Intro to VB.NET with emphasis on database data entry screens

Prereqs:

- ADO.NET in 21 days, days 1-5
- Scope a data entry screen (current tables and fields) – one unique screen for each programmer (to be given on Monday and developed Thursday with XML test file)
- VS.NET workstations
- Connectivity to RDBMS from workstation via ADO.NET successfully tested

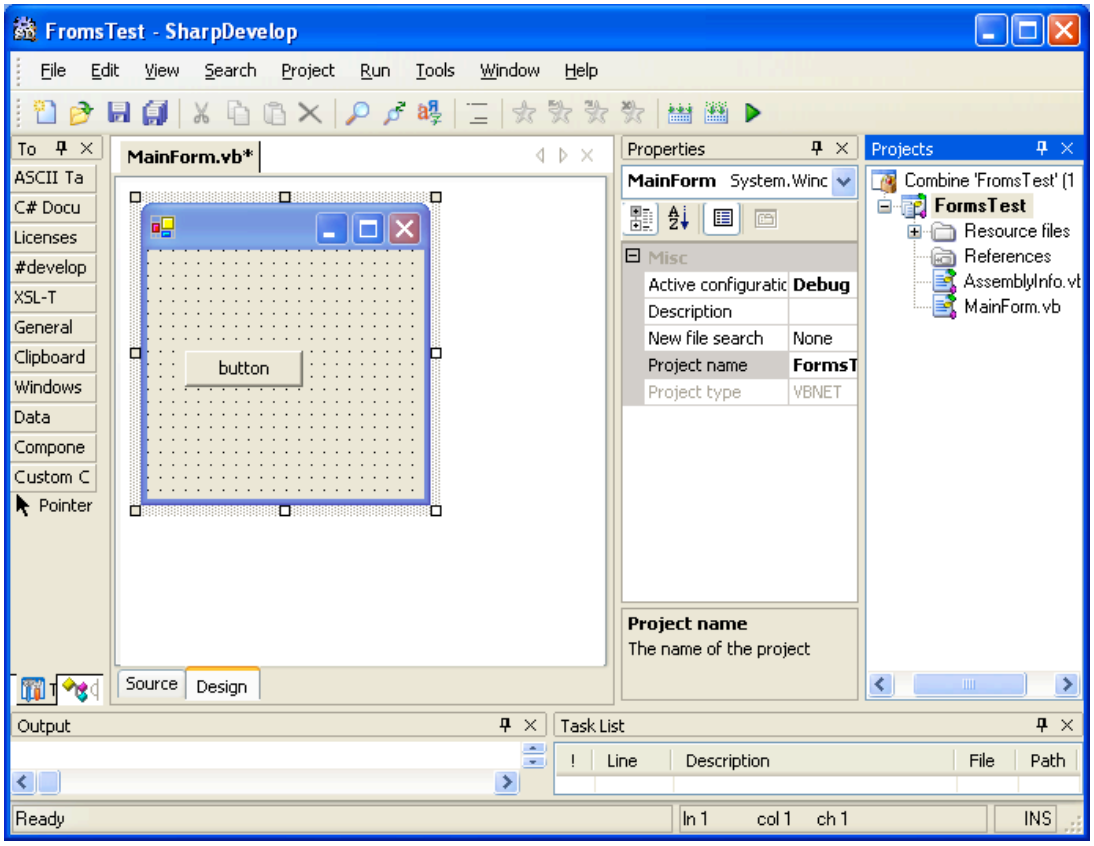
Day 1

Tools

If you've got Windows2k or above, it's pretty easy to get started programming VB.NET applications. All you absolutely have to have is Notepad and the .NET runtime/SDK. You can type up your code in Notepad and compile with the command line application, vbc.exe (on my system, vbc.exe is found here: C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322). It doesn't cost you a thing past download and installation time, though admittedly it's a large download. Though it's neat to use a text editor and the command line to build .NET applications, it's not exactly recommended.

It's quite a bit easier to use an Integrated Development Environment, however. There are two main choices for VB.NET IDEs. The best and most robust IDE is, naturally, Microsoft's Visual Studio.NET, but it's also the priciest. An open source alternative – which costs nothing to use and is ready for you to hack in any way you find useful – is SharpDevelop (or #develop), found at this url: http://icsharpcode.net/OpenSource/SD/Default.aspx

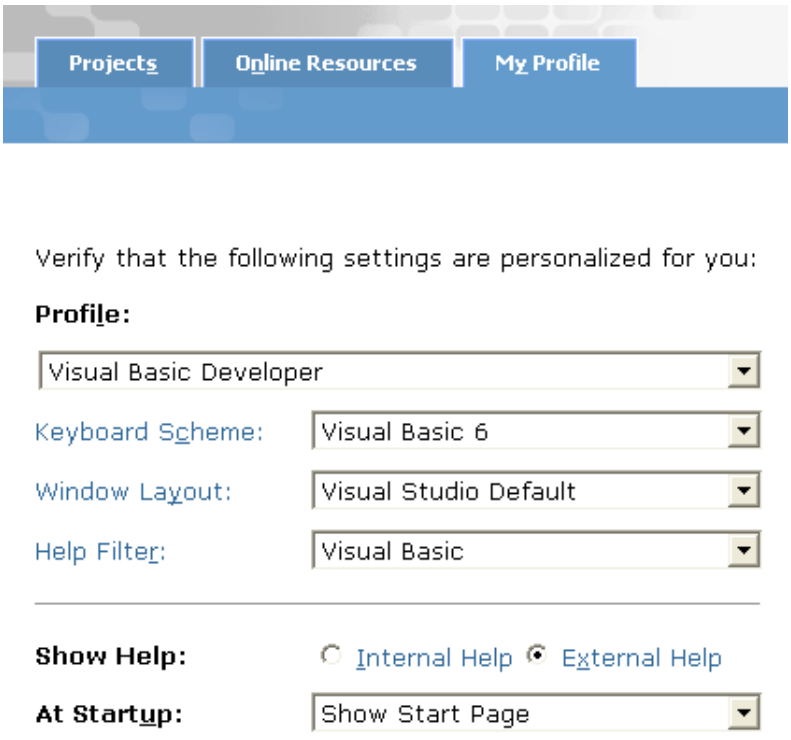
Both of these IDEs come with utilities to navigate to the .NET files on your hard drive quickly as well as design GUIs with their own respective Rapid Application Development (RAD) systems. Though its autocompletion ("Intellisense" in VS.NET-ese) isn't as good as VS.NET's and it's a bit rough around the edges, if you'd like to work on VB.NET at home, you might give SharpDevelop a look, pictured below



