OBJECTSTORE

ACTIVE TOOLKIT REFERENCE

RELEASE 6.0

May 1999

ObjectStore Active Toolkit Reference

ObjectStore Active Toolkit Release 6.0, May 1999

Object Design, Inc.
Twenty Five Mall Road
Burlington, MA 01803-4194

Contents

Preface

Purpose

The *ObjectStore Active Toolkit Reference* provides a reference on the Active Toolkit ActiveX and OLE DB programming interfaces to ObjectStore.

Audience

This book is for experienced Visual Basic, Visual Basic Script, Java, Java Script, or C++ developers who are developing applications that run under Windows NT or Windows 95/98 and want to access objects stored in an ObjectStore database through an ActiveX interface.

Scope

This book supports Release 6.0 of the ATK interface to ObjectStore Release 6.0. Information in this book assumes that ATK is installed and configured.

How This Book Is Organized

ATK and Inspector

When you define data views and instance formats using Inspector, ObjectStore saves them in the metaknowledge. ATK uses the metaknowledge when accessing data views and instance formats. See Chapter 1, ATK and the Inspector, on page 1.

ActiveX Server

ATK is an ActiveX server that provides access to any ObjectStore database. See Chapter 2, ATK ActiveX Server, on page 7.

ATK OLE DB

ATK is also an OLE DB provider. OLE DB is a standard interface for accessing data, and ATK provides access to any ObjectStore database through OLE DB. See Chapter 3, Active Toolkit OLE DB Provider, on page 15.

Configuration

The ATK installation program configures ATK to use the same metaknowledge as Inspector, and assigns several other default settings. If you reconfigure Inspector, you should update the corresponding ATK settings. You can also reconfigure ATK to

meet the specific needs of an application. See Chapter 4, Configuring ATK, on page 41.

ATK object model

The ATK object model consists of 17 COM classes. This chapter lists each class alphabetically by class name. Within the entry for each class, the class's properties and methods appear alphabetically. See Chapter 5, ATK Object Model, on page 51.

Sample Data

This document refers to sample applications and demonstration databases. If you installed ATK using the installation program defaults, you can find them in these directories:

Component	*Location*
ATK	**C:\odi\ATK6.0**
Sample Applications	**C:\odi\ATK6.0\Samples**
Demonstration Databases	**C:\odi\ATK6.0\DemoDBs**

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
Bold	Bold typeface indicates user input or code.
Sans serif	Sans serif typeface indicates system output.
Italic sans serif	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.
Italic serif	In text, italic serif typeface indicates the first use of an important term.
[ ]	Brackets enclose optional parameters.
{ *a* | *b* | *c* }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	An ellipsis indicates missing code that is not pertinent to the current example. In syntax lines, it indicates that the previous item can be repeated.

Internet Sources of More Information

Object Design

Object Design's site on the World Wide Web is the source for company news, white papers, and information about product offerings and services. Point your browser to **http://www.objectdesign.com/** for more information.

Other ObjectStore products

In addition to ObjectStore, the industry's leading object database, Object Design offers a comprehensive set of rapid development and enterprise integration tools. For information about these and other Object Design products, point your browser to **http://www.objectdesign.com/products/products.html**.

Product support

Object Design's support organization provides a number of information resources and services. Their home page is at **http://support.objectdesign.com/**. From the support home page, click Tech Support Information to learn about support policies, product discussion groups, and the different ways Object Design can keep you informed about the latest release information — including the web, **ftp**, and email services.

Training

If you are in North America, for information about Object Design's educational offerings, call 781.674.5320, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time. Outside these hours, call 800.706.2507.

If you are outside North America, call your Object Design sales representative.

Your Comments

Object Design welcomes your comments about ObjectStore documentation. Send your feedback to **support@objectdesign.com**. To expedite your message, begin the subject with **Doc:**. For example:

Subject: Doc: Incorrect message on page 76 of reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
ATK and the Inspector

Introduction

ATK uses the data views and instance formats that you define in Inspector and store in the metaknowledge.

In this chapter

This chapter presents the following topics:

ATK Metaknowledge and Inspector

What is metaknowledge?

Inspector *metaknowledge* contains data view definitions, including the query and ordering information set by the query's author, and other information about the database schema and the defined instance formats.

ATK provides read and write access, by means of the ATK ActiveX server or the ATK OLE DB provider, to data views you define in the Inspector. To do this, ATK must process the ObjectStore database's metaknowledge, which Inspector stores.

ATK's use of metaknowledge

Inspector uses the metaknowledge to display persistent instances, and ATK uses the Inspector metaknowledge to retrieve information about the available data views, instance formats, and schema structure. Thus, the Inspector metaknowledge logically becomes the ATK metaknowledge. Although ATK never modifies this metaknowledge, by using ATK you can customize the data view and instance format definitions and reuse Inspector-defined data views in different contexts.

Metaknowledge location

Inspector metaknowledge can reside in the file system, in the same inspected database, or in another ObjectStore database that contains only Inspector metaknowledge. The metaknowledge location is specified in the Inspector configuration.

ATK must be able to access the Inspector metaknowledge. When you install ATK, the installation program verifies that the Inspector is installed, examines the Inspector configuration to determine where the metaknowledge resides, and configures ATK with that location. Thus, ATK can access the data views and instance formats you define using Inspector.

To modify the ATK configuration, refer to Chapter 4, Configuring ATK, on page 41.

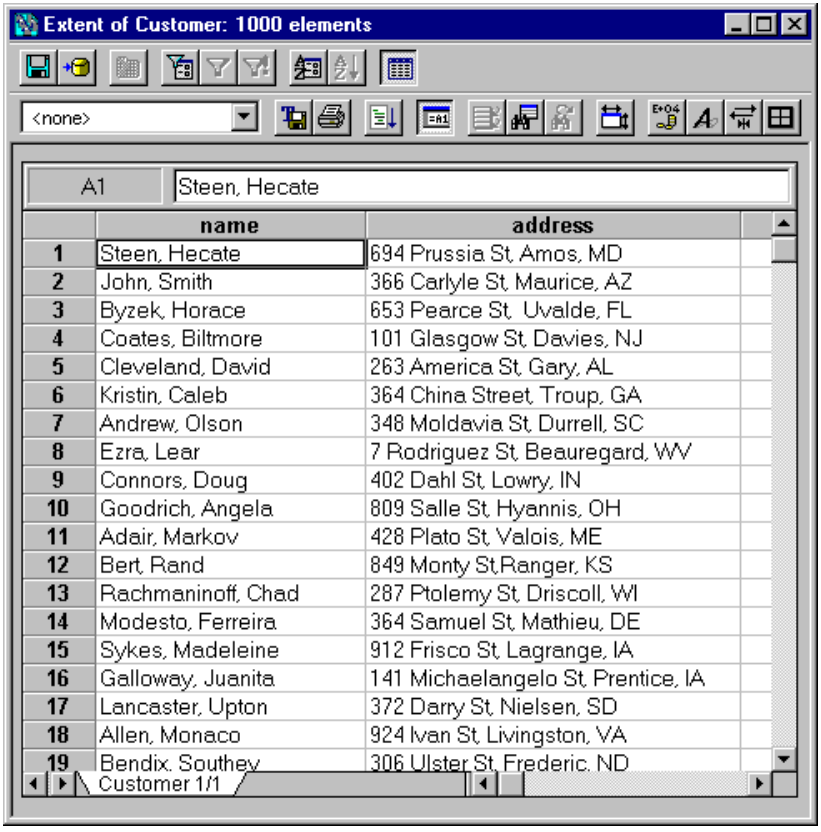
Accessing Data Views and Instance Formats from ATK

What is a data view?

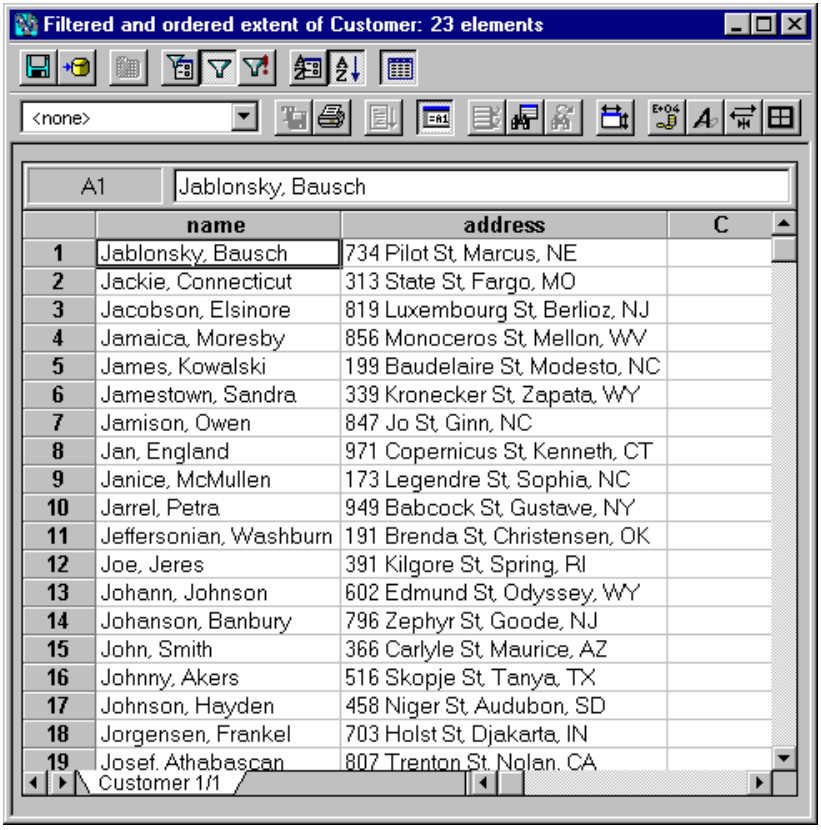
A *data view* is a tabular representation of a persistent ObjectStore collection that uses a specified instance format to display persistent instances. You can define a data view based on any of the following:

- Filtering options
- Ordering options
- Arrays
- User-defined extent methods

For example, suppose the class **Customer** has two data members, **name** and **address**. You can use Inspector to build a data view based on a collection of customers. A data view's *instance format* specifies the set of data members that must be displayed for an instance of the class in the data view. If you are displaying a table of instances, the instance format specifies what columns have to be displayed. This simple data view has an instance format of two columns that represent the name and address of each **Customer** object contained in the persistent collection.


Extent of Customer: 1000 elements
<none>
A1
Steen, Hecate
name
address
1 Steen, Hecate 694 Prussia St, Amos, MD
2 John, Smith 366 Carlyle St, Maurice, AZ
3 Byzek, Horace 653 Pearce St, Uvalde, FL
4 Coates, Biltmore 101 Glasgow St, Davies, NJ
5 Cleveland, David 263 America St, Gary, AL
6 Kristin, Caleb 364 China Street, Troup, GA
7 Andrew, Olson 348 Moldavia St, Durrell, SC
8 Ezra, Lear 7 Rodriguez St, Beauregard, WV
9 Connors, Doug 402 Dahl St, Lowry, IN
10 Goodrich, Angela 809 Salle St, Hyannis, OH
11 Adair, Markov 428 Plato St, Valois, ME
12 Bert, Rand 849 Monty St,Ranger, KS
13 Rachmaninoff, Chad 287 Ptolemy St, Driscoll, WI
14 Modesto, Ferreira 364 Samuel St, Mathieu, DE
15 Sykes, Madeleine 912 Frisco St, Lagrange, IA
16 Galloway, Juanita 141 Michaelangelo St, Prentice, IA
17 Lancaster, Upton 372 Darry St, Nielsen, SD
18 Allen, Monaco 924 Ivan St, Livingston, VA
19 Bendix, Southey 306 Ulster St, Frederic, ND
Customer 1/1

Defining a filter to list only those customers whose names start with the letter J, or defining an order that lists the customers alphabetically, makes this data view more complex.



	name	address	C
1	Jablonsky, Bausch	734 Pilot St, Marcus, NE	
2	Jackie, Connecticut	313 State St, Fargo, MO	
3	Jacobson, Elsinore	819 Luxembourg St, Berlioz, NJ	
4	Jamaica, Moresby	856 Monoceros St, Mellon, WV	
5	James, Kowalski	199 Baudelaire St, Modesto, NC	
6	Jamestown, Sandra	339 Kronecker St, Zapata, WY	
7	Jamison, Owen	847 Jo St, Ginn, NC	
8	Jan, England	971 Copernicus St, Kenneth, CT	
9	Janice, McMullen	173 Legendre St, Sophia, NC	
10	Jarrel, Petra	949 Babcock St, Gustave, NY	
11	Jeffersonian, Washburn	191 Brenda St, Christensen, OK	
12	Joe, Jeres	391 Kilgore St, Spring, RI	
13	Johann, Johnson	602 Edmund St, Odyssey, WY	
14	Johanson, Banbury	796 Zephyr St, Goode, NJ	
15	John, Smith	366 Carlyle St, Maurice, AZ	
16	Johnny, Akers	516 Skopje St, Tanya, TX	
17	Johnson, Hayden	458 Niger St, Audubon, SD	
18	Jorgensen, Frankel	703 Holst St, Djakarta, IN	
19	Josef, Athabascan	807 Trenton St, Nolan, CA	

You can name and save the view, and reload it later. If the collection changes, the data view changes accordingly.

Accessing Data Views from ATK

To access a data view defined in Inspector, use the ATK ActiveX server or ATK OLE DB provider and specify the name of the data view.

For example, to access a data view named **my_first_dataview** through the ATK ActiveX server, create an **IATKDataView** object to represent the view:

```
atkDataView = atkDatabase.GetDataView("my_first_dataview")
```

Using the ATK OLE DB provider, open a record set representing the same data view:

```
adoRecordSet.Open("my_first_dataview", adoConnection)
```

You can also use SQL statements with OLE DB:

```
adoRecordSet.Open("SELECT * FROM my_first_dataview",
adoConnection)
```

Note that if you use SQL statements, the data view name must follow SQL syntax rules.

Accessing Instance Formats from the ATK ActiveX Server

Using the ATK ActiveX server, you can directly access the instance formats defined for the classes contained in the database schema. An instance format can include navigation pointers and collections of other classes, which provide a powerful means of navigating among objects transparently and displaying the equivalent of relational SQL joins faster and more efficiently than SQL syntax.

When you create a data view in Inspector, you can save the data view template with a symbolic name. When you modify an instance format in Inspector, you usually modify the default instance formats associated with each class. However, the Inspector will save any modification to the template instance format using the symbolic name. The ATK ActiveX server accesses this name and displays data according to the specified instance format.

Refer to Chapter 2, Accessing ATK ActiveX Server from ASP Applications, in the *ObjectStore Active Toolkit Tutorial*, for an example of defining multiple instance formats for a specific class and accessing them from the ATK ActiveX server, and customizing a data view using the ATK ActiveX server or ATK OLE DB provider.

