

Working with the Version 9 ADM: Using SmartDataObjects from Outside the ADM

John Sadd
October 2000

This document is one of a series on version 9.1 ADM development topics.

Copyright © 2000, Progress Software Corporation
14 Oak Park, Bedford, MA 01730.
All rights reserved.

THIS DOCUMENT IS A DRAFT AND IS KNOWN TO BE INCOMPLETE. IT MAY NOT BE COPIED OR REDISTRIBUTED WITHOUT THE EXPRESS WRITTEN PERMISSION OF PROGRESS SOFTWARE CORPORATION.

This document is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular use, or non-infringement. This document may contain technical inaccuracies, typographical errors, or other errors. Progress Software Corporation makes no commitment to deliver any new products, nor any enhancements to any existing products, incorporating the technologies, programs, products, or product features discussed in this document.

Progress Software Corporation may make changes and/or improvements to the information herein. Such changes and/or improvements may be incorporated into new editions of this document. Progress Software Corporation may make new improvements and/or changes to the product(s) and/or the programs described in this document at any time.

1. Overview

The ADM and the SmartObjects supplied with the product provide a framework within which it is possible to develop new applications in which a great deal of the standard behavior is taken care of, so that the developer can concentrate on writing 4GL code specific to the application. However, the ADM is not a closed environment, and it is quite possible to use individual SmartObjects together with non-ADM Progress code. In particular, the SmartDataObject (SDO) has an API which makes it quite useful as a general data management object, callable from non-ADM 4GL, or from outside Progress, using the Open4GL interface and ProxyGen.

This paper will describe the API of the SmartDataObject from the perspective of using it outside the context of SmartObjects. In doing so, it is not always possible to be precise about exactly what entry points and properties are "intended" to be used from an open interface. A minority of entry points in the SDO are clearly intended to be used only internally (to be called only from other SDO routines), and these are documented as such. Other internal procedures or functions are useful only in the context of an ADM-based application. And some of the SDO properties are likewise useful only in a SmartObject application, for example, those which hold handles of other objects connected by SmartLinks.

But there is necessarily a gray area between those routines which are basic to any use of the SDO and those which could be considered part of an "advanced" API to the SDO. This paper will try to be reasonably complete in describing entry points that can be used from outside the ADM, while emphasizing those which are most basic to using SDOs at all. The reader should keep in mind that the complete API of the SDO and other SmartObjects is part of the standard product documentation and on-line help; in addition, of course, all of the source is available for study, and the headers of internal procedures and functions contain useful information on the purpose of each routine (generally this is similar information to what is in the appendix to the reference documentation on the ADM, as well as in the on-line help). If you think you need something which is not described here, check the doc and the code before assuming with certainty that it does not exist.

Also, although some aspects of the API have been created with “external” use in mind, the large majority of the development effort thus far for SDOs, as for other SmartObjects, has been to support their use in the context of SmartObject applications. Thus the API is not as complete or as simple to use in all cases as it might be, and ought to be. It is taken for granted that an understanding of the existing API can and should be the starting point for building one or more additional layers or “drivers” on top of the SDO to optimize its use from various environments. One such driver, the “Java SDO”, has already been created to simplify the use of SDOs from Java, through the Open4GL interface.

For an understanding of what SDOs do and how the SDO template is designed, the reader is referred first of all to the product documentation in the ADM2 Developers Guide, and also to the ADM2 white paper on the SDO available through the Progress web site (at www.progress.com/adm). The SDO white paper in particular describes the structure and use of the temp-tables which manage the data for display and update, and how the data moves from server to client and back in a distributed application.

2. Starting and Stopping an SDO

The SDO uses the same basic methods as any other SmartObject to start and stop itself. Any procedure using an SDO will need to make use of these. First of all, an SDO is designed to be run persistently, so that its internal entry points can then be run individually. So the program using the SDO must run it PERSISTENT, save the returned procedure handle, and run all other internal procedures and functions in that handle.

When the SDO is first run, it will establish a chain of Progress super procedures which define its standard behavior. These are the procedures smart.p, query.p, data.p, and dataext.p. These will be run automatically as persistent procedures, if they are not already running, and the SDO then makes them its super procedures, so that it can inherit their behavior. Smart.p contains behavior common to all SmartObjects. The methods described in this document which are specific to SDOs are found in query.p, data.p, or dataext.p. Starting the SDO also creates a temp-table record containing a set of properties for the SDO. The properties which are relevant to using the SDO from outside the ADM will be described in this document.