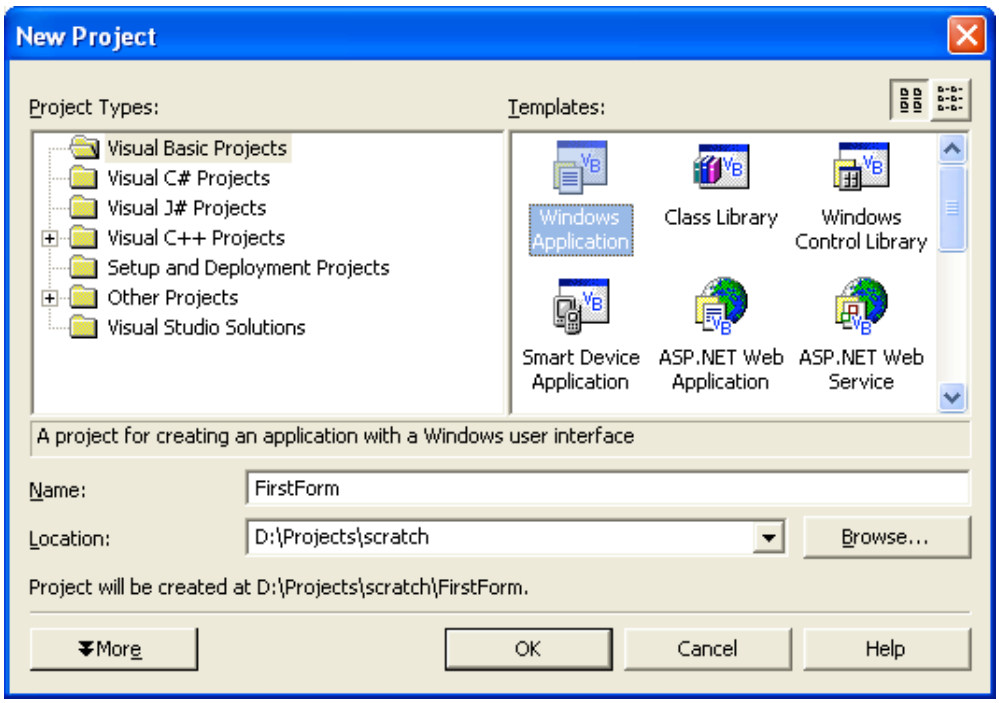
In this class, we'll be using VS.NET as we create examples.

Intro to the VS.NET, the Form, and Event Handlers

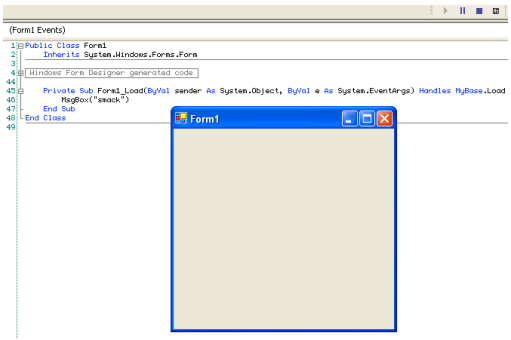
We'll start out getting as much bang for our keystrokes as we can. Open VS.NET. If this is the first time you've opened VS.NET on your machine, you'll see the "My Profile" set-up screen. If in doubt, match the settings you see below.



Choose "New Project". Make sure you've got "Visual Basic Projects" highlighted in the "Project Types" list and "Windows Application" selected from Templates. Give a descriptive name to your project, and select OK.



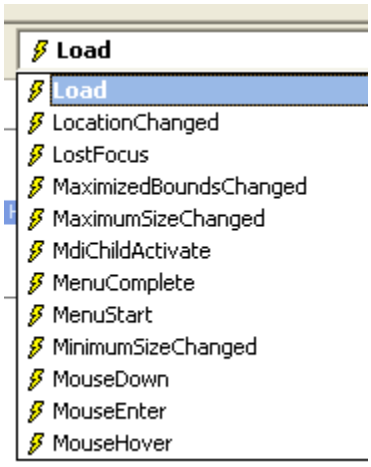
Once you do this, you should see a form, ready to be designed, on your screen. If you double-click this form, VS.NET will automatically insert code into your application to handle the Form's most commonly used event, OnLoad.



Note: A Form is a specific type of Control. A Control in .NET is essentially anything that can be displayed in a GUI. We're creating stand-alone applications for now, so we'll be using Controls from a Namespace called "System.Windows.Forms". All of our Controls will be said to extend the System.Windows.Forms.Control object. Don't worry about understanding all of these terms just yet – just remember that a Form is a specific type of Control, and that we'll be adding more Controls onto the form in the near future.

Here's what you should see after double-clicking the form in the VS.NET RAD.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

End Sub
```



The Sub we see here – a "subroutine" – is called an *event handler*. It handles the event of our newly created form loading. A form has a number of events that you can view from the drop downs atop your code editing pane, as seen on the left.

Note: The underscore, "_", can be used to put a break in a line in the code editor, but the compiler will pretend it's all still one line. So the following two bits of code are identical as far as the compiler is concerned, though one is just a touch easier to read for the coder. As declarations get longer, having each parameter on a different line is awfully useful, and is one good use for the underscore line break.

```
Public Function executeSqlQuery(ByVal strSql As String, _
    Optional ByVal strDsName As String = "DataSet") As DataSet

Public Function executeSqlQuery(ByVal strSql As String, Optional ByVal strDsName As String = "DataSet") As DataSet
```

Let's add a little code to make the form do something as it's loaded. Type in the following between the lines shown above.

```
MsgBox("Yo, Adrian.")
```

You're welcome to change the string between the quotes, of course.

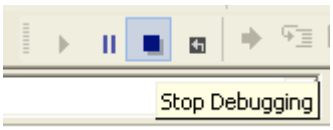
There you go, in the interest of getting the most out of our keystrokes, now hit F5 and watch the form go. Poof! You've already got a working application in .NET. To make it stop, just hit the "X" in the top right corner or hit alt-f4, or whatever you prefer. Unlike some programming languages, these "disposal commands" are already hooked up. In Java, for instance, you actually have to write code that listens for someone clicking on the "X" or hitting alt-f4 which then closes your application. For Windows.Forms, that's not a requirement.

Now say that you've accidentally written an infinite loop in your event handler for the Form's Load event, like the one below.

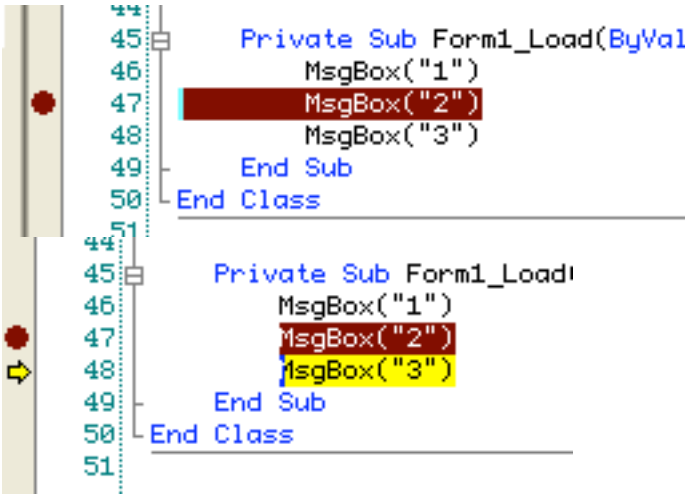
```
While True
   Console.WriteLine("I'm stuck!")
End While
```

Now you'd have to kill the application from within the IDE; you'll never get a chance to hit "X". The app will be running through this loop over and over and will never even display the form. How do you make it stop?

Look for the stop button in your VS.NET toolbars, and hit the stop "block". Voila. Application killed and you're back in design mode.



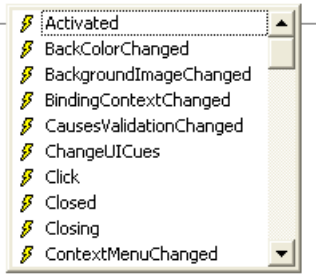
Now say we wanted to see what was going on as your application ran. VS.NET allows us to set *breakpoints* within our code to slow things down and take a look. To set a breakpoint, simply click in the "gutter" to the right of the line where you'd like the application to break. Once your application gets to that line, it will stop and highlight the line in bright yellow. You can hit F5 again to have the code head back off to the races, or you can hit F8 to move one line at a time.



Later on, when we're dealing with more complicated subroutines and functions, you'll want to remember that you can hit shift-F8 to step line by line through code without actually traveling down through the subroutines or functions in question, saving your some time. Essentially shift-F8 says, "Move to the next line in the code I can see in front of me now, regardless of where the logic might need to go in between." This is called "stepping over" code.

Now for some more complicated code. We can write event handlers for ourselves, from scratch, without double-clicking the form. Create some space in your code editor below the End Sub line but before the End Class line. Insert the following:

```
49      End Sub
50
51      public sub handRolled(sender as System.Object, e as System.EventArgs) handles MyBase.
52  End Class
53
```



As you can tell, you can handle a ton of different events that can occur in a form. Let's try handling the Click event.

