

Volume

2

CREATIVE DATA TECHNOLOGIES, INC.

DATALAYER.NET[™]

User's Guide

Table of Contents

Table of Contents.....	1
Chapter 1 – DataLayer.NET Overview	3
1.1 Overview of the DataLayer.NET Components.....	3
1.2 Summary.....	4
Chapter 2 – Using the DataConnection Component.....	5
2.1 Adding a Reference to the DataLayer.NET assembly.....	5
2.2 Adding the Project Level IMPORT for the CDT.DATALAYER Namespace.....	6
2.3 Connecting to the Database	7
2.4 Automatic Null Conversions	8
2.5 RunSQLStatement Routine.....	10
2.6 Scalar Functions	10
2.7 Using Parameterized SQL for best practices	12
2.8 Transaction Handling Routines in DataLayer.NET	13
2.9 Summary for the DataConnection component	15
Chapter 3 – Using the DataHandler Component	16
3.1 Using the Code Generator Program.....	16
3.2 Importing newly generated Class Source files into your Project	18
3.3 Decorating your class files with Attributes	19
3.4 Using your DataHandler classes for ANYTHING ! Really!.....	20
3.5 Using the DataHandler classes in read-only mode.....	20
3.6 Using Parameterized Queries for best practices	20
3.7 The DataSet buffer; where the data is kept while you are working with it	22
3.8 Retrieving Data into the DataSet buffer using the GetAllRows() routine ...	22
3.9 Retrieving Data into the DataSet buffer using the Paging routines.....	23
3.10 Strongly Typed Interface for the DataHandler Classes.....	24
3.11 Adding and Deleting Rows	25
3.12 Updating DataSet buffer data using 3 available methods	26
3.13 Sending Updates to the database	26
3.14 Adding new rows from scratch (without first fetching other rows)	26

TABLE OF CONTENTS

3.15 Sequencer Columns..... 28

Chapter 4 – A Simple Sample Program..... 31

4.1 Overview of the Simple Sample Program 31

4.2 Detailed Review of the Simple Sample Program..... 32

4.3 Summary..... 38

Chapter 5 – An In-Depth Sample Program..... 39

5.1 Overview of the In-Depth Sample Program 39

5.2 Detailed Review of the In-Depth Sample Program..... 42

5.3 Summary..... 57

Appendix A – License Activation 59

A.1 Overview of how the License Activation System works..... 59

A.2 Deactivating Machines..... 61



Chapter 1 - DataLayer.NET Overview

1.1 Overview of the DataLayer.NET Components

The DataLayer.NET Library is broken into two main components. The first component is called the **DataConnection** class. It is used to help you manage the database connection, manage transactions (if needed), and provide scalar database functions for your use. The second component is called the **DataHandler** class. This is the main component that does most of the database work for you when performing data retrieval and managed updates.

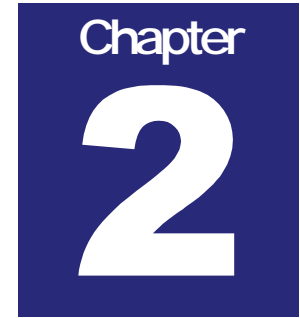
Here is the design philosophy we used when we built the DataLayer.NET Library:

- § Make the components intuitive and easily understood by average programmers.
- § Provide functionality that saves the programmer from having to work at such a low level with the ADO.NET Framework (directly interacting with ADO.NET classes such as the SqlConnection, SqlDataAdapter, SqlDataReader, SqlTransaction, SqlCommand, SqlParameter, and all the equivalent classes for the ODBC and OLE-DB interfaces).
- § Provide a programming platform that is independent of any particular back-end database, where none of interaction with the ADO.NET classes above will need to be coded by the programmer, making it much easier to scale an application from SQL Server to Oracle (OLE-DB driver), for example.
- § Make the library capable of accessing any of the following three types of databases:
 - * Microsoft SQL Server
 - * Any ODBC database
 - * Any OLE-DB database
- § Handle NULLS for the VB programmer. When reading or setting database values, VB programmers do not have the ability to represent a NULL value with program variables. The DataLayer.NET has a feature you can enable or disable that will automatically convert null values to each data type's MinValue constant. For example, a NULL integer will be read in as Integer.MinValue, and a NULL datetime value will be read in as DateTime.MinValue.
- § Dramatically decrease the amount of coding that is required to create typical database interactive programs.
- § Provide support for Parameterized SQL statements for best practices. The use of parameterized SQL protects your applications from SQL Injection attacks, and the SQL also runs much faster, as the back-end database (particularly SQL Server) can cache the SQL command signatures on the cache and run the SQL much faster upon subsequent matching SQL execution.

- § Fully support the use of the DataLayer.NET library for VB.NET programmers and C#.NET programmers for Client/Server, Web (ASP.NET) or 3-tier applications.
- § Provide a strongly typed interface to the column data, exposing the actual database column types read from the database.
- § Leverage the IntelliSense feature of the Visual Studio.NET environment to save typing time and avoid typing errors during programming. This is an additional benefit of the strongly typed interface of the DataLayer.NET library.
- § Trust and empower the programmers, but do not tie their hands behind their backs. Provide them with the data access objects to make their lives easier, but do not prevent them from directly sending some SQL to the database should they see fit. Extend their SQL Knowledge; do not try to replace it.
- § From the programmer's perspective, the library must be flexible to changes in the database structures (tables). The programmer will only have to update the schema information in a single class (the inherited DataHandler class for the table).
- § Performance must be a key goal in the design, as programmers will not want to use the library if they find out that there is a 10 – 20% reduction in performance by doing so. On the contrary, our clients experience a significant increase in performance when they adopt the parameterized SQL methodology as well as using the DataLayer.NET library.
- § Provide a record retrieval system that can either retrieve ALL of the rows of the result set into a buffer (DataSet), or retrieve the rows one page at a time, where page size and which page to retrieve can be specified by the user at runtime.

1.2 Summary

You can see from the above list of features above, the DataLayer.NET is going to save you a lot of coding and make your applications a lot easier to create and maintain. Some customers are stating that they are saving as much as 60 to 70% code savings in the data access layers of their programs.



Chapter 2 - Using the DataConnection Component

2.1 Adding a Reference to the DataLayer.NET assembly

Before you can use the DataLayer.NET components in your program, you have to add the **DataLayer.dll** assembly as a reference to your project. In order to do this, follow these steps:

1. Expand the "References" folder in the Solution Explorer for the your application, and then right-click on the References and choose "Add Reference..." from the popup menu.
2. Click the **Browse** button and navigate to the C:\Program Files\DataLayer.NET folder, and select the DataLayer.dll file and click Open.
3. You will see the DataLayer.dll assembly selected in the bottom part of the window as follows: (screen image on next page)

Or, you can add a Project Level IMPORT in one place (preferred method) by following these steps:

1. Right-click on the your Project name in the Solution Explorer, and select Properties from the popup menu.
2. Click on the **Imports** item on the left side (under “Common Properties”).
3. In the **Namespace** box, type in “CDT.DATALAYER” and then click on the **Add Import** button.
4. Click on the OK button to accept the changes.

2.3 Connecting to the Database

In order to connect to the database, you must declare an object of type `DataConnection`, as follows:

```
Dim objConnection As DataConnection
```

Often, however, you will want to make this a global variable in your application. There are several exceptions to this. For example, if you are developing an ASP.NET web application, you may want to declare the connection variable local to your subroutine, perform the work required, and close the database connection all within the same subroutine. Another example is when you are programming a 3-tier application. For the middle tier application’s code, you are typically connecting, performing work, and disconnecting from the database all in a single function call.

For a typical desktop VB.NET application, however, it is usually the case where you are simply declaring a global variable in your application for the database connection. To do this, right-click on your project and select “Add”, then “Add Module” from the popup menu. We usually simply give the new module a filename such as “globals.vb” (you can name it anything you like). After that, you would add the public declaration inside the globals module as follows:

```
Module Globals
    Public gSQLConn As DataConnection
End Module
```

Notice several things: I have used the prefix “g” instead of “obj” to indicate that the object is a global variable, and not a local variable. Also notice that I have called it `gSQLConn` (using “SQL” in the variable name). This further conveys that this connection handle is used to connect to a SQL Server database, as opposed to some other database type such as DB2, Oracle, etc. Now you can use this `gSQLConn` database handle throughout your VB.NET application.

Note: The sample code snippet given above, and throughout these discussion chapters are not really meant for you to immediately try out as a sample program. There is no sample program being built here. The source code is only given for reference so you can become familiar with the syntax used.

Now that we have declared a variable that will hold the database connection, we need to decide where and when to actually connect to the database. Some applications need the user to log in and

give their UserID and Password. For these type applications, you would build a login window asking the user for their UserID and Password, and your application would actually connect to the database in the code behind the **Login** button on that window. On the other hand, some applications do not require login, and can simply connect to the database when they start up, such as in the Form's Load event.

Wherever you determine appropriate, the following steps are required to create a `DataConnection` object and connect to the database:

```
gSQLConn = New DataConnection(DataLayer_ConnectionType.SQLServer)
gSQLConn.ConnectionString = "user id=guest;password=guestpass;" & _
                          "initial catalog=Northwind;data source=localhost"

Try
    gSQLConn.Connect()
Catch ex As Exception
    MsgBox(ex.Message, MsgBoxStyle.Critical, "Problem:")
Exit Sub
End Try
```

Notice that when you instantiate the `DataConnection` object, you can optionally specify the connection type (i.e. – `SQLServer`) as a parameter. You can also set this property directly in your code as well:

```
gSQLConn.ConnectionType = DataLayer_ConnectionType.SQLServer
```

Next in the code above is the setting of the `ConnectionString` property. Each type of database has its own custom type of tokens that must be set to form the `ConnectionString` property. The example given above is for SQL Server. Check your database documentation for the required `ConnectionString` format and content, or refer to <http://www.connectionstring.com> for a great reference guide to connection strings for many types of databases.

There is one more important token of which you should be aware. When your application tries to connect to the database, you can set a connection timeout value (in seconds). The default value is 15 seconds if you do not include this parameter. Here is a sample `ConnectionString` that includes this `Connect Timeout` parameter:

```
gSQLConn.ConnectionString = "user id=guest;password=guestpass;" & _
                          "initial catalog=Northwind;data source=localhost;" & _
                          "Connect Timeout=30;"
```

And finally, in the code snippet above, you can see the statement `gSQLConn.Connect()` is actually connecting to the database. As usual, you should always wrap database calls in a **Try / Catch** error handling block, as in the sample code.

2.4 Automatic Null Conversions

The `DataConnection` component has a Boolean property called “`AutoConvertNulls`” that you can set to `True` or `False`. The default value is `True`, which means it will convert all `NULL` values read from the database into the appropriate data type’s `MinValue` constant when reading the data. This is especially necessary with VB.NET programs because you can not load a `NULL` value into a

variable. For example, let's suppose you had directly opened a `DataReader` object and were reading in some field's data into a local variable as follows:

```
intAge = dr.Item("age")
```

If the value for the "age" field in that record has a `NULL` value, you will get an error on this line. The normal way that you have to handle this as a programmer is something like the following:

```
If IsDBNull(dr.Item("age")) Then
    intAge = 0 ' or -1, or some special value to let you know no value was present.
Else
    intAge = dr.Item("age")
End If
```

With the `DataLayer.NET` strongly typed interface and the `AutoConvertNulls` property set to true, you would be able to keep it to a single line of code, as follows:

```
intAge = objPerson.AGE(1)
```

If the record's "age" value is `NULL`, you will have `intAge` set to the `Integer.MinValue` constant. You can use this in your code if you want to detect nulls:

```
If (intAge = Integer.MinValue) Then
    MsgBox("Hey, there is a Null value for the Age detected!")
End If
```

The following `MinValue` constants are set for you by the `DataLayer.NET` library, depending upon the column's data type:

```
Integer.MinValue
Double.MinValue
Decimal.MinValue
DateTime.MinValue
```

For `String` variables, `NULL` value detection is handled a little differently. There is no "MinValue" constant for `Strings`. For most applications, simply converting the `NULL` value to an empty `String` "" is the desired behavior, so that is what we programmed into the `DataLayer.NET` library. However, this doesn't allow your program to distinguish a record that had a `NULL` value, as opposed to a record that had an empty string stored in the database. If you need to be able to detect `NULL` values in strings, you need to use the **`NullStringValue`** property. The default value for this property is an empty string "". However, if you want to detect null string values in your database, you could set this property to something special like "<NULL>" that is not likely for you to get in your actual database values. Here is suggested sample syntax for setting this property:

```
gSQLConn.NullStringValue = "<NULL>"
```

This **`NullStringValue`** property is a two-way street. Not only can it be used for reading and detecting null string values in the database, it can also be used if you want to set a `String` field in your database record to a `NULL` value. Here is a sample of how to do this (assuming you have set the `NullStringValue` per the above sample code to "<NULL>"):

```
objEmployee.MIDDLE_INITIAL(1) = gSQLConn.NullStringValue
```

After executing the statement above and then performing the Update(), the MIDDLE_INITIAL column for the record indicated will contain a NULL value.

Again, for 99% of all applications, you don't have to be overly concerned with NULL string detection and processing, so the default setting for the NullStringValue = empty string "" works fine for most programs.

2.5 RunSQLStatement Routine

Sometimes when you are programming, you just need to run an SQL statement directly against the database that performs some function, but does not necessarily return a result set.

For these situations, we have provided you with the DataConnection component's **RunSQLStatement** Routine.

Here is the call interface for the routine:

Method Signature: RunSQLStatement(byval strSQL As String)

Argument 1: strSQL As String: The SQL Statement to run

Returns: Nothing

As you can see, it takes a single argument: an SQL string containing the SQL command to run. For example, if you wanted to run an SQL Command to create a temporary table containing a copy of the Products table:

```
Dim strSQL As String
strSQL = "select * into #TempProducts FROM Products"
Try
    gSQLConn.RunSQLStatement(strSQL)
Catch
    MsgBox("Error creating the temp products table: " & ex.Message)
    Exit Sub
End Try
```

2.6 Scalar Functions

At times, you need to retrieve a single result from the database, and you don't want (nor should you have to) go through the trouble of creating a bunch of classes and source code to support the retrieval of the single result.

