OLE DB for the Context Interchange System

by

Chiuyi Chan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 1, 2000

Copyright 2000 Chiuyi Chan. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science
September 1, 2000

Certified by

Stuart E. Madnick
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
OLE DB for the Context Interchange System

by

Chiuyi Chan

Submitted to the

Department of Electrical Engineering and Computer Science

September 1, 2000

In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The OLE DB development is an extension to the Context Interchange System that aims to provide semantic interoperability between heterogeneous data sources. The development of OLE DB Provider is a front-end extension, which provides users in the Windows working environment the capability to access the Cameleon Web Wrapping Engine using Windows applications. The current implementation of the OLE DB Provider not only allows users to use Microsoft SQL Server to access Cameleon Engine, but also enhances the query functionality of web data sources in the Context Interchange System with SQL Server. The primary goal of the thesis is to present the design and implementation of the OLE DB Provider. The development of OLE DB Provider will also serve as a guideline for future integration between common applications and the Context Interchange System.

Thesis Supervisor: Stuart E. Madnick

Title: John Norris Maguire Professor of Information Technology &
Leaders for Manufacturing Professor of Management Science
MIT Sloan School of Management
# Table of Contents

**ABSTRACT** .......................................................................................................................... 2

**Table of Contents**.................................................................................................................. 3

**List of Figures** ......................................................................................................................... 7

**List of Tables** .......................................................................................................................... 8

**Chapter 1 Introduction**.......................................................................................................... 9

1.1 Context Interchange Framework ....................................................................................... 10

1.2 Motivation and Goal .............................................................................................................. 10

1.3 Organization of Thesis ......................................................................................................... 11

**Chapter 2 Background**........................................................................................................ 12

2.1 Context Interchange System ............................................................................................... 12

2.1.1 Context Mediator ............................................................................................................. 12

2.1.2 Web Wrapper .................................................................................................................. 12

2.2 Data Access Technology .................................................................................................... 13

2.2.1 Microsoft Universal Data Access .................................................................................. 13

2.2.2 Microsoft Component Object Model (COM) ............................................................... 13

2.2.3 OLE DB ................................................................................................................................ 14

OLE DB components ................................................................................................................ 15

OLE DB Core Components ....................................................................................................... 16

Comparison of OLE DB and ODBC .......................................................................................... 17

2.3 Extensible Markup Language (XML) ................................................................................ 18

**Chapter 3 Design of the OLE DB Provider for Cameleon** .................................................... 20

3.1 Role of the Provider .............................................................................................................. 20

3.2 Structure of the Provider ....................................................................................................... 21

Object ......................................................................................................................................... 21

Interface ...................................................................................................................................... 21

Object ......................................................................................................................................... 22

Interface ...................................................................................................................................... 22

3.2.1 Properties .......................................................................................................................... 23

3.2.2 Data Source Object .......................................................................................................... 23

Initialization ................................................................................................................................. 24

Setting Data Source properties .................................................................................................. 24

Create Session ............................................................................................................................. 25

3.2.3 Session Object .................................................................................................................. 25

Schema Rowset ............................................................................................................................ 25

Creating Schema Rowsets .......................................................................................................... 28

Creating Rowsets and Commands ............................................................................................ 29

Getting Pointer to Data Source object ........................................................................................ 29

Session Properties ....................................................................................................................... 30
Chapter 4 Implementation

4.1 Visual C++ 6.0 ATL Provider Wizard ......................................................... 38
  4.1.1 C++ Templates .................................................................................... 39
  4.1.2 Active Template Library (ATL) ............................................................. 40
    Special Data Types in ATL ........................................................................ 41
    Smart Pointers: CComPtr and CComQIPtr ................................................. 41
    BSTR ........................................................................................................ 42
    VARIANT ................................................................................................. 44
  4.1.3 ATL Wizard ........................................................................................ 44

4.2 Implementation of Cameleon Provider using the ATL Provider Wizard ...... 45
  4.2.1 Wizard generated OLE DB Provider ..................................................... 45
    General Organization ................................................................................ 45
    The COM Server Module ........................................................................ 46
    Self-Registration ....................................................................................... 46
    DLL Activation and Object Creation ....................................................... 47
    Server Lifetime Management .................................................................... 47
  4.2.2 Implementation of Cameleon Provider components using the ATL Wizard... 48
    Component inheritance model ................................................................... 48
    Object Interface Map ............................................................................... 49
    Implementation of individual object ....................................................... 50
    Data Source object .................................................................................. 50
    Session Object ....................................................................................... 52
    Command Object and Rowset Object ..................................................... 54

4.3 Displaying Result Rowset in Cameleon Provider ...................................... 56
  4.3.1 Rowset and Command with static binding of data columns ................. 56
  4.3.2 Rowset and Command with dynamic binding of data columns ............ 57
    The CCamProvRecord class ..................................................................... 57
    The CCamProvArray class ...................................................................... 58
    FillColInfo method in Command ............................................................ 59
    Execute method in Command .................................................................. 60
G.1 Using ad hoc distributed queries in SQL Server Query Analyzer .............. 99
G.2 Adding Linked Server ....................................................................................... 101
G.3 Creating View and using Access to retrieve OLE DB data sources .......... 102
References ....................................................................................................................... 105
List of Figures

Figure 2.2-1 OLE DB components............................................................15
Figure 2.3-1 A sample Cameleon XML document output..............................18
Figure 3.1-1 Role of OLE DB Providers.....................................................20
Figure 3.2-1 Mapping between Cameleon Registry and Schema Information of Cameleon Provider.................................................................27
Figure 3.2-2 Fragment of Cameleon Schema information.................................28
Figure 3.3-1 OLE DB Components of Cameleon Provider.................................35
Figure 3.3-2 Detailed Application Flow......................................................36
Figure 3.3-3 Use of Query Interface..........................................................37
Figure 4.1-1 Rewrite of Figure 3.3-4 using CComQIPtr....................................41
Figure 4.1-2 A BSTR encoding the character string “a bstr”..........................42
Figure 4.1-3 Code fragment to shows conversion from VARIANT to BSTR; and from BSTR to CString..........................................................43
Figure 4.2-1 Inheritance chain of Data Source object of the Cameleon Provider.....48
Figure 4.2-2 Interface Map of the Data Source object of the Cameleon Provider...49
Figure 4.2-3 Property Map of Data Source object of the Cameleon Provider........50
Figure 4.2-4 Sample Column Map............................................................53
Figure 4.2-5 Schema Map of the Cameleon Provider.....................................54
Figure 4.2-6 Three parts to Cameleon HTTP output.....................................55
Figure 5.2-1 Invoking the Cameleon Provider and supplying with the Cameleon Server path in the Data Source field and the Schema information in the Provider String field...67
Figure 5.2-2 Rowset Viewer displaying Column Schema Rowset information........68
Figure 5.2-3 Rowset Viewer setting command text.......................................68
Figure 5.2-4 Rowset Viewer displaying result rowset after execution of command....69
Figure 5.2-5 Sample distributed query with the use of the OpenRowset function...71
Figure 5.2-6 Query with multiple conditions..............................................73
Figure 5.2-7 Nested queries with special function.......................................73
Figure 5.2-8 A join statement on Cameleon source and SQL Server source........74
Figure 5.2-9 Transact-SQL query to create the moneyrates view....................74
Figure 5.2-10 Sample use of the moneystatus view......................................74
Figure 5.2-11 Query of figure 5.3-1 rewritten using View............................75
Figure 5.2-12 The use of Access data page to access moneyrates data..............75
Figure D-1 Create new OLE DB Provider using ATL COM project...................92
Figure D-2 The ATL Object Wizard.........................................................93
Figure D-3 The ATL Object Wizard Properties window................................93
Figure G.1-1 Execution of the xp_enumoledb_providers function...................100
Figure G.1-2 Execution of ad hoc distributed query in Query Analyzer.............101
Figure G.1-3 Execution of OpenQuery function in Query Analyzer..................104
Figure G.3-1 Data Link Properties window in Access..................................105
List of Tables

Table 3.2-1 Minimum requirement for an OLE DB Provider..........................22
Table 3.2-2 Additional interfaces implemented on Cameleon Provider...............22
Table 3.2-3 Example property sets and property groups...............................23
Table 5.2-1 ODBC Core SQL Grammar conformance.................................77
Chapter 1 Introduction

In the past several years, advances in computer networking and telecommunications have led to the explosive growth in the number of information sources that are being connected. This connectivity has given way to a proliferation of data stored in various databases in a distributed fashion. With this growing abundance of data, the problem of retrieving information and interpreting information becomes an important challenge.

The ability to exchange information physically has taken giant steps forward with the explosive growth in computer networking, but the ability to exchange information in a meaningful manner has lagged significantly behind. The meaning of information is often dependent on a particular context – a context that embodies a number of underlying assumptions. Different contexts of data lead to possible confusion and conflicts when information from heterogeneous sources is brought together. This problem is referred to as the need to semantic interoperability among distributed data sources; any data integration effort must be capable of reconciling possible semantic conflicts [4].

At the same time, the dramatic increase in the amount of different data sources and data format posts another problem in efficient data retrieval and manipulation from different data sources. One example would be the integration of World Wide Web data sources, which contain enormous quantity of information, with traditional Database Management Systems (DBMS). Due to the lack of structure of data on web sites, it often requires a lot of programming effort for an organization to transform web data sources to usable format. Moreover, the rapidly changing nature of web sources makes maintenance extremely hard and cost ineffective. End users may want standard access to all types of data so that they can combine different types of information. However, it would require too much effort to port all useful data to one data source.
1.1 Context Interchange Framework

The Context Interchange (COIN) project aims to provide a solution to users by consolidating data sources and providing a unified view to them. COIN technology presents all the data sources in the form of relational databases by providing generic wrappers for them. The Context Interchange project also seeks to mediate the problem of interoperability using a context mediator.

In the Context Interchange System, the Context Mediator is responsible for detecting and resolving semantic conflicts between data sources and data receivers; while the Wrapper are responsible for providing uniform access to different data sources in the form of queries. In particular, the Cameleon Web Wrapper Engine works as a relational front-end to web sources. The Cameleon Engine treats the Internet as a giant relational database and supports Structured Query Language (SQL) for querying it.

In the current Context Interchange System prototype, a web browser is the main front end for the system. The Context Interchange System uses Structured Query Language (SQL) as the standard of issuing queries to the data sources. In order to provide better interaction between users and the Context Interchange System, supporting other front-end working environments becomes necessary. Ideally, components of the Context Interchange System, such as the Cameleon Web Wrapper Engine and the Context Mediator, should allow complete database-like access, and they should implement the corresponding query functionality. Moreover, other applications that support standard SQL queries, such as Microsoft Access or Microsoft SQL Server, should be able to communicate with different components of the Context Interchange System.

1.2 Motivation and Goal

The goal of this thesis is to investigate the feasibility of providing the Context Interchange System components with query functionality using an industry standard technology, OLE DB, by implementing an OLE DB Provider for the Cameleon Web Wrapper Engine. By providing an OLE DB interface to the Context Interchange System component, any application that complies with the OLE DB standard can access the
COIN components. Therefore, an OLE DB provider for the Cameleon Web Wrapper Engine will enable all OLE DB consumers, such as Microsoft Access, SQL Server and Active Data Object (ADO) to access all the wrapped web sources.

The increase in accessibility of different Context Interchange System components is not the only improvement that an OLE DB interface would bring. By connecting a powerful commercial DBMS as an OLE DB consumer to the Context Interchange System components, the commercial database engine will empower the different components by providing additional relational functionality not yet implemented in the individual components of the Context Interchange System. For example, currently, the Cameleon Web Wrapper Engine does not support sub-query or special functions such as group-by or max. If a sophisticated database engine, such as SQL Server 7.0, is used to drive the Cameleon Web Wrapper Engine through an OLE DB Provider, then SQL Server 7.0 will provide the missing functionality to the Cameleon Engine. As a result, the components of the Context Interchange System do not have to implement their individual front end to support extra query processing. In addition, components of the Context Interchange System can rely on a commercial database engine to enhance their relational capabilities instead of providing all the underlying relational functionality on their own.

Finally, the OLE DB interfaces can turn each of the Context Interchange System component into a standalone data source that allows ordinary database access. In other words, with the help of OLE DB technology, users would then be able to combine data sources from Context Interchange components with regular DBMS data sources.

### 1.3 Organization of Thesis

The rest of the thesis will be organized as follow. In Chapter 2, background information on the Context Interchange System architecture and the idea behind OLE DB will be presented. XML technology will also be discussed in this chapter. Chapter 3 will focus on the design of the OLE DB Provider. The implementation of the OLE DB Provider will be discussed in detail in Chapter 4. Chapter 5 will present the result of testing on the OLE DB Provider. Finally, Chapter 6 will give the conclusion of the thesis. It will give some insights into how the system can be improved, directions for future research and the limitation of the current technology.
Chapter 2 Background

2.1 Context Interchange System

The Context Interchange (COIN) strategy seeks to address the problem of semantic interoperability by consolidating distributed data sources and providing a unified view to them. COIN technology presents all data sources as SQL databases by providing generic wrappers for them. The underlying data integration technology of the COIN model is to use the Context Mediator to detect and resolve semantic conflicts among heterogeneous data sources. Refer to [4] for more information on the Context Interchange System.

2.1.1 Context Mediator

The Context Mediator sits between the users of data and the sources. The meaning and underlying assumptions about the data set are explicitly represented as data contexts. When the need to bring together data of different context arises, it is the job of the Context Mediator to determine semantic conflicts between contexts and apply any transformation necessary to exchange the data in a meaningful way.

The Context Mediator uses Structured Query Language (SQL) as a means of issuing queries to data sources. Mediation is the process of rewriting queries posed in the receiver’s or the client’s context into a new set of queries where all potential conflicts are explicitly solved [4]. Each query issued by the receiver is translated to a source context and then passed on to the data source. The data received is similarly translated to the receiver’s context and presented to the receiver. This process allows users to extract disparate data from different sources and produce consistent results.

2.1.2 Web Wrapper

The web wrapper serves a crucial role in the Context Interchange System, in which it allows the incorporation of data from web sites into the system. In order to achieve seamless integration of aggregated information from different web sites and other data from systems such as relational databases, the current version of the web wrapping engine, Cameleon, is designed to be a relational front-end to web sources.
There are two major components to Cameleon: the relational front-end and the core engine. The relational front-end consists of a planner, optimizer, and an executioner. The front-end takes an SQL query, creates a plan, optimizes it and then runs it using the core engine. The core engine is responsible for locating the appropriate specification files based on the input queries and the registry. Based on the schema information and extraction rules of the specification files, the core engine extracts the right piece of information from the wrapped web sites. The retrieved data is either presented in HTML or XML table format [2]. Currently, the Cameleon Web Wrapper Engine runs as a Java servlet and users can send SQL queries to Cameleon through http or ODBC drivers. They can also call Cameleon through Microsoft Excel by obtaining the Cameleon web query file.

2.2 Data Access Technology

2.2.1 Microsoft Universal Data Access

OLE DB is a part of Microsoft’s Universal Data Access strategy. The goal of Universal Data Access resembles that of the Context Interchange System, which aims to solve the problem of interoperability among heterogeneous data sources within an enterprise. While the Context Interchange System puts emphasis on resolving differences on data meaning via context mediation, Universal Data Access concentrates on providing a consistent programming model for access of any type of data within an organization. More specifically, Universal Data Access provides the tool-independent and language-independent technology platform for access of varieties of relational and non-relational data sources. [12]

2.2.2 Microsoft Component Object Model (COM)

Microsoft’s Component Object Model, also known as COM, is the underlying technology that makes Universal Data Access possible. Effective code reuse has always been a challenge in the software development world. Component Object Model is an alternative to building reusable Dynamic Linked Libraries (DLLs) and other binary components. COM enables the tool-independent and language-independent nature of Universal Data Access. An easy way to picture COM is to think of them as binary object black boxes.
that communicate through a set of standard interfaces, which define the functionality of
the objects. An object can expose multiple interfaces, therefore, exposing different
functionality. Since in COM programming, only interfaces are immutable, it is called
interface-based programming.

COM objects live inside COM servers. COM servers can be an in process DLLs,
which COM client and object share the same address space; or EXEs, which COM client
and object reside in different address space. In other words, COM objects are held
together by COM servers in the form of DLLs or EXEs, and the COM servers provide the
code infrastructure and API for clients to activate COM objects.

Each COM object and interface is identified by a 16 bytes GUID (globally unique
identifier). A COM object or interface needs the same identifier on all machines so that
any clients can use them. In order to create a COM object, the machine looks into the
registry entries to find the components, locates their DLLs and creates the components.
The Microsoft Data Access Components (MDAC), are the set of components that enable
the Universal Data Access strategy, they include ODBC, OLE DB and ActiveX Data
Object (ADO). [14]

2.2.3 OLE DB

OLE DB is a major part of the Universal Data Access architecture. It is a specification
for a set of COM-based data access interfaces that encapsulate various data management
services. OLE DB provides a uniform way to access different types of information
sources, and it also provides the ability to implement additional database services.

OLE DB partitions the functionality of a traditional relational database into
logical components, and the events needed for those to communicate. It does so by
defining an open, extensible collection of the COM interfaces that encapsulate reusable
portions of the DBMS functionality. These interfaces define different components of a
DBMS, such as row containers, query processors, and transaction coordinators. By
defining a uniform set of interfaces to access data, OLE DB components not only
contribute to uniform data access among diverse information sources, but also help to
reduce the application's complexity by allowing developers to use only the DBMS
functionality they need. [13]
OLE DB components

OLE DB components can be broken down into three categories, figure 2.2-1 shows some possible combinations of the three different OLE DB components:

- **Data Providers**: Any software component that exposes an OLE DB interface. They own the data they expose to the outside world. Even though each provider differs in implementation details, all providers expose their data in a tabular format.