Chapter 2
ATK ActiveX Server

Introduction

ATK is an OLE Automation Server. You can create ActiveX objects to open and inspect databases on local and remote hosts. You can also integrate ATK with OSAX to modify a database, and gather information from ATK ActiveX error codes.

In this chapter

This chapter presents the following topics:

What Is the ATK ActiveX Server?

The *ATK ActiveX server* is ATK OLE Automation component that exposes the ATK object model to any ActiveX client programming environment. (These include Active Server Pages, Visual Basic, VBscript, Visual C++, Java, Visual J++, and JavaScript.)

Using the ATK Active X server

You can use the ATK ActiveX server to access and customize the tabular data views you define in Inspector. For a detailed description of the ATK object model, refer to Chapter 5, ATK Object Model, on page 51.

You can also navigate among persistent objects in an ObjectStore database or retrieve information about the database schema. The ATK ActiveX server supports the OLE Multithreaded Apartment (MTA) model architecture. For more information about threading models, refer to the Microsoft Knowledge Base (**http://support.microsoft.com/support/**).

The ATK ActiveX server is distributed as a DLL (for in-proc use), and as an executable (for simplified out-of-proc use).

Examples

For examples of applications that use the ATK ActiveX server, refer to Chapter 2, Accessing ATK ActiveX Server from ASP Applications, in the *ObjectStore Active Toolkit Tutorial*.

Accessing the ActiveX Server Using DCOM

The Distributed Component Object Model (DCOM) underlying ATK allows instances of ActiveX objects to be located and accessed remotely from the ActiveX server. Thus, an application can access the ATK ActiveX server running on a remote machine to open and inspect a remote database.

The DCOM protocol is essential for developing ActiveX controls that run on remote workstations and that access one or more ATK ActiveX servers. The grid control that is distributed with ATK can connect to remote ATK servers using DCOM.

To create an object that will run on a remote ATK ActiveX server, use **CoCreateInstanceEx** instead of **CoCreateInstance** and specify the host on which the ATK ActiveX server is running.

For more information

For detailed information about DCOM, refer to the Microsoft documentation. For an example of an application that uses DCOM, refer to Chapter 6, Using ATK ActiveX Server from DCOM, in the *ObjectStore Active Toolkit Tutorial*.

Integrating ATK and OSAX

What is OSAX?

The *ObjectStore ActiveX Interface* (*OSAX*) is an interface that complements ATK's access to ObjectStore databases. Using OSAX, you can encapsulate persistent C++ objects inside COM objects and create, read, update, or modify them.

The ATK and OSAX services can function independently of each other, or you can integrate them. When ATK and OSAX are integrated in one application, you can build object representations in OSAX based on ATK objects, and in ATK based on OSAX objects.

From ATK to OSAX

Use the **OSAXObjectStore** interface to build an OSAX object from an **ATKReference** object. **OSAXObjectstore** retrieves the information it needs from the ATK object and builds an **IX** object representing the persistent instance, in which **X** is the name of the class.

For example, suppose you have a database whose schema contains the **Customer** class, and an **ATKReference** object represents a **Customer** instance. You have built an OSAX interface on top of your classes, and OSAX provides the **CustomerClass** and **Customer** interfaces. Using Visual Basic, you could create a persistent **osaxCustomer** object:

```
Dim atkCustomer as IATKReference
Dim ObjectStore as IOSAXObjectStore
...
Dim osaxCustomer as ICustomer
osaxCustomer = ObjectStore.LoadATKReference(CCustomer,
atkCustomer)
```

After you created an **osaxCustomer** object, you could use the methods you exposed in the **Customer** interface to modify **osaxCustomer** as necessary.

From OSAX to ATK

Suppose you have a **Customer** object called **osaxCustomer**. You could build an **ATKReference** object representing the **osaxCustomer** instance by using this Visual Basic code:

```
Dim atkKernel as IATKKernel
...
Dim atkCustomer as IATKReference
atkCustomer = atkKernel.LoadOSAXObject(osaxCustomer)
```

Once you create the **atkCustomer** object, you can use it to access all of the features that ATK offers. For example, you can retrieve the data members in the instance, navigate a relation, build a table based on a collection that the object contains, retrieve information about classes associated with the object, and so on.

Retrieving Error Information

When ATK ActiveX server calls an ATK method and an error occurs, ATK returns an error code. The file **\include\atkkernel.h** lists all ATK error codes.

You can retrieve extended information about any error. These code samples in Visual Basic, VBScript, and Visual C++ show how to retrieve information about an error that occurs when ATK ActiveX server tries to open a database.

Visual Basic

```
    On Error GoTo handle_error
    Dim ObjectStore As New ATKKernel
    Dim OSDatabase As ATKDatabase
    Set OSDatabase = ObjectStore.OpenDatabase("mydb.db")
    Exit Sub
handle_error:
    TmpStr = "ATK Error# " & Hex(Err.Number)
    TmpStr = TmpStr & vbCrLf & "Generated by: " & Err.Source
    TmpStr = TmpStr & vbCrLf & "Description: " & Err.Description
    MsgBox TmpStr
```

VBScript in Active Server Pages

```
On Error Resume Next
set ATKKernel=Server.CreateObject("ATKKernel.Document")
set ATKDatabase=ATKKernel.OpenDatabase("mydb.db")
if Err.Number <> 0 then
    Response.Write("ATK Error# " & Hex(Err.Number) & "<BR>")
    Response.Write("Generated by: " & Err.Source & "<BR>")
    Response.Write("Description: " & Err.Description & "<BR>")
    Exit Sub
End If
```

Visual C++

```
IID_IATKKernel* pATK;

IID_IATKDatabase* pATKdb;
HRESULT hr=pATK->OpenDatabase("mydb.db", &pATKdb);
if(FAILED(hr)) {
    ISupportErrorInfo* pSEI;
    if(SUCCEEDED(
            pDisp->QueryInterface(IID_ISupportErrorInfo,(LPVOID*)&pSEI) &&
            pSEI->InterfaceSupportsErrorInfo(IID_IATKKernel)==S_OK)){
        IErrorInfo* pEI;
        GetErrorInfo(0,&pEI);
        char szBuffer[512];
        BSTR source,description;
        pEI->GetSource(&source);
        pEI->GetDescription(&description);
        wsprintf(szBuffer,"ATK Error# %X\nSource: %S\nDescription: %S",
            hr,source,description);
        MessageBox(NULL,szBuffer,"Error",MB_OK);
        SysFreeString(source);
        SysFreeString(description);
        pEI->Release();
    }
}
```

Chapter 3
Active Toolkit OLE DB Provider

Introduction

ATK can retrieve ActiveX objects from an ObjectStore database by means of the ObjectStore Active Toolkit OLE DB provider. You can access OLE DB providers directly, or through active data objects (ADO). ObjectStore Active Toolkit OLE DB also lets you use an ODBC data source, query with SQL syntax, and perform create, read, update, and delete operations using user-defined methods.

In this chapter

This chapter presents the following topics:

What Is OLE DB?

OLE DB is Microsoft Corporation's component database architecture — the fundamental component object model (COM) building block for storing and retrieving records. This set of interfaces for accessing and manipulating data provides data integration regardless of the data type. Microsoft's Open Database Connectivity (ODBC) data access interface is included in the OLE DB architecture, and continues to provide access to relational data.

You can access OLE DB providers through ActiveX Data Objects (ADO). ADO is a high-level object model that provides access to any OLE DB source. The ADO interface is similar to the Data Access Objects (DAO) interface and the Remote Data Objects (RDO) interface.

ObjectStore Active Toolkit OLE DB

The ObjectStore Active Toolkit OLE DB provider Release 6.0 is built on top of OLE DB version 2.0. When you install ObjectStore Active Toolkit, the installation program automatically registers ObjectStore Active Toolkit as an OLE DB provider, labeled **ObjectStore Active Toolkit OLE DB Provider**. This registration lets you access the ObjectStore Active Toolkit services through the standard interface provided by OLE DB or ADO.

Additional information about OLE DB and ADO

For details about the OLE DB object model, refer to the Microsoft OLE DB SDK.

To obtain a complete description of the ADO object model, consult the on-line documentation for Microsoft Visual Basic, Active Server Pages, or Visual C++.

For more information about OLE DB and ADO, browse the Microsoft web site at **http://www.microsoft.com/data**.

Navigation in ObjectStore ATK OLE DB

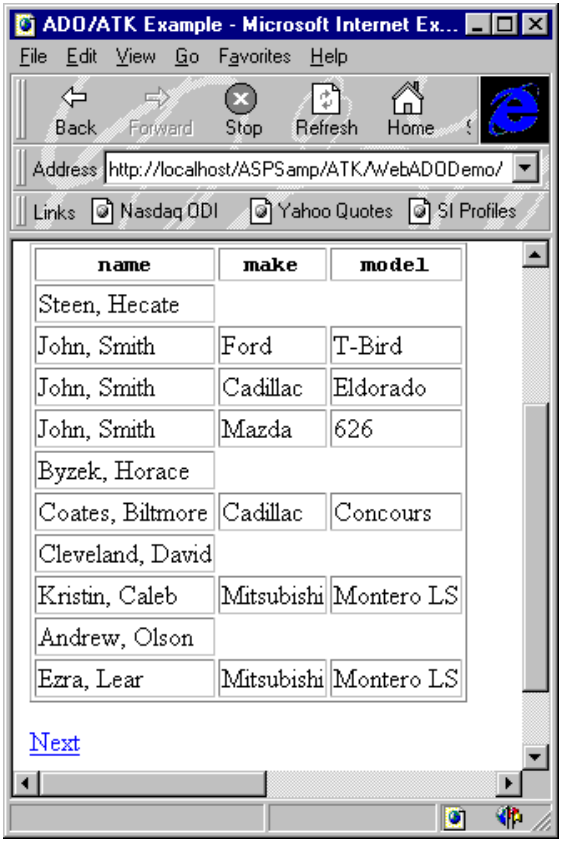
Navigating between objects is a common technique for exposing a tabular representation of a class hierarchy. For example, suppose an ObjectStore database contains the two classes **Customer** and **Vehicle.** The **Customer** class has a relationship that lists each **Vehicle** owned by a **Customer** and each **Vehicle** is linked to its **Customer**. (The sample database **carsdemo.db** contains this kind of schema.)

Implicit navigation

Using this collection of **Customer**s, you can build a data view in the Inspector that displays the name of each **Customer**, and the **make** and **model** of each **Vehicle** he or she owns.

When you access the **Vehicle** class by means of the **Customer** class, rather than accessing the **Vehicle** class directly, you are implicitly navigating the relationship between the **Customer** and the **Vehicle** classes.

Using ObjectStore Active Toolkit OLE DB, you can publish this data view in an HTML table.



This ASP code generated the HTML table. **RS** is the ADO **RecordSet** representing the collection of **Customer**s:

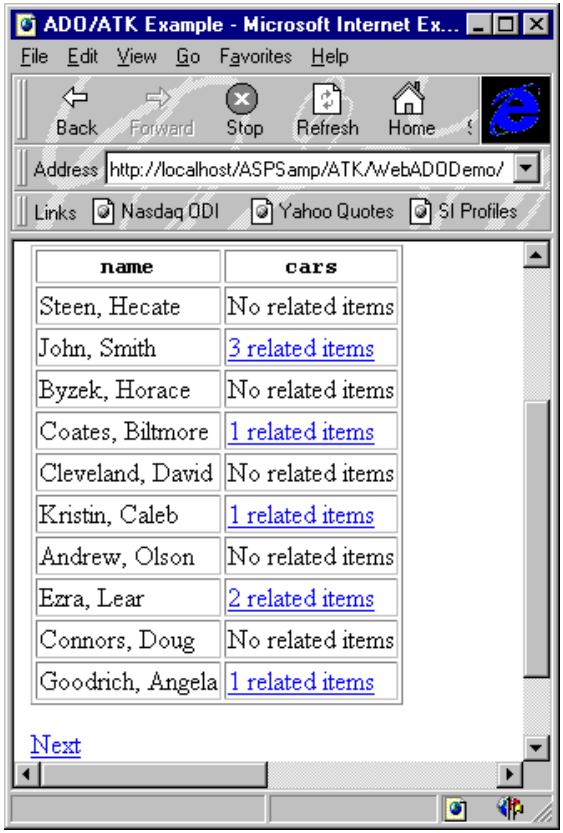
```
Response.Write("<TABLE BORDER=1>")
For Each colHeader in RS.fields
   Response.Write("<TH><B><PRE>" & colHeader.Name & "</PRE></B></TH>")
Next
Do While Not RS.EOF
   Response.Write("<TR>")
   For Each aField in RS.fields
      Response.Write("<TD>" & aField.Value & "</TD>")
   Next
   Response.Write("</TR>")
   RS.MoveNext
Loop
Response.Write("</TABLE>")
```

Explicit navigation with ADO

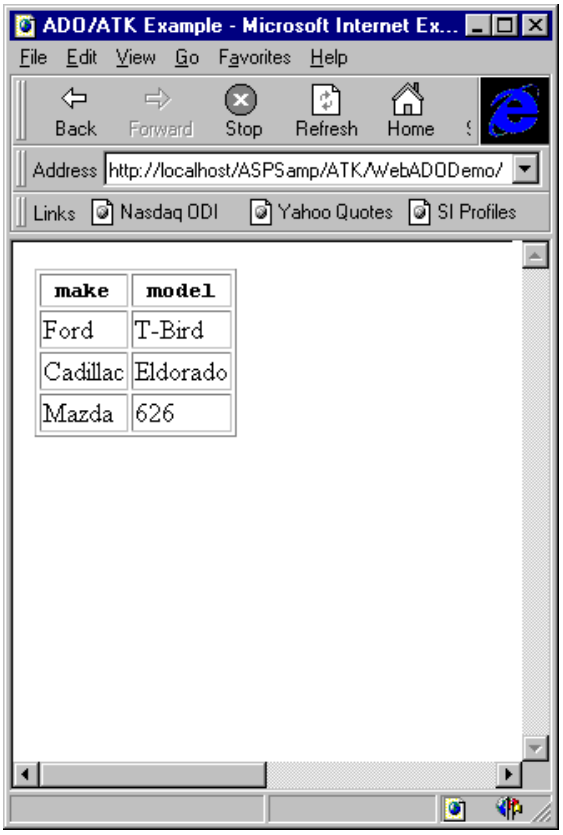
You might want to build a table that does not implicitly navigate relationships among classes, but provides a way to navigate such relations explicitly.

ADO 1.5 and later supports chaptered row sets, which provide a standard way to nest record sets. In this case the **Field.Value** item *is* a **RecordSet** object and you can use it to display the navigated elements.

