ADOX Extensions For DDL And Security

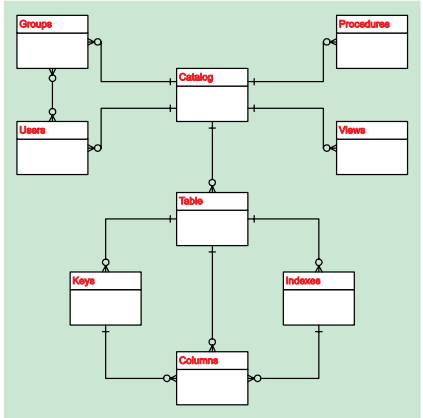
by Guy Smith-Ferrier

ADOX is an additional set of interfaces for ADO. Its purpose is straightforward: to allow programmers to handle the creation, maintenance and security of databases programmatically. Traditionally, these problems have been solved by using SQL Data Definition Language (DDL) and SQL Data Control Language (DCL). Although DDL and DCL are initially very simple languages (using `CREATE`, `ALTER`, `DROP`, `GRANT` and `REVOKE` commands) the exact syntax used by vendors varies more widely than that of SQL Data Manipulation Language (`SELECT`, `INSERT`, `UPDATE`, `DELETE`). So, whereas ADO itself can be viewed very broadly as a replacement for DML, ADOX can be viewed as a replacement for DDL and DCL.

ADOX was originally added to Microsoft Data Access Components (MDAC) in MDAC 2.1 (released March 1999). When Windows 2000 was released in February 2000, MDAC 2.5 (and therefore ADO2.5) was also released. Very little about ADOX has changed in ADO 2.5 and everything in this article is relevant to both releases.

Before we get stuck into ADOX, a warning is in order. If you are familiar with ADO, you'll be aware that although ADO defines feature sets and the programmer's interface to those feature sets, it does not follow that any particular OLE DB Provider supports any complete ADO feature set. This is still true for ADOX, but considerably more of ADOX is left unimplemented by most OLE DB Providers than you might expect. Of all the OLE DB Providers, the Jet 4.0 one (ie Access) has the most complete implementation and therefore that is what I will use here.

➤ *Figure 1*



ADOX Type Library

At the time of writing, the current release of Delphi is 5.01 (Delphi 5 Update Pack 1) and ADOX support is not included in ADOExpress. However, this presents only a small inconvenience as the ADOX type library is easily imported. To import the ADOX type library, select `Project | Import Type Library` and then choose the `Microsoft ADO Ext. 2.1 For DDL And Security (Version 2.1)` type library. If you have installed ADO 2.5 then the type library is called `Microsoft ADO Ext. 2.5 For DDL And Security (Version 2.5)`. If the type library isn't listed then click on `Add` and select MSADOX.DLL. You will need to change all of the class names in the dialog box to prevent clashes with existing Delphi classes (see Table 1).

Not all these changes are necessary: some are made simply for consistency. Give the `Palette Page` a suitable name, such as `ADOX`. Whether you check the `Generate component wrapper` checkbox or not depends on how you want to use ADOX in your apps. ADOX doesn't have any events to surface, so both choices have similar results. Once the type library has been imported you'll have an ADOX_TLB.PAS file in your Delphi Imports directory and be ready to use ADOX.

ADOX Object Model

Figure 1 shows the ADOX Object Model. In the centre of the diagram is `Catalog`, which is analogous to a database and holds collections of tables, views, procedures, groups and users.

Catalogs

Table 2 shows the properties of `Catalog`.

The `ActiveConnection` property is the ADO connection object which is used to connect to a data source. It is shown in brackets because it is only available as a property if you use the `Catalog`'s dispinterface or `IDispatch`. If you use the `Catalog` interface (which, like any interface, offers the best performance) you must use the `Get_ActiveConnection` and `Set_ActiveConnection` methods.

So, let's see what we can do with a `Catalog`. Drop a `TADOConnection` on a form, set its `ConnectionString` to the Northwind.mdb Access database (using the Jet 4 OLE DB Provider), set `LoginPrompt` to `False` and `Connected` to `True`. The code in Listing 1 uses `Catalog` to display all the table names in a memo.