- **Data Consumers**: Any piece of system or application code that needs to access data from OLE DB providers. A number of programming tools and sophisticated applications, such as ActiveX Data Object (ADO), Microsoft SQL Server and Access fit into this category.

- **Service Components**: Objects that encapsulate a DBMS functionality, such as query processor and rowset container. [9]
OLE DB Core Components

OLE DB defines seven core COM objects. A consumer may use the objects in different combinations to achieve different purposes.

- **Enumerator object:** Enumerators search for available data sources and other enumerators. Consumers that are not customized for a particular provider and data source use enumerators to search for one to use. For example, SQL Server as an OLE DB consumer, needs to make use of enumerator objects to locate a particular OLE DB provider (or allow users to specify one) since it is a general-purpose application.

- **Data Source object:** A Data Source object is the first object returned by an enumerator and contains the necessary information and mechanism to connect to the underlying Data Source. Underlying Data Source can be a file directory, email inbox, or a relational database.

- **Session object:** A Session object controls the scope of transactions and is responsible for generating data and metadata rowsets. A single Data Source object can create multiple sessions. Session in turn creates Rowsets, Commands and Transactions.

- **Rowset object:** A Rowset is a container for data. It allows all providers to expose data in tabular form.

- **Command object:** A Command object is responsible for specifying, preparing, and executing text commands, such as queries. For instance, if the underlying data source is a relational database, Command object should be able to handle SQL queries to the underlying data source. Multiple Command objects can be created by one Session object. A Command object can also create Rowset objects to display result.

- **Transaction object:** A Transaction object carries out transactions like updates, inserts or deletes of data to the data sources.

- **Error object:** An Error object can be created by any interface on any OLE DB object. They contain additional information about an error, such as a description of the error. Custom error objects can also be defined with providers. [9]

Further discussion on OLE DB provider is carried out in chapter 3 and 4 – the design and implementation of the OLE DB provider for Cameleon.
Comparison of OLE DB and ODBC

An COIN Open Database Connectivity (ODBC) driver was previously developed for components of the Context Interchange System. The ODBC extension to the Context Interchange System allows users working in Windows to access the system through Microsoft Excel. However, the ODBC extension did not work with more sophisticated ODBC-compliant applications, such as Microsoft Access and SQL Server, due to the lack of support of schema information of the underlying data source [1]. Microsoft Access, SQL Server, or other DBMS engines require the presentation of metadata of the data source before submitting SQL queries. That kind of metadata describes the data contents of the DBMS such that users know what to expect prior to submitting a query. The name of catalogs, schemata and tables, as defined in SQL92 standard, are examples of metadata. In OLE DB development, those metadata are referred to as schema information. The Session object of OLE DB has an optional interface called Schema Rowset, which allows the OLE DB Provider to expose schema information. Therefore, developing an OLE DB front end to the Context Interchange System can possibly allow access to the system through more sophisticated DBMS engines such as SQL Server.

Other than the above, an OLE DB extension to the Context Interchange System may lead to improvement over the ODBC front end in the following areas:

- OLE DB supplies the tool to access both relational and non-relational data, while ODBC is mainly for relational data. Therefore, adapting to OLE DB technology is crucial to future development as more and more data sources are stored in non-database files.

- OLE DB provides better performance in terms of data access. Eventually, ODBC will be replaced by OLE DB’s more thorough approach to data.

- OLE DB tools and languages have full access to all ODBC drivers and ODBC-based data through the use of OLE DB provider for ODBC, but not vice versa. Therefore, OLE DB allows a wider variety of connectivity and it also provides backward capability with ODBC drivers. [13]
2.3 Extensible Markup Language (XML)

Extensible Markup Language, also known as XML, is a method for putting structure data in a text file. Development of XML started in 1996 and it is a W3C standard since February 1998. Like HTML, XML file is text file, which makes use of tags (words bracketed by ‘<’ and ‘>’) and attributes (in the form of name="abc"). However, unlike HTML, which has a specific predefined set of tag and attribute for data representation, XML leaves the definition of and interpretation of data to users. Tags in XML are only used for delimitation of pieces of data in a structure and meaningful ways defined by the composers of the documents. For example, in HTML, one uses tags to tell the browser to display the contents in bold or italic; in XML, one uses tags to describe the data and what the data represents, such as balance, names, and temperature. [16]

The basic data object for XML is called a XML document. An XML document is composed of nested elements and has a root called the document element. An XML element is made up of a start tag and an end tag, and with the data embedded in between. The name of the tags also describes the data value of the element. Figure 2.3-1 shows a sample of Cameleon output in a well-formed XML document.

```xml
<?xml version="1.0" ?>
<DOCUMENT>
  <ELEMENT>
    <DATEDDATE>11-15-1998</DATEDDATE>
    <FIRSTCOUPON>05-15-1999</FIRSTCOUPON>
    <PAYFREQUENCY>Semi-Annual</PAYFREQUENCY>
    <SETTLEMENTDATE>08-10-2000</SETTLEMENTDATE>
  </ELEMENT>
</DOCUMENT>
```

Figure 2.3-1. A sample Cameleon XML document output.

The industry found XML to be an extremely valuable tool in internet and intranet development because of XML’s ability to separate data from presentation. Data can be exchanged between different data sources in XML format easily through Hypertext Transfer Protocol (HTTP) due to XML’s text base nature. Furthermore, using XML as
the middle tier among different systems and data sources can increase data interoperability since the use of self-descriptive data allows more flexible and effective means to interpret and manipulate data. XML data can be displayed through HTML pages at the front end or it can be viewed from an internet browser directly.

Different tools have been developed for display and manipulation of XML data. Style sheets such as Extensible Stylesheet Language (XSL) and Cascading Style Sheets (CSS) are used to apply different display of XML documents through web browsers. A set of Application Programming Interface (API), called DOM, has been developed as a standard way to interact with elements of XML documents. The OLE DB Provider for Cameleon uses XML DOM to retrieve data. Refer later chapters for details.
Chapter 3 Design of the OLE DB Provider for Cameleon

3.1 Role of the Provider

As shown in Figure 3.1-1, the OLE DB Provider is the middleware between the OLE DB compliant applications and the data sources. Each OLE DB Provider is custom made to access one data source. All providers should handle OLE DB function calls of making connection to the data sources, fetching data from the data sources, executing commands and transactions (if supported), and returning data to the client applications in the appropriate data type. Moreover, OLE DB Providers also contain information of the properties of their underlying data sources. These properties are essential since they allow OLE DB consumers to communicate with the data sources in coherent manners. Different COM objects within an OLE DB Provider have their own set of properties.

![Figure 3.1-1: Role of OLE DB Providers](image)

For example, the Data Source object has properties that specify connection to the underlying data sources; the Command object has properties that specify the nature of the
command the underlying data source supports (SQL/non-SQL), or whether parameters are supported in command. Refer to appendix B for the complete list of properties that apply to the Cameleon Provider.

The OLE DB Provider for Cameleon serves as the middle tier between the Cameleon Web Wrapper Engine and client applications. The OLE DB Provider for Cameleon communicates Cameleon’s metadata, properties and behaviors to OLE DB-compliant consumers. Furthermore, when consumers submit SQL statements to the Cameleon Provider, the Provider translates the SQL statements into Cameleon input format (HTTP request), retrieves the XML result from Cameleon, parses out the results and finally organizes the result in the tabular format and returns it back to consumers. The OLE DB Provider for Cameleon is designed to treat all web sites that are accessible by Cameleon as a relational data base. From the Context Interchange framework perspective, the Cameleon Provider and the OLE DB compliant applications together can also be regarded as the new relational front end of the aggregation side of the Context Interchange System.

3.2 Structure of the Provider

In this section, the structure of the OLE DB Provider for Cameleon will be discussed. The general architecture of an OLE DB Provider is discussed. Specific designs that apply only to the Cameleon Provider are also mentioned.

In Section 2.2.3, the seven core components of OLE DB are defined. However, at a minimum, a simple OLE DB provider must only implement and support the object types and functionality listed in table 3.2-1. [10]

<table>
<thead>
<tr>
<th>Object</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSource</td>
<td>IDBCreateSession</td>
</tr>
<tr>
<td></td>
<td>IDBInitialize</td>
</tr>
<tr>
<td></td>
<td>IDBProperties</td>
</tr>
<tr>
<td></td>
<td>IPersist</td>
</tr>
<tr>
<td>Session</td>
<td>IGetDataSource</td>
</tr>
<tr>
<td></td>
<td>IOpenRowset</td>
</tr>
<tr>
<td></td>
<td>ISessionProperties</td>
</tr>
<tr>
<td>Rowset</td>
<td>IAccessor</td>
</tr>
<tr>
<td></td>
<td>IConvertType</td>
</tr>
<tr>
<td></td>
<td>IColumnsInfo</td>
</tr>
</tbody>
</table>
Table 3.2-1 Minimum requirement for an OLE DB Provider.

With the three basic core components, the OLE DB Provider is able to connect to the underlying data source, set up the environment for further operations, and display data in tabular rowsets to consumers. For the Cameleon Provider to be able to take in SQL queries and convey useful information to consumers, Command object and Error object are necessary. Moreover, some optional interfaces of the Data Source, Session and Rowset object are necessary for the Cameleon Provider to work with general-purpose consumers. Table 3.2-2 shows the list of additional interfaces which Cameleon Provider implements in order to meet the design goal. Refer to Appendix A for a complete list of interfaces and methods the Cameleon Provider implements.

<table>
<thead>
<tr>
<th>Object</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Source</td>
<td>IInternalConnection</td>
</tr>
<tr>
<td></td>
<td>ISupportErrorInfo</td>
</tr>
<tr>
<td>Session</td>
<td>IDBCreateCommand</td>
</tr>
<tr>
<td></td>
<td>IDbSchemaRowset</td>
</tr>
<tr>
<td></td>
<td>IObjectWithSiteSession</td>
</tr>
<tr>
<td></td>
<td>ISupportErrorInfo</td>
</tr>
<tr>
<td>Rowset</td>
<td>ISupportErrorInfo</td>
</tr>
<tr>
<td>Command</td>
<td>IAccessor</td>
</tr>
<tr>
<td></td>
<td>IColumnsInfo</td>
</tr>
<tr>
<td></td>
<td>ICommand</td>
</tr>
<tr>
<td></td>
<td>ICommandProperties</td>
</tr>
<tr>
<td></td>
<td>ICommandText</td>
</tr>
<tr>
<td></td>
<td>IConvertType</td>
</tr>
<tr>
<td></td>
<td>ISupportErrorInfo</td>
</tr>
<tr>
<td>Error</td>
<td>IErrorInfo</td>
</tr>
<tr>
<td></td>
<td>IErrorRecords</td>
</tr>
</tbody>
</table>

Table 3.2-2 Additional Interfaces implemented on Cameleon Provider
3.2.1 Properties

Before the discussion of each individual OLE DB component, it is necessary to give a brief overview on OLE DB Properties. OLE DB Properties are attributes of OLE DB objects. Each core component of an OLE DB Provider has several property groups. A property group is a set of logically related properties that could apply to an OLE DB object. For identification, each property is grouped under a property set, which has a Globally Unique Identification (GUID). Within a property set, each property has a property ID that identifies itself. The following table shows example property sets and property groups.[9]

<table>
<thead>
<tr>
<th>Property Set GUID</th>
<th>Property Group</th>
<th>Description</th>
<th>Example Property ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPROPSET_COLUMN</td>
<td>Column</td>
<td>Properties used to create columns.</td>
<td>DBPROP_COL_NONNULL</td>
</tr>
<tr>
<td>DBPROPSET_DBINIT</td>
<td>Initialization</td>
<td>Properties used to initialize the data source object.</td>
<td>DBPROP_INIT_LOCATION</td>
</tr>
</tbody>
</table>

Table 3.2-3 Example property sets and property groups.

Properties serve a crucial role in defining OLE DB Provider and data source behaviors and characteristics. For example, a rowset object has property that describes the maximum number of rows that can be open at one time. OLE DB Consumers set property values to request specific object behavior. Consumers get property values to determine the capabilities of an object. For example, Consumers use properties to determine whether the OLE DB Provider supports transaction. Refer to Appendix B for a complete list of OLE DB Properties of the Cameleon Provider.

3.2.2 Data Source Object

As mentioned in the previous section, the Data Source object is responsible for connection to the underlying data source of the Provider. Specifically, the Data Source object of Cameleon carries out three major tasks. Moreover, there are three property groups that belong to the Data Source object, they are the Initialization, Data Source and Data Source Information Property groups.
Initialization

The IDBInitialize interface is used to initialize and uninitialize Data Source objects. During initialization, OLE DB Providers usually try to connect to the underlying data source. If the underlying data source requires security information, such as user name and password of a DBMS, to gain access, Providers should have the necessary information to gain access, or otherwise Providers will prompt users for the necessary information. The underlying data source for the Cameleon Provider is the Cameleon Engine together with the web sites that it wraps. The Cameleon Provider only needs to interact with the Cameleon Engine, which currently does not require any security information to gain access. Therefore, the initialization routine for the Cameleon Provider is relatively simple since no connection is required at this stage. However, to facilitate later needs, the Cameleon Provider needs to gain information on the Cameleon Server path and Schema information path at the initialization stage. The decision of making users to input those two important connection paths, instead of embedding the paths in the code makes the use of the Provider more flexible when data source is being moved frequently.

When an OLE DB Provider is invoked, users have the option to fill in some parameters such as Data Source, Location, Login Name, Password, Provider String etc, depends on the Provider's setting. During initialization, the Cameleon Provider checks whether the users have filled in the Cameleon Server path in the Data Source field and the Schema information in the Provider String field. If the user fails to provide the above two pieces of information, the Cameleon Provider will once again prompt user for them. However, the actual usage of the two paths is in Command object and Session object.

Setting Data Source properties

The IDBProperties interface is used to set and get the values of properties on the data source object, and to get information about all properties supported by the provider. Before the data source object is initialized, the consumer can work only with properties in the Initialization property group. The Cameleon Provider sets certain properties in the Initialization property group such that it will prompt users for the Cameleon Server path
and the Schema information path. SetProperties and GetProperties are the methods that the Provider implements which allow consumers to set and get properties values in the Data Source respectively.

**Create Session**

The IDBCreateSession interface has a CreateSession method that allows Consumers to create a new session from the data source. In order to allow for multiple sessions, the Cameleon Provider sets the DBPROP_ACTIVESESSIONS property of the Data Source Information Property group to the maximum number of sessions that can exist at the same time.

**3.2.3 Session Object**

The Session Object is the factory for Rowset and Command objects. Usually, the primary function of a session is to define a transaction and how different units relate to each other in a system with parallel activities. However, the Cameleon Provider is intended to be a read-only OLE DB Provider without support for transactions. Therefore, the discussion on how the Session Object defines the scope of transaction is not relevant. Other than the creation of Rowset and Command objects, the Session Object of the Cameleon Provider is also responsible for fetching and presenting the Schema information to Consumers.

**Schema Rowset**

As mentioned in Section 2.2.3, Schema information is referred to the metadata that is presented to Consumers by the OLE DB Provider prior to execution of command, which describes the data organization of the underlying data source and the nature of the Provider. The IDBSchemaRowset interface of the Session object is responsible for presenting Schema information in tabular format to consumers. There are many Schema Rowsets that an OLE DB Provider can implement, out of which three Schema Rowsets are mandatory when the Provider implements IDBSchemaRowset. They are the Column Rowset, the Table Rowset and Provider Type Rowset. On top of the three mandatory
Schema Rowsets, the Cameleon Provider also implements the Catalog Rowset and the Schemata Rowset. Each Schema Rowset contains a set of restriction columns that describes an aspect of the data source or Provider. For example, the Table Rowset contains, for each table in the underlying data source, a row data that describes the characteristics of that table. Refer to the Appendix C for a complete listing of Cameleon Provider Schema Rowset data.

The implementation of the Catalog and Schema Rowsets allow the Cameleon Providers to interact with more sophisticated OLE DB consumers, such as Microsoft SQL Server. The ANSI SQL92 specification defines a catalog as one or more schemas; while a schema is a collection of database objects that are owned or have been created by a particular user. In Microsoft SQL Server and Microsoft Access terms, a catalog is a database, a schema is an owner. Catalog, Schema, Table and Column together form a four-part naming scheme that is useful when SQL Server interacts with OLE DB Provider. Refer to Chapter 5 for more discussion.

A mapping between Cameleon Registry information and Cameleon Provider Schema information can be easily recognized if one thinks of each of the wrapped web site as a single table in a regular DBMS, and the attributes wrapped in each web site as columns in a table. In relational terms, each wrapped web site, such as “Etrade” or “Bonds Online”, is a schema relation; while the attributes wrapped in each site are considered the tuples of the schema relation. The Catalog and Schema that contain all the tables are Cameleon and COIN respectively. With the above defined, the rest of the Schema information can be easily filled in according to the data values.
With the exception of the Provider Type Schema Rowset, which describes the base data types supported by the data provider, all the other Schema Rowset information is stored in XML format. The Cameleon system design stores schema information in Registry table in a relational DBMS. This design makes the Cameleon Provider dependent on the DBMS, making connection from the client machine to the DBMS whenever Schema information is needed. In addition, the client machine would need to have installed the DBMS interface software in addition to the Cameleon Provider. Using an XML document interface to specify Schema information not only eliminates the above two problems, but also enhances the Cameleon Provider in the following ways:

- **Eliminate extra interface:** The Cameleon Web Wrapping Engine presents results in XML format. Having the Schema information also in XML format provides a more coherent way for accessing data and makes reuse of code possible.

- **Simplify backend system requirement:** With the XML interface, Schema information can be stored in as simple as a XML document that is available through
HTTP. The XML interface can replace a DBMS system but still provide the data organization structure needed for Cameleon’s Schema information.

