# CoDAL (C# Data Access Layer)

## Introduction

CoDAL is a code generator that creates C# data access code to interface with a SQL Server database. These C# classes are inherited by business classes. The business classes extend the functionality of the base classes to add “business logic” such as data validation and calculated attributes. In many cases, if the SQL Server database is designed correctly, the hand coded business classes will contain very little code.

## A Simple Example

Many systems will have an account table that may look something like this:

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Required** |
| AccountID | Int (identity) | Yes |
| AccountName | VarChar(50) | Yes |
| Balance | Money | Yes |
| LastStatement | DateTime | No |

A simple Account business class would have at minimum the following:

* Properties for each of the fields.
* Insert, update and delete methods for making changes to the database.
* Static methods to retrieve one or more rows from the table and create Account objects.

Once this class is created, it can be used in various ways to work with accounts. For example, a web page used to add an account may contain the following C# code:

Account objNewAccount = new Account;

objNewAccount.AccountName = Request[“AccountName”]

objNewAccount.Balance = 0

objNewAccount.Insert()

A web page that displays account information may contain the following code.

Account objAccount = Account.GetItem(Request.QueryString[“AccountID”]

txtAccountID.Text = objAccount.AccountID

txtAccountName.Text = objAccount.AccountName

txtBalance.Text = objAccount.Balance

txtLastStatement.Text = objAccount.LastStatement

Writing code using the Account business class is much easier than the alternative of writing all the plumbing code using SqlConnections, SqlCommands and SqlDataReaders.

## Using CoDAL

CoDAL consists of a simple executable (codal.exe) designed to be used on the command line. There is no GUI. This makes it easy to use in batch files.

The code generator expects exactly 4 parameters to be passed in. These are:

1. The server on which SQL Server is installed.
2. The database that code should be generated for.
3. The folder where the generated classes should be saved.
4. The folder where the “skeleton” classes should be placed (see the section on sub classes below).

A call to the code generator may look like this:

CodeGenerator.exe localhost Northwind “c:\project\generated” “c:\project”

## Base Class Features

A base class is generated for each table in the database. It consists of the following regions.

### Enumerations

Each base class will contain at least one enumeration that maps the column names of a table to the column ordinals. In our Account table example, the AccountBase class would contain the following code:

#region Enumerations

public enum Columns : byte

{

AccountID = 0,

AccountName = 1,

Balance = 2,

LastStatement = 3

}

#region

This enumeration is mainly used internally, so that the SqlDataReader can be used with column ordinals instead of field names. This provides a healthy performance advantage.

### Fields

A protected field is generated for each column in the table. The SQL Server data types are mapped to C# data types in the following way:

|  |  |
| --- | --- |
| **SQL Server Data Type** | **C# Data Type** |
| Int | int |
| SmallInt | short |
| TinyInt | byte |
| Char | string |
| VarChar | string |
| Text | string |
| DateTime | DateTime |
| SmallDateTime | DateTime |
| Money | decimal |
| SmallMoney | decimal |
| Decimal | decimal |
| Bit | bool |

Note that the other SQL Server data types are not supported at this time, although all will be supported in a future version.

Additional protected fields are created for each foreign key where this table is the child table. The type of each of these fields is a business class. For example, if an InvoiceDetail table has a foreign key on InvoiceHeaderID to the InvoiceHeader table, then a field will be created of type InvoiceHeader. This allows an InvoiceDetail object to be used in the following way:

InvoiceDetail objInvoiceDetail = InvoiceDetail.GetItem(x)

string BillTo = objInvoiceDetail.InvoiceHeader.BillTo

Note that the InvoiceHeader object is created within the InvoiceDetail object only when needed.

All of these fields are read and set using public properties (see below).

### Properties

Properties are generated to allow the protected fields to be read and set. This is a much better alternative to just making the fields public as it allows data validation on writes and caching on reads.

### Constructors

Each base class will contain 2 constructors. The 1st constructor has no parameters and is used when a new object is to be created. Each of the fields is set to a value which represents “null”.

