Speaker Name: dave espinosa-aguilar

Course: Integrating Microsoft Excel and Access with AutoCAD VBA

Course Description: For years AutoCAD users have been trying to learn how to integrate the power of Excel spreadsheets and Access databases with AutoCAD drawings. This course introduces intermediate and advanced AutoCAD users to the programming concepts required to link and maintain the integration of these three products through Visual Basic. Examples of code to generate bills of materials, block counts and facilities management applications will be discussed in depth.
Hi, my name is Dave Espinosa-Aguilar. I’ve been using and training folks on AutoCAD since the mid-1980’s, I visit about 200 companies a year, and I get to see how people throughout the country use AutoCAD in powerful and clever ways. It’s an interesting fact that most AutoCAD users have Microsoft Office installed on their systems these days, and applications like Microsoft Excel and Microsoft Access lend tremendous resources to extending the capabilities of AutoCAD, especially now that the AutoCAD’s VBA interface is “fully cooked.”

This class is not an introduction to AutoCAD, Excel, Access or VBA. Several classes at this year's Autodesk University treat the subject of introductory VBA programming in AutoCAD, and this course makes no attempt to accomplish that in addition to discussing all the concepts involved in integrating AutoCAD, Excel and Access. The sole purpose of this course is to build on existing knowledge of these applications and interfaces so that AutoCAD can exploit Microsoft Office spreadsheet and database functionality. These notes may fast become an AutoCAD VBA “bible” on this subject.

What does Microsoft Excel Offer an AutoCAD User?

If you’ve ever done a manual count of blocks or block attribute values, if you’ve ever had to generate a manual schedule of drawing components or a report of entity counts in a CAD drawing, you might already be able to appreciate the benefit of a spreadsheet application which can automate this process for you. Through AutoCAD’s programming interfaces, tallies of AutoCAD entities and their values can be generated quickly to save time and make valuable information available to other applications.

Consider the typical strengths of a spreadsheet: the ability to organize data, the ability to process data, the ability to query data and treat ranges of data, the ability to work with data types and best of all, pass that information in a grid format to other applications. This class is all about how to bring the power of a spreadsheet application right into AutoCAD drawings. Let's examine typical limitations of AutoCAD TEXT entities and how a spreadsheet application can circumvent them. Consider the figure to the left. Suppose this small schedule was drawn strictly using AutoCAD entities (TEXT and LINE entities) to report the counts for 5 different block types named ITEM1 through ITEM5.

1. These numbers would have to be initially generated by manual counts since there are no functions in vanilla AutoCAD which will put together custom schedules like this.
2. If new blocks are added to the drawing, the schedule would have to be updated manually since there is no relationship between the TEXT entities and the blocks.
3. To get a total for all items, you would have to do a manual calculation (you can't add TEXT entities in AutoCAD). Maybe adding up five numbers isn't a problem, but imagine adding up 20, or 100, or 1000 items. Imagine trying to keep track of hundreds of item types as any design size increases.

In short, by using VBA with a spreadsheet application like Excel, not only can these counts be automated as new items get added to the drawing, but the counts are fast, accurate, and a total can be automatically generated. Add to these benefits the fact that it is easier to stretch the size of an OLE-pasted grid object than it is to redefine text styles, the fact that it is easier to insert a new row or new column to a grid object than it is to move existing text entities around to accommodate new rows and columns, and add the fact that this schedule information can be shared with other applications like word processors and databases--- and you have some pretty convincing arguments for examining the integration of Microsoft Excel with AutoCAD.
And VBA makes this all possible so that you can easily generate bill of materials, schedules, reports, and share all this information with other Windows applications. Not only can information from AutoCAD be tallied, but it can also be numerically processed (sums, averages, minimums maximums, etc) And since Excel is OLE-capable, you can drop-and-drag your integrated worksheets right back into AutoCAD as stretchable, linked (aka re-editable), evenly-spaced plot-able entities unto themselves.

**What does Microsoft Access Offer an AutoCAD User?**
The information in many types of CAD drawings essentially reduces to a vast database of numeric, financial, and textual information. Consider for example a drawing which might have thousands of point entities reflecting survey location and elevation data, or a landscaping plan filled with hundreds of plant types that each have field data associated with them (plant type, age of plant, water requirement, plant nursery source, plant cost, etc), or an architectural design which has cost and manufacturer information associated with materials, parts and details. In many cases, much of this information is non-associated with CAD entities and it can be a time-consuming (read expensive) process not only keeping tabs on all of it, but working with it.

There are also many entities in AutoCAD which can hold data (such as Block attributes) but which are extremely limited in their capacity to be updated, queried, processed, exported to other data formats and shared with other Windows applications. The figure to the left shows six insertions of a block with two attributes. Imagine that your design has hundreds of blocks like this in it, and hundreds of other block definitions in it as well, each with a unique set of attribute values. There are some limited ways to "pull" information from these blocks and export it to a text file (using templates with the ATTEXP command), but:

1. You could not add up the total number of blocks that have "chair" for their first attribute value since all six of these blocks are named the same.
2. You could not add up the total (nor take the average value, nor find the minimum or maximum value) of the second attribute value in those blocks which have "seat" for their first attribute value.
3. You could not replace the values of the second attribute of those blocks which have "chair" for their first attribute value.
4. You could not create a selection set or change the properties of those blocks based on an SQL-type query for a visual understanding of the data.