The resulting HTML table looks like this:



Clicking on an item in the **cars** column explicitly navigates the **Customer** to **Vehicle** relation:



To navigate explicitly in ObjectStore Active Toolkit OLE DB, you have to write additional code to handle **Field.Value**. If you are using ADO 1.5 or later, ATK handles explicit navigation using the chaptered row set functionality. From the user standpoint, this means that the **Field.Value** is registered as type (**Field.Type**) 136 and that it is an ADO **RecordSet** object that can be used to display the navigated rows.

(To learn about specifying explicit navigation, see SQL Support in ATK OLE DB on page 28.)

Sample code ADO 2.0

This sample code provides explicit navigation:

```
For Each aField in RS.fields
    If aField.Type=136 Then 'Is it a Chaptered Rowset?
        set navigatedRS=aField.Value
```

```
        'navigatedRS is an ADO RecordSet object
    else
        Response.Write(aField.Value)
    end if
Next
```

An OLE DB consumer (that is, a consumer that directly accesses OLE DB, without using ADO) can detect explicit navigation by checking the type of the column.

If the column is registered as **DBTYPE_IDISPATCH**, the column contains a pointer to an **IDispatch** interface that exposes the **GetCardinality**, **GetRecordSet** and **GetIRowset** methods. The OLE DB consumer can then access these methods to retrieve the navigated table.

If the column is registered as **DBTYPE_HCHAPTER**, the column contains a handle to the chapter (**HCHAPTER**). Refer to the Microsoft OLE DB documentation for information on how to handle chaptered row sets.

Multimedia Object Managers in ATK OLE DB

ATK supports ObjectStore multimedia Object Managers. ATK can also access multimedia data ATK through OLE DB and ADO.

Using ADO

Using ADO to access a field that contains a multimedia Object Manager is straightforward. Suppose **RS** is a **RecordSet**, and the first column contains a multimedia Object Manager.

osmmData=RS(0).GetChunk(RS(0).ActualSize)

Then use **osmmData** to access an **IStream** or a **SAFEARRAY OLE** interface. For example, in ASP, use **osmmData** in conjunction with the **BinaryWrite Response** object method:

Response.BinaryWrite(osmmData)

The example **WebADODemo** shows how to display a collection of multimedia Object Managers using the ObjectStore Active Toolkit OLE DB provider. This HTML page displays a collection of images stored in the ObjectStore database as **osmmImage** objects

and retrieved using the ObjectStore Active Toolkit OLE DB provider:

Using OLE DB

When you access multimedia Object Managers directly from an OLE DB consumer, rather than through the ADO object model, note that the columns containing multimedia values are registered as **DBTYPE_BYTES** and **DBCOLUMNFLAGS_ISLONG**. Follow the OLE DB specifications to retrieve the binary large object (BLOB) content. The OLE DB consumer can provide a memory buffer to be filled by ObjectStore Active Toolkit OLE DB, or to require an OLE interface such as **ISequentialStream**, **IStream**, or **ILockBytes**.

Accessing the ATK OLE DB Provider

Any OLE DB or ADO client can access an ObjectStore Active Toolkit OLE DB provider. The examples included with ATK show how to access ObjectStore Active Toolkit OLE DB through ADO from a Visual Basic or ASP/VBScript application. For additional information, see Chapter 1, A Visual Basic Application with ATK ActiveX Server and Chapter 2, Accessing ATK ActiveX Server from ASP Applications, in the *ObjectStore Active Toolkit Tutorial*.

Opening a connection

In general, to open an ObjectStore Active Toolkit OLE DB connection, provide a string in this format:

provider=ObjectStore Active Toolkit OLE DB Provider;
data source=<*full-db-path***>;**
[Join=[SQL|Inspector];]
[Import=<*import-database***>;]**

where

- **provider** is the registered OLE DB source you want to open.
- **ObjectStore Active Toolkit OLE DB Provider** is the name used by ObjectStore Active Toolkit OLE DB.
- **data source** is the name of the ObjectStore database you want to open. It can specify any ObjectStore DB path, such as **c:\odi\mydb.db** (a file database) or **myHost:/dbs/my.db** (a remote database) or **myHost::/dbs/my.db** (an ObjectStore rawfs database).
- **Join** specifies whether you want to obtain SQL-like or Inspector-like navigations/joins from this connection. Use this field to override the default behavior of the ObjectStore Active Toolkit OLE DB provider.

Inspector relationships

If you specify **Inspector**, ATK expands one-to-many relationships in a tree-like way. For example, suppose you have two classes named **Customer** and **Vehicle**; a **Customer** might have several **Vehicle**s. ATK provides output in this form:

Name	*Make*	*Model*
John, Smith	Ford	Escort
	Cadillac	DeVille
Bob, Dylan	Ford	Ranger
	Mazda	626
	Dodge	Spirit

SQL relationships

If you specify **Join=SQL**, ATK expands one-to-many relationships to show all the possible values that meet the conditions of the query (as an SQL join would). The output takes this form:

Name	*Make*	*Model*
John, Smith	Cadillac	Deville
Bob, Dylan	Ford	Ranger
Bob, Dylan	Mazda	626
Bob, Dylan	Mazda	626
Bob, Dylan	Dodge	Spirit

Choosing between the two

Usually a tree-like view is more readable, but an SQL-oriented tool such as a report generator cannot easily use it. However, the SQL-like view is less readable and, if you navigate multiple one-to-many relationships in a row, the number of generated rows is much greater than in the tree-like view.

For example, using ASP and ADO you can create and open an ObjectStore Active Toolkit OLE DB connection:

```
Set adoConnection = Server.CreateObject("ADODB.Connection")
Call adoConnection.Open("
   provider=ObjectStore Active Toolkit OLE DB Provider;
   data source=c:\odi\ATK6.0\Examples\demodbs\carsdemo.db")
```

Then you can open a **RecordSet**:

```
Set RS = Server.CreateObject("ADODB.RecordSet")
Call RS.Open("myDataView", adoConnection)
```

where **myDataView** is the name of a data view in the ATK metaknowledge of the current database, or a SQL query following the syntax accepted by ObjectStore Active Toolkit OLE DB. (For more information, see SQL Support in ATK OLE DB on page 28.)

Providing user and password values

ObjectStore Active Toolkit OLE DB can also obtain information about the user and password values to open the ObjectStore database. You can provide this kind of information in two ways:

- Specifying the user and password when you open the OLE DB connection:

```
Call adoConnection.Open("
    provider=ObjectStore Active Toolkit OLE DB Provider;
    data source=c:\odi\ATK6.0\Examples\demodbs\carsdemo.db",
    "userName",
    "password")
```

- Allowing ObjectStore Active Toolkit OLE DB to prompt the user each time it requires a user name and password.

  Through the ADO interface, set the ADO connection **Prompt** variable **adoConnection.Properties("Prompt")** to a value different from **adPromptNever**.

  Through the OLE DB interface, set the **DBPROP_INIT_PROMPT** property in OLE DB to a value different from **DBPROMPT_NOPROMPT**.

If ATK cannot get the user name and password information it requires (because it is not specified when the connection is opened and ATK is not enabled to prompt the user), an error condition is generated.

Accessing the ATK OLE DB Source Through ODBC

OLE DB providers and ODBC drivers provide similar services, making it possible to build a bridge between any OLE DB provider and ODBC.

Use ISG Navigator/Bridge

ISG Navigator/Bridge includes an ODBC driver for OLE DB data sources. It consumes OLE DB providers and allows ODBC applications to access any OLE DB data source. ISG Navigator/Bridge is distributed by ISG International Software Group at **http://www.isgsoft.com**, and by Microsoft Corporation at **http://www.microsoft.com**.

Download and install ISG Navigator/Bridge. Then you can configure it to access ObjectStore Active Toolkit OLE DB through any ODBC client. Refer to Chapter 5, Using Crystal Reports with ATK, in the *ObjectStore Active Toolkit Tutorial* for an example of configuring ISG Navigator/Bridge.

SQL Support in ATK OLE DB

ATK supports a subset of the SQL syntax. This SQL support lets you dynamically modify data views defined in Inspector by customizing the instance format and the filter, and by ordering definitions. ATK supports these statements:

- **SELECT**
- **FROM**
- **WHERE**
- **INSERT INTO**
- **UPDATE**
- **DELETE FROM**
- **ORDER BY ... DESC/ASC**
- **ALIAS**

Examples of supported syntax

Suppose you are using a data view called **my_data_view**, based on a collection of **Customer** instances:

- **Customer** is a class containing **name** and **address** data members, as well as a relationship to the **Vehicle** class, implemented by the **cars** data member.
- **Vehicle** is a class containing **make** and **model** data members and a reverse relationship to the class **Customer** called **owner**.

ATK accepts SQL statements such as these:

SQL Statement	*Description*
SELECT * FROM my_data_view	This statement maintains the default settings of **my_data_view**.
SELECT name, address FROM my_data_view	This statement modifies the instance format used in **my_data_view** to include only the **name** and **address** of each customer.

SQL Statement	*Description*
SELECT name, address FROM my_data_view WHERE name like 'john*'	This statement modifies the instance format and filter defined in **my_data_view**. The resulting table contains only those customers whose names begin with **john** and also displays the customers' names and addresses. ATK translates the filter defined in the SQL statement into an ObjectStore query. This allows you to perform efficient ObjectStore queries through an SQL statement.
SELECT name, address FROM my_data_view ORDER BY name DESC	This SQL statement modifies the instance format and order defined in **my_data_view**. The resulting table shows the customers' names and addresses displayed alphabetically in descending name order.
SELECT name, cars#make FROM my_data_view	ATK maps the SQL column names to data member names. To navigate implicitly from one class to another, you can specify a column name containing the concatenation of the navigated data members separated by the **#** character. This statement lets you build a table based on the persistent collection used to define **my_data_view**. The table adopts the same filtering and ordering settings as the data view, but it contains a column that shows the customer's name and the makes of the cars the customer owns. In general, you can concatenate any number of relationships. For example, **cars#owner#cars#model** is a valid column name. This column identifies the **model**s of the cars by navigating from **car** to **owner** and to **customer**.

SQL Statement	*Description*
SELECT name, cars#make, cars#model FROM my_data_view WHERE cars#make='Ford'	This SQL statement uses the navigated data members to build a new filter in **my_data_view**. As the filter is translated into a native ObjectStore query expression, the ObjectStore query executed by the above SQL statement **iscars[: !strcmp(make,"Ford") :]** This query is satisfied by any customer who owns *at least one* **Ford** car. This result is different from what you could expect reading the SQL expression.
SELECT name, {SELECT make, model FROM cars} FROM my_data_view	To build a table containing explicitly navigated relationships in the OLE DB provider, ObjectStore Active Toolkit OLE DB accepts nested SQL commands. (The ATK ActiveX server does not accept this syntax.) This query creates a table that contains the customer **name** column and the **car** column. For each **name**, the **car** column contains an OLE DB table, which contains the cars that the customer owns. That table contains two columns, the **make** and **model** of the **car**s.
SELECT Employee::salary FROM person_data_view	This SQL statement builds a table containing columns that are associated with data members which are defined in classes derived from the one associated with the dataview. Here, the **person_data_view** contains instances of the **Person** class, and the SQL command displays a column filled with the field **salary** defined in the **Employee** class which derives from **Person**.
INSERT INTO person_data_view (name, surname) VALUES ("Owen", "Foster")	This SQL statement inserts a row in the **person_data_view** table, creating the corresponding persistent object(s) in the process. In order to run this command, a **create** user-defined method (**Person::oledb_create**) must have been registered in Inspector, and the **name** and **surname** columns must be part of the arguments handled by the registered method.

SQL Statement	*Description*
UPDATE person_data_view SET name="Charles" WHERE name="Carlo"	This SQL statement updates all the rows (and hence, the corresponding persistent objects) whose **name** column contains the value "Carlo". In order to run this command, an **update** user-defined method (**Person::oledb_update**) must have been registered in Inspector, and the **name** column must be part of the arguments handled by the registered method.
DELETE FROM person_data_view WHERE name="Carlo"	To delete all the rows (and hence, the corresponding persistent objects) whose "name" column contains the value "Carlo". In order to run this command, a **Person::oledb_delete** "delete" user-defined method must have been registered from Inspector.

CRUD Operations in ATK OLE DB

You can use the Active Toolkit OLE DB interface to perform CRUD operations (**create**, **read**, **update**, and **delete**) on objects in an ObjectStore database. This section describes the ways you can implement CRUD operations using ATK.

Prerequisite

You should understand how to work with user-defined methods before trying to implement CRUD operations in ATK. Information about creating, registering, and using user-defined methods is described in Chapter 7, User-Defined Methods, of the *ObjectStore Inspector User Guide*.

Ways to Implement CRUD Operations

You can perform CRUD operations in ADO clients using either

- ADO **RecordSet** methods
- SQL commands

The specific **create**, **update**, and **delete** operations are peformed by user-defined methods registered in Inspector, regardless of whether you use ADO **RecordSet** methods or SQL. Each type of user-defined method has a required signature — the OLE DB interface is a standard protocol that is unaware of the underlying user-defined methods mechanism. Note there are no such requirements to implement a user-defined **read** method.

Write access is required

If you are using ADO **RecordSet** methods, ADO requires that the recordset itself be opened in a writable mode. The **adLockOptimistic** mode is a suitable way of addressing this requirement.

Use original C++ data member names

The Inspector *instance format* allows you to use user-defined names in place of C++ data member names. (The instance format affects the display in Inspector only, not the database itself.) If you are performing **create** or **update** operations, remember to use the original C++ data member name. This is true regardless of whether you are using ADO **RecordSet** methods or SQL commands to perform the operation.

Note: Data member names can also be modified using the SQL **ALIAS** command.

Editing Objects and Relationships

Editing objects

ADO clients are written against OLE DB tables; an OLE DB table corresponds to a collection of objects in an ObjectStore database. Consider the following classes, **CAuthor** and **CBook**:

CAuthor

m_FirstName **m_LastName** **m_Books**

CBook

m_ISBN **m_Title** **m_Author**

To create, update, or delete persistent instances of **CAuthor** from an ADO client, you would first retrieve a table of authors by execute a statement like this one:

```
adoRS.OPEN(
   "SELECT m_FirstName, m_Books#m_ISBN FROM CAuthor")
```

The resulting table would look like this:

FirstName	ISBN
Owen	1-879239-02-3
Owen	1-879239-10-4
Henry	1-879239-07-1
Minollo	1-879239-09-5

Next, the client would edit the table, for example, adding a new author:

```
adoRS.AddNew "m_FirstName" , "Ugo"
```

The **AddNew** command calls a C++ user-defined method registered in Inspector, **CAuthor::oledb_create**, to create a new instance of **CAuthor**. You can learn more about the details of this method in Creating Objects on page 35.