- **Decouple back-end schema storage technology from the Provider interface:** The Cameleon site may store the Schema information in an XML file or generate it on demand from its own database. The Provider is not tied to the specifics of the back-end Cameleon schema representation.

Cameleon’s Schema information is accessed as an XML document. The Cameleon Provider locates the Schema information XML document through HTTP using the path name obtained during initialization. The information on the XML document is organized in a hierarchical manner to resemble the organization in a DBMS. The Catalog node encloses Schema nodes; Schema nodes enclose Table nodes and Table nodes will have children of Column nodes. Figure 3.2-2 shows a fragment of the XML document that encodes the Schema information of one web site. Maintenance of Cameleon’s Schema information stored as XML document is easy when the amount of data is small. However, it will become difficult to edit the XML document manually as the amount of data grows. A generator program that automates the update of the XML document from the Cameleon Registry can be an easy remedy.

```xml
<CAMOLEDB>
  <CATALOG name="CAMELEON" description="Cameleon web wrapper system">
    <SCHEMA name="COIN" owner="dbo">
      <TABLE name="ETRADE" type="TABLE" description="E*Trade">
        <COLUMN ordinal="1" name="Date" datatype="DBTYPE_STR" maxlen="20" description="Whatever date" octetlength="1"
          columnflags="DBCOLUMNFLAGS_ISNULLABLE,
          DBCOLUMNFLAGS_MAYBENULL,
          DBCOLUMNFLAGS_MAYDEFER"
          column_hasdefault="Variant_false" is_nullable="Variant_true" />
      </TABLE>
    </SCHEMA>
  </CATALOG>
</CAMOLEDB>
```

Figure 3.2-2 Fragment of Cameleon Schema information.

**Creating Schema Rowsets**

The two methods of the IDBSchemaRowset interface: GetSchemas and GetRowset allow consumers to retrieve Schema information from the OLE DB Provider. GetSchemas returns all the Schema Rowsets that are implemented by the Provider; while GetRowset returns one Schema Rowset at a time. Schema Rowsets are identified by Globally Unique Identifications (GUIDs). DBSCHEMA_TABLES, DBSCHEMA_COLUMNS,
DBSCHEMA_PROVIDER_TYPES, DBSCHEMA_CATALOG AND
DBSCHEMA_SCHEMATA are the GUIDs of the Schema Rowsets that Cameleon
Provider implements.

When Consumers call GetSchemas or GetRowset, the Cameleon Provider
retrieves and parses the Schema information XML document and fills the values of all the
restriction columns of each Schema Rowset. In the case of the Provider Type Schema
Rowset, XML retrieval is not needed since Cameleon Provider code will fill in the data
type values when the Provider Type Schema Rowset is being called. The discussion on
how the Cameleon Provider uses XML DOM technology to retrieve and parse Cameleon
Schema information is carried out in Section 4. The Session object then creates Rowset
object to display Schema information in tabular format.

Creating Rowsets and Commands

The IOpenRowset interface enables Consumers to open and work directly with data from
the data source by using IOpenRowset::OpenRowset, which generates a rowset of all
rows in the table or index. Since the Cameleon Provider requires the submission of
Command before generating result data, IOpenRowset interface is only used for exposing
Schema Rowset information. The IDBCreateCommand interface supports the creation of
Command objects. Consumers have to first obtain a new Command object from the
Session object, and through the execution of command to obtain result rowsets.

Getting Pointer to Data Source object

When there is a need to access properties or other interfaces of the Data Source object,
the IDataSource interface supports the method of GetDataSource, which returns an
interface pointer on the Data Source object that created the Session. The notion of
getting from one object to the other within an OLE DB Provider is particularly important
as one tries to access data of the parent objects. For example, the Cameleon Provider
needs a way to propagate values of the Initialization Properties set by users from the Data
Source object to the Session object for locating Schema information, and to the
Command object for locating the Cameleon Web Wrapper Server path. The Rowset and
Command objects also have methods to get a hold of their parent Session objects.
Session Properties

Similar to the IDBProperties interface of the Data Source object, the ISessionProperties interface also has GetProperties and SetProperties methods to allow Consumers to manipulate Session Properties. Even though there is only one OLE DB-defined Property in the Session Property group, and it is for the definition of transaction isolation level, the Cameleon Provider still implements this mandatory interface.

3.2.4 Command Object

The Command object of the Cameleon Provider is another object whose design is specifically tailored to Cameleon’s need. The Command object serves the functionality of a query processor, including query formulation, preparation, and execution when we think of the Provider together with the underlying data source as a database engine. There are four stages involve in carrying out a command by an OLE DB Provider. This section discusses each of the four stages, and the specific tasks that the Cameleon Provider carries out to meet its design goal.

Initial

A Command object is in initial stage when the Consumers have just created a Command object using the IDBCreateCommand interface of the Session object. At this point, there is no command text for execution.

Unprepared

A Command object is in Unprepared stage when the Consumers have called SetCommandText of the ICommandText interface and submitted a text command. The Cameleon Provider allows users to input SQL query as command text, and it stores the SQL query text for later process. Moreover, the Cameleon Provider also keeps a flag to indicate whether the command text has ever been set and does not allow Consumers to proceed without setting the command text. This extra step of precaution is due to the fact
that submitting a SQL command is the only way for Consumers to extract result data from Cameleon.

**Prepared**

A Command object is in Prepared stage when the Consumers have called Prepare method of the ICommandPrepare interface. The ICommandPrepare interface encapsulates the process of command optimization, or the generation of a command execution plan. The Cameleon Provider does not yet implement this optional Prepare interface, since it is primarily for the enhancement of the efficiency, which is not the focus of our design at this stage.

**Executed**

A Command object is in Executed stage after the command text is being executed. Consumers call the Execute method of the ICommand interface to perform command execution. During execution, the Cameleon Provider has to perform the following three tasks:

- Converts the standard SQL query stored as command text to the form that Cameleon Engine accepts as input, i.e. HTTP request format.
- Sends the converted command to Cameleon Engine, retrieves and parses the output XML document.
- Creates Rowset object to display result data in tabular format.

Furthermore, the Execute method of the Cameleon Provider is used not only in the actual execution of command, but also in determining result column information after the Consumers have set the command text. Multiple uses of the Execute method is due to the fact that the Cameleon Provider does not know the column information for the resulting rowset until the command is actually executed. Refer to Chapter 4 for more in depth discussion on dynamic binding of column information.

Consumers can also call the Cancel method or the GetDBSession method of the ICommand interface to cancel execution of command or to get an interface pointer to its parent Session object respectively.
3.2.5 Rowset Object

Rowsets are the central objects that enable OLE DB Provider to expose and manipulate data in tabular form. A Rowset object is a set of rows in which each row has columns of data. The Cameleon Provider presents data and metadata to Consumers in the form of Rowsets. The Command object of Cameleon Provider (query processor) also presents query results in the form of Rowsets. Since the Rowset object plays such a crucial role as data container for the Cameleon Provider, some of its interfaces are also implemented in other object to allow creation of Rowset. For example, the Command object implements the IAccessor, IColumnsInfo and IConvertType that are also interfaces of the Rowset object.

Creating Rowsets

There are three ways an OLE DB Provider can create Rowsets. The Cameleon Provider makes use of two mechanisms. First, Consumers can call IOpenRowset::OpenRowset from the Session object to create a rowset directly without executing any command. Since the Cameleon Provider only displays result when a valid SQL query is submitted, the Cameleon Provider does not support the first mechanism. Second, Consumers call ICommand::Execute from the Command object and it returns the resulting rowsets. As mentioned in Section 3.2.4, this mechanism is the only way for Cameleon Provider to display result data. Third, Consumers can invoke a method that returns a rowset or schema rowset. The Cameleon Provider allows Consumers to call IDBSchemaRowset::GetRowset (or GetSchemas) to get rowsets of metadata information.

Other functionality

The Rowset object, being a general-purpose container for data, has different interfaces that work on definition, binding and conversion of data types for display. It is also equipped with functionality for Consumers to navigate through different rows and fetching column data.
IAccessor

The IAccessor interface is an important interface that is responsible for data transferal between OLE DB Consumers and Providers. An accessor is a collection of information that describes how data is stored in the Consumer’s buffer. One can also think of an accessor as a collection of bindings, and each binding associates a single column to the Consumer’s buffer. When transferring data, Consumer first creates an accessor to describe how it wants the data to be laid out in the Consumer’s data buffer. The accessor contains for each column a binding that describes the column’s data type, ordinal value and destination in the Consumer’s buffer. Then the Consumer will pass to the Provider the handle to the accessor when it calls a method that results in data transferal. The IAccessor interface provides methods to create and maintain accessor.

IRowset

The IRowset interface is the base of rowset interface. It provides methods for fetching rows sequentially, getting the data from those rows and managing rows. To fetch rows from the data source, the Consumer calls the method GetNextRows. The GetNextRows method does not actually put data in the Consumer’s data buffer, it merely returns a handle to the rows. The Consumer needs to call GetData to perform data transfer. GetData converts the data to the appropriate type, and puts the columns as specified in the accessor bindings to the right destinations in the Consumer’s data buffer. The IConvertType interface gives information about the availability of type conversions during GetData.

IRowsetInfo

Consumers can get the Rowset object’s capabilities and properties through the IRowsetInfo interface. The interface provides the GetProperties method for Consumers to retrieve Rowset Property values. The Rowset Property Group is the biggest group of OLE DB-defined Property among the rest of the Provider components. Some
sophisticated OLE DB Provider that allows Rowset updates and other transactions supports most of the Properties in the Rowset Group. However, only a few of the Properties apply to the Cameleon Provider since rowsets are only used for displaying data in the Cameleon Provider.

3.2.6 Error

Return Codes

Each OLE DB method returns a return code, which indicates the overall success or failure of the method. These return codes are of type HRESULT. There are two classes of return codes: success and warning codes, and error codes. Success and warning codes begin with S_ or DB_S_ and indicate that the method completed. For example, the return code S_OK is an “everything went well” signal. When the method fails, error code is returned which begins with E_ or DB_E_, such as E_INVALIDARG. Even though error codes appropriately signify errors, if an OLE DB Provider demands more detailed error information, it has to implement the Error object.

3.3 Application Flow

After in depth discussion of each component of the Cameleon Provider, this section pieces them altogether to show how different components interact with each other. This section also presents the application flow of the program through the examination of a complete interaction between the Cameleon Provider and an OLE DB Consumer in two figures. Figure 3.3-1 illustrates the flow of the Cameleon Provider. Figure 3.3-2 shows a more detailed chart with specific function calls to illustrate how an OLE DB Consumer maneuvers around different interfaces.
Figure 3.3-1 OLE DB Components of Cameleon Provider.
CoCreateInstance(...) – create a Data Source object

Data Source

IDBProperties::SetProperties
Set Data Source Initialization Properties

IDBInitialize::Initialize
Initialize Data Source object.
Checks for Server and Schema Info Path

OK

No Path

Prompt User

IDBProperties::SetProperties
Set the rest of the Properties in Data Source Group

IDBCreateSession::CreateSession() – create a Session object

Session Object

ISessionProperties::SetProperties
Set Session Properties

Consumers do not need Schema Info

IDBSchemaRowset::GetSchema
Get Schema Information

Rowset Object

IAccessor::CreateAccessor
Create column bindings

ICommand::Initialize

IDBCreateCommand::CreateCommand()

ICommand::Execute
Check is command text is set. Converts SQL command, retrieves and parses XML result.
Since there are numerous optional interfaces that an OLE DB Provider may or may not implement, it is important to understand how OLE DB Consumers know what are available for them to call, and how to invoke a particular interface. Some OLE DB-defined Properties may convey some interface information to Consumers, however, we need the IUnknown interface for the query and acquiring handle capabilities.

3.3.1 IUnknown Interface

The IUnknown interface is a fundamental base interface in COM. It provides three important methods: QueryInterface to allow consumers to find out what interfaces an object implements; AddRef and Release to keep a reference count of the object. Every COM object implements this interface; and every COM interface support the above three functions. Consumers use QueryInterface to get a hold of an interface or object they need. Figure 3.3-3 shows a code fragment that illustrates the use of QueryInterface. [3]

```c
//This is a fragment of the Initialize method of the IDBInitialize interface within
//the Data Source object.
IDBProperties * pIDBProperties = NULL; //pointer to IDBProperties interface
HRESULT hr; //Return code
hr = pIUnknown -> QueryInterface (IID_IDBProperties,
   (void **) &pIDBProperties))

pIDBProperties -> Release(); //free the pointer
```

Figure 3.3-3 Use of QueryInterface.
The figure shows a fragment of code from the Initialize method of the IDBInitialize interface when it tries to get a hold of the IDBProperties interface. Every interface has access to the pointer of the underlying IUnknown interface (pIUnknown) of the parent object (Data Source object in this case), and using that one can call QueryInterface to obtain any of the available interface of the parent object by passing in the appropriate GUID (IID_IDBProperties). QueryInterface does not allow cross-object interface query, therefore, in order to obtain an interface of another object, it will first have to obtain a handle to the object that implements the desired interface. Finally, the interface pointer has to be released at the end of usage.
Chapter 4 Implementation

Chapter 3 has presented the overall design of the OLE DB Provider for Cameleon, and also described how the Cameleon Provider works. This chapter focuses on the actual implementation of the Cameleon Provider. This chapter presents a more low-level description on the tools, data structures, algorithms, and other techniques involved in building the Cameleon Provider.

4.1 Visual C++ 6.0 ATL Provider Wizard

There are more than 70 COM interfaces in the OLE DB version 2.5 specification, and each COM interface has an average of two to three methods that need to be implemented. The OLE DB Provider for Cameleon implements more than 30 COM interfaces. However, as discussed in Chapter 3, not all of the interfaces require unique implementation for the Cameleon Provider. In fact, most of the interfaces are rather standard on all OLE DB Providers, such as IDBProperties, IAccessor; or only require minor changes such that it can fit other Providers; not to mention the large amount of standard base code each COM object requires. Besides, implementing so many COM interfaces on one application can be time-consuming and tedious. Microsoft has addressed this problem by incorporating in Visual C++ 6.0 the OLE DB Provider Templates. The Provider Templates, an extension to the Visual C++ Active Template Library (ATL), provide implementation for all of the required interfaces of an OLE DB Provider. Moreover, the OLE DB Provider Wizard within Microsoft Visual C++ 6.0 can generate a ready to run OLE DB Provider, one only needs to modify parts of the code and add additional interfaces to turn it into another OLE DB Provider. This feature not only saves time and effort in developing the framework of the Provider, but having a working Provider as a basis also helps minimize potential problems that would have encountered if one develops from scratch. Lastly, using Wizard generated code can probably help the Cameleon Provider conform to the OLE DB standard defined by Microsoft. Based on the above advantages, the Cameleon Provider is decided to be implemented using the OLE DB Provider Wizard generated Provider as the basis.
Despite all the advantages promised by the OLE DB Provider Wizard, using the OLE DB Provider Wizard is not as easy a task as imagined. The development of OLE DB and COM interface is not a simple concept to master. Even though the OLE DB Provider Wizard may have implemented some of the crucial parts of an OLE DB Provider, to twist the Wizard generated code requires thorough understanding of each interface. Moreover, the Active Template Library has special programming methodology, macros and data types, which require some background knowledge. Worse of all, the amount of documentation available on the Template library that applies to the development of OLE DB Provider is minimal. Often times, one has to dig into the ATL source codes to find out how they work, and the use of template also makes the code hard to understand. My opinion is that the use of the OLE DB Provider Wizard definitely saves a lot of coding, but whether it saves developers’ time is largely dependent on the developers’ familiarity with Visual C++ ATL and COM programming. In this chapter, some of the special techniques used in the development of the Cameleon Provider using the Provider Wizard will be discussed. Before diving into implementation details, some background information will be given to help understand later concepts.

4.1.1 C++ Templates

Each COM class has to meet a list of common requirements in order to work. Some of the requirements include implementing the IUnknown base interface, the IDispatch interface and the self-registration code, just to name a few. This kind of infrastructure code stays the same for every single COM class, and it is not efficient for each COM class to implement those code. One can solve the above problem using Inheritance, of which subclasses can inherit all the functionalities of their base classes in object-oriented programming methodology. However, by doing inheritance, one still needs to implement a base class for each data type or object. The C++ template is an effective solution to the above problem. It allows the reuse of C++ code in an elegant way, and also uphold the principle of object-oriented programming at the same time. In short, C++ Templates allow polymorphism. Instead of having a class that only works on one particular kind of object, with templates, one can create classes that work on various kinds of objects. How to program C++ template class will not be discussed in this document, since it is
available from any of the programming reference book on C++. Instead, it is important to understand how C++ Templates help simplify development of COM objects.

COM development can make use of C++ Templates in the following ways:

- Parameterize the implementation of the IUnknown base interface. Every COM class must implement IUnknown interface, which is responsible for keeping reference of the COM object. However, the way it keeps reference may change based on the thread mode being used. Passing in different thread models as parameters can help manage the implementation of the IUnknown interface.

- Create a single function for generation of all kinds of objects. Whenever the consumers call CoCreateInstance with an object identifier, it will return the right class.

- Allow share use of common interfaces. As shown in the previous chapter, some interfaces such as the IAccessor and IConvertType are used in different objects. With C++ Templates, the caller object can pass itself as a parameter such that different objects can share common interfaces.

- Create COM specific smart pointers to hold different interface members. The CComPtr and the CComQIPtr make programming more convenient since they are able to hold different COM interfaces and allow users to treat interface pointers as objects.

