IBM VisualAge C++ Professional
for AIX®

Data Access Builder Concepts and Tasks

Version 5.0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

Second Edition (January 1999)

This edition applies to Version 4.0 of IBM VisualAge C++ Professional and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term ″VisualAge″ as used in this publication, refers to the VisualAge C++ product set.

If you have comments about this document, address them to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
Toronto, Ontario, Canada, M3C 1H7
FAX: (416) 448-6161
torrcf@ca.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program’s tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

Chapter 1. Data Access Builder

Data Access Builder Workflow

OS/2 WIN Windows and OS/2 Workflow

1. **Creating a New Data Access Part**
   Data Access applications use C++ code to access a database. To create a Data Access application, first create a Data Access target in the Integrated Development Environment (IDE) (see Create a Project and Add or Change a Target).

   Then create a new Data Access part in that target. This launches the Data Access Builder SmartGuide, where Data Access Builder creates a *mapping* from a database *schema* (a database table, view, or an SQL query). The mapping contains sufficient information for Data Access Builder to generate C++ classes. For example, it has definitions for all attributes matching all the columns in the database table, view, or SQL query.

   When you finish creating the mapping, the new Data Access part is added to the Data Access Launch view in the IDE. The part consists of the following files:

   - .dax file: The Data Access session file, which contains all the information about the mapping.
   - .vbe and .vbf files: Parts that correspond to Data Access classes that can access the database.
   - .icc file: The build configuration file that lists the .dax, .vbe, and .vbf files.

   Note: Only one Data Access part can be created for a target. This means that only one Data Access session (a .dax file) can be associated with a target. However, you may include in that session several schemas and mappings.
2. **Editing a Data Access Part**
   Once the part is created, customize the mapping to suit your application's needs. For example, you can change attribute names or add your own methods. To edit the new part, select it and click the **Edit Part** button. This will launch the Data Access Builder main window. See Customize the Mapping for information on the many ways a mapping can be customized.

   You can also add schemas and mappings to the session. To add a schema, select **Map Schema** from the **File** menu. To add a mapping, select a schema and then select **Create Mapping** from the schema's popup menu.

Save the session information to the IDE. Corresponding .vbe and .vbf files will be added to the Data Access part in the IDE.

Then generate the code for the mapping (see Generate Source Code). If you chose the ESQL access method, an .sqx file will be generated. Otherwise, if you chose CLI or ODBC, a .cpp file and a .hpp file will be generated.

When you have finished customizing the mapping, close Data Access Builder and return to the IDE.

3. **Create the Application that Uses the Generated Classes**
   Create other visual or nonvisual parts by using the IDE and Visual Builder to create a program that uses the generated data access classes to access and manipulate database tables. See the Data Access Builder Samples or the Data Access / Visual Builder Tutorial for examples that show how to do this.
4. **Building the Part into the Target**
   Now you can Build the project (see Build). This will generate source code, if needed, translate .sqx files into .cxx files, and compile and link all code in the part. The build process uses the default build options for the target type. You can customize these options if you want (see Set Build Options).

AIX Workflow

1. **Launching Data Access Builder and Creating the Mapping**
   Start Data Access Builder by running `cppdata` from the working directory command line. In the startup window, select **Create New Mapping** or **Work with Existing Mapping**. Create and customize the mapping as needed.
2. **Generating Source Code**
   The generated files will be stored in the working directory.
3. **Create the Application that Uses the Generated Classes**
   Create other visual or nonvisual parts by using the IDE and Visual Builder to create a program that uses the generated data access classes to access and manipulate database tables. See the Data Access Builder Samples or the Data Access / Visual Builder Tutorial for examples that show how to do this.
4. **Importing Classes Into the IDE and Building the Project**
   In the IDE, create a project and target for your application. Edit the project's configuration file to list all the .cpp source files for the data access classes, the visual parts, and the non visual parts that make up the program (see Add Part Source to the Configuration File). Then build the project (see Build)

Data Access/Visual Builder Tutorial

Creating a Visual Data Access Application- Follow the instructions in this tutorial to create a basic data access application visually.

User-Defined Methods

By default, Data Access Builder generates methods that you can use in a C++ application to access your database. However, you may want to add your own methods (user-defined methods) to the generated classes. There are several ways for you to add these methods.

Note: These methods will be generated as static methods if they do not access any of the class’ attributes.

The three types of user-defined methods are:

- SQL Statement methods - methods that query the schema and return an object.
- SQL Predicate methods - methods that query the schema and return a collection of rows.
- Stored Procedure call methods - methods that call a stored procedure in the database and return whatever the stored procedure returns.

The following sections describe each of these types of user-defined methods in detail.

Customized SQL Statement

Suppose you want to add a method to find the average salary of all the employees. Data Access Builder lets you add this method to the mapping by selecting **Methods** from the mapping icon’s popup menu. When you are adding the method, you will be asked to enter an SQL statement to define what the method does, for example:

```
SELECT AVG(SALARY) FROM EMPLOYEE
```

You then give a name to the method, for example, findAvgSalary, and describe the returned value (for example, its data type is float). When the code is generated, findAvgSalary is coded as a method in the Employee class.

This process of entering an SQL statement to specify the user-defined method is called *User-Defined Methods with Customized SQL Statement*. You will see this terminology used in the Data Access Builder user interface.

SQL Predicate

The Data Access Builder, by default, generates a select method in the <*class*>Manager class. The `select` method lets you retrieve a collection of rows from the database into memory, based on an SQL predicate passed in at run time. For example, if you want to work with all the department E11 employees from a table mapped to the object Employee, you can call the following method:

```
EmployeeManager theEmpMgr;
theEmpMgr.select("WORKDEPT = 'E11'");  //"WHERE" is automatically
                                       //appended for select statements
```

If you know you are going to run this select method for department E11 frequently, you can add auser-defined method called DeptE11 which contains the SQL predicate, `WHERE WORKDEPT = ’E11’`. As a result, two methods, selectDeptE11 and openDeptE11, will be added to EmployeeManager class when code is generated. If you have specified a data ID for the table, two similar methods are also added to EmployeeDataIdManager class. Because the SQL predicate always returns a result set, this type method is only generated into the Manager classes (for example, EmployeeManager class and EmployeeDataIdManager class). Data Access Builder lets you add this method to the mapping by selecting Manager Methods from the mapping icon’s popup menu. When you are adding the method, you will be asked to enter an SQL predicate, for example:

```
WHERE WORKDEPT = 'E11'
```

You give a name to this method, for example, DeptE11, and selectDeptE11, and openDeptE11 will be generated in EmployeeManager class and EmployeeDataIdManager class (if it exists). You can then use the methods from the generated class as follows:

```
theEmpMgr.selectDeptE11(const char* theSqlSuffix);
```

The SQL suffix parameter is an optional additional SQL statement that may be passed in at run time. For example, you may want to pass in ”ORDER BY LASTNAME“ to specify how the returned result set will be ordered.

You can also define methods that have variable values for parameters that are specified at run time. For example the SQL predicate for a method called AnyDept may be entered as:

```
WORKDEPT = ?
```

In this case the following methods are generated:

```
selectDeptE11(const IString& deptno, const char* theSqlSuffix=0)
openDeptE11(const IString& deptno, const char* theSqlSuffix=0)
```

This process of entering an SQL predicate to specify your method is called *User-Defined Methods with SQL Predicate*. You will see this terminology used in the Data Access Builder user interface.

Stored Procedure Call

You can also add a method that calls a database stored procedure. Stored procedures are procedures that operate on the database and are stored on the server of a remote database. When the stored procedure call method returns a single row or if it is a function call, you would add it to the <*class*> class. For example, if the stored procedure returns a single database row for the employee with the longest length of service, this stored procedure call method should be added to Employee persistent object class. Data Access Builder lets you add the method to the mapping by selecting **Methods** from the mapping icon's popup menu. When code is generated, your method is in Employee persistent object class.

However, if the stored procedure call returns a result set and you want to handle all the rows in the result set, you would add the method to the <*class*>Manager class. For example, if the stored procedure returns all employees eligible for retirement, you would add the method that calls it to EmployeeManager class. If you have specified a data ID, the method is also added to EmployeeDataIdManager class. Data Access Builder lets you add this method to the mapping by selecting **Manager Methods** from the mapping icon's popup menu. Two methods are generated, open<*methodname*> and select<*methodname*>, in both the EmployeeManager class and (if it exists) the EmployeeDataIdManager class.

Before you add the stored procedure call method, Data Access Builder must be told what the definition of the stored procedure is. Data Access Builder can connect to the database and retrieve definitions of its registered stored procedures. If a database connection is not available or if a stored procedure is not registered, you can define the stored procedure and its parameters to Data Access Builder manually. Of course in this situation, you are responsible for the correctness of the defined stored procedure. Once the stored procedure signature becomes known to Data Access Builder, you enter a name for the stored procedure call method, for example, FindRetireeCandidates, and map its parameters.

These methods that you have created are referred to as *Stored Procedure Call User-Defined Methods*. You will see this terminology used in the Data Access Builder user interface.

User-Defined Methods in Generated Classes

In summary, you can add user-defined methods in the generated classes as follows:

Class	**User-Defined Methods that You can Add**
<*class*>	Customized SQL Statement Stored Procedure Call (returning a row or acting as a function call)
<*class*>DataId	Not Applicable

Class	User-Defined Methods that You can Add
<*class*>Manager Two methods are generated: select<*methodname*> and open<*methodname*>	Stored Procedure Call (returning a result set) SQL Predicate
<*class*>DataIdManager Two methods are generated: select<*methodname*> and open<*methodname*>	Stored Procedure Call (returning a result set) SQL Predicate
Other classes	Not applicable

Keyless Table Support and Data Identifiers

A ”keyless“ table is one that does not have a column (or group of columns) that is specified within the table as uniquely identifying each row. Normally, Data Access Builder maps a table’s primary keys to attributes called *data identifiers* (data IDs). If the table does not have any primary keys, you can modify one or more attributes to form the data identifier. **Note**: You should not map non-unique columns to be data identifiers. If you do, the retrieve method will retrieve only the first record with a specified data identifier, and the update and delete methods will affect all rows with the specified data identifier (an exception may be thrown if this happens).

For example, your table contains a list of employees with columns for job, department, phone number, and employee number, where the employee number column is a primary key. By default, Data Access Builder maps ”employee number“ as the data identifier and it generates code that lets you retrieve, delete, or update specific rows when you specify ”employee number“.

If the table contained only ”department“, ”job“, and ”last name“, it is conceivable that you would have more than one employee with a particular last name in a department. In this example, there is no key for the table, and so the Data Access Builder would not generate methods such as delete, update, or retrieve. However, it does generate the add method to add a row to the database and the updateFetched and deleteFetched methods to allow you to update or delete using a cursor, the last fetched row. For example:

```
// Open the cursor
theDeptMgr.open( "WHERE job = 'ADMINISTRATOR' FOR UPDATE" );

// Fetch the first row that has ADMINISTRATOR as the job
theDeptMgr.fetchNext();

// Change the Job description for cached object
theDeptMgr.element()->setJob("CLERK");
```

```
// Update the last fetched row with the cached object, close the cursor
theDeptMgr.updateFetched();
theDeptMgr.close();
```

For keyless tables, the add method will be generated, since you need not identify one particular row in a table in order to add a row to it. However, if your schema is a table join, no add method is generated. You can use user-defined methods to add your own methods.

Unique Indices
Some tables have *unique indices* defined. Unique indices are sets of columns, other than the primary key, that uniquely identify rows in the table, and are specified as such within the table definition. When creating a mapping for a table, you can use a unique index as an alternate mapping for the data ID (that is, you can change the default data ID mapping from the primary key to a unique index). By mapping the data ID to one of these pre-defined set of columns, you ensure that the data IDs do uniquely identify rows in the table. This prevents your data access application from unintentionally changing or deleting more than one row at a time.

Table Joins and Query Support

You have a table join when you specify more than one table or view in the FROM clause of the SQL statement. You can use Data Access Builder to generate code from schemas that come from table joins and other SQL queries. You do this by specifying an SQL statement for mapping in the Data Access Builder SmartGuide. The following statement is an example of the kind of SQL statement you can use to customize tables used for a schema:

```
SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
FROM PROJECT, EMPLOYEE WHERE DEPTNO=WORKDEPT
```

The Data Access Builder analyzes the combined tables or views, creates a schema mapping, and generates the appropriate methods.

For a table join or a specified SQL query, Data Access Builder does not generate add, update, delete, retrieve, deleteFetched, and updateFetched methods. You can use collections and cursors (for example, the select method) to retrieve rows and work with them. You can also add methods that you define to implement adding, updating, and deleting rows in the database tables involved. In the above example, you can add a method that adds a row for a new project to the project table and another method to add a new employee to the employee table.

