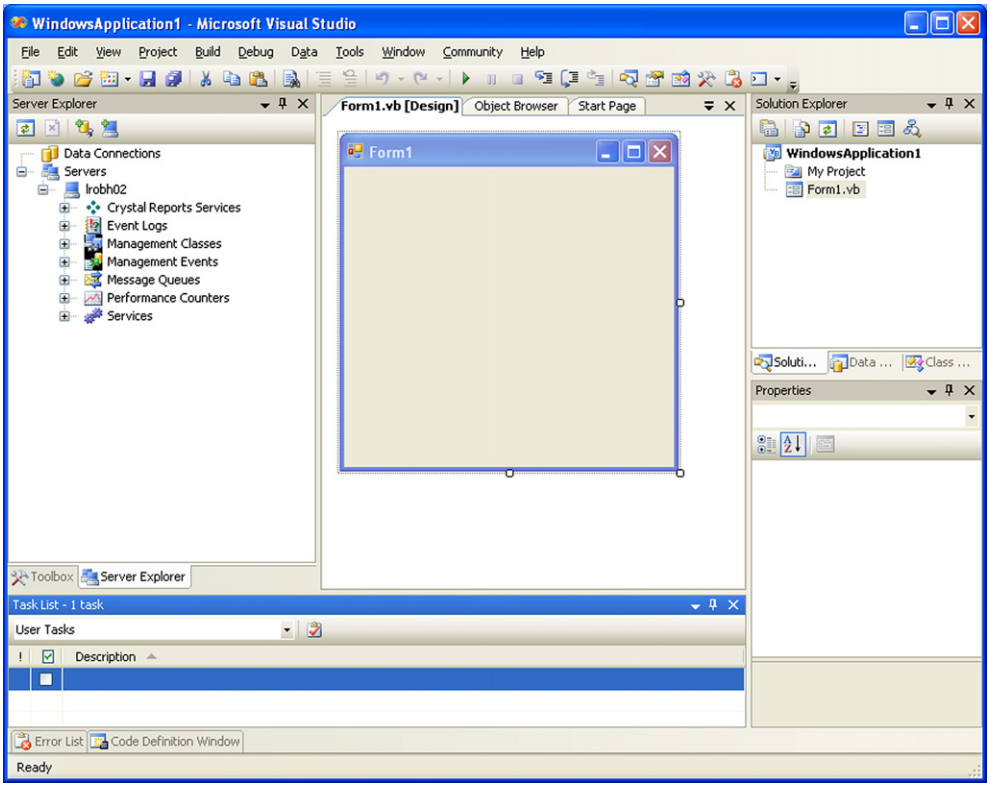

Figure 1. Microsoft Visual Studio 2005 IDE.

types of applications can share components and much of the complexity of coding Web applications is kept hidden. The support for writing code is a major advance in the IDE. When writing code, suggestions are made and syntax errors and warnings are highlighted. There is a library of reusable code snippets that can be dragged into the code window to speed up development. It is also possible to define and reuse your own code snippets.

Debugging has always been well supported in VB and this has been enhanced. One useful addition is the help dialog that will be displayed with trouble-shooting tips and recommendations on what changes need to be made when an error is encountered in the code. All code can be edited at this point and execution resumed. Also included are *Tracepoints* which are breakpoints that don't break. Instead, they can report the current program status such as a variable value and let the code continue. This is very useful for programs where having a break in the code would create an unrealistic testing environment such as when waiting for a response from an instrument.

However, it is the new features and tighter integration to data that really make the difference over previous versions. For example, it is now possible to test the methods and properties of a class using the Object Test Bench. An instance of the class can be tested without having to write a test harness for all but the most complicated classes. This is particularly useful for applications that work with instrumentation as you can test each device, assuming a class has been written for each, before testing the entire program. Although this feature would not be high on the list on why you should use Microsoft Visual Studio, it is these additions to the IDE that makes it easier to create a program.

Visual Studio 2005 is available in various editions, Express, Standard, Professional, Tools for Office, and Team System.[8] The *Express Edition*, which is free to download from the Microsoft web site, is a cut-down version targeted at novice developers and is an ideal choice if you wish to complete a basic evaluation of what the product has to offer. As we move up through the versions, the price and the available features increase. The high-end products are the *Team System Editions*. These are supplied individually as *Software Architect*, *Software Programmer*, and *Software Tester* packages or the *Team Edition*, which is a package of all three versions. The *Team System Editions* are primarily for large development organizations and are not really suitable for writing the types of applications found in the laboratory. The most suitable choice is the *Standard* edition, which is the cheapest version available that has all the functionality required to write a wide range of applications. This version also comes with the Microsoft Developers Network help library, a very useful resource for anyone who takes VB programming seriously.