And that's a huge benefit of databases: that they can be queried numerically, financially, by date, by string matches, etc. If the values you are storing in block attributes need to be examined, processed, tallied, manipulated, etc, you have few if any options within AutoCAD to do this, and these are just some of the typical limitations in AutoCAD that a database can overcome.

*Note: DBCONNECT, ASE and other database technologies internal to AutoCAD do make it possible to not only link AutoCAD entities with databases, but to query them, select them, change them and tally them etc.*

But in all fairness (as anyone who has ever used them will tell you), these tools are severely limited in their capacity to analyze entities, generate reports, import/export data and mass-process field values internal to AutoCAD entities. The required linking interface alone (manually selecting entities to link them with database records) can be a prohibitive process. VBA makes it possible for database records to be linked with AutoCAD entities based on their existing properties and attribute values.
Yet another extremely beneficial use of external databases with AutoCAD is that they keep AutoCAD sessions fast. When you rely on database information to be stored in AutoCAD, it slows AutoCAD down (the bigger the drawing, the slower it processes). By externalizing data and linking it with entities in AutoCAD, you not only gain full database functionality over that information, but you significantly reduce the drawing size. For example, if you had a series of blocks that represented tree types, each block containing up to 20 fields of unique data, and you needed to change the values of certain fields for certain types of trees, you would have to manually edit the data in blocks (ever change over 20 attribute values in a block manually?) which is time consuming and downright mundane. But if this data were in a database, it would be easy to create an index of those records which met your criteria and change their field values in seconds. By each CAD tree symbol being linked to a record rather than containing data within itself, you gain database functionality over your CAD drawing entities and the information associated with them. You also wind up with faster CAD drawings to work with.

Finally, like spreadsheets, databases understand data types. They know the difference between a numeric (integer, real, etc) value and a dollar value and a textual value and a date value and they know how to store these value types efficiently. Block attributes see every data value as a textual value no matter what it represents to you, and this is why block attributes can’t be numerically processed or queried effectively. When you explode a block with attribute values (using Express Tools or other tools which retain the attribute values), you wind up with TEXT entities which have the same limitations as discussed in the Excel example above. To streamline your storage of data and keep AutoCAD sessions fast, to gain full control over data that you associate with CAD entities, and to query and manipulate huge amounts of data and the CAD entities it is associated with, quickly, an external database is often the best alternative.

Essential Preparations for Connecting AutoCAD with Excel and Access
Obviously, to take advantage of AutoCAD’s VBA, we need to launch AutoCAD and use a command like VBAMAN or VBAIDE to bring up the VBA interface within AutoCAD. This class assumes you have Excel and Access loaded on your system. By default, AutoCAD VBA does not recognize Excel and Access objects, so you need to use the VBA Tools/References pulldown menu function and select a Microsoft Excel Object Library and/or a Microsoft DAO (Data Access Objects) Library to use Excel and/or Access Objects and their properties in your VBA code. Public variables and non-private functions are used in the code examples for this class to keep things simple.

AutoCAD VBA/Excel Coding Concepts
The following General Declarations for treating the Excel Application and workbooks and worksheets within it are assumed for the code examples provided in this course:

```vba
Public excelApp As Object
Public wkbObj As Object
Public shtObj As Object
```

When working with any Microsoft Office application such as Excel, your VBA code must first establish a link with it. You cannot assume that a session of Excel is already running, so the code has to accomplish several things: it has to detect if Excel is already running, so the code has to establish a link with it, and if Excel isn’t already running, launch a session of it. The GetObject function assumes that Excel is already running. If it isn’t, an error (non-zero condition) is generated and we use the CreateObject function to launch Excel. If Excel cannot be launched for some reason, then another error is generated and we notify the user. If Excel can be launched, we add a workbook (which has 3 default sheets to it) and set the current worksheet to the first worksheet. A function to rename the first sheet is also provided. The current workbook and worksheet pointers are also set.

```vba
Sub CommandButton1_Click()
```
On Error Resume Next
Set excelApp = GetObject(, "Excel.Application")
If Err <> 0 Then
    Err.Clear
    Set excelApp = CreateObject("Excel.Application")
    If Err <> 0 Then
        MsgBox "Could not start Excel", vbExclamation
        Exit Sub
    End If
End If
excelApp.Visible = True
Set wbkObj = excelApp.Workbooks.Add
Set shtObj = excelApp.Worksheets(1)
End Sub

To work with an existing spreadsheet file, you can use the code below to set the current workbook:
Set wbkobj = Workbooks.Open(filename:="c:\dir\filename.xls")

If the above code is assigned to a command button and the VBA code is run, pressing this button will either launch Excel and link your VBA application to it, or link it to an existing session of Excel.