These methods are basic to using SDOs:

- **initializeObject** – (internal procedure, no arguments) – Apart from possibly setting one or more initial property values, the first method to run in the SDO is *initializeObject*. This will check to see if there is an AppService defined, and if so, will connect to the AppServer represented by that Service or Partition name, and run a copy of the SDO on that server (details of how this works are in other documents). It will also open the database query defined for the SDO (unless otherwise specified), populate the client-side RowObject temp-table with the first batch of rows from the database, and do a *fetchFirst* operation to position the temp-table query to its first row.
- **destroyObject** – (internal procedure, no arguments) – this is the destructor for SmartObjects, and will delete the SDO procedure. If the SDO is divided between client and AppServer, it will delete the server-side procedure and disconnect from the AppServer.

3. Accessing Data One Record at a Time

The SDO manages a dataset of rows representing the results of a database query, and any updates to that data, in a pair of temp-tables called RowObject and RowObjUpd. In most cases an application wishing to make use of the SDO can and should access the data through the API without ever making direct reference to these temp-tables. This simplifies the programming in most cases. This section will describe the basic 4GL entry points which can be used to accomplish this.

- **openQuery** (function, no arguments) – The database query is opened automatically when the SDO is initialized unless the **OpenOnInit** logical property is set to false. The *openQuery* function can be used at any time to refresh the dataset by reopening the database query. The latest Where clause (if it has changed) will be prepared, the query opened, and the client-side temp-table populated.
- **fetchFirst, fetchNext, fetchPrev, fetchLast** – (internal procedures, no arguments) – These procedures can be run to reposition the client-side temp-table query as the names suggest. Additional batches of rows will be fetched from the server database query automatically, as needed.
- **rowAvailable** – (function, INPUT pcDirection, RETURNS LOGICAL) – This function tells the caller whether the SDO has a row available or not. The pcDirection parameter can be set to “NEXT” (or “PREV”), in which case the function will return true or false to indicate whether another row is available to be positioned to *after* (respectively *before*) the current row, or it can be set to “CURRENT” (or blank or unknown) to find out if there is a current row available. The function can be used, for example, to manage a construct such as a *fetchNext* loop which needs to walk through all the records in the dataset. Note that, as with other SDO functions, batches of rows are retrieved into the temp-table as necessary, transparent to the caller; so a call to *rowAvailable('NEXT')* will return TRUE if the temp-table is currently on its last row but there are still more database rows to be added to it. It only returns FALSE when the last row of the database query is reached.
- **colValues** – (function, INPUT pcViewColList, a comma-separated list of column names whose values are wanted; RETURNS CHARACTER) – This is the basic method to use to retrieve multiple values for the current client-side row. The RETURN value is a CHR(1)-delimited list of those values, in CHARACTER form, but without the field FORMAT applied. Values are in character format to allow them to be returned as one parameter. The first value returned is always the RowIdent value for the current row, which is a list of the ROWIDs of the database records from which it is derived. This can generally be discarded.
- **columnValue** – (function, INPUT pcColumnName, RETURNS CHARACTER) – to retrieve the value of a single column, this function can be used. The value is returned in CHARACTER format (to satisfy the requirements of the function declaration), but unformatted. As an alternative, use:
- **columnStringValue** – (function, INPUT pcColumnName, RETURNS CHARACTER) – returns the formatted string value of the requested column.

The SDO is built with a starting query definition, naming the database table(s) to be used and the columns to be used in the SDO temp-table. Optionally, a Where clause can be specified. At runtime, the query Where clause can be modified using several different methods. For a more complete discussion of customizing the SDO Where clause, see the ADM2 *WhereClause* white paper.

- **setQueryWhere** – (function, INPUT pcWhere, RETURNS LOGICAL success flag) – This is the simplest of the methods to manipulate the where clause. The input parameter can be a logical expression to be added to the base query definition, or blank to reset the query to its initial definition, or a complete FOR EACH statement to specify a complete new Where clause for the query (the list of tables cannot be changed). However, this function does not allow a Where clause to be built up in stages, and does not optimize a Where clause involving multiple tables. It causes any previous changes to the query, including any Sort (BY) phrase, to be replaced with the new expression. The following methods are more flexible and efficient in building or modifying a Where clause:
- **assignQuerySelection** – (function, INPUT pcColumns, INPUT pcValues, INPUT pcOperators, RETURNS LOGICAL success flag) – This function takes a comma-separated list of column names, and a matching CHR(1)-delimited list of values for those columns, as input parameters, along with an optional list of operators: by default the operator for each column-value pair is EQ, otherwise a list of operators for each pair can be specified in this parameter. The individual expressions (Column Operator Value) are added to the

QueryString property which is used to build the Where clause on the client side of the SDO. If the query involves a join, then each expression will be properly associated with the corresponding table expression in the query definition. When the openQuery method is invoked, the Where clause will be sent to the server-side SDO automatically, prepared, and the database query opened and the RowObject temp-table repopulated. Note that the QueryString property is not intended to be modified directly; it is for internal use by these and other functions only.