```
Public Sub handRolled(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Click
    MsgBox("I've been clicked.")
End Sub
```

Hit F5 and run the application, and then click on your Form. Voila! Event handled.

Note that you can have more than one event handler listening to each event. So we could add another MyBase.Click event handler and they'd both fire. When you have more than one handler for a single event, the first handler from the top of the file seems to be executed, in full, before the next.

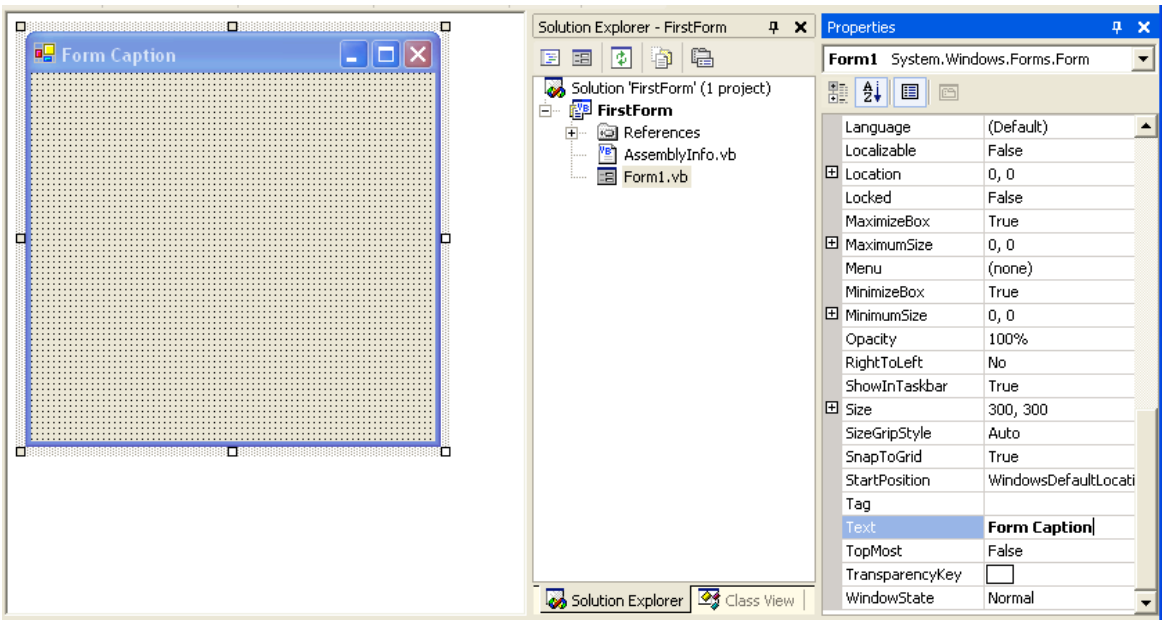
Now we've been doing awfully simplistic things in our code to this point, but that doesn't absolve us of the requirement to comment our code. Commenting means to slap in some explanations, in plain English or whatever language works best for you, right next to the code you've written. As long as the comments are kept up to date, this will make your code much easier to read for the next person using it, and will stop your application from becoming a cyborg – computer code that nobody can understand or run without having a specific human joined to it.

To add a comment, start a line with a single quote/apostrophe. Then you can type whatever you what afterwards. You can also add a comment after a line of code by adding the apostrophe.

Note: Just for kicks, you can also use the letters "REM" to mean the same thing as the single quote. You shouldn't, mind you, but that's there so that you can cut & paste some really old BASIC code into your VB.NET application and still expect it to work, without changes!

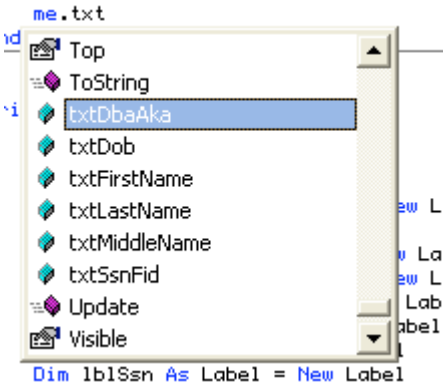
```
' The following subroutinue writes out super-secret messages to the console
' wherever the form is clicked.
REM Never, ever, start a comment with REM.
' This is called a "flowerbox" comment, and is often used at the beginning
' of a sub or function to explain what the sub does.  It's often quicker
' to read a quick comment than to scan down through the sub's code, line by
' line.
Public Sub handRolled(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Click
    Dim i As Integer