MultiTasking

There are two ways of running code asynchronously in VB.NET, using Delegates[9] or Threads.[10] Most VB programmers will be familiar with the concept of Events because it is an event driven language. For example, when you click a button a click event is raised. Events can be raised from inside a class so that when a particular event occurs, the client that instantiated the object can be notified. Events are still present in VB.NET but are now based upon Delegates.[11] Using Delegates, it is possible to start a method asynchronously, similar to an event, but it can also have a callback raised upon completion, something which Events cannot do. To run a method through a Delegate, in this example the *Read* method in the *ThermoReader* class, a delegate must first be declared.

An instance of the delegate can then be created using an instance of the *ThermoReader* class. The signature of this delegate must be the same as the object it will be working with. In this example, the *Read* method of the *ThermoReader* class has no parameters, so neither does the delegate, *DelegateRead*.

```
Delegate Sub DelegateRead()
```

An AsyncCallback object must also be created that passes in the method that will be called when objThermo.Read has completed. The only proviso is the method has the parameter of the type *IAsyncResult*.

```
Dim AsyncRead As DelegateRead = New DelegateRead(AddressOf objThermo.Read)
```

It is now possible to invoke the delegate which will execute *objThermo.Read* along with the Asynchronous callback that has just been created.

```
Public Sub ReadComplete(ByVal IAsyncR As IAsyncResult)
  MessageBox.Show("Read Complete")
End Sub
Dim asCallBack As AsyncCallback = New AsyncCallback(AddressOf ReadComplete)
```

Code Example 7. Creating an Asynchronous Callback.

BeginInvoke will not be a blocked call and any code immediately after this call would be executed. For example, the other reader, *TecanReader*, could be started so both readers are multitasked. When the "*Read*" method completes, the Asynchronous Callback is executed as shown in Code Example 7—Creating an Asynchronous Callback. Delegates are probably one of the most radical changes to VB.NET and it is certainly not the easiest concept to understand. However, it is a powerful feature that can make applications run at a faster speed.

```
Dim state As Object = New Object
AsyncRead.BeginInvoke(asCallBack, state)
```

Code Example 8. Invoke Delegate Using the Asynchronous Callback.

The other way to run code asynchronously is to use Threads. There are various ways to implement threads, the simplest being the BackgroundWorker component. The best method to

gain complete control over a thread is to use the Thread class. This can be implemented using the following code, which again uses the *ThermoReader* class from the previous examples.

```
Dim ReadThread As Thread = New Thread(AddressOf objThermo.Read)
ReadThread.Start()
```

Code Example 9. Run Method in New Thread.

This code will run the *Read* method in a separate thread to the calling code. To then wait until the thread has completed, to synchronize with the calling thread, the following code should be used.

```
ReadThread.Join()
```

There is much more functionality available that allows complete control over the behavior of each thread. It should be noted that although the example shown is relatively simple, writing multithreaded applications that are thread-safe is difficult because threads tend to need to interact with each other. For example, the *Read* method in the *Reader* class may contain code that interacts with a resource, hardware, or software that can only be interacted with by one caller at a time. If two or more instances of the *Reader* class were then run in separate threads, there could be a scenario where all of the threads are trying to access the resource simultaneously. This often leads to a fatal errors such as memory violations, although the IDE does a good job in handling many of these exceptions.

There are various synchronization techniques available to ensure that code is thread-safe. One way is to use a *Lock*. In Code Example 10—Sync lock the Read Method, a lock is placed around the code that interacts with Eventlog, which in this example is a resource that writes to a data device. The lock around this code prevents more than one thread accessing this code at any one time. If a thread was running this section of code, another thread which was also running this "Read" method would wait when it gets to this point until the first thread has completed this section and released the lock.

```
Public Sub Read()
  Dim WriteLogLock As New Object
  '### Code here to run this reader ###
  SyncLock WrteLogLock
    '### Code in here to write to EventLog object ###
  End SyncLock
End Sub
```

Code Example 10. Sync Lock the Read Method.