- **addQueryWhere** – (function, INPUT pcWhere, pcBuffer, pcAndOr, RETURNS LOGICAL success flag) – This alternative to assignQuerySelection allows you to specify a complete expression at a time, along with an optional Buffer parameter specifying which table the expression is to be applied to, and an optional specification of whether the expression should be joined to others by 'AND' or 'OR' (the default is 'AND'). The same steps occur when the query is reopened as for assignQuerySelection.
- **removeQuerySelection** – (function, INPUT pcColumns, INPUT pcOperators) – removes expressions for one or more columns which were previously added using assignQuerySelection.
- **setQuerySort** – (function, INPUT pcSort, RETURNS LOGICAL success flag) – This function can be used to append a BY phrase to the end of the query.

These really constitute the basic methods to be used to initialize and read data through an SDO. A number of other methods can be used for more specialized operations. These include a set of functions to return specific attributes of a column in the SDO. Each of these takes the column name as an INPUT parameter, and returns the requested attribute:

- **columnDataType**
- **columnDbColumn** (returns the name of the database column this temp-table column maps to – the name could be changed in the definition of the SDO)
- **columnHandle**
- **columnQuerySelection** (returns any Where clause expressions for this column which have been added using *assignQuerySelection*)
- **columnTable** (returns the database table this column came from)
- **columnValExp**
- **columnValMsg**
- **columnLabel**
- **columnColumnLabel**
- **columnHelp**
- **columnInitial**
- **columnModified**
- **columnPrivateData**
- **columnReadOnly**
- **columnWidth**

There is also a function to return multiple properties in a single call:

- **columnProps** – (function, INPUT pcColList, INPUT pcPropList, RETURNS CHARACTER) – This function takes a comma-separated list of columns plus a comma-separated list of properties as input parameters and returns a formatted list of those property values for those columns.

Ordinarily the SDO retrieves additional batches of rows automatically as needed to satisfy fetchNext or fetchLast requests. This method can be used to append an additional batch of rows to the current client-side table:

- **fetchBatch** – (internal procedure, INPUT plForwards) – This internal procedure takes a logical argument which signals whether an additional batch of rows should be retrieved moving forward from the current set (if TRUE) or backwards (if FALSE). The batch is

Working with the Progress Version 9 ADM: Using SmartDataObjects from Outside the ADM
Copyright 2000 Progress Software Corporation

retrieved and appended to the current client-side dataset without changing the current record position. This procedure is currently used by the OFF-END and OFF-HOME triggers of the SmartDataBrowser, to allow it to browse multiple batches as transparently as possible.

This function can be used to refresh the current row from the database:

- **refreshRow** – (internal procedure, no arguments) – This re-retrieves the current temp-table row from the database, in case it has been modified by some other program since it was read.

These functions can be used to reposition the temp-table query to a particular row:

- **findRow** – (function, INPUT pcKeyValues, RETURNS LOGICAL success flag) – There is a **KeyFields** property which holds the list of fields which make up the key which should be used by default to retrieve specific rows. This property is automatically initialized to the fields which make up the primary index of the database record (if there is only one) from which the SDO is derived. The INPUT parameter is a comma-separated or CHR(1)-separated list of values for those fields. The temp-table query is repositioned to that row. Or, as an alternative:
- **findRowWhere** – (function, INPUT pcColumns, INPUT pcValues, INPUT pcOperators) – Using the same parameters as assignQuerySelection, this function allows the query to be repositioned to a row matching *any* value(s) for any column(s), not just the Key Fields.

To populate a combo box or other object with all the values from a small dataset, this function can be used:

- **rowValues** – (function, INPUT pcColumns, INPUT pcFormat, INPUT pcDelimiter, RETURNS CHARACTER) – This special-purpose function returns all the values for the specified column(s) for all rows of the dataset, so it is appropriate for populating a Combo Box or other object which displays a limited set of values. The values will be retrieved from the client-side temp-table, which was populated when the SDO was initialized. See the code in data.p for details on the formats of the parameters.

4. Update Methods

The following methods can be used to update or delete the currently selected row in the dataset, or to create a new row. By default, each operation will be saved to the database in its own transaction. Alternatively, the logical property **AutoCommit**, which is true by default, can be set to false to cause change operations to be accumulated on the client until an explicit Commit or Undo operation is performed.