For this purpose, we have created a set of scalar functions for you to use to retrieve the single column, single row (scalar) results:

```
GetIntegerSQLResult
GetStringSQLResult
GetDoubleSQLResult
GetDecimalSQLResult
GetDateTimeResult
```

Here is the calling interface for the first one listed (GetIntegerSQLResult):

Method Signature: GetIntegerSQLResult(byval strSQL As String)

Argument 1: strSQL As String: SQL Query that returns a single row, single column Integer result

Returns: Integer (or Integer.MinValue if the database returns a NULL response)

Here is a sample of when you would want to use this function. Let's suppose you have a table called EMPLOYEE that contains all the information about your employees. The Primary Key for the table is an Integer field called ID_EMPLOYEE, and it is the responsibility of the application to generate unique ID numbers for each employee as it inserts the records. In order to do this, you will need to perform a MAX() lookup on the current maximum value for the ID_EMPLOYEE column in the table, as follows:

```
Dim intMaxID As Integer
Dim intNewID As Integer
Dim strSQL As String

strSQL = "SELECT MAX(id_employee) FROM EMPLOYEE"
Try
    intMaxID = gSQLConn.GetIntegerSQLResult(strSQL)
Catch
    MsgBox("Error getting MAX employee ID number: " & ex.Message)
    Exit Sub
End Try

If (intMaxID = Integer.MinValue) Then
    intNewID = 1
Else
    intNewID = intMaxID + 1
End If
...(SQL operation to INSERT the new EMPLOYEE record would go here)
```

Notice how I am checking for the null value (MinValue) and assigning a value of 1. This will happen when you are inserting the very first employee into the EMPLOYEE table.

Here are the calling interfaces of the other scalar functions that are available:

Method Signature: GetStringSQLResult(byval strSQL As String)

Argument 1: strSQL As String: SQL Query that returns a single row, single column String result

Returns: String (or NullStringValue if the database returns a NULL response)

Method Signature: GetDoubleSQLResult(byval strSQL As String)

Argument 1: strSQL As String: SQL Query that returns a single row, single column Double result

Returns: Double (or Double.MinValue if the database returns a NULL response)

Method Signature: GetDecimalSQLResult(byval strSQL As String)

Argument 1: strSQL As String: SQL Query that returns a single row, single column Decimal result

Returns: Decimal (or Decimal.MinValue if the database returns a NULL response)

Method Signature: GetDateTimeSQLResult(byval strSQL As String)

Argument 1: strSQL As String: SQL Query that returns a single row, single column DateTime result

Returns: DateTime (or DateTime.MinValue if the database returns a NULL response)

2.7 Using Parameterized SQL for best practices

Often, when you are sending SQL statements to the database, you need to specify values in your syntax. For example, if you wanted to retrieve all of the Employees from the EMPLOYEE table that have a last name starting with the letters MAR, in response to some search window you put together that the user can type in last name (or partial last name) to search for. You might be tempted to code the following statements:

```
Dim strSQL As String = "SELECT id_employee, nme_first, nme_last from EMPLOYEE " & _
```

```

        " WHERE nme_last LIKE " & Trim(txtSearchBox.Text) & "%”
Dim objEmployees as New EMPLOYEE(gSQLConn, strSQL)
Try
    ObjEmployees.GetAllRows()
Catch
    MsgBox("Error fetching employees: " & ex.Message)
End Try

```

With respect to pure syntax, the example above would work. However, there are two very important problems with doing it directly like this.

First of all, it opens up your application to SQL Injection attacks. This is of particular concern for web based applications where you are accepting input from the public. SQL Injection attacks occur when the user types special characters into input boxes to try to trick your program into performing unintended operations when the screen is submitted and your SQL runs.

The second big concern is performance. If you use parameterized SQL instead of directly plugging the values into your SQL, the back-end database will be able to cache the SQL's parameter signature, and the SQL statement will run much faster on subsequent executions, regardless of the actual value (or last name in this sample) that you are seeking.

Here is the same sample above, modified to make proper use of parameters:

```

Dim strSQL As String = "SELECT id_employee, nme_first, nme_last from EMPLOYEE " & _
    " WHERE nme_last LIKE @LastName"
Dim objEmployees as New EMPLOYEE(gSQLConn, strSQL)
Dim strValue As String = Trim(txtSearchBox.Text) & "%"
objEmployees.Parameters.Add(New CmdParameter("@LastName",DBType.String,strValue))
Try
    ObjEmployees.GetAllRows()
Catch
    MsgBox("Error fetching employees: " & ex.Message)
End Try

```

As you can see, with the help of the DataLayer.NET library, with only a single line of code added, you are able to protect your program from SQL Injection attacks, make the SQL perform much better, and clean up the ugly SQL concatenation that was required to sew in the values with the previous method used.

2.8 Transaction Handling Routines in DataLayer.NET

Sometimes when you are developing a multi-user database, it becomes necessary to execute a series of SQL operations in an environment that is guaranteed not to be affected by the interaction of any other users who are using the program at the same time, and all of the statements must successfully execute, or they must all be cancelled (rolled back) in the database.

For these situations, we must wrap the series of SQL statements or operations with a transaction block. A transaction block is treated as a single unit by the database. Either every single one of the statements in the block execute successfully, or if there is any error along the way, they are all rolled back in the database.

The DataLayer.NET library provides 4 routines relative to transaction processing:

```
StartTransaction()
CommitTransaction()
RollbackTransaction()
InTrans()
```

Here is what a typical Transaction block looks like from the point of view of your program:

```
Try
    gSQLConn.BeginTransaction()
    ...(SQL operation 1)
    ...(SQL operation 2)
    ...(etc, etc.)
    gSQLConn.CommitTransaction()
Catch
    gSQLConn.RollbackTransaction()
    MsgBox("Error encountered, transaction rolled back: " & ex.Message)
End Try
```

As you can see, if any of the SQL operations throw an exception, all of them are rolled back.

For a real-world example, take the sample earlier in the program where we are needing to generate a new Employee ID number. This is not likely to happen until you have thousands of people using the program all at once who are heavily using this same screen that enters new employee records all day (not very likely, but we'll go ahead and dress up this code for the example). Here is the modified code fragment to generate and use the next ID number:

```
Dim intMaxID As Integer
Dim intNewID As Integer
Dim strSQL As String

strSQL = "SELECT MAX(id_employee) FROM EMPLOYEE"
Try
    gSQLConn.BeginTransaction()
    intMaxID = gSQLConn.GetIntegerSQLResult(strSQL)

    If (intMaxID = Integer.MinValue) Then
        intNewID = 1
    Else
        intNewID = intMaxID + 1
    End If
```

```

    ...(SQL operation to INSERT the new EMPLOYEE record would go here)
    gSQLConn.CommitTransaction()
Catch
    gSQLConn.RollbackTransaction()
    MsgBox("Error saving new Employee record: " & ex.Message)
    Exit Sub
End Try

```

The code in the sample above would prevent any other user from being assigned the same next available Employee ID Number, thereby preventing any errors that might occur as a result of this.

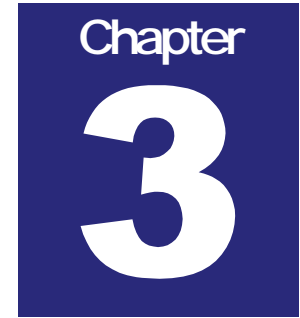
There is one more function available relating to Transactions. It is the **InTrans()** routine. This function will return a **True** if the connection is in the middle of a open / pending transaction, or a **False** if there is no open transaction.

2.9 Summary for the DataConnection component

You can see that the DataConnection component is used for global operations such as managing the database connection, transactions, scalar functions, directly executing SQL, and other housekeeping chores such as specifying desired handling of NULL values.

In your applications, you will typically only need to use a single DataConnection component. Exceptions to this are when you have a program that must connect simultaneously to more than one database. For these, you will need multiple DataConnection objects declared, and all of this is seamlessly supported by the DataLayer.NET library.

In the next chapter, you will be learning about the DataHandler component. It is used to manage data retrieval and updates for single entities (tables). You will typically have a descendant DataHandler class defined for each one of the tables in your system that you are using.



Chapter 3 - Using the DataHandler Component

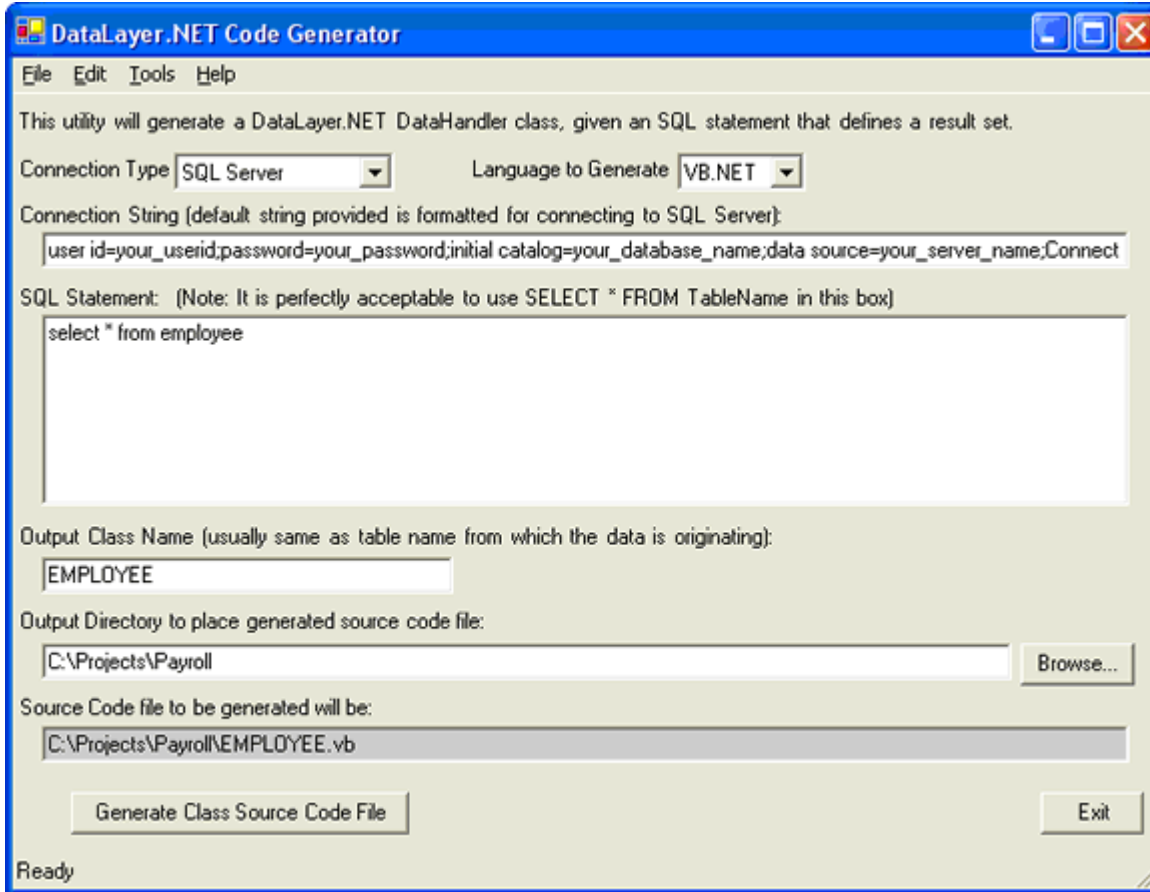
3.1 Using the Code Generator Program

In your programs, you do not directly use the DataHandler class. Instead you declare classes (one for each table in your database) that inherit from the DataHandler class, and then you use your inherited classes in your program.

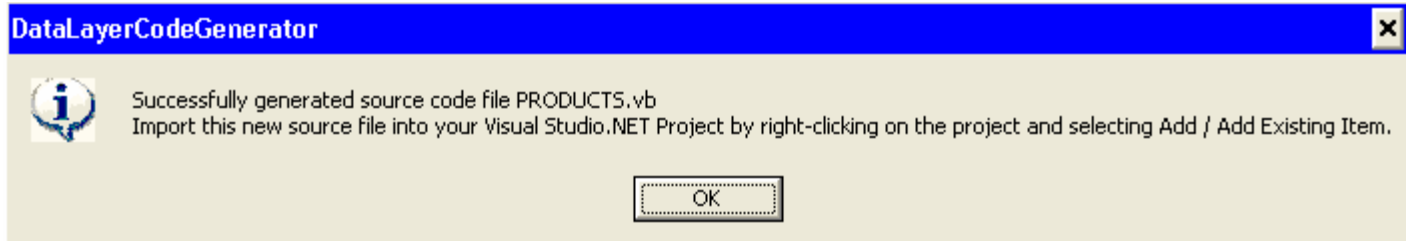
The process of creating these inherited classes is made virtually effortless by the DataLayer.NET Code Generator Program. All you have to do is feed it a SQL Query, and it will build a class file for you (in VB.NET or C#.NET) that you simply import into your project. The SQL query can be a very simple "SELECT * FROM TableName" type query, or it can contain complex left outer joins, correlated sub-queries, etc. That last statement may seem odd to you, since the DataHandler classes are used for both retrieval and updates. You see, the data retrieval functionality is kept completely separate from the data update facilities of the class. Because of this, you can have a display / browse window of employees that also includes information that is left outer joined from another table, such as the name of the department to which they belong, and at the same time, you can use that same exact object for updating the individual employee records when you go to code the employee edit window. How does it do this, you ask? Well, the data retrieval portion of the component works strictly off the SQL query you feed into it. The update facility, on the other hand, depends heavily on several things you need to define in your class, including using the attributes (more on this later) to identify which columns are updateable and which ones are part of the primary key. You also need to tell the class which table name should be used for the updates. Using this information, the class is able to manufacture the INSERT, UPDATE, and DELETE statements needed when updating the database after changes have been made.

Here is a brief walk-thru of using the Code Generator to generate and import a new class file:

1. After launching the desktop icon for the Code Generator Program, you should see the main window (screen shot on next page):



2. For the Connection Type, you have options for a) SQL Server, b) ODBC, and c) OLE-DB databases.
3. For the Language to Generate, you have options for a) VB.NET and b) C#.NET. For this sample application, leave it as **VB.NET**.
4. For the Connection String, this will vary depending on database type and location.
5. For the SQL Statement, you can normally enter any SQL Statement you would like that returns a result set.
6. For the Output Class Name, enter the name of the table in UPPER CASE, by standard convention.
NOTE: We usually create these particular class names in UPPER CASE. This will help you distinguish all the DataLayer.NET Code Generated entity classes in your application from all the other classes as you develop your application.
7. For the Output Directory, use the **Browse** button to select the folder where your project is located on your hard drive.
8. If everything is set up correctly, you should notice the complete path and filename for the class source code file that will be generated displayed in the box at the bottom of the screen.
9. The **Generate Class Source Code File** button is used to generate the class source code file. You will receive a confirmation window similar to the following:



10. Click OK, and then exit the Code Generator Program using the **Exit** button.

3.2 Importing newly generated Class Source files into your Project

After generating the new class source code file, you will need to import the new file into your project. Here are the steps to accomplish this task:

1. In the Visual Studio.NET environment, right click on the project's name and select "Add", and then "Add Existing Item..." from the sub-menu.
2. Select the class source code file and click the **Open** button to import the file.