The call to `CreateCOMObject` creates a new Catalog object. You could use `CreateOLEObject` or the proxy class which Delphi creates

➤ *Table 1*

Default Name	Changed Name
TTable	ADOXTable
TColumn	ADOXColumn
TIndex	ADOXIndex
TKey	ADOXKey
TGroup	ADOXGroup
TUser	ADOXUser
TCatalog	ADOXCatalog

➤ *Table 2*

(ActiveConnection)	The active Connection object
Groups	Collection of Group objects
Procedures	Collection of Procedure objects
Tables	Collection of Table objects
Users	Collection of User objects
Views	Collection of View objects

when it imports the type library (CoCatalog). The code iterates through the Catalog's collection of tables displaying the name of each table in Memo1. The list includes all tables (even system tables, but we'll come back to this later). Of course you could have achieved the same result using TADOConnection.OpenSchema, but this first step is small enough to work as a good starting point for ADOX.

Catalog is unusual in that it has a single method. Not because it only has *one* method, but because it has any methods *at all*. Apart from Collections, which I will come to in a moment, no other ADOX object has any methods at all. The method is called Create and is used to create a new database. It takes a single parameter which is a connection string identifying the database to be created and the OLE DB Provider which is responsible for creating it (see Listing 2).

There is no method in ADOX for deleting a database. The simplest solution is to use DeleteFile on the database file itself.

```
var
  Cat: Catalog;
  intTable: integer;
begin
  Cat:=CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(ADOConnection1.ConnectionObject);
  for intTable:=0 to Cat.Tables.Count - 1 do
    Memo1.Lines.Add(Cat.Tables.Item[intTable].Name);
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
// create an Access 2000 MDB
Cat.Create(
  'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Temp\Test.mdb');
// create an Access 97 MDB
Cat.Create(
  'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Temp\Test.mdb;'+
  'Jet OLEDB:Engine Type=4');
```

Collections

ADOX is a mass of collections: collections of tables, collections of columns, collections of indexes, etc. We'll take a quick look at what these collections have in common. Collections have just two properties which most programmers will be interested in: Count and Item. Count returns the number of items in the list and Item is an array which accepts either a number indicating the ordinal position within the collection or the name of the item. There is also a _NewENum function for anyone brave enough to make use of IENumVariant, but as this approach is usually only of benefit when the collection is a volatile one we'll skip it (database structures are, by their very nature, non-volatile).

In addition to the properties, dynamic collections (collections which can be added to and deleted from) have Append, Delete and Refresh methods. The Append method adds a new item to the list and the parameters are different for each type of collection. The Delete method accepts a number or a name of an item to remove

```
var
  Cat    : Catalog;
  intProp: integer;
  tbl    : Table;
  Prop   : ADOX_TLB.Property_;   // use ADOX Property instead of ADO Property
  strVal : string;
begin
  Cat:=CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(ADOConnection1.ConnectionObject);
  tbl:=Cat.Tables['Customers'];
  for intProp:=0 to tbl.Properties.Count - 1 do begin
    Prop:=tbl.Properties.Item[intProp];
    strVal:=Prop.Value;
    if (Prop.Attributes and adPropWrite) = adPropWrite then
      Memo1.Lines.Add(Prop.Name+'='+strVal)
    else
      Memo1.Lines.Add(Prop.Name+'='+strVal+' (Read only)');
  end;
end;
```

➤ *Above: Listing 3*

```
for intTable := 0 to
  Cat.Tables.Count - 1 do
    if Cat.Tables.Item[intTable].Type_='TABLE' then
      Memo1.Lines.Add(Cat.Tables.Item[intTable].Name);
```

➤ *Below: Listing 5*

from the collection. The `Refresh` method tells the collection to refresh itself because some external force has modified the collection without the collection's knowledge (typically this will be some dynamic SQL statement).

Tables

ADOX's `Table` describes a table in its entirety. Its properties are shown in Table 3.