These first two functions are “state” methods, in the sense that they cause a new row to be created locally (just at the level of the client-side temp-table), but do not initiate any change to the database, even if AutoCommit is true. This enables the application to display initial values for a new row to the user, and then save to the database any changes the user makes to those initial values.

- **addRow** – (function, INPUT pcColList, RETURNS CHARACTER) – This function accepts a comma-separated list of column names, and creates a new row in the client-side temp-table, setting its columns to their initial values, and returning the initial values for the requested columns. As with the *colValues* function, the first value in the return list is the RowIdent for the new row; this can be discarded. Note that nothing goes back to the database until the *submitRow* function is invoked.
- **copyRow** – (function, INPUT pcColList, RETURNS CHARACTER) – This function is similar to *addRow*, except that it uses the values for the currently selected row as the initial values for a new row.

The remaining functions are “transactional”; if *AutoCommit* is true, the values will go back to the database. Thus, in effect, *createRow* is a substitute for *addRow* + *submitRow* for cases where the initial values do not need to be displayed in the client application.

- **createRow** – (function, INPUT pcValueList, RETURNS LOGICAL success flag) – This function takes a CHR(1) – delimited list of alternating column names and values and causes a new database row to be created with those values. (As noted, the database row will be created in this operation only if *AutoCommit* is true; otherwise the newly created row will be held in a client-side temp-table until Commit).
- **submitRow** – (function, INPUT pcRowIdent, INPUT pcValueList, RETURNS LOGICAL success flag) – The RowIdent parameter is not required and can be left unknown (or the RowIdent returned by *addRow* can be passed back). The ValueList is the usual CHR(1)-delimited list of alternating column names and values. This function will do a “Save” operation; if *AutoCommit* is on, the newly added or modified row will go back to the database.
- **updateRow** – (function, INPUT pcKeyValues, INPUT pcValueList, RETURNS LOGICAL success flag) – This function accepts an optional comma-or-CHR(1)-delimited list of values for the *KeyFields*, if a specific row needs to be retrieved to be updated; otherwise the pcKeyValues parameter is left blank and the currently selected row in the client-side dataset is updated. The pcValueList is the usual CHR(1)-delimited list of alternating column names and values, in CHARACTER form. This does the same sort of “Save” operation as *submitRow*, but the parameters are more convenient for most uses (because the RowIdent argument is not used, and the KeyValues can be handy for repositioning to a different row).
- **deleteRow** – (function, INPUT pcRowIdent) – As elsewhere, the RowIdent is optional; either the unknown value or the RowIdent of the current row can be passed in to delete the current row; or the RowIdent of another row in the RowObject table can be specified to reposition to and delete that row. The row will be deleted from the client-side temp-table, and if *AutoCommit* is true, from the database.

To cancel an Add or Copy operation which has not yet been saved (by *submitRow*), this function is used:

- **cancelRow** – (function, no arguments) – This removes the new row from the client-side temp-table, effectively canceling the Add or Copy operation. This must be done before *submitRow* is done (or it can be done if *submitRow* fails and the program wants to cancel the Add/Copy). If the *AutoCommit* property is false, so that changes are not automatically saved to the database, then the *undoTransaction* procedure must be used to undo uncommitted changes.

If the *AutoCommit* property is false, these procedures are used to Commit or Undo a set of changes:

- **commitTransaction** – (internal procedure, no arguments) – This procedure sends all modified rows from the client back to the server to be changed in the database in a single transaction.
- **undoTransaction** – (internal procedure, no arguments) – This procedure undoes all uncommitted changes in the client-side temp-tables.

These functions can be used to handle error messages generated by a database operation:

- **anyMessage** – (function, no arguments, returns LOGICAL) – This function can be run for example after a *commitTransaction* to see if there were any errors generated by the operation. It returns true if there are any error messages.
- **fetchMessages** – (function, RETURNS CHARACTER) – Returns a delimited list of any error messages generated by the most recent database operation. See documentation or code for precise format.

5. Running the SDO Proxy on the Client

These descriptions make reference to “client-side” and “server-side” operations. The standard doc and white papers describe this in much more detail, but to summarize, an SDO can be run on an AppServer from a non-Progress application using the Open4GL interface. In this case, all SDO operations happen on the “server side”. An SDO can also be run in a Progress 4GL client session which does not have a connection to the database the SDO uses. In this case, the SDO automatically runs a copy of itself on the associated AppServer where the database connection is available, and sends data back and forth between client and server transparently. In this case, it is necessary to run the SDO “proxy” procedure on the client. This is an alternative compilation of the SDO, with the name <SDOName>_cl.r, and with any references to the database compiled out. Creating and compiling this alternative version is handled automatically when SDOs are saved from the Progress AppBuilder.