- Allow share use of data structures. Common data structures are used to hold Properties and Property Info for different COM object. Using Templates helps to identify which Property groups are being called. [3]

4.1.2 Active Template Library (ATL)

The Active Template Library (ATL) was designed from the ground up to make developing COM objects in Visual C++ easy and flexible. The Active Template Library is a set of template-based C++ classes. Template libraries such as ATL differ from traditional C++ class libraries in that they are typically supplied only as source code and they are not necessary hierarchical in nature. Rather than deriving from a class to get the functionality one desires, one instantiates a class from a template. Frequent use of Map structures and Macros are also characteristics of the ATL.
Special Data Types in ATL

Since COM is a language-independent model, it has its own set of data types to facilitate object handling and data transfer from one component to the others. To simplify COM programming, the Active Template Library provides support for dealing with those COM specific details by implementing wrapper classes or helper classes on top of the COM primitive data types. This section discusses several of the ATL specific data types, which are frequently used in the development of the Cameleon OLE DB Provider.

Smart Pointers: CComPtr and CComQIPtr

The CComPtr and the CComQIPtr smart pointers are template C++ wrapper classes that manage the lifetime of an COM interface pointer. As mentioned in section 3.3, every COM interface implements the functionality of IUnknown, which keeps track of the reference count of the object. In addition, as shown in the code fragment of Figure 3.3-4, every time the client accesses a COM interface, it needs to dynamically allocate a pointer, call QueryInterface to obtain the pointer, and then Release the pointer at the end of it.

The idea behind CComPtr and CComQIPtr is to create template classes that would encapsulate the calls to AddRef and Release such that the class constructor and destructor call them automatically. With the use of smart pointers, the code snippet in Figure 3.3-4 can be transformed to the one in Figure 4.1-1.

```c++
//This is a fragment of the Initialize method of the IDBInitialize interface within
//the Data Source object.
CComQIPtr<IDBProperties> pIDBProperties; //pointer to IDBProperties interface
pIDBProperties->GetProperties(...); //use the IDBProperties Interface

//Pointer will be automatically Release when out of scope.
}
```

**Figure 4.1-1 Rewrite of Figure 3.3-4 using CComQIPtr.**

The CComQIPtr handles QueryInterface, AddRef and Release without developers worrying about calling the methods. The CComPtr differs from CComQIPtr in that CComPtr does not support QueryInterface functionality, therefore, it is a subset of CComQIPtr.

Using CComQIPtr makes code cleaner and allows one to use interface pointer as if it is a class pointer. Furthermore, both smart pointer classes offer helper methods to
compare, attach and detach pointers etc, which provide extra convenience for COM development.

**BSTR**

In COM, text characters are represented using the 16-bit OLECHAR, also known as WCHAR, data type. This design comes with the intention that using 2-byte character allows COM to support all kind of codes, including Unicode. Each OLECHAR contains a null terminating value and each character occupies two bytes of space instead of one.

The use of null terminating OLECHAR still is not general enough for string passing between components. Instead, the BSTR type, which is the standard on Visual Basic and Java, is used. BSTR shares the same 2-byte character and null terminating value characteristics with OLECHAR, and BSTR also has extra four bytes at the beginning of each string to indicate the length of the text string. Figure 4.1-2 shows the format of a BSTR.

![Figure 4.1-2 A BSTR encoding the character string “a bstr”.

The extra four bytes length prefix of BSTR poses inconvenience to C and C++ programmer since regular memory management functions, such as malloc and delete, cannot be used on BSTR. As a result, the ATL provides the CComBSTR class that encapsulates the functionality of BSTR and provides other frequently used string manipulation methods to the BSTR data type. Every CComBSTR object has a BSTR data member. The CComBSTR class also allows convenient conversion from LPCSTR (Long character pointer to regular C++ string data) to BSTR. There are various overloaded constructors of CComBSTR, which can be used to allocate a new BSTR given an LPCSTR, LPCOLESTR (Long character pointer to OLECHAR string data), or
another CComBSTR. Refer to Visual C++ Reference of a complete listings of CComBSTR methods.

Even though the CComBSTR class has made creation of BSTR from regular single byte character string easy, going the other way requires some more indirect methods. CComBSTR does not support the extraction of LPCSTR pointer from the underlying BSTR object. Therefore, if one needs to convert a BSTR value to regular C++ string for display purpose or to work with non-ATL modules, operators from the BSTR have to be used. One effective way in converting BSTR back to regular C++ single byte character string was discovered during early stage of development of the Cameleon OLE DB Provider. The following fragment of code illustrates the use of it.

```
VARIANT vResult;
hr=pNode->get_nodeValue(&vResult);
BSTR bResult=NULL;
CString result;
if(vResult.vt==VT_BSTR)
{
    bResult=V_BSTR(&vResult);
    result=CString((LPCWSTR)(wchar_t*)bResult);
}
```

Figure 4.1-3 Code fragment to shows conversion from VARIANT to BSTR; and from BSTR to CString.

The line of code pointed by the arrow in figure 4.1-3 shows the conversion from BSTR data type to single byte regular character string type, which is then feed into the CString constructor to create a CString object. First, the wchar_t* operator extract the OLECHAR pointer from the BSTR data type, then the pointer is cast into a Long Pointer to Constant Wide character String (LPCWSTR) that is accepted by one of the overloaded constructor of CString.

It is extremely important to understand BSTR when developing OLE DB Providers since it is the most frequently used data type. Furthermore, it is also important to understand the underlying buffer structure of BSTR when calculating buffer offset in the implementation of GetColumnInfo. Refer to section 4.4 for further details on GetColumnInfo.
VARIANT

Similar to the BSTR class, VARIANT is another special data type that is not quite C++-friendly. VARIANT is a data structure that can hold most of the data types including BSTR, integer, char, etc. It has a special type variable to indicate the data member type. By using VARIANT, developers can perform operations without worrying about the actual data type it represents. When there is a need for conversion, the VariantChangeType API will be called automatically to perform the conversion. The ATL also provides a wrapper class for VARIANT, called the CComVariant, to ease the use of VARIANT type in C++. The CComVariant class actually inherits from VARIANT data type, but provides extra functionality to better manage VARIANT.[3]

One of the important usages of VARIANT in OLE DB programming is in data conversion between Provider data type and Consumer data type. When an OLE DB Consumer retrieves data from the Provider, VARIANT serves as the perfect data container since the Consumer may expect a different data type than what the Provider returns. Similarly, most of the interface member functions, which perform data retrieval from one source to the other, would use VARIANT as a temporary storage.

Figure 4.1-3 also shows an example of how to make use of the VariantChangeType API to convert VARIANT data type to BSTR data type. First, the vt field of the VARIANT is checked to confirm the type, and then the V_BSTR method is called to do the data conversion.

4.1.3 ATL Wizard

With the help of the Visual C++ 6.0 ATL COM AppWizard, one can generate a COM server as the basis to hold COM objects. Using the ATL Object Wizard, one can insert COM objects to the server. To create an OLE DB Provider, one simply chooses to insert an Data Access object using the ATL Object Wizard, and then picks OLE DB Provider. The ATL Object Wizard then inserts implementation of the required interfaces of an OLE DB Provider, either in DLL or in EXE form, to the COM server created by the ATL.
COM AppWizard. At this point, a functional OLE DB Provider has been created. Refer to Appendix D for further details on how to use ATL Wizards to generate an OLE DB Provider.

4.2 Implementation of Cameleon Provider using the ATL Provider Wizard

As mentioned before, the ATL Wizard benefits developers a great deal by providing implementation for a lot of set up codes and required interfaces. However, due to the complicated template code of the ATL and all the tricky macros used in the codes, deep understanding of OLE DB Programming is required to turn the Wizard generated code to a Provider that meets the design goal. This section discusses some of the implementation details of the Cameleon Provider using the ATL Wizard generated OLE DB Provider as the framework. First, the structure and organization of the Wizard generated OLE DB Provider will be examined, with a focus on the general modules that apply to all OLE DB Providers. Then, discussion on Cameleon Provider specific implementation and modification will be carried out.

4.2.1 Wizard generated OLE DB Provider

The Provider Wizard generates a ready-to-run OLE DB Provider with the Data Source, Session, Command, and Rowset object in Dynamic Link Library form. All the mandatory interfaces of the above four objects are implemented, with the addition of IDBSchemaRowset interface of Session object. The Provider provides the functionality to display file directory information in tabular format. Users have the option to specify the directory they intend to view; otherwise the current directory is used as default. The Wizard generated Provider is a read-only Provider that does not support transaction nor rowset updates.[11]

General Organization

Within the Visual C++ workspace of the Wizard generated Provider, each core component has its own header file for definition and source file for actual source code
with the exception of the Command object, which header file is merged with that of Rowset’s. There are also source files that contain the COM server side implementation. Furthermore, there is a resource file that contains the crucial Registry information, and a string table that maps all the OLE DB-defined Properties that are used in the Provider to a numeric value and the caption.

There are a total of eight classes defined in the Wizard generated Provider. Each of the four core component implements its own class. Within the Session object, there are three Schema Rowset classes implementing the three required Rowsets: Table Rowset, Column Rowset and Provider Type Rowset. The Wizard generated Provider also implements a class calls WindowsFile class, which name is base on the fact that the Wizard generated Provider displays file information on Microsoft Windows operating system. This class serves as a data structure that holds a row of the result data.

The organization of the Wizard generated Provider is relatively simple and the amount of code generated is of minimal. This is because most of the required interfaces are included in ATL, and the Provider only needs to inherit those interfaces.

**The COM Server Module**

One of the most important modules of the Wizard generated code includes the necessary support structure that ATL provides for serving COM objects to clients. This module provides different functions to manage COM registration and the lifetime of COM server itself. Together, they are known as the COM server module. The `CComModule` class within ATL is the basis for server implementation. The Wizard generated Provider has an instance of the `CComModule` in the Provider’s main source code file.

**Self-Registration**

As mentioned in section 2.2.2, COM objects have to be registered such that COM clients can locate and invoke them. Performing self-registration is therefore the most fundamental task that an OLE DB Provider should implement. The `CComModule` class provides several methods that handle the self-registration process. The following are two methods implemented by the ATL Wizard Provider, which make use of the `CComModule`’s registration capabilities:
- **DllRegisterServer**: This function calls `CComModule::RegisterServer` to add the appropriate keys and values to the Windows Registry such that COM clients know about the availability of the Provider and how to launch it.

- **DllUnregisterServer**: This function calls `CComModule::UnRegisterServer` to remove Registry entries when objects are no longer available.

There is also a resource script that is invoked by calling the `RegisterServer` or `UnRegisterServer` methods. The resource script is a text-based representation of the entries to make for an object in the Registry.

### DLL Activation and Object Creation

The COM Server module generated by ATL Wizard also provides components that are responsible for the activation of the DLL server and the creation of class objects. When COM client requests an object that resides in DLL, COM loads the DLL into the client’s process first, then calls the following function:

- **DllMain**: This is the entry point for DLL type COM servers. `Init` is called when COM client process is attached; while `Term` is called when COM client process is detached.

- **DllGetClassObject**: This method is called once the DLL is initialized and `DllMain` has returned. It is used to retrieved class object by supplying the appropriate ID. It calls `CComModule::GetClassObject` to get the class object. If it is the first time a class object is requested, `CComClassFactory` interface will be used to create that object.

`CComClassFactory` is the ATL version of the `IClassFactory` interface, which is responsible for the creation of COM object. ATL Wizard generated Provider also holds an object map in its COM server module to keep track of all the available COM object and their corresponding IDs. Functions of `CComClassFactory` have to refer to the object map when creating objects.

### Server Lifetime Management

The last part of the COM server module contains code to manage the lifetime of the DLL server itself. When client finishes using the COM objects a server contains, it will have
to get rid of the server object from its memory. The DllCanUnloadNow method in the server module checks whether the DLL can be freed according to the reference count on the DLL. Whenever an object in the DLL server is being used, the reference count of the DLL is incremented by one. When the object is destroyed, the reference count will decrease by one. The DLL can be freed as long as its reference count is zero.

4.2.2 Implementation of Cameleon Provider components using the ATL Wizard

This section will examine some of the important code snippets from the Wizard generated Provider, and discuss how to turn the Wizard generated Provider to one that meets the design goal of the Cameleon Provider. First, common features to all components will be discussed. Then, modifications to individual method will be carried out in detail.

Component inheritance model

Each object of the Wizard generated Provider implements its interfaces by means of multiple inheritance. Figure 4.2-1 shows the inheritance chain of the Data Source object of the Cameleon Provider.

```cpp
// CDataSource
class ATL_NO_VTABLE CCamProvSource :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCamProvSource, &CLSID_CamProv>,
public IDBCreateSessionImpl<CCamProvSource, CCamProvSession>,
public IDBInitializeImpl<CCamProvSource>,
public IDBPropertiesImpl<CCamProvSource>,
public IPersistImpl<CCamProvSource>,
public IInternalConnectionImpl<CCamProvSource>,
public ISupportErrorInfoImpl<CCamProvSource>
```

Figure 4.2-1 Inheritance chain of Data Source object of the Cameleon Provider.

Every component inherits three different kinds of template classes. The CComObjectRootEx provides all the implementation of the base IUnknown interface. The CComCoClass is responsible for error handling. Lastly, the rest of the “Impl” classes are ATL classes that provide the template implementation of Data Source object’s interfaces. By inheriting the “Impl” class, the Provider chooses to expose that particular interface and the methods included, and vice versa. The object can also implement
interfaces that are not included in the ATL library. For example, some of the more sophisticated interfaces that support transactions are not implemented in ATL. Developers have to create their own implementation class such that the object can inherit those functionality.

Object Interface Map

Inheriting the interface "Impl" classes only equips the object with the functionality of an interface. Consumers need a way to find out what interfaces the object exposes. As mentioned in Section 3.3, Consumers use the QueryInterface method from the IUnknown interface to solve that problem. On the other hand, Provider components also need a way to keep track of their own interfaces. The ATL Wizard generated Provider keeps an Interface Map in each object. Whenever Consumers use the QueryInterface function of the CComObjectRootEx object, the object's Interface Map is searched to yield all the interfaces. Figure 4.2-2 shows the Interface Map of Cameleon Provider's Data Source object in correspondence to its inheritance chain. It is necessary to include all the implemented interfaces in the Interface Map such that Consumers can access them. The BEGIN_COM_MAP macro is used to manage the Interface Map.

BEGIN_COM_MAP(CCamProvSource)
  COM_INTERFACE_ENTRY(IDBCreateSession)
  COM_INTERFACE_ENTRY(IDBInitialize)
  COM_INTERFACE_ENTRY(IDBProperties)
  COM_INTERFACE_ENTRY(IPersist)
  COM_INTERFACE_ENTRY(IInternalConnection)
  COM_INTERFACE_ENTRY(IISupportErrorInfo)
END_COM_MAP()

Figure 4.2-2 Interface Map of the Data Source object of Cameleon Provider.

Object Property Map

ATL also provides a unique way to manage OLE DB-defined Properties within each object through the use of macros. Each object in the Provider contains a Property Map similar to the Interface Map. Properties are separated into their own Property Group, and the PROPERTY_SET macros specify the entries in the Property Map. Figure 4.2-3 shows a fragment of the Property Map of the Data Source object of the Cameleon Provider.
BEGIN_PROPSET_MAP(CCamProvSource)
BEGIN_PROPERTY_SET(DBPROPSET_DATASOURCEINFO)
  PROPERTY_INFO_ENTRY_VALUE(ACTIVESESSIONS, 0)
  PROPERTY_INFO_ENTRY_VALUE(DATASOURCEREADONLY, VARIANT_TRUE)
  PROPERTY_INFO_ENTRY(BYREFACCESSORS)
  PROPERTY_INFO_ENTRY_VALUE(PROVIDERNAME, "OLEDCamProv.dll")
  PROPERTY_INFO_ENTRY(PROVIDEROLEDBVER)
  PROPERTY_INFO_ENTRY(DSOTHREADMODEL)
END_PROPERTY_SET(DBPROPSET_DATASOURCEINFO)
BEGIN_PROPERTY_SET(DBPROPSET_DBINIT)
  PROPERTY_INFO_ENTRY(AUTHPASSWORD)
  PROPERTY_INFO_ENTRY(INITDATASOURCE)
  PROPERTY_INFO_ENTRY(INITHWND)
  PROPERTY_INFO_ENTRY(INITLOCATION)
  PROPERTY_INFO_ENTRY_VALUE(INITPROMPT, DBPROMPTPROMPT)
  PROPERTY_INFO_ENTRY(INITPROVIDERSTRING)
  PROPERTY_INFO_ENTRY(INITTIMEOUT)
END_PROPERTY_SET(DBPROPSET_DBINIT)
END_PROPSET_MAP()

Figure 4.2-3 Property Map of Data Source object of the Cameleon Provider.

The BEGIN_PROPERTY_SET macro separates Properties into their own Property group. The above code snippet shows that the Cameleon Provider supports Properties from two Property group: the Data Source Information group and the Initialization Group. Specifying a property using the PROPERTY_INFO_ENTRY indicates the Provider supports that particular property, and the property value is being set to default. If the Provider supports a particular property with a non-default value, one can use the PROPERTY_INFO_VALUE macro to specify the Property together with a new value. The default value is automatically determined within the ATL macro depends on the Property's data type. Note that the Property ID used in the macro are different from the Property ID in the OLE DB version 2.5 specification. They are short of the DBPROP_ prefix such that the ATL macro can append either the DBPROP_ or the IDS_DBPROP_ prefix internally as need arises (the IDS_DBPROP_ prefix is the ID type used in the mapping specified in the Provider's resource file).

Implementation of individual object

Next, we will take a closer look at each of the object within the Cameleon Provider.

Data Source object

All of the interfaces in Data Source that Cameleon Provider requires are implemented in the ATL, therefore, the Data Source class of the Cameleon Provider only needs to inherit
the correct "impl" classes and fills in the Interface and Property Maps appropriately.
However, the Cameleon Provider requires the fetching of the Cameleon Server path and
the Schema Information path during initialization, which the Wizard generated Provider
does not support. In order to produce the desired behavior, the
IDBInitializeImpl::Initialize method has to be re-implemented. Also, the correct use of
Properties in the Initialization group is required such that users are prompted for the
correct information during Data Source initialization.