The code below closes Excel, but a dialog may ask if you want to save changes to the current workbook.
Sub CommandButton2_Click()
    excelApp.Quit
End Sub

To send a value from your VBA application to a particular cell in the linked Excel application, first establish which worksheet you're sending information too and then then set the worksheet cell value by specifying the row and column using integer values. Be sure to use proper variable types when setting the values of cells. In the code below, the first worksheet is specified, and an integer, a real number and a string are passed to it. If you pull your values from textboxes and other object sources in your forms, this is especially important.

Sub CommandButton3_Click()
    Set shtObj = wbkobj.Worksheets(1)
    ival% = 1
    rval& = 1.5
    sval$ = "text"
    shtObj.Cells(1, 1).Value = ival%
    shtObj.Cells(2, 1).Value = rval&
    shtObj.Cells(3, 1).Value = sval$
End Sub

Notice that the real value 1.5 passed to cell A2 in the spreadsheet reports as a value of 2. You can set the format of any cell in the spreadsheet by formatting the data you are passing to it from your VBA application first.

The code below shows how various types of formatted values from your VBA application appear in the Excel spreadsheet once they are passed including system dates and times, system and date formats, decimal precisions, and capitalization functions:

Sub CommandButton4_Click()
    Set shtObj = wbkobj.Worksheets(1)
    Dim MyTime, MyDate, MyStr

MyTime = #5:04:23 PM#
shtObj.Cells(1, 1).Value = MyTime
MyDate = #1/27/93#
shtObj.Cells(2, 1).Value = MyDate
MyStr = Format(Time, "Long Time")
shtObj.Cells(3, 1).Value = MyStr
MyStr = Format(Date, "Long Date")
shtObj.Cells(4, 1).Value = MyStr
MyStr = Format(MyTime, "h:m:s")
shtObj.Cells(5, 1).Value = MyStr
MyStr = Format(MyTime, "hh:mm:ss AMPM")
shtObj.Cells(6, 1).Value = MyStr
MyStr = Format(MyDate, "ddd, mmm d yyyy")
shtObj.Cells(7, 1).Value = MyStr
MyStr = Format(23)
shtObj.Cells(8, 1).Value = MyStr
MyStr = Format(5459.4, "##,##0.00")
shtObj.Cells(9, 1).Value = MyStr
MyStr = Format(334.9, "###0.00")
shtObj.Cells(10, 1).Value = MyStr
MyStr = Format(5, "0.00%")
shtObj.Cells(11, 1).Value = MyStr
MyStr = Format(334.9, "$###.##")
shtObj.Cells(12, 1).Value = MyStr
MyStr = Format("HELLO", ")")
shtObj.Cells(13, 1).Value = MyStr
MyStr = Format("This is it", ")")
shtObj.Cells(14, 1).Value = MyStr
End Sub

Font name, size, bold or italics, and columnwidths can also be controlled from your VBA application:

Sub CommandButton5_Click()
    Set shtObj = wkbObj.Worksheets(1)
    shtObj.Cells(1, 1) = "Hi there."
    shtObj.Cells(1, 1).ColumnWidth = 30
    shtObj.Cells(1, 1).Font.Bold = True
    shtObj.Cells(1, 1).Font.Name = "Courier"
    shtObj.Cells(1, 1).Font.Size = 20
    shtObj.Cells(1, 1).Justify
    shtObj.Cells(2, 1) = "Goodbye."
    shtObj.Cells(2, 1).ColumnWidth = 20
    shtObj.Cells(2, 1).Font.Italic = True
    shtObj.Cells(2, 1).Font.Name = "Times Roman"
    shtObj.Cells(2, 1).Font.Size = 15
    shtObj.Cells(2, 1).Justify
    shtObj.Cells(3, 1) = "Check"
    shtObj.Cells(3, 1).ColumnWidth = 5
    shtObj.Cells(3, 1).Justify
End Sub

You can work with ranges and formulas in Excel from your VBA application as well:

Sub CommandButton6_Click()
    With shtObj.Range("A1:B2")
        .Font.Name = "Bookman Old Style"
        .Font.Size = 10
        .Font.Bold = True
        .Value = 4
    End With
    Range("A3").Formula = @"average(A1:A2)"
    Range("B3").Formula = @"sum(B1:B2)"
    Range("C3").Formula = @="A3-B3"
End Sub

You can pull information from a cell into an object in your form. The code below shows how a value in a cell can be brought into a textbox object, or how values found in a range of cells (A1:B4) can be brought into a listbox
or combobox object (notice the use of the ColumnCount property is required for proper display of cells involved in the imported range):

```vba
Sub CommandButton7_Click()
    TextBox1.Text = Worksheets("Sheet1").Range("A1").Value
    ComboBox1.ColumnCount = 2
    ComboBox1.List = Worksheets("Sheet1").Range("A1:B4").Value
End Sub
```