Query with Parameters
You can also enter parameters in the SQL statement when creating the schema mapping in the SmartGuide. For example:

```
SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT=?
UNION SELECT EMPNO FROM EMP_ACT WHERE PROJNO=?
```

The parameters are mapped as param1, param2, param3 and so forth. Use the Schema Mapping Description step of the SmartGuide to document these parameters and the information will be added to the generated code. Furthermore, get and set methods are generated for each of these parameters in the <*class*>Manager class so that you can change the values of these parameters before calling the select or open methods for the query. A constructor for <*class*>Manager that includes the parameters is also generated, for example:

```
QueryManager(const IString& aParam1, const IString& aParam2 )
```

Large Object Support

Data Access Builder provides support for mapping DB2 Large Objects (LOBs) to attributes in a C++ class. The class library provides classes and methods for handling the following large object types:

- Character Large Object (CLOB); for example, a magazine article.
- Double-Byte Character Large Object (DBCLOB); for example, a magazine article in Japanese characters.
- Binary Large Object (BLOB); for example a scanned-in photograph, a sound clip or a video clip.

Class Library methods allow you to access the contents of a LOB, and then manipulate the contents in various ways, including inserting, removing, overlaying, concatenating, comparing, and finding the length. You can save a LOB to a file, or read one in from a file.

Data Access LOB support lets you control whether you work with LOBs by downloading them to the client space, or by accessing them on the database server. Since DB2 LOBs can be as large as 2.0 GB, you might not want to work with every LOB by downloading it. However, if your connection to the database server is not very fast, you might want to work on smaller LOBs on the client.

You can choose to work with all LOBs on the server or on the client, or you can let the application decide where to store LOBs based on their sizes. The *threshold* represents the critical LOB size. Objects smaller than the threshold value will be downloaded to the client; those larger than the threshold value will remain on the server and will be accessed from the client by using a database locator.

For example, if you set the threshold value to 5000 bytes, then objects smaller than 5000 bytes will be downloaded to the client for manipulation, while those larger than 5000 bytes will remain on the server. The LOB classes allow you to set the threshold from a data access application.

To work with LOBs on the server, the application uses a datastore connection, which also is set using class library methods. **Warning!Turn auto-commit OFF for the LOB's datastore [that is, call theDatastore.enableAutocommit(false)]. Database locators become invalid after one transaction, and if the auto-commit setting is left on, then a LOB locator will be lost after the first time you do something with the LOB.**

LOBs have another setting called *delta*. The delta is the tolerance for the threshold; it prevents the object being sent back and forth between the client and server if its size is hovering about the threshold. If the object is on the client, the delta is added to the threshold to determine the maximum size it can reach before being sent to the server. If the object is on the server, the delta is subtracted from the threshold to determine the minimum size it can reach before being sent to the client.

For example, suppose you set the class threshold to 500 bytes and the delta value to 25 bytes by running the following code:

```
IDALobDO.setClassThreshold( 500, 25 );
```

If the current LOB is 450 bytes, it will be stored on the client. If you add 60 bytes to it, for a total of 510 bytes, then it will still be kept on the client, even though it has passed the threshold, because of the delta (the upload threshold is effectively 525 bytes). If you then add 30 more bytes, for a total of 540 bytes, the LOB will be moved to the server. If you then remove 50 bytes from it, for a total of 490 bytes, it will still be kept on the server (the download threshold is effectively 475 bytes).

Support for DB2 Extended Scalar Functions for CLOBs

The class for CLOB support, IDAClobDO, contains C++ methods that run the DB2 Extended Scalar Functions for CLOBs, so that you can use these methods directly from a data access application. Support exists for the following functions:

- ASCII
- CONCAT
- SUBSTRING
- INSERT
- REPLACE
- LOCATE
- LCASE
- LTRIM
- RTRIM

- LEFT
- RIGHT
- CHAR (C++ method is called CHAR_OF)
- REPEAT
- SPACE

See the class library reference documentation for more information on all LOB support methods.

Generated Methods

After you have generated the code, you can use the methods in the generated classes to access and manipulate the information in the database. For keyed tables (mappings that have data IDs), the following methods are generated:

- retrieve **-** retrieves a row from the table into a <*class*> object that corresponds to a given DataId object
- add **-** adds a row to the table based on the information in the current <*class*> object
- delete **-** deletes a row in the table corresponding to the current <*class*>DataId object
- update **-** updates a row in the table corresponding to the current <*class*>DataId object; it will also update the DataId column(s) if a DataId object is passed in. The update method is dynamically built at run time to include only the attributes that have changed. Another method, called updateAll, updates all attributes, regardless of whether the have changed.

If your mapping is to a keyless table, by default the mapping will have no data ID, and so retrieve, delete, update, and updateAll will not be generated. The add method will be generated. If your mapping is to a schema that is an SQL statement, then none of these methods will be generated.

Customize or Suppress Code Generation for add, retrieve, or delete Methods

The default add, retrieve, and delete methods are listed in the Methods window (click mapping icon and select **Methods**). The update method is not listed in the window. If you will not be using one or more of these listed methods, you can select the appropriate methods and click **Hide** to suppress their code generation. If you want to tailor these methods, for example exclude some attributes in the add method, you can copy the add method to create a new add1 method and customize it. You can then hide the default add method. In the generated code, you will get the add1 method.

There are many other methods for <*class*> objects and <*class*>DataId objects. They are described in the Data Access Class Library and Generated Code reference. You can also customize the generated classes by defining your own methods to work with the data.

Read-Only Considerations

A persistent object can be read-only, meaning that it cannot be updated with new attribute values. The following methods let you handle read-only considerations (all are PersistentObject methods):

- isReadOnly
- isDefaultReadOnly
- setReadOnly

The read-only status of an object can be checked by calling the isDefaultReadOnly method. If the class was generated from a read-only schema, or the Read Only checkbox is selected in the Mapping Properties notebook, then this method returns true. In this case, update, delete, and add methods will throw exceptions.

If the part is not generated from a read-only view, is DefaultReadOnly returns false. You can update, delete from, or add to the schema table or view by using generated persistent objects.

You can set any given persistent object to be read-only by calling its setReadOnly method and passing in true. Then you cannot change attributes in the object to update the row. You can check the read-only status of an object by calling its isReadOnly method; it will return true if the object is read-only, or false if it is not.

If the default read-only setting is true, then calling setReadOnly with a parameter value of false throws and exception.

Null Value Considerations

When a default persistent object is constructed, all its attributes have null values. You can give any of its attributes values or leave them null. Some attributes, however, are not nullable. Although a non-nullable attribute can be set to null, it cannot be committed to the database as null. If you try to commit to the database an update or add transaction with an object whose non-nullable attribute is null, an exception will be thrown.

Three methods that handle null settings are generated for each of the object’s attributes:

- is<*Attribute*>Nullable
- is<*Attribute*>Null
- set<*Attribute*>ToNull

Call is<*Attribute*>Nullable to see if the specified attribute can be set to null. Note that attributes in the data ID will return false for this method. Attributes mapped to database columns that are specified as non-nullable will also return false. If you call set<*Attribute*>To Null for an attribute that is non-nullable, an exception is thrown.

Call is<*Attribute*>Null to check if an object's attribute is currently set to null. Call set<*Attribute*>ToNull to set the specified attribute to null.

Notifications

Most of the Data Access and generated class methods that are long-running or that access a database, throw notifications upon completion so that listeners know that they are complete. Methods that notify listeners have notification IDs defined for them as public data members in their class.

When a method is running on a background thread, its notification ID tells listeners when the method is complete.

Background Threads

Background thread support allows you to put long-running methods on a background thread so that your application can continue rather than wait for a method to finish. You can set certain methods in the Data Access classes and generated classes to run in *asynchronous* mode, that is, on a background thread. Most methods that can be blocked by the database or that could take a long time to finish can be set as asynchronous, including user-defined methods.

Note: Background thread support is for applications with graphical user interfaces, and will not work for command-line applications.

You always have the option of running all methods *synchronously*; the methods will run one after another in the order called. The next method does not start until the current one is finished. A method is complete when control returns to the application. Methods are set to run in synchronous mode by default.

If you use asynchronous methods, a method is not necessarily complete when control returns to the program, but rather when the background thread reports a method complete event.

Exceptions can be thrown by the foreground thread, before the method is placed on a background thread; this completes the method. Also, they can be thrown by the background thread, in which case the thread will report an exception event, and will complete the method.

Data Access Builder generates code that supports background threads. By default, this is turned off. However, Data Access classes allow you to set and test background thread with the following methods:

- IPersistentObject& setAsynchronous(IBoolean flag = true);
  Sets the methods so that they can be run in the background. By default, setAsynchronous is set to false which means that background thread support is not enabled. Pass in `true` to set the object to run asynchronously, or `false` to set it to be synchronous.
- IBoolean isAsynchronous() const { return _isAsync; }
  Finds out whether an object is set to run methods on the background thread. Returns `true` if the object is set to be asynchronous, or `false` if it is not.
- IBoolean isBusy() const;
  Tests if the object is currently running a method. Returns true if an object is running a threadable method, or false if it is not. Only one thread per object can execute in the background.

To run an object’s methods on the background thread, use code similar to the following example:

```
Employee theEmployee;
theEmployee.setAsynchronous(true);
theEmployee.add();  // the add method will run on a background thread;
```

Data Access Class Library and Generated Code

The Data Access classes enable you to connect to databases, map database tables to objects, and work with the data inside the databases by using object-oriented programming. The base classes in the Data Access Class Library provide generic methods for accessing and manipulating database information, interacting with other builders, handling messages and exceptions, and working with large objects. The generated classes extend the base classes and implement operations tailored for specific situations, as you define them in the main Data Access Builder window. In most cases, you will use the generated classes in your data access applications.

Reference Library Notation

In all documentation for generated code, the notation <*class*> represents the name of the Data Access class for which you are generating code. The notation <*attribute*> represents any one of the class' attributes, as mapped from the database table or view (corresponding to a column from the table), and <*attributeType*> represents that attribute's type.

Also, square brackets enclosing one or more parameters in a method's signature indicate that the parameters are repeated to include all attributes of the object. For example,

```
public <class>([<attributeType> an<Attribute>, ...])
```

Here, the parameter list will be a list of each attribute (and its type) mapped from the table to the object.

When a method name includes <*attribute*>, it means that the generated code will include an implementation of that method for each attribute that the class has. For example, if <*class*> has two attributes, `Attr1` and `Attr2`, and `set`<*attribute*> is one of the generated methods for <*class*>, then two methods, `setAttr1` and `setAttr2` will be generated.

Some generated class members may use one of three different Datastore class types: IDatastore, IDatastoreDB2, or IDatastoreODBC, depending on which access method you chose at mapping time. The notation IDatastore*xxx* indicates that one of these three is used, depending on your mapping choice.

Data Access Class Library Reference

The Data Access Class Library includes the following classes. Click on the name of the class to access class member reference information.

Class Name	**Description**
IDatastoreBase	Base class for other Datastore classes. Parent class of IDatastoreBaseCLI. Manages database connections, providing client connection to a database, disconnection from the database, and the ability to commit and rollback database transactions.
IDatastoreBaseCLI	Parent class of IDatastoreODBC and IDatastoreDB2.
IDatastoreODBC	Parent class for generated Datastore class using ODBC access.
IDatastoreDB2	Parent class for generated Datastore class using DB2 ESQL or DB2 CLI access, and parent class for IDatastore.

Class Name	**Description**
IDatastore	Base class for generated Datastore class using DB2 ESQL access from previous releases of VisualAge C++ Data Access Builder.
IPersistentObject	Provides interfaces for adding, deleting, retrieving, and updating a row from a table.
IPODataId	Base class for objects that uniquely identify rows in tables
IDAManager	Provides the ability to work with a collection of rows from a table, and facilitates movement through a collection of rows via a database cursor.
IDAException	Defines exceptions thrown by the Data Access Builder and its generated code.
IDALobDO	Base class for Data Access Large Object (LOB) support (DB2 only).
IDAClobDO	Class for accessing Character Large Objects (CLOBs) (DB2 only).
IDADBClobDO	Class for accessing Double-Byte Character Large Objects (DBCLOBs) (DB2 only).
IDABlobDO	Class for accessing Character Large Objects (CLOBs) (DB2 only).

Data Access Generated Classes Reference

Data Access Builder generates the following classes. Click on the name of the class to access class member reference information.