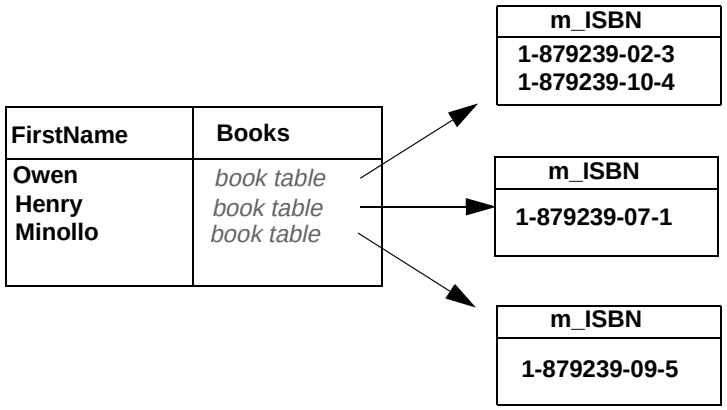
Editing relationships

You can also edit relationships themselves. Note that the **CAuthor** and **CBook** classes are related, in ObjectStore, by the **m_Books** and **m_Authors** data members.

To edit the relationship between two objects — and not the objects themselves, as seen earlier — you would first execute a statement like the following in the ADO client:

```
adoRS.OPEN(
   "SELECT m_FirstName, {SELECT m_ISBN FROM m_Books}
   FROM CAuthor")
```

Here is the resulting table:



Notice that the **SELECT** statement creates *chaptered rowsets*, that is, tables within tables. It is against these chaptered rowsets that you executing statements in order to edit a relationship.

Next, you locate the chaptered rowset for a given instance, in this case the first instance, for the author **Owen**.

```
adoRS.MoveFirst
```

Next, you retrieve the chaptered rowset for the author:

```
SET owenBooks=adoRS("m_Books")
```

Finally, you execute the desired CRUD statement, such as adding a new book for that author:

```
owenBooks.AddNew "m_ISBN", "1-879239-06-06"
```

The result here is that a new book (identified by its ISBN) is added to the relationship identified by the selected author, in this case, **Owen**. As before, the **AddNew** command calls a C++ user-defined

method registered in Inspector. This method requires a different signature because it is used to edit a relationship, not an object: **CAuthor::oledb_create_relationship**

Navigating relationships

If you are using ADO **RecordSet** methods to implicitly navigate relationships in order to perform **create** or **update** operations, you must specify the complete C++ path using **#** as the separation character. For example consider a relationship between **Customer**, **Order**, and **Warehouse** classes. You would represent the navigation between the **Customer** and **Warehouse** classes as **Customer#Order#Warehouse**.

Creating Objects

Method signature

You need to define and register in Inspector a **create** user-defined method for each class you want to persistently instantiate. The **create** method must have the following signature:

```
static PersistentClass* PersistentClass::oledb_create(
    os_database* aDB,
    const os_Dictionary<char*, void*>& args)
```

Using ADO

To create a row in an OLE DB table using ADO, use the **RecordSet::AddNew** method. Open a **RecordSet** based on the class for which you want to create a new instance.

The fields you specify in the **AddNew** method must be the same as (or a subset of) the data members exposed through the instance format of the data view corresponding to the **RecordSet** and the arguments specified in the registered **oledb_create** user-defined method.

Using SQL

To create a row in an OLE DB table using SQL, use the **INSERT INTO** command. Using SQL enables you to create a new object without explicitly opening a **RecordSet**. For example, to create a new instance of **CCustomer**, you could specify the following SQL command:

```
INSERT INTO CCustomer (m_FirstName, m_LastName)
VALUES ('Hank', 'Foster')
```

Interpreting class names

Active Toolkit is able to interpret class names as Inspector data views — even if no data view has been formally defined for that class in Inspector. It does this by implicitly creating a data view based on the class, using the default instance format representation for that class.

Reading Objects

Reading information about objects using a user-defined method does not require any special code regardless of whether you use ADO or SQL. The only requirements are that the user-defined method is registered in Inspector and that it is part of the instance format for the data view corresponding to the **RecordSet**.

Updating Objects

Method signature

You need to define and register in Inspector an **update** user-defined method for each class of any persistent instance you want to modify. The **update** method must have the following signature:

```
int PersistentClass::oledb_update(
    const os_Dictionary<char*, void*>& args)
```

Using ADO

To update a row in an OLE DB table using ADO, position on the row you want to modify and then use the **RecordSet::Update** method. The **RecordSet** you open must be based on a data view that contains the objects of a persistent class for which an **oledb_update** user-defined method has been registered.

The fields you modify must be the same as (or a subset of) the data members exposed through the instance format of the data view corresponding to the **RecordSet** and the arguments specified in the registered **oledb_update** user-defined method.

Using SQL

To update a row in an OLE DB table using SQL, use the **UPDATE** command. Using SQL enables you to update an object without explicitly opening a **RecordSet**. For example, to update an instance of **CCustomer**, you could specify the following SQL command:

```
UPDATE CCustomer SET m_FirstName='Henry'
WHERE m_FirstName='Hank' AND m_LastName='Foster'
```

Updating relationships

In order to update a *relationship*, you need to define and register an **update** user-defined method with the following signature:

```
int PersistentClass::oledb_create_relationship(
    const os_Dictionary<char*, void*>& args)
```

This method must be able to update a relationship of the **PersistentClass** on which it is defined. This method always receives an argument, **oledb_relationship_name**, which identifies the relationship that is to be updated. The other arguments

received by **oledb_create_relationship** are the parameters specified by the user in the ADO command that invoked the method. These parameters must be sufficient to uniquely identify the other object participating in the relationship.

From ADO you use chaptered rowsets to represent relationships in OLE DB tables. You can invoke the **oledb_create_relationship** by calling the **RecordSet::AddNew** method on a **RecordSet** obtained by an OLE DB chaptered rowset.

For example, if you open a **RecordSet** with the SQL command

```
SELECT m_FirstName, {SELECT m_OrderID, m_OrderDescription
    FROM m_Orders}
FROM CCustomer
```

the second column identifies a chaptered rowset. This means that the fields of the **RecordSet** corresponding to the second column are **RecordSet** objects themselves; each contains a description of all the orders made by a customer.

If you call **RecordSet::AddNew** on the navigated **RecordSet** that contains the order information, ATK OLE DB tries to invoke **CCustomer::oledb_create_relationship**, specifying **m_Orders** in the **oledb_relationship_name** argument and passing the other arguments specified in the **RecordSet::AddNew** method.

Deleting Objects

Method signature

You need to define and register in Inspector a **delete** user-defined method for each class from which you want to delete a persistent instance. The **delete** method must have the following signature:

```
static int PersistentClass::oledb_delete(
    void* anObject)
```

Using ADO

To delete a row in an OLE DB table using ADO, position on the row you want to delete and then use the **RecordSet::Delete** method. The **RecordSet** you open must be based on a data view that contains the objects of a persistent class for which an **oledb_delete** user-defined method has been registered.

Using SQL

To delete a row in an OLE DB table using SQL, use the **DELETE** command. Using SQL enables you to delete an object without explicitly opening a **RecordSet**. For example, to delete an instance of **CCustomer**, you could specify the following SQL command:

```
DELETE FROM CCustomer WHERE m_FirstName='Henry'
AND m_LastName='Foster'
```

Deleting relationships

In order to delete a *relationship*, you need to define and register a **delete** user-defined method with the following signature:

```
int PersistentClass::oledb_delete_relationship(
    const os_Dictionary<char*, void*>& args)
```

This method must be able to delete a relationship of the **PersistentClass** on which it is defined. This method always receives an argument, **oledb_relationship_name**, which identifies the relationship that is to be deleted. The other arguments received by **oledb_delete_relationship** are provided by ATK — they are the intersection of all the chaptered rowset columns and the arguments registered in the **oledb_delete_relationship** method.

From ADO you use chaptered rowsets to represent relationships in OLE DB tables. You can invoke the **oledb_delete_relationship** by calling the **RecordSet::Delete** method on a **RecordSet** obtained by an OLE DB chaptered rowset.

For example, if you open a **RecordSet** with the SQL command

```
SELECT m_FirstName, {SELECT m_OrderID, m_OrderDescription
    FROM m_Orders}
FROM CCustomer
```

the second column identifies the chaptered rowset. This means that the fields of the **RecordSet** corresponding to the second column are **RecordSet** objects themselves; each contains a description of all the orders made by a customer.

If you position on order number **23**, the **myOrderedBook** row of the navigated **RecordSet**, and then call **RecordSet::Delete**, ATK OLE DB tries to invoke **CCustomer::oledb_delete_relationship**, specifying **m_Orders** in the **oledb_relationship_name** argument, **23** in the **m_OrderID** argument, and **myOrderedBook** in the **m_OrderDescription** argument.

Retrieving Error Information

Using ADO

When an ADO method that accesses the ObjectStore Active Toolkit OLE DB provider causes an error, ADO returns a standard ADO error code or generates the error code **#80041000**. ObjectStore Active Toolkit OLE DB uses the error code to identify its internal error conditions. In either case, you can retrieve extended information about the generated error.

For example, this code attempts to open a database in an Active Server Page:

```
On Error Resume Next
Set adoConnection = Server.CreateObject("ADODB.Connection")
Call adoConnection.Open("provider=ObjectStore OLE DB Provider;data source=mydb.db","","")
if Err.Number <> 0 then
   Response.Write("Error# " & Hex(Err.Number) & "<BR>")
   Response.Write("Generated by: " & Err.Source & "<BR>")
   Response.Write("Description: " & Err.Description & "<BR>")
   Exit Sub
End If
```

If the database is not available, the Active Server Page generates this output:

```
Error# 80041000
Generated by: ObjectStore OLE DB Provider
Description: Could not open in MVCC mode the database 'mydb.db'!
```

You might find it useful to scan the **Connection::Errors** collection to retrieve multiple descriptions for the error. (Refer to Microsoft's ADO documentation for further information about the **Connection::Errors** collection.)

Using OLE DB

If you are accessing OLE DB directly (that is, without using the ADO layer), you can retrieve information about the errors generated by the ObjectStore Active Toolkit OLE DB provider. Refer to Microsoft's OLE DB documentation for details.

Chapter 4
Configuring ATK

Introduction

The ATK installation configures ATK to use the same ObjectStore application schema, metaknowledge, and string formats as Inspector. Using the ATK configuration utility, you can change these settings and the default settings for the types of joins ATK uses in queries. You can also check the status of the ObjectStore Server on the application host.

In this chapter

This chapter presents the following topics:

The ATK Configuration Utility

The ATK configuration utility is an application that you can use to change the ATK ActiveX server and OLE DB provider default settings. The ATK installation program establishes these defaults based on the locations of ATK and Inspector. You can synchronize all ATK settings with the corresponding Inspector settings, or you can reconfigure individual ATK settings.

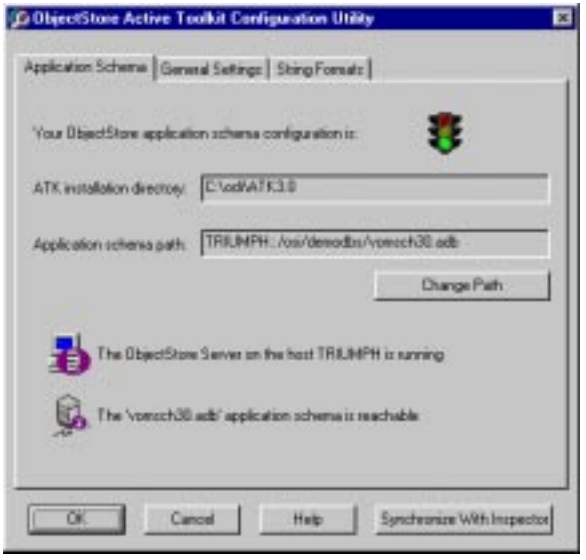
Running the ATK Configuration Utility

Before configuring

The database schema is stored in the ATK library, and all other settings are stored in the Windows NT registry. Before configuring ATK, unlock the ATK libraries by closing all applications that created ATK ActiveX server objects or access the ATK OLE DB provider, including Active Server Pages. If the ATK libraries are locked when you run the ATK configuration utility, it cannot modify the application schema path. Furthermore, any changes applied to the ATK settings are not visible to the already running ATK instances. Even after a reboot, the application schema path is not updated under such circumstances.

Starting the configuration utility

To start the ATK configuration utility, choose **Start | Programs | ObjectStore ATK 6.0 | Configuration Utility**.



The ATK configuration utility has three sheets:

- **Application Schema**
- **General Settings**
- **String Formats**

Procedures for configuring the information on these sheets are described in the following sections.

Synchronizing with Inspector

ATK and Inspector share metaknowledge, data views, instance formats, and the ObjectStore application schema. To synchronize these ATK settings, the ATK string display format, and all other ATK settings with the corresponding Inspector settings, click **Synchronize with Inspector** from any sheet. (For more information about how ATK works with the Inspector, refer to Chapter 1, ATK and the Inspector, on page 1.)

Setting the Application Schema Path

What is the application schema?

The *application schema* is an ObjectStore database in the file **vomsch30.adb**. In order for the application to access the application schema, the schema must be under the control of an ObjectStore Server. If you manually move the application schema, you must reconfigure ATK with the schema's new location.

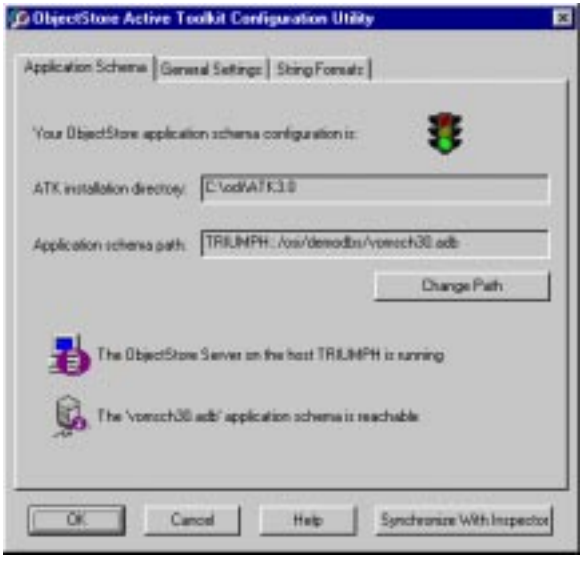
For example, suppose an application running on a Windows NT workstation that has no ObjectStore Server queries a Solaris ObjectStore Server. To successfully query the Solaris Server database, you must copy the application schema to the Solaris machine and reconfigure ATK on the Windows NT machine with the remote location of the Solaris workstation.

Default location

By default, the **vomsch60.adb** is placed in **C:\odi\ATK6.0\lib**. Use the **Application Schema** sheet to check or modify the ATK application schema path, and to configure ATK to use a particular ObjectStore Server located on your network.