Here is a short sample of what the generated source code files look like:

```
Imports CDT.DATALAYER

' Class PRODUCTS generated by DataLayer.NET Code Generator.
<Serializable()> Public Class PRODUCTS
    Inherits DataHandler

    Sub New(ByVal DataConn As DataConnection, ByVal SQL As String)
        MyBase.New(DataConn, SQL)

        Me.UpdateTable = "PRODUCTS"
    End Sub

    Public Shared Function GetBaseSQL() As String
        Dim SQL As String
        SQL = "SELECT productid,productname,supplierid,categoryid,quantityperunit," & _
            "unitprice,unitsinstock,unitsonorder,reorderlevel,discontinued " & _
            "FROM PRODUCTS"
        Return SQL
    End Function

    <Updateable()> Public Property PRODUCTID(ByVal RowNum As Integer) As Integer
        Get
            PRODUCTID = GetIntegerData(RowNum, "PRODUCTID")
        End Get
        Set
            SetData(RowNum, "PRODUCTID", Value)
        End Set
    End Property

    <Updateable()> Public Property PRODUCTNAME(ByVal RowNum As Integer) As String
        Get
```

```

        PRODUCTNAME = GetStringData(RowNum, "PRODUCTNAME")
    End Get
    Set
        SetData(RowNum, "PRODUCTNAME", Value)
    End Set
End Property

<Updateable()> Public Property UNITPRICE(ByVal RowNum As Integer) As Decimal
    Get
        UNITPRICE = GetDecimalData(RowNum, "UNITPRICE")
    End Get
    Set
        SetData(RowNum, "UNITPRICE", Value)
    End Set
End Property

End Class

```

The general structure of these files is the Class name declaration at the top, followed by a **New** constructor and then a function called **GetBaseSQL**. This function is useful for building SQL statements in your descendant classes, as it provides all the columns in a bare SQL statement. After that, you have Get / Set properties defined for each column in the table. Notice that the data type and methods called by each property are particular to the datatype of each column.

3.3 Decorating your class files with Attributes

The DataHandler class uses a .NET technology called “reflection” to learn a little bit about each column of the table being represented in your class. At runtime, it looks downstream at the inherited objects to see what attributes are defined, and changes its behavior accordingly.

After you generate each class file, you need to modify the class to decorate each column with the appropriate attributes. Here is a list of the available attributes:

Updateable() = The column is an updateable column (to be included in update statements sent to the database).

PrimaryKey() = The column is part of the primary key for the table.

Identity() = The column is an IDENTITY column (automatically generated number) in the database.

Sequencer() = The column is a child table sequencer (don't worry about this for now, more on this can be found in later on in this User's Manual).

DB2Timestamp() = The column is a DB2 timestamp. DB2 requires a very specific format for sending updates and inserts based on these column types.

As you can see from looking at the generated source code, the “Updateable()” attribute is added by default by the code generator for every column. Note that when you are adding the attributes, you do not have to enter the parenthesis (they will automatically be added).

Typically, all you need to do is go add the **PrimaryKey()** attribute for each one of the primary key columns. In order to specify more than one attribute for a column, simply put a comma between them, as in the following sample:

```

<PrimaryKey(), Updateable()> Public Property PRODUCTID(ByVal RowNum As Integer) As
Integer
    Get
        PRODUCTID = GetIntegerData(RowNum, "PRODUCTID")
    End Get
    Set
        SetData(RowNum, "PRODUCTID", Value)
    End Set
End Property

```

3.4 Using your DataHandler classes for ANYTHING ! Really!

Don't be afraid to take these generated class files and add properties and methods to them to carry out your specific business logic processing. As long as you do not include any of the attributes defined above for your new custom properties, they will be completely ignored by the underlying DataHandler class while processing your retrievals and updates.

Another common thing for customers to do is called encapsulation. Let's suppose you have a parent table called EMPLOYEE, and several children tables that are related to the EMPLOYEE table. You could declare public DataHandler object handles for each one of the children tables inside the EMPLOYEE class. Once you do that, you could create a method called "Load" that takes an employee ID number as an argument. The Load method would not only load the single EMPLOYEE record into the object, it would also instruct each one of the child DataHandler objects to load the multiple records that belong to this particular employee up into their DataSet buffers. A similar reverse implementation for a "Save" method could be easily done. This way, you have a single intelligent Employee object managing all the operations behind the scenes to manage all the information (in multiple tables) associated with this individual employee.

3.5 Using the DataHandler classes in read-only mode

If you have an application that does not perform any updates to the database, you can enforce this by removing all the Updateable() and PrimaryKey() attributes from the class file. Alternatively, you can simply remove the line that specifies the UpdateTable in the New() constructor. An exception will be thrown if any programmer tries to program an Update() against a table that does not have these attributes properly defined.

3.6 Using Parameterized Queries for best practices

This was mentioned above for the DataConnection component, but is worth bringing up once again to make sure that you are aware that the Parameterized Queries are also available for you to use when you are retrieving data using the SQL Queries for your DataHandler classes.

For example, let's suppose you have put together a search window, and the user can search for employees by Employee ID, Last Name, or Date of Hire date range. To handle this type of flexibility in the WHERE clause, your program will need to inspect which search boxes the user actually filled

out at runtime, and decide which parameters to use in the SQL based on the input conditions. Here is a sample of what the source code behind the “Search” button may look like:

```
' Now build the SQL to fetch the data...
mEmployees = New EMPLOYEES(gSQLConn, "")
Dim SQL As String
SQL = "SELECT EmployeeID,FirstName,LastName,Address1,Address2,City,State,Zip " & _
      "FROM EMPLOYEES WHERE "

' Add a WHERE clause parameter for each one of the search conditions given...
If (txtEmployeeID.Text <> "") Then
    SQL &= "(EmployeeID = @EmployeeID) AND "
    mEmployees.Parameters.Add(New CmdParameter("@EmployeeID", DbType.Int32, _
Cint(txtEmployeeID.Text)))
End If

If (txtLastName.Text <> "") Then
    SQL &= "LastName LIKE @LastName AND "
    mEmployees.Parameters.Add(New CmdParameter("@LastName", DbType.String, _
txtLastName.Text & "%"))
End If

If (txtHireDateStart.Text <> "") Then
    SQL &= "(HireDate BETWEEN @HireDateStart AND @HireDateEnd) AND "
    mEmployees.Parameters.Add(New CmdParameter("@HireDateStart", DbType.DateTime, _
CDate(txtHireDateStart.Text)))
    mEmployees.Parameters.Add(New CmdParameter("@HireDateEnd", DbType.DateTime, _
CDate(txtHireDateEnd.Text)))
End If

' Remove the trailing AND clause, if present...
If (MyRight(SQL, 4) = "AND ") Then
    SQL = Mid(SQL, 1, Len(SQL) - 4)
End If

' Remove the WHERE clause if no condition was given...
If (MyRight(SQL, 6) = "WHERE ") Then
    SQL = Mid(SQL, 1, Len(SQL) - 6)
End If

' Add the ORDER BY clause...
SQL &= " ORDER BY LastName, FirstName"

' Now retrieve the Employees...
mEmployees.SQLQuery = SQL
Try
    mEmployees.GetAllRows()
Catch ex As Exception
    MsgBox("Error Retrieving list of Employees: " & ex.Message)
Exit Sub
End Try
```

Note: In the sample code above, you would of course have to add validation on the criteria given, to catch such things as invalid dates or non-numeric Employee ID numbers input into the search boxes, etc.

3.7 The DataSet buffer; where data is kept while you're working with it

When you retrieve data from the database using the a DataHandler component, it fetches the data from the database and stores it into a DataSet buffer. This DataSet buffer is completely accessible by your program directly, if needed, using the **DataSet** property of your inherited DataHandler class.

You can use this DataSet property to populate a DataGrid for browsing or editing records. A DataSet, by definition, is an object that can contain multiple tables. For DataHandler DataSets, however, there is only a single table ever created inside the DataSet, and the table name is always called "data". Therefore, if you wanted to directly access the Last_Name column of the DataSet buffer from the Employee class created in the previous section to populate a combo box, the syntax would be as follows:

```
Dim I As Integer
For I = 0 To mEmployees.DataSet.Tables("data").Rows.Count - 1
    cboEmployees.Items.Add(mEmployees.DataSet.Tables("data").Rows(I).Item("Last_Name"))
Next
```

However, there is rarely any need to dive down into such ugly raw syntax as the sample above. In section 3.9 below, you will learn how to use the Strongly Typed interface for easy access to the row data.

3.8 Retrieving Data into the DataSet buffer using the GetAllRows() routine

When you have finished preparing the SQL query for the DataHandler component, you command the object to fetch the data using one of two different methods. Most of the time, a simple call to the GetAllRows() method will be used. This fetches all of the records meeting the query directly into the DataSet buffer. Typically, if you are working with a table that have a large number of rows, you will want to first count how many rows there are matching the search criteria, and decide whether you want to allow the user to retrieve that many rows or not (impose a "governor"). There is a special function called **GetQueryRowCount()** that should be used for this purpose. This method will take your standard SQL Query, and instead of fetching the actual row data, it will perform a SELECT COUNT(*) operation against the SQL Query to see how many rows there will be if the query is run. Using this information, you can decide whether you want to allow the user to run the query or not.

Here is a typical example showing the use of the GetQueryRowCount() routine and the GetAllRows() routine. Let's suppose that this particular window enforces that they always type in a last name to search for (and that's the only search field):

```
mEmployees = New EMPLOYEES(gSQLConn, "")
Dim SQL As String
SQL = "SELECT EmployeeID,FirstName,LastName,Address1,Address2,City,State,Zip " & _
    "FROM EMPLOYEES WHERE "

SQL &= "LastName LIKE @LastName"
```

```

mEmployees.Parameters.Add(New CmdParameter("@LastName", DbType.String, _
txtLastName.Text & "%"))

' Add the ORDER BY clause...
SQL &= " ORDER BY LastName, FirstName"
mEmployees.SQLQuery = SQL

' First perform a count of the matching Employees in the database...
Dim intCount As Integer
Try
    intCount = mEmployees.GetQueryRowCount()
Catch ex As Exception
    MsgBox("Error performing Count of matching Employees: " & ex.Message)
    Exit Sub
End Try
If (intCount > 500) Then
    MsgBox("There were too many employees (" & CStr(intCount) & _
        ") matching your search conditions.")
    Exit Sub
End If

' Now go ahead and retrieve the Employees...
Try
    mEmployees.GetAllRows()
Catch ex As Exception
    MsgBox("Error Retrieving list of Employees: " & ex.Message)
    Exit Sub
End Try

```

3.9 Retrieving Data into the DataSet buffer using the Paging routines

Sometimes you need to support being able to retrieve a very large result set, but you don't necessarily want to read all the records into memory (the DataSet buffer) all at once.

For this need, the DataLayer.NET library supports the notion of paging through your result set. Let me start by saying there are usually ways to work around having to do this, so the set of circumstances is fairly narrow. You can usually simply restrict the user from retrieving more than 10,000 rows, and the data retrieval will still occur in about 2 or 3 seconds. Even for web applications, you can store the DataHandler object on a Session variable, and simply have a DataGrid control to page through the records in the DataSet. This DataSet is typically held in memory in the session variable with little or no performance degradation (not to mention that the records only have to be retrieved once).

However, if there is a high traffic website supporting thousands of concurrent users, you can't simply go allocating that kind of memory on a per user (per Session) basis, as that would not scale up very well. In that case, using the Paging routines here would be appropriate.

Here are the routines you would use for the Paging interface (instead of the GetAllRows Routine):

PageSize – Before retrieving the first page, set this to state how many rows should be fetched for each page.

- GetPage(n)** – Gets a particular page number from the database (1 is the first page, not 0)
- CurrentPage** – A read-only property telling which page is currently in the DataSet buffer.
- PageCount** – A read-only property that runs a query to calculate how many pages there are.
- GetNextPage** – Retrieves the next page of data
- GetPrevPage** – Retrieves the previous page of data
- GetFirstPage** – Retrieves the first page of data
- GetLastPage** – Retrieves the very last page of data

Typically, the user interface for implementing a paged DataGrid will have buttons underneath the grid that read “First”, “Previous”, “Next”, and “Last”, along with a display that tells the user how many records there are, and the current page number, such as:

“1,458 Employees Page 1 of 78”

Again, to re-iterate, using the paging interface is usually only needed for high transaction volume web apps. If you are building a normal application under medium load, particularly if it is a desktop application, you can simply retrieve all of the row data into a DataSet and let the DataGrid present a scrollbar for the user to scroll quickly through all the records. The users actually prefer not to have to fuss with the paging controls unless they have to, as it is much quicker for them to perform alternate searching and sorting of their data, and navigate through the long result sets.

3.10 Strongly Typed Interface for the DataHandler Classes

As is often the case, you will need to directly interact with the data in the DataSet buffer. This is where the DataLayer.NET library’s Strongly Typed Interface really shines.

To access the row data, you simply type in the DataHandler’s object name, type a period, and you are presented with an alphabetical list of all the columns in the table. All you have to do is type the first few letters of the column name until the correct one is highlighted, and then press the **tab key** to accept the column name. The complete column name will be typed into your source code for you. This is compliments of a technology in Visual Studio.NET called IntelliSense.

Here are a few samples of reading and updating some rows in the Employee object:

```
‘ Obtain the State of the 4th employee in the buffer...
```

```
Dim strState As String
StrState = objEmployees.STATE(4)
```

```
‘ Update the Last Name of the 1st employee on the list...
```

```
objEmployees.LAST_NAME(1) = “SMITH”
```

Note that the second sample above is only updating the values in the DataSet buffer. The updates are not being sent to the database at the time this statement runs. You need to call the Update() method (covered in the section 3.11 below) to send the updates to the database.

IMPORTANT NOTE: The Strongly Typed row data interface is always 1 based. In other words, the following line will always return the FIRST employee's row value for the Last Name:

```
MsgBox(objEmployees.LAST_NAME(1))
```

This 1-based interface makes looping through the rows much easier, as you don't have to monkey around with remembering to subtract 1 from the indexers, etc.