Thread synchronization is very important when developing multithreaded applications and requires careful planning to ensure it is thread-safe. The main advantage to using threads is the increased speed in running tasks through the application. In the previous code example, we are making provisions for running Plate Readers at the same time as opposed to one after the other. When running multiple plates in an assay, through many devices, the time savings can be significant, fully justifying the extra time and effort required to develop a multithreaded application.

Hardware Interfacing

VB.NET comes with a series of control components that makes it a fairly simple process to integrate hardware. Most devices found in the laboratory are connected to the PC using one of the following types of connection.

- RS232/RS485
- Ethernet

RS232 and RS485 can be handled using the SerialPort component that is part of the Framework and available in the IDE. The control is then simply dropped onto the form that will be used to send commands, although it is also possible to create an instance of this control without having to place it (sink it) onto a form. For example, if an instrument is used that has a command set that uses the string "HOME" to move the instrument back to the home position, the following code would execute this.

```
With SerialPort
  PortName = 1      'Set Port
  WriteLine("HOME")  'Send Command to Device
End With
```

Code Example 12. SerialPort Component.

Ethernet connections can be made by using the System.Net.Sockets namespace in the .NET Framework. The principles behind using sockets in previous versions of VB are pretty much the same in VB.NET. The main advantage now is the ability to use delegates which gives much more flexibility over the asynchronous control of each socket.

Data handling

XML

It is no surprise that the main way to handle data in VB.NET is by using Extensible Markup language, known as XML.[12] The .NET framework is supplied with two base classes, XMLReader and XMLWriter. These are the foundation classes for working with XML, making it very easy to write an application that uses XML documents. For example, to write software to reformat data as it is produced from an instrument such as a 2D Barcode Reader, XML would be by far the best choice as seen in Code Example 13—2D Barcode XML Data.

```
<?xml version="1.0" encoding="utf-8" ?>
<Plate Barcode ="A123456">
  <Well Barcode="DEF123">A1</Well>
  <Well Barcode="DEF234">A7</Well>
  <Well Barcode="DEF345">C5</Well>
  <Well Barcode="DEF456">H8</Well>
</Plate>
```

Code Example 13. 2D Barcode XML Data.

Using the *Reader* class, we can query the attribute of each element in the XML file with the barcode of the tube we want to return its location. This is a very attractive way to control cherry picking applications. An XML Template file, XSL, could be written for the data so that any other application could read and most importantly, understand what the data represented.

Serialization

In any program that uses objects, data is moved around and sometimes it is necessary to save the data. Previously, the best way was to create a load and save method in the class and then write code to read and write all the properties of the class to the required format. To use XML, the Document Object Model available in the Microsoft XML Parser, MSXML, has to be used. With .NET, Serialization is available and once again it simplifies the whole process for persistent data. To achieve this, pass the object as a constructor parameter into a new instance of the Serialization class. Then define the output stream, XML or Text and call the serialize method. All public properties are then serialized to the defined stream. This functionality is bidirectional so it is possible to deserialize into a new instance of the object. A lot of assumptions are made in the process and the object has to be fairly simple, although it can handle properties that are collections. If data has to be handled differently, implement the Serialization interface in the class, and using the relevant stream, specify how to handle the data.

ADO.NET

In business programming, VB has one of the largest user bases and one of the primary reasons for this is because of the built-in tools for database connectivity. In VB.NET, ADO.NET (Abstract Data Object) can be used to connect to a variety of databases such as SQL Server, Oracle, and MS Access.[13] Nearly all data handling in the .NET environment uses XML, and ADO.NET is no different. All data sets are moved around as XML and because XML is now an industry standard, any application on any platform that can read XML, can receive data from an ADO.NET enabled application.[14] Another key improvement is the ability to use typed data sets allowing data to be accessed through typed programming. A problem with database programming has been writing SQL statements at design time and then only finding out there is a problem when running the application. Using typed data sets, where the field names are actually properties of the data set, this problem is removed because the code is checked at compile time.

Example Application

The following example looks at how a small Windows application can be created to control a Thermo Labsystems Multidrop 96/384. This is a simple device to integrate and one of the most common instruments found in the laboratory.

The first step is to look at the required functionality. Use-case analysis would reveal the following methods:

- *SetPort*
- *Initialize* the carrier
- *Prime* a set volume
- *Empty* a set volume
- *Dispense* a set volume to a whole 96 or 384 plate.