The 2nd constructor takes a SqlDataReader as a constructor and is used (mainly internally) to create an object for each row as a SqlDataReader is traversed.

### Methods

Each base class will contain 4 methods. These are Insert, Update, Delete and Validate. The Insert, Update and Delete have obvious uses. The following example created a new account, updates it and then deletes it.

Account objAccount = new Account

objAccount.AccountName = “Ben’s account”

objAccount.Insert()

objAccount.AccountName = “Ben’s wife’s account”

objAccount.Update()

objAccount.Delete()

The Validate method is used to validate the fields that have been set. The base class Validate method checks that all required fields have a value and that none of the strings are too long. The Validate method can be overriden to allow extremely complex data validation. The Validate method throws a ValidationException if there are any problems. The Insert and Update methods call the Validate method before any attempt is made to save the record to the database.

### Retrieval Methods

Retrieval methods are used to create one or more instances of the classes from the database. A retrieval method is created for each stored procedure in the database that matches a certain pattern. In our account example, the code generator looks for stored procedures that start with Account\_GetItem or Account\_GetList. The GetItem stored procedures must return one row that is used to create a single instance. The GetList stored procedures return 0 or more rows that are used to create a collection of instances (more on the collections later).

### Blank Values

This region contains a set of constants that represent the “blank” value for each field. For example, a constant called BlankBalance is created for our Balance field in the Account table. The internal value for the “blank” value is decimal.MinValue. This value is converted to null when the record is saved to the database (although in our case the Balance is a required field and the Validate method would return a ValidationException). The nice thing is that the user of the class doesn’t have to remember to use decimal.MinValue and can use Account.BlankBalance instead.

## Sub Class Features

Each base class must be sub classed. The sub class is the class that is actually used by the developer (the base class is abstract and cannot be instantiated). The sub class is intended to be hand coded and contains the “business logic” such as complex validation and calculated properties. For example, in our Account example, the developer might want to override the Validate method to add code to make sure that the LastStatement date is not greater than today.

Even though the sub class is intended to be hand coded, the code generator will generate skeleton code for the sub class. This is a one time generation and the subsequent hand coding will not be overwritten. Note that this is why the code generator requires 2 paths – the path for the classes that are totally generated and the path for the classes that are initially generated and subsequently hand coded.

## Collection Class Features

The code generator creates a collection class for each of the tables in the database. This allows a strongly typed collection of business objects to be created. In our Account example, the code generator generates a class called AccountCollection. The example below shows how an AccountCollection is retrieved using a retrieval method of the base class (see above) and then iterated to output a list of accounts.

AccountCollection ac = Account.GetList();

foreach (Account a in ac)

{

Response.Write(“<tr><td>” + a.AccountName + “</td></tr>”);

}

## Other Generated Classes

The code generator creates a few other necessary classes:

### RecordNotFoundException

The RecordNotFoundException is thrown by the base class “GetItem” retrieval methods where no record is found in the database.

### ValidationException

The ValidationException is thrown by the base class Validation method when the record is not valid.

### Queries

Sometimes it is too expensive to create a large collection of business objects just to display a list of data. Instead, it would be better to retrieve the data using a SqlDataReader and allow the developer to iterate directly through the SqlDataReader. However, this means that the developer has to write all the plumbing code using SqlConnections and SqlCommands. To keep life simple for the developer, the code generator creates a Queries class with wrapper methods to call any stored procedure beginning with “Query\_”. These methods return a SqlDataReader. For example, if the developer wants to show a report which contains a list of all accounts (AccountID + AccountName), a simple call to the generated wrapper method returns a SqlDataReader which can be iterated through to display the list:

SqlDataReader dr = Queries.GetAccountListForReport()

while (dr.Read())

{

Response.Write(“<tr><td>” + dr.Item[“AccountID”] + “</td><td>” +

dr.Item[“AccountName”] + “</td></tr>”;

}

This mechanism is really a way of bypassing the business objects for performance purposes and should only be used when necessary.