One can override the ATL interface implementation by re-implementing the
function that requires change within the object class of the Provider. The Cameleon
Provider implements its own Initialize method inside the Data Source object such that the
IDBInitializeImpl::Initialize method is not called. However, the re-implementation of a
function can be tricky since one has to make sure the re-implementation works with other
ATL template interfaces and internal data structure perfectly. This usually requires the
correct use of template arguments.

The Cameleon Provider requires change in the Initialize method such that during
Data Source initialization, the Provider checks whether users have submitted the
Cameleon Server path in the Data Source field and Schema Information path in the
Provider String field before proceeds. Users' input when Provider is invoked are stored
as property values of the Initialization Property Group: DBPROP_INIT_DATASOURCE
Property stores the value of the Data Source field; while
DBPROP_INIT_PROVIDERSTRING Property stores the value of the Provider String
field. In order to check the value of the above two properties within Initialize, a
CComQIPtr is used to get a hold to the IDBProperties interface of the Data Source
object. Then, the IDBProperties::GetProperties is called with a DBPROPIDSET
structure specifying the two properties wanted and the values of the properties will be
returned through a DBPROPSET structure. Both DBPROPIDSET and the DBPROPSET
are array type structures that make transfer of set of Property information easy. Refer to
the OLE DB 2.5 specification for details on those structures. If users fail to submit the
two path of the first trial, Initialize calls the IDBPromptInitialize::PromptDataSource
to activate the Data Link Properties window to allow users to re-submit the paths.

By using the IDBProperties::GetProperties and the
IDBPromptInitialize::PromptDataSource method, Initialize actually behaves like an OLE
DB Consumer, which calls upon different, interfaces of the Provider. This method is actually more elegant and allows developers to bypass the use of ATL internal data structures, which are all templaterized and require serious hacking of the ATL source codes.

**Session Object**

The Session object of the Cameleon Provider requires more implementation than the DataSource object. Similar to the DataSource object, ATL provides all of the implementation classes for all interfaces implemented by the Cameleon Provider’s Session object. However, the IDBSchemaRowset interface requires major modifications. Also, since the Wizard generated Provider only implements the three required Schema Rowsets, the Cameleon Provider has to build the Catalog and Schemata Rowsets from scratch.

Each Schema Rowset has its own class. For example, the Table Schema Rowset of the Cameleon Provider is implemented as class `CCamProvSessionTRSchemaRowset`. Each Schema Rowset class is derived from the `CRowsetImpl` class, which is a standard OLE DB rowset implementation without requiring multiple inheritance of many implementation interfaces. The only method one needs to implement when using `CRowsetImpl` is the `Execute` method, which performs the filling of rowset data. The instantiation of the `CRowsetImpl` object requires three template parameters. The class declaration of the Table Schema Rowset of the Cameleon Provider is shown to illustrate the concept:

```cpp
class CCamProvSessionTRSchemaRowset : public CRowsetImpl< CCamProvSessionTRSchemaRowset, CTABLESRow, CCamProvSession>
```

The first parameter is the user’s class that derives `CRowsetImpl`. The second parameter is the storage class to hold one row of data of the Table Rowset. The third parameter is the class that contains the properties of the rowset, the Session object in this case. The fourth parameter, which is not used in Schema Rowset, will be introduced in the discussion of the Rowset object. Every Schema Rowset class uses a unique container class similar to the CTABLESRow due the fact that different Schema Rowsets hold different restriction columns, and therefore, require different member variables to hold the restriction column values. A rowset container class holds one member variable to represent each restriction
column, and it also provides the implementation of *ClearMember* method that initializes all the data member values to null or zero. Furthermore, each rowset container class contains a Provider Column Map that binds its member variables to the appropriate column names for display. Figure 4.2-4 shows a Column Map that binds the CTABLESRow member variables to their column restriction names.

BEGIN_PROVIDER_COLUMN_MAP(CTABLESRow)
  PROVIDER_COLUMN_ENTRY("TABLE_CATALOG", 1, m_szCatalog)
  PROVIDER_COLUMN_ENTRY("TABLE_SCHEMA", 2, m_szSchema)
  PROVIDER_COLUMN_ENTRY("TABLE_NAME", 3, m_szTable)
  PROVIDER_COLUMN_ENTRY("TABLE_TYPE", 4, m_szType)
  PROVIDER_COLUMN_ENTRY("TABLE_GUID", 5, m_guid)
  PROVIDER_COLUMN_ENTRY("DESCRIPTION", 6, m_szDesc)
  PROVIDER_COLUMN_ENTRY("TABLE_PROPID", 7, m_ulPropID)
END_PROVIDER_COLUMN_MAP()

Figure 4.2-4 Sample Column Map.

*PROVIDER_COLUMN_ENTRY* is a complicated macro that takes in the name, ordinal and member variables of a column and automatically determines the rest of the column information such as column size and data type. Determining column information is necessary for the use of IAccessor and IColumnsInfo interfaces. All Rowset objects with column information that can be determined before compile time use the Provider Column Map to provide static binding of columns and their corresponding variables. Section 4.3 will discuss how to create dynamic bindings of columns in presenting result data from Cameleon.

ATL provides the implementation of the container classes for the Table, Column and Provider Type Scheme Rowsets. The Cameleon Provider has to implement its own container classes for the Catalog and Schemata Rowsets, in addition to all the *Execute* methods of the five Schema Rowset classes.

The *Execute* method of each Schema Rowset is relatively similar, with the exception of that of Provider Type Rowset, which implementation does not require XML parsing. The implementation of the *Execute* method can be separated into three parts:

- **Setup:** This stage includes the initialization of the container class, the acquisition of the Schema Information path and the creation of the XML DOM class object.
- **Value filling:** This stage involves picking out the appropriate data from the XML DOM object to fill in the values of the container class. A loop is used to iterate
through the all the rows in the rowset. Refer to Section 4.4 for detail discussion on XML parsing.

- **Installation:** Finally, the value-filled rows needed to be added to a data structure called `m_rgRowData` such that they can be displayed. `m_rgRowData` is a member variable of the `CRowsetImpl` class. It is an array of the container class object, which holds all the row data that belongs to its parent Rowset object. Since all the data fetching methods of the `CRowsetImpl` class fetch directly from `m_rgRowData`, adding the container object to `m_rgRowData` is the only way to display Rowset data using the `CRowsetImpl` class.

Finally, in order to let Consumers know about what Schema Rowset are available, the Cameleon Provider needs to update the Schema Map. The Schema Map shares similar functionality and structure with all the Maps introduced so far. It is a macro that maps the Schema Rowset GUID to the class that implements the Schema Rowset. Figure 4.2-5 shows the Schema Map of the Cameleon Provider.

```
BEGIN_SCHEMA_MAP(CCamProvSession)
  SCHEMA_ENTRY(DBSCHEMA_TABLES, CCamProvSessionTRSchemaRowset)
  SCHEMA_ENTRY(DBSCHEMA_COLUMNS, CCamProvSessionColSchemaRowset)
  SCHEMA_ENTRY(DBSCHEMA_PROVIDER_TYPES, CCamProvSessionPTSchemaRowset)
  SCHEMA_ENTRY(DBSCHEMA_CATALOGS, CCamProvSessionCatSchemaRowset)
  SCHEMA_ENTRY(DBSCHEMA_SCHEMATA, CCamProvSessionSchSchemaRowset)
END_SCHEMA_MAP()
```

**Figure 4.2-5 Schema Map of Cameleon Provider.**

**Command Object and Rowset Object**

The ATL also supports the implementation classes of all the required interfaces of the Rowset and Command objects of the Cameleon Provider. However, the behavior of these two objects deviates significantly from that of the Wizard generated Provider, and therefore, these two object require some serious modifications. Most of the changes required stem from the fact that the Cameleon Provider does not know before compile time the binding of columns. Therefore, it cannot reuse the same static binding strategy introduced in the previous section. The changes involved in creating dynamic column bindings are complicated and deserve a whole section of discussion. In the remaining of this section, changes not related to column bindings will be mentioned. And the next section will carry out a thorough discussion on dynamic column bindings.
The Command object, like the Data Source and the Session object, uses inheritance to specify the interfaces it implement. On the other hand, the Rowset object does not require the long inheritance chain due to the fact that the Rowset object inherits from the CRowsetImpl class, which provides implementation of all the necessary interfaces for the Cameleon Rowset object.

Parts of the Command object requires re-implementation to meet the design goal. First, the SetCommandText method of the ICommandText interface is changed such that it first acquires the Cameleon Server path from the Data Source object, and then appends to the Cameleon Server path the actual SQL query command submitted by users to form a http request string that fits Cameleon input format. The input format for the Cameleon Web Wrapping Engine through HTTP is as follow:

![Figure 4.2-6 Three parts to Cameleon HTTP input.](image)

By appending the three parts together, one gets a complete command string that can be sent through HTTP to retrieve result in XML format. The first part of the command is obtained through user input during initialization (the Data Source field) and the concatenation of the “query=” flag. The second part of the command, the SQL query, is stored in the user input command CComBSTR variable: m_strCommandText of the ATL ICommandTextImpl class. And since the design of the Cameleon Provider only handles XML output from Cameleon, the third part of the command, which specifies output format of Cameleon, is appended to the end of the command string. The completely new command string is stored back to the m_strCommandText variable for execution.

Second, a boolean flag, m_bNoCommand, is added as a member variable in the Cameleon Provider Command class to indicate whether the command text has been set. This is to make sure that users have submitted a query and the query has been turned into a Cameleon command string. At the end of the SetCommandText method, the m_bNoCommand is set to zero to indicate the command string is properly set.
Execute method of the Command object will check the status of the m_bNoCommand flag before it proceeds to process the command.

4.3 Displaying Result Rowset in Cameleon Provider

The Wizard generated Provider differs significantly from the Cameleon Provider in its Query Processing capability. As mentioned before, the Wizard generated Provider allows users to submit command optionally. Users can use command to specify the directory they want to view file information from, but no matter which directory users specify, the columns in the resulting rowset always stay the same. In other words, the Wizard generated Provider does not allow users to specify what result columns they want to get back. On the other hand, the Cameleon Provider provides more sophisticated Query Processing power by allowing users to submit SQL queries that specify the columns they want to view data from. Since the Cameleon Provider cannot determine column information before compile time, it cannot use the same methods it uses to display Schema Rowset. Instead, the Cameleon Provider has to determine column information during run-time, which requires structural change in the Command and Rowset objects of the Wizard generated Provider. This section first gives a brief overview on the layout of the Rowset and Command object in the Wizard generated Provider, and then proceeds to examine the changes involve.

4.3.1 Rowset and Command with static binding of data columns

The Wizard generated Provider’s Rowset object has a helper class called the WindowsFile class. The WindowsFile serves as the container class to the Rowset object, which contains and describes the columns in the Provider’s result rowset. Within the WindowsFile class, there is a Provider Column Map that binds column names to their ordinal and data members. The use of this WindowsFile container class in Rowset class is the same as that of CTABLESRow in the Table Schema Rowset as discussed in 4.2.2. Since the Rowset class of the Wizard generated Provider is also derived from the CRowsetImpl class, it also requires the implementation of the Execute function to fill the data to the result rowset. Similarly, all the result row data needs to be added to the
m_rgRowData structure for display. The general framework for the original rowset implementation is basically the same as that of the Schema Rowsets.

However, unlike Schema Rowsets, which are invoked by the calling of IDCBSchemaRowset::GetSchemas, the result Rowset was generated by the Command object upon execution of a valid query. Therefore, in the Execute method of the Command object, it has to call forth the creation of the Rowset object, which will then trigger the Execute method of the Rowset object to fill and display the result row data.

4.3.2 Rowset and Command with dynamic binding of data columns

With undetermined column information, the Cameleon Provider can no longer supports the use of the Provider Column Map and its fellow macros. A completely new set of container classes are defined, and new methods are implemented to determine column information during run-time. This section discusses each of them in details.

The CCamProvRecord class

The CCamProvRecord class is the new container class to hold one row of data for the Cameleon Provider. Instead of a statically assigned Provider Column Map, the CCamProvRecord class implements a template function GetColumnInfo to dynamically determine column information of the result Rowset during run-time.

The GetColumnInfo function returns a pointer to an ATL COLUMNINFO structure defined in the ATL. The ATL COLUMNINFO holds all the necessary information about a particular column, including the name, ordinal, data type, precision, size, OLE DB-defined column flags, scale, column id (if defined) and the offset to that column in the buffer. The GetColumnInfo method is to provide the functionality provided by the PROVIDER_COLUMN macros.

However, another problem arises in determining the column information of the result data due to the fact that the Cameleon Provider has no control on what SQL statement the users submit. The Cameleon Provider would not know about the column information until it has executed the command and gotten back the result from the Cameleon Engine. As the result, GetColumnInfo needs to carry out the functionality of
Execute function of the Command object at first. But instead of implementing the same set of code twice, the GetColumnInfo method calls Execute method of the Command object. Furthermore, since the Command object also requires the use of column information for its IColumnInfo interface, Execute method of the Command object is also responsible for computing column information and putting them in the m_rgColInfo array. The m_rgColInfo is an array structure that holds ATLCOLUMNINFO structures in both the Command object and the Rowset object. For each column in the result Rowset, m_rgColInfo should contain a corresponding ATLCOLUMNINFO structure that describes the column. By calling Execute method of the Command object, GetColumnInfo has accomplished most of its requirement. The remaining task is to return the m_rgColInfo array to the caller.

The CCamProvArray class

The CCamProvRecord class solves the problem of determining column information. However, it does not provide member variables within the container class to hold the actual values of each column. In fact, it is no longer feasible to specify a few variables within a container class to hold the result column data since different commands yield different column combinations. The Cameleon Provider instead uses a large data buffer to hold all the result data, and using the column offsets specified in the column information to locate each column data in the correct destination within the buffer. This data buffer is implemented in CCamProvArray class.

Section 4.2.2 talks about the template parameters required to use the CRowsetImpl class, however, the fourth template parameter is not used in the Schema Rowset. The new Rowset class of the Cameleon Provider makes use of the fourth parameter to specify the use of a new array type data structure: CCamProvArray. To help the understanding of the template parameters, the declaration of CRowsetImpl class is shown as follow:

```cpp
template <class T, class Storage, class PropClass, class ArrayType = CSimpleArray<Storage>>
class CRowsetImpl
```

The fourth parameter is the storage array class for the rowset’s data. When this parameter is not passed in, the default CSimpleArray, as defined in the ATL, is used. The important m_rgRowData variable of the CRowsetImpl class is an instance of the class to
the fourth parameter. The CCamProvArray not only provides all the methods supported in the CSimpleArray class, which include array initialization and indexing functionality, it also holds the following member variables:

- **BYTE* m_pData**: This is a data buffer that is big enough to hold all the data in the result Rowset.

- **int m_nSize**: This variable indicates the number of rows of data in the result Rowset.

- **int m_cbElemSize**: This variable specifies the size in byte of each row. It is equal to the sum of the sizes of all columns in a row. The array class uses this to provide indexing to different rows within the Rowset. For example, if a row consists of 4 columns and each of the column is of size 5 bytes, then m_cbElemSize is 20; and &m_pData[0] holds the beginning of the data for the first row, and &m_pData[20] holds the beginning of the data for the second row.

- **XMLDOMWrapper* m_pResultDoc**: This is the pointer to the XMLDOMWrapper class that holds the XML result from the Cameleon Engine.

The CCamProvArray also provides the functionality to retrieve result data from the XML object, and insert each column data into the correct destination in the buffer bases on the information in m_rgColInfo. Since all the data is stored within a data buffer, it is crucial for the member functions of CCamProvArray to perform the correct pointer arithmetic such that they return the correct piece of data in the data buffer.

**FillColInfo method in Command**

Within the Command object, a new method, FillColInfo, is implemented. FillColInfo is a helper function used in Execute, which is responsible for retrieving and parsing out XML result from Cameleon, and determines column information for each column in the result set. The XML parsing is done through the help of the XMLDOMWrapper class, which will be discussed in details in the next section. For each column in the result data, FillColInfo fills a ATL_COLUMNINFO structure with the column information values and then adds it to the m_rgColInfo array structure for display and storage. In addition, FillColInfo also computes some important information needed for the initialization of the m_rgRowData, such as the total number of rows in the result Rowset, and size (in byte) of a row of data.
There are two special pieces of implementation details that are worth mentioning in the `FillColInfo` method. They are the algorithms employed to determine column size and column offset. Since column values varies from row to row, therefore, it is necessary to pick a column size that is big enough to hold all possible values of a column. The current implementation would scan through all the result data of a particular column, and uses the maximum data size as the column size. And since all the column data is of the form of `CComBSTR`, when calculating offset, two extra bytes have to be added for each column data to account for the null terminating value. The `Length` method within `CComBSTR` class is used to return the number of character excluding the null terminating character in the string. To calculate the size of the `CComBSTR`, one is added to the length of the `CComBSTR` to account for the null terminating character and then multiplied by the size of `OLECHAR`. It is very important to consider the null terminating character when calculating column size and offset since it separates one column data from another.

**Execute method in Command**

The `Execute` method in Command holds the different pieces together and drives the whole process from command execution, column information determination and finally the filling of rowset data to the `m_rgRowData` structure. When a Consumer calls forth `ICommand::Execute`, the following steps happen:

- **Rowset Creation**: A Rowset object is created within the `Execute` method to hold the result data.
- **Result retrieval and filling in column information**: The `FillColInfo` is called to perform this task. The `m_rgColData` is filled with column information at the end of the stage.
- **Filling in row data**: The `m_rgRowData` is initialized. Methods in `CCamProvArray` is called to fill `m_rgRowData` with result row data bases on column information as specified in `m_rgColData`. 
At the end of Execute, the \texttt{m\_rgColData} and \texttt{m\_rgRowData} arrays contain the column information and the row data result respectively. And other methods in \texttt{CRowsetImpl} class can access the two arrays to perform data transfer and data display.