Although *SetPort* has been identified as a method for setting the RS232 port number, it would make more sense for this to be a property. For this example, ignore the fact that it is possible to dispense to different columns and consider only the entire plate. Now create a new windows application project in Visual Studio. The default form should be renamed to *frmMultidrop* and this will also be the startup form. A new class file is then added and this is called *clsMultidrop*. This class is the abstraction of the Multidrop instrument and will contain the methods and properties as described above.

Next, create a class for sending commands using RS232. This class will need a method to send the command and another property to set the port number. In this example, another class file will be created to implement this new class even though it could be written in the existing *clsMultidrop* class file.

```
Imports System.IO.Ports
Public Class clsRS232
    Private mSerialPort As SerialPort
    Public Property Port() As String
        Get
            Return mSerialPort.PortName()
        End Get
        Set(ByVal value As String)
            mSerialPort = New SerialPort(value, 9600, Parity.None, 8, StopBits.One)
        End Set
    End Property
    Public Sub Send(ByVal Command As String, Optional ByVal Timeout As Integer = 5)
        Dim sRet As String
        With mSerialPort
            Try
                .Open() 'Open the COM Port
                .ReadTimeout = Timeout * 1000 'Set Timeout to be 30 Seconds
                .WriteLine(Command)  'Send Command to Device
                sRet = .ReadLine 'Read up to terminiator(CF-LF)
                If sRet <> "OK" Then Throw New Exception("There is an error")
            Catch e As TimeoutException
                MessageBox.Show("Timeout")
            Catch e As Exception
                MessageBox.Show(e.Message)
            Finally
                .Close() 'Close the COM Port
            End Try
        End With
    End Sub
    Public Sub New(ByVal RS232Port As Integer)
        Port = RS232Port
    End Sub
    Public Sub New()

    End Sub
End Class
```

Code Example 15. RS232 Serial Port Class.

The *SerialPort* object is created using an instance of the class directly from the .NET Framework using the *System.IO.Ports* namespace, instead of using the *SerialPort* component. The *clsRS232* class shows another feature now available and that is the ability to use constructors with parameters. In previous versions, it was only possible to have a single constructor, *New*. Constructors can now be overloaded so that when an instance of a class is created, the parameters can be defined in the same line of code. An example of this is highlighted, in Red, in Code Example 15—RS232 Serial Port Class.

Now create an instance of the RS232 class in the Multidrop class so that each method can send the relevant command to the Multidrop.

```
Public Class clsMultidrop
    Private mRS232 As clsRS232
    Public Enum ee_PlateType
        pt_96 = 1
        pt_384 = 2
    End Enum
    Public Sub Initialize()
        mRS232.Send("Q")
    End Sub
    Public Sub Prime(ByVal Volume As Integer)
        mRS232.Send("P" & Volume.ToString)
    End Sub
    Public Sub Empty(ByVal Volume As Integer)
        mRS232.Send("E" & Volume.ToString)
    End Sub
    Public Sub Dispense(ByVal Volume As Integer, ByVal PlateType As ee_PlateType)
        Dim iTimeout As Integer

        'Set Plate Type
        mRS232.Send(IIf(PlateType = ee_PlateType.pt_96, "T0", "T1"))

        'Formula to calculate how long the Timeout should be
        iTimeout = IIf(PlateType = ee_PlateType.pt_96, _
              (Volume * 0.07) + 46, _
              (Volume * 0.3) + 49)

        mRS232.Send("V" & Volume.ToString)
        mRS232.Send("D", iTimeout)
    End Sub
    Public WriteOnly Property Port() As Integer
        Set(ByVal value As Integer)
            mRS232 = New clsRS232(value)
        End Set
    End Property
End Class
```

Code Example 16. Multidrop Class.

This code, encapsulated by *clsMultidrop* will, when instantiated, control the Multidrop (Fig. 2).

To produce a user interface, use the form that already exists in the project. For this example, simply add one button for each function that is available, using input controls to pass in parameters.

Conclusion

After reading this review, it should be apparent that VB.NET is seen as an improvement over its predecessors. The sheer size of the .NET Framework is perhaps overwhelming and it can be difficult to navigate to find the required function. However, this has been addressed in the latest version of VB.NET 2005 with the introduction of the *My* namespace[15] to provide quick access to the most commonly used functions.