Here is a sample that demonstrates an easier way to implement the sample code from section 3.6 above, using the strongly typed interface:

```
Dim I As Integer
For I = 1 To mEmployees.RowCount
    cboEmployees.Items.Add(mEmployees.LAST_NAME(I))
Next
```

3.11 Adding and Deleting Rows

As with any database application, you need to be able to add and delete rows from your database tables. To provide this functionality, the DataHandler class provides the AddRow and the DeleteRow routines.

Here is the calling interface for the AddRow method:

Method Signature: AddRow()

Arguments: (none)

Returns: Integer : The row number that was added to the DataSet buffer

Here is an example where a row is being added to a table:

```
' Add the new row...
Dim intNewRow As Integer
intNewRow = mEmployees.AddRow()

' Populate a few values for the new row...
mEmployees.EmployeeID(intNewRow) = 279
mEmployees.FirstName(intNewRow) = "Fred"
mEmployees.LastName(intNewRow) = "Flinstone"
```

Note: In the sample above, nothing has been saved yet to the database. That occurs when you call the Update() method. This way, all of your changes to all of your records are "queued" up and send to the database all at once. If you are processing updates in a loop, you can make repeated calls to the Update() method with each iteration of the loop. Only the new updates made since the last invocation of the Update() method will be sent to the database.

And here is the calling interface for the DeleteRow method:

Method Signature: DeleteRow(ByVal intRow As Integer)**Arguments:** intRow As Integer: The row number that you would like to delete**Returns:** (nothing)

Here is an example that deletes the 7th row from the DataSet:

```
mEmployees.DeleteRow(7)
```

3.12 Updating DataSet buffer data using three available methods

Just to bring it all together in one place for you, there are three methods available for updating the row data in the DataSet buffer:

- 1) Updating the rows using the strongly typed interface (preferred over #2 below):


```
mEmployees.LAST_NAME(1) = "SMITH"
```
- 2) Updating the rows of the DataSet table directly using the ADO.NET syntax:


```
mEmployees.DataSet.Tables("data").Rows(0).Item("Last_Name") = "SMITH"
```
- 3) Connecting a DataGrid Control to the DataSet and letting the user directly update the row data:


```
Dim dView As New DataView(mEmployees.DataSet.Tables("data"))
DataGrid1.DataSource = dView
```

3.13 Sending Updates to the database

Use the Update() method for sending updates to the database:

Method Signature: Update()**Arguments:** (none)**Returns:** (nothing)

The Update method will throw an exception if there are any errors sending the updates to the database, so you need to wrap the call to the Update() method in a Try / Catch block, as follows:

```
Try
    mEmployees.Update()
Catch ex As Exception
    MsgBox("Error Updating the Employees table: " & ex.Message)
Exit Sub
End Try
```

3.14 Adding new rows from scratch (without first fetching other rows)

Sometimes you need to insert a new record into a table, but it is not associated with any previous rows or DataHandler object that already has some rows fetched.

This situation presents an interesting dilemma for the DataHandler class. You see, it needs to have some kind of result set already retrieved into the DataSet so that the DataSet table is properly initialized with all of the columns and data types loaded (table “schema” information).

To provide a solution for this issue, we have a routine called **InitializeEmptyBuffer()**. Here is the calling interface:

Method Signature: InitializeEmptyBuffer()

Arguments: (none)

Returns: (nothing)

The InitializeEmptyBuffer method takes the SQL statement and temporarily mangles the WHERE clause to say “WHERE 1 = 0” to force the database to return an empty result set. However, even though no rows are returned into the DataSet, it does return the schema information for the result set, which allows you to then subsequently add the new row (or as many rows as needed).

Here is an example:

```
mEmployees = New EMPLOYEES(gSQLConn, EMPLOYEES.GetBaseSQL)

' Initialize the DataSet buffer...
Try
    mEmployees.InitializeEmptyBuffer()
Catch ex As Exception
    MsgBox("Error initializeing Employees buffer: " & ex.Message)
    Exit Sub
End Try

' Add the new Employee record to the DataSet buffer in memory...
Dim intRow As Integer
IntRow = mEmployees.AddRow()

' Set a few of the values for the new Employee...
mEmployees.ID_EMPLOYEE(intRow) = 247
mEmployees.FIRST_NAME(intRow) = "Henry"
mEmployees.LAST_NAME(intRow) = "Longfellow"

' Now go ahead and INSERT the new Employee record into the database...
Try
    mEmployees.Update()
Catch ex As Exception
    MsgBox("Error Inserting the Employee: " & ex.Message)
    Exit Sub
End Try
```

3.15 Sequencer Columns

If you are working with a set of child records that belong to a parent table, then you most likely will have some sort of sequence number column that becomes part of the primary key for the child table.

The following methods help automate the tedious process of sequence key generation:

<Sequencer(> Attribute that you decorate for the sequencing column (if there is one)

GetNextSequenceNumber() Scans the current values of the Sequencer Column values in the data buffer, and returns the next available sequence number. It returns 1 if no rows are present.

Here is the calling interface for the GetNextSequencerNumber routine:

Method Signature: GetNextSequencerNumber()

Arguments: (none)

Returns: Integer : The next available sequence number to use for a new row being added.

For example, let's suppose we have a table called EMPLOYEE_ACTION that contains all the payroll raises and promotions for a particular employee over their tenure. The table contains an ID_EMPLOYEE to identify which employee it is, and a sequencer column called NUM_SEQ_ACTION (Integer) that gets assigned numbers in sequence (1, 2, 3...) as new rows are added.

Here is a sample of what the class file for the table would look like (with just a few columns shown):

```
Imports CDT.DATALAYER

' Class EMPLOYEE_ACTION generated by DataLayer.NET Code Generator.
<Serializable(> Public Class EMPLOYEE_ACTION
    Inherits DataHandler

    Sub New(ByVal DataConn As DataConnection, ByVal SQL As String)
        MyBase.New(DataConn, SQL)

        Me.UpdateTable = "EMPLOYEE_ACTION"
    End Sub

    Public Shared Function GetBaseSQL() As String
        Dim SQL As String
```

```

    SQL = "SELECT
employeeid,num_seq_action,dte_action,cde_action,amt_adjustment,amt_new_salary" & _
        "FROM EMPLOYEE_ACTION"
    Return SQL
End Function

<PrimaryKey(), Updateable()> Public Property EMPLOYEEID(ByVal RowNum As Integer) As
Integer
    Get
        EMPLOYEEID = GetIntegerData(RowNum, "EMPLOYEEID")
    End Get
    Set
        SetData(RowNum, "EMPLOYEEID", Value)
    End Set
End Property

<PrimaryKey(), Updateable(), Sequencer()> Public Property NUM_SEQ_ACTION(ByVal RowNum
As Integer) As Integer
    Get
        NUM_SEQ_ACTION = GetIntegerData(RowNum, "NUM_SEQ_ACTION")
    End Get
    Set
        SetData(RowNum, "NUM_SEQ_ACTION", Value)
    End Set
End Property

<Updateable()> Public Property DTE_ACTION(ByVal RowNum As Integer) As Datetime
    Get
        DTE_ACTION = GetDateTimeData(RowNum, "DTE_ACTION")
    End Get
    Set
        SetData(RowNum, "DTE_ACTION", Value)
    End Set
End Property

<Updateable()> Public Property AMT_ADJUSTMENT(ByVal RowNum As Integer) As Decimal
    Get
        AMT_ADJUSTMENT = GetDecimalData(RowNum, "AMT_ADJUSTMENT")
    End Get
    Set
        SetData(RowNum, "AMT_ADJUSTMENT", Value)
    End Set
End Property

<Updateable()> Public Property AMT_NEW_SALARY(ByVal RowNum As Integer) As Decimal
    Get
        AMT_NEW_SALARY = GetDecimalData(RowNum, "AMT_NEW_SALARY")
    End Get
    Set
        SetData(RowNum, "AMT_NEW_SALARY", Value)
    End Set
End Property
End Class

```

As you can see, I have added the Sequencer() attribute to the NUM_SEQ_ACTION column, as well as adding the PrimaryKey attributes to the EMPLOYEEID and NUM_SEQ_ACTION columns. This

means the NUM_SEQ_ACTION column now has three attributes defined. This is completely OK for this situation.

Next, here is the sample code that looks at all the EMPLOYEE_ACTIVITY rows present, and returns the next available sequence number:

```
Dim intNewSeq As Integer
IntNewSeq = mEmployeeActions.GetNextSequenceNumber()
```

Note: In order to use the GetNextSequence() method, you must have to have first retrieved the existing rows into the DataSet buffer (even if there are no rows in existence yet). The method loops through all the rows that currently exist in the DataSet buffer. It does not perform any database lookup of the next available sequence number.

Here is the typical sequence of events that would be followed for our example with the Employee Actions table:

1. A particular Employee record is selected by the user for editing.
2. The EMPLOYEE record is read into a DataHandler class object.
3. All of the EMPLOYEE_ACTION records for the selected employee are read into another DataHandler class object.
4. The employee's record, along with all his or her action records, are displayed in a window for the user to interact with.
5. The user clicks the "Add Action" button to add a new EMPLOYEE_ACTION record.
6. At this time, the application would call the GetNextSequenceNumber() routine to determine the next available NUM_SEQ_ACTION value to use for the new record.



Chapter 4 - A Simple Sample Program

4.1 Overview of the Simple Sample Program

Since a picture is worth a thousand words, let's start with a screen shot of the completed Simple DataLayer.NET Sample Program:

The screenshot shows a Windows application window titled "Simple DataLayer.NET Sample Program". The interface is divided into three main steps:

- Step 1: Connect to the Database:** A text box contains the connection string "user id=guest;password=guestpass;initial catalog=Northwind;data source=localhost". A "Connect" button is to the right.
- Step 2: Retrieve an Employee's Record:** A text box for "Enter the Employee ID Number to Retrieve" is followed by a "Retrieve" button and the text "(try Employee ID 1 thru 9)".
- Step 3: Modify the Employee Info and Save the Changes:** A form with several input fields: "Mr./Ms:" (dropdown), "First Name:" (text box), "Last Name:" (text box), "Job Title" (text box), "Birth Date" (text box), "Date of Hire" (text box), "Address" (text box), "City" (text box), "State" (dropdown), "Zip" (text box), and "Comments" (text area).

At the bottom of the window, there are three buttons: "Save Changes", "Database: Not Connected" (in red text), and "Exit Program".

As you can see, the window is broken into three sections that help the user understand the flow of how to use the program. Please keep in mind that this is definitely NOT being put forth as a good way to design a program that edits the EMPLOYEES table. This is more of a simple “raw” editor that will serve as a teaching tool for you to learn how the DataLayer.NET library works.

When you run the program, it will fill in most of the Connection String for you. All you have to do is modify it slightly to connect to your Northwind database.

Step 1 for you is to modify the connection string accordingly and click the **Connect** button to connect to the database. If it is successful, you will see the red message at the bottom of the screen change from “Not Connected” to “Connected” in a green colored font.

Step 2 is for you to type in an Employee ID number to retrieve and then click the **Retrieve** button. The typical Northwind sample database comes with 9 employee records in the EMPLOYEES table, with ID numbers 1 thru 9 for you to choose from. After clicking the Retrieve button, you should see the employee’s information populated in the lower section of the window.

Step 3 is for you to make some changes to the employee’s information, and then click the **Save Changes** button. This will save all your changes to the database using the Update() method behind the scenes.

4.2 Detailed Review of the Simple Sample Program

Please note that even though we are presenting some sections of the source code below, this is not to be considered a full tutorial like the QuickStart application in the Getting Started Guide. The code is only presented here for your review.

The completed Simple Sample Program is included in a ZIP archive called Simple_DataLayer_Sample.ZIP in your C:\Program Files\DataLayer.NET folder. If you want to open it up and look at it using Visual Studio.NET, create a folder called C:\Projects\Simple_DataLayer_Sample, and unzip the contents of the archive into that folder and then open the solution file using Visual Studio.NET.

Let’s take a look at some of the source code that makes all this work. First of all, several module level variables (ones that are module level wide in scope), are given an “m” as part of their variable name prefix. These declarations are located just above the Form1_Load event:

```
Private mConnected As Boolean = False
Private mSQLConn As DataConnection ' DataLayer connection object.
Private mEmployee As EMPLOYEES ' DataLayer object to manage the employee record
                                ' (retrieval, updates, etc.)
```

The mConnected Boolean variable’s purpose is to keep track of whether the user has successfully connected to the database yet. If not, the program will not let them use the **Retrieve** or **Save Changes** buttons.

The mSQLConn is the database connection handle (a DataConnection object).

The mEmployee is a DataHandler class object that will manage the retrieval and update of the Employee record selected.

Here is the code behind the **Connect** button's Click event:

```
' Attempt to connect to the Northwind database...
Me.Cursor = Cursors.WaitCursor

' Create the DataLayer Connection object set for SQL Server mode...
mSQLConn = New DataConnection(DataLayer_ConnectionType.SQLServer)
mSQLConn.ConnectionString = Trim(txtConnectionString.Text)
Try
    mSQLConn.Connect()
Catch ex As Exception
    mConnected = False
    lblDatabase.Text = "Database: Not Connected"
    lblDatabase.ForeColor = Color.Red
    Me.Cursor = Cursors.Default
    MsgBox("Error attempting to connect to Northwind database: " & vbCrLf &
ex.Message, MsgBoxStyle.Critical, "Problem:")
    Exit Sub
End Try

mConnected = True
lblDatabase.Text = "Database: Connected"
lblDatabase.ForeColor = Color.ForestGreen
Me.Cursor = Cursors.Default
txtEmployeeID.Focus()
```

Nothing surprising in the code above that you haven't already been introduced to already. The main thing going on here is the mSQLConn.Connect() method call to connect to the database.