Before I get too far into the `Table` itself I want to take a look at the `Properties` property as it is common to `Table`, `Column`, `Index`, `Group` and `View`. The `Properties` property contains what are called 'dynamic properties'. In essence `Properties` is a cargo slot where all the information which an OLE DB Provider needs beyond the basic ADOX object is placed. For example, Oracle has requirements which Access does not and vice versa. Rather than create a huge class which covers every eventuality, Microsoft solved the problem by having dynamic properties. Many of the properties truly are 'dynamic'. By this I mean that in one context an object supplied by an OLE DB Provider might report no properties at all but in a different context (where the properties are relevant) the same OLE DB Provider may report many properties.

The `Properties` property is a collection of `Property` objects. `Property` objects have `Attributes`, `Name`, `Type` and `Value` properties. The code in Listing 3 shows all the properties of a `Table` object. For an Access table the result is as shown in Listing 4.

There are a couple of points worth making about the code. Notice the definition of the `Prop` interface. Rather than declare it as type `Property` it is declared as `ADOX.Property_` (the underscore prevents a clash with a Delphi class). The ADO and ADOX type libraries both contain a `Property` object and if both are used in the same unit a distinction has to be made.

Secondly notice the `IF` line which tests the properties' `Attributes`. This is performing bitwise arithmetic with `adPropWrite` to see if it is contained in the `Attributes` (and therefore the property can be written to). What's interesting about this line is that this is the way all sets are

```
Temporary Table=0 (Read only)
Jet OLEDB:Table Validation Text=
Jet OLEDB:Table Validation Rule=
Jet OLEDB:Cache Link
Name/Password=0
Jet OLEDB:Remote Table Name=
Jet OLEDB:Link Provider String=
Jet OLEDB:Link Datasource=
Jet OLEDB:Exclusive Link=0
Jet OLEDB:Create Link=0
Jet OLEDB:Table Hidden In Access=0
```

➤ *Listing 4*

implemented in ADO (and COM in general). It is unusual only because as Delphi programmers we don't usually write code like this, as Delphi has an excellent implementation of enumerated types and sets of enumerated types. If ADOX support gets included in Delphi 6 then the resulting class is much

➤ *Table 3*

Columns	Collection Of Column Objects
DateCreated	Date of creation
DateModified	Date of last modification
Indexes	Collection of Index objects
Keys	Collection of Keys objects
Name	Name
ParentCatalog	Catalog which owns the table
Properties	Collection of Property objects
Type	Table type (normal, system or temporary)

➤ *Table 4*

"TABLE"	(Jet, SQL Server)
"SYSTEM TABLE"	(Jet, SQL Server)
"ACCESS TABLE"	(Jet)
"SYSTEM VIEW"	(SQL Server)
"VIEW"	(SQL Server)
"GLOBAL TEMPORARY"	