    For i = 1 To 1000
        Console.WriteLine("The super secret number is NOT: " & i)
    Next
End Sub
```

The last thing we want to look at are properties. Click on the form in the VS.NET RAD. Look over at your Properties explorer on the right side of your screen (if you used the default profile settings described previously). Properties will allow you to change seemingly any number of values for a control, from size to location to color to… For now, find the "Text" property and change it to something more descriptive than "Form1". Also head over to the "Name" property and change it to "FrmTest".



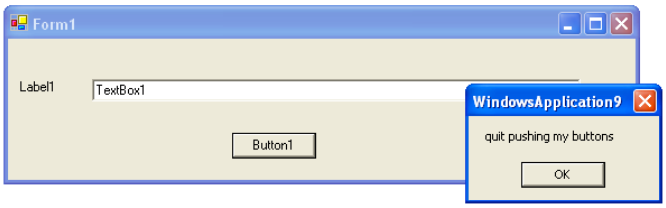
We'll prefix the name all of our Forms with "frm". This helps a coder know what kind of object a variable represents ("frmFoo" should always be a Form, etc). This naming convention is called "Hungarian Notation" (a variation on theme directly specifically for .NET is here). It's obviously not necessary that you use this naming convention – in fact many people greatly dislike it – but you'll likely want to adopt some namely convention to ensure that your programmers are using not only the same language, but also the same dialect.

With IDEs with autocomplete, Hungarian's a big help, and also helps you quickly view what sorts of child objects something you're messing with has by grouping them alphabetically. It's a useful commenting scheme, and those that say that comments lie and shouldn't be used are missing the mark. Sure, there's no machine readable link between comments and code, but that's no excuse to be lazy. Let's pretend the IDE wouldn't build your app if comments were out of sync -- a good coder will go through the same amount of work with or without the extra check.

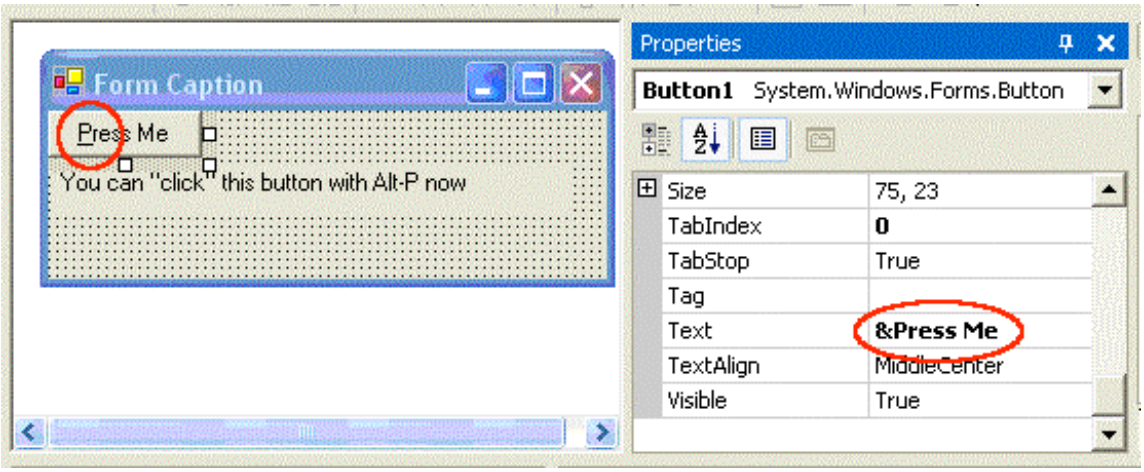


Three Controls

A form that opens message boxes is nice, but is none too useful. We'll now learn three of the most commonly used controls for the sorts of data entry screens we'll be creating shortly. Below are the Control names with its Hungarian prefix, a few important properties and other hints below them.



- Label
  - Prefix: "lbl"
  - Text Property
- Textbox
  - Prefix: "txt"
  - Text Property
  - SelectAll Sub
- Buttons.
  - Prefix: "cmd"
  - Text Property – one special trait with buttons is that you can code a "hotkey" or "accelerator key" in the text of your button by prefixing the letter with an ampersand (&). Alt-[hotkey] will essentially click the button in your application, if it doesn't conflict with other buttons or menus, etc. To insert an ampersand in your string, enter it twice, eg, "Chutes && Ladders".
  - AcceptButton – Property on a Form – user presses "ENTER" and it gets clicked, if appropriate (so no if you're in a multiline RichTextField)
  - CancelButton – Property on a Form – user presses "ESC" and it gets clicked, if appropriate



We touched on keyboard transversal in your applications above with button hotkeys. An example of how this might work is shown above.

There are two main reasons you should always have keyboard transversal in mind when making an application. First, having every function available without using a mouse speeds up data entry as an expert user won't have to use their mouse to use your application. Secondly, having fully-functioning keyboard transversal makes an application more accessible to people who may be able to operate a keyboard, but do not have the ability to use a mouse easily.

Along those same lines, we have the TabIndex property on controls. If you create a form, add two textboxes and a button, and hit F5, you'll notice that you can tab between the three Controls. If you go back and add a third textbox, you'll like notice that you'll now tab from textbox1 to textbox2 to button1 to textbox3, which is likely not what you wanted. To ensure a user can, for example, tab from entering a first name to a last name to a phone number before getting to the submit button, you need to make sure your TabIndex is in the order they'll expect. Note that some items that can't receive focus (like labels) can still be assigned TabIndexes. You can safely ignore these Controls when assigning tabs by hand; VS.NET will assign them whatever's left when you're done. Also remember that the first TabIndex is actually 0, then 1, 2, 3…

Exercise:

Create a form that does the following:

1. Has empty textboxes to hold a first name, a last name, and an email address.
2. Has labels explaining which field should be entered into which textbox.
3. Has a command button that clearly says that it submits the form.
4. Has a hotkey accelerator on the button to increase accessibility.
5. Has each Control named according to Hungarian notation guidelines.
6. Creates a message box whenever the submit button is clicked or otherwise invoked that alerts the user that their values have been submitted.
7. Extra Credit: Make your submit button the default button, submitted when enter is pressed.

Variables

Introduce variables, dimensioning them, and Hungarian notation/prefix for each.

- String – from MSDN help: "String variables are stored as sequences of unsigned 16-bit (2-byte) numbers ranging in value from 0 through 65535. Each number represents a single Unicode character. A string can contain up to approximately 2 billion (2 ^ 31) Unicode characters."
- Integer – "any of the natural numbers, the negatives of these numbers, or zero", in
- this case, "limited" to the range -2,147,483,648 through 2,147,483,647
- Long – a **very** large integer, -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
- Double – can have a decimal fraction
- Boolean – a "flag" field. Can be True or False. From MSDN: "When Boolean values are converted to numeric types, False becomes 0 and True becomes -1."
- ANY object – We've gone over a few literals at this point. Note that **any** object can be stuck into a variable. We can dimension a space for a Form, slap in a new Form created out of thin air, and start manipulating it like any other form.

```
Dim strFoo As String = "Foo"
Dim intFoo As Integer = 8
Dim lngFoo As Long = 34442020233
Dim dblFoo As Double = 4.531
Dim bFoo As Boolean = False

Dim frmTemp As Form = New Form

frmTemp.Show()
```

Functions and Subroutines

Any object, including our Forms and UserControls, can hold bits of procedural code called, generically, *methods*. There are two types of methods, broadly speaking, in VB.NET. There is the *subroutine* that executes without giving any value back to the code that called the subroutine and the *function* that, by definition, has to have a return value. Luckily, unlike VB6, a function can easily return pretty much any object we want, including custom objects we've written ourselves.

One thing you always want to avoid is creating *spaghetti code*, or code that changes variables that are out of scope. It's possible to put a number of variables in a Module and reference them in your subroutines as if they were locally dimensioned. That's bad news.

Say you've got a Module called Module1:

```
Module Module1
   Public strSmack As String
End Module
```

You could then have a subroutine like the one below.

```
' It's fun to eat, but not to see in your code.
Public Sub italianCooking()
    ' blah blah blah
    strSmack = "smack"
    ' blah blah blah
End Sub
```

Problem is, if the "blah blah blah" portions of the code are large enough, it'll get right difficult to see that you'd changed strSmack in Module1. And even if someone does catch the line, out of context strSmack looks like a local variable. That's awful. It's not, and code that uses strSmack elsewhere will have a hard time noticing all the places where this spaghetti code lives.

Instead, you'll want to have a Function that returns a String, perhaps, like this:

```
Public Function happyFunction() As String
    Dim strReturn As String

    ' blah blah blah
    strReturn = "smack"
    ' blah blah blah

    Return strReturn
End Function
```

… and you'll explicitly put that number into another variable – which will be preceded by its module's name, if you have to use a module.

```
Module1.strSmack = Me.happyFunction()
```

If you need essentially more than one return value from a Function, *don't* use spaghetti logic. Create a custom object to return from your function instead. Function's called, return value's stored, and there's one logical place in the code for the changes to take effect.

Logical Control: Branches and Loops

We're going to need to know at least four different logical control devices to program our VB.NET apps. The first is a simple If… Then logical branch. Then we'll look at two loops, one of fixed length (For… Next) and one of indefinite duration (While… End While). Finally we'll look at the Select… Case branch, which is like a multifaceted If statement.

If… Then

If… Then statements evaluate a simple Boolean and then transfer the flow of the logic down one of two branches. This can be a "null" branch, where if the Boolean doesn't evaluates to true we do nothing. Or we can set up an "Else" branch to handle the second fork of the logic. The Else branch can also itself be a fork based on Boolean logic by using an ElseIf statement. Examples are below.

If with null branch

```
If 2 = 1 Then
   MsgBox("Never get here")
End If
```

If with else branch

```
If 2 = 1 Then
   MsgBox("Never get here")
Else
   MsgBox("Will always go here.")
End If
```

If with else branch that uses ElseIf to branch again.