Application Schema Sheet

The **Application Schema** sheet displays the ATK application schema configuration:



- The configuration status, signified by a green, yellow, or red traffic signal icon
- The location of the application schema database
- Whether the application schema database is accessible to an ObjectStore Server

Traffic signal icon

The color of the traffic signal summarizes the application schema configuration:

- Green — ATK is properly configured to use an application schema database on a workstation where an ObjectStore Server is running.
- Yellow — Although ATK is configured to use an application schema database on a workstation where an ObjectStore Server is running, the application schema database has not been copied to the specified directory.
- Red — ATK is configured to use an application schema database on a host where no ObjectStore Server is running.

An ATK application that runs when the traffic signal is yellow or red encounters an error when it attempts to open a database.

Changing the application schema database path

To establish a new location:

1 Click **Change App Schema Path**.

2 Enter another directory location. You can specify a directory on the local machine, an ObjectStore rawfs, or a network file server directory.

3 Copy **vomsch60.adb** to the directory location you specified.

ObjectStore Server

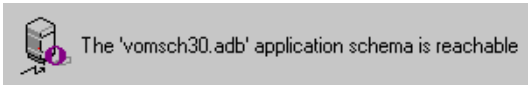
The **Application Schema** sheet also indicates whether an ObjectStore Server is running on the host designated by the application schema path:

- 

  An ObjectStore Server is running on the host.

- 

  No ObjectStore Server is running on the host. The traffic signal is red.

- 

  An ObjectStore Server is running on the host. Because ATK is properly configured and ready to be run, the traffic signal is green.

- 

  The ATK application schema database is not accessible, because an ObjectStore Server is not running on the host designated by the application schema path, or because the database file **vomsch60.adb** is not present in that directory. The traffic signal is yellow if the Server is running, or red if no Server is running.

Synchronizing with Inspector

To use the application schema database path that Inspector uses, click **Synchronize with Inspector**. Note that this synchronizes all ATK configuration settings in all tabs with the equivalent Inspector settings.

Modifying the ATK General Settings

Use the **General Settings** sheet to specify the location of the ATK metaknowledge, and to choose the default type of join ATK uses when displaying relationships in a single table.

Metaknowledge

Because ATK and Inspector use the same metaknowledge, the ATK installation checks the location of the Inspector knowledge and uses that setting for ATK. (For details of how ATK and the Inspector share metaknowledge, refer to Chapter 1, ATK and the Inspector, on page 1.)

By default, the ATK installation designates the file system as the metaknowledge location. To manually modify this setting, specify the location from which ATK should load the metaknowledge.

Default X-to-Many Joins

By default, the ATK ActiveX server displays one-to-many and many-to-many relationships in a table using Inspector-like joins, and the ATK OLE DB provider returns one-to-many and many-to-many relationships using SQL-like joins. You can choose the join format you prefer. For more information about these types of

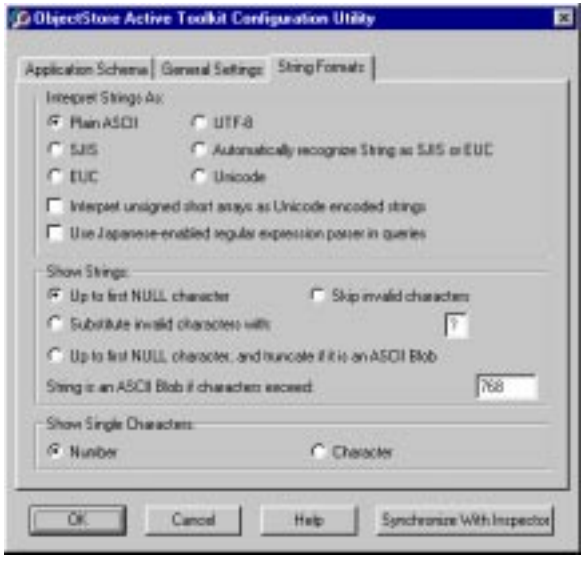
joins, refer to Chapter 3, Active Toolkit OLE DB Provider, on page 15.

Synchronizing with Inspector

If you have modified the location of the Inspector metaknowledge and want to update all ATK settings to match the corresponding Inspector setting, click **Synchronize with Inspector**.

Modifying the String Format Settings

Use the **String Formats** sheet to specify the default string display format for the current database. For additional information about string formats and string interpretation, refer to Viewing Blob in the *ObjectStore Inspector User Guide.*



Interpreting Strings

By default, ATK interprets strings as plain ASCII text. You can interpret strings using any format that the Inspector supports. This is especially helpful if the database contains strings that are encoded using SJIS, EUC, or Unicode.

Japanese Expressions

If the application retrieves objects encoded in Japanese characters, you can ensure that ATK uses a Japanese-enabled parser that supports DBCS and Unicode. This parser is included in the ATK kernel.

Show Strings

By default, ATK displays strings up to the first null character it encounters. Instead, ATK can skip displaying invalid characters or display an alternate character for each invalid one.

BLOBs

By default, ATK displays ASCII strings that are longer than 768 characters as binary large objects (BLOBs). You can increase or decrease this limit.

Show Single Characters

You can display single characters as either a character or an ASCII value. For example, the letter **D** can be shown as **D** or **68**. This also affects the type of the column in which the value is displayed; it is either char[n] or integer.

Chapter 5
ATK Object Model

Introduction

This chapter describes the ActiveX object model exposed by ATK, whose members provide a programmatic interface to ActiveX objects in an ObjectStore database.

Overview

The ATK ActiveX object model is divided into five categories:

- Top-level objects

  ATKKernel and **ATKDatabase** are the entry points into the ATK object model. These provide global utility functions and access to the database.

- Persistent object handling classes

  ATKReference, **ATKReferences**, **ATKObjectManager**, and **ATKRoot** let you navigate objects in your database, retrieve information about the kind of object you are accessing, retrieve the tabular representation for this object, and handle multimedia objects (ObjectStore Object Managers).

- Tabular view classes

  ATKDataView, **ATKTable**, and **ATKInstanceFormat** let you access ObjectStore collections as tables.

- Database schema classes

  ATKClass and **ATKClassSlot** provide high-level information about the database schema.

- User-defined method handling classes

  ATKMethod, **ATKMethods**, and **ATKMethodArguments** help you work with user-defined methods registered in Inspector.

In this chapter

This chapter documents the following ATK classes:

ATKClass

ATKClass provides a high-level interface to the ATK metaknowledge. **ATKClass** retrieves information about the declaration of a class in the database schema, and can retrieve a class extent if the user defines that behavior in Inspector.

Methods

ATKClass contains these methods:

Method	*Description*
GetClassExtent	Returns an **ATKReferences** object that contains the extent of the current class.
GetClassName	Returns the name of the class.
GetFatherClasses	Returns an **ATKClasses** object containing classes that are first-level ancestors of the current class.
GetMethods	Returns all the available user-defined methods registered for the current class.
GetSonClasses	Returns an **ATKClasses** object containing classes that are first-level derivations of the current class.
GetSlots	Returns an **ATKClassSlots** object that represents the attributes declared for the current class or its ancestors.
IsTopLevelClass	Checks if the class has ancestors.

ATKClass::GetClassExtent

ATKClass::GetClassExtent returns an **ATKReferences** object that contains the extent of the current class, as defined in Inspector.

ATKReferences GetClassExtent

Return value

ATKReferences

Comments

GetClassExtent checks the metaknowledge to determine which roots and collections are defined as part of the extent of the class.

ATKClass::GetClassName

ATKClass::GetClassName returns the name of the class as it is declared in the database schema.

BSTR GetClassName

Return value — A string containing the name of the represented class.

ATKClass::GetFatherClasses

ATKClass::GetFatherClasses returns an **ATKClasses** object. This collection contains first-level ancestor classes of the current class (that is, the classes from which this class directly inherits).

ATKClasses GetFatherClasses

Return value — **ATKClasses**

ATKClass::GetMethods

ATKClass::GetMethods () returns all the available user-defined methods registered for the current class.

Return value — **ATKMethods**

ATKClass::GetSonClasses

ATKClass::GetSonClasses returns an **ATKClasses** object. The classes in the **ATKClasses** collection are first-level derivations of the current class; that is, they inherit directly from the current class.

ATKClasses GetSonClasses

Return value — **ATKClasses**

ATKClass::GetSlots

ATKClass::GetSlots returns an **ATKClassSlots** object, which is a collection of data members declared for the current class or its ancestors.

ATKClassSlots GetSlots

Return value — **ATKClassSlots**

ATKClass::IsTopLevelClass

ATKClass::IsTopLevelClass checks if the class has ancestors.

Boo IsTopLevelClass

Return values

Return Value	*Description*
TRUE	The class has no ancestors and is therefore a top-level class.
FALSE	The class has ancestors and is therefore not a top-level class.

ATKClasses

ATKClasses is a collection of **ATKClass** objects.

Property

ATKClasses contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKClasses contains these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKClasses::Count

ATKClasses::Count returns the number of **ATKClass** objects in the **ATKClasses** collection.

long Count

Return value

A **long** representing the number of objects.

ATKClasses::Item

ATKClasses::Item retrieves a specific **ATKClass** object from the collection.

ATKClass Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Identifies an object in the collection. A **Long** value specifies the position of the required object in the collection. A **BSTR** value specifies the name of the object.

Return value

ATKClass

ATKClassSlot

ATKClassSlot represents a data member of a class. Use **ATKClassSlot** to retrieve information about the data members of the classes declared in the database schema.

Methods

ATKClassSlot contains these methods:

Method	*Description*
GetAccess	Returns the attribute access type.
GetClass	Returns the name of the class in which the slot is declared.
GetName	Returns the attribute name.
GetRelationCardinality	Returns the cardinality of the relationship.
GetRelatedClass	Returns the **ATKClass** object of the related class.
GetType	Returns the attribute type.
IsRelation	Checks if an attribute is a relationship between two classes.

ATKClassSlot::GetAccess

ATKClassSlot::GetAccess returns the attribute access type for this data member, as it is declared in the database schema.

BSTR GetAccess

Return values

GetAccess can return any of the following values:

- **private**
- **protected**
- **public**

ATKClassSlot::GetClass

ATKClassSlot::GetClass returns the name of the class to which this data member belongs.

BSTR GetClass

Return value

A string containing the name of the class.

ATKClassSlot::GetName

ATKClassSlot::GetName returns the name of the data member as it is declared in the database schema.

BSTR GetName

Return value

A string containing the name of the data member.

ATKClassSlot::GetRelationCardinality

ATKClassSlot::GetRelationCardinality returns the cardinality of the relationship.

short GetRelationCardinality

Return values

GetRelationalCardinality can return any of the following values:

Return Value	*Description*
0	The data member is not a relationship.
1	The cardinality of the relationship is 1.
2	The cardinality of the relationship is many.

Comments

If this data member is a relationship between two classes, you can use **GetRelationCardinality** to retrieve its cardinality. To determine if the attribute is a relationship, **GetRelationCardinality** checks if the value of **IsRelation** is **TRUE**.

ATKClassSlot::GetRelatedClass

ATKClassSlot::GetRelatedClass returns the **ATKClass** object representing the class reached by navigating the relationship.

ATKClass GetRelatedClass

Return value

ATKClass

Comments

If this data member is a relationship between two classes, you can use **GetRelatedClass** to retrieve the class that is reached by navigating the relationship. **GetRelatedClass** first checks if the value of **IsRelation** is **TRUE** to determine if the attribute is a relationship.

ATKClassSlot::GetType

ATKClassSlot::GetType returns the data member type, as it is declared in the database schema.

BSTR GetType

Return value

A string containing the name of the data member type.

ATKClassSlot::IsRelation

ATKClassSlot::IsRelation checks the metaknowledge to determine whether the attribute is classified as a relationship between two classes.

Bool IsRelation

Return values

IsRelational can return the following values:

Return Value	*Description*
TRUE	The data member is a relationship between two classes.
FALSE	The data member is not a relationship between two classes.

ATKClassSlots

ATKClassSlots is a collection of **ATKClassSlot** objects.

Property

ATKClassSlots contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKClassSlots contains these method:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKClassSlots::Count

ATKClassSlots::Count returns the number of **ATKClassSlot** objects in the collection.

long Count

Return value

A **long** representing the number of objects.

ATKClassSlots::Item

ATKClassSlots::Item retrieves a specific **ATKClassSlot** object from the collection.

ATKClassSlot Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Identifies an object in the collection. A **Long** value specifies the position of the required object in the collection. A **BSTR** value specifies the name of the object.

Return value

ATKClassSlot

ATKDatabase

ATKDatabase represents an ObjectStore database, and provides access to all the operations you can perform on an ObjectStore database through ATK. To create an **ATKDatabase** object, you need an **ATKKernel** object. Using **ATKDatabase**, you can retrieve all available data views and roots and all the classes declared in the database schema, and set or retrieve default settings about ATK ActiveX behaviors.

Methods

ATKDatabase contains these methods:

Method	*Description*
CreateObject	Invokes a registered user-defined method to create an instance of the specified class.
CreateObjectInSegment	Invokes a registered user-defined method to create an instance of the specified class.
GetAllClasses	Returns an **ATKClasses** object that lists the classes declared in the database schema.
GetClass	Returns the **ATKClass** represented by a specific class name.
GetInstanceFormat	Returns the instance format associated with a specific instance format name.
GetDataView	Returns the data view saved in Inspector using the specified name.
GetDataViews	Returns a list of all the data views defined in the database metaknowledge.
GetRoot	Returns the **ATKRoot** corresponding to a specific root name.
GetRoots	Returns a list of all the roots stored in the database.
SetJoinType	Specifies whether tables are returned in SQL-like or Inspector-like tabular format.

ATKDatabase::CreateObject

ATKDatabase::CreateObject invokes a registered user-defined method to create an instance of the specified class. *ClassName* contains the name of the class on which the **create** method *MethodName* has been defined and registered. *Arguments* is the map you use to specify the arguments needed to run *MethodName*.

```
ATKDatabase::CreateObject(
    BSTR ClassName,
    BSTR MethodName,
    ATKMethodArguments* Arguments)
```

Return value **ATKReference**

ATKDatabase::CreateObjectInSegment

ATKDatabase::CreateObjectInSegment invokes a registered user-defined method to create an instance of the specified class. *ClassName* contains the name of the class on which the **create** method *MethodName* has been defined and registered. *Arguments* is the map you use to specify the arguments needed to run *MethodName*. *SegmentNumber* specifies the number of the ObjectStore database segment in which the object should be persistently allocated.

```
ATKDatabase::CreateObjectInSegment(
    BSTR ClassName,
    BSTR MethodName,
    long SegmentName
    ATKMethodArguments* Arguments)
```

Return value **ATKReference**

ATKDatabase::GetAllClasses

ATKDatabase::GetAllClasses returns an **ATKClasses** object that lists the classes declared in the database schema. These are the same classes that Inspector displays in the schema representation.

```
ATKClasses GetAllClasses
```

Return value **ATKClasses**

Comments **GetAllClasses** retrieves the list of declared classes from the metaknowledge, and the other information about classes from the database schema. You can use the returned **ATKClasses** object to iterate on all available classes.