4.4 XML Parsing

As mentioned in chapter two, both Schema Information and query result are stored in XML format to provide the Cameleon Provider a common XML interface on Schema Rowset and Query Processing. The XMLDOMWrapper class is created to provide abstraction on the retrieval and parsing of XML data. This section examines the algorithms used in XML parsing and the actual implementation of the XMLDOMWrapper class. The structure of Schema information XML document and the Cameleon Result XML document will also be discussed to provide a better understanding of the design of the parsing algorithms.

4.4.2 XML Document Object Model (XML DOM) Technology

The XML Document Object Model (XML DOM) provides a standard set of objects for representing XML documents, and a standard interface for accessing and manipulating those objects. The XMLDOMWrapper class is an encapsulation of some of the often used XML DOM programming interface syntax in Cameleon Provider. There are more than ten interfaces defined in XML DOM, and each of the interface provides the methods to construct and manipulate an object with a XML document. Some of the most often used C++ interfaces by the XMLDOMWrapper class are listed as follow, refer to Visual C++ manual for a more complete list of interfaces:

- \texttt{IXMLDOMDocument}: this represents the top node (Document node) of the XML DOM tree, it includes methods to obtain or create all of the other XML DOM interfaces.
- \texttt{IXMLDOMNode}: this represents a single node in the document.
- \texttt{IXMLDOMNodeList}: this represents a collection of XMLDOMNode. Users can index and iterate through the collection of nodes using methods in this interface.
• **IXMLDOMNodeNamedNodeMap**: this represents a collection of attributes for a single node. Methods are provided for access and iteration of different attributes in the map.

• **IXMLDOMElement**: this represents a XML Element object. Mainly used for the retrieval of Document Element.

By using the above object interfaces, the XMLDOMWrapper class can transverse the XML DOM trees representing the Cameleon Result and Schema Information. It also provides the methods for easy access of data in the two XML document without callers going through the details of the XML DOM interface syntax.

**Structure of Cameleon Result XML document**

Figure 2.3-1 shows a sample fragment of the Cameleon result XML document. It has a Document Element named “DOCUMENT”, which encloses all the result data. Each row of the result data is enclosed by an “ELEMENT” node; and within the “ELEMENT” node are the column data with the column names as the node names. This kind of hierarchy structure facilitates parsing of row data since each “ELEMENT” node is equivalent to a row of the result Rowset. And by evaluating the node name and node value of all the child nodes of each “ELEMENT” node, one can easily obtain column names and their corresponding column data of the row.

**Structure of Cameleon Schema information XML document**

Figure 3.2-2 shows a sample fragment of the Cameleon Schema information XML document. As mentioned in section 3.2, the Schema information on the XML document is organized in a hierarchical manner resembles that of a DBMS. Each Schema Rowset is a type of node. Each type of node contains a number of attribute name and value pairs, which represent the restriction column names of each Schema Rowset and their values. This particular organization also makes parsing easy. To process each kind of Schema Rowset, one only needs to gather all the nodes with their node names equal to the Schema Rowset and assign the restriction column values to the appropriate attribute values.
The XMLDOMWrapper Class

The XMLDOMWrapper class is developed during the implementation of the Cameleon Provider. The XMLDOMWrapper class encapsulates the use of XML DOM interfaces and provides all the necessary functionality to access data in both XML documents in simple manners. It also turns XML document into a C++ object, and allows users to keep the object around and uses its member functions to access data when needed.

The XMLDOMWrapper class contains as member variable a IXMLDOMDocument pointer, m_pDoc, which points to the top of the underlying XML document source. From this top level pointer, the XMLDOMWrapper class uses different interface calls to get to the right data piece embedded within the XML document. To summarize, there are four different kinds of member functions in the XMLDOMWrapper class designed to access different kinds of data in the Result and Schema Information XML documents.

- **Document initialization:** the LoadXMLDoc method takes in a URL in CComBSTR form. It initialis the m_pDoc variable to point to the XML document specified in the input URL through the use of IXMLDOMDocument::Load method. Before the execution of the IXMLDOMDocument::Load method, LoadXMLDoc call forth the IXMLDOMDocument::put_async method and sets download mode to synchronous to make sure download is finished before the program moves on.

- **Nodes retrieval by tags:** the GetNodeListbyTag method takes in the desired node name and returns all the nodes of that name in the form of a node list. It is done by first acquiring the Document Element of the underlying XML source and then calling the IXMLDOMElement::getElementsByTagName method, which will return all its child nodes with name equals to the tag name. And using the GetNodeListbyTag method, different kind of nodes can be retrieved. For example, to get all the result rows from the Result XML, all the nodes with the name equals “ELEMENT” are first retrieved. Also, Schema Rowset of the same kind can also be retrieved this way (by supplying “CATALOG”, “SCHEMA”, “TABLE” or “COLUMN” as tag name).
- **Attributes evaluation:** In order to find out values of restriction columns of each Schema Rowset, the XMLDOMWrapper class needs some mechanisms to parse out different attribute values. The `GetAttributebyTag` method first creates an `IXMLDOMNamedNodeMap` structure of the node, which contains all the attributes and the corresponding values. And then it uses the `IXMLDOMNamedNodeMap::getNamedItem` method to get the attribute with the specified attribute named and stores it in a `IXMLDOMNode` structure. Finally, the `IXMLDOMNode::get_nodeValue` is used to get the value of the attribute.

- **Column evaluation:** The XMLDOMWrapper class provides the mechanism to find out the names and values of all the columns within a result row data from the Cameleon Result XML document. As mentioned in the previous section, each the “ELEMENT” node in the Cameleon Result XML document represents a result row. And the all the child nodes to a “ELEMENT” node is the columns within the result row. To obtain column name and value pairs, the `EnumerateResultColumn` method iterates through all the “ELEMENT” nodes and evaluates all the node names and values of each “ELEMENT” node. Even though it seems redundant to evaluate all the “ELEMENT” nodes, since one would expect all rows within a Rowset to have the same columns, this process is necessary since `GetColumnInfo` of the Command object uses `EnumerateResultColumnSize` method of the XMLDOMWrapper class to find out the maximum size for a particular column.

The above four kinds of functions make up the basis of the XMLDOMWrapper class. There are more high level functions defined within the class to achieve different purposes using the above four kinds of functions. For example, `EnumerateCatalog` and `EnumerateSchema` are methods that call `GetAttributeByTagName` to retrieve attributes with different names. All the methods that require access to XML data use the XMLDOMWrapper class in the Cameleon Provider. Refer to Appendix E for the documentation of the XMLDOMWrapper class.
Chapter 5 Testing and Experiments using the Cameleon Provider

This section talks about the testing and experiments that have been carried out on the OLE DB Provider for Cameleon. First, testing is carried out to make sure that all the interfaces implemented by the Cameleon Provider conform to the OLE DB specifications and all the methods produce desired behavior as described in the design goal. Then, experiments are performed in using the Cameleon Provider with OLE DB-compliant applications, such as Microsoft SQL Server and Access, to determine whether the OLE DB Cameleon Provider can truly deliver what it promises: serves as a bridge between Cameleon data source and OLE DB-compliant applications.

5.1 Unit Testing

Several general purpose OLE DB Consumers come with the Microsoft Platform Software Development Kit (SDK) package. One of them is the Rowset viewer. It offers a simple way to view and manipulate rowsets, with the added ability to call and manipulate other OLE DB methods from Data Source, Session, Command, Rowset, Transaction and Error objects supported by any OLE DB Provider. The Rowset Viewer is used to perform unit testing since it provides a handy way to manipulate different interfaces of the OLE DB Provider.

The Rowset Viewer is a particularly effective tool in debugging and testing the Cameleon Provider. First of all, it allows users to test each individual method in all of the interfaces implemented by the OLE DB Provider and therefore allows users to confirm the interface conformance of the OLE DB Provider. Developers can make interface calls in the order they want to test out the Provider’s different modules. Second, the Rowset Viewer allows users to examine OLE DB Properties in each components, and set different Property values to see how the Provider behaves. Third, the Rowset Viewer also supports the display and manipulation of Schema Rowset Information and the execution of command, which comprise the most significant part of the Cameleon Provider. Finally, upon encountering of error, the Rowset Viewer invokes the debugger.
of Microsoft Visual C++ 6.0 to facilitate debugging of the Provider. The feature is particularly useful since it singles out the line of code that generates the error, and developers can save a lot of time in locating errors.

**Result of Rowset Viewer Testing**

Test run with the Rowset Viewer showed that the Cameleon Provider supports the required interfaces. Each method on all the interfaces implemented by the Cameleon Provider was tested upon using the Rowset Viewer and they all generate desired behavior. In addition, some provider-specific behaviors are tested. They include:

- **Setting Initialization Properties:** Figure 5.2-1 shows the screen shot when the Cameleon Provider is invoked and users are prompted for initialization properties. Note that the DataSource field is supplied with the Cameleon Server path, and the ProvString field is supplied with the Schema information path.

- **Schema Rowsets retrieval:** Figure 5.2-2 shows the Rowset Viewer retrieves the Column Schema Rowset information from the Cameleon Provider. All other Schema Rowset retrieval are tested to make sure they return correct values.

- **Command Text setting and Command Execution:** Figure 5.2-3 and figure 5.2-4 show the execution of ICommandText::SetCommandText and ICommand::Execute. Testing is performed to make sure that commands yield correct result, and command cannot be executed without command text being set.
Figure 5.2-1 Invoking the Cameleon Provider and supplying with the Cameleon Server path in the Data Source field and the Schema information path in the Provider String field.
Figure 5.2-2 Rowset Viewer displaying Column Schema Rowset Information.

Figure 5.2-3 Rowset Viewer setting command text.
The result of testing with Rowset Viewer confirms that the Cameleon Provider’s interfaces meet OLE DB specifications. It also confirms that the Cameleon Provider meets its design goal by producing the correct behavior when each method is called.

5.2 Experiments with OLE DB-compliant applications

Microsoft SQL Server 7.0 is chosen to be the cornerstone application to perform experiments on the Cameleon Provider based on the following:

- It is the most powerful DBMS engine among all the OLE DB-compliant applications. Therefore, SQL Server 7.0 is actually the ideal application to be used to drive Cameleon queries through the Cameleon OLE DB Provider. SQL Server 7.0 can provide the most relational-query capabilities to the Cameleon source.
- Microsoft SQL Server 7.0, being an industry standard, can be accessed through other popular Microsoft applications, such as Microsoft Access, Microsoft Excel,
and Active Data Object (ADO). It also accommodates the use with Oracle DBMS and other major data systems. Therefore, making SQL Server 7.0 as the underlying engine for Cameleon opens up Cameleon’s data to all the applications that normally work with Microsoft SQL Server.

- The use of OLE DB Provider with SQL Server 7.0 is well documented. Experimentation with SQL Server 7.0 can be categorized into two stages. First, ad hoc distributed queries are used to access data on Cameleon through the Cameleon Provider using the *OpenRowset* and *OpenQuery* functions. This method allows users to use complicated SQL queries to manipulate result from Cameleon. However, users do not have access to metadata (i.e. Schema Information) of Cameleon data source, therefore users have to know exactly what is available from Cameleon to make a query. Second, the linked server feature on SQL Server 7.0 is used to treat the Cameleon Provider and its underlying data source as a database-like source that allows submission of regular SQL queries. This method allows users to treat Cameleon Provider as any other regular database sitting on the SQL Server and allows users to browse through Cameleon Schema Information. It is a more demanding scheme and therefore requires the Cameleon Provider to work fully with SQL Server in all aspects.

### 5.2.1 Linked Server and Distributed Queries of MS SQL Server 7.0

Through Linked Server and distributed queries, Microsoft SQL Server 7.0 can access OLE DB data sources. SQL Server implements persistent storage of an OLE DB provider name and data source definition called a linked server. With the set up of an OLE DB Provider and its underlying data source as linked server, one can execute Transact-SQL (SQL used in MS SQL Server) statements against the OLE DB data source, retrieve metadata of the underlying data source and use the Linked Server definition to issue distributed queries. Distributed Query is a feature of Microsoft SQL Server 7.0 that allows users to access data outside a SQL Server-based server, either within other servers running SQL Server or other data sources that expose an OLE DB interface, such as a Linked Server encapsulating an OLE DB data source.
With the set up of a Linked Server for the Cameleon Provider and the use of distributed queries, there are two possible ways to access Cameleon data sources through the use of SQL Server:

- **Ad hoc distributed queries:** For infrequent references to an OLE DB data sources, one can use the OpenRowset function to specify the OLE DB Provider name one intend to use during run-time without setting up a persistent linked server storage. The rowset resulted from the execution of OpenRowset function can be referenced the same way a regular database table is referenced. Figure 5.3-5 shows the use of the OpenRowset function on Cameleon data source through the Cameleon OLE DB Provider.

```
select * from
openrowset ('OLEDBCamProv.CamProv', '', 'Select bankname, rate, yield, minbalance from moneyrates')
```

**Figure 5.2-5 Sample distributed query with the use of the OpenRowset function.**

Other than the OpenRowset function, one can also use the OpenQuery function to achieve the same result. However, OpenQuery requires the set up of a linked server with all the connection properties specified for the OLE DB Provider. The syntax of the OpenQuery function is as follow:

```
OPENQUERY(linked_server, 'query')
```

The use of OpenQuery function is important when other clients, such as Microsoft Access on another machine, want to access OLE DB data through SQL Server but do not have a copy of the Provider installed on the client machine. Client applications without a copy of the OLE DB Provider installed on the machine can only access OLE DB data source views defined with OpenQuery function with linked server names specified through SQL Server. If an client application tries to access view defined by OpenRowset function through SQL Server, an error statement “could not instantiate OLE DB Provider” is returned since OpenRowset function tries to locate the Provider on client machine with the name supplied. On the other hand, if access is done through view defined by OpenQuery, the request will be processed by the linked server residing on the SQL Server machine, where the OLE DB Provider is
hosted. Therefore, using the OpenQuery function enables remote access to OLE DB data from client machines without installing the OLE DB Provider on client machine. Note also that when executing remote access, the Distributed Transaction Coordinator of SQL Server 7.0 has to be turned on. [7]

- **Linked Server name:** The system stored procedure sp_addlinkedsrv is used to give a server name to an OLE DB data source. Objects in these linked servers can be referenced in Transact-SQL statements using four-part names. For example, if the Cameleon Provider is set up as a linked server called CAM, then to access data within the moneyrates web site, one would do the following:

```
SELECT * FROM CAM.CAMELEON.COIN.MONEYRATES
```

The four part object reference comprises of the names of linked server, catalog, schema, and table arranged in the following manner:

`linked server name.catalog name.schema name.table name`.

Since it requires the use of catalog, schema and table name, the OLE DB provider has to provide accurate metadata of its underlying data source for this method to work.[8]

**5.2.2 Experiments with Microsoft SQL Server 7.0**

In this section, some of the example experiments on using the Cameleon Provider with Microsoft SQL Server 7.0 are listed. Refer to appendix G for more detail description on using OLE DB data source with SQL Server.

**Stage One: Experiments with ad hoc Distributed Queries**

Testing of the Cameleon Provider with ad hoc distributed queries helps confirm that users can submit SQL commands to the Cameleon Provider through OLE DB-compliant applications. It is because once SQL Server can access Cameleon data through distributed queries, it also guarantees that all the other OLE DB-compliant applications, such as Microsoft Access and ADO, will also be able to access Cameleon data due to the fact that most of the popular Microsoft applications allow the import of SQL Server data.

Furthermore, with the help of the OpenRowset function, one can treat the retrieved result rowset as a table and perform other queries on top of the result, or combine the retrieved result with other data source that can be accessed by SQL Server. In a distributed query statement with OpenRowset function, one can treat the whole
OpenRowset function as a table, and anything SQL statement outside of the OpenRowset function will be executed by the query processor of SQL Server 7.0 [8]. As a result, users can use sophisticated queries to manipulate Cameleon data source with the help of a more powerful query processing engine from SQL Server 7.0. The testing performed in this stage aims to measure how effective does SQL Server enhance the query capabilities of Cameleon data through the use of OLE DB Provider. Also, investigation on the use of Access to retrieve Cameleon data through SQL Server is also carried out.

**Result**

Through distributed queries, users can manipulate Cameleon data result in various fashions. Test queries that have been carried out on the Cameleon Provider and they all yielded correct result within an acceptable amount of time. The nested query shown in figure 5.3-6, which requires sending two requests to the Cameleon Engine, took about 12 seconds to complete, and all the other queries finish within that time frame. Some sample types of query include nested query, queries with special function such as order by and max, etc. Join statements can also be used on two Cameleon sources or on a Cameleon source with a DBMS source. Figure 5.3-6 to figure 5.3-8 show some of the queries executed through SQL Server Query Analyzer.

```sql
select BANKNAME, RATE, YIELD from openrowset ('OLEDBCamProv.CamProv', ", 'Select bankname, rate, yield, minbalance from moneyrates')
where RATE > 6.00 and yield < 6.75
```

*time took to execute: 5 sec*

**Figure 5.2-6 Query with multiple conditions.**

```sql
select BANKNAME, RATE from openrowset ('OLEDBCamProv.CamProv', ", 'Select bankname, rate, yield, minbalance from moneyrates')
where RATE = (select max(RATE) from openrowset ('OLEDBCamProv.CamProv', ", 'Select bankname, rate, yield, minbalance from moneyrates'))
```

*time took to execute: 12 sec*

**Figure 5.2-7 Nested queries with special function.**
select a.BANKNAME, b.LOCATION
from openrowset ('OLEDBCamProv.CamProv', '', 'Select bankname, rate, yield, minbalance from moneyrates') as a,
CamProvDemo..Bank as b
where a.BANKNAME=b.BANK_NAME

time took to execute: 5 sec

Figure 5.2-8 A join statement on Cameleon source and SQL Server source.