Class and an Example	**Description**
<*class*>Datastore e.g. EmployeeDatastore	Inherits from IDatastoreDB2 or IDatastoreODBC, depending on which access method is chosen at mapping time. Objects represent connections to the database.
<*class*> e.g. Employee	Inherits from IPersistentObject. Objects represent rows from the schema. Class contains database access methods, including user-defined methods and stored procedure calls, if any are defined.
<*class*>DataId e.g. EmployeeDataId	Inherits from IPODataId. Objects represent the set of columns that uniquely identify a row. This class is generated only if the mapping specifies a data ID column or columns.

Class and an Example	Description
<*class*>ManagerBase <*class*>ManagerTemplate <*class*>Manager e.g. EmployeeManager	Three classes that inherit from IDAManager. Enable you to select a collection of rows from the table and work with them, or to open a database cursor and access a collection of rows one at a time. Contains user-defined manager methods and stored procedure calls, if any are defined.
<*class*>DataIdManagerBase <*class*>DataIdManagerTemplate <*class*>DataIdManager e.g. EmployeeDataIdManager	Three classes that inherit from IDAManager. Enable you to select a collection of Data IDs from the table and work with them, or to open a database cursor and access a collection of Data IDs one at a time. These classes are generated only if the mapping specifies a data ID column or columns.

Files in Data Access Parts

OS/2 WIN **Files in the Data Access Part**

When you create a new Data Access part, the following files are added to the Data Access target within the IDE:

- .dax file: The Data Access session file, which contains all the information about the mapping.
- .vbe and .vbf files: Parts that correspond to Data Access classes that can access the database.
- .icc file: The build configuration file that lists the .dax, .vbe, and .vbf files for the target.

These are all stored in the IDE's working directory.

The following example shows the contents of a Data Access config file for an executable target called ”CarLot“, with a Data Access session called ”CarLot1“, and a mapping called ”Car“. The working directory is F:\project.

```
target "F:\\project\\CarLot.exe"
  {
    source "F:\\project\\CarLot1.dax"
    source "F:\\project\\CarDatastore.vbf"
    source "F:\\project\\Car.vbf"
    source "F:\\project\\CarDataId.vbf"
    source "F:\\project\\CarManagerTemplate.vbf"
    source "F:\\project\\CarManagerBase.vbf"
    source "F:\\project\\CarManager.vbf"
    source "F:\\project\\CarDataIdManagerTemplate.vbf"
    source "F:\\project\\CarDataIdManagerBase.vbf"
    source "F:\\project\\CarDataIdManager.vbf"
    source "F:\\project\\CarForm.vbf"
  }
```

Generated Code Files
If you generate code for the Data Access mapping, or if you build the parts into the target, then the C++ classes that correspond to the new part will be generated. the file names are based on the mapping name(s).

OS/2 WIN Data Access Builder generates files that are FAT-, HPFS- and NTFS-compatible that contain the C++ classes for each mapping in the part;

If you chose ESQL as the database access method, then a .sqx file is generated; otherwise, a .hpp and a .cpp file are generated. These files are also stored in the IDE's working directory (in the Data Access Builder working directory for AIX).

The generated files contain the implementation of the generated classes. Generated classes are named after the mapping. For example, if you map to an object called `Employee`, then all the generated class names will start with '`Employee`'. Before you generate the classes, you can change the base name of the generated classes by opening the Properties notebook (select **Properties** from the mapping icon's popup menu) and typing in a new name.

For the above example, where the mapping is called Employee, the following names would be assigned to the generated classes:

Employee (the persistent object class)
EmployeeDatastore (the datastore connection class)
EmployeeDataId (the data ID class, generated if there is a set of columns that represents a data ID)
EmployeeManagerBase
EmployeeManagerTemplate
EmployeeManager (the class for managing Employee objects)
EmployeeDataIdManagerBase
EmployeeDataIdManagerTemplate
EmployeeDataIdManager (the class for managing EmployeeDataId objects, generated if there is a set of columns that represents a data ID)

Data Access/Visual Builder Tutorial
Creating a Visual Data Access Application- Follow the instructions in this tutorial to create a basic data access application visually, by using some of the parts described above.

Connection Considerations

When you generate classes for your mapping, a class called <*class*>Datastore is generated; it extends one of the Datastore base classes, depending on which access method you use (IDatastoreDB2 for DB/2; IDatastore for ESQL; or IDatastoreODBC for ODBC). Objects of <*class*>Datastore represent connections to the datastore from which the class was mapped. A <*class*>Datastore object can be used by any number of <*class*> objects at a time, but a <*class*> object can only use one <*class*>Datastore object at a time.

This section discusses the three levels of datastore assignment:

- The Application Datastore
- The Default Class Datastore
- The Object's Datastore
- Which Datastore Will Your Object Use? The Current Datastore

Then it addresses different datastore transaction scenarios:

- Single Active Transaction for the Application
- One Active Transaction per <*Class*>
- Multiple Active Transactions per <*Class*>
- Other Typical of Transaction Scenarios

The Application Datastore

Your data access application has a datastore associated with it by default. All <*class*>, <*class*>Manager, and <*class*>DataIdManager objects use this datastore connection for performing add, update, delete, retrieve, select, open, refresh, and user-defined method actions, unless either the default <*class*> datastore or the object's datastore has been set (see the following sections).

Initially, the application datastore is `null`, but all IDatastore, IDatastoreDB2, and IDatastoreODBC constructors check to see whether the application datastore is `null`, and if it is, set the application datastore to the newly constructed datastore object. You can use aDatastore.setAsApplicationDefault to set the application datastore to the current datastore object. You can set the application datastore to null by calling the aDatastore.removeAsApplicationDefault method (where aDatastore is the current application datastore).

Use the applicationDefault method to return the current application datastore.

The Default Class Datastore

If you prefer to determine class-by-class which datastore objects will use, you can associate a datastore with a particular class by using the default class datastore setting (also called the *default datastore*). All <*class*>, <*class*>DataId,

<*class*>Manager, and <*class*>DataIdManager objects use the default datastore connection for performing add, update, delete, retrieve, and user-defined method actions, unless the object's datastore has been set (see the following section).

Initially, the default datastore is null, but all <*class*>Datastore constructors change a null default datastore setting to the newly constructed datastore object. You can use the <*class*>.setDefaultDatastore(aDatastore) method to set the default datastore to any datastore object, or to null by not passing in any parameter. You can also use aDatastore.setAsDefaultDatastore to set the default datastore to the current datastore object.

Use <*class*>.defaultDatastore to return the current <*class*>-level default datastore.

The Object's Datastore
Finally, datastore settings can be made at the object's level (that is, for a particular instance of <*class*>). When a <*class*> object's datastore setting is set to a particular datastore (that is, not null), then the object will use that datastore connection for performing add, update, delete, retrieve, and user-defined method actions.

<*Class*> objects created by calling the default constructor have a null object's datastore. If you pass in a datastore object to the constructor, it will set the object's datastore to that datastore. You can use the a<*Class*>Object.setObjectsDatastore(aDatastore) method to set the object's datastore to any datastore object, or to null, by not passing in any parameter.

Use a<*Class*>Object.objectsDatastore to return the current <*class*> object's datastore.

Similar object-level datastore set and get methods exist for Manager objects. If you set the <*class*>Manager object's datastore, then each transaction done on fetched objects will be done via that datastore. The default datastore for the <*class*>Manager class is always the <*class*> default datastore.

Which Datastore Will Your Object Use? The Current Datastore
Because you have three levels of datastore assignment, it is important to know which connection an object will use when it accesses the database. All transactions on an object use the object's *current datastore*. The a<*Class*>Object.currentDatastore method returns the current datastore for the object. The currentDatastore method determines which datastore is current as follows:

- If the object's datastore is not null, then the current datastore is the object's datastore

- If the object’s datastore is null and the default (class-level) datastore is not null, then the current datastore is the default datastore
- If both object’s and default (class level) datastores are null, then the current datastore is the application datastore.

Note that if the currentDatastore method returns null, then calling connect or attempting a transaction throws an exception; you cannot access a database when all datastore settings for the current object are null.

Single Active Transaction for the Application
The most common and easy to use scenario is when you use one datastore, and therefore have only one active transaction for the entire application. In this situation, you set the *application datastore* once (implicitly, using a datastore constructor), and all objects access the database through that one transaction. The following example shows how to implement this scenario. Consider two classes, Employee and Department, both mapped from one DB/2 database, SAMPLE:

```
IDatastoreDB2    aDatastore;            // aDatastore is set to be application datastore
Employee         anEmp;
Department       aDept;
aDatastore.setDatastoreName("SAMPLE");
aDatastore.connect();
{                                       // start the transaction with the first retrieve, add, etc.
  // Do some actions on anEmp and aDept
}
aDatastore.commit();                    // complete the transaction
```

Similarly, you could do the following, which still results in both classes using the same application datastore:

```
EmployeeDatastore  anEmpDS;  // Uses database name you specified in the mapping.
Employee           anEmp;
Department         aDept;
anEmpDS.connect();
{                                       // start the transaction with the first retrieve, add, etc.
  // Do some actions on anEmp and aDept
}
anEmpDS.commit();                       // complete the transaction
```

By creating the datastore object with the EmployeeDatastore constructor, you avoid having to enter the database name because the generated code picks it up from the mapping settings. This datastore, anEmpDS, not only becomes the default datastore for the Employee class, but also sets itself as the application datastore (this assumes that this code runs before any other part of the application has set the application datastore to something other than null). So, anEmpDS is also the current datastore for the Department class, and Department objects use it for all their transactions.

One Active Transaction per *<Class>*

The next-to-easiest scenario is to use one active transaction for each *<class>*. In this scenario, you set each *<class>*'s *default datastore*, and, while there is still only one active transaction per class, it is independent from the transactions of other classes. To simplify this scenario, you may want to set autoCommit to true for each datastore, so that each method call is committed to the database immediately and automatically. Then you don't have to keep track of which transactions you have completed, and which you have not.

For example, suppose your application has two *<class>* classes, Employee and Department. If they are mapped from different databases, it is impossible to use the same datastore connection. Even if they are from the same database, you may want to keep their transactions separate. In this case, you would do the following:

```
EmployeeDatastore      anEmpDS; // uses database name you specified in the mapping.
DepartmentDatastore    aDeptDS;
Employee               anEmp;
Department             aDept;
anEmpDS.connect();
aDeptDS.connect();
{
  // Do some actions on anEmp and aDept
}
anEmpDS.commit();      // Here, only anEmp transaction is committed.  If you now try to disconnect
                       // from aDeptDS, you may get an exception, or you may lose the changes in
                       // the transaction.
```

In this scenario, you use only the default (class-level) datastores for transactions; the application datastore, although it was set by the first datastore constructor (EmployeeDatastore in the above example), is not used.

Multiple Active Transactions per *<Class>*

There may be situations when you wish to handle transactions at the object level. In this case, you set the *object's datastore* for each object. For example, if you have a *<class>* called Employee, you might want to handle transactions for employees with more than ten years' experience separate from transactions on those with less than ten years' experience. The following code shows what you might do:

```
Employee              anEmp;
EmployeeDatastore     anEmpDSLessThanTen;
EmployeeDatastore     anEmpDSMoreThanTen;
{
  {
    // Get input into anEmp
  }
  if (anEmp.getYearsExperience() < 10) anEmp.setObjectsDatastore(anEmpDSLessThanTen);
    else anEmp.setObjectsDatastore(anEmpDSMoreThanTen);
}
// Repeat above for a collection of objects.
```

```
anEmpDSLessThanTen.connect();
anEmpDSMoreThanTen.connect();
{
  // Do some actions on these objects
}
anEmpDSLessThanTen.commit();
anEmpDSMoreThanTem.commit();
```

If you repeat this for a group of objects, one transaction will handle some objects, and the other transaction will handle the rest, depending on the number of years of experience. These transactions commit changes to the database, as specified later in the code. Although the class default datastore or application datastore may have been used to retrieve the objects originally, they are not used for transactions, once the objects' datastores have been set.

Other Typical Transaction Scenarios

Depending on the number of <*class*>es and objects you use, and depending on how much control you want to have, you can have any combination of the above scenarios. One example could be three classes: Employee, Department, and Vehicle, all mapped from the same database. In this case, it might make sense to have Employee and Department classes use the same datastore, but to have the Vehicle class use its own datastore. Your code might look like this:

```
VehicleDatastore      aVehDS;   // sets default datastore for Vehicle class
EmployeeDatastore     anEmpDS;  // sets default datastore for Employee class
Department            aDept;
Department.setDefaultDatastore(anEmpDS);  // sets default datastore for Department to anEmpDS
```

Now, all Employee and Department objects will use the EmployeeDatastore for transactions, while all Vehicle objects will use the VehicleDatastore.