Here is the code behind the **Retrieve** button's Click event:

```
' Make sure the database is currently connected...
If (Not mConnected) Then
    MsgBox("You must first connect to the database before using this button!",
MsgBoxStyle.Critical, "Problem:")
    Exit Sub
End If

' Make sure a number has been entered...
If (Not IsNumeric(Trim(txtEmployeeID.Text))) Then
    MsgBox("Invalid Employee ID Number entered (numbers only)", MsgBoxStyle.Critical,
"Problem:")
    Exit Sub
End If

Dim intEmployeeID As Integer = CInt(Trim(txtEmployeeID.Text))

' Create the EMPLOYEES DataLayer object to do all the work for you...
mEmployee = New EMPLOYEES(mSQLConn, EMPLOYEES.GetBaseSQL & " WHERE EmployeeID =
@EmployeeID")
' Add the parameter to pass the Employee ID into the SQL...
```

```

mEmployee.Parameters.Add(New CmdParameter("@EmployeeID", DbType.Int32,
intEmployeeID))

' Now retrieve the Employee record from the database...
Try
    mEmployee.GetAllRows()
Catch ex As Exception
    MsgBox("Error retrieving the Employee Record: " & ex.Message,
MsgBoxStyle.Critical, "Problem:")
    Exit Sub
End Try
' Make sure it was found...
If (mEmployee.RowCount <> 1) Then
    txtSalutation.Text = ""
    txtFirstName.Text = ""
    txtLastName.Text = ""
    txtJobTitle.Text = ""
    txtBirthDate.Text = ""
    txtHireDate.Text = ""
    txtAddress.Text = ""
    txtCity.Text = ""
    txtState.Text = ""
    txtZip.Text = ""
    txtComments.Text = ""
    MsgBox("Employee ID Number " & CStr(intEmployeeID) & " was not found.",
MsgBoxStyle.Critical, "Problem:")
    Exit Sub
End If

' Go ahead and load the screen full of the employee's data (scatter)...
txtSalutation.Text = mEmployee.TITLEOF COURTESY(1)
txtFirstName.Text = mEmployee.FIRSTNAME(1)
txtLastName.Text = mEmployee.LASTNAME(1)
txtJobTitle.Text = mEmployee.TITLE(1)
If (mEmployee.BIRTHDATE(1) = Date.MinValue) Then
    txtBirthDate.Text = ""
Else
    txtBirthDate.Text = Format(mEmployee.BIRTHDATE(1), "MM/dd/yyyy")
End If
If (mEmployee.HIREDATE(1) = Date.MinValue) Then
    txtHireDate.Text = ""
Else
    txtHireDate.Text = Format(mEmployee.HIREDATE(1), "MM/dd/yyyy")
End If
txtAddress.Text = mEmployee.ADDRESS(1)
txtCity.Text = mEmployee.CITY(1)
txtState.Text = mEmployee.REGION(1)
txtZip.Text = mEmployee.POSTALCODE(1)
txtComments.Text = mEmployee.NOTES(1)

txtSalutation.Focus()

```

As you can see, it is using Parameterized SQL for best practices to retrieve only the selected employee's record, using the following statement:

```
' Add the parameter to pass the Employee ID into the SQL...
```

```
mEmployee.Parameters.Add(New CmdParameter("@EmployeeID", DbType.Int32,
intEmployeeID))
```

The actual retrieval of the record is happening when the GetAllRows() method is called:

```
mEmployee.GetAllRows()
```

Next, it is checking whether or not the record was found by inspecting the RowCount in the following conditional statement:

```
' Make sure it was found...
If (mEmployee.RowCount <> 1) Then
```

If the record was not found, it simply loads blanks into all the text edit boxes. If the record was successfully loaded, it loads all the values into the text boxes using the following statements:

```
' Go ahead and load the screen full of the employee's data (scatter)...
txtSalutation.Text = mEmployee.TITLEOFCOURTESY(1)
txtFirstName.Text = mEmployee.FIRSTNAME(1)
txtLastName.Text = mEmployee.LASTNAME(1)
txtJobTitle.Text = mEmployee.TITLE(1)
If (mEmployee.BIRTHDATE(1) = Date.MinValue) Then
    txtBirthDate.Text = ""
Else
    txtBirthDate.Text = Format(mEmployee.BIRTHDATE(1), "MM/dd/yyyy")
End If
If (mEmployee.HIREDATE(1) = Date.MinValue) Then
    txtHireDate.Text = ""
Else
    txtHireDate.Text = Format(mEmployee.HIREDATE(1), "MM/dd/yyyy")
End If
txtAddress.Text = mEmployee.ADDRESS(1)
txtCity.Text = mEmployee.CITY(1)
txtState.Text = mEmployee.REGION(1)
txtZip.Text = mEmployee.POSTALCODE(1)
txtComments.Text = mEmployee.NOTES(1)
```

Notice how it is looking for the possibility of NULL values (Date.MinValue) for the BirthDate, and displaying a blank text box if there is a NULL value.

And finally, here is the source code behind the **Save Changes** button's Click event:

```
' Make sure the database is currently connected...
If (Not mConnected) Then
    MsgBox("You must first connect to the database before using this button!",
MsgBoxStyle.Critical, "Problem:")
    Exit Sub
End If

' Make sure they have retrieved a record...
If (mEmployee Is Nothing) Then
    MsgBox("You have to first use the Retrieve button to fetch a record before using
the Save Changes button.", MsgBoxStyle.Information, "Problem:")
    Exit Sub
```

```

End If
If (mEmployee.RowCount = 0) Then
    MsgBox("No record present to save!", MsgBoxStyle.Information, "Problem:")
    Exit Sub
End If

' Gather up all the updated information for the employee (gather) and store back
into the DataLayer object...
mEmployee.TITLEOFCOURTESY(1) = Trim(txtSalutation.Text)
mEmployee.FIRSTNAME(1) = Trim(txtFirstName.Text)
mEmployee.LASTNAME(1) = Trim(txtLastName.Text)
mEmployee.TITLE(1) = Trim(txtJobTitle.Text)

If (Trim(txtBirthDate.Text) = "") Then
    mEmployee.BIRTHDATE(1) = Date.MinValue
ElseIf (Not IsDate(Trim(txtBirthDate.Text))) Then
    MsgBox("Invalid Date of Birth Entered!", MsgBoxStyle.Exclamation, "Problem:")
    Exit Sub
Else
    mEmployee.BIRTHDATE(1) = CDate(Trim(txtBirthDate.Text))
End If

If (Trim(txtHireDate.Text) = "") Then
    mEmployee.HIREDATE(1) = Date.MinValue
ElseIf (Not IsDate(Trim(txtHireDate.Text))) Then
    MsgBox("Invalid Date of Hire Entered!", MsgBoxStyle.Exclamation, "Problem:")
    Exit Sub
Else
    mEmployee.HIREDATE(1) = CDate(Trim(txtHireDate.Text))
End If

mEmployee.ADDRESS(1) = Trim(txtAddress.Text)
mEmployee.CITY(1) = Trim(txtCity.Text)
mEmployee.REGION(1) = Trim(txtState.Text)
mEmployee.POSTALCODE(1) = Trim(txtZip.Text)
mEmployee.NOTES(1) = Trim(txtComments.Text)

' Now UPDATE the employee's record in the database...
Try
    mEmployee.Update()
Catch ex As Exception
    MsgBox("Error updating the Employee Record: " & ex.Message, MsgBoxStyle.Critical,
"Problem:")
    Exit Sub
End Try

txtSalutation.Focus()

```

The first thing that the **Save Changes** button does is make sure the user has successfully connected, and has already retrieved a valid Employee record.

Next, it gathers up all the values from the window's text boxes, and updates the DataHandler's DataSet buffer with the new values:

' Gather up all the updated information for the employee (gather) and store back into the DataLayer object...

```

mEmployee.TITLEOFCOURTESY(1) = Trim(txtSalutation.Text)
mEmployee.FIRSTNAME(1) = Trim(txtFirstName.Text)
mEmployee.LASTNAME(1) = Trim(txtLastName.Text)
mEmployee.TITLE(1) = Trim(txtJobTitle.Text)

If (Trim(txtBirthDate.Text) = "") Then
    mEmployee.BIRTHDATE(1) = Date.MinValue
ElseIf (Not IsDate(Trim(txtBirthDate.Text))) Then
    MsgBox("Invalid Date of Birth Entered!", MsgBoxStyle.Exclamation, "Problem:")
    Exit Sub
Else
    mEmployee.BIRTHDATE(1) = CDate(Trim(txtBirthDate.Text))
End If

If (Trim(txtHireDate.Text) = "") Then
    mEmployee.HIREDATE(1) = Date.MinValue
ElseIf (Not IsDate(Trim(txtHireDate.Text))) Then
    MsgBox("Invalid Date of Hire Entered!", MsgBoxStyle.Exclamation, "Problem:")
    Exit Sub
Else
    mEmployee.HIREDATE(1) = CDate(Trim(txtHireDate.Text))
End If

mEmployee.ADDRESS(1) = Trim(txtAddress.Text)
mEmployee.CITY(1) = Trim(txtCity.Text)
mEmployee.REGION(1) = Trim(txtState.Text)
mEmployee.POSTALCODE(1) = Trim(txtZip.Text)
mEmployee.NOTES(1) = Trim(txtComments.Text)

```

Notice again how it is checking for an empty string in the Date of Birth textbox, and storing a NULL value (Date.MinValue) into the buffer if it is left blank. It is also checking for invalid dates, which will produce a user friendly message and refuse to proceed with the Save operation.

Next, the actual record is updated using the Update() method, as follows:

```

Try
    mEmployee.Update()
Catch ex As Exception
    MsgBox("Error updating the Employee Record: " & ex.Message, MsgBoxStyle.Critical,
"Problem:")
    Exit Sub
End Try

```

4.3 Summary

As you can see, this is a very simple single window (Form) application with very few lines of code, but it represents completely functioning round trip editing (retrieval and update) for the EMPLOYEES table.

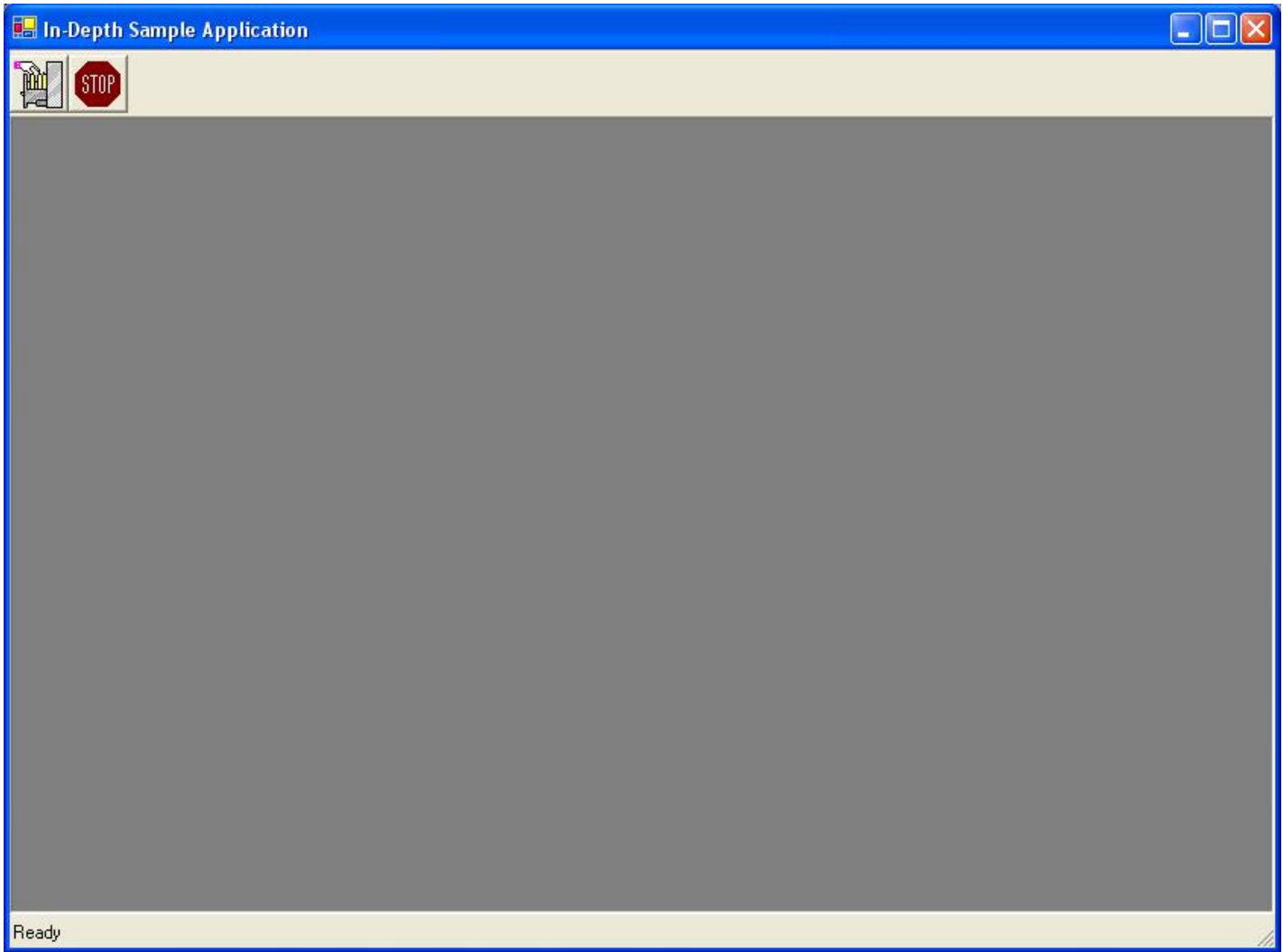
In the next chapter, you will be given a chance to see a more in-depth sample so you can see what a real-world application with a rich user interface would look like, so you can really see the time that the DataLayer.NET library saves you in programming.



Chapter 5 - An In-Depth Sample Program

5.1 Overview of the In-Depth Sample Program

Again, let's start with the presentation of the user interface, and then show you how it was built. When the program starts up, it displays an empty MDI (Multiple Document Interface) Frame Window with a toolbar containing several buttons across the top: (see screen shot on next page)



When you click on the first toolbar button (that looks like a hand reaching into a file cabinet drawer), you will see the following window:

Order Date	Order #	Ship To Name	Customer ID	Zip Code	City, Region
05/06/1998	11074	Simons bistro	SIMOB	1734	
05/06/1998	11075	Richter Supermarkt	RICSU	1204	
05/06/1998	11076	Bon app'	BONAP	13008	
05/06/1998	11077	Rattlesnake Canyon Grocery	RATTC	87110	Albuquerque, NM
05/05/1998	11070	Lehmans Marktstand	LEHMS	60528	
05/05/1998	11071	LILA-Supermercado	LILAS	3508	Barquisimeto, Lara
05/05/1998	11072	Ernst Handel	ERNSH	8010	
05/05/1998	11073	Pericles Comidas clásicas	PERIC	05033	
05/04/1998	11067	Drachenblut Delikatessen	DRACD	52066	
05/04/1998	11068	Queen Cozinha	QUEEN	05487-020	Sao Paulo, SP
05/04/1998	11069	Tortuga Restaurante	TORTU	05033	
05/01/1998	11064	Save-a-lot Markets	SAVEA	83720	Boise, ID
05/01/1998	11065	LILA-Supermercado	LILAS	3508	Barquisimeto, Lara
05/01/1998	11066	White Clover Markets	WHITC	98124	Seattle, WA
04/20/1998	11060	Frank's S.A.	FRANF	10100	

This window lets the user browse through all the Order records, and provides search fields to filter the list to Orders meeting only certain conditions. The way the searching works is that the user enters some criteria, and then clicks the **GO** button to search for the matching records.