```
If 2 = 1 Then
   MsgBox("Never get here")
ElseIf 2 > 1 Then
   MsgBox("Looks true to me.")
Else
   MsgBox("Still no good.")
End If
```

Again, just think of an If… Then statement as a branch in logic. Two roads diverged in a wood and all that jive.

Select Case

If you want to branch in several potential directions at once and not just two like with an If… Then. If so, you'll want to use a Select Case statement. Below, we've got a variable that's a special Enum type, and we're goingt to set a String variable to a different value depending on the server type we've got currently.

Two things to pay attention to:

1.) You can have more than one type in each Case line in case two branches should do the same thing, as in the next to last Case, below.
2.) You should usually use a Case Else branch in case something you didn't expect to happen happens. With the Else branch below, for instance, the code will raise an error if an unhandled server type is in the currentServerType variable.

```
Dim strForShow As String = ""

Select Case Me.currentServerType
   Case SERVER_TYPES.FORSYTH_LIVE
      strForShow = "Live"
   Case SERVER_TYPES.FORSYTH_TEST
      strForShow = "test"
   Case SERVER_TYPES.SMITHERS_ORACLE_EDITABLE
      strForShow = "Editable"
   Case SERVER_TYPES.SMITHERS_SQL, SERVER_TYPES.LOCAL_SQL
      strForShow = "Sql Server"
   Case Else
      strForShow = "Type not found."
        Throw New Exception("Unknown Server Type in Controller.exampleSelect")
End Select
```

Sometimes we won't want to branch, quite the opposite. Instead, we'll want to do the exact same thing, give or take, over and over again. For this, we'll use loops.

While… End While

The While… End While loop is likely going to be the most popular, and will continue executing until the Boolean statement in the opening While line is true. This means that you'll never want to see a loop written like the following:

```
While 1 = 1
   Console.WriteLine("I'm stuck.")
End While
```

One will always equal one and, ss we discussed before, if something like this happens in your code, you're going to need to force-quit your application using the stop button inside of VS.NET.

For Loops

The oldest of the For loops is the For with a loop counter. For these, you need to declare an Integer or, if you're counting obscenely large numbers, a Long which will be used in the loop's declaration. You'll use For loops with counters will you know beforehand how many times you need to loop through something.

You can start with any number and loop through to any other number, give or take. Using the "Step" keyword, you can step by a number other than positive 1.

```
Public Sub exampleForLoop()
    Dim lngSum As Long
    Dim i As Integer

    lngSum = 0
    For i = 1 To 6
        Console.WriteLine(lngSum & " :: " & i)
        lngSum += i
    Next
    Console.WriteLine(lngSum)

    lngSum = 0
    For i = -10 To 6
        Console.WriteLine(lngSum & " :: " & i)
        lngSum += i
    Next
    Console.WriteLine(lngSum)

    lngSum = 0
    For i = -10 To 6 Step 2
        Console.WriteLine(lngSum & " :: " & i)
        lngSum += i
    Next
    Console.WriteLine(lngSum)

    lngSum = 0
    For i = 6 To -10 Step -2
        Console.WriteLine(lngSum & " :: " & i)
        lngSum += i
    Next
    Console.WriteLine(lngSum)
End Sub
```

For Each loops are quite useful for iterating through collections where you might not know beforehand exactly how many objects are in the collection. This is useful, for example, when iterating through a generic DataSet.

```
Public Sub loopThroughDataSetSmarter(ByVal dsIn As DataSet)
    Dim dt As DataTable
    Dim dr As DataRow
    Dim dc As DataColumn

    ' Write out every col's value from every row in every table
    ' in the DataSet.
    For Each dt In dsIn.Tables
        For Each dr In dt.Rows
            For Each dc In dt.Columns
                If Not IsDBNull(dr.Item(dc)) Then
                    Console.WriteLine(dc.ColumnName & ": " & dr.Item(dc))
                Else
                    Console.WriteLine(dc.ColumnName & ": NULL")
                End If
            Next
        Next
    Next
End Sub
```

What the For Each… Next loop does is check the collection for items of the same type as the temporary variable you're passing in in the first line. So the line…

```
For Each dt In dsIn.Tables
```

… means to deal with each DataTable in the Tables collection the same way in the loop. You would not, of course, say...

```
Dim strTemp As String
For Each strTemp In dsIn.Tables
```

… as there are no Strings in the Tables collection. And though you could use Objects for any of the temporary variables, as most every collection evaluates to some sort of Object, this is a bit too loose a cast and will lose some specificity for your code. You'll also need to cast the Object to something new if you want to use Intellisense to work with the loop items.

You could use the For… Next loop with a loop counter if you wanted to, even without knowing how many of each object is in each collection. It'd be an unecessary pain in many instances, however. Take a look.

```
Public Sub loopThroughDataSetLong(ByVal dsIn As DataSet)
    Dim dt As DataTable
    Dim dr As DataRow
    Dim dc As DataColumn

    Dim i As Integer
    Dim j As Integer
    Dim k As Integer

    ' Write out every col's value from every row in every table
    ' in the DataSet.
    For i = 0 To dsIn.Tables.Count - 1
        dt = dsIn.Tables(i)

        For j = 0 To dt.Rows.Count - 1
            dr = dt.Rows(j)

            For k = 0 To dt.Columns.Count - 1

                dc = dt.Columns(k)
                If Not IsDBNull(dr.Item(dc)) Then
                    Console.WriteLine(dc.ColumnName & ": " & dr.Item(dc))
                Else
                    Console.WriteLine(dc.ColumnName & ": NULL")
                End If
            Next
        Next
    Next
End Sub
```

Not only do you have to use the counter to fill in what should be in dt, dc, and dr, but you also have the real possibility of throwing an "off by one" (and perhaps also a fencepost?) error, as you have to remember both that the collection is zero-based (starts with 0, not 1) and that the Count is one more than the highest index in the collection (a collection with a count of 7 has items 0 through 6). Ick! Use the For Each when appropriate!

Error Handling: Try, Catch, Finally

The final thing to consider before hacking too much is error handling. You'll get errors in your applications as you write. Some you can't do anything about if you've already shipped your application. If you deploy the infinite While loop code we looked at above, you're toast. Your user will have to kill your application from the Task Manager and bug you for a new, fixed version.

Some errors, however, can be handled from within the application without the end user finding out – or can be handled in a way that lets the end user know exactly what's happening so that they can deliver a useful, intelligent error report. To handle these errors at run time, we use error trapping and the Try… Catch… Finally statements.

The way the statements are used are almost self-explanatory. Whatever you put after the Try keyword your application will attempt to execute. If an error is "thrown", the Catch block will, naturally enough, catch that error – called an Exception – and deal with it right there. The Finally clause, if you add one, will be executed after the Try block regardless of it the Catch block caught an exception or not.