It is confirmed that most of the commonly used SQL queries can be used effectively with the OpenRowset function to manipulate Cameleon result.

The use of View

Even though the use of ad hoc distributed query is effective, however, it requires users to be familiar with the syntax of distributed query. To increase the ease of use of Cameleon source through SQL Server, one can create a View that contains the openquery function such that users can treat the whole openquery rowset as a table. This method allow users to submit regular SQL query to manipulate Cameleon data source. Furthermore, through the use of View, other applications, such as Microsoft Access, is able to access the View on SQL Server as a regular table. Refer to appendix G for detail instruction on using Access to access OLE DB data sources through SQL Server. Figure 5.3-9 shows the query used to create the moneyrates View, and figure 5.3-10 illustrates the use of the moneyrates View. Figure 5.3-11 repeats the query of 5.3-6 using View and illustrates that performance is not affected through the use of View. Finally, figure 5.3-12 shows how Access retrieves Cameleon data source.

create view moneyrates (bankname, rate, yield, minbalance) as
select * from openquery (CAM, 'Select bankname, rate, yield, minbalance from moneyrates')

time took to execute: 4 sec

Figure 5.3-9 Transact-SQL query to create the moneyrates view.

select * from moneyrates where rate = 6.56

time took to execute: 16 sec

Figure 5.3-10 Sample use of the moneyrates view.
select BANKNAME, RATE, YIELD from moneyrates where rate > 6.00 and yield < 6.75

time took to execute: 3 sec

Figure 5.3-11 Query of figure 5.3-6 rewritten using View

![Moneyrates View](image)

Figure 5.3-12 The use of Access data page to access moneyrates data.

Stage one experimentation was extremely successful. Users can access Cameleon data source through SQL Server by means of ad hoc distributed queries. The use of distributed queries also enables users to manipulate Cameleon data source using SQL Grammar not currently supported by the Cameleon Engine. More importantly, by wrapping the distributing queries in Views, other OLE DB-compliant applications, such as Access are able to access Cameleon data sources. At this stage, the Cameleon Provider has achieved everything the COIN ODBC driver provides, plus more [1].
Stage Two: Experiments with Linked Server

With the use of the Linked Server feature, an OLE DB Provider that fully works with SQL Server would allow users to submit regular Transact SQL queries to retrieve OLE DB data source as described in section 5.3.1. When a Transact SQL query is submitted with a four part name to access an OLE DB data source, the SQL Server query processor promises to determine the capabilities of the underlying data source through properties specified in the provider, and plan the query such that whatever relational capabilities requested by the submitted query that are not supported by the underlying OLE DB data sources would be carried out by the SQL Server query processor. For examples, if the underlying data source does not support the use of join statement, then SQL server will perform the join using its own query processor without sending the join request to the underlying OLE DB data source. This method is the ideal way to incorporate Cameleon data source to OLE DB-compliant applications since it not only avoids the use of special function OpenRowsset, but also allows users to browse Cameleon metadata. However, this is also a more demanding method and requires the underlying OLE DB data source to satisfy more rigorous requirements.

There are four categorization of OLE DB providers based on their capabilities from a SQL Server querying standpoint, and each kind of provider needs to satisfy different criteria in order to fully work with SQL Server using Linked Server name. The Cameleon Provider belongs to the SQL Command Provider category due to its support of Command object, which allows users to submit SQL queries. All SQL Command Providers intend to fully work with SQL Server need to satisfy the following requirements:

- The OLE DB Provider must support the Command object and all the mandatory interfaces of the Command object. They include: ICommand, ICommandText, IColumnsInfo, ICommandProperties, and IAccessor.
- The SQL dialect supported by the OLE DB Provider must be at least ODBC Core level on the ODBC conformance scale or SQL 92 Entry level on the SQL-92 conformance scale. The DBPROP_SQLSUPPORT property of the Data Source Info Property Group is used to report the SQL dialect supported. [6]
As mentioned in Chapter 3, the Cameleon Provider implements the Command object and all its mandatory interfaces. Therefore, it satisfies the first requirements. However, the SQL dialect supported by Cameleon Web Wrapper Engine is not sufficient to satisfy neither the ODBC Core level nor the SQL 92 Entry level. Table 5.3-1 lists the ODBC Core SQL Grammar conformance, which is a subset of the SQL 92 Entry level conformance [15].

<table>
<thead>
<tr>
<th>Type</th>
<th>Sample Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Definition Language</td>
<td>CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW</td>
</tr>
<tr>
<td>Data Manipulation Language</td>
<td>Full SELECT, INSERT, UPDATE</td>
</tr>
<tr>
<td>Expressions</td>
<td>Subquery, set functions such as SUM and MIN</td>
</tr>
<tr>
<td>Data Types</td>
<td>CHAR, VARCHAR, DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE, PRECISION</td>
</tr>
</tbody>
</table>

Table 5.3-1 ODBC Core SQL Grammar conformance.

Currently, the Cameleon Web Wrapping Engine only supports partial SELECT statements. For some of the web sources, users can specify "where" statement for some designated attributes. However, users are not allowed to use "where" statement on all the available attributes. Besides, since the Cameleon Provider is only a read-only provider and does not support transaction, all the features related to Data Definition and some features of the Data Manipulation Language such as INSERT and UPDATE are not allowed. Finally, Cameleon Wrapper Engine treats all the result data as character strings and stores them as XML document without the support for other data types listed in ODBC Core SQL Grammar. With all the above discrepancies between Cameleon functionality and the required SQL standard, it is currently impossible to make the Cameleon Provider to work fully with SQL Server unless the Cameleon Web Wrapper Engine undergoes some major enhancement, which is beyond of scope of this thesis.
Chapter 6 Conclusions

In this chapter, future development relating to the OLE DB extensions of the Context Interchange System will be discussed. In particular, further development on Cameleon Provider and the Cameleon Engine will be suggested to enhance the capability of the OLE DB Cameleon Provider. In addition, issues on performance tuning on the current OLE DB Cameleon Provider will also be discussed. Finally, the last section of the chapter will summarize the entire OLE DB development project.

6.1 Performance tuning on result data retrieval

As mentioned in section 4.3.2, the Cameleon Provider makes use of the dynamic column binding strategy when retrieving and displaying result rowsets, which requires the determination of each column size during run-time. Currently, the Cameleon Provider uses the XMLDOMWrapper class to loop through all the result data to determine the maximum data size for each column and use the maximum size of each column as the default column size for the result rowset column. Even though this method guarantees to yield to correct column sizes, however, as the amount of result data grows, the performance of the Cameleon Provider will be affected accordingly.

To loop through the result data again during determination of column information is not the most efficient way, consider that the same process also occurs in two other parts of the system. First, all the result data should have been processed once when the Cameleon Web Wrapping Engine goes to the targeted web site to retrieve result data and records them as an XML document. Second, during execution of command, the XMLDOMWrapper class parses the Cameleon result XML file and put all the data in the data buffer of the CCamProvArray. However, since the filling of rowset data to the CCamProvArray data buffer requires the use of correct column information to determine the offset of each piece of data in the buffer, it is therefore not possible to combine this process with the determination of column size. In fact, the ideal solution to this problem would be for the Cameleon Web Wrapping Engine to provide some of the column information, such as column size, as part of its result. This approach eliminates the need to process all result data several times and therefore, improves the performance of the
Cameleon Provider. Furthermore, in future development, when the need for specifying different data types arises to meet certain SQL Grammar standard, the Cameleon Engine can also use its result to indicate data type and other column information of each column. This approach is the most effective and efficient one, but it requires changes on the Cameleon Engine and therefore is not implemented as part of the thesis. Future OLE DB development for Cameleon should consider the proposed design.

6.2 Future Development

After thorough testing and performance evaluation of the OLE DB Provider for Cameleon, several development enhancements are concluded. First, as mentioned in section 6.1, the result rowset retrieval strategy can be made more efficient with the inclusion of column information by Cameleon output. Second, more SQL query capabilities can be added to the Cameleon Engine or supplemented by the Cameleon Provider such that the Cameleon OLE DB system supports at least ODBC Core SQL Grammar. Currently, Cameleon Provider fails to work as a SQL Command Provider within SQL Server because neither the Cameleon Web Wrapping Engine nor the Cameleon Provider supports SQL dialect that is at least ODBC Core level or SQL 92 Entry level. Table 5.3-1 lists the minimum SQL conformance level that the Cameleon Engine should support if it were to work fully with SQL Server. And further development on the Cameleon Engine should base on the minimum SQL conformance level required.

An alternative to enhancing Cameleon Engine is to use the Cameleon Provider on the POE system. The POE system is a relational server that provides access to Cameleon data sources. Besides, it implements its own Query Planner, Optimizer and Execution Engine to enhance query capability of Cameleon data sources. Since the input format of the POE system is similar to the Cameleon Engine (both use HTTP request as input), one can use the Cameleon Provider on the POE system instead. Minor change is needed such that the Cameleon Provider can understand and parse the result from POE. In other words, by modifying the Cameleon Provider, it can be used on other relational front end of the Cameleon Engine rather than on the Cameleon Engine itself, which does not provide adequate relational capability.
Finally, with the success of the Cameleon OLE DB Provider protocol, more OLE DB Provider functionality can be added such that it allows users to manipulate Cameleon data source in more sophisticated ways. Some of the useful features that are missing from the current implementation of the Cameleon Provider include implementing index and bookmark on Rowset object such that users can have more control over data retrieval.

6.3 Conclusion

The OLE DB development for the Context Interchange System is a successful one. The Cameleon Provider enables popular Microsoft applications to access Cameleon data. Furthermore, query capability to Cameleon data source is supplemented by the query processor of SQL Server when accessing Cameleon data sources through SQL Server. Even though the Cameleon Provider does not enable full integration of Cameleon data source with SQL Server due to inadequate SQL support of the Cameleon Engine, this project serves as a useful guideline for future efforts to integrate common applications and the Context Interchange System.
Appendix A  Summary of Interfaces implemented by the OLE DB Provider for Cameleon

This appendix lists all the interfaces and their member functions implemented by the Cameleon Provider. Interfaces are grouped by the Provider object they belong to. Refer to the OLE DB version 2.5 specification [9] for more detail information on each Property.

Interfaces on the Data Source object

**IDBCreateSession**

Interfaces method include:
- **CreateSession**: Creates a new Session object from the Data Source object.

**IDBInitialize**

Interfaces method include:
- **Initialize**: Initializes the Data Source object.
- **Uninitialize**: Returns the Data Source object back to uninitialize stage.

**IDBProperties**

Interfaces method include:
- **GetProperties**: Returns the values of the properties belong to the Data Source.
- **SetProperties**: Set Properties belong to the Data Source.
- **GetPropertyInfo**: Returns information on all properties supported by the Provider

**IPersist**

Interfaces method include:
- **GetClassID**: Retrieves the class identifier (CLSID) of an object.

**IInternalConnection**

Interfaces method include:
- **AddConnection**: Increments the reference count of session by one.
- **ReleaseConnection**: Decrements the reference count of session by one.

**ISupportErrorInfo** (Also implemented in Session, Rowset and Command objects)

Interfaces method include:
- **InterfaceSupportsErrorInfo**: Indicates whether an OLE DB interface can return OLE DB Error objects.
Interfaces on the Session object

IGetDataSource

Interfaces method include:
GetDataSource: Returns an interface pointer to the Data Source object.

IOpenRowset

Interfaces method include:
OpenRowset: Opens and returns a Rowset.

ISessionProperties

Interfaces method include:
GetProperties: Returns the list of Properties and their values supported by Session.
SetProperties: Sets Properties belong to Session.

IDBCreateCommand

Interfaces method include:
CreateCommand: Creates a new command.

IDBSchemaRowset

Interfaces method include:
GetRowset: Returns a Schema Rowset.
GetSchemas: Returns all Schema Rowsets implemented by the Provider.

IObjectWithSiteSession

Interfaces method include:
SetSite: Provides the site’s pointer to the object being managed.
GetSite: Retrieves the last site set.

Interfaces on the Rowset object

IAccessor (also implemented in Command object)

Interfaces method include:
AddRefAccessor: Adds a reference count to an existing accessor.
CreateAccessor: Creates an accessor from a set of bindings.
**GetBindings:** Returns the bindings in an accessor.

**ReleaseAccessor:** Releases an accessor.

**IConvertType** (Also implemented in Command object)

Interfaces method include:

**CanConvert:** Gives information about the availability of type conversions on a command or a rowset.

**IColumnsInfo** (Also implemented in Command object)

Interfaces method include:

**GetColumnInfo:** Returns the column metadata needed by consumers.

**MapColumnIDs:** Returns an array of ordinals of the columns in a rowset that are identified by column IDs.

**IRowset**

Interfaces method include:

**AddRefRows:** Adds a reference count to an existing row handle.

**GetData:** Retrieves data from the rowset.

**GetNextRows:** Fetches the next rows in the sequence.

**ReleaseRows:** Releases rows.

**RestartPosition:** Resets the next fetch position to the beginning of the Rowset.

**IRowsetIdentity**

Interfaces method include:

**IsSameRow:** Compares two row handles to see if they are pointing to the same row.

**IRowsetInfo**

Interfaces method include:

**GetProperties:** Returns properties and values supported by the Rowset object.

**GetReferenceRowset:** Returns a pointer to the rowset that a bookmark applies.

**GetSpecification:** Returns specification on the object that created the rowset.

**Interfaces on the Command object**

**ICommand**

Interfaces method include:

**Cancel:** Cancels command execution.

**Execute:** Executes the command.
GetDBSession: Returns a pointer to the Session that creates the Command object.

ICommandProperties

Interfaces method include:
GetProperties: Returns the list of properties in the Rowset property group that are requested for the rowset.
SetProperties: Sets properties in the Rowset property group.

ICommandText

Interfaces method include:
GetCommandText: Returns the text command just set.
SetCommandText: Sets the command text.

Interfaces on the Error object

IErrorInfo

Interfaces method include:
- GetDescription: Returns a text description of the error.
- GetGUID: Returns the GUID of the interface that defined the error.
- GetHelpContext: Returns the Help context ID for the error.
- GetHelpFile: Returns the path of the Help file that describe the error.
- GetSource: Returns the name of the component that generates the error.

IErrorRecords

Interfaces method include:
- AddErrorRecord: Adds a record to an OLE DB Error object.
- GetBasicErrorInfo: Returns basic information about the error, such as the return code and provider-specific error number.
- GetCustomErrorObject: Returns a pointer to the custom error object.
- GetErrorInfo: Returns a pointer to an IErrorInfo object.
- GetErrorParameters: Returns the error parameters.
- GetRecordCount: Returns the number of record in the Error object.
Appendix B OLE DB Properties in the Cameleon Provider

This appendix lists all the OLE DB properties and their values that apply to the Cameleon Provider. Properties are grouped by their Property Group.

### Properties in Data Source

<table>
<thead>
<tr>
<th>Property Group</th>
<th>Property Set</th>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>DBPROPSET_DBINIT</td>
<td>DBPROP_INIT_DATASOURCE</td>
<td>For users to set during initialization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_HWND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_LCID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_LOCATION</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_MODE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_PROMPT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_INIT_PROVIDER-STRING</td>
<td></td>
</tr>
<tr>
<td>Data Source</td>
<td>DBPROPSET_DATASOURCEINFO</td>
<td>DBPROP_ACTIVESESSIONS</td>
<td>0 (unlimited)</td>
</tr>
<tr>
<td>Information</td>
<td></td>
<td>DBPROP_DATASOURCE-READONLY</td>
<td>VARIANT_TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_THREADMODEL</td>
<td>OLEDBCamProv.dll</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_PROVIDER-Filename</td>
<td></td>
</tr>
</tbody>
</table>

### Properties in Session

<table>
<thead>
<tr>
<th>Property Group</th>
<th>Property Set</th>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>DBPROPSET_SESSION</td>
<td>DBPROP_SESS_AUTOCOMMITISOLEVELS</td>
<td>DBPROPCAL_TL_ISOLATED</td>
</tr>
</tbody>
</table>

### Properties in Rowset

<table>
<thead>
<tr>
<th>Property Group</th>
<th>Property Set</th>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rowset</td>
<td>DBPROPSET_ROWSET</td>
<td>DBPROP_IACCESSOR</td>
<td>VARIANT_TRUE for all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_ICOLUMNSINFO</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBPROP_ICONVERTTYPE</td>
<td></td>
</tr>
<tr>
<td>DBPROP_IROWSET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBPROP_IROWSETIDENTITY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBPROP_IROWSETINFO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBPROP_ISUPPORTERROR-INFO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C Schema Rowsets of the Cameleon Provider

This appendix lists the five Schema Rowsets and their restriction column values in the Cameleon Provider. Refer to OLE DB version 2.5 specification [9] for details of all OLE DB-defined properties.