In order to achieve this client-to-AppServer separation, then, the client-side 4GL code must run the SDO proxy rather than running the “full” SDO .r file. It must also supply, at a minimum, the name of the AppService as a property, as described in the Properties section below. The SDO proxy will then handle the AppServer connection automatically, along with all of the transfer of data from server to client and back.

As an alternative, a 4GL application can take advantage of the transparency of the client-AppServer distribution just as a SmartObject-based application does. This requires running the *constructObject* procedure (which is in the super procedure *adm2/containr.p*) instead of running the SDO directly. The code in *constructObject* determines whether the proxy or the full SDO should be run, allowing the application to behave identically with or without a separation between 4GL client and AppServer. Using the *constructObject* procedure requires that the parent application procedure include the ADM include file *src/adm2/containr.i*, which causes the super procedures *smart.p* and *containr.p* to be started and made super procedures of the application. [It would be straightforward to modify this code such that all of *smart.p* and *containr.p* do not need to be run in support of a non-SmartObject application, if this were desired.] The calling application should also make the definition

&SCOPED-DEFINE ADM-CONTAINER VIRTUAL

ahead of the reference to *containr.i*, in order to compile out code for visual objects. Here’s the description of *constructObject* itself:

- **constructObject** – (internal procedure, INPUT pcProcName, INPUT phParent, INPUT pcPropList, OUTPUT phObject) – PcProcName is the filename of the SDO to run, appended with CHR(3) + ‘DB-AWARE’. This tag signals to *constructObject* that this is an object which has database references and needs to be run in proxy form on the client if the required databases are not connected. *ConstructObject* takes care of determining whether the required database is connected; alternatively, if the SDO is not present *at all* on the client, that is, neither the full SDO nor the proxy, then *constructObject* will automatically run an instance of the Dynamic SDO procedure (*dyndata.r*) on the client and set its properties so that it will in turn run and communicate with the full SDO on the AppServer. The phParent argument is for visual objects and can be passed as the unknown value. The pcPropList argument gives the caller the opportunity to set one or more Instance Properties of the SDO at the time it is created. These properties are described below; Instance Properties which can be set at this time include the AppService name, and the number of RowsToBatch for each transfer of data from server to client. The format for this is argument is <Property> CHR(4) <Value> CHR(3)... The procedure handle to the newly created (client-side) SDO is returned as phObject.

6. SDO Instance Properties

As noted in the method descriptions, some of the properties defined for the SDO are relevant to using it from an “open” interface. Those which can be assigned values before the SDO is initialized will be described here. Again, the on-line help and standard ADM doc describe all of the SDO properties. The ADM convention for property access is that they are set by running a function called *set<propname>*, which accepts the property value as a single INPUT parameter of the appropriate datatype, and returns a LOGICAL success flag. Alternatively, the Instance Properties of the SDO can be set by using the *constructObject* procedure. The Instance Properties are those properties which are intended to be assigned initial values that will affect how the SDO starts up. Other properties can be read or set by the calling application to get or set information about the SDO (such as its Column list or the handle of one of its internal data structures). To read a property value, the application executes a function called *get<propname>*, which takes no parameters and returns the value of the property. These additional properties are described in a later section. So here are the Instance Properties:

- **AppService** – This CHARACTER Instance Property is the logical Partition name for the Progress AppServer on which this SDO should run (where the database connection is available). If you are writing a Progress 4GL application which wants the SDOs to run on one or more AppServers, then this property must be defined before the SDO is initialized (this can be done in a call to *constructObject*, as described above, or by setting the property after running the SDO but before running *initializeObject*. If this is not defined, then the SDO runs only in the client application (and all references to the “server-side” in the descriptions above essentially “collapse”, that is, all of the database-related code is executed in the client session, requiring a database connection in that Progress session). If the SDO is being run from a non-Progress environment, or if the client proxy is otherwise not being used, then the SDO must be run directly on the AppServer by the client code. In this case, the AppService property is not used. Note that there are two other instance properties which can be used to pass values to the standard Progress AppServer connect method: ASUsePrompt, and ASInfo. If your application needs to set these parameters, which respectively tell the AppServer whether to prompt for userid, and pass other information to the connect request, then setting these Instance Properties will do that for you. See the AppServer documentation for more details.
- **RowsToBatch** – This INTEGER Instance Property sets the number of rows to be retrieved from the database and loaded into the SDO temp-table at a time. The default value is 200. This property can be set to change that default.
- **CheckCurrentChanged** – This LOGICAL Instance Property sets a flag which signals to the update code in the SDO whether it should reject a change to a record which has been modified by another user since it was originally read by the SDO. The default value is TRUE, meaning that such a conflict will be rejected.
- **RebuildOnRepos** – This LOGICAL Instance Property tells the client side of the SDO whether it should empty its RowObject temp-table and start over if the database query is repositioned to a row that was not already in the temp-table, through an operation such as *fetchLast*. This can make jumping to the last record in large datasets faster (dramatically so if the dataset is large), at the expense of maintaining a contiguous and growing copy of the dataset in the SDO temp-table. The property’s default value is FALSE, meaning that the temp-table dataset will be built up from beginning to end, as needed.