```
Public Sub throwErrors()
    Dim intTemp As Double
    Dim dsTemp As DataSet
    Dim objTemp As Object

    Try
        dsTemp = New DataSet
        objTemp = dsTemp
        intTemp = objTemp
    Catch ex As Exception
        MsgBox(ex.Message)
    Finally
        MsgBox("will always happen")
    End Try
End Sub
```

Usually you'd use Finally to do some garbage collection, disposing of objects that you may no longer need once the block of code has finished executing.

String Manipulation

When you're creating data entry screens, you're going to be dealing with a lot of text. Textboxes, Comboboxes, and RichTextFields, to name a few, all return a String in their Text property no matter whether they're returning date fields, phone numbers, emails, or donation amounts.

To deal with strings appropriately, there are a number of functions you can use, all from the Microsoft.VisualBasic.Strings namespace.

(see StringFunctions project)

```
Left("The quick brown fox JUMPED over the LAZY dog.",4)              The
Right("The quick brown fox JUMPED over the LAZY dog.",4)             dog.
Mid("The quick brown fox JUMPED over the LAZY dog.",4, 5)             quic
Len("The quick brown fox JUMPED over the LAZY dog.")                 45
```

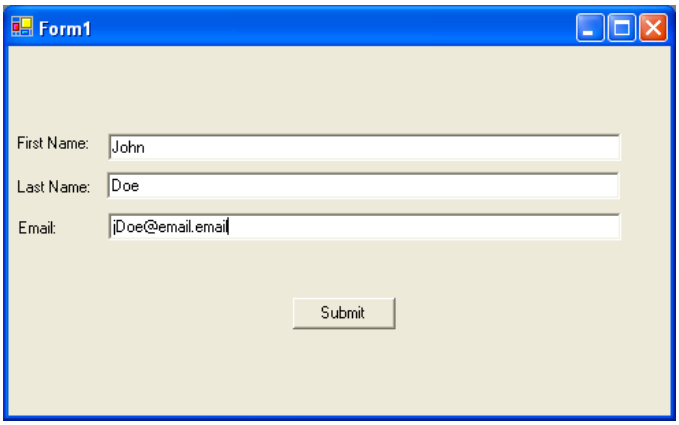
```
InStr("The quick brown fox JUMPED over the LAZY dog.", "the")        33
"The quick brown fox JUMPED over the LAZY dog.".indexOf("the")       32
"The quick brown fox JUMPED over the LAZY dog.".LastIndexOf("the")   32

UCase("The quick brown fox JUMPED over the LAZY dog.")               THE QUICK BROWN
FOX JUMPED OVER THE LAZY DOG.
LCase("The quick brown fox JUMPED over the LAZY dog.")               the quick brown
fox jumped over the lazy dog.

IsNumeric("The quick brown fox JUMPED over the LAZY dog.")           False
IsDate("The quick brown fox JUMPED over the LAZY dog.")              False
```

And later… isDbNull, .Equals()

Quotes in strings – “” So to get “smack”, you’d say strFoo = “””smack”””

String concatenation
strFoo = strFoo & strBar & intSmack
strFoo += strBar

Simple Data Entry Validation



With these simple text manipulation functions, you can do pretty much anything you need to do to ensure your user is entering valid values in your data entry screen. Just as we set the Text property in the VS.NET Properties window, we can pull it right back out at run-time the same way.

```
Dim strTemp As String
strTemp = Me.txtFirstName.Text
```

Note: Run-time vs. design-time

Excercises:

1. Create a form with three textboxes with labels for first & last name and an email address (or reuse from earlier exercise)
2. On click, have submit button event handler check to ensure the first character (and only the first) is upper case in the first & last name fields.
3. On submit, also ensure that the name entered is your own first and last name, or don’t allow the user to continue.

Regular Expression Intro

Regular Expressions – quick introduction.
http://etext.lib.virginia.edu/helpsheets/regex.html

> *Regular expressions trace back to the work of an American mathematician by the name of Stephen Kleene (one of the most influential figures in the development of theoretical computer science) who developed regular expressions as a notation for describing what he called "the algebra of regular sets." His work eventually found its way into some early efforts with computational search algorithms, and from there to some of the earliest text-manipulation tools on the Unix platform (including ed and grep). In the context of computer searches, the "*" is formally known as a "Kleene star."*

When you're dealing with particularly convoluted text manipulation, regular expressions are likely the best way to go. It's been said that regular expressions are much much easier to write than to read and understand, which is its disadvantage when compared to the typical String functions we dealt with earlier. But when you've got a regular expression that works, it makes your code much more compact and efficient, and writing it is much easier than using the above functions once you've gotten over the initial regular expressions learning curve.

Excercises:
Use regular expressions to check the value a user enters for the email textbox and add that check to cmdSubmit's check for valid data entry/edits.

If you have to cheat…

```
Public Function validateEmail(ByVal strIn As String) As Boolean
    Return Regex.IsMatch(strIn, ("^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-
9]{1,3})(\]?)$"))
End Function
```

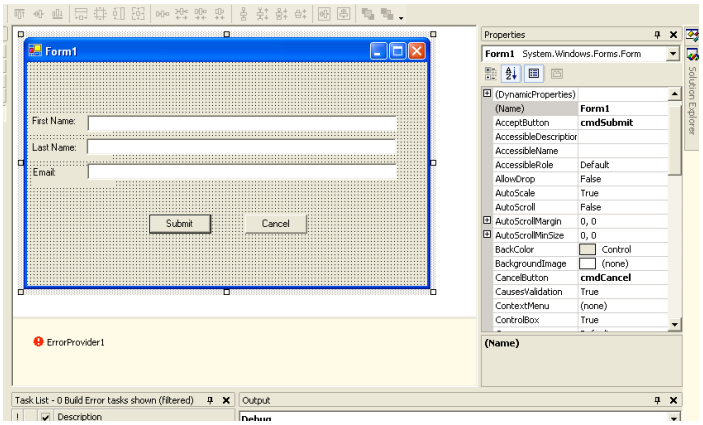
Extra credit: add a birthdate and ensure that it's in mm/dd/yyyy format, using whatever method you prefer.

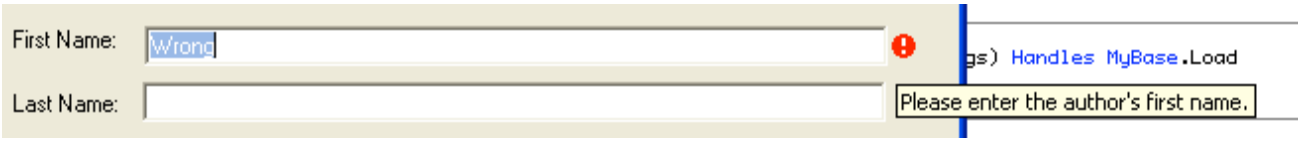
The Validation Event in Controls

In VB6, many coders checked textbox values as we did earlier – on submit, but not before. As you could tell just with the simple three textbox form we had then, the errors can really pile up.

Luckily .NET provides us with the ErrorProvider class of objects and a special event called Validating. You can create your own validation event for a control by hand or by clicking it from the combo boxes at the top of VS.NET.

```
Private Sub txtFirstName_Validating(ByVal sender As Object, _
   ByVal e As System.ComponentModel.CancelEventArgs) Handles txtFirstName.Validating

   If Not ActiveControl Is cmdCancel Then

      With Me.txtFirstName
         If Not .Text.Equals("Ruffin") Then
            e.Cancel = True
            Me.ErrorProvider1.SetError(sender, _
               "Please enter the author's first name.")
            .Focus()
            .SelectAll()
         Else
            ErrorProvider1.SetError(sender, "")
            ' clear any previous error
         End If
      End With
   Else
         e.Cancel = False    ' this line isn't actually needed,
         ' but is here to show that if you hit the cancel button,
         ' the QA check is ignored/bypassed.
   End If
End Sub
```





Adding a Cancel button – more than just setting the CancelButton property.

Dealing with Numbers

Because users will be entering and editing numbers in Textboxes, we'll be doing quite a bit of String to numeric conversions. If someone's entering a dollar amount paid, there's nothing about the textbox by itself that stops them from entering "Fred" as a value. And if we take a String that has the value "31.12", we're not going to get anywhere if we try to subtract that String from a Double with a value of 100. We have to ensure the String is a number, and, if it checks out, move that value into a numeric type variable.

To check if a String evaluates to a number, use the isNumeric function.