If you are unsure which transaction is handling a particular object, you can always check by calling the currentDatastore method.

Chapter 2. Start a Data Access Builder Session

Start Data Access Builder SmartGuide

The SmartGuide creates a schema mapping for you in a few simple steps.

OS/2 WIN To start the SmartGuide from the VisualAge C++ IDE, create a Data Access target, and add a Data Access Part to the target.

AIX To start Data Access Builder, enter `cppdata` from the command line. If the Startup Window appears, select **Create Class Mapping** and the SmartGuide will start.

To start the SmartGuide from the Data Access Builder window, select **Map schema** from the **File** menu.

Import a Data Access Builder Session

You can import a .dax file that contains mappings from another Data Access Builder session into a Data Access part in the IDE. The generated files from the mappings are not part of this import process, but can be generated after you import the session. You can use the import function to migrate Data Access session files from previous versions of VisualAge C++ and VisualAge for Java. It is also useful for sharing work among team members; a mapping can be exported by one team member and distributed to others by network or floppy disk. Team members then import the session into their own workspaces.

To import the session into a Data Access part for the target, do the following steps:

1. Create a new Data Access Part for the target. This launches the Data Access Builder SmartGuide.
2. On the first page of the SmartGuide, select the option to import a .dax file.
3. On the following page, enter the path and file name for the desired .dax file.
4. When you exit the SmartGuide, the Data Access Part reflects the addition of the mapping.
5. To customize the mapping, click **Edit Part**. Data Access Builder starts and the mapping is displayed in the main window. From here you can customize the mapping and generate classes.

6. In the Data Access Builder main window, select **Save** from the **File** menu. This will generate the .vbe and .vbf files needed by the Data Access Part.

A target may only have one Data Access Builder session associated with it. If the current target already has a Data Access Builder part, and you want to import a session that will *replace* the current part, then do the following steps:

1. Select the current Data Access part in the Data Access Launch view. Click **Edit Part**. This will launch Data Access Builder with the mapping information in the current part.
2. Start the SmartGuide by selecting **Map schema** from the **File** menu.
3. On the first page of the SmartGuide, select the option to import a .dax file.
4. On the following page, enter the path and file name for the desired .dax file.
5. When you exit the SmartGuide, the previous Data Access part is *overwritten* with the new part information.
6. To customize the mapping, click **Edit Part**. Data Access Builder starts and the mapping is displayed in the main window. From here you can customize the mapping and generate classes.
7. In the Data Access Builder main window, select **Save** from the **File** menu. This will generate the .vbe and .vbf files needed by the Data Access Part.

Chapter 3. Work with Database Tables

Modify Table Filter

The table filter in SmartGuide step to **Select Tables for Mapping** enables you to reduce the number of tables to be displayed when you click **Get tables**. This is useful when the database you are accessing contains many tables. To set the table filter:

1. Type wild cards in the **Owner name** and **Table name** entry fields. Owner and table names are case-sensitive. Two wild cards are valid:

   ? is a single character wild card
   * is a multiple character wild card
2. Click on the **Table type** list box to see the different types of tables available for the filter. Select the type of filtering you want.

Note: Data Access Builder has a limit of how many tables can be displayed in the **Tables** list box. The exact limit depends on your machine and the size of its virtual memory. If your database has many tables (in the order of several thousand), try to use a specific filter.

Specify Table Joins in Schema

Specify table joins for the schema by selecting **By entering an SQL statement** in the SmartGuide. When you enter the SQL statement for the query, enter the desired table join. See Create a Mapping from More than One Database Table for an example of an SQL statement for a table join.

View Database Table or View Properties

To view a database table or view's information, such as data type of its columns, open the schema properties notebook by double-clicking on the schema icon ▦ ▦ .

Recover When Database Definitions Change

Changes made to database tables are *not* automatically updated in Data Access Builder. Database schema synchronization is not maintained. The only connection between the actual table and the Data Access Builder object occurs once - when the SmartGuide is creating this schema mapping.

It is possible to open a saved session that contains old table definitions and to resume work on it. But if the changes to the database table or view are substantial, your generated Data Access Builder code may no longer be usable. Examples of safe updates to a table that normally will not affect your generated code's viability are adding a column, deleting an unmapped column, or adding a foreign key.

To update the schema mapping:

1. In the Data Access Builder window, delete the schema object.
2. Start the SmartGuide again to recreate the schema mapping.

Chapter 4. Create Schema Mappings

Map a Schema

You create a new Data Access Builder schema mapping with the SmartGuide. This information in the mapping is used to generate classes. The schema mapping is visually represented by icons in a tree structure in the Data Access Builder window. To create a schema mapping, click **File** on the menu bar of the Data Access Builder window, and select **Map Schema**.

Create Mappings

If you do not have the schema that you want in the Data Access Builder window, create a schema mapping by invoking the Data Access Builder SmartGuide. In the SmartGuide steps, specify your schema and then the mapping is created for you. The schema mapping is represented as icons in a tree structure in the Data Access Builder window.

If you already have the schema that you want in the Data Access Builder window, you can create another mapping from the *same* schema:

1. Select the schema icon ▦ ▦ .
2. Select the appropriate **Create Mapping** option from the schema's popup menu. If the database source is DB2, you will have the options **Create Mapping (CLI)** and **Create Mapping (ESQL)**. If the source is ODBC, the option is simply **Create Mapping**.

The Data Access Builder window displays the new mapping linked to the schema.

Create a Mapping from More than One Database Table

To create one mapping from more than one database table, select the mapping method **By entering an SQL statement** in the Data Access Builder SmartGuide Step - Select Database and Mapping Method. When you enter the SQL statement, you can use table joins in the statement, as shown in the following example:

```
SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME,
SALARY+BONUS+COMM
AS TOTAL_PAY FROM PROJECT, EMPLOYEE
```

The resulting schema icon ▦ in the Data Access Builder window represents a logical grouping of all the specified columns in your SQL statement. The mapping for the schema is created so that you can generate classes that access data from table joins. Some methods are not generated for table joins, for details see Table Joins and Query Support.

Create More than One Mapping for a Schema

If you have a schema mapping, and you want to create another mapping for the same schema:

1. In the Data Access Builder window, select the desired schema.
2. Click Mouse Button 2 to display its pop-up menu.
3. Select **Create mapping**. If you are accessing a DB2 database, choose from the two options (**Create mapping (CLI)** or **Create mapping (ESQL)**).

Another mapping icon for the schema is displayed in tree structure.

Create Mapping from SQL Statements

You can enter an SQL query statement as the schema and Data Access Builder will create a mapping from it. Enter the SQL statement during the SmartGuide steps.

Create Read-Only (R/O) or Read/Write (R/W) Classes

When you have created a Data Access Builder mapping, a default read-only or read / write access is set for the generated classes. View and change this setting from the mapping’s properties notebook. If the table for the mapping is read-only, then its generated classes will be, by default, read-only.

Create Mapping to Generate DB2 ESQL

For accessing DB2 Universal Database products, you can specify if you want to generate code that uses ESQL or CLI to access your database. When you are in the Data Access Builder SmartGuide, you can select this in the Select Database Access Method step. Once you have created the mapping you cannot change the access method. You have to delete the mapping icon and Create Mapping from the same schema icon.

If you are creating a mapping to a DB2 schema that is already defined, select the schema or , and then select **Create Mapping (ESQL)** from the schema’s popup menu.

Create Mapping to Generate DB2 CLI

For accessing DB2 Universal Database products, you can specify if you want to generate code that uses ESQL or CLI to access the database. When you are in the Data Access Builder SmartGuide, you can select this in the Select Database Access Method step. Once you have created the mapping you cannot change the access method. You have to delete the mapping icon and Create Mapping from the same schema icon.

If you are creating a mapping to a DB2 schema that is already defined, select the schema or , and then select **Create Mapping (CLI)** from the schema’s popup menu.

Create Mapping to Generate ODBC

If you select an ODBC datasource in the first SmartGuide step, the mapping will automatically be one that uses ODBC to access data from the generated classes. Once you have created the mapping, you cannot change the access method.

If you are mapping from an ODBC schema that has already been defined, select the schema or , and then select **Create Mapping** from the icon’s popup menu.

Add Comments for a Mapping

You can add comments for mappings and they will be included in the generated code (<*class*> class). The comment can be entered when you are first creating the schema mapping in the Data Access Builder SmartGuide steps. Add or change comments by using the mapping’s properties notebook:

1. In the Data Access Builder window, select the mapping icon.
2. Double-click on the mapping icon to open its properties notebook.
3. Select the Comment Page and type the comment in the edit box.

4. Click **OK** to save the comment and close the notebook.

The comment will be in the generated code.

Chapter 5. Customize Schema Mappings

Customize the Mapping

There are several ways you can customize a mapping, including:

- Change Mapping Properties
- Change Attribute Properties
- Add Comments to Generated Code
- Add User-Defined Methods
- Delete Attributes
- Exclude Methods in the Generated Code
- Recover When Database Definitions Change

Change Mapping Properties

Some of the customization you can do to the mapping is done by changing the mapping's properties. Double-click on the mapping icon to open its properties notebook. Make the following changes in the mapping properties notebook:

- In the Source Page
  - Change class name which is used as the base name for generated classes. Type the new class name in the **Class name** entry field.
  - Change qualifier for the generated classes. Type the new qualifier in the **Table Qualifier** entry field.
    - Specify a static table qualifier that has the required access to the table.
    - Change the qualifier to a blank field. The user ID for the person executing the application is used as the table qualifier.
    - Change the qualifier to `%QUAL%`. The run-time qualifier is used. At run time, change the value of `%QUAL%` with <*class*>::setQualifier.
  - Change Read-only access to database to specify that all instances of your class are not allowed to add, update, or delete database information. An exception is thrown if a write access is attempted with classes that have been created with this option selected. **Note**: Data Access Builder may place a check in this checkbox if it can determine that a table or view is read-only. Click the **Read-only access to database** check box to select or deselect the option.

After you have completed the changes, click **OK** to apply the change and close the notebook.

Change Attribute Properties

The columns in the schema are mapped to attributes. Open the attribute's properties notebook by double-clicking the attribute icon or .

Change the attribute's properties by doing any of the following tasks:

- **Change attribute name**
  - – Type a new name in the **Name** entry field.
- **Change attribute type**
  - – Click the down arrow in the **Type** box to view all supported types and select the type you want. See Data Types Conversion for DB2 Data Types and ODBC Data Types for the default and supported data types.
- **Change attribute access method**
  - – Change the attribute access method for attributes that have DATE, TIME, and TIMESTAMP data types. Click the down arrow in the **Access Method** box to view all supported access methods and select the method that you want:
    - **Attribute Value -** always use attribute value for update and add actions
    - **CURRENT on add -** use attribute value for update action and use CURRENT DATE/TIME/TIMESTAMP value for add actions
    - **CURRENT on update -** use attribute value for add action and CURRENT DATE/TIME/TIMESTAMP value for update actions
    - **Always CURRENT -** always use CURRENT DATE/TIME/TIMESTAMP value for update and add actions
- **Change Data Identifier setting**
  - – Select the **Data Identifier** checkbox to identify the attribute as the data identifier. If no data identifier attribute is specified for a class, then retrieve, update, and delete methods will not be generated for it. By default, the primary keys of the database table are mapped as data identifiers. For additional information generated code and the data identifier, see Keyless Table Support and Data Identifiers.
- View or change **Is Displayed** setting.
  - – Select the **Is Displayed** checkbox. The attribute will be included in the concatenated string for the **forDisplay** method.
- **Add Comment**
  - – On the Comment page, enter a comment for the attribute and it will be added to the generated code.

After you have completed the changes, click **OK** to apply the change and close the notebook.

Note: You can also change attribute's Data Identifier and Is Displayed settings from the Attributes Window by selecting the mapping icon , and selecting **Attributes** from its popup menu.

Add Comments for an Attribute

To add descriptive comments for attributes, enter the comment in the attribute's properties notebook, by doing the following steps:

1. In the Data Access Builder window, select the attribute icon or .
2. Double-click on the attribute icon to open its properties notebook.
3. Select the Comment Page and type the comment in the edit box.
4. Click **OK** to save the comment and close the notebook.

The comment will be added to the generated code.

Delete Attribute

To delete an attribute from a mapping, do the following steps:

1. Select the attribute or that you want to delete.
2. Select **Delete** from the attribute's popup menu.

The attribute icon disappears, and the generated code will not include the attribute. To recreate a previously deleted attribute follow the steps in Map Attribute to Columns.