7. Linking Multiple SmartDataObjects

It is worth noting that there is an additional Instance Property called **ForeignFields** which is used in a SmartObject application to link multiple SDOs together in a parent-child relationship. This relationship requires the use of the Data SmartLink. Generally this document is describing the use of SDOs outside the scope of a SmartObject application, so any discussion of links has been avoided. However, as an illustration of how the individual elements of the ADM can frequently be used without requiring the entire application to be built from SmartObjects, this section will

describe how to link multiple SDOs and use the ForeignFields property to pass key fields from one to the other.

Let us suppose that the application wants to associate two SDOs, one which browses Customers (dcust.w) and one which browses their Orders (dorder.w), such that the Order SDO will automatically re-open its query for Orders of the current Customer whenever the Customer SDO is repositioned. To do this, the parent application must do the following:

- Run the Customer SDO and save its handle (lets call it hCust).
- Run the Order SDO and save its handle (hOrder).
- RUN addlink IN hCust (hCust, 'Data', hOrder).
- DYNAMIC-FUNCTION('setForeignFields' IN hOrder, 'Order.CustNum,CustNum').

The call to addLink will create a Data link between the two SDOs. The call to setForeignFields will associate the CustNum column in the Customer SDO with the database field for CustNum in the Order SDO, so that its database query can automatically be changed and re-prepared each time the Customer SDO is repositioned.

Now when the application is run, any operation which changes the current customer will cause the Customer SDO to send a message to the Order SDO to change, re-prepare, and re-open its query.

8. Other Useful SDO Properties

The preceding section described the SDO instance properties, which can be assigned initial values. Other properties can be useful to read (or in some cases set, as noted) during the execution of the application. As always, see the documentation for more complete descriptions of these and other properties. These include:

- **ASHandle** – the procedure handle of the SDO procedure running on the AppServer (if any). This can be used to run methods directly in the AppServer session.
- **AutoCommit** – true by default, it can be set to FALSE to allow multiple updates to be saved on the client before being committed to the database. By default each add/update/delete operation is sent to the server as its own transaction.
- **DataHandle** – the handle of the client-side temp-table (“RowObject”) query. The example in the next section shows how this can be used to attach a browse to this query.
- **KeyFields** – the key database field names to be used to retrieve or reposition to a record using a method such as findRow. This property is set by default to the fields in the primary key of the database table from which the SDO is built, if there is no join in the SDO definition (using the primary index if unique, otherwise the first unique index). It can be reset if needed.
- **NewRow**— This LOGICAL property will return TRUE if an Add or Copy is pending (following a call to *addRow* or *copyRow*, but before *submitRow*, in other words), otherwise FALSE.
- **QueryWhere** – this property holds the where clause for the query. As noted, it can be set using the setQueryWhere function as shown in the example below. But generally the addQueryWhere and assignQuerySelection functions will manipulate the query more flexibly and efficiently.
- **RowObject** – the handle of the temp-table buffer in the SDO. This can be used if it is necessary to reference the buffer and its fields directly, though methods are provided do this without requiring access to the buffer itself.
- **RowObjectTable** – the handle of the RowObject temp-table itself. Again, this can be used if it is necessary to reference or manipulate the temp-table directly, but generally this should be avoided.
- **RowObjUpd** – the handle of the temp-table buffer where updates are stored before being returned to the database.
- **RowObjUpdTable** – the handle of the RowObjUpd temp-table itself.

- **DataColumns** – comma-separated list of the columns in the SDO.
- **ForeignFields** – foreign key fields for which values should be retrieved from a parent SDO to provide an expression to be used in defining the child SDO’s query, as shown in the example below.
- **ForeignValues** – list of the current values of the ForeignFields.
- **OpenOnInit** – logical flag to signal whether the SDO’s query should be opened automatically when it is initialized. True by default, it can be set to FALSE to suppress opening the query until wanted.
- **Tables** – list of the database table names from which the SDO is derived.
- **UpdatableColumns** – list of the SDO columns which are enabled for update.