```
If IsNumeric(strNumber) Then…
```

We've looked at the TypeOf statement briefly, and will now move on to its cousin, the CType statement. With CType, we can "cast" one type of variable into another. CType can parse a number in a String, if there's one to be found, in code similar to the following:

```
Public Sub castingNumbers()
   Dim strTest As String
   Dim dblTest As Double
   Dim intTest As Integer
```

```
    strTest = "32.12"

    dblTest = CType(strTest, Double)
    Console.WriteLine(dblTest)  ' should hold 32.12
    intTest = CType(strTest, Integer)
    Console.WriteLine(intTest)  ' should round to nearest Integer, so 32
End Sub
```

Note that CType works for anything, as in the following when we shove a Form instance into a generic object and pull it back out into another form.

```
Public Sub formToObjectToForm()
    Dim frm1 As Form
    Dim frm2 As Form
    Dim obj1 As Object

    frm1 = New Form
    frm1.Text = "Hand-rolled Form"

    obj1 = frm1

    frm2 = obj1              ' these two lines
    frm2 = CType(obj1, Form)    ' evaluate to the same thing

End Sub
```

Note that there are also a number of convenience conversion functions for numbers, mostly to preserve compatibility/familiarity with VB6. These make your code a bit faster to type and arguably a little easier to follow.

```
    Dim strNumber As String

    Dim dblTemp As Double
    Dim intTemp As Integer
    Dim lngTemp As Long

    strNumber = "-123.321"

    If IsNumeric(strNumber) Then
        dblTemp = CDbl(strNumber)
        intTemp = CInt(strNumber)
        lngTemp = CLng(strNumber)
    End If
```

<,>,=,+,-,*,/
\,MOD – integer division

Add a “Amount Paid” textbox. Check for accidental “$”’s. Ensure the correct number of digits is to the left of the point. If no point, insert “.00” at the end. Alert if less than $50.

Day 3

DataSets

Getting data from DataSets
-- Strongly typed DataSets intro (likely won't use just yet)
Create by hand
Create from XML file

```
Dim strPath As String
Dim dsTest As DataSet

dsTest = New DataSet
strPath = System.AppDomain.CurrentDomain.BaseDirectory() & "testOneEntity.xml"
Console.WriteLine(strPath)
dsTest.ReadXml(strPath)
```

In the final version, we'll create from a Controller, but from your class's point of view, it doesn't matter.

Iterate through a dataset, table by table, row by row, column by column.

```
Public Sub loopThroughDataSetSmarter(ByVal dsIn As DataSet)
    Dim dt As DataTable
    Dim dr As DataRow
    Dim dc As DataColumn