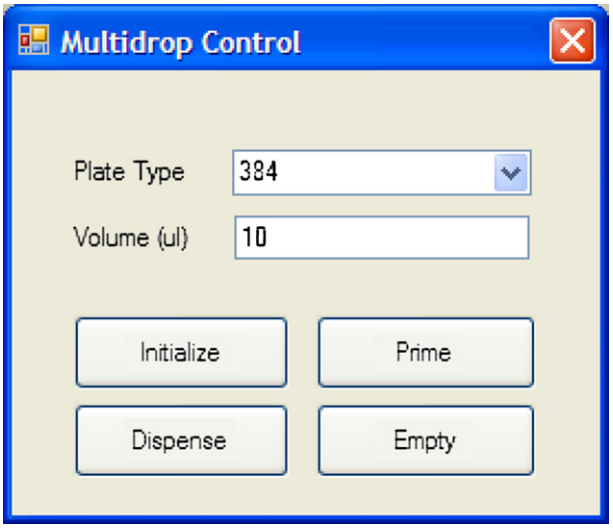

Figure 2. Multidrop Control GUI.

It is important to realize that VB.NET is not the only language that works with the Framework. Visual C#, Visual C++, and Perl also make use of the functionality that is offered. Which language is more efficient or productive really is a matter of personal choice and application specific. For some projects, C# is more suitable than VB and vice versa as each interacts with the .NET framework in different ways. There is no common consensus on the language that is best, although all programmers seem to have an opinion. The choice of language ultimately comes down to how easy the syntax is to use. An application developed in VB.NET may require more code than an application developed using C++, but if it is easier to understand and therefore quicker to code in, then this becomes irrelevant. Once the application is complied and it works as required, no-one will be concerned about the lines of code used or the language choice. Apart from a few changes such as structured exception handling, the basic syntax in VB.NET is very similar to VB6. VB.NET should still be seen as an ideal tool for developing applications in the Laboratory, for the same reasons that made VB6 suitable, with the added bonus of the .NET Framework at its disposal.

Further Reading

For more information on using VB.NET, please consult the following resources.

1. The Microsoft Developers Network, http://www.msdn.microsoft.com/vbasic.
2. Matthew McDonald. Pro.NET 2.0 Windows Forms and Custom Controls in VB 2005. Apress. August 2006
3. http://www.codeproject.com.
4. Christoper M. Frenz. Visual Basic and Visual Basic.NET for Scientists and Engineers. Apress. February 2002
5. http://www.experts-exchange.com.
6. http://www.codeguru.com/.
7. http://www.tek-tips.com.

8. Mark Russo, Martin E. Echos. Automating Science and Engineering Laboratories with Visual Basic. Wiley. March 1999

References

1. TIOBE Programming Community Index, http://www.tiobe.com/tpci.htm Accessed April 2006.
2. Introduction to Objects in Visual Basic, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vbconobjectsinvisualbasic.asp. Accessed April 2006.
3. Type System, http://en.wikipedia.org/wiki/Dynamic_type. Accessed May 2006.
4. Visual Basic.NET, http://en.wikipedia.org/wiki/Visual_Basic_.NET. Accessed March 2006.
5. Introducing Generics in the CLR, http://msdn.microsoft.com/msdnmag/issues/06/00/NET/default.aspx. Accessed May 2006.
6. 64 Bit.NET Framework, http://msdn.microsoft.com/netframework/programming/64bit/. Accessed July 2006.
7. Microsoft Visual Studio, http://en.wikipedia.org/wiki/Microsoft_Visual_Studio. Accessed July 2006.
8. Visual Studio Versions, http://msdn.microsoft.com/vstudio/products/compare/. Accessed October 2006.
9. Implementing Callbacks with a Multicast Delegate, http://msdn.microsoft.com/msdnmag/issues/03/01/BasicInstincts/. Accessed May 2006.
10. Thread Class, http://msdn2.microsoft.com/en-US/library/system.threading.thread.aspx. Accessed April 2006.
11. Events and Delegates, http://msdn2.microsoft.com/en-us/library/17sde2xt.aspx. Accessed April 2006.
12. XML Programming using VB.NET, http://www.vbdotnetheaven.com/Code/Apr2003/005.asp. Accessed May 2006.
13. Riordan, Rebbeca M. *Microsoft ADO.NET Step by Step*. Microsoft Press: 2002. Accessed April 2006.
14. Benefits of ADO.NET, http://msdn.microsoft.com/library/?url=/library/en-us/vbcon/html/vbconBenefitsOfADO.asp. Accessed April 2006.
15. Navigate the.NET Framework and Your Projects with "My", http://msdn.microsoft.com/msdnmag/issues/04/05/VisualBasic2005/. Accessed July 2006.