Map Attributes to Columns

When a mapping is created, attributes are created for all the columns in the schema. If you deleted an attribute while customizing the mapping, but decide you want to add it back to the mapping, do the following steps:

1. Select **Attributes** from the mapping icon's popup menu.
2. In the Attributes window, scroll to find the column you want map and select it.
3. Click **Map Column** to add the attribute for the selected column.

The attribute is displayed in the Attributes window.

Delete Schema Object

To delete a schema object from the Data Access Builder window, do the following steps:

1. Select the schema icon ▦ ▦ .
2. Select **Delete** from the icon's popup menu. All the mappings for the schema are also deleted.

Use the SmartGuide to recreate the schema mapping.

Delete Mapping Object

To delete a mapping object in the Data Access Builder window do the following steps:

1. Select the mapping icon ● that you want to delete.
2. Select **Delete** from the icon's popup menu.

View Schema Properties

You can view (but *not* modify) schema properties in the schema properties notebook. Double-click the schema icon ▦ ▦ in the Data Access Builder window. The notebook pages vary depending on whether Data Access Builder creates a mapping from a database table, database view, or SQL query.

View Column Properties

You can view a schema column's properties but you cannot modify its contents. To view a column's properties, do the following:

1. Select the mapping icon ● .
2. Select **Attributes** from the mapping icon's popup menu.
3. Scroll to find the column that you want to view and select it.
4. Click **Properties** to open its properties notebook.

Column information is also available in the schema properties notebook on the **Columns Page**.

Add User-Defined Methods

You can define your own methods in a Data Access Builder session and code will be generated for them. The method is added to the generated classes for a mapping. Do the following steps to add a method:

1. From the Data Access Builder window, open the Methods or Manager Methods window to add a user-defined method:
   a. Select the mapping icon ● to which you want to add the method.
   b. Select either **Methods** or **Manager Methods** from the icon's popup menu. Data Access Builder generates <*class*> classes that provide database access to handle a single object and it generates the <*class*>Manager classes to handle a collection of objects (returns a result set). If you want to add your method to the <*class*> class, select **Methods**. If you want to add your method to the <*class*>Manager class, select **Manager Methods**.
   c. Either in the Methods window or the Manager Methods window is opened for you to add the method.
2. Indicate whether your method is specified with a customized SQL statement, as a stored procedure call, or with an SQL predicate.
   a. In the Methods window or Manager Methods window, click **Add** under the appropriate container to indicate the method that you want to add. For example, click **Add** under the **Customized SQL Statement** container when you want to add a method by typing the SQL statement for it.
   b. A window is opened for you to add your method.
3. Perform the steps in one of the following user-defined methods:
   a. Add User-Defined Methods with Customized SQL Statements
   b. Add User-Defined Methods That Are Stored Procedure Calls
   c. Add User-Defined Methods with SQL Predicates
4. Click **Close** to close the Methods or Manager Methods window.

When you generate code for the mapping, the method you have defined will be included in the appropriate class.

Add User-Defined Methods with Customized SQL Statements

After specifying that you want to add or modify a user-defined method with a customized SQL statement (see Add User-Defined Methods or Modify User-Defined Methods on how to do this), you perform the following steps in the Customized SQL Statementnotebook:

1. Enter the name of the method in the **Method Name** entry field.
2. On the SQL Statement page, enter the SQL statement in the **SQL Statement** entry field. For example, to create a method to update the SALARY and BONUS columns of the EMPLOYEE table, enter:

   ```
   UPDATE EMPLOYEE SET SALARY=?, BONUS=? WHERE EMPNO=?
   ```

   Use %QUAL% to specify that a run-time qualifier for the table is used when executing the SQL query. For example, enter this SQL statement:

   ```
   UPDATE %QUAL%EMPLOYEE SET SALARY=?, BONUS=? WHERE EMPNO=?
   ```

   The default value of %QUAL% is the qualifier in the mapping's properties notebook. At run time, use the setQualifier method to set the value of %QUAL%.
3. Click **Validate**. A database connection is required for the validation. An error window opens if the SQL statement is not valid. If the SQL statement is valid, the **Parameters** list box on the Parameter page lists all the parameters for the user-defined method.

   Warning! If you are using DB2, the following lines must be in the DB2CLI.INI file in the SQLLIB directory, or else the validate action will return without an error message regardless of the SQL statement's actual validity:

   ```
   [common]
   DEFERREDPREPARE=0
   ```
4. Your SQL Statement may affect multiple rows. Since the Data Access Builder does not know if this is expected behavior, specify whether an exception should be thrown if multiple rows are affected by the SQL statement. To include the generation of this code, select the **Throw exception if multiple rows affected** check box. By default, this box is checked.
5. Enter a **Display Name** for the method. This name can contain double-byte characters and is used instead of the **Method Name** by builders such as the Visual Builder. By default, this is the same as **Method Name**.
6. The SQL Type page shows the SQL Types of parameters in the SQL statement. If the Validate action was successful, then all of the information is filled in for your information. However, if it was unable to determine the type of a parameter, there will be an "X" next to the parameter name in the **Parameters** list box. Then you must assign the parameter an SQL type by doing the following steps:
   a. Select the parameter in the Parameter list box.
   b. Select the desired type from the **SQL Type** drop-down list.
   c. Depending on what type you choose, fill in the appropriate values in the **Length**, **Precision**, and **Scale** fields.

   d. Click **Modify**. The ”X“ will be replaced by a check-mark.

7. On the Parameter page, map the parameters to parts of the C++ method and to C++ types by doing the following steps for each parameter:
   a. Select the parameter in from the Parameters list box.
   b. Select a radio button to specify the part of the C++ method that the parameter will be **Mapped To**. Depending on the parameter type (from the SQL Type page), some of these radio buttons will be enabled and some will be disabled. Choose one of the following settings:
      - **Parameter -** Maps the SQL statement parameter to a parameter in the C++ method (it will appear in the method signature)
      - **Attribute -** Maps the SQL statement parameter to an attribute in the mapping. Input values for the method will be taken from the specified attribute in the current <*class*> object, and output values (if any) will be assigned to the specified attribute in the current <*class*> object.
      - **Return Value -** Maps the SQL statement parameter to the return value of the C++ method (it will be assigned to the variable on the left-hand side of an assignment call). This option is enabled only for output parameters.
      - **None -** Leaves the SQL statement parameter out of the method mapping. This option is enabled only for output parameters.
   c. Depending on your choices for the following option, fields to the right-hand side of the radio button option will be enabled for you to complete, as follows:
      - If you choose **Parameter**, fill in a **Name** for the parameter, and select a **Type** from the drop-down list of appropriate types (one is already selected by default).
      - If you choose **Attribute**, select a **Name** from the drop-down list of appropriate <*class*> attributes.
      - If you choose **Return Value**, select a **Type** from the drop-down list of appropriate types (one is already selected by default).
      - There is nothing further to specify if you choose **None**.
   d. Click the **Modify** button and move on to the next parameter.
8. Enter a comment for the method on the Comment Page. This comment will be added to the generated code for the method.
9. Click **OK** to close the window and the method is added to the method’s container in the Methods window.

Add User-Defined Methods That Are Stored Procedure Calls

After specifying that you want to add or modify a user-defined method which is stored procedure call (see Add User-Defined Methods or Modify User-Defined Methods on how to do this), you perform the following steps in the Stored Procedure Call notebook:

1. Enter a name for the method in the **Method Name** entry field.
2. On the Stored Procedure page, click the drop-down list to view all the stored procedures defined for the database in the Data Access Builder session and select the name of the stored procedure that you want to work with.

   If you do not find the stored procedure in the list, you have to define the stored procedure first and then the stored procedure will show up in the **Stored Procedure Name** list box.
3. Enter a **Display Name** for the method. This name can contain double-byte characters and is used instead of the **Method Name** by builders such as the Visual Builder. By default, this is the same as **Method Name**.
4. On the Parameter page, map the parameters to parts of the C++ method and to C++ types by doing the following steps for each parameter:
   a. Select the parameter in from the Parameters list box.
   b. Select a radio button to specify the part of the C++ method that the parameter will be **Mapped To**. Depending on the parameter type (from the stored procedure definition), some of these radio buttons will be enabled and some will be disabled. Choose one of the following settings:
      - **Parameter -** Maps the stored procedure parameter to a parameter in the C++ method (it will appear in the method signature).
      - **Attribute -** Maps the stored procedure parameter to an attribute in the mapping. Input values for the method will be taken from the specified attribute in the current <*class*> object, and output values (if any) will be assigned to the specified attribute in the current <*class*> object.
      - **Return Value -** Maps the stored procedure parameter to a return value of the C++ method (it will be assigned to the variable on the left-hand side of an assignment call). This option is enabled only for output parameters and stored procedure return value parameters.
      - **None -** Leaves the stored procedure parameter out of the method mapping. This option is disabled for input parameters.

   c. According to which of the following options you choose, fields to the right-hand side of the radio button option will be enabled for you to complete, as follows:

    - If you choose **Parameter**, fill in a **Name** for the parameter, and select a **Type** from the drop-down list of appropriate types (one is already selected by default).
    - If you choose **Attribute**, select a **Name** from the drop-down list of appropriate <*class*> attributes.
    - If you choose **Return Value**, select a **Type** from the drop-down list of appropriate types (one is already selected by default).
    - There is nothing further to specify if you choose **None**.

   d. Click the **Modify** button and move on to the next parameter.

5. Enter a comment for the method on the Comment Page. This comment will be added to the generated code for the stored procedure call method.
6. Click **OK** to close the notebook and the method is added to the method's container in the Methods or Manager Methods window.

Define Stored Procedures

Before you can define a method that calls a database stored procedure, you must describe it to Data Access Builder. To view the list of stored procedures that have been defined within Data Access Builder, open the Stored Procedures List window by doing one of the following actions:

- Select **Stored Procedures** from the popup menu of the desired database; or,
- In the Add Stored Procedure Call window, the **Procedure Name** drop-down box lists all those defined. Click the **Show Procedures** button to open the Stored Procedures List window.

If the stored procedure you want to work with is not listed in the **Imported Stored Procedures List** in the Stored Procedures List window, then you must add the definition to Data Access Builder. There are two ways to define the stored procedure:

1. Import the definition from the database.
2. Define the stored procedure manually.

The following sections describe how to define the stored procedures by each of these methods.

Importing a Definition from the Database

If the stored procedure that you want to use is defined in the database, and if a database connection is presently available, you can import the definition into Data Access Builder by doing the following steps:

1. On the Stored Procedures List window, click **Get Stored Procedures**. This accesses the database and returns a list of available stored procedure definitions to the **Stored Procedures in** *Database* list box.

2. Select the stored procedure definition that you want to import and click **Import**. It will be added to the **Imported Stored Procedure List**.
3. It is advisable that you check the imported procedure definition for accuracy before you map a stored procedure call to it. For information on doing this, see Modify Stored Procedure Definitions.

Manually Defining a Stored Procedure
If the a database connection is not available, or if the stored procedure that you want to use is not defined there, you can provide a definition manually. The definition you provide must accurately reflect the stored procedure on the database. Otherwise, run time errors will occur when the stored procedure call method is invoked.

To define a stored procedure manually, do the following steps:

1. From the Store Procedures List window, Click **Add** to launch the Add Stored Procedures window.
2. In the Stored Procedure field, provide a name for the stored procedure. This is the name that appears in the **Imported Stored Procedures List**, and in the **Procedure Name** drop-down list in the Add / Modify Stored Procedure Call notebook.
3. The Parameters list box contains a list of all parameters defined for the stored procedure. To add a parameter, do the following steps:
   a. Select a **Parameter Type**, according to what the parameter does within the stored procedure:
      - **In -** Designates an input parameter to the stored procedure.
      - **Out -** Designates an output parameter from the stored procedure.
      - **In/Out -** Designates a parameter that is used for both input and output.
      - **Result Column -** Designates a column in the result set returned by the stored procedure.
      - **Return Value -** Designates the return value for the stored procedure.

   b. Select an **SQL Name** and **Type**; as appropriate, enter the **Length**, **Precision**, and **Scale** of the SQL parameter.
   c. Click **Add**.
   d. Adjust its position in the parameter list by using the **Move Up** and **Move Down** buttons. Parameters must appear in the same order as they do in the stored procedure on the database.
4. To work with an existing parameter, select it and do any of the following actions on it:
   a. **Delete -** Deletes the selected parameter.
   b. **Copy -** Copies the selected parameter and generates a new, unique name for the new parameter.

c. **Move Up -** Moves the selected parameter up in the order of parameters.Parameters must appear in the same order as they do in the stored procedure on the database.

d. **Move Down -** Moves the selected parameter down in the order of parameters.Parameters must appear in the same order as they do in the stored procedure on the database.

e. **Modify -** Change the **Parameter Type**, **SQL Name**, **SQL Type**, **Length**, **Precision**, or **Scale**, and click **Modify**.