ATKDatabase::GetClass

ATKDatabase::GetClass returns the **ATKClass** object with the name *aClassName*.

ATKClass GetClass(
BSTR *aClassName***)**

Parameter

GetClass accepts this parameter:

Parameter	*Description*
aClassName	The name of the class you want to retrieve.

Return value

ATKClass

Comments

The parameter *aClassName* is case sensitive.

ATKDatabase::GetDataView

ATKDatabase::GetDataView returns an **ATKDataView** object that contains the data view saved as *dataViewExpression* in Inspector.

ATKDataView GetDataView(
BSTR *dataViewExpression***)**

Parameter

GetDataView accepts this parameter:

Parameter	*Description*
dataViewExpression	Specifies the data view, as defined in the ATK metaknowledge. *dataViewExpression* can also contain an SQL command (following the syntax described in Chapter 3, Active Toolkit OLE DB Provider) that allows you to customize a data view.

Return value

ATKDataView

ATKDatabase::GetDataViews

ATKDatabase::GetDataViews returns an **ATKDataViews** object,which lists all the data views defined in the metaknowledge.

ATKDataViews GetDataViews

Return value

ATKDataViews

Comments

You can use the returned **ATKDataViews** object to iterate on all the available data views.

ATKDatabase::GetInstanceFormat

ATKDatabase::GetInstanceFormat returns the instance format associated in the metaknowledge with a specific instance format name.

ATKInstanceFormat GetInstanceFormat(
BSTR *instanceFormat***)**

Parameter

GetInstanceFormat accepts this parameter:

Parameter	*Description*
instanceFormat	The name of the instance format that you want to access.

Return value

ATKInstanceFormat

ATKDatabase::GetRoot

ATKDatabase::GetRoot returns the **ATKRoot** corresponding to *aRootname*.

ATKRoot GetRoot(
BSTR aRootName)

Parameter

GetRoot accepts this parameter:

Parameter	*Description*
aRootname	The name of the root as stored in the ObjectStore database.

Return value

ATKRoot

ATKDatabase::GetRoots

ATKDatabase::GetRoots returns an **ATKRoots** object that contains a collection of the roots stored in the database.

ATKRoot GetRoots

Return value

ATKRoots

Comments

Use **GetRoots** to iterate on all the roots defined in the ObjectStore database.

ATKDatabase::SetJoinType

ATKDatabase::SetJoinType instructs ATK to use an SQL-like or Inspector-like tabular representation when displaying tables that allow navigation of one-to-many relationships.

void SetJoinType(
enumerator *joinType***)**

Parameter

SetJoinType accepts this parameter:

Parameter	***Description***
joinType	**InspectorLike** to use an Inspector-like join.
	SQLLike to use an SQL-like join.

Comments

Use this field to override the default behavior of the ATK ActiveX server. For example, suppose you have two classes named **Customer** and **Vehicle**; a **Customer** might have several **Vehicle**s.

If you specify **InspectorLike**, ATK expands one-to-many relationships in a tree-like way:

Name	***Make***	***Model***
John, Smith	Ford	Escort
	Cadillac	DeVille
Bob, Dylan	Ford	Ranger
	Mazda	626
	Dodge	Spirit

If you specify **SQLLike**, ATK expands one-to-many relationships to show all the possible values that meet the conditions of the query (as an SQL join would). The output takes this form:

Name	***Make***	***Model***
John, Smith	Ford	Escort
Johh, Smith	Cadillac	DeVille
Bob, Dylan	Ford	Ranger
Bob, Dylan	Mazda	626
Bob, Dylan	Dodge	Spirit

ATKDataView

ATKDataView is the ATK ActiveX representation of a data view that has been defined in Inspector. Use **ATKDataView** to set the parameter values required by the filter, which is defined in the data view, and to execute the ObjectStore query on the underlying collection object.

ATKDataView works with three types of queries:

- Nonfiltered, with no constraint specified on the collection
- Filtered with no parameters (a simple ObjectStore query)
- Filtered with constraints that must be specified at execution time

You can use **ATKDataView** to set the query parameters, if any, and then to jump to the **ATKReferences** or **ATKTable** query result.

Methods

ATKDataView contains these methods:

Method	*Description*
GetAllParameters	Returns an **ATKStrings** object of strings that list the parameters in a query.
GetATKClass	Returns the class on which the data view is defined.
GetATKTable	Returns an **ATKTable** object that contains objects satisfying a query.
GetDataInstanceFormat	Returns the **ATKInstanceFormat** object associated with this data view.
GetDataInstanceFormatName	Returns an **ATKInstanceFormat** object associated with the data view.
GetName	Returns the name of the data view as defined in Inspector.
GetParameterPrompt	Returns the prompt to return to the client when specified values are retrieved.
GetParameterType	Returns the type of the parameter.

Method	*Description*
GetParameterValues	Retrieves specific values and returns them as strings.
IsFiltered	Checks if a query contains constraints.
IsFilterParameterized	Checks if a query contains constraints that must be fulfilled before the query is performed.
IsParameterList	Checks if a parameter value must be chosen from a list of values predefined by the query author.
IsParameterMultipleSelection	Checks if a parameter value must be chosen from a list of values that the query author has defined.
IsParameterSorted	Checks if the list of supplied values must be sorted.
PerformFiltering	Returns an **ATKReferences** object that lists the object satisfying the query constraints and parameters.
SetParameter	Sets the value of a parameter.

ATKDataView::GetAllParameters

ATKDataView::GetAllParameters returns an **ATKStrings** object that contains the names of all parameters used in the query, as defined in the data view.

ATKStrings GetAllParameters

Return value **ATKStrings**

ATKDataView::GetATKClass

ATKDataView::GetATKClass returns the class on which the data view is defined.

ATKClass* ATKDataView::GetATKClass( )

ATKDataView::GetATKTable

ATKDataView::GetATKTable returns an **ATKTable** object containing a tabular representation of the data view.

ATKTable GetATKTable[(
BSTR *instanceFormatName*)**]**

Parameter

GetATKTable accepts this parameter:

Parameter	*Description*
instanceFormatName	Optional. The name of the instance format, as defined in the metaknowledge, that ATK uses to format the table referenced by **ATKDataView**.

Return value

ATKTable

Comments

If *instanceFormatName* is an empty string ("") or is not supplied, ATK uses the default instance format for the class to which the elements of the referenced collection belong.

If the data view requires any filtering, **GetATKTable** executes the corresponding ObjectStore query. This is the most efficient means of accessing the tabular format of a data view through an **ATKDataView** object, and of filling in a table or list with representations of objects that satisfy a query.

ATKDataView::GetDataInstanceFormat

ATKDataView::GetDataInstanceFormat returns an **ATKInstanceFormat** object that contains the instance format associated with the data view in Inspector.

ATKInstanceFormat GetDataInstanceFormat

Return value

ATKInstanceFormat

ATKDataView::GetDataInstanceFormatName

ATKDataView::GetDataInstanceFormatName returns a string that contains the name of the instance format associated with the data view in Inspector.

BSTR GetDataInstanceFormatName

Return value

A string containing the name of the instance format.

ATKDataView::GetName

ATKDataView::GetName returns the name of the data view as defined in Inspector.

BSTR GetName

Return value

A string containing the name of the data view.

ATKDataView::GetParameterPrompt

ATKDataView::GetParameterPrompt returns the prompt that should be sent to the client when one or more of the parameter values is retrieved. The data view author defines the parameter values in Inspector.

BSTR GetParameterPrompt(
BSTR *parameterName***)**

Parameter

GetParameterPrompt accepts this parameter:

Parameter	***Description***
parameterName	The name of the parameter for which you want to retrieve a prompt.

Return value

A string containing the prompt.

ATKDataView::GetParameterType

ATKDataView::GetParameterType returns the type of the parameter (such as **string** or **number**) of the attribute associated with the parameter.

BSTR GetParameterType(
BSTR *parameterName***)**

Parameter

GetParameterType accepts this parameter:

Parameter	***Description***
parameterName	The name of the parameter whose type you want to retrieve.

Return values

GetParameterType can return the following values:

number

string

ATKDataView::GetParameterValues

ATKDataView::GetParameterValues returns an **ATKStrings** object that contains a collection of all acceptable values for the specified parameter.

ATKStrings GetParameterValues(
BSTR *parameterName***)**

Parameter

ATKDataView::GetParameterValues accepts this parameter:

Parameter	*Description*
parameterName	The name of the parameter whose possible values you want to retrieve.

Return value

ATKStrings

Comments

Call this function and specify a parameter for which **IsParameterList(***parameterName***)** returns **TRUE**. If **ATKDataView::IsParameterList** is **TRUE**, **GetParameterValues** returns an **ATKStrings** object that contains all possible values of the parameter. **GetParameterValues** queries the ObjectStore database to retrieve all unique values of the data member associated with the specified parameter.

ATKDataView::IsFiltered

ATKDataView::IsFiltered checks if the **ATKDataView** object contains filter definitions on the ObjectStore collection it represents.

Bool IsFiltered

Return values

IsFiltered can return the following values:

Return Value	*Description*
TRUE	The data view contains constraints; that is, at least one filter is defined on the data view.
FALSE	The data view does not contain constraints; that is, there is no filter defined on the data view.

ATKDataView::IsFilterParameterized

ATKDataView::IsFilterParameterized checks if the data view that **ATKDataView** represents contains a filter that must be fully specified at execution time before the query is performed.

Bool IsFilterParameterized

Return values

IsFilterParameterized can return the following values:

Return Value	*Description*
TRUE	The data view contains at least one constraint that must be fully specified before the query is performed.
FALSE	The data view does not contain any constraints that must be fully specified before the query is performed.

ATKDataView::IsParameterList

ATKDataView::IsParameterList checks if the parameter can be assigned only to a value that the data view author has specified in Inspector.

Bool IsParameterList(
BSTR *parameterName***)**

Parameter

IsParameterList accepts this parameter:

Parameter	*Description*
parameterName	The name of the parameter you want to check.

Return values

IsParameterList can return the following values:

Return Value	*Description*
TRUE	The *parameterName* can be assigned only to a value that the data view author has specified in Inspector.
FALSE	The *parameterName* can be assigned to any value.

Comments

IsParameterList retrieves this information from the data view definition in the ATK metaknowledge.

ATKDataView::IsParameterMultipleSelection

ATKDataView::IsParameterMultipleSelection checks if the client can perform a multiple selection on the list of possible parameter values.

Bool IsParameterMultipleSelection(
BSTR *parameterName***)**

Parameter

IsParameterMultipleSelection accepts this parameter:

Parameter	*Description*
parameterName	The name of the parameter that you want to check.

Return values

IsParameterMultipleSelection can return the following values:

Return Value	*Description*
TRUE	The client can specify multiple choices for the specified *parameterName*.
FALSE	The client can perform a single selection on the list of possible *parameterName* values.

Comments

When defining the query in Inspector, the data view author determines if a multiple selection or single selection is possible. **IsParameterMultipleSelection** retrieves this information from the data view definition in the metaknowledge.

ATKDataView::IsParameterSorted

ATKDataView::IsParameterSorted checks if the list of acceptable parameter values must be sorted.

Bool IsParameterSorted(
BSTR *parameterName***)**

Parameter

IsParameterSorted accepts this parameter:

Parameter	*Description*
parameterName	The name of the parameter that you want to check.

Return values

IsParameterSorted can return the following values:

Return Value	*Description*
TRUE	The list of acceptable *parameterName* values must be sorted.

Return Value	*Description*
FALSE	The list of acceptable *parameterName* values does not require sorting.

Comments

When defining the data view in Inspector, the author determines if a sort will be required. **IsParameterSorted** retrieves this information from the data view definition in the metaknowledge.

ATKDataView::PerformFiltering

ATKDataView::PerformFiltering returns an **ATKReferences** object that contains a list of objects that satisfy the query and any constraints and/or parameters.

ATKReferences PerformFiltering

Return value

ATKReferences

Comments

If there is no filter defined for the data view, **PerformFiltering** returns the entire ObjectStore collection from the data view.

This is the simplest means of traversing a collection.

To retrieve the object representations rather than the objects themselves (for example, when filling a table), using **ATKDataView::GetATKTable** to return an **ATKTable** object is more efficient.

ATKDataView::SetParameter

ATKDataView::SetParameter sets the *parameterName* to the specified *value* before executing the query.

void SetParameter(
BSTR *parameterName*, **VARIANT** *value*)

Parameters

SetParameter accepts these parameters:

Parameter	*Description*
parameterName	The name of the parameter whose value you want to set.
value	The value to which the parameter must be set.

Comments

SetParameter accepts different **VARIANT** types and translates each to the required parameter type. If *parameterName* accepts multiple selection, **SetParameter** accepts an array of values and binds the list of *value*s to the *parameterName*.

ATKDataViews

ATKDataViews is a collection of **ATKDataView** objects.

Property

ATKDataViews contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKDataViews contains these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKDataViews::Count

ATKDataViews::Count returns the number of **ATKDataView** objects in the collection.

long Count

Return value

A **long** representing the number of objects.

ATKDataViews::Item

ATKDataViews::Item retrieves a specific **ATKDataView** object from the collection.

ATKDataView Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Identifies an object in the collection. A **Long** value specifies the position of the required object in the collection. A **BSTR** value specifies the name of the object.

Return value

ATKDataView

ATKInstanceFormat

ATKInstanceFormat stores information about a given instance format, as it was created in Inspector. Use **ATKInstanceFormat** to access the definition of an instance format that was defined in Inspector and is contained in the ATK metaknowledge. You can also retrieve the name of the instance format and the names of the column headers defined in the instance format.

Methods

ATKInstanceFormat contains these methods:

Method	*Description*
GetFormatName	Returns the name of the instance format, as it was defined in Inspector.
GetHeaders	Returns a list of strings that represent the names of the column headers in a given instance format.

ATKInstanceFormat::GetFormatName

ATKInstanceFormat::GetFormatName returns the name of the instance format, as it is stored in the ATK metaknowledge.

BSTR GetFormatName

Return value

A string containing the name of the instance format.

ATKInstanceFormat::GetHeaders

ATKInstanceFormat::GetHeaders returns an **ATKStrings** object that contains the headers of the columns in the instance format.

ATKStrings GetHeaders

Return value

ATKStrings

ATKKernel

ATKKernel is the main entry point to the ATK ActiveX object model. Use **ATKKernel** to open an ObjectStore database, to refresh the ATK metaknowledge (that is, to make ATK aware of any change you have made through Inspector), and to obtain an **ATKReference** object from a string representation of a persistent object. When you create an **ATKKernel.Document** ActiveX object, you actually obtain an **ATKKernel** object.