Note: Normally, you would design the program to ask the user for the search criteria first, and then display the matching records. For this sample, since there are only about 800 order records in the database, the design chosen was to display a full list of all the orders when the window opens, and let the user search or filter the records by entering search criteria after that.

When the user clicks on the **Edit Order** button, they will see the following window that allows them to edit the Order information:

The screenshot shows a window titled "Order Edit Window" with the following fields and controls:

- Order Date: 02/04/1997
- Date Required: 03/18/1997
- Date Shipped: 02/07/1997
- Order ID: 10435
- Customer: Consolidated Holdings (dropdown)
- Employee: Callahan, Laura (dropdown)
- Ship Via: Fed-X Overnight (dropdown)
- Feight: 9.21 lbs.
- Ship To: Consolidated Holdings
- Address: Berkeley Gardens 12 Brewery
- Country: UK
- City: London
- Region: (empty)
- Zip Code: WX1 6LT

Order Items:

Product	Price	Qty	Discount
Aniseed Syrup	15.20	10	0.0
Grandma's Boysenberry Spread	16.80	12	0.0
Uncle Bob's Organic Dried Pears	27.80	10	0.0
Boston Crab Meat	42.00	1	0.0

Buttons: Add Item, Delete Item, Save, Close.

As you can see, there are drop-down boxes (combo boxes) provided for the user to select the Customer for the Order, the Employee, the Ship Via method, and even the Product Names.

The completed In-Depth Sample Program is included in a ZIP archive called `InDepth_DataLayer_Sample.ZIP` in your `C:\Program Files\DataLayer.NET` folder. If you want to open it up and look at it using Visual Studio.NET, create a folder called `C:\Projects\InDepth_DataLayer_Sample`, and unzip the contents of the archive into that folder and then open the solution file using Visual Studio.NET.

When the program starts up, you will notice some messages at the bottom of the window that briefly display when the list of Customers, Employees, and Products are loaded into global objects. The program loads these only once at startup, so they can be available to populate the drop-down boxes as needed on the various windows as they are used, without having to re-read the records from the database every time one of these window is opened.

5.2 Detailed Review of the In-Depth Sample Program

First, let's take a look at the code that is involved with loading up these three global list objects. First, we have a VB source code module called "AppGlobals.vb" that contains all of the public

variable declarations, along with some global functions that are available to the entire program (public).

The declarations for the 3 global list objects is at the top of the module, and look like this:

```
Public gCustomerList As CUSTOMERS
Public gEmployeeList As EMPLOYEES
Public gProductList As PRODUCTS
Public gSQLConn As DataConnection
```

Notice that I also show you that we have a global database connection handle variable, gSQLConn.

In support of this program, we have generated the DataHandler class files using the Code Generator Program. The class names referenced above are the CUSTOMERS.vb, EMPLOYEES.vb, and the PRODUCTS.vb.

Notice also that these declarations are only declarations of the global variables. The objects themselves are not being created by these statements.

The actual creation of the three global list object occurs in the MainForm.vb source file in the form's Load event. A timer is fired as soon as the form loads, and the loading of the three objects occurs in the Timer's Elapsed event, as follows:

```
' First connect to the database...
gSQLConn = New DataConnection(DataLayer_ConnectionType.SQLServer)
gSQLConn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;User
ID=guest;Password=guestpass"
Try
    gSQLConn.Connect()
Catch ex As Exception
    MsgBox("Error connecting to Northwind database: " & ex.Message,
MsgBoxStyle.Critical, "Problem:")
Exit Sub
End Try

' Load all the lookup objects.
' 1. Load the Customers into a global object...
AppFrame.Cursor = Cursors.WaitCursor
AppFrame.StatusBar1.Text = "Loading List of Customers..."
gCustomerList = New CUSTOMERS(gSQLConn, "SELECT CustomerID, CompanyName FROM
CUSTOMERS ORDER BY CompanyName")
Try
    gCustomerList.GetAllRows()
Catch ex As Exception
    MsgBox("Error loading list of Customers from the database: " & ex.Message,
MsgBoxStyle.Critical)
Exit Sub
End Try

' 2. Load the Employee into a global object...
AppFrame.StatusBar1.Text = "Loading List of Employees..."
gEmployeeList = New EMPLOYEES(gSQLConn, "SELECT EmployeeID, FirstName, LastName FROM
EMPLOYEES ORDER BY LastName, FirstName")
Try
    gEmployeeList.GetAllRows()
```

```

Catch ex As Exception
    MsgBox("Error loading list of Employees from the database: " & ex.Message,
MsgBoxStyle.Critical)
    Exit Sub
End Try

' 3. Load the Products into a global object...
AppFrame.StatusBar1.Text = "Loading List of Products..."
gProductList = New PRODUCTS(gSQLConn, "SELECT ProductID, ProductName FROM PRODUCTS
ORDER BY ProductName")
Try
    gProductList.GetAllRows()
Catch ex As Exception
    MsgBox("Error loading list of Products from the database: " & ex.Message,
MsgBoxStyle.Critical)
    Exit Sub
End Try

```

Notice that the program is connecting to the database here in this event.

Also, notice that the SQL Query for all three objects is directly specified here instead of simply using the GetBaseSQL() method, as I did in many samples I have shown you so far. The purpose of these global list objects is to support the drop-down list boxes. The only information needed to support the drop-down boxes is the Employee / Customer / Product ID Number, and the Employee / Customer / Product Name. There is an important concept to be gained here. Never fetch data you don't need from the database. It is inefficient, and will consume extra memory and slow down your application, and could potentially slow down your entire database and impact the overall performance of your system. It is important to understand that even though your DataHandler classes have every single column of the database defined as properties, that does not mean you are obligated to retrieve all the columns in the SQL Query. This is particularly true for those DataHandler objects you only plan to use in Read-Only mode (not for updates), as is the case for these three global list objects. When you are performing updates, you will want to include ALL the columns that are marked with the Updateable() and PrimaryKey() attributes, at a minimum.

Next, notice what happens when the user clicks on the first toolbar button. The following two subroutines get involved in the opening of the Browse Orders Window:

```

Private Sub ToolBar1_ButtonClick(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ToolBarButtonClickEventArgs) Handles ToolBar1.ButtonClick
    Select Case e.Button.Tag
        Case "BROWSE"
            OpenBrowseWindow()
        Case "ABOUT"
            MsgBox("DataLayer.NET Sample Application, v1.0")
        Case "EXIT"
            Me.Close()
        End Select
End Sub

Private Sub OpenBrowseWindow()
    ' Check to see if it is already open, and maybe minimized, etc...

```

```

For Each objForm As Form In Me.MdiChildren
  If (objForm.Name = "BrowseOrders") Then
    If (objForm.WindowState = FormWindowState.Minimized) Then
      objForm.WindowState = FormWindowState.Normal
    End If
    objForm.Focus()
    Exit Sub
  End If
Next

' Since we fell thru to here, we need to open a new instance of the Transmission
List Window...
Dim objBrowseOrdersWindow As New BrowseOrdersWindow
objBrowseOrdersWindow.MdiParent = Me
objBrowseOrdersWindow.Show()
End Sub

```

Notice that the `OpenBrowseWindow` routine is checking to see if there is already an instance of this window opened, and if there is, it simply brings that window back to the foreground. Otherwise, it creates a new window object and shows it.

When the `Browse Orders Window` opens, the `Form's Load` event does a lot of formatting of the columns of the `DataGrid` control on the form:

```

' Reposition to the upper left corner...
Me.Left = 0
Me.Top = 0

' Format the DataGrid...
DataGrid1.CaptionVisible = False
DataGrid1.ReadOnly = True
DataGrid1.AlternatingBackColor = Color.LightCyan
DataGrid1.RowHeadersVisible = False

' Create custom Column Style objects to handle the columns...
Dim tsOrders As New DataGridTableStyle
tsOrders.MappingName = "data"
tsOrders.RowHeadersVisible = False

' NOTE: The NonEditColumnStyle is a custom Style class that was created to help
'       display grids with a highlight bar.
' 0. Order Date...
Dim styleOrderDate As New NonEditColumnStyle
styleOrderDate.SelectColumnNumber = 7
styleOrderDate.MappingName = "OrderDate"
styleOrderDate.HeaderText = "Order Date"
styleOrderDate.Width = 70
styleOrderDate.ReadOnly = True
styleOrderDate.NullText = ""

' 1. Order ID...
Dim styleOrderID As New NonEditColumnStyle
styleOrderID.SelectColumnNumber = 7
styleOrderID.MappingName = "OrderID"
styleOrderID.HeaderText = "Order #"

```

```

styleOrderID.Width = 50
styleOrderID.ReadOnly = True
styleOrderID.NullText = ""

' 2. Ship To Name...
Dim styleShipToName As New NonEditColumnStyle
styleShipToName.SelectColumnNumber = 7
styleShipToName.MappingName = "ShipName"
styleShipToName.HeaderText = "Ship To Name"
styleShipToName.Width = 190
styleShipToName.ReadOnly = True
styleShipToName.NullText = ""

' 3. Customer ID...
Dim styleCustomerID As New NonEditColumnStyle
styleCustomerID.SelectColumnNumber = 7
styleCustomerID.MappingName = "CustomerID"
styleCustomerID.HeaderText = "Customer ID"
styleCustomerID.Width = 70
styleCustomerID.ReadOnly = True
styleCustomerID.NullText = ""

' 4. Zip Code...
Dim styleZipCode As New NonEditColumnStyle
styleZipCode.SelectColumnNumber = 7
styleZipCode.MappingName = "ShipPostalCode"
styleZipCode.HeaderText = "Zip Code"
styleZipCode.Width = 60
styleZipCode.ReadOnly = True
styleZipCode.NullText = ""

' 5. City, Region...(computed column)
Dim styleCityRegion As New NonEditColumnStyle
styleCityRegion.SelectColumnNumber = 7
styleCityRegion.MappingName = "CityRegion"
styleCityRegion.HeaderText = "City, Region"
styleCityRegion.Width = 150
styleCityRegion.ReadOnly = True
styleCityRegion.NullText = ""

' 6. Ship Date...
Dim styleShipDate As New NonEditColumnStyle
styleShipDate.SelectColumnNumber = 7
styleShipDate.MappingName = "ShippedDate"
styleShipDate.HeaderText = "Date Shipped"
styleShipDate.Width = 70
styleShipDate.ReadOnly = True
styleShipDate.NullText = ""

' 7. Selected flag (invisible)...
Dim styleSelectedFlag As New NonEditColumnStyle
styleSelectedFlag.SelectColumnNumber = 7
styleSelectedFlag.MappingName = "Selected"
styleSelectedFlag.ReadOnly = True
styleSelectedFlag.NullText = ""
styleSelectedFlag.Width = 0

```

```

tsOrders.GridColumnStyles.Add(styleOrderDate)
tsOrders.GridColumnStyles.Add(styleOrderID)
tsOrders.GridColumnStyles.Add(styleShipToName)
tsOrders.GridColumnStyles.Add(styleCustomerID)
tsOrders.GridColumnStyles.Add(styleZipCode)
tsOrders.GridColumnStyles.Add(styleCityRegion)
tsOrders.GridColumnStyles.Add(styleShipDate)
tsOrders.GridColumnStyles.Add(styleSelectedFlag)
DataGrid1.TableStyles.Add(tsOrders)

' Load the values for the Customer dropdown box...
cboCustomer.Items.Add(New ComboItem("ALL", "ALL"))
Dim I As Integer
For I = 1 To gCustomerList.RowCount
    cboCustomer.Items.Add(New ComboItem(gCustomerList.COMPANYNAME(I),
gCustomerList.CUSTOMERID(I)))
Next
Select_Combo_Value(cboCustomer, "ALL")

```

Notice that at the end, it is also loading the values for the Customer drop-down box, using the global list object called gCustomerList (a DataHandler object populated with the Customer ID's & Names).

Notice also the mention of the NonEditColumnStyle. There is poor native support for row highlighting in the DataGrid control that Microsoft provides. It can highlight the selected row, but when the user clicks on a particular row, it also gives focus to a particular cell, which changes the highlight bar color over that particular cell, which is not the intended behavior. If you look at the custom programmed NonEditColumnStyle class, it uses the notion of a "Selected" record flag that keeps track of which row is selected and highlighted. The extra column in the SQL Query called "Selected" is used for this purpose.

Next, take a look at the Load_Data subroutine, where the user's search criteria is taken into account and the matching orders are retrieved from the database and displayed in the DataGrid control:

```

Public Sub Load_Data(Optional ByVal blnRowSet As Boolean = True)
    AppFrame.Cursor = Cursors.WaitCursor
    AppFrame.StatusBar1.Text = "Loading Orders..."

    ' Using the current criteria on the screen, obtain a list of the matching orders.

    ' Order ID...
    Dim intOrderID As Integer = Integer.MinValue
    If (Not MyEmpty(txtOrderID.Text)) Then
        If (Not IsNumeric(txtOrderID.Text)) Then
            AppFrame.Cursor = Cursors.Default
            AppFrame.StatusBar1.Text = "Ready"
            MsgBox("Order ID numbers must be numbers only!", MsgBoxStyle.Critical)
            Exit Sub
        End If

        intOrderID = CInt(txtOrderID.Text)
    End If

    ' Customer ID...