9. Example Program

This sample code shows some of the SDO methods and properties in a non-SmartObject example procedure. This is absolutely *not* intended to be seen as an example of appropriate programming of SDOs. As noted at the beginning of this document, it is expected that a development organization would build an SDO “driver” appropriate for their circumstances, just as the SmartObjects such as the SmartDataViewer and SmartToolbar act as a form of driver in a SmartObject application, hiding the specifics of the programming interface from individual application components. The example is intended rather to be a simplified illustration of some of the SDO API in action.

The sample procedure demoSDOWin.w puts up a window that looks like this:

Cust Num	Order Num	Ordered	PO	Sales Rep
1	6	09/19/00	6-1 <98> [1919.09] <	BBB
1	36	09/10/00	36-1 <140>	HXM
1	79	09/10/00	79-1 <12> [34754.41	HXM
1	177	09/03/00	177-1 [4605.91]	HXM
1	185	09/11/00	185-1 <6> [951.62]	HXM
1	1335	02/04/98	1335-1 <2> [65]	HXM
1	1340	09/19/97	1-1340 <42> [1059.5	Web
1	1350	01/01/98		Web
1	1390	10/16/97		Web
1	6050	01/29/98	PO0603976050	BBB
1	6055	03/02/98	PO0604976055	BBB

This procedure is a standard Window procedure (*not* a SmartWindow) as built in the AppBuilder. It runs two SDOs, dcust.w to browse Customers, and dorder.w to browse Orders. It displays the Customer Name from the Customer SDO and the CustNum and OrderNum from the Order SDO. It also displays a browse control which is connected to the temp-table query of the Order SDO. The Definitions section of the procedure defines variables to hold the procedure handles of the two SDOs:

```

/* ***** Definitions ***** */
/* Parameters Definitions --- */
/* Local Variable Definitions --- */

DEFINE VARIABLE hCust AS HANDLE NO-UNDO.
DEFINE VARIABLE hOrder AS HANDLE NO-UNDO.

```

The Main Block of the procedure, after the standard code (part of the Window template) to assign CURRENT-WINDOW and set up a trigger to run the disable_UI destructor, creates and initializes the SDOs and the browse. The browse is created in the AppBuilder by selecting a browse control, dropping it onto the design window, and canceling out of the Table selection in the browse wizard. In this way a static browse with no table, no query, and no columns is created. The Main Block code below associates the browse with the SDO query, and then defines the columns to be displayed. At the end of the Main Block, we run a procedure that displays values in the fill-ins.

```

/* ***** Main Block ***** */
/*
/* Set CURRENT-WINDOW: this will parent dialog-boxes and frames.
*/
ASSIGN CURRENT-WINDOW = {&WINDOW-NAME}
THIS-PROCEDURE:CURRENT-WINDOW = {&WINDOW-NAME}.

/* The CLOSE event can be used from inside or outside the procedure to
*/
/* terminate it.
*/
ON CLOSE OF THIS-PROCEDURE
    RUN disable_UI.

/* Best default for GUI applications is...
*/
PAUSE 0 BEFORE-HIDE.

RUN dcust.w PERSISTENT SET hCust.
RUN dorder.w PERSISTENT SET hOrder.
RUN addlink IN hCust(hCust, 'data', hOrder).
DYNAMIC-FUNCTION('setForeignFields' in hOrder,
'Order.CustNum,CustNum').

RUN initializeObject IN hCust.
RUN initializeObject IN hOrder.

OrderBrowse:QUERY IN FRAME {&FRAME-NAME} =
    DYNAMIC-FUNCTION('getDataHandle' IN hOrder).

OrderBrowse:ADD-LIKE-COLUMN('RowObject.CustNum').
OrderBrowse:ADD-LIKE-COLUMN('RowObject.OrderNum').

```

```

OrderBrowse:ADD-LIKE-COLUMN('RowObject.OrderDate').
OrderBrowse:ADD-LIKE-COLUMN('RowObject.PO').
OrderBrowse:ADD-LIKE-COLUMN('RowObject.SalesRep').

/* Now enable the interface and wait for the exit condition.
*/
/* (NOTE: handle ERROR and END-KEY so cleanup code will always fire.
*/
MAIN-BLOCK:
DO ON ERROR UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
  ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.
  IF NOT THIS-PROCEDURE:PERSISTENT THEN
    WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.

RUN displayValues.