When you are finished modifying the definition, click **OK** to exit the dialog, or **Cancel** to cancel your changes and exit. The method will now appear in the **Imported Stored Procedures List** in the Store Procedure List window, and will be available for mapping to stored procedure call methods in the Add/Modify Stored Procedure Call notebook.

Modify Stored Procedure Definitions

Before you can define a method that calls a database stored procedure, you must describe it to Data Access Builder. To view the list of stored procedures that have been defined within Data Access Builder, open the Stored Procedures List window by doing one of the following procedures:

- Select **Stored Procedures** from the popup menu of the desired database; or,
- In the Add Stored Procedure Call window, the **Procedure Name** drop-down box lists all those defined. Click the **Show Procedures** button to open the Stored Procedures List window.

If the stored procedure you want to work with is not listed in the **Imported Stored Procedures List** in the Stored Procedures List window, then you must add the definition to Data Access Builder (see the section Define Stored Procedures).

Stored procedures definitions that were previously imported or created can be modified to correct errors or to reflect changes made to the stored procedure on the database. To modify a stored procedure definition, select the definition from the **Imported Stored Procedures List** and click **Modify**.

Warning! Once a stored procedure has been defined to Data Access Builder, any modification to its definition deletes all stored procedure call methods mapped to it.

In the Modify Stored Procedure window, adjust the following fields as needed. Ensure they accurately reflect the database stored procedure's definition.

1. **Stored Procedure** field - Enter a name for the stored procedure definition.

2. **Parameters** list box - Each parameter is listed in the list box. To work with one of them, select it and do any of the following actions on it.
   a. **Delete -** Deletes the selected parameter.
   b. **Copy -** Copies the selected parameter and generates a new, unique name for the new parameter.
   c. **Move Up -** Moves the selected parameter up in the order of parameters.Parameters must appear in the same order as they do in the stored procedure on the database.
   d. **Move Down -** Moves the selected parameter down in the order of parameters.Parameters must appear in the same order as they do in the stored procedure on the database.
   e. **Modify -** Change the **Parameter Type**, **SQL Name**, **SQL Type**, **Length**, **Precision**, or **Scale**, and click **Modify**. See the next list item, below, for details about each of these fields.
3. To add a new parameter, ensure you have finished modifying the currently-selected parameter, and then do the following steps:
   a. Select a **Parameter Type**, according to what the parameter does within the stored procedure:
      - **In -** Designates an input parameter to the stored procedure.
      - **Out -** Designates an output parameter to the stored procedure.
      - **In/Out -** Designates a parameter that is used for both input an output.
      - **Result Column -** Designates a column in the result set returned by the stored procedure.
      - **Return Value -** Designates the return value for the stored procedure.
   b. Select an **SQL Name** and **Type**; as appropriate, enter the **Length**, **Precision**, and **Scale** of the SQL parameter.
   c. Click **Add**.
   d. Adjust its position in the parameter list by using the **Move Up** and **Move Down** buttons. Parameters must appear in the same order as they do in the stored procedure on the database.

When you are finished modifying the definition, click **OK** to exit the dialog, or **Cancel** to cancel your changes and exit.

Display List of Defined Stored Procedures

To view the list of stored procedures that have been defined to the Data Access Builder, select Stored Proceduresfrom the schema icon's or

popup menu. If the stored procedure you want is not listed in the **Stored Procedure Name** list, click the **Show Procedures** button to open the *Database*-Stored Procedures Listwindow (where *Database* is the name of the database the schema is from).

The name of the database is displayed in the window's title bar and the stored procedures that have been defined to Data Access Builder are displayed in the **Imported Stored Procedures List** container. Once a stored procedure has been defined to Data Access Builder, any modification to it deletes all its stored procedure call methods.

Import Database Stored Procedure Definitions

To import a stored procedure's definition to the Data Access Builder so that you can add a method that calls the stored procedure, do the following steps:

1. Open the Stored Procedure List window by selecting Stored Procedures from the schema icon's popup menu.
2. If stored procedure that you want is not listed in the **Stored Procedure Name** list, click the **Show Procedures** button to open the *Database*-Stored Procedures List window, where *Database* is the schema's database. The name of the database is displayed in the window's title bar and the stored procedures that have been defined to Data Access Builder are displayed in the **Imported Stored Procedures List** container. To import another stored procedure definition:
   a. Click **Get Stored Procedures** to connect to the database and display all its stored procedures in the **Stored Procedures** container.
   b. Select the stored procedure that you want to import and click **Import**. The stored procedure is added to the **Imported Stored Procedure List** container.
   c. Click **Close** to close the window.

Add User-Defined Methods With SQL Predicates

After specifying that you want to add or modify a user-defined method with an SQL predicate (see Add User-Defined Methods or Modify User-Defined Methods on how to do this), you perform the following steps in the Add SQL Predicate notebook:

1. Type the name of the method in the **Method Name** entry field.
2. On the SQL Predicate page, to help you construct the SQL predicate, the first part of the SQL statement from the mapping is shown above the SQL Predicate entry field. To enter the SQL predicate, type the SQL predicate in the **SQL Predicate** edit box. For example, to predefine the query of retrieving all CARs that are of a certain color and price over $10,000, type the following SQL:

   ```
   WHERE COLOR=? AND PRICE>10000
   ```
3. Click **Validate**. A database connection is required for validation. Each parameter (indicated by question marks in the SQL Predicate) is mapped to a parameter and a default C++ type. To change the C++ type mapping, go to the Parameter page.

   Warning!If you are using DB2, the following lines must be in the DB2CLI.INI file in the SQLLIB directory, or else the validate action will return without an error message regardless of the SQL statement's actual validity:

   ```
   [common]
   DEFERREDPREPARE=0
   ```
4. Enter a **Display Name** for the method. This name can contain double-byte characters and is used instead of the **Method Name** by builders such as the Visual Builder. By default, this is the same as **Method Name**.
5. The Parameter page shows the default mapping of SQL parameters to C++ parameters. If the Validate action was unable to determine the SQL type of a parameter, there will be an ”X“ next to the parameter name in the list box. You must select an SQL type for the parameter by doing the following steps:
   a. Select the parameter in the Parameters list box.
   b. Select the correct SQL type for the parameter in the **SQL Type** drop-down list.
   c. Click the **Modify** button. The ”X“ will change to a check-mark.

   To change the default mapping of parameters to C++ types, do the following steps for each parameter:
   a. Select the parameter in from the Parameters list box.
   b. If desired, change the parameter's name in the **Name** field.
   c. Select the desired C++ type from the **Type** drop-down list, which contains only appropriate C++ types.
   d. Click the **Modify** button.
6. On the Comment page, enter a comment for the method. This comment will be added to the generated code for the method.
7. Click **OK** to close the window and the method is added to the method's container in the Manager Methods window.

Add Comments to User-Defined Methods

You can add comments for your user-defined methods from the Methods Window or Manager Methods Window. The comment is added with the method in the generated code. To add comments follow these steps:

1. Select mapping icon that has the user-defined methods.
2. Click mouse button 2 to display its menu and select **Methods** or **Manager Methods** depending on where your user-defined methods were added. The Methods or Manager Methods window is opened.
3. Click to open the Method Comments window.
4. Scroll to see all the user-defined methods.
5. Type your comment for each method in the multi-line edit box under the method name.
6. Click **OK** to close the window.

Otherwise, enter the comment as you add each user-defined method, on the Comment page of the Add / Modify notebook.

Copy User-Defined Methods

Data Access Builder lets you create a copy of a user-defined method or of a default method (for example, the retrieve method). This is an easy way to create a new method that is similar to an existing one. To copy a method:

1. Select the mapping icon that has the user-defined method.
2. Select **Methods** or **Manager Methods** from the icon's popup menu.
3. Select the method that you want to copy.
4. Click **Copy**.

A name is generated for the new method. Modify the new method by selecting it and clicking **Modify**.

Delete User-Defined Methods

To delete a method that you have created, do the following steps:

1. In the Data Access Builder window, select the mapping that has the user-defined method.

2. Select **Methods** or **Manager Methods** from the mapping icon’s popup menu to display all the user-defined methods for the <*class*> class or <*class*>Manager class, respectively.
3. Select the method that you want to delete from one of the containers.
4. Click the **Delete** button under the container.

You can choose to hide (exclude) a user-defined method rather than deleting it. If you hide it, the method will not be included in the generated code.

Modify User-Defined Methods

To modify a method that you have created, do the following steps:

1. In the Data Access Builder window, select the mapping icon that has the user-defined method.
2. Select **Methods** or **Manager Methods** from the mapping icon’s popup menu to display all the user-defined methods for the <*class*> Class or <*class*>Manager Class, respectively.
3. Select the method that you want to modify from one of the containers.
4. Click **Modify** under the container to open the Modify notebook to make your changes.

Once you are in the Modify notebook, customize the method definition by following the steps in one of the following sections:

- Add User-Defined Methods with Customized SQL Statements
- Add User-Defined Methods That Are Stored Procedure Calls
- Add User-Defined Methods with SQL Predicates

Exclude Methods In the Generated Code

To exclude user-defined methods and the default add, retrieve, and delete methods when generating code, do the following steps:

1. Select the mapping icon which has the methods.
2. Click mouse button 2 to display the mapping’s menu.
3. Select **Methods** or **Manager Methods** from the menu.
4. In the Methods or Manager Methods, select the method that you want to exclude.
5. Click **Hide**; a black ’X’ is displayed to the left of the method to indicate that the method is excluded from code generation.
6. Click **Close** to close the window.

Fix Broken Mappings

After user-defined methods have been defined, changing certain parts of a mapping can break the mapping. For example, if you have defined a method that takes an attribute, *A*, as an input parameter, and then you delete *A* from the mapping, the user-defined method will be broken. Similarly, if you change *A*'s type, the method might be broken. Also, when a method is initially validated, it will be broken if Data Access Builder is unable to determine a parameter's SQL type.

If a mapping is broken, the mapping icon has a red 'X' . To fix a mapping, you must fix each broken method:

1. Check both **Methods** and **Manager Methods** windows for broken methods, designated by a red 'X' . These may be in the **Customized SQL Statement** list, the **SQL Predicate** list, or the **Stored Procedure Call** list.
2. Do the following steps to fix a broken method:
   a. Select the method.
   b. Click **Modify**. Parameters that are causing the method to break are indicated by an 'X' in the **Parameters** list.
   c. To assign a type to a parameter of unknown type, go to the SQL Type page, select the broken parameter, and select the appropriate type from the **SQL Type** drop-down list.
   d. To fix a broken parameter, go to the Parameter page, select the broken parameter, and change its settings to match the mapping or the stored procedure definition, as appropriate.
   e. Click **OK**. If the method definition is valid, the red 'X' icon will be replaced by a method icon .

   Alternatively, remove the method by selecting it and clicking **Delete**.

When you have fixed all broken methods, the mapping icon will return to normal .

Note: If a method is broken when code is generated for a mapping, the code for the broken method is excluded from the generated classes.

Chapter 6. Generate Source Code

After you have customized the schema mapping, generate the C++ classes for the mapping. From the Data Access Builder window, select **Generate** from the mapping icon's popup menu. When the source code is generated the color of the mapping icon changes.

Add Comments to Generated Code

You can add comments for the mapping, for each attribute in the mapping, and for each user-defined method. The comments will be in the generated source code. Add comments in any of the following ways:

- Enter comments to the mapping when you are creating the schema mapping in SmartGuide steps.
- Enter comments to the mapping in the mapping's properties notebook.
- Enter comments in the attribute's properties notebooks.
- Enter comments for user-defined methods when you add or modify them.

Migrate Existing Code

Binary compatibility of this release of Data Access class library and generated code with previous releases is not guaranteed. If you want to use Data Access applications developed with previous releases with this release's runtime classes and generated Data Access classes, you must do the following things:

1. Generate new Data Access classes by importing the Data Access Builder session used for the original application's Data Access classes and regenerating. This release of VisualAge runtime classes will not work with Data Access classes from previous releases.
2. Re-compile and link the old application source with the new Data Access classes. The old application executable will not work with the new class library.

Since your old application will not have taken advantage of any of Data Access Builders new features, you will have to add code to the application if you want to use functionality such as stored procedure calls and large object access.

Chapter 7. End a Data Access Builder Session

Save a Data Access Builder Session

When you are in the Data Access Builder window, use **Save** to save the information about *all* mappings for the session. Select **Save** from the **File** menu bar. Saving the session also updates the IDE; the Data Access Launch view will show the information in the session (that is, it updates changes to the .dax file and any .vbe and .vbf files). The session information is saved in the current project directory.