```

```

Dim strCustomerID As String = ""
Dim objComboItem As ComboItem
objComboItem = CType(cboCustomer.SelectedItem, ComboItem)
If (objComboItem.Value <> "ALL") Then
    strCustomerID = objComboItem.Value
End If

' Order Date Range...
Dim dteOrderDateStart As Date = Date.MinValue
Dim dteOrderDateEnd As Date = Date.MinValue
If (Not MyEmpty(txtDate1.Text)) Or (Not MyEmpty(txtDate2.Text)) Then
    ' Make sure it is a valid date...
    If (Not IsDate(txtDate1.Text)) Then
        AppFrame.Cursor = Cursors.Default
        AppFrame.StatusBar1.Text = "Ready"
        MsgBox("Invalid Order Date FROM given.", MsgBoxStyle.Critical)
        Exit Sub
    End If
    If (Not IsDate(txtDate2.Text)) Then
        AppFrame.Cursor = Cursors.Default
        AppFrame.StatusBar1.Text = "Ready"
        MsgBox("Invalid Order Date TO given.", MsgBoxStyle.Critical)
        Exit Sub
    End If

    dteOrderDateStart = CDate(txtDate1.Text)
    dteOrderDateEnd = CDate(txtDate2.Text)
End If

' Checkbox for Showing only Not Shipped Orders...
Dim blnOnlyNotShipped As Boolean = False
If cbxNotShipped.Checked Then
    blnOnlyNotShipped = True
End If

' Ship To Name..
Dim strShipName As String = ""
If (Not MyEmpty(txtShipToName.Text)) Then
    strShipName = Trim(txtShipToName.Text) & "%" ' Percent added for the LIKE clause.
End If

' Zip Code...
Dim strZipCode As String = ""
If (Not MyEmpty(txtZipCode.Text)) Then
    strZipCode = Trim(txtZipCode.Text) & "%" ' Percent added for the LIKE clause.
End If

' Now build the SQL to fetch the data...
mOrders = New ORDERS(gSQLConn, "")
Dim SQL As String
SQL = "SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate, " & _
    "ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, " & _
    "ShipPostalCode, ShipCountry, 'N' As Selected " & _
    "FROM ORDERS WHERE "

' Add a WHERE clause parameter for each one of the search conditions given...
If (intOrderID <> Integer.MinValue) Then

```

```

    SQL &= "(OrderID = @OrderID) AND "
    mOrders.Parameters.Add(New CmdParameter("@OrderID", DbType.Int32, intOrderID))
End If
If (Not MyEmpty(strCustomerID)) Then
    SQL &= "CustomerID = @CustomerID AND "
    mOrders.Parameters.Add(New CmdParameter("@CustomerID", DbType.String,
strCustomerID))
End If
If (dteOrderDateStart <> Date.MinValue) Then
    SQL &= "(OrderDate BETWEEN @OrderDateStart AND @OrderDateEnd) AND "
    mOrders.Parameters.Add(New CmdParameter("@OrderDateStart", DbType.DateTime,
dteOrderDateStart))
    mOrders.Parameters.Add(New CmdParameter("@OrderDateEnd", DbType.DateTime,
dteOrderDateEnd))
End If

If blnOnlyNotShipped Then
    SQL &= "(shippeddate IS NULL) AND "
End If
If (Not MyEmpty(strShipName)) Then
    SQL &= "(ShipName LIKE @ShipName) AND "
    mOrders.Parameters.Add(New CmdParameter("@ShipName", DbType.String, strShipName &
"%"))
End If
If (Not MyEmpty(strZipCode)) Then
    SQL &= "(ShipPostalCode LIKE @ZipCode) AND "
    mOrders.Parameters.Add(New CmdParameter("@ZipCode", DbType.String, strZipCode &
"%"))
End If

' Remove the trailing AND clause, if present...
If (MyRight(SQL, 4) = "AND ") Then
    SQL = Mid(SQL, 1, Len(SQL) - 4)
End If

' Remove the WHERE clause if no condition was given...
If (MyRight(SQL, 6) = "WHERE ") Then
    SQL = Mid(SQL, 1, Len(SQL) - 6)
End If

' Add the ORDER BY clause...
SQL &= " ORDER BY OrderDate DESC"

' Now retrieve the Orders...
mOrders.SQLQuery = SQL

Try
    mOrders.GetAllRows() ' no error handling here, throw it up to the UI if an error
occurs.
Catch ex As Exception
    AppFrame.Cursor = Cursors.Default
    AppFrame.StatusBar1.Text = "Ready"
    MsgBox("Error Retrieving list of Orders: " & ex.Message, MsgBoxStyle.Critical,
"Problem:")
Exit Sub
End Try

```

```

' Add the computed column to display the City, State...
Dim dCol As New DataColumn("CityRegion", GetType(String))
dCol.Expression = "shipcity + ', ' + shipregion"
mOrders.DataSet.Tables("data").Columns.Add(dCol)

' Load the information into the DataGrid...
Dim dView As New DataView(mOrders.DataSet.Tables("data"))
DataGrid1.DataSource = dView

' Update the record count display at the bottom...
lblRecordCount.Text = CStr(mOrders.RowCount()) & " Order(s) Listed"

' If there were any rows found, select the first row...
If blnRowSet Then
    SelectFirstRow()
End If

' Set focus to the input box...
txtOrderID.Focus()
AppFrame.Cursor = Cursors.Default
AppFrame.StatusBar1.Text = "Ready"

End Sub

```

Notice how the WHERE clause for the SQL Query is built on the fly, using parameters, depending upon which search fields they used. In the end, the key action is the call to the GetAllRows() method to retrieve the data, and the creation of the DataView object to hook up the DataSet to the DataGrid control for displaying the data.

Next, there is an **Edit** button to pull up and edit an Order:

```

Private Sub btnEdit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnEdit.Click
    ' Edit the selected order...
    Dim intOrderID As Integer = CInt(DataGrid1.Item(DataGrid1.CurrentRowIndex, 1))

    DataGrid1.Focus()

    Dim objOrderEditWindow As New OrderEditWindow
    objOrderEditWindow.intOrderID = intOrderID
    objOrderEditWindow.MdiParent = Me.MdiParent
    objOrderEditWindow.Show()
End Sub

```

Notice it is creating a new Edit window object every time. Since this is an MDI application, the users can open as many Order Edit Windows as they want.

When the Order Edit window opens, it fires a timer off to load the data. Here is what that timer event looks like that loads the Order record into the window:

```

Private Sub PostOpenTimer_Elapsed(ByVal sender As System.Object, ByVal e As
System.Timers.ElapsedEventArgs) Handles PostOpenTimer.Elapsed

```

```

PostOpenTimer.Enabled = False

If (intOrderID = -1) Then
    ' We are creating a new Order...
    txtOrderID.Text = "(new)"

    mOrder = New ORDERS(gSQLConn, ORDERS.GetBaseSQL)
    mOrder.InitializeEmptyBuffer()

    ' Add a new blank row...
    mOrder.AddRow() ' will always return 1 for the new row number, so the hard
coding of 1 below is OK.

    ' Populate -1 as the OrderID to act as a signal that this is a brand new record
being added...
    mOrder.ORDERID(1) = -1
Else
    ' Load the existing order from the database...
    txtOrderID.Text = CStr(intOrderID)

Try
    mOrder = New ORDERS(gSQLConn, ORDERS.GetBaseSQL & " WHERE OrderID = @OrderID")
    mOrder.Parameters.Add(New CmdParameter("@OrderID", DbType.Int32, intOrderID))
    mOrder.GetAllRows()
Catch ex As Exception
    MsgBox("Error loading Order from database: " & ex.Message, MsgBoxStyle.Critical)
Exit Sub
End Try

' Load the values for the order onto the screen's controls...
If (mOrder.ORDERDATE(1) <> Date.MinValue) Then
    txtOrderDate.Text = Format(mOrder.ORDERDATE(1), "MM/dd/yyyy")
End If
If (mOrder.REQUIREDDATE(1) <> Date.MinValue) Then
    txtRequiredDate.Text = Format(mOrder.REQUIREDDATE(1), "MM/dd/yyyy")
End If
If (mOrder.SHIPPEDDATE(1) <> Date.MinValue) Then
    txtShippedDate.Text = Format(mOrder.SHIPPEDDATE(1), "MM/dd/yyyy")
End If
If (mOrder.FREIGHT(1) <> Decimal.MinValue) Then
    txtFreight.Text = Format(mOrder.FREIGHT(1), "#,##0.00")
End If
txtShipTo.Text = mOrder.SHIPNAME(1)
txtAddress.Text = mOrder.SHIPADDRESS(1)
txtCity.Text = mOrder.SHIPCITY(1)
txtRegion.Text = mOrder.SHIPREGION(1)
txtZipCode.Text = mOrder.SHIPPOSTALCODE(1)
txtCountry.Text = mOrder.SHIPCOUNTRY(1)
End If ' If we are creating a new Order, or reading in an existing order.

' Load the values for the Customer dropdown box...
Dim I As Integer
cboCustomer.Items.Add(New ComboItem(" ", " "))
For I = 1 To gCustomerList.RowCount
    cboCustomer.Items.Add(New ComboItem(gCustomerList.COMPANYNAME(I),
gCustomerList.CUSTOMERID(I)))
Next

```

```

If Is_Valid_Customer(mOrder.CUSTOMERID(1)) Then
    Select_Combo_Value(cboCustomer, mOrder.CUSTOMERID(1))
End If

' Load the values for the Employee dropdown box...
cboEmployee.Items.Add(New ComboBoxItem(" ", 0))
For I = 1 To gEmployeeList.RowCount
    cboEmployee.Items.Add(New ComboBoxItem(gEmployeeList.LASTNAME(I) & ", " &
gEmployeeList.FIRSTNAME(I), gEmployeeList.EMPLOYEEID(I)))
Next
If Is_Valid_Employee(mOrder.EMPLOYEEID(1)) Then
    Select_Combo_Value(cboEmployee, mOrder.EMPLOYEEID(1))
End If

' Hard code the choices for the Ship Via drop-down...
cboShipVia.Items.Add(New ComboBoxItem("UPS Ground", 1))
cboShipVia.Items.Add(New ComboBoxItem("Fed-X Overnight", 2))
cboShipVia.Items.Add(New ComboBoxItem("US Post Office", 3))
If Is_Valid_Shipping_Method(mOrder.SHIPVIA(1)) Then
    Select_Combo_Value(cboShipVia, mOrder.SHIPVIA(1))
End If

' Now load the Order Details object...
' Notice this is even being done with -1 as the Order # to initialize the empty
DataSet buffer.
Try
    ' Create the new ORDER_DETAIL object...
    mDetails = New ORDER_DETAILS(gSQLConn, ORDER_DETAILS.GetBaseSQL & " WHERE OrderID
= @OrderID")
    ' Add the parameter to fill in the OrderID into the SQL...
    mDetails.Parameters.Add(New SqlParameter("@OrderID", DbType.Int32, intOrderID))

    ' Fetch the Order row from the Database...
    mDetails.GetAllRows() ' no error checking in this layer, throw it back up to the
UI.
Catch ex As Exception
    MsgBox("Error reading order detail records: " & ex.Message)
    Exit Sub
End Try

' Format the DataGrid for the Order Details...
DataGrid1.CaptionVisible = False
DataGrid1.ReadOnly = False
DataGrid1.AlternatingBackColor = Color.LightCyan
DataGrid1.RowHeadersVisible = True

' Create custom Column Style objects to handle the columns...
Dim tsOrderDetails As New DataGridTableStyle
tsOrderDetails.MappingName = "data"
tsOrderDetails.RowHeadersVisible = False

' 0. Product ID...
Dim styleProductID As New CGridComboBoxStyle("ProductID", 300, _
HorizontalAlignment.Left, _
"Product", " ", _
ComboBoxStyle.DropDownList)
' Add a blank choice for when they are adding new items...

```

```

styleProductID.cgCombo.Items.Add(New ComboItem(" ", "-1"))
' Load all the product names into the drop-down in the grid...
For I = 1 To gProductList.RowCount
    styleProductID.cgCombo.Items.Add(New ComboItem(gProductList.PRODUCTNAME(I),
CStr(gProductList.PRODUCTID(I))))
Next

' 1. Unit Price...
Dim styleUnitPrice As New AlternatingColorEditableColumnStyle
styleUnitPrice.MappingName = "UnitPrice"
styleUnitPrice.HeaderText = "Price"
styleUnitPrice.Width = 50
styleUnitPrice.ReadOnly = False
styleUnitPrice.NullText = ""
styleUnitPrice.Format = "####0.00"
styleUnitPrice.TextBox.TextAlign = HorizontalAlignment.Right

' 2. Quantity...
Dim styleQuantity As New AlternatingColorEditableColumnStyle
styleQuantity.MappingName = "Quantity"
styleQuantity.HeaderText = "Qty"
styleQuantity.Width = 50
styleQuantity.ReadOnly = False
styleQuantity.NullText = ""
styleQuantity.TextBox.TextAlign = HorizontalAlignment.Right

' 3. Discount...
Dim styleDiscount As New AlternatingColorEditableColumnStyle
styleDiscount.MappingName = "Discount"
styleDiscount.HeaderText = "Discount"
styleDiscount.Width = 60
styleDiscount.ReadOnly = False
styleDiscount.NullText = ""
styleDiscount.Format = "##0.0"
styleDiscount.TextBox.TextAlign = HorizontalAlignment.Right

' Add the style columns to the grid style object...
tsOrderDetails.GridColumnStyles.Add(styleProductID)
tsOrderDetails.GridColumnStyles.Add(styleUnitPrice)
tsOrderDetails.GridColumnStyles.Add(styleQuantity)
tsOrderDetails.GridColumnStyles.Add(styleDiscount)
DataGrid1.TableStyles.Add(tsOrderDetails)

' Refresh the DataGrid with the detail records...
Dim dView As New DataView(mDetails.DataSet.Tables("data"))
DataGrid1.DataSource = dView

' Disable the Microsoft Access style Append Row mode (extra blank row appearing at
end of grid!)...
DisableAppendRow(DataGrid1)

blnChangesMade = False
txtOrderDate.Focus()

End Sub

```

As you can see, if the user clicks on the **New Order** button on the browse window, it will open this window and pass a -1 to this window for the Order ID. This is the signal to this window that a new order is being created. The event above checks for this, and initializes an empty DataSet buffer for the order. Otherwise, if the user is opening an order that already exists in the database, it uses a DataHandler object to load the parent ORDER record, and all of the ORDER ITEM records as well into the DataGrid control at the bottom of the window.