    ' Write out every col's value from every row in every table
    ' in the DataSet.
    For Each dt In dsIn.Tables
        For Each dr In dt.Rows
            For Each dc In dt.Columns
                If Not IsDBNull(dr.Item(dc)) Then
                    Console.WriteLine(dc.ColumnName & ": " & dr.Item(dc))
                Else
                    Console.WriteLine(dc.ColumnName & ": NULL")
                End If
            Next
        Next
    Next
```

Create a class that:

1. Has a null constructor
2. Takes in a DataSet in a constructor
3. Has a Public Sub to add/replace the running DataSet
4. Validates the data in the DataSet and returns a Boolean

Before leaving, make sure you have your XML file for your entry screen, and test that it's all there on your workstation. Alternately, depending on how the new database looks, we'll just use a test XML file for Taxable Owner.

For "homework" tomorrow morning, create a DataSet validation class for your DataSet like the one we created today for names and emails.

Questions from trying to create class from Tuesday.
Spillover from 1-2.

Combining DataSet validation with GUI validation

Objects and Object Oriented Programming

Modules
Classes
Constructors
Interfaces
Extending & inheritance

More Controls and Advanced Control Usage

Comboboxes
Checkboxes
Radio buttons
Datagrids
Panels (Controls array)
Menus (and why we're not using them yet)
Context Menus (creating at run time)

Anchor Property
GridLayout

Preparing to Create Data Entry Screens

Day 4

Morning: Entry screen work in offices.

Overview of new TAXSYS system architecture

Diagram from Visio of tiered system.

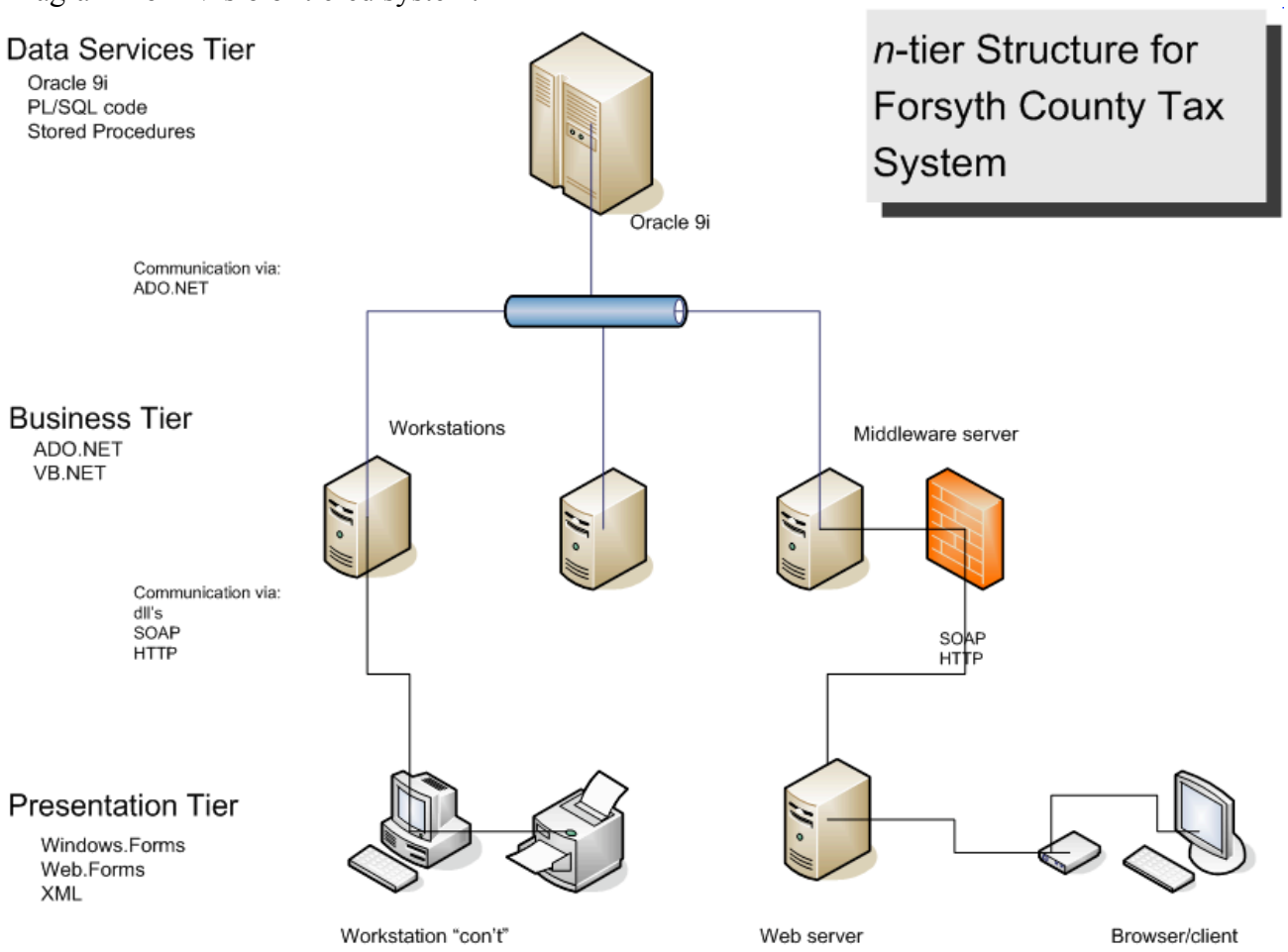
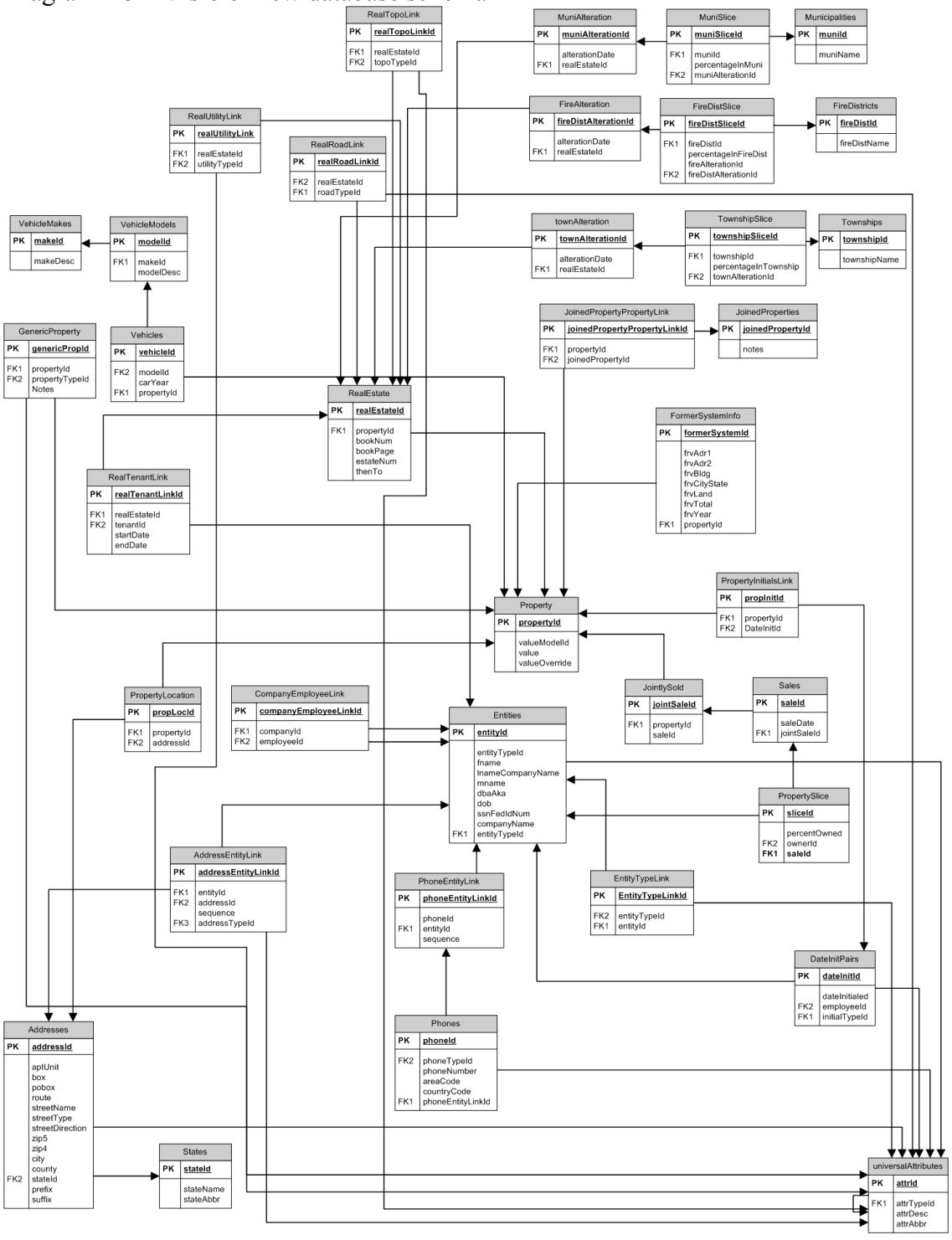


Diagram from Visio of new database schema



How creating screens as we've described will fit in with the system.
.NET Namespaces

Steps for creating an entry screen:

1. DataPanels
2. BusinessTier
3. Make a UserControl.
4. Implement IEntryScreen
5. Create your GUI
6. Add to a test form.
7. Have test form create dataset from XML file to pass to UserControl.
8. Ensure that the form has validation routines on the GUI and DataSet.
9. Test it.
10. Fix it. Goto 9.

Hooking New Screens Into the New System

Note that this will be done, initially, by Rok. After we've got a number of screens up and running in the Shell and the Shell tested out fairly well, we'll push out that code as well.

Introduction to reporting

Crystal Reports
Xsd files

From Powerpoint presentation for 3/1

Design Guidelines/Methodology

- Design using **modular** interfaces
- **Refactor** when updating code
- **Document** code
- Use **Source Control**

Modular

- Single most important design guideline
- Allows for manageable system testing
  - Test only one piece of functionality
  - Contain bugs
- Keeps individual programming tasks simpler
  - No need to understand entire application
  - Code to an API, not an intertwined mess
- Easy part substitution
  - SQL Server, PostreSQL,etc for Oracle
  - ASP.NET/Web.Forms for Windows.Forms

Refactor

> "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring."
>
> **-- Refactoring Home Page**

- Never completely replace code – and the lessons it contains
- Works with modularity – update one small function at a time
- Reduce risk of major issues in new versions of code

Documentation

- During Design
  - System tables & fields identified
  - Cleanly defined purpose for each module of code
  - Unit/functional testing instructions, where appropriate
- In Code
  - Flowerboxes

- APIs

Source Control

- Keeps track of all changes
- Eliminates issues with two coders working on the same code
- Serves as an extra layer of backup
- Allows easy reversion to old code when necessary
- Note: Only check out the files you're explicitly working on
- Using Visual SourceSafe and VssConnect

Interfaces

- Contracts for objects
- IReportScreen

```
Imports BusinessTier.Controller

Public Interface IReportScreen

    Event queryExecuted(ByVal strWhere As String, ByVal intScreenId As SCREEN_TYPES)

    ' Reports can come in a number of ways, from html to a DataSet to XML, etc.
    ' This sub should marshall plain object types to the correct overloaded implementation.
    ' Its existence here is to remind coders that we need an event to raise when a query's
    ' been run by the user and an event to handle the returned contents.
    Sub relayReportContents(ByVal objReportContents As Object)
End Interface
```

- IUiScreen

```
Imports BusinessTier.Controller

Public Interface IUiScreen
    Event updateDataset(ByVal dsEdited As DataSet, ByVal intScreenId As SCREEN_TYPES)
End Interface
```