AutoCAD VBA/Excel Sample Application #1: Block Counter/Schedule Generator

The following code can be pasted into the General Declarations area. You can create the GUI below by inserting a UserForm with 3 command buttons and 2 listboxes. Button1 cycles through all blocks found in the drawing block collection and adds their names (except for those beginning with an asterisk) to ListBox1. It then cycles through all ListBox1 names, totaling all entities in ModelSpace which are both blocks and named the current ListBox1 value to ListBox2. Button2 sends the ListBox data to a new spreadsheet. Button3 exits the VBA application.

```vba
Public excelApp As Object
Public wkbObj As Object
Public shtObj As Object

Sub CommandButton1_Click()
    Dim i, j, btot As Integer
    Dim bnam As String
    Dim ent As Object
    btot = ThisDrawing.Blocks.Count
    For i = 0 To btot - 1
        bnam = ThisDrawing.Blocks.Item(i).Name
        If Not Mid$(bnam, 1, 1) = "*" Then ListBox1.AddItem bnam
    Next i
    For i = 0 To ListBox1.ListCount - 1
        bnam = ListBox1.List(i): btot = 0
        For j = 0 To ThisDrawing.ModelSpace.Count - 1
            Set ent = ThisDrawing.ModelSpace.Item(j)
            If ent.EntityType = acBlockReference And ent.Name = bnam Then btot = btot + 1
        Next j
        ListBox2.AddItem btot
    Next i
End Sub

Sub CommandButton2_Click()
    On Error Resume Next
    Set excelApp = GetObject(, "Excel.Application")
    If Err <> 0 Then
        Err.Clear
        Set excelApp = CreateObject("Excel.Application")
        If Err <> 0 Then
            MsgBox "Could not start Excel!", vbExclamation
        End If
    End If
    excelApp.Visible = True
    Set wkbObj = excelApp.Workbooks.Add
    Set shtObj = wkbObj.Worksheets(1)
    shtObj.Name = "Block Count"
    Dim i, j, btot As Integer
    Dim bnam As String
    j = 1
    For i = 0 To ListBox1.ListCount - 1
        bnam = ListBox1.List(i)
        btot = ListBox2.List(i)
        shtObj.Cells(j, 1).Value = bnam
        shtObj.Cells(j, 2).Value = btot
        j = j + 1
    Next i
End Sub
```
With further VBA coding, you can have complete control over which blocks get reported (such as those with particular string wildcard values for their names, blocks with certain attribute tag names or attribute values that fall within numeric ranges, etc). You can also calculate totals, averages, minimum and maximum counts of blocks and values of block attributes in Excel, once the values are passed to it or once you’ve sent formulas to certain cells which do it for you. You could even update entities in AutoCAD according to cell values. Once you’ve generated a spreadsheet report of the block counts, you can copy and paste cell ranges back into AutoCAD as a schedule report entity which can be stretched, replaced with new information, plotted/printed and manually updated.

**AutoCAD VBA/Excel Sample Application #2: Bill of Materials**

The first sample application essentially shows how AutoCAD information can be passed to a spreadsheet so that a grid-like report (Schedule) can be pasted back into AutoCAD. In this second application, entities which define areas are placed on layers with an associated cost per square inch. By pressing a single button, the user can know the overall cost of a design. Some standards are assumed. On the spreadsheet side of things, an existing spreadsheet with layer names and costs already entered is assumed: its column A is assigned string values reflecting layer names, and its column B is assigned real (dollar) values reflecting cost of material per square inch. For example, The value of cell A1 might be "MATERIAL1" and the value of cell B1 might be $1.50. On the AutoCAD side of things, only polylines are used to define areas on layers whose names match values in column A of the spreadsheet. If a new layer is defined in column A with a new cost of material in column B, area-defining entities can now be created on that new layer in AutoCAD and be added to the overall cost when the routine is used. This application could easily be modified to associate block names with associated costs (instead of area-defining entities) to drive the numbers as well.