➤ *Table 5*

Attributes	Is the column fixed length and/or nullable ?
DefinedSize	Size of a column if the column is variable length
Name	Name
NumericScale	Scale of a column if the column is numeric
ParentCatalog	Catalog which owns the column
Precision	Precision of a column if the column is numeric
Properties	Collection of Property objects
RelatedColumn	Name of the column in a related table (if column is part of a Key)
SortOrder	Ascending or descending sort order
Type	Data type

```
procedure TForm1.btnStructClick(Sender: TObject);
var
  Cat: Catalog;
  tbl: Table;
begin
  Cat:=CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(
  ADOConnection1.ConnectionObject);
  tbl:=Cat.Tables['Customers'];
  Memo1.Lines.Add(TableToSQLCREATETABLE(tbl));
end;
function TableToSQLCREATETABLE(tbl: Table): string;
var
  intColumn: integer;
  Col: Column;
begin
  Result:='CREATE TABLE '+tbl.Name+' (';
  for intColumn:=0 to tbl.Columns.Count - 1 do begin
    Col:=tbl.Columns.Item[intColumn];
    Result:=Result+Col.Name+' '+ColumnToSQLDataType(Col);
    if intColumn <> tbl.Columns.Count - 1 then
      Result:=Result+', ';
  end;
  Result:=Result+')';
end;
function ColumnToSQLDataType(C: Column): string;
begin
  Result:=IntToStr(C.Type_);
  // this list is only a subset of the full list
  case C.Type_ of
    adDate    : Result:='DATE';
    adDouble, adCurrency, adSingle : Result :=
      'NUMERIC('+IntToStr(C.Precision)+','+IntToStr(C.NumericScale)+')';
    adInteger: Result := 'INTEGER('+IntToStr(C.Precision   )+')';
    adVarChar: Result := 'VARCHAR('+IntToStr(C.DefinedSize )+')';
    adWChar   : Result := 'VARCHAR('+IntToStr(C.DefinedSize )+')';
  end;
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
procedure TForm1.btnCreateTableClick(Sender: TObject);
var
  Cat: Catalog;
  tbl: Table;
begin
  Cat:=CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(ADOConnection2.ConnectionObject);
  tbl:=CreateCOMObject(Class_Table) as Table;
  tbl.Name:='Customers';
  tbl.Columns.Append('CUSTID'   , adInteger, 0 );
  tbl.Columns.Append('CUSTNAME', adVarWChar, 15);
  tbl.Columns.Append('PHONE'    , adVarWChar, 15);
  Cat.Tables.Append(tbl);  // create the table
end;
```

```
Col:=CreateCOMObject(Class_Column) as Column;
Col.Name:='CUSTNO';
Col.Type_:=adInteger;
Col.ParentCatalog:=Cat;
// this line will fail if ParentCatalog is not set
Col.Properties['Autoincrement'].Value:=True;
tbl.Columns.Append(Col, adInteger, 0);
```

➤ *Listing 8*

more likely to use sets, so the same line would be rewritten:

```
if axPropWrite in
  Prop.Attributes then
```

Getting back to `ADOX.Table`, we can use the `Table.Type` property to ensure that the list of tables we dumped in the memo only includes non-system tables: see Listing 5.

The `Type` property is actually `Type_` to prevent a clash with Delphi. It is compared against a string. The string is decided by the writer of the OLE DB Provider, so there is no hard and fast list. To give you an idea of what to expect Table 4 has a list of known strings.

Columns

Columns are essential to a table definition (they are also used in `ADOX.Index` and `ADOX.Key`). Their properties are shown in Table 5.

Care should be taken with some properties because they are context dependent. For example, the `SortOrder` property is only relevant when the `Column` is part of an `Index`.

The code in Listing 6 shows how you could use `Table` and `Column` to write an equivalent SQL `CREATE TABLE` command to rebuild the table. Note that this code is simply an example of mining the `Columns` property for data: the resulting SQL statement should not be considered complete.

Creating A Table

So far we've just read information about the database and created a new database. Now it's time to create a table. To create a table you need an ADO connection. You could use the ADO type library for this purpose or use `TADOConnection` in ADO Express. I'll take the latter approach. Assume a `TADOConnection` called `ADOConnection2` has been added to the form and the connection string has been set to the TEST.MDB database we created earlier. Set the `LoginPrompt` to `False` and `Connected` to `True`. Now add a button with the code in Listing 7.

Like all dynamic collections, the `Tables.Columns` property has an `Append` method, but the parameters are specific to the `Columns` collection. In this case the parameters are the field name, field type and field size. What is interesting about this process is that it is the act of appending the table to the `Catalog`'s list of tables which creates the table. This is typical of ADOX, in that ADOX is very immediate.

Altering an existing table is simply an extension of what we already know. The following code deletes the `PHONE` field and adds a new `FAX` field:

```
tbl.Columns.Delete('PHONE');
tbl.Columns.Append(
  'FAX', adVarWChar, 15);
```

Notice the two lines of code are two separate operations. First the `PHONE` is deleted and then the `FAX` is added. There is no way of collecting a set of changes to be applied in a single operation. Anyone who has used a little SQL DDL will be familiar with this behaviour.

Adding An AutoIncrement Column

Each of the `Table.Columns.Append` operations performed so far worked on columns which could

be defined simply by the three parameters passed to the Append method. Not all columns can be so easily defined. An AutoIncrement field is one example, so we'll look at creating Column objects from scratch. In the example shown in Listing 8 a new auto increment column called CUSTNO is added.