After the user has made some changes to the order, the **Save** button loads up all the data back into the DataSet buffer, and the Update method is called for both the Employee object as well as the Details object:

```
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnSave.Click
    ' Save the changes...

    ' Make sure that there are no duplicate or NULL product ID's selected on the grid...
    Dim objList As New SortedList
    Dim dRow As DataRow
    Dim intProductID As Integer
    Dim I As Integer
    For I = 1 To mDetails.RowCount
        If (mDetails.PRODUCTID(I) = Integer.MinValue) Then
            MsgBox("You have at least 1 Order Item that you have not selected the Product
Type for yet!" & vbCrLf & "Changes can not be Saved!", MsgBoxStyle.Critical, "Problem:")
            Exit Sub
        End If
        intProductID = mDetails.PRODUCTID(I)
        If (intProductID = -1) Then
            MsgBox("You have at least 1 Order Item that you have not selected the Product
Type for yet!" & vbCrLf & "Changes can not be Saved!", MsgBoxStyle.Critical, "Problem:")
            Exit Sub
        End If
        If objList.ContainsKey(intProductID) Then
            Dim strProductName As String = Get_Product_Name(intProductID)
            MsgBox("You have the following Product Name selected more than once (this is not
allowed): " & strProductName & vbCrLf & "Changes can not be Saved!",
MsgBoxStyle.Critical, "Problem:")
            Exit Sub
        Else
            ' Go ahead and add the Product ID to the sorted array list...
            objList.Add(intProductID, intProductID) ' Key & Value are the same for this
list.
        End If
    Next

    ' Singal failure unless we make it all the way thru the Save procedures to the end.
    blnSaveOK = False
    AppFrame.StatusBar1.Text = "Saving Changes..."
    AppFrame.Cursor = Cursors.WaitCursor

    ' Load the updated values back into the Order object...
    ' Order Date...
    If (Not IsDate(txtOrderDate.Text)) Then
        AppFrame.StatusBar1.Text = "Ready"
```

```

    AppFrame.Cursor = Cursors.Default
    MsgBox("Invalid Order Date given.", MsgBoxStyle.Critical)
    Exit Sub
Else
    mOrder.ORDERDATE(1) = CDate(txtOrderDate.Text)
End If
' Required Date...
If MyEmpty(txtRequiredDate.Text) Then
    mOrder.REQUIREDDATE(1) = Date.MinValue
ElseIf (Not IsDate(txtRequiredDate.Text)) Then
    AppFrame.StatusBar1.Text = "Ready"
    AppFrame.Cursor = Cursors.Default
    MsgBox("Invalid Date Required given.", MsgBoxStyle.Critical)
    Exit Sub
Else
    mOrder.REQUIREDDATE(1) = CDate(txtRequiredDate.Text)
End If
' Shipped Date...
If MyEmpty(txtShippedDate.Text) Then
    mOrder.SHIPPEDDATE(1) = Date.MinValue
ElseIf (Not IsDate(txtShippedDate.Text)) Then
    AppFrame.StatusBar1.Text = "Ready"
    AppFrame.Cursor = Cursors.Default
    MsgBox("Invalid Date Shipped given.", MsgBoxStyle.Critical)
    Exit Sub
Else
    mOrder.SHIPPEDDATE(1) = CDate(txtShippedDate.Text)
End If
' Customer ID...
Dim objComboItem As ComboItem = CType(cboCustomer.SelectedItem, ComboItem)
If (Not (objComboItem Is Nothing)) Then
    mOrder.CUSTOMERID(1) = objComboItem.Value
End If
' Employee ID...
If (Not (objComboItem Is Nothing)) Then
    objComboItem = CType(cboEmployee.SelectedItem, ComboItem)
    mOrder.EMPLOYEEID(1) = objComboItem.NumericValue
End If
' Ship Via...
objComboItem = CType(cboShipVia.SelectedItem, ComboItem)
If (Not (objComboItem Is Nothing)) Then
    mOrder.SHIPVIA(1) = objComboItem.NumericValue
End If
' Freight...
If MyEmpty(txtFreight.Text) Then
    mOrder.FREIGHT(1) = Decimal.MinValue
ElseIf (Not IsNumeric(txtFreight.Text)) Then
    AppFrame.StatusBar1.Text = "Ready"
    AppFrame.Cursor = Cursors.Default
    MsgBox("Invalid Freight given.", MsgBoxStyle.Critical)
    Exit Sub
Else
    mOrder.FREIGHT(1) = CDb1(txtFreight.Text)
End If
' Ship To name...
mOrder.SHIPNAME(1) = Trim(txtShipTo.Text)
' Address...

```

```

mOrder.SHIPADDRESS(1) = Trim(txtAddress.Text)
' City...
mOrder.SHIPCITY(1) = Trim(txtCity.Text)
' Region (State)...
mOrder.SHIPREGION(1) = Trim(txtRegion.Text)
' Zip Code...
mOrder.SHIPPOSTALCODE(1) = Trim(txtZipCode.Text)
' Country...
mOrder.SHIPCOUNTRY(1) = Trim(txtCountry.Text)

' Now save the changes to the Order and Order details as 1 transaction...
Try
    gSQLConn.BeginTransaction()

    ' First check to see if we are saving a brand new Order, signaled by OrderID = -1.
    ' If we are, then take special steps to generate the next available OrderID
first...
    If (intOrderID = -1) Then
        Dim strSQL As String = "SELECT MAX(OrderID) FROM ORDERS"
        Dim intMaxID, intNewID As Integer
        intMaxID = gSQLConn.GetIntegerSQLResult(strSQL)
        If (intMaxID = Integer.MinValue) Then
            intMaxID = 0
        End If
        ' Assign the new value to the Order ID...
        intNewID = intMaxID + 1
        mOrder.ORDERID(1) = intNewID
        ' Also, update all the records in the Details to the new Order ID...
        For I = 1 To mDetails.RowCount
            mDetails.ORDERID(I) = intNewID
        Next
    End If

    ' Now save the Order and modified, created Order detail row information...
    mOrder.Update()
    mDetails.Update()

    gSQLConn.CommitTransaction()
Catch ex As Exception
    gSQLConn.RollbackTransaction()
    AppFrame.StatusBar1.Text = "Ready"
    AppFrame.Cursor = Cursors.Default
    MsgBox("Error during update: " & ex.Message, MsgBoxStyle.Critical)
End Try

' If we just saved a new order, update the display box for the OrderID and the
instance variable...
If (intOrderID = -1) Then
    intOrderID = mOrder.ORDERID(1)
    txtOrderID.Text = CStr(intOrderID)
End If

' Reset the changes made indicator...
blnChangesMade = False
blnSaveOK = True

' Refresh Order Browse Window if it is currently open...

```

```

Dim objBrowseOrdersWin As BrowseOrdersWindow
For Each objForm As Form In AppFrame.MdiChildren
  If (objForm.Name = "BrowseOrdersWindow") Then
    objBrowseOrdersWin = CType(objForm, BrowseOrdersWindow)
    objBrowseOrdersWin.Load_Data(False)
    For I = 0 To objBrowseOrdersWin.DataGrid1.VisibleRowCount - 1
      If (objBrowseOrdersWin.DataGrid1.Item(I, 1) = intOrderID) Then
        objBrowseOrdersWin.DataGrid1.CurrentRowIndex = I
        Dim dgcell As New DataGridViewCell(objBrowseOrdersWin.DataGrid1.CurrentRowIndex,
0)
        objBrowseOrdersWin.DataGrid1.CurrentCell = dgcell

objBrowseOrdersWin.DataGrid1.Item(objBrowseOrdersWin.DataGrid1.CurrentRowIndex, 7) = "Y"
      Exit For
    End If
  Next
End If
Next
End If
Next

AppFrame.StatusBar1.Text = "Ready"
AppFrame.Cursor = Cursors.Default
End Sub

```

In the **Save** button logic above, the program goes through the trouble of using a Sorted list class to make sure there are no duplicate product names selected on the list of order items. It has to do this because the Product ID is part of the primary key for the Order Items table, and you would get a database error if you tried to save the record with the duplicate Product ID listed.

Next, you see a bunch of validation going on, such as date field validation, numeric validation, etc, and then if everything is OK, it loads the updated information into the DataSet buffer.

Notice that no copying of information from the Order Items grid back to the DataSet is necessary, because the DataGrid is directly wired to the DataSet buffer, so all updates that the user makes are directly updating the DataSet buffer along the way.

As you can see, the entire update block is wrapped in a transaction block (i.e. – BeginTransaction(), CommitTransaction(), RollbackTransaction()).

Also, after the update is successful, the browse Order window is updated with the new information as well.

5.3 Summary

There was a lot of code presented in this in-depth sample program. Much of the code presented was in support of the presentation layer (user interface) portions of the program. At Creative Data Technologies, we are selling the DataLayer.NET product that targets making the coding of the database layer of your applications easier.

Please note that although we did include all of this presentation layer source code as an aid to help you create nicer looking database applications, it is not within the scope of our technical support venue to address questions in this area. Therefore, on our technical support system, we are limited to discussions of our DataLayer.NET components (DataConnection and DataHandler classes).



Appendix A – License Activation

A.1 Overview of how the License Activation System works

We won't spend too much time boring you with this information, I promise. The bottom line is that the DataLayer.NET library is intellectual property that took a significant amount of time and effort to put together by a number of people. We do not intend to allow people to give out our CD-KEY numbers so that our program can be used all over the place without monetary compensation to us. That would be neither fair nor legal.

Because we are developers like you, we want to be as flexible as possible, and give you as broad of a usage range as possible to the people who support us by purchasing the library.

In order to meet this challenge, we have implemented a License Activation plan that allows you to install and actively use the software on up to three computers at a time. This means, as a single developer, you can install the DataLayer.NET library on your computer at work, your computer at home, and even your laptop computer if needed. Furthermore, we made the License Activation engine an interactive one where you can dynamically move the three activations you are allowed among any machines you wish. This means you can install the software on more than three machines, but only three machines can be actively used at the same time. This is all done using the DataLayer.NET Code Generator Program to activate and deactivate your various machines.

When you run the Code Generator Program for the very first time on any machine, you will see the following message box that warns you will need to activate your license on your machine:



When you pull up the **Tools** menu and select the **Activate License for this Machine** option, you will see the following window:



For the CD-Key, you can copy and paste the CD-KEY that was emailed to you when you downloaded the DataLayer.NET software. Also, enter in the password that you used when you created your account while downloading the software.

For the Machine Name, you can enter in any descriptive name that will help you remember which machines you have already installed the software. By default, it will populate the field for you with the machine's TCP/IP network name. You can change it to anything you like. The purpose of entering a descriptive machine name here is so that you can recognize the machine's name on your list when you go back in later possibly needing to temporarily de-activate a machine to move one of your three activations to a different machine.

After you click on the Activate License button, you should get the following confirmation message:



If you have any trouble at all getting it activated, please contact techsupport@creativdatatech.com

You can repeat the steps above to activate up to three machines for using the DataLayer.NET software.

A.2 Deactivating Machines

Later on, should you need to use the software on a fourth machine, use the following procedure to de-activate one of your machines to free up an activation for the new machine:

Run the Code Generator on the new machine (or one of your old machines, it really doesn't matter which machine you choose to perform deactivations from). Go to the **Tools** menu and choose the **Select Machines to de-activate** menu item. You will see the following window:

Select Machines to De-activate...

Note: You are allowed to have up to 3 computers activated at any one time with this license. After that, you must de-activate a machine in order to use your license on another machine.

Step 1: Enter your credentials and retrieve your list of Licenses

Please enter your License CD-Key: Password:

XXXXXXXXXXXXXXXXXXXX Password: ***** Retrieve List of Active Machines

Step 2: Select the License(s) to De-activate:

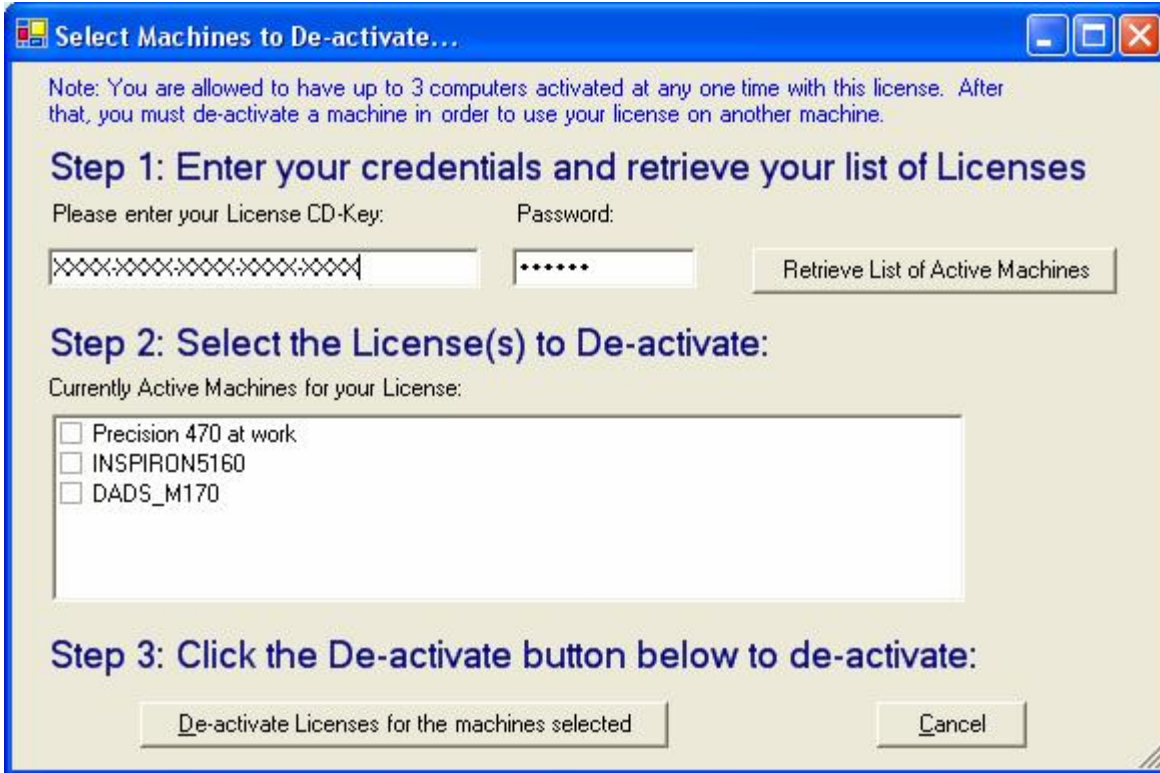
Currently Active Machines for your License:

[Empty list box]

Step 3: Click the De-activate button below to de-activate:

De-activate Licenses for the machines selected Cancel

If you are running the program on a machine that has already been activated, it will automatically fill in the CD-KEY information for you. After you enter your CD-KEY and password, click on the button labeled **Retrieve List of Active Machines**. You will see the list of currently activated machines for your license account:



Next, simply click the checkbox next to one or more of the licensed machines that you would like to deactivate, and then click the **De-activate Licenses** button. After you perform this operation, you should be able to activate your new machine as desired.

Notes about deactivating machines: When you deactivate a machine, you do not have to uninstall the DataLayer.NET software from the machine. The only thing it does is disable the DataLayer.NET Code Generator Program from working on deactivated machines.