You can create a GUI shown to the right by inserting a UserForm with 2 command buttons, 1 label and 1 textbox. The routine opens the existing spreadsheet and loops through text values found in column A until it finds a blank cell and ends. For each non-blank cell it finds in Column A, it pulls the price value in column B. It then searches the drawing database for polylines (EntityType 24) on the column A layer name and multiplies each polyline entity area property with the column B value, adding this value to a total value for that layer. Each layer total is then added to a grand total. In this way, no matter how many polylines of any shape are drawn on layers represented with cost values in the spreadsheet, the press of one button will yield the cost total for all areas according to their layer pricings. Now that’s some serious estimating power!

```vba
Public excelApp As Object
Public wkbObj As Object
Public shtObj As Object
Private Sub CommandButton1_Click()
    On Error Resume Next
    Set excelApp = CreateObject("Excel.Application")
    If Err <> 0 Then
        Err.Clear
        MsgBox "Could not start Excel!", vbExclamation
    End If
    excelApp.Visible = True
    Set wkbObj = Workbooks.Open(filename:="c:\pricing\costing.xls")
    Set shtObj = wkbObj.Worksheets("Sheet1")
```

```vba
Public excelApp As Object
Public wkbObj As Object
Public shtObj As Object
Private Sub CommandButton1_Click()
    On Error Resume Next
    Set excelApp = CreateObject("Excel.Application")
    If Err <> 0 Then
        Err.Clear
        MsgBox "Could not start Excel!", vbExclamation
    End If
    excelApp.Visible = True
    Set wkbObj = Workbooks.Open(filename:="c:\pricing\costing.xls")
    Set shtObj = wkbObj.Worksheets("Sheet1")
```
Dim i, j As Integer  
Dim pnum, anum, atot As Double  
Dim lnam, enam As String  
Dim ent As Object  
i = 1: anum1 = 0#: anum2 = 0#: atot = 0#  
lnam = shtObj.Cells(i, 1).Value  
pnum = shtObj.Cells(i, 2).Value  
Do While Not (lnam = "")  
    For j = 0 To ThisDrawing.ModelSpace.Count - 1  
        Set ent = ThisDrawing.ModelSpace.Item(j)  
        If ent.EntityType = 24 And ent.Layer = lnam Then  
            anum1 = ent.Area  
            anum2 = anum2 + anum1  
        End If  
    Next j  
    atot = atot + (anum2 * pnum)  
    anum1 = 0#: anum2 = 0#  
    i = i + 1  
    lnam = shtObj.Cells(i, 1).Value  
    pnum = shtObj.Cells(i, 2).Value  
Loop  
excelApp.Quit  
TextBox1.Text = atot  
End Sub  
Private Sub CommandButton2_Click()  
End  
End Sub  

Other Things to Keep in Mind about Working with Excel  
The Task Manager (ALT+CTRL+DEL) can be very helpful in assessing if you have sessions of Excel left open while you're programming in VBA. A common error many beginning programmers make when linking to Microsoft Office applications is to forget to close those applications in addition to exiting your own routine. If you don't use 

```
excelApp.Quit
```

(excelApp is the object assigned to the Excel Application in the above code) for example, when finishing the command button code, you will leave a session of Excel running in the background even though your VBA routine has exited. There may be times when you want to leave Excel running after your VBA routine is finished. But be aware that running the VBA routine again and again will launch session after session of Excel. If you're not sure how many sessions of Excel are open, or if Excel forbids you to edit a spreadsheet, use the Task Manager to see if Excel sessions are running and use it to close down an Excel session if Excel itself won't let you.

Also, remember that new workbooks typically bring up three sheets by default. It is a good practice to either create your own sheets before pointing to them, or renaming existing sheets so that you can specify exactly what sheet you're passing and pulling information to and from. It is not a good idea to assume that everyone's Excel sessions will bring up the same default workbooks and worksheets you have on your system: many users rely on different default spreadsheet templates.

AutoCAD VBA/Access Coding Concepts  
The following General Declarations for treating the Access Application, workspaces, databases, tables and fields within it are assumed for the code examples provided in this course:

```
Public wksObj As Object  
Public dbsObj As Object  
Public tblObj As Object  
Public fldObj As Object  
Public rstObj As Object
```
In the Excel treatment of this class, Excel was launched and potentially left visible to the user while the VBA applications were running. But there may be times when you want to exploit Excel or Access without those applications being visible during your VBA routine, so for the Access treatment of this class, Access will be linked and utilized, but not made visible to the user during execution of VBA code so that the VBA routine runs with Access 'in the background.'

With Access, you must establish several essential objects to link to databases, create and edit them and query them. Remember, this is not a class on essential concepts in Access, but rather a treatment of the integration of Access with AutoCAD. You should be completely familiar with Access models: the initial workspace (like the Excel Application) is needed to create and edit databases (.mdb files) which contain tables with multiple field types. There are several ways to work with Microsoft Access databases from VBA, and the code examples used in this class are not intended to portray "the right way" to do things. In the code examples for this course, table-type Recordsets are used to keep things simple (one table at a time, base-table directly manipulated, no ODBC, can be indexed by underlying table, etc).

Important Reminder: both Excel and Access have Online help for VBA programming if you launch them. Also, the AutoCAD VBA IDE's View/Object Browser (F2) function brings up Excel and Access objects and properties if you have established a reference to them.

The code below shows how you can create a default workspace, create a new database file (.mdb), and create a table definition within it with typical field types in the table:

Sub CommandButton1_Click()
    On Error Resume Next
    Set wksObj = DBEngine.Workspaces(0)
    Set dbsObj = wksObj.CreateDatabase("mydbase.mdb", dbLangGeneral)
    Set tblObj = dbsObj.CreateTableDef("mytable")
    With tblObj
        .Fields.Append .CreateField("text", dbText)
        .Fields.Append .CreateField("integer", dbInteger)
        .Fields.Append .CreateField("long", dbLong)
        .Fields.Append .CreateField("double", dbDouble)
        .Fields.Append .CreateField("boolean", dbBoolean)
        .Fields.Append .CreateField("memo", dbMemo)
        .Fields.Append .CreateField("currency", dbCurrency)
        .Fields.Append .CreateField("date", dbDate)
    End With
    dbsObj.TableDefs.Append tblObj
    dbsObj.TableDefs.Refresh
End Sub