**CATALOGS Rowset**

<table>
<thead>
<tr>
<th>CATALOG_NAME (DBTYPEWSTR)</th>
<th>DESCRIPTION (DBTYPEWSTR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cameleon</td>
<td>Catalog for Cameleon Registry information</td>
</tr>
</tbody>
</table>

**SCHEMATA Rowset**

<table>
<thead>
<tr>
<th>CATALOG_NAME (DBTYPEWSTR)</th>
<th>SCHEMA_NAME (DBTYPEWSTR)</th>
<th>SCHEMA_OWNER (DBTYPEWSTR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>COIN</td>
</tr>
</tbody>
</table>

**TABLES Rowset**

<table>
<thead>
<tr>
<th>TABLE_CATALOG (DBTYPEWSTR)</th>
<th>TABLE_SCHEMA (DBTYPEWSTR)</th>
<th>TABLE_NAME (DBTYPEWSTR)</th>
<th>TABLE_TYPE (DBTYPEWSTR)</th>
<th>DESCRIPTION (DBTYPEWSTR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>TABLE</td>
<td>E*Trade</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>BONDS</td>
<td>TABLE</td>
<td>bondsonline</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>MONEYRATES</td>
<td>TABLE</td>
<td>moneyrates</td>
</tr>
</tbody>
</table>

All rows of columns TABLE_GUID (DBTYPE_GUID), TABLE_PROPID (DBTYPE_U14), DATE_CREATED (DBTYPE_DATE) and DATE_MODIFIED (DBTYPE_DATE) have value of DBTYPE_NULL.
## COLUMNS Rowset

<table>
<thead>
<tr>
<th>TABLE_CATALOG (DBTYPEWSTR)</th>
<th>TABLE_SCHEMA (DBTYPEWSTR)</th>
<th>TABLE_NAME (DBTYPEWSTR)</th>
<th>COLUMN_NAME (DBTYPEWSTR)</th>
<th>ORDINAL_POSITION (DBTYPE_UI4)</th>
<th>CHARACTER_MAXIMUM_LENGTH (DBTYPE_UI4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>Date</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>Transaction</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>Security</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>Quantity</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>ETRADE</td>
<td>Price</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>BONDS</td>
<td>DatedDate</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>BONDS</td>
<td>FirstCoupon</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>BONDS</td>
<td>PayFrequency</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>BONDS</td>
<td>SettlementDate</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>MONEYRATES</td>
<td>Bankname</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>MONEYRATES</td>
<td>Rate</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>MONEYRATES</td>
<td>Yield</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Cameleon</td>
<td>COIN</td>
<td>MONEYRATES</td>
<td>Minbalance</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Other columns:

**DATA_TYPE (DBTYPE_UI2):** All rows of this column have value DBTYPEWSTR.

**CHARACTER_OCTET_LENGTH (DBTYPE_UI4):** All rows of this column have value 1.

**COLUMN_FLAGS (DBTYPE_UI4):** All rows of this column have values DBCOLUMNFLAGS.IsNullABLE, DBCOLUMNFLAGS_MAYBENULL and DBCOLUMNFLAGS_MAYDEFER.

**COLUMN_HASDEFAULT (DBTYPE_BOOL):** All rows of this column have values of VARIANT_FALSE.

**IS_NULLABLE (DBTYPE_BOOL):** All rows of this column have values of VARIANT_TRUE.

The rest of the columns all have DBTYPE_NULL values, they include:
The Provider Types rowset identifies the data types supported by the data provider. The Cameleon Provider only supports one data type, DBTYPE_WSTR, wide character string.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type Indicator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type_Name</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>DBTYPE_UI2</td>
<td>DBTYPE_WSTR</td>
</tr>
<tr>
<td>COLUMN_SIZE</td>
<td>DBTYPE_UI4</td>
<td>4000</td>
</tr>
<tr>
<td>LITERAL_PREFIX</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>LITERAL_SUFFIX</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>CREATE_PARAMS</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>IS_NULLABLE</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_TRUE</td>
</tr>
<tr>
<td>CASE_SENSITIVE</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
<tr>
<td>SEARCHABLE</td>
<td>DBTYPE_UI4</td>
<td>DB_UNSEARCHABLE</td>
</tr>
<tr>
<td>UNSIGNED_ATTRIBUTE</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
<tr>
<td>FIXED_PREC_SCALE</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
<tr>
<td>AUTO_UNIQUE_VALUE</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
<tr>
<td>LOCAL&gt;Type_NAME</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>MINIMUM_SCALE</td>
<td>DBTYPE_I2</td>
<td>NULL</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>MAXIMUM_SCALE</td>
<td>DBTYPE_I2</td>
<td>NULL</td>
</tr>
<tr>
<td>GUID</td>
<td>DBTYPE_GUID</td>
<td>GUID_NULL</td>
</tr>
<tr>
<td>TYPELIB</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>VERSION</td>
<td>DBTYPE_WSTR</td>
<td>NULL</td>
</tr>
<tr>
<td>IS_LONG</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_TRUE</td>
</tr>
<tr>
<td>BEST_MATCH</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
<tr>
<td>IS_FIXEDLENGTH</td>
<td>DBTYPE_BOOL</td>
<td>VARIANT_FALSE</td>
</tr>
</tbody>
</table>
Appendix D Using Visual C++ 6.0 ATL Wizard to generate OLE DB Provider

Microsoft Visual C++ 6.0 provides a wizard that enables developers to easily create an OLE DB Provider. The OLE DB Wizard is designed to generate an OLE DB directory structure provider that one can then change to suit his own needs. To access the wizard, perform the following five steps:

1. Create a new ATL COM project by clicking File->New and clicking the project tab. In Figure D-1, a new ATL COM project is created called SampleProvider. Click OK, then accept all the defaults, and click Finish. A Provider in DLL form is generated.

2. The ATL Wizard creates the infrastructure code necessary to add other ATL objects. Next, click Insert->New ATL object to add new object to the provider.

3. The ATL Object Wizard will show up. Choose the Data Access category, and click Provider, as shown in Figure D-2. Then click Next.

Figure D-1: Create new OLE DB Provider using ATL COM project.
4. The ATL Object Wizard Properties window will show up. Type in the name of the OLE DB Provider in the Short Name textbox. In Figure D-3, I typed SamProv to indicate that any consumer should choose SamProv as their OLE DB provider if they want to use the provider I am generating. The rest of the textboxes are filled in automatically.
Note that the name appears in ProgID field and the Version number are very important. Microsoft SQL Server uses this name to perform distributed transaction through the OLE DB Provider.
Appendix E Documentation on XMLDOMWrapper Class

This appendix lists the member functions in the XMLDOMWrapper class. The XMLDOMWrapper class provides an object implementation of the XMLDOM interfaces of the MSXML package that is tailor made to the need of the Cameleon Provider.

**XMLDOMWrapper();**
-- Constructor of the XMLDOMWrapper class.

**virtual ~XMLDOMWrapper();**
-- Destructor of the XMLDOMWrapper class.

**XMLDOMWrapper::StateError(const char* err);**
-- Function to display error messages.

**HRESULT XMLDOMWrapper::LoadXMLDoc(CComBSTR pURL);**
-- Takes an input URL in CComBSTR format. The pURL should specify the file path or the URL where an XML document can be retrieved. Succeeds when the XML document is successfully loaded.

**HRESULT XMLDOMWrapper::GetDocElement();**
-- Parses out the Document Element of the underlying XML Document and stores it in a member variable.

**HRESULT XMLDOMWrapper::GetNodeListbyTag(BSTR tagname, MSXML::IXMLDOMNodeList **pResultNodeList);**
-- Takes an input tagname and retrieves all the nodes in the XML Document with their node names equal to the tagname, and returns the pointer to the resulting Node list.

**BSTR XMLDOMWrapper::GetAttributebyTag(BSTR tagname, MSXML::IXMLDOMNamedNodeMap *pMap);**
-- Takes a tagname and a NamedNodeMap as input, and return a BSTR value of the attribute within the NamedNodeMap specified by attribute name equals to tagname.

**long XMLDOMWrapper::GetCatalogList();**
-- For parsing Schema information. This function retrieves the list of all Catalog nodes, stores them in member variable, and returns the length of the node list.
HRESULT XMLDOMWrapper::EnumerateCatalog(int index, Catalog* pCat);
--This function enumerates the name of the Catalog in the list of Catalog nodes at the
position given by the input index. The result will be stored in a Catalog structure.

long XMLDOMWrapper::GetSchemaList();
--For parsing Schema information. This function retrieves the list of all Schema nodes,
stores them in member variable, and returns the length of the node list.

HRESULT XMLDOMWrapper::EnumerateSchema(int index, Schema* pSch);
--This function enumerates the name of the Schema in the list of Schema nodes at the
position given by the input index. The result will be stored in a Schema structure.

long XMLDOMWrapper::GetTableList();
--For parsing Schema information. This function retrieves the list of all Table nodes,
stores them in member variable, and returns the length of the node list.

HRESULT XMLDOMWrapper::EnumerateTable(int index, Table* pTab);
--This function enumerates the name of the Table in the list of Table nodes at the
position given by the input index. The result will be stored in a Table structure.

long XMLDOMWrapper::GetColumnList();
--For parsing Schema information. This function retrieves the list of all Column nodes,
stores them in member variable, and returns the length of the node list.

HRESULT XMLDOMWrapper::EnumerateColumn(int index, Column* pCol);
--This function enumerates the name of the Column in the list of Column nodes at the
position given by the input index. The result will be stored in a Column structure.

long XMLDOMWrapper::GetResultColList();
--For parsing Cameleon result data. This function retrieves the list of all result column
nodes, stores them in member variable, and returns the length of the node list.

HRESULT XMLDOMWrapper::EnumerateResultColumns(int index, BSTR* name);
--This function enumerates the column name of the result column in the list of result
column nodes at the position given by the input index. The result will be stored and
returned through the name variable.
HRESULT XMLDOMWrapper::GetResultColValue(int index, BSTR* value);
-- This function retrieves the result column data values in the list of result column nodes
at the position given by the input index. The result will be stored and returned through
the value variable.

int XMLDOMWrapper::EnumerateResultColumnsSize(int index);
-- This function scan through all the data in a result column specified by column index
and return the maximum data size, in number of bytes, of that column.

int XMLDOMWrapper::GetRowNum();
-- This function returns the number of rows in the Cameleon Result XML document.
Appendix F Setting up OLE DB Provider

This appendix discusses the requirements and procedures on setting up OLE DB Provider.

F.1 Software Requirement

The whole development and testing process of the Cameleon Provider is carried out on Windows platform with the Microsoft Windows 2000 operating system with Service Pack 1 installed. Testing has also been carried out on Windows NT and Windows 98 operating systems and produces the same behavior. Other software requirements include the following:

- **MDAC 2.5**: Microsoft Data Access Components version 2.5.
- **Microsoft Platform SDK**: The Microsoft Platform Software Development Kit. This provides all the OLE DB header files for development, documentations on OLE DB Provider development and debugging tools including Trace Provider, and the Rowset Viewer.
- **Microsoft Visual C++ 6.0**: This is the development environment used to implement the Cameleon Provider. It includes the ATL and OLE DB Provider Wizard. Note that one of the library file in ATL, ATLDB.h, has to be replaced by ATLDBFIX.h for the generated OLE DB Provider to behave correctly. Refer to Microsoft knowledge base article Q198520 [5] to locate the file and perform other required changes to the ATL.
- **Microsoft SQL Server 7.0**: This version has the linked server and distributed query features that are essential in incorporating OLE DB data sources to Microsoft applications. Service pack 2 for SQL Server 7.0 is also required.
- **Microsoft Office 2000**: Previous versions of Office applications are not OLE DB compliant.

F.2 Installing the OLE DB Provider

Appendix D discusses how to build an OLE DB Provider using Visual C++ 6.0 Provider Wizard. In order to allow OLE DB consumers to locate and access an OLE DB Provider,
one has to register the DLL file of the OLE DB Provider on the server machine. Even though Microsoft articles promise that Visual C++ generated OLE DB Providers will perform self-registration once they are compiled, it turns out that manual registration is required for once. After the Provider DLL file is registered manually for the first time, upon recompilation, the Provider will perform self-registration automatically. To register the Provider DLL file, the `regsvr32` command is used in the following way:

In command prompt, run `regsvr32 filename and path of the provider DLL`.

To check whether the OLE DB Provider has been successfully register, follow the procedures listed below:

1) Go to `Windows Explorer` (or `My Computer` on Win2000) and click on `File->New->Microsoft Data Link`. A Microsoft Data Link UDL file will be created.

2) Right click on the UDL file and choose `Properties`. The Data Link Properties window will pop up.

3) Choose the `Provider` tab and a list of OLE DB Provider available on the machine will be shown.

4) If the name of the Provider just registered is on the list, then registration is successful.

Note that the Data Link Properties feature requires the proper installation of MDAC 2.5 on the operating system. Also, the DLL files of all the OLE DB Providers from Microsoft is stored in the following directory:

`<Drive where operating system is installed>:\Program Files\Common Files\System\Ole DB`

With the proper installing of the OLE DB Provider, all the applications that are able to interact with OLE DB Providers directly should be able to access the Provider.
Appendix G User Manual on using Cameleon Provider with SQL Server 7.0

G.1 Using ad hoc distributed queries in SQL Server Query Analyzer

The OpenRowset function is used to perform ad hoc distributed queries to access OLE DB data sources. The syntax of OpenRowset function is as follow:

```
OPENROWSET('provider_name'
   {'datasource';'user_id';'password'
    | 'provider_string'
   },
   {[catalog.] [schema.] object
    | 'query'
   })
```

Users specify the name of the provider, all the necessary connection information and the object or command they want to access in the OpenRowset function. Note that the provider name is not equal to the DLL file name of the provider. To find out the names of all the available OLE DB Providers, one can use the `xp_enum_oledb_providers` function in Query Analyzer as shown in figure G.1-1.

![Figure G.1-1 Execution of the xp_enum_oledb_providers function.](image)
Figure G.1-2 Execution of ad hoc distributed query in Query Analyzer.

Other than the OpenRowset function, one can also use the OpenQuery function to achieve the same result. However, OpenQuery requires the set up of a linked server with all the connection properties specified for the OLE DB Provider. The syntax of the OpenQuery function is as follow:

```
OPENQUERY(inked_server, 'query')
```

Figure G-1.3 Execution of OpenQuery function in Query Analyzer.
The use of OpenQuery function is important when other clients, such as Microsoft Access on another machine, want to access OLE DB data through SQL Server but do not have a copy of the Provider installed on the client machine. Client applications without a copy of the OLE DB Provider installed on the machine can only access OLE DB data source views defined with OpenQuery function with linked server names specified through SQL Server. If an client application tries to access view defined by OpenRowset function through SQL Server, an error statement “could not instantiate OLE DB Provider” is returned since OpenRowset function tries to locate the Provider on client machine with the name supplied. On the other hand, if access is done through view defined by OpenQuery, the request will be processed by the linked server residing on the SQL Server machine, where the OLE DB Provider is hosted. Therefore, using the OpenQuery function enables remote access to OLE DB data from client machines without installing the OLE DB Provider on client machine. Note also that when executing remote access, the Distributed Transaction Coordinator of SQL Server 7.0 has to be turned on.

G.2 Adding Linked Server

There are two ways to add a linked server on SQL Server 7.0. The first method is to use the SQL Server Enterprise Manager UI and the second method is to use stored procedure sp_addlinekdserver.

To make use of the SQL Server Enterprise Manager UI to add linked server, one can follow the steps listed below:

1) Go to Enterprise Manager and select the SQL Server where the OLE DB Provider should be hosted. Note that server machine has to have a local registered copy of the provider’s DLL.

2) Under the targeted SQL Server, click on Security. Within the Security folder, right click on Linked Servers->New Linked Server. The Linked Server Properties window will pop up.

3) Supply a name to the Linked Server and choose Other data source. From the pull down manual, select the appropriate OLE DB Provider. Then fill in the rest of the initialization properties if needed. Click OK.
To make use of the stored procedure `sp_addlinkedserver`, one has to execute the `sp_addlinkedserver` stored procedure in SQL Server Query Analyzer and supply initialization information in the following manner.

```sql
sp_addlinkedserver [@server =] 'server', [@srvproduct =] 'product_name'
[, [ @provider =] 'provider_name' ] [, [ @datasrc =] 'data_source' ]
[, [ @location =] 'location' ] [, [ @provstr =] 'provider_string' ]
[, [ @catalog =] 'catalog' ]
```

**G.3 Creating View and using Access to retrieve OLE DB data sources**

One can use a View to hide the underlying distributed queries such that a regular Transact SQL queries can be used to access OLE DB data source. To create a View to encapsulate a distributed query, one can execute the following command in Query Analyzer:

```sql
CREATE VIEW viewname (list of all the column names) AS SELECT * FROM openquery
(linked server name, command)
```

It is important to use the `OpenQuery` function instead of the `OpenRowset` function to create the View such that client machines can access the OLE DB data sources as explained in the previous section. Note that the view would be created in the current database indicated by Query Analyzer.

With the creation of a View, one can use other OLE DB compliant applications to access the OLE DB data source. An example on using Access to retrieve Cameleon data will be shown here. In the following example, a View called `moneynrates` is created to access the `moneynrates` Cameleon web source is defined under the database `CamProvDemo` of SQL Server Arbouse.mit.edu.

1) In Access, click on **File->New**. Select the **General** tab and choose **Data Access Page**. Click on **OK**.
2) In the **New Data Access Page** window, select **Design View** and click **OK**.
3) The **Data Link Properties** window will pop up. Under **Provider** tab, choose the OLE DB Provider for SQL Server. Under **Connection**, enter the SQL Server
name and log in information where the view is defined. Select the database where
the view is defined. Figure F.3-1 shows the Data Link Properties window.
Click OK.

![Data Link Properties window](image)

**Figure G.3-1 Data Link Properties window in Access.**

4) In the Field List window, select Views and a list of all the available views will be
shown. Click on the plus symbol of the targeted view and all the available
columns defined in the view will be shown. Use the Add to Page button to add
all the column you want to display. After you are done, go to View->Page View
and the Cameleon data would be displayed. One can use the navigation bar at the
bottom of the page to go through the data. Figure F-3.2 shows a screen shot of
the Access data page.
There are other ways in Access to access OLE DB data source through SQL Server, but only one method is shown here. Refer to Microsoft Access manual on other means to access SQL Server data from Access.
References


URL: http://www.microsoft.com/data/oledb/default.htm

URL: http://www.microsoft.com/com/

URL: http://www.object-tools.com/manuals/lib/dale/ch_04.html

URL: http://www.w3.org/XML/