To save the session to a location of your choice, use **Export** from the File menu item.

Export a Data Access Builder Session

All the information for a single Data Access Builder session is stored in a .dax file, which is a component of a Data Access part. It is stored in the current project's directory when the session is saved. To export this file to the file system for archiving or for sharing with another user, do one of the following actions:

- In Data Access Builder, select **Export** from the **File** menu to save the current session to a .dax file anywhere in the file system; or,
- In DOS, OS/2, or Windows, use the operating system's copy command to copy the .dax file from the current IDE project directory to the desired location.

The generated class files from the mappings are not part of the exported package, and so must be generated when the other user imports the .dax file.

Chapter 8. Write Applications that Use Class Library Generated Code

Derive Your Own Classes from Generated Classes

You can derive your own classes from the generated classes. This is useful when you want to add functionality to the generated classes without defining the new methods in Data Access Builder user-defined methods support, which would require that you re-generated the code. Deriving a new class is preferable to editing the generated class code, because if you ever re-generate the classes from Data Access Builder, you will lose any changes you made.

In the following instructions, the example given is for extending a generated class, Employee, with a new class, myEmployee:

1. Create your derived class from Employee, as usual with the following code:

   ```
   class myEmployee: public Employee
   {
     // add new functionality here
   };
   ```

2. Create a derived manager with the following code:

   ```
   typedef EmployeeManagerTemplate<myEmployee> myEmployeeManager
   ```

If the derived object is for a Visual Builder application, then you derive the myEmployee class, as in Step 1, above. Then, to use the generated manager class, you must also provide feature code for a copy constructor of this new class (see Visual Builder help for deriving classes and generating feature code). The copy constructor code should be of the following form:

- In myEmployee.HPV:

  ```
  myEmployee( const myEmployee& aDerivedEmployee );
  ```

- In myEmployee.CPV:

  ```
  myEmployee::myEmployee( const myEmployee& aDerivedEmployee)
                       : Employee( aDerivedEmployee)
  {
    // your copy code here
  }
  ```

This is necessary because the Visual Builder does not support reference counted element members of sequences.

Connect and Disconnect

The classes generated by Data Access Builder help you access and manipulate information in one or more database tables. Before working with the data in an application, you must connect to the databases that your classes use. To do this, you must specify the database name (set the database name in either the code, as shown below, or in the mapping's properties notebook). Then connect, passing in user ID and password parameters if they are needed. You can also do this in one step, passing in the database name, user name, and password to the connect method.

The following sample code shows how to connect to the desired database:

```
theDeptDatastore.setDatastoreName (datastoreName);
theDeptDatastore.connect (userid, password);
```

or:

```
theDeptDatastore.connect (datastoreName, userid, password);
```

After connecting, specify settings for the datastore, such as whether it automatically commits transactions. In the following code, the database is set so that it does *not* automatically commit transactions:

```
theDeptDatastore.enableAutoCommit(false);
```

To disconnect from the database, run the following command:

```
theDeptDatastore.disconnect();
```

Commit and Roll Back

When you have made changes to database information using the Data Access classes, the changes are not permanent in the database until you commit them. Use the following command to set a database to automatically commit changes:

```
theDeptDatastore.enableAutoCommit();
```

However, since you may wish to verify changes before committing them to the database, you will probably want to pass `false` in to this method. **Note:** the autoCommit setting is on (`true`) by default.

When autoCommit is turned off, you must manually commit your changes using the `commit` method, as follows:

```
theDeptDatastore.commit();
```

This command will commit the current unit of work (all changes made since the last time you committed).

If you make changes and decide that you do not want to commit them to the database, use the `rollback` method to undo the current unit of work:

```
theDeptDatastore.rollback();
```

The `rollback`method is effective only if the autoCommit setting is turned off. You can enable or disable autoCommit at any point in your application.

Retrieve Rows from the Database

Similar to database cursors, a Data Access Builder cursoris a means of moving through a collection of row objects one at a time from top to bottom. It is a more memory-wise alternative to retrieving a collection of rows.

Using Select Method
When you retrieve a collection of rows using the `select` method, or your own method defined using SQL Predicate, a cursor is opened, the desired rows are fetched into memory, and the cursor is closed; after that, you access the fetched rows with items method. For example,

```
// Select all managers where phoneno is null
theDeptMgr.select( "PHONENO IS NULL" );

// Print each row
for(int i=0; i< theDeptMgr.items()->numberOfElements(); i++)
{
  cout << theDeptMgr.items()->elementAt(i)->forDisplay() << endl;
}
```

If the method retrieves many rows, they may take up a lot of memory.

Using fetchNext Method
With cursor-enabled access, you can open a cursor and leave it open. When you use fetchNext, the cursor fetches each row into a cached object, which is overwritten when it fetches the next object. Memory usage is minimal.

Using methods that the tool generates for managing multiple rows, fetch each row as you need it, for example:

```
// Open the cursor on the entire Department Manager database,
// ordering the records by department name
theDeptMgr.open( "ORDER BY DEPTNAME" );

// Repeat for each row
while( theDeptMgr.fetchNext() )
{
  // Get the row
  Department* theDept = theDeptMgr.element();
  // Print out the row
  cout << theDept->forDisplay() <<endl;
}
// close the cursor
theDeptMgr.close();
```

Use generated methods such as updateFetched and deleteFetched to update and delete rows, respectively. (These methods are not generated for read-only tables nor table-join schemas.)

You can also fetch a row directly into a PersistentObject object by using fetchedNextInto method. This allows you to directly access the attributes in the PersistentObject.

Using Vectors to Fetch Several Rows
To improve the efficiency of the code, use fill method to fetch a specified number of rows into a cached vector, process the rows, and then fetch the next set of rows. For example:

```
// Open the cursor on the entire Department Manager database,
// ordering the records by department name
theDeptMgr.open( "ORDER BY DEPTNAME" );
// Fetch 10 rows
int num = theDeptMgr.fill(10);
// Print each row
for(int i=0; i< num; i++)
{
  cout << theDeptMgr.items()->elementAt(i)->forDisplay() << endl;
}
// Also prints the actual number of rows retrieved
// The actual number of rows retrieved is returned by fill
cout << "number of elements is " << num << endl;
// close the cursor
theDeptMgr.close();
```

Each time you call the fill method, it overwrites what was previously in the cached vector.

The append method does the same thing as the fill method, except it does not overwrite the previously filled vector, but rather appends the next rows onto what is currently in the vector. You can also specify a specific vector for the append method to fetch methods into, as shown in the following example:

```
// Create a vector, listManager, to be used for fetched rows
ISequence<Department>    listManager;
// Open a cursor
theDeptMgr.open( "ORDER BY DEPTNAME" );
// Fetch 10 rows
int num = theDeptMgr.append( *listManager, 10 );
```

```
// Print each row
for (int i=0; i < num; i++)
{
  cout << listManager.elementAt(i)->forDisplay() << endl;
}
// Fetch 5 more rows
int num2 = theDeptMgr.append( *listManager, 5 );
// Print out the size of the listManager vector, which will be 15
cout << "Size of listManager vector: " << listManager->numberOfElements(); << endl;
```

See Data Access Class Library and Generated Class Reference for more information on generated classes and methods.

Set and Retrieve IDate, ITime, and ITimestamp Attributes

Data Access Builder supports mapping columns of SQL types DATE, TIME, and TIMESTAMP to attributes with C++ types IDate, ITime, and ITimestamp. See the related links to the data type conversion tables, below, for specific information on default and supported mappings of DATE, TIME, and TIMESTAMP.

If a <*class*> object has attributes of these types, it is important to know what format to use when setting these attributes, and what format to expect when retrieving their values. Data Access Builder uses the ODBC standard format for both getting and setting IDate, ITime, and ITimestamp attributes. The following table shows the format for passing in and retrieving these types:

SQL Type	**Default C++ Type Mapping**	**Get and Set Format**
DATE	IDate	YYYY-MM-DD
TIME	ITime	hh:mm:ss
TIMESTAMP	ITimestamp	YYYY-MM-DD hh:mm:ss.nnnnnnnnn

Note: YYYY = Year; MM = Month; DD = Day; hh = Hours (24-hour clock); mm = Minutes; ss = seconds; nnnnnnnnn = parts of a second.

For example, to set a TimeOfSale attribute for a Car object, you would call the following code:

```
Car.setTimeOfSale('14:26:00');
```

Write Data Access Applications with the Visual Builder

Visual Builder handles Data Access parts the same as it handles any other type of part. To create a Visual Builder application that can access a database, create the necessary Data Access parts by using Data Access Builder, and then load the needed parts into Visual Builder. The generated Data Access Builder parts are automatically Visual Builder-enabled.

Data Access Builder also provides some classes that you can use directly in applications, without generating any code. Load them into the Visual Builder to use in data access applications. All of these files are located in the include directory of the install image. The following Data Access .vbe files are provided:

- IDS.vbe - IDatastore; datastore connection for DB2 Embedded SQL.
- IDSB.vbe - IDatastoreBase; datastore connection abstract base class.
- IDSBCLI.vbe - IDatastoreBaseCLI; abstract base class for datastores supporting CLI/ODBC.
- IDSDB2.vbe - IDatastoreDB2; connection for DB2 databases.
- IDSODBC.vbe - IDatastoreODBC; connection for ODBC databases.
- IDAExcep.vbe - IDAException; Data Access Builder base exceptions.
- IDAMgr.vbe - IDAManager; abstract base for Data Access managers.
- IPObj.vbe - IPersistentObject; abstract base class for persistent objects.
- IPODID.vbe - IPODataId; abstract base class for persistent objects' data IDs.
- IVBDSODB.vbe - IDatastoreODBC with background thread support.
- IVBDSDB2.vbe - IDatastoreDB2 with background thread support.
- IVBDS.vbe- IDatastore with background thread support.

Two "connect canvas" parts are provided, also in the include directory. Drop one of them on your Visual Builder application when you want to have a dialog that lets the user connect to a database. Code for these parts is generated by the Visual Builder. The following Visual Builder parts are provided:

- idsccabs.vbf - IDSConnectCanvasBase; base class for a connect canvas using abstract IDatastoreBase.
- idsconcv.vbf - IDSConnectCanvas; a connect panel for IDatastore.

AIX The vbload.dat File

Create a file called vbload.dat in your project's working directory. In this file, list all the .vbe and .vbf files that the Visual Builder needs to load in addition to the base parts that it automatically loads. List all Data Access parts from the include directory:

```
usr/vacpp/include/IDS.vbe
usr/vacpp/include/IDSB.vbe
usr/vacpp/include/IDSBCLI.vbe
...and so on.
usr/vacpp/include/idsccabs.vbf
usr/vacpp/include/idsconcv.vbf
```

Then list all the generated parts from the mapping you created. For example, if the mapping was called Dept, then the list of files would look like this:

```
Dept.vbe
DeptDataId.vbe
DeptDataIdManager.vbe
DeptDataIdManagerBase.vbe
DeptDataIdManagerTemplate.vbe
DeptDatastore.vbe
DeptForm.vbf
DeptManager.vbe
DeptManagerBase.vbe
DeptManagerTemplate.vbe
```

Then start Visual Builder from the working directory with the command `ivb`. The list of loaded parts will show all of these parts from the vbload.dat file.

Each time you generate a new .vbe or .vbf file for the project, update vbload.dat with the file name so each time you start Visual Builder, the file is automatically loaded.

Data Access/Visual Builder Tutorial
Creating a Visual Data Access Application- Follow the instructions in this tutorial to create a basic data access application visually, by using some of the parts described above.

Tutorial: Create a Visual Data Access Application

In this tutorial, you will create a Data Access application by generating parts in Data Access Builder and in Visual Builder. The application connects to a database table called Department (or Dept), which contains records with information about departments in a company. You can see a summarized list of all the departments, and you can select a single department and view all the table’s columns of information for that department.

Preparing for the Tutorial

1. Create a working directory, `x:\IBMVAC\Working\datutorial`, where`x:\IBMVAC` is the directory where the product is installed.
2. If you want your application to use a DB2 database, create the DB2 sample database, if it is not created. To create the sample database, enter the following commands, one after the other, from a DB2 command line:

```
db2sampl
```

Note: If you choose to use ODBC, rather than DB2, you will use CPPASAMP40, the sample database provided with Data Access Builder.