Once a database and its list structures (tables with field definitions) have been defined, you can fill it with records using a table--type recordset object. The code below opens an existing database file (the one created above) and adds a new record to it:

Sub CommandButton2_Click()
    On Error Resume Next
    Set dbsObj = DBEngine.Workspaces(0).OpenDatabase("mydbase.mdb")
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
    rstObj.AddNew
    rstObj!Text = "a text value"
    rstObj!integer = 1
    rstObj!double = 3.1415926
    rstObj!Boolean = True
    rstObj!currency = "$4,240.54"
    rstObj!Date = "2/10/00"
    rstObj.Update
    rstObj.Close
End Sub
Set dbsObj = Nothing
End Sub

Note: As with Excel, it is important to track what recordsets and databases are "left open". In the above code, once the recordset has been appended with a new record, it is then closed. The database object is also set to "Nothing" to free up memory.

Throughout the remainder of this class, you will want to occasionally launch Access and verify the changes you are making to the database you're working with since our VBA application is not running Access as a visible session but rather in the background. If the mydbase.mdb file was created and opened with Access, you would see the figure below. Be sure to close Access before running your VBA code again to avoid read-only error messages.

Before we can start applying database functionality to AutoCAD tasks, we have to examine some remaining essential functions with database manipulations: how to view a record, delete a record, edit a record, find a particular record, etc. There are many resources which are supplied with different versions of Access, Microsoft Office, Visual Basic, Development SDK's and the like (such as the ADO library of objects, or a variety of Data Control objects), but this class assumes you have nothing but AutoCAD/VBA and Microsoft Access installed on your system. Therefore, all code examples use basic objects (mostly textboxes and command buttons) to accomplish all of these tasks. For the remainder of this class, an example database is also used, generated by the code below:

```vba
Sub CommandButton1_Click()
    On Error Resume Next
    Set wksObj = DBEngine.Workspaces(0)
    Set dbsObj = wksObj.CreateDatabase("mydbase.mdb", dbLangGeneral)
    Set tblObj = dbsObj.CreateTableDef("mytable")
    With tblObj
        .Fields.Append .CreateField("id", dbInteger)
        .Fields.Append .CreateField("manufacturer", dbText)
        .Fields.Append .CreateField("x", dbDouble)
        .Fields.Append .CreateField("y", dbDouble)
        .Fields.Append .CreateField("z", dbDouble)
    End With
    dbsObj.TableDefs.Append tblObj
    dbsObj.TableDefs.Refresh
End Sub
```

The database is populated (using the code below) with 3 records, each with a part ID, its manufacturer, and 3 coordinates reflecting a location in a CAD drawing:

```vba
Sub CommandButton2_Click()
    On Error Resume Next
    Set dbsObj = DBEngine.Workspaces(0).OpenDatabase("mydbase.mdb")
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
End Sub
```

```plaintext
<table>
<thead>
<tr>
<th>id</th>
<th>manufacturer</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>maker1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>maker2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>maker2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
To manipulate the database, the userform interface shown to the right (comprised solely of command buttons and textboxes) is used. Once a database is opened, the far right buttons can set the current record pointer. The left buttons can add a record, update or delete the current record. Once the user is finished, the database can be closed and the application can be quited. The code for each button is given below:

Sub CommandButton1_Click()
    'Opens the database file and populates the table-type recordset
    Set dbsObj = DBEngine.Workspaces(0).OpenDatabase("mydbase.mdb")
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
    rstObj.MoveLast
    rstObj.MoveFirst
End Sub

Sub CommandButton2_Click()
    'Appends a new record with values currently shown in the textboxes.
    rstObj.AddNew
    rstObj!id = TextBox1.Text
    rstObj!manufacturer = TextBox2.Text
    rstObj!x = TextBox3.Text
    rstObj!y = TextBox4.Text
    rstObj!z = TextBox5.Text
    rstObj.Update
End Sub

Sub CommandButton3_Click()
    'Updates the current record with values currently shown in the textboxes.
    rstObj.Edit
    rstObj!id = TextBox1.Text
    rstObj!manufacturer = TextBox2.Text
    rstObj!x = TextBox3.Text
    rstObj!y = TextBox4.Text
    rstObj!z = TextBox5.Text
    rstObj.Update
End Sub

Sub CommandButton4_Click()
    'Deletes the current record.
    rstObj.Delete
End Sub

Sub CommandButton5_Click()
    'This function closes the recordset and the database
    rstObj.Close
    Set dbsObj = Nothing
End Sub

Sub CommandButton6_Click()
    'Views the current record by passing values to the textboxes
    TextBox1.Text = rstObj!id.Value
    TextBox2.Text = rstObj!manufacturer.Value
Note: this class is not intended as an introduction to SQL.

Moving record-by-record through the entire database to set the current record pointer can be incredibly inefficient if your database has hundreds or thousands of records. Rather than setting the table-type recordset to all records in the database, you can open a recordset with an SQL query statement (string) instead so that the only records you step through are those which match your criteria. The code below shows how another command button and textbox can be used to do this. The form at the right shows how a sample SQL query statement finds a new record when the 'First Record' button is used.

SQL query-generated recordsets speed up your capacity to view, add, edit and delete any record in the complete database. To restore the recordset to a complete listing of all records in the database, simply reset the recordset object.
Entity/Record Linking Concepts

Now that essential coding concepts to manipulate Access from AutoCAD's VBA interface have been covered, let's look at a simple method of linking records with AutoCAD entities. Whether you're trying to relate a single CAD entity to a single record, a group of CAD entities to a single record, or a single CAD entity to a group of records, the basic process of "strapping" records to CAD entities requires using some unique piece of information about each CAD entity involved and passing it to a dedicated field of involved records. For years, many programmers have been placing entity handle values in a dedicated handle field of those records they wanted "linked" to accomplish this.

Note: an entity handle is a hexadecimal value generated by AutoCAD automatically whenever a new entity is created. The user does not have control over handle values assigned to entities, but handle values are preserved in drawings and are always unique. The handle value of any entity can be viewed by LISTing it.

There are many popular ways to associate unique information with CAD entities (such as attaching special extended entity data values to entities), but to keep things simple, this course will use entity handles and a dedicated entity handle field in the database to establish "links."

Once you select an entity and pass its handle value to a record handle field, you can now bi-directionally manipulate either the entity or the record based on the selection of either through your VBA code. Although there is no actual "link" established between any CAD entity and any record through this method, your utility knows how to treat the information in the database as a link to an entity in the drawing, and "unlinking" a record to an entity is as simple as setting a record's handle field to a null value.

If a database with a table "mytable" and a text field "handleval" were already opened and linked (as in above examples before records are added, edited, deleted, etc), the code below shows how an AutoCAD entity can be selected and its handle can be passed to the current record's "handleval" field:

```vba
Sub CommandButton1_Click()
    Dim sset As Object
    Dim entObj As Object
    Set sset = ThisDrawing.SelectionSets.Add("handler")
    UserForm1.hide
    sset.SelectOnScreen
    Set entObj = sset.Item(0)
    hand$ = entObj.Handle
    rstObj.AddNew
    rstObj!Handleval = hand$
    rstObj.Update
    sset.Delete
    UserForm1.Show
End Sub
```

There are a few important things to note in the above code. Temporary object variables are declared (Dim-ed) for a selection set collection and an entity object. A temporary selection set named "handler" is added to the selection set collection to hold the eventually selected entity. The selected entity itself is not deleted, but the selection set which collected it is deleted after the first (and only) entity's handle value is passed to the Handleval field of the current record. It is also very important to handle certain selection set functions (ex: SelectOnScreen, Delete) while the form is hidden. The handle value itself is treated as a string variable type since its values are hexadecimal (aka alpha-numeric).

AutoCAD VBA/Access Sample Application #1: Database Builder/Reporter/Highlighter
For this application, the database structure shown on the right, as well as the userform below it is assumed. The *entityhandle* field holds the hexadecimal value of any linked AutoCAD entity: if this field is empty, then the record is unlinked. This application has a new function which enables linked entities to be highlighted and selected based on an SQL query. It also utilizes the new entity selection concepts to link and unlink entities to records. If a database file already exists and the Create New Database function is used, the old database is destroyed and a new one is created. The Handle field on the userform uses a label instead of a textbox since this field is never entered by the user (it is strictly set by AutoCAD and the linking process and will appear to be blank if the current record is unlinked). There is a safeguard built into the linking function: the Link Selected Entity to Current Record function checks to see if any record is already linked to the selected entity. If it is already linked, a warning is issued and the new link is not allowed. You must unlink an already-linked entity with the Unlink Selected Entity from All Records function before you may re-link it to a different record. In other words, in this application it is assumed that any entity may only be linked to a single record at a time.

The figure to the right shows the AutoCAD drawing file to be linked. Of course, the whole point of this application is that once the user is capable of selecting and highlighting entities based on an SQL query, he/she can change their properties, delete them, WBLOCK them out to a file, etc. It's an elaborate FILTER-like function which uses linked Access data instead of AutoCAD properties. The application could be expanded to place record field values as text entities near or on top of any linked entity, to unlink groups of entities at a time or even to generate new records from unlinked entities in the drawing.

Code for the userform is provided:
Public wksObj As Object
Public dbsObj As Object
Public tblObj As Object
Public fldObj As Object
Public rstObj As Object
Public hstr As Variant

Sub CommandButton1_Click()
    On Error Resume Next
    Kill "mydbase.mdb"
    Set wksObj = DBEngine.Workspaces(0)
    Set dbsObj = wksObj.CreateDatabase("mydbase.mdb", dbLangGeneral)
    Set tblObj = dbsObj.CreateTableDef("mytable")
    With tblObj
        .Fields.Append .CreateField("entityhandle", dbText)
        .Fields.Append .CreateField("color", dbText)
        .Fields.Append .CreateField("ycoord", dbLong)
        .Fields.Append .CreateField("zcoord", dbLong)
        .Fields.Append .CreateField("price", dbCurrency)
    End With
dbsObj.TableDefs.Append tblObj
dbsObj.TableDefs.Refresh
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
End Sub

Sub CommandButton10_Click()
    sqlstr$ = TextBox2.Text
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenDynaset)
    rstObj.FindFirst sqlstr$
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub

Sub CommandButton11_Click()
    Dim sset As Object
    Dim entObj As Object