Notice the assignment of the ParentCatalog property. This is essential because this allows the Column to know what additional OLE DB Provider-specific properties it has (eg Autoincrement). Also notice that the Append method must still take all three parameters even though the first parameter is a Column which contains the information passed to the third parameter. This is the nature of using an interface. Unfortunately the Delphi COM trick of passing EmptyParam for parameters to be ignored does not work in ADOX and so real, duplicate, data must be passed instead.

Altering A Column

In SQL DDL there is no command to allow you to alter an existing column. Instead you have to drop the old column and add it back with the new definition. If the column has data in, it will be lost. To prevent this, a temporary column holding the data can be created for use whilst the original column is dropped.

In ADOX the problems and the solution are exactly the same. The code in Listing 9 alters a column. This code isn't foolproof. If the data needs to be truncated most DBMSs will generate an exception if the old data won't fit into the new column width. It also ignores any other column attributes, such as whether it accepts NULL, has CHECK constraints or is a foreign key.

Indexes

Now that we know how to create a table, creating an index follows the same principles. Table 6 shows the properties of an index. The example in Listing 10 creates both single field and composite indexes.

Keys

Keys allow us to define and interrogate referential integrity. ADOX supports three key types (identified by the Type property): primary, foreign and unique. Table 7 shows the Keys collection properties. The Type property is a KeyTypeENum and is one of the constants in Table 8.

So, to define a primary key for the Customers table consisting of just the CUSTNO field you would use the following code:

```
Cat.Tables[
  'Customers'].Keys.Append(
  'PKEY', adKeyPrimary,
  'CUSTNO', '', '');
```

The fourth and fifth parameters are used only by foreign keys.

The following code specifies that the CUSTNO column in the Orders table is a foreign key

```
procedure TForm1.btnAlterColClick(Sender: TObject);
begin
 AlterColumn(ADOConnection2, 'Customers', 'CUSTNAME', 30);
end;
procedure TForm1.AlterColumn(Conn: TADOConnection;
  strTable: string; strColumn: string; intSize: integer);
var
  Cat: Catalog;
  tbl: Table;
  Col: Column;
  DataType: DataTypeENum;
begin
  Cat:=CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(Conn.ConnectionObject);
  tbl:=Cat.Tables[strTable];
  Col:=tbl.Columns.Item[strColumn];
  DataType:=Col.Type_;
  tbl.Columns.Append('TMP', DataType, Col.DefinedSize);
  Conn.Execute('UPDATE '+strTable+' SET TMP='+strColumn);
  tbl.Columns.Delete(strColumn);
  tbl.Columns.Append(strColumn, DataType, intSize);
  Conn.Execute('UPDATE '+strTable+' SET '+strColumn+'=TMP');
  tbl.Columns.Delete('TMP');
end;
```

➤ *Above: Listing 9*

➤ *Below: Listing 10*

```
Cat.Tables['Customers'].Indexes.Append('CUSTNAMEINDEX1', 'CUSTNAME');
Cat.Tables['Customers'].Indexes.Append(
  'CUSTNAMEINDEX2', VarArrayOf(['CUSTNAME', 'PHONE']));
```

Clustered	Is the index a SQL Server clustered index ?
Columns	Collection of Column objects
IndexNulls	Indicates what happens to entries that contain nulls
Name	Name
PrimaryKey	Specifies whether the index is the primary key of the table
Properties	Collection of Property objects
Unique	Indicates whether the keys must be unique

➤ *Above: Table 6*

➤ *Below: Table 7*

Columns	Collection of Column objects
DeleteRule	Specifies what happens when a primary key is deleted
Name	Name
RelatedTable	The name of the foreign table
Type	The type of the key
UpdateRule	Specifies what happens when a primary key is updated

➤ *Table 8*

Constant	Value	Description
adKeyPrimary	1	The key is a primary key (and unique)
adKeyForeign	2	The key is a foreign key (and not unique)
adKeyUnique	3	The key is unique