Creating the Project and Target

1. Start the IDE. On the Welcome page, select **Create a new project** and click **OK**. If the Welcome page does not appear, then select **Create Project** from the **Project Workbook** menu.
2. In the Project SmartGuide, enter the project name `DataApp`. Set the project directory to `x:\IBMVAC\Working\datutorial`. Click **Next**.
3. Ensure the New Target setting is **Executable**. Click **Add Target** to launch the Target SmartGuide.
4. Enter the target name `DataApp`. Set the target directory to `x:\IBMVAC\Working\datutorial`.
5. In the Program Type list, select **Win IOC** or **OS/2 IOC**, depending on which operating system you are working on. Click **Next**.

Creating Parts for the Application

For this application, you will be creating three parts:

- a data access part
- a visual part
- a nonvisual part

Creating the Data Access Part

The data access part generated will provide access to a database table and also the actions needed to retrieve rows in the table.

1. In the New Part drop-down list, select **Data Access Part**. Click **Create Part**. This launches the Data Access SmartGuide.
2. On the Select Database and Mapping Method page, select either **DB2** or **ODBC**from the Database Source drop-down list.
3. If you selected **DB2**, select **SAMPLE** from the Databases list. If you selected **ODBC**, select **CPPASAMP40**.
4. Ensure the mapping method **By selecting database tables or views** is selected. Click **Next**.
5. If you selected DB2 for the Database source, select **DB2 Call Level Interface (CLI)** as the Access Method. Click **Next**. You will not see this page if you selected ODBC.
6. Click **Get Tables**. If you selected DB2, select the table *userid***.DEPARTMENT**. If you selected ODBC, select the table **DEPT**. Click **OK** to exit the Data Access SmartGuide and return to the Target SmartGuide.

Creating the Visual Part
The visual part will provide a window on which the GUI for the application sits.

1. In the New Part drop-down list, select **Visual Part**. Click **Create another part**.
2. Enter the Visual Part name `MainView`. Click **Next**.
3. Select the base class **IFrameWindow**. Click **Next**.
4. Select the client area **IMultiCellCanvas**. Click **OK** to return to the Target SmartGuide.

Creating the Nonvisual Part
The nonvisual part is the main code that launches the application.

1. In the New Part drop-down list, select **Nonvisual / Class Interface Part**. Click **Create another part**.
2. Enter the Nonvisual Part name `DataApp`.
3. On the Base Class page, select the **Nonvisual** radio button, and select **IVBMain**from the Base Class drop-down list. Click **OK**.

All three parts have now been added to the DataApp target. Click **OK** in the Target SmartGuide and **Finish** in the Project SmartGuide. When asked if you want to open the project in the IDE, click **Yes**. When asked if you want to build it, click **No**.

Modifying the Data Access Part
Go to the Data Access page in the Project tab. Select **Refresh View** from the pop-up menu in the upper pane. Select the data access part, called DATAAPP.DAX, and click **Edit Session**. This will launch the Data Access Builder main window.

If you were creating an application that required customized data access, then this is when you would customize the mapping to suit your needs. However, for this tutorial, the default mapping, with the following modifications, is sufficient:

1. Double-click on the mapping icon .
2. If you are using the DB2 sample database, SAMPLE, then change the mapping name so that the instructions that follow are easy to follow. Change both **Class Name** and **File Stem** from `Department` to `Dept`.
3. Ensure that the **Generate Forms** option is enabled.
4. Click **OK**.
5. Select **Save** from the **File** menu. When asked whether you want to remove the old .vbf objects from the configuration file, click **Yes**.

Close Data Access Builder. Go to the Parts page and refresh the parts view.

Adding the Connection Canvas Part to the Configuration File
The Data Access Class Library includes a class called IDSConnectCanvasBase that provides a visual front end to connecting to and disconnecting from a database. You need this class for the application, so it must be added to the configuration file. IDSConnectCanvasBase is in a source file called idsccabs.vbf, in the product's include directory. To add it:

1. Click the **Configuration** tab.
2. On the Source page, add the line `source "idsccabs.vbf"`, so that the list of source files looks like this (the added line is shown third from the bottom):

   ```
   source "DeptManagerTemplate.vbf"
   source "DeptManagerBase.vbf"
   source "DeptManager.vbf"
   source "DeptForm.vbf"
   source "DeptDatastore.vbf"
   source "Dept.vbf"
   source "DataApp.dax"
   source "idsccabs.vbf"
   source "MainView.vbf"
   source "DataApp.vbf"
   ```
3. Type Ctrl+S to save the configuration file.
4. Return to the Parts page in the Project and refresh the view.

Setting Up the Nonvisual Main Part
To set up the nonvisual main part that you created:

1. In the Parts page, select the nonvisual part, DataApp.
2. Click **Edit Part** to launch Visual Builder.
3. Select **Add Part** from the **Options** menu.
4. Enter the class name `MainView` (which is the visual part you generated). Click **OK**.
5. Click in the upper left-hand area of the Visual Builder canvas to place the window.
6. Click mouse button 2 on the canvas. Select **Connect**, **ready**.
7. Your pointer will now have a line attached to it. Click on the MainView part and select **Show**. A arrow points from the canvas to the MainView part, indicating that the nonvisual executable will open the MainView visual part.
8. Select **Save** from the File menu, and then exit.

Creating the Data Access Application GUI
The next step is creating the GUI that uses the data access parts to retrieve

database table contents. The GUI is built on MainView, the visual part you generated. In the upcoming steps, you will add the following parts to the main view:

- A connection panel that lets you click buttons to connect, disconnect, commit, and rollback.
- A message box that informs you if an error occurred.
- A list that shows all the departments in the database by department number and name.
- A form that shows all column values for a department when you select it in the list.
- The data access parts that let the GUI actually access and display the contents of the Department table.

To create the GUI and connect it to the data access parts:

1. In the Parts page of the IDE, select the visual part, MainView, and click **Edit Part**. The Visual Builder will start. The MainView part, with the Cancel connection made, is shown.
2. **Create GUI for connecting to the database:**
3.
   a. Select **Add Part** from the **Options** menu. Enter the class name `IDSConnectCanvasBase` and click **OK**. **Note:** An alternative to typing in the whole class name is clicking the **List** button. In the list, type in the first few letters of the class name, and then select it from the list and click **OK**.
   b. Click in the upper left-hand corner of the MainView window to place the part.
   c. Similarly add the part `DeptDatastore`, but drop it off the MainView window, on the canvas.
   d. From the new DeptDatastore part's pop-up menu, select **Connect**, **this**. With a line attached to your mouse pointer, click on the IDSConnectCanvasBase part. From the pop-up menu, select **All Features**. From the attributes list, select **datastoreBase**.

   The application can now connect to the Department table in the sample database, commit or rollback changes, and disconnect from it, all by clicking the buttons on the connection canvas. You will see this in action later, when you run the application.
4. **Create GUI for listing the departments by department number and name:**
5.
   a. From the Visual Builder parts pallet, select the Lists category, and the IVBContainer part.

b. Drop it on MainView, just below the connection canvas. Stretch the part so that it fills up the space below the connection canvas.

c. Double-click on the new container. In the Properties list, click on **viewType**. Select **showDetailsView** from the drop-down list.

d. Click mouse button 2 on the IVBContainer part and select **Change Template Arguments** from the pop-up menu. Select the **class Element** item and type `Dept*` in the entry field. Click **OK**. This sets the table to be list of pointers to Dept objects (rows from the Department table).

e. From the pallet, select the IContainerColumn part and click on the IVBContainer on MainView. Select it again, and drop another column on the container. The container now has two columns, one for department numbers, and the other for department names.

f. Double-click on the left-hand column.

g.

   1) Select the **Column Definition** property and click the **...** button.

   2) Select the **Use an attribute from the part** radio button, and select **deptNoAsString**from the Attributes list. Click **OK**.

   3) Select the **headingText** property and enter `Dept No`. Click **OK**.

   The column is now set to display deptno attributes (department numbers) from Dept objects.

h. Double-click on the right-hand column.

i.

   1) Select the **Column Definition** property and click the **...** button.

   2) Select the **Use an attribute from the part** radio button, and select **deptnameAsString**from the Attributes list. Click **OK**.

   3) Select the **displayWidth** property and enter `200`.

   4) Select the **headingText** property and enter `Dept Name`. Click **OK**.

   The column is now set to display deptname attributes (department names) from Dept objects.

6. **Connect the list GUI to the data access part that accesses rows in the table:**

7.

   a. Select **Add Part** from the **Options** menu. Enter the class name `DeptManager` and click **OK**. Click on the canvas to drop the part.

   b. From the new DeptManager part's pop-up menu, select **Connect**, **items**. With a line attached to your mouse pointer, click on the IVBContainer object. Be very careful not to click on one of the columns; the red dotted line should highlight the entire IVBContainer part. From the pop-up menu, select **items**.

8. **Create connection to show list departments when a successful database connection is made:**
9.
   a. From the pop-up menu for the DeptDatastore part, select **Connect**, **All Features**. From the Events list, select **Connected**. With a line attached to your mouse pointer, click the DeptManager part. From the pop-up menu, select **select**.
   b. The connection line that you just made is dotted because the select method requires a parameter: an SQL statement that indicates which rows to select. Double-click the dotted line and click **Set Parameters** button.
   c. Select the clause parameter and enter the following string, which will select all rows in the table and display them in order of department number:

      ```
      1=1 order by deptno
      ```
   d. Click **OK** twice.
10. **Handle exceptions:**
11.
    a. If an error occurs while the application is connecting to the database or retrieving the rows, you need to pass the error message on to the user. Add a message box for the error message by selecting the Other category on the pallet. Select the IMessageBox part button and drop the part near the connection you just made (on the canvas).
    b. Click mouse button 2 on the connection line. From the pop-up menu, select **Connect**, **exceptionOccurred**. With a line attached to your mouse pointer, click the new IMessageBox part. From the pop-up menu, select **showException**.
12. **Create a form that shows details of a selected department:**
13.
    a. To add the data access form part in this step, you may have to stretch the MainView part so that everything will fit in. Click on the MainView window's title bar, and then grab the black square at the bottom right-hand corner of the window. Drag it down and to the right to make the window larger. You can also move the nonvisual parts (DeptDatastore, DeptManager, and MessageBox) by clicking and dragging them. The connections will remain.
    b. Select **Add Part** from the **Options** menu. Enter the class name `DeptForm` and click **OK**. Drop the part in the right-hand section of the MainView window, next to the IVBContainer. The form shows each of the Department table's columns as an attribute. When the application runs, the fields will show the column values for the department selected in the IVBContainer.

   c. Select the IVBContainer (be sure not to select one of its columns). From its pop-up menu, select **Connect**, **selectedItem**. With a line attached to your mouse pointer, click the new DeptForm part. Select **All Features** from the pop-up menu. From the Action list, select **targetAsDept**. Click **OK**. The connection line created is very short, between the IVBContainer and the DeptForm part.

14. You have now created the entire GUI and made all necessary connections. Select **Save and Generate**, **Part Source** from the **File** menu. Exit Visual Builder.

Building and Running the Visual Data Access Application
In the IDE, click the **Save all changes and build** button in the toolbar. The build process may take a few minutes, depending on your system.

To run the application from within the IDE, go to the Debug page of the Project, and click the **Run** button. To run it from the VisualAge for C++ command line, go to `x:\IBMVAC\Working\datutorial` and type `DataApp`.

Browsing the Application
The application you just created has a graphical front end to a real database table. If you had implemented an update feature, changes made in the application would actually change the content of the table. This application, however, only lets you look at the contents of the table. The database table itself contains records for departments in a company. The application shows a list of all the department names and numbers, ordered by number. When you select a department, the application shows all column values for that department, including the manager's employee number, the administrating department, and the department's location.

To connect to the database and retrieve the list of all departments, click **Connect**. If you are not logged in to your system under the same user ID with which you created the sample database, fill in the **User ID** and **Password** fields; use a user ID that has access to the sample database.

In the list of departments, select one by clicking on it with the mouse or by using the arrow keys on the keyboard. The contents of the selected department are displayed in the form on the right.

When you are done, click **Cancel**. In the tutorial steps, you did not provide functionality for the **OK** or **Help** buttons, so they do nothing.

Prepare a Target Machine for Running a Data Access Application

Once you have completed building the Data Access part into your DLL or EXE target, you may want to install the Data Access application on a target machine (that is, a machine other than the one on which you wrote the application).

The target machine must be able to access the database that was specified in the part, and it must have all the appropriate authorities on that database so that it can run the methods that the application calls. So, for example, if you specified that the part was to use ESQL to access a DB2 database, then a DB2 client that can access that database must be installed on the target, and the client must have access to the tables that the application uses.

Also, install all runtime class library DLLs that the application uses, and the application DLL or EXE itself.

Created in Canada