    Set sset = ThisDrawing.SelectionSets.Add("handler1")
    UserForm1.hide
    sset.SelectOnScreen
    Set entObj = sset.Item(0)
    hand$ = entObj.Handle
    rstObj.Edit: rstObj!entityhandle = hand$: rstObj.Update
    sset.Delete
    UserForm1.Show
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub

Sub CommandButton12_Click()
    Dim sset As Object
    Dim entObj As Object
    Set sset = ThisDrawing.SelectionSets.Add("handler")
    UserForm1.hide
    sset.SelectOnScreen
    Set entObj = sset.Item(0)
    hand$ = entObj.Handle
    Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenDynaset)
    ustr$ = "entityhandle = " & Chr(34) & hand$ & Chr(34)
    rstObj.FindFirst ustr$
    rstObj.Edit
    rstObj!entityhandle = Null
rstObj.Update
sset.Delete
UserForm1.Show
hstr = rstObj!entityhandle.Value
If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
TextBox4.Text = rstObj!Color.Value
TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton13_Click()
On Error Resume Next
Set dbsObj = DBEngine.Workspaces(0).OpenDatabase("mydbase.mdb")
Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
rstObj.MoveLast
rstObj.MoveFirst
hstr = rstObj!entityhandle.Value
If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
TextBox4.Text = rstObj!Color.Value
TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton14_Click()
rstObj.Close
End End Sub
Sub CommandButton15_Click()
rstObj.Edit: rstObj!entityhandle = Null: rstObj.Update
hstr = rstObj!entityhandle.Value
If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
TextBox4.Text = rstObj!Color.Value
TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton16_Click()
Set rstObj = dbsObj.OpenRecordset("mytable", dbOpenTable)
rstObj.MoveLast: rstObj.MoveFirst
hstr = rstObj!entityhandle.Value
If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
TextBox4.Text = rstObj!Color.Value
TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton17_Click()
Dim i As Integer
sqlstr$ = TextBox3.Text
Set rstObj = dbsObj.OpenRecordset(sqlstr$)
rstObj.MoveLast: rstObj.MoveFirst
hstr = rstObj!entityhandle.Value
If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
TextBox4.Text = rstObj!Color.Value
TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
For i = 0 To ThisDrawing.ModelSpace.Count - 1
    ThisDrawing.ModelSpace.Item(i).Highlight (False)
Next i
For i = 0 To ThisDrawing.ModelSpace.Count - 1
    hstr2$ = ThisDrawing.ModelSpace.Item(i).Handle: TextBox4.Text = hstr2$
    If hstr2$ = hstr Then ThisDrawing.ModelSpace.Item(i).Highlight (True)
Next i
    rstObj.MoveNext
    Loop
End Sub
Sub CommandButton2_Click()
    rstObj.AddNew
    rstObj!Color = TextBox4.Text: rstObj!xcoord = TextBox5.Text
    rstObj!ycoord = TextBox6.Text: rstObj!zcoord = TextBox7.Text
    rstObj!price = TextBox8.Text: rstObj.Update: rstObj.MoveLast
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton3_Click()
    rstObj.MoveFirst
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton4_Click()
    rstObj.MovePrevious
    If rstObj.BOF = True Then rstObj.MoveNext
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton5_Click()
    rstObj.MoveNext
    If rstObj.EOF = True Then rstObj.MovePrevious
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton6_Click()
    rstObj.MoveLast
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton7_Click()
    sqlstr$ = TextBox1.Text
    Set rstObj = dbsObj.OpenRecordset(sqlstr$)
    rstObj.MoveLast
    rstObj.MoveFirst
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton8_Click()
    rstObj.Edit
    rstObj!Color = TextBox4.Text: rstObj!xcoord = TextBox5.Text
    rstObj!ycoord = TextBox6.Text: rstObj!zcoord = TextBox7.Text
    rstObj!price = TextBox8.Text
    rstObj.Update
    hstr = rstObj!entityhandle.Value
    If Not hstr = "" Then Label3.Caption = hstr Else Label3.Caption = ""
    TextBox4.Text = rstObj!Color.Value
    TextBox5.Text = Format(rstObj!xcoord.Value, "0.00####")
    TextBox6.Text = Format(rstObj!ycoord.Value, "0.00####")
    TextBox7.Text = Format(rstObj!zcoord.Value, "0.00####")
    TextBox8.Text = Format(rstObj!price.Value, "$#,###.00")
End Sub
Sub CommandButton9_Click()
    rstObj.Delete
    Label3.Caption = "": TextBox4.Text = "": TextBox5.Text = ""
    TextBox6.Text = "": TextBox7.Text = "": TextBox8.Text = ""
End Sub