Methods

ATKKernel contains these methods:

Method	*Description*
OpenDatabase	Opens a database.
ReloadMetaKnowledge	Causes the ActiveX server to refresh the metaknowledge for all cached databases.
ResolveReference	Returns an **ATKReference** object for a specific *reference*.

ATKKernel::OpenDatabase

ATKKernel::OpenDatabase opens the database specified by *databaseName* and returns an **ATKDatabase** object.

ATKDatabase OpenDatabase(
BSTR *databaseName* **[, BSTR** *userName***] [, BSTR** *passWord***]**
[, BSTR *import***])**

Parameter

OpenDatabase accepts these parameters:

Parameter	*Description*
databaseName	The full pathname of the database.
userName	Optional. The user name that ATK uses to access the ObjectStore client and connect to the ObjectStore database.
passWord	Optional. The password that ATK uses to access the ObjectStore client and connect to the ObjectStore database.
import	Optional. The name of the database from which ATK should import the metaknowledge.

Return value

ATKDatabase

Comments

The *databaseName* can contain any ObjectStore DB path:

- A file database, such as **c:\odi\mydb.db**
- A remote database, such as **myHost:/dbs/my.db**
- An ObjectStore rawfs database, such as **myHost::/dbs/my.db**

ATKKernel::ReloadMetaKnowledge

ATKKernel::ReloadMetaKnowledge forces the ActiveX server to refresh the metaknowledge stored for all currently cached databases.

void ReloadMetaKnowledge

ATKKernel::ResolveReference

ATKKernel::ResolveReference returns an **ATKReference** object specified by *reference*.

ATKReference ResolveReference(
BSTR *reference***)**

Parameter

ResolveReference accepts this parameter:

Parameter	***Description***
reference	An ObjectStore dumped **os_reference**.

Return value

ATKReference

Comments

An **os_reference** and its dumped representation identify a persistent object inside a database. Refer to the ObjectStore documentation for further details.

The *reference* must be in the same format as the object returned by **os_reference::dump()**.

ResolveReference requires that the *reference* meet these conditions:

- The format is correct.
- ATK can reach and open the database.
- The coordinates provided correspond to a persistent object.

ATKMethod

ATKMethod is used to retrieve information about a user-defined method.

Methods

ATKMethod exposes these methods:

Method	*Description*
GetArguments	Returns a map of argument-name / argument-value that you can populate.
GetCategory()	Returns the category to which the user-defined method belongs.
GetClass()	Returns the name of the persistent class that defines the method.
GetName()	Returns the unmangled name of the registered user-defined method.
GetReturnType()	Returns the method type.
IsStatic()	Returns **TRUE** or **FALSE** based on whether or not the user-defined method is defined as static.
IsVirtual()	Returns **TRUE** or **FALSE** based on whether or not the user-defined method is defined as virtual.

ATKMethod::GetArguments()

ATKMethod::GetArguments returns a map of argument-name / argument-value that you can populate and pass to the function used to invoke the method. The map is returned with the proper keys filled, but with no associated values.

ATKMethodArguments GetArguments()

Return value

ATKMethodArguments

ATKMethod::GetCategory()

ATKMethod::GetCategory returns the category name of the method.

BSTR GetCategory()

Return values

GetCategory can return the following values:

- Create
  - **create_in_database**
  - **create_in_segment**
- Read
- Update
- Delete

ATKMethod::GetClass()

ATKMethod::GetClass returns the name of the persistent class which defines the method.

BSTR GetClass()

Return value

GetCategory returns a string representing the class name.

ATKMethod::GetName()

ATKMethod::GetName returns the unmangled name of the registered user-defined method.

BSTR GetName()

Return value

GetCategory returns a string representing the user-defined method.

ATKMethod::GetReturnType()

ATKMethod::GetReturnType returns the method return type.

BSTR GetReturnType()

Return value

GetReturnType can return the following values:

- *PersistentClassName**
- **os_boolean**
- **os_Collection***<PersistentClassName*>*
- C++ native types (**int**, **unsigned long**, and so on)

ATKMethod::IsStatic()

ATKMethod::IsStatic returns **TRUE** or **FALSE** based on whether or not the user-defined method is defined as static.

Bool IsStatic

Return value

IsStatic can return the following values:

Return Value	*Description*
TRUE	The user-defined method is defined as static.
FALSE	The user-defined method is not defined as static.

ATKMethod::IsVirtual()

ATKMethods::IsVirtual returns **TRUE** or **FALSE** based on whether or not the user-defined method is defined as virtual.

Bool IsVirtual

Return value

IsVirtual can return the following values:

Return Value	*Description*
TRUE	The user-defined method is defined as virtual.
FALSE	The user-defined method is not defined as virtual.

ATKMethods

ATKMethods is a collection of **ATKMethod** objects.

Property

ATKMethods contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKMethods contains these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKMethods::Count

ATKMethods::Count returns the number of **ATKMethod** objects in the collection.

long Count

Return value

A **long** representing the number of objects.

ATKMethods::Item

ATKMethods::Item retrieves a specific **ATKMethod** object from the collection.

ATKMethod Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Identifies an object in the collection. A **Long** value specifies the position of the required object in the collection. A **BSTR** value specifies the name of the object.

Return value

ATKMethod

ATKMethodArguments

ATKMethodArguments is a type of map used to pass arguments to user-defined methods. This class must be transiently created, which you can do

- Using Visual Basic (**Dim params as New ATKMethodArguments**, for example)
- Using the object returned by **ATKMethod::GetArguments()**.

Methods

ATKMethodArguments contains these methods:

Method	*Description*
Add	Adds a key / value pair to the map.
Count	Returns the number of key / value pairs contained in the map.
Item	Returns the value associated with the specified key.
Remove	Removes the key / value pair associated with the specified key.

ATKMethodArguments::Add()

ATKMethodArguments::Add adds a key / value pair to the map.

void Add(BSTR key, VARIANT value)

ATKMethodArguments::Count()

ATKMethodArguments::Count returns the number of key / value pairs contained in the map.

long Count()

Return value

A **long** representing the number of objects.

ATKMethodArguments::Item()

ATKMethodArguments::Item returns the value associated with the specified key.

VARIANT Item(BSTR key)

Return value

A **VARIANT**representing the key value.

ATKMethodArguments::Remove()

ATKMethodArguments::Remove removes the key / value pair associated with the specified key.

Bool Remove (BSTR key)

Return values

Remove can return the following values:

Return Value	*Description*
TRUE	The operation was completed successfully.
FALSE	The operation was not completed successfully.

ATKObjectManager

ATKObjectManager is the ATK-ActiveX interface built on top of ObjectStore multimedia Object Managers. Use **ATKObjectManager** to retrieve information specific to an Object Manager and to access its data directly.

Call **QueryInterface** to obtain an **IStream**, and handle it as a standard COM stream.

Methods

ATKObjectManager contains these methods:

Method	*Description*
GetDataArray	Returns a safe array containing the data associated with the Object Manager.
GetDataSize	Returns the length of the data field associated with the Object Manager.
GetOMMimeType	Returns the MIME type defined for this Object Manager.
GetOMTypeName	Returns the name of the ObjectManager that this object represents.
GetStream	Returns the **IStream** containing the data associated with the Object Manager.

ATKObjectManager::GetDataArray

ATKObjectManager::GetDataArray returns a safe array that contains the data associated with the Object Manager.

VARIANT GetDataArray

Return value

VT_ARRAY|VT_UI1

GetDataArray returns an array of bytes that contains a copy of the multimedia Object Manager data.

ATKObjectManager::GetDataSize

ATKObjectManager::GetDataSize returns the length of the multimedia data field associated with the Object Manager.

long GetDataSize

Return value

The numeric length of the multimedia field.

ATKObjectManager::GetOMMimeType

ATKObjectManager::GetOMMimeType returns the MIME type defined for this Object Manager.

BSTR GetOMMimeType

Return value

A string containing the MIME type.

ATKObjectManager::GetOMTypeName

ATKObjectManager::GetOMTypeName returns the name of the multimedia ObjectManager type that this object represents.

BSTR GetOMTypeName

Return values

GetOMMimeType can return the following values:

- **Text**
- **HTML**
- **Java**
- **Audio**
- **Video**
- **Image -** *<Type> <width>***x***<height>*; for example: **"Image - GIF 120x200"**

ATKObjectManager::GetStream

ATKObjectManager::GetStream returns the **IStream** containing the data associated with the Object Manager.

IStream GetStream

Return value

A stream interface.

Comments

Use **GetStream** to access the multimedia Object Manager data through an **IStream** interface.

ATKReference

An **ATKReference** object is the ATK ActiveX representation of a generic persistent ObjectStore object. Although **ATKReference** can refer to any persistent object, it determines if a specific object is an ObjectStore collection or an Object Manager. Using **ATKReference**, you can also retrieve an object's tabular representation and persistent data member values, including multimedia content.

Methods

ATKReference contains these methods:

Method	*Description*
DeleteObject	Invokes a **delete** user-defined method on a persistent object.
GetATKDatabase	Returns an **ATKDatabase** object for the referenced object.
GetATKObjectManager	Returns an **ATKObjectManager** for the referenced object.
GetCardinality	Returns the cardinality of a collection; returns **1** otherwise.
GetCollectionItems	Traverses a collection and returns an **ATKReferences** object.
GetClass	Returns the **ATKClass** to which an object belongs.
GetReference	Returns the dumped **os_reference** representation of the referenced object.
GetRepresentation	Returns a string containing a flat representation of an object.
GetSlotValue	Returns the value of a specific attribute, or a generic **ATKReference** object.
GetTabularRepresentation	Returns lists of strings that represent the rows in a table.
IsCollection	Checks whether an object is a collection.
IsObjectManager	Checks whether an object is an Object Manager.

Method	*Description*
ReadObject	Invokes a **read** user-defined method on a persistent object.
UpdateObject	Invokes an **update** user-defined method on a persistent object.

ATKReference::DeleteObject

ATKReference::DeleteObject ( ) invokes a **delete** user-defined method on the persistent object represented by **ATKReference**. *DeleteMethodName* is the name of the **delete** method registered for the class to which the current **ATKReference** belongs.

Bool ATKReference::DeleteObject(
BSTR *DeleteMethodName***)**

Return values

Return Value	*Description*
TRUE	The user-defined method returned a non-zero value.
FALSE	The user-defined method returned a zero value.

ATKReference::GetATKDatabase

ATKReference::GetATKDatabase returns an **ATKDatabase** object for the referenced object.

ATKDatabase GetATKDatabase

Return value **ATKDatabase**

Comments **GetATKDatabase** is particularly useful if you use **ATKKernel::ResolveReference** to get an **ATKReference** object.

ATKReference::GetATKObjectManager

ATKReference::GetATKObjectManager returns an **ATKObjectManager** object for the referenced object.

ATKObjectManager GetATKObjectManager

Return value **ATKObjectManager**

Comments **GetATKObjectManager** works only if the persistent object referenced by **ATKReference** is an ObjectStore Object Manager object.

ATKReference::GetCardinality

ATKReference::GetCardinality returns the cardinality of a collection.

long GetCardinality

Return values

GetCardinality can return the following values:

Return Value	*Description*
The collection cardinality	If the object that **ATKReference** refers to is an ObjectStore collection
1	The object is not an ObjectStore collection

ATKReference::GetCollectionItems

ATKReference::GetCollectionItems traverses a collection and returns an **ATKReferences** object that contains a collection of **ATKReference** objects.

ATKReferences GetCollectionItems

Return value

ATKReferences

Comments

GetCollectionItems works only if the persistent object referenced by **ATKReference** is an ObjectStore collection.

ATKReference::GetClass

ATKReference::GetClass returns the **ATKClass** to which an object belongs.

ATKClass GetClass

Return value

ATKClass

Comments

GetClass joins the persistent objects and schema knowledge of the ATK object model. If the current **ATKReference** object refers to an instance of a class in the database schema, you can retrieve information about the declaration of that class by calling **GetClass**.

ATKReference::GetReference

ATKReference::GetReference returns the dumped **os_reference** representation of the referenced object.

BSTR GetReference

Return value — The **os_reference** representation of the referenced object.

Comments — The return value is in **os_reference::dump()** format.

You can use the returned value when calling **ATKKernel::ResolveReference**.

ATKReference::GetRepresentation

ATKReference::GetRepresentation returns a string containing a flat representation of an object.

BSTR GetRepresentation[(
BSTR *instanceFormatName*)**]**

Parameter — **GetRepresentation** accepts this parameter:

Parameter	***Description***
instanceFormatName	Optional. The name of the instance format from which you want to retrieve the object.

Return value — A string representation of the object.

Comments — If *instanceFormat* is "" (an empty string) or is not supplied, **GetRepresentation** uses the default instance format of the class to which the object belongs.

ATKReference::GetSlotValue

ATKReference::GetSlotValue returns the value of the data member *attributeName* from the persistent object.

VARIANT GetSlotValue(
BSTR *attributeName*)**

Parameter — **GetSlotValue** accepts this parameter:

Parameter	***Description***
attributeName	The name of the data member whose value you want to retrieve.

Return values

VARIANT

If *attributeName* is a data member of the class to which the object referenced by **ATKReference** belongs, **GetSlotValue** returns a **VARIANT**.

Return Value	*Description*
Long	If the data member specified by *attributeName* is a numeric integer data member, the **VARIANT** is a **Long** containing the value of the field.
Double	If the data member is a numeric floating-point data member, the **VARIANT** is a **Double** containing the value of the field.
BSTR	If the data member is a string, the **VARIANT** is a **BSTR** containing a copy of the value of the field.
ATKReference	If the data member is a relationship, the **VARIANT** is an **ATKReference** object representing the related objects.

Comments

The *attributeName* must be an attribute of the referenced object's class. **GetSlotValue** processes only the values of integer data member types (**string** and **number**) and of relationships.

ATKReference::GetTabularRepresentation

ATKReference::GetTabularRepresentation returns an **ATKStringsList** object, which contains a collection of strings that represent the rows in a table containing the referenced object.

ATKStringsList GetTabularRepresentation[(
BSTR *instanceFormatName*)**]**

Parameter

GetTabularRepresentation accepts this parameter:

Parameter	*Description*
instanceFormatName	Optional. The name of the instance format, defined in the ATK metaknowledge, that ATK uses to format the object referenced by **ATKReference**.

Return value

ATKStringsList

For example, suppose **ATKReference** references a **Customer** instance, and the specified instance format instructs ATK to show the **Customer name** and the **make** and **model** of the cars the customer owns. The **ATKStringsList** object contains this output in tabular format:

Name	*Make*	*Model*
John, Smith	Ford	Escort
	Cadillac	DeVille

Comments

The data in the strings is presented according to the specified *instanceFormatName*, which indicates the navigation pattern of the data in the object. If *instanceFormatName* is "" (an empty string) or is not supplied, **GetRepresentation** uses the default instance format of the class to which the object belongs.