Constant	Value	Description
adRINone	0	Deletes/updates are prevented
adRICascade	1	Deletes/updates are cascaded
adRISetNull	2	Deletes/updates set foreign key to NULL
adRISetDefault	3	Deletes/updates set foreign key to its default value

➤ *Above: Table 9*

➤ *Below: Table 10*

Feature	Jet	SQL Server	Oracle
Catalog.Create	Yes	No	No
Tables	Yes	Properties are read only	Properties are read only, Append and Delete are not supported
Views	Yes	Not supported	Append, Delete and Command not supported
Procedures	N/A	Append, Delete and Command not supported	Append, Delete and Command not supported
Keys	Yes	Append and Delete not supported	Append and Delete not supported
Indexes	Yes		Append and Delete not supported
Users	Yes	Not supported	Not supported
Groups	Yes	Not supported	Not supported

```
procedure TForm1.btnAddGroupsClick(Sender: TObject);
var
  Cat: Catalog;
  grp: Group;
begin
  Cat := CreateCOMObject(Class_Catalog) as Catalog;
  Cat.Set_ActiveConnection(ADOConnection2.ConnectionObject);
  grp:=CreateCOMObject(Class_Group) as Group;
  grp.Name:='Technical';
  Cat.Groups.Append(grp);
end;
```

➤ *Listing 11*

referencing the `CUSTNO` column of the `Customers` table:

```
Cat.Tables['Orders'].Keys.Append(
  'FKEY',adKeyForeign,'CUSTNO',
  'Customers','CUSTNO');
```

In this case the `DeleteRule` and `UpdateRule` properties would have their default values so deletions of the parent record in `Customers` and updates of the `Customers.CUSTNO` field would fail while there are records in `Orders` which reference them. Both `DeleteRule` and `UpdateRule` are `RuleENums` which can be any of the values in Table 9.

Procedures

The `Procedures` object theoretically allows stored procedures to be created, retrieved and dropped. Unfortunately this is a weak area of ADOX. The SQL Server and Oracle OLE DB Providers cannot create or delete stored procedures so the only solution is to go back to SQL `CREATE PROCEDURE` and `DROP PROCEDURE` (or whatever commands your dialect uses). Alternatively you could use SQL DMO (SQL Server Distributed Management Objects) to create, alter and drop SQL Server stored procedures.

Unsupported Features

As I mentioned at the start, ADOX is simply a specification, it is left to the author of the OLE DB Provider to decide whether a feature is actually implemented. Table 10 summarises the features which are not supported by the Jet, SQL Server and Oracle OLE DB Providers. As you can see from the table, Jet supports the majority of ADOX but SQL Server and Oracle have very large gaps in their support.

Security, Users And Groups

Of all of the areas of ADOX, security has the poorest support. Jet is the only provider which has support for `Users` and `Groups` and these collections are available only in Access 2000, not Access 97. Access security has to be set up before attempting to access the `Users` and `Groups` collections (accessing these collections for a database which does not have security generates an error). Unfortunately setting up Access security is a difficult process for the uninitiated.

ADOX allows us to interrogate the lists of users and groups and add, update and delete from them. The `ConnectionString` must be modified before a secure database can be opened, by setting the `Jet OLE DB:System database` argument (in the `All` page of the connection string editor) to the filename of the workgroup information file (the .mdw file). The code in Listing 11 adds a new group.

In ADOX each `User` in the `Users` collection can belong to a `Group` and each `Group` in the `Groups` collection can have many users. Both `User` and `Group` objects have extensive support to allow programmers to get and set any security permissions on any object using `GetPermissions` and `SetPermissions`.

Conclusion

ADOX attempts to insulate the programmer from the different dialects of SQL DDL and DCL by offering a type library of interfaces common to all DBMSs. Making use of this type library in Delphi is straightforward. Unfortunately true portability of this part of the application is still a little way away yet because complete support for ADOX is lacking in all but the Jet OLE DB Provider.

Guy Smith-Ferrier is Technical Director of Enterprise Logistics Ltd. (www.EnterpriseL.com), a training and development company specialising in Delphi which runs ADO courses. He can be contacted at gsmithferrier@EnterpriseL.com