ATKReference::IsCollection

ATKReference::IsCollection checks if an object has been classified as a collection.

Bool IsCollection

Return values

IsCollection can return the following values:

Return Value	*Description*
TRUE	The object is an ObjectStore collection.
FALSE	The object is not an ObjectStore collection.

ATKReference::IsObjectManager

ATKReference::IsObjectManager checks if the object has been classified as an Object Manager (for example, image, video, audio, HTML).

Bool IsObjectManager

Return values

IsObjectManager can return the following values:

Return Value	*Description*
TRUE	The object is an ObjectStore Object Manager.
FALSE	The object is not an ObjectStore Object Manager.

ATKReference::ReadObject

ATKReference::ReadObject ( ) invokes a **read** user-defined method on the persitent object represented by **ATKReference**. *ReadMethodName* is the name of the **read** method registered for the class to which the current **ATKReference** belongs.

VARIANT ATKReference::ReadObject(
BSTR *ReadMethodName***)**

Return value

The function returns the value returned by the invoked user-defined method. The type of the returned value varies according to the user-defined method that is called.

ATKReference::UpdateObject

ATKReference::UpdateObject ( ) invokes an **update** user-defined method on the persitent object represented by **ATKReference**. *UpdateMethodName* is the name of the **update** method registered for the class to which the current **ATKReference** belongs. **Arguments** is a map of argument-name / argument-value pairs used to pass arguments to the method.

Bool ATKReference::UpdateObject(
BSTR *UpdateMethodName***,**
IATKMethodArguments* Arguments)

Return values

UpdateObject can return the following values:

Return Value	*Description*
TRUE	The user-defined method returned a non-zero value.
FALSE	The user-defined method returned a zero value.

ATKReferences

ATKReferences is a collection of **ATKReference** objects. Because this class is the ATK equivalent of an ObjectStore collection, you can use an **ATKReferences** object to retrieve all the items contained in the underlying ObjectStore collection, and to create **ATKDataView** and **ATKTable** objects based on the same collection.

Property

ATKReferences contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKReferences exposes these methods:

Method	*Description*
Add	Adds a single persistent object, or a collection of persistent objects to the current **ATKReferences** object.
GetATKDataView	Returns an **ATKDataView** associated with the *dataViewExpression*.
GetATKTable	Returns an **ATKTable** associated with the *instanceFormatName*.
Item	Returns the specified item in the collection.
Remove	Removes a single persistent object, or a collection of persistent objects from the current **ATKReferences** object.

ATKReferences::Add

ATKReferences::Add adds a single persistent object represented by **ATKReference**, or a collection of persistent objects contained in the *ItemCollection* set, to the current **ATKReferences** object.

ATKReferences::Add(ATKReference* Item)

ATKReferences::Add(ATKReferences* ItemCollection)

Comments

The **ATKReferences** instance on which these methods are invoked must be transiently allocated. Such instances can be built using the Visual Basic **new** operator or with **CreateObject**. An error is thrown if the **ATKReferences** is not transiently allocated.

ATKReferences::Count

ATKReferences::Count returns the number of **ATKReference** objects contained in the collection.

long Count

Return value

A **long** representing the number of objects.

ATKReferences::GetATKDataView

ATKReferences::GetATKDataView returns an **ATKDataView** object that contains the data view definition specified by *dataViewExpression*.

ATKDataView GetATKDataView(
BSTR *dataViewExpression***)**

Parameter

GetATKDataView accepts this parameter:

Parameter	*Description*
dataViewExpression	The data view, as defined in the ATK metaknowledge, that ATK opens on top of the collection referenced by **ATKReferences**.

Return value

ATKDataView

Comments

The *dataViewExpression* parameter can also contain an SQL command that follows the syntax described in Chapter 3, Active Toolkit OLE DB Provider.

ATKReferences::GetATKTable

ATKReferences::GetATKTable returns an **ATKTable** object in the specified *instanceFormatName*.

ATKTable GetATKTable[(
BSTR *instanceFormatName***)]**

Parameter

GetATKTable accepts this parameter:

Parameter	*Description*
instanceFormatName	Optional. The name of the instance format, as defined in the ATK metaknowledge, that ATK uses to format the table referenced by **ATKReferences**.

Return value

ATKTable

Comments

If *instanceFormatName* is "" (an empty string) or is not supplied, **ATKReferences::GetATKTable** uses the default instance format for the class.

This is the most efficient means of accessing the tabular format of a collection through an **ATKReferences** object.

ATKReferences::Item

ATKReferences::Item retrieves a specific **ATKReference** object from the collection.

ATKReference Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Specifies the position of the object in the collection.

Return value

ATKReference

ATKReferences::Remove

ATKReferences::Remove ( ) removes a single persistent object represented by **ATKReference**, or a collection of persistent objects contained in the *ItemCollection* set, to the current **ATKReferences** object.

ATKReferences::Remove(ATKReference* Item)

ATKReferences::Remove(ATKReferences* ItemCollection)

Return value

Remove can return the following values:

Return Value	*Description*
TRUE	The function is successfully completed.
FALSE	The function does not complete successfully.

Comments

The **ATKReferences** instance on which these methods are invoked must be transiently allocated. Such instances can be built using the Visual Basic **new** operator or with **CreateObject**. An error is thrown if the **ATKReferences** is not transiently allocated.

ATKRoot

ATKRoot provides access to the ObjectStore roots defined in a database. By using an **ATKRoot** object, you can directly access the persistent objects.

Methods

ATKRoot contains these methods:

Method	*Description*
GetATKReference	Returns the **ATKReference** object associated with the root.
GetATKTable	Returns the **ATKTable** representing a collection.
GetName	Returns the name of the root.
GetReference	Returns a string that contains the value associated with the root.

ATKRoot::GetATKReference

ATKRoot::GetATKReference returns the **ATKReference** object that represents the persistent object associated with the root.

ATKReference GetATKReference

Return value

ATKReference

ATKRoot::GetATKTable

ATKRoot::GetATKTable returns the **ATKTable** object that represents the ObjectStore collection associated with the referenced root, in the specified instance format.

ATKTable GetATKTable[(
BSTR *instanceFormatName*)**]**

Parameter

GetATKTable accepts this parameter:

Parameter	*Description*
instanceFormatName	Optional. The name of the instance format, as defined in the metaknowledge, that ATK uses to format the table.

Return value

ATKTable

Comments

If you know that the persistent object associated with the root is an ObjectStore collection, use this function to create an **ATKTable** object containing the tabular representation of the collection.

If *instanceFormatName* is "" (an empty string) or is not supplied, **GetATKTable** uses the default instance format for the class to which the items in the collection belong.

ATKRoot::GetName

ATKRoot::GetName returns the symbolic name of the root, as it is stored in the ObjectStore database.

BSTR GetName

Return value

A string that contains the name of the root.

ATKRoot::GetReference

ATKRoot::GetReference returns a string representation of the persistent object associated with the root.

BSTR GetReference

Return value

A string in **os_reference::dump()** format that contains the value of the persistent object.

ATKRoots

ATKRoots is a collection of **ATKRoot** objects.

Property

ATKRoots contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKRoots contains these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKRoots::Count

ATKRoots::Count returns the number of **ATKRoot** objects in the collection.

long Count

Return value

A **long** representing the number of objects.

ATKRoots::Item

ATKRoots::Item retrieves a specific **ATKRoot** object from the collection.

ATKRoot Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Identifies an object in the collection. A **Long** value specifies the position of the required object in the collection. A **BSTR** value specifies the name of the object.

Return value

ATKRoot

ATKStrings

ATKStrings is a collection of **BSTR** object types.

Property

ATKStrings contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKStrings contains these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKStrings::Count

ATKStrings::Count returns the number of **BSTR** values in the collection.

long Count

Return value

A **long** representing the number of string objects.

ATKStrings::Item

ATKStrings::Item retrieves a specific **BSTR** value from the collection.

BSTR Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Specifies the position of the required string in the collection.

Return value

A string value.

ATKStringsList

ATKStringsList is a collection of **ATKStrings** objects.

Property

ATKStringsList contains this property:

Property	*Description*
Count	Returns the number of objects in the collection.

Methods

ATKStringsList exposes these methods:

Method	*Description*
Count	Returns the number of objects in the collection.
Item	Returns the specified item in the collection.

ATKStringsList::Count

ATKStringsList::Count returns the number of **ATKStrings** objects in the collection.

long Count

Return value

A **long** representing the number of string objects.

ATKStringsList::Item

ATKStringsList::Item retrieves a specific **ATKStrings** object from the collection.

ATKStrings Item(
VARIANT *item***)**

Parameter

Item accepts this parameter:

Parameter	*Description*
item	Specifies the position of the required list of strings in the collection.

Return value

ATKStrings

ATKTable

ATKTable is a collection of objects that corresponds to the underlying ObjectStore collection. An **ATKTable** object is strictly related to the instance format used to create it.

ATKTable provides two kinds of string representations for an object:

- A flat representation that is contained in a single line
- A grid-like representation, which can be expressed in multiple lines

You can use an **ATKTable** object to retrieve the content of the ObjectStore collection. For each object in the collection, you can retrieve the rows corresponding to its tabular representation in the instance format you specify. **ATKTable** also provides an easy, efficient means of adding data to a table using representations of the objects contained in a collection.

Methods

ATKTable contains these methods:

Method	*Description*
GetCardinality	Returns the cardinality of a collection.
GetHeaders	Returns a list of the column headers in the instance format.
GetObject	Returns the **ATKReference** object that the **ATKTable** cursor points to.
GetReference	Returns the reference string representation of the object that the **ATKTable** cursor points to.
GetRepresentation	Returns the flat representation of the object that the **ATKTable** cursor points to.
GetTabularRepresentation	Returns lists of strings that represent the rows in a table associated with the object that the **ATKTable** cursor points to.
GetWholeHTML	Returns a string containing the default HTML representation of the entire table.

Method	*Description*
GetWholeXML	Returns a buffer containing the complete XML representation of the table.
IsBOT	Checks if the **ATKTable** cursor has reached the beginning of the collection.
IsEOT	Checks if the **ATKTable** cursor has reached the end of the collection.
MoveFirst	Moves the **ATKTable** cursor to the first item in the collection.
MoveLast	Moves the **ATKTable** cursor to the last item in the collection.
MoveNext	Moves the **ATKTable** cursor to the next item in the collection.
MovePrevious	Moves the **ATKTable** cursor to the previous item in the collection.
MoveTo	Moves the **ATKTable** cursor to a specific position in the collection.

ATKTable::GetCardinality

ATKTable::GetCardinality returns the cardinality of the collection.

long GetCardinality

Return value — The number of objects in the collection.

Comments — Because **GetTabularRepresentation** can return more than one row per object, **GetCardinality** might return a value that is smaller than the actual number of rows in the table.

ATKTable::GetHeaders

ATKTable::GetHeaders returns an **ATKStrings** object that contains a list of strings that are the column headers in the instance format.

ATKStrings GetHeaders

Return value — **ATKStrings**

ATKTable::GetObject

ATKTable::GetObject returns the **ATKReference** object representing the persistent object that the **ATKTable** cursor points to.

ATKReference GetObject

Return value **ATKReference**

ATKTable::GetReference

ATKTable::GetReference returns the reference string representation of the object that the **ATKTable** cursor points to.

BSTR GetReference

Return value A string in **os_reference::dump()** format that contains the value of the persistent object.

ATKTable::GetRepresentation

ATKTable::GetRepresentation returns the flat representation of the object that the **ATKTable** cursor points to.

BSTR GetRepresentation

Return value A string representation of the table.

Comments **ATKTable::GetRepresentation** uses the instance format associated with the **ATKTable** object to compute this representation.

ATKTable::GetTabularRepresentation

ATKTable::GetTabularRepresentation returns a list of strings that represent the object that the **ATKTable** cursor points to. **GetTabularRepresentation** presents the data in the strings according to the specified *instanceFormat*, which indicates the navigation pattern of the data in the object.

ATKStringsList GetTabularRepresentation

Return value

ATKStringsList

The collection of collections of strings contains the tabular representation of the current table object. For example, the return value could be a matrix:

Name	*Make*	*Model*
John, Smith	Ford	Escort
	Cadillac	DeVille

ATKTable::GetWholeHTML

ATKTable::GetWholeHTML returns a string containing the default representation of the entire HTML table. You can display the table in a web browser.

BSTR GetWholeHTML

Return value

A string representation of the entire table in HTML.

Comments

Use **GetWholeHTML** to retrieve an HTML version of the table. Because **GetWholeHTML** returns a BLOB containing the entire table, Object Design recommends that you use this method as a prototype tool, and on tables that contain only a small sample of data.

ATKTable::GetWholeXML

ATKTable::GetWholeXML returns a buffer containing the complete XML representation of the table.

BSTR ATKTable::GetWholeXML

Return value

A string representation of the entire table in HTML.

Comments

Use **GetWholeXML** to retrieve an HTML version of the table. Because **GetWholeXML** returns a BLOB containing the entire table, Object Design recommends that you use this method as a prototype tool, and on tables that contain only a small sample of data.

ATKTable::IsBOT

ATKTable::IsBOT checks if the **ATKTable** cursor is at the first object in the collection.

Bool IsBOT

Return values

IsBot can return the following values:

Return Value	*Description*
TRUE	The **ATKTable** cursor is at the first object in the collection.
FALSE	The **ATKTable** cursor is located in the collection but not at the first object.

ATKTable::IsEOT

ATKTable::IsEOT checks if the **ATKTable** cursor is at the last object in the collection.

Bool IsEOT

Return values

IsEot can return the following values:

Return Value	*Description*
TRUE	The **ATKTable** cursor is at the last object in the collection.
FALSE	The **ATKTable** cursor is located in the collection but not at the last object.

ATKTable::MoveFirst

ATKTable::MoveFirst moves the **ATKTable** cursor to the first object in the collection.

void MoveFirst

Comments

By default, the **ATKTable** cursor starts in this position.

ATKTable::MoveLast

ATKTable::MoveLast moves the **ATKTable** cursor to the last object in the collection.

void MoveLast

ATKTable::MoveNext

ATKTable::MoveNext moves the **ATKTable** cursor to the next object in the collection.

void MoveNext

ATKTable::MovePrevious

ATKTable::MovePrevious moves the **ATKTable** cursor to the previous object in the collection.

void MovePrevious

ATKTable::MoveTo

ATKTable::MoveTo moves the **ATKTable** cursor to a specific object in the collection.

void MoveTo(
long *position***)**

Parameter

MoveTo accepts this parameter:

Parameter	***Description***
position	The object number to which you want to move the table cursor. The first object in a collection is numbered 0.

Index

A

B

C

D

E

G

I

J

M

O

P

R

S

V