```

This is the displayValues procedure:

```

PROCEDURE displayValues:

CustName:SCREEN-VALUE IN FRAME {&FRAME-NAME} =
  DYNAMIC-FUNCTION('columnValue' IN hCust, 'name').
OrderCust:SCREEN-VALUE =
  DYNAMIC-FUNCTION('columnValue' IN hOrder, 'Custnum').
OrderNum:SCREEN-VALUE =
  DYNAMIC-FUNCTION('columnValue' IN hOrder, 'OrderNum').
CustName:MODIFIED = NO. /* prepare this for possible modification. */

END PROCEDURE.

```

This is the CHOOSE trigger for the *Next Customer* button, which repositions the SDO query and then runs this same procedure to display the new values:

```

/* ON CHOOSE OF nextCust */
DO:
  RUN fetchNext IN hCust.
  RUN displayValues.
END.

```

And for the *Next Order* button:

```

/* ON CHOOSE OF NextOrder */
DO:
  RUN fetchNext IN hOrder.
  RUN displayValues.
END.

```

The *Customer Where Clause* fill-in allows the Customer query to be filtered by entering an expression to match. Its LEAVE trigger sets the Customer SDO *QueryWhere* property to that value and reopens the query:

```

/* ON LEAVE OF WhereClause */
DO:
  IF WhereClause:SCREEN-VALUE NE "" THEN
  DO:
    DYNAMIC-FUNCTION('setQueryWhere' IN hCust,
      WhereClause:SCREEN-VALUE).
    DYNAMIC-FUNCTION('openQuery' IN hCust).
  END.
END.

```

The *Reposition to Cust#* fill-in accepts a customer number and repositions the query to that customer using the findRow function:

```

/* ON LEAVE OF CustRepos */
DO:
  IF CustRepos:SCREEN-VALUE NE "0" THEN
  DO:
    IF NOT (DYNAMIC-FUNCTION
      ('findRow' IN hCust, CustRepos:SCREEN-VALUE))
      THEN MESSAGE "No such Customer" VIEW-AS ALERT-BOX.
    ELSE RUN displayValues.
  END.
END.

```

The *Customer Name* fill-in enables the Save button if the Name value was modified:

```

/* ON LEAVE OF CustName */
DO:
  IF CustName:MODIFIED THEN
    SaveCust:SENSITIVE = YES.
END.

```

And the Save button will save that new value to the database:

```

/* ON CHOOSE OF SaveCust */
DO:
  IF CustName:MODIFIED THEN
  DO:
    DYNAMIC-FUNCTION('updateRow' IN hCust,
      '', /* No Key to repos to */
      'Name' + CHR(1) + CustName:SCREEN-VALUE).
    SaveCust:SENSITIVE = NO. /* Disable the Save button. */
  END.
END.

```

10. Managing the SDO Temp-tables Directly

The examples shown thus far have all involved record-at-a-time access to the SDO data, where a particular record is positioned to, using the SDO methods, and then values from that row are read or written. In some cases, a procedure may wish to receive the entire RowObject table directly, rather than manipulating it indirectly, and also create the records in the RowObjUpd table directly,

Working with the Progress Version 9 ADM: Using SmartDataObjects from Outside the ADM
 Copyright 2000 Progress Software Corporation

and return that to the server, rather than doing updates one at a time using the API calls such as *updateRow*. This can be more efficient, especially for a non-4GL application which does not have a 4GL client SDO proxy available, but obviously makes the coding more complex, because the programmer must now be aware of the format of the RowObject and RowObjUpd temp-tables and their special fields. Again, it is expected that a developer using this technique will build or use an appropriate “driver” which provides a client API of an appropriate form which can be used by application components. Thus the example reproduced below (taken from the src/samples area of the commercial product) is not intended as an example of good programming style, but simply as an example of what calls are involved in transferring data back and forth. In addition to the standard ADM2 documentation, the SDO white paper provides detailed information on the construction of the RowObject and RowObjUpd temp-tables and how records are moved back and forth between client and server.

In a sense the API becomes very simple when using this technique: since the developer is responsible for maintaining the temp-tables, only the calls to send and receive the temp-tables are required to perform basic read and update operations. Of course, other aspects of the SDO API such as functions to set and get properties, calls to *addQueryWhere* or *assignQuerySelection* to manipulate the Where clause, and so forth, are all still valid.

In this example there are just three calls to the SDO API:

- **initializeObject** – (internal procedure, no arguments) – This is run in the server-side SDO after it is created.
- **serverSendRows** – (internal procedure, INPUT piStartRow, INPUT pcRowIdent, INPUT plNext, INPUT piRowsToBatch, OUTPUT iRowsReturned, OUTPUT TABLE RowObject) – This procedure takes the number of rows to be returned (the RowsToBatch property) and returns a temp-table to the caller. StartRow can be used to specify a specific row within an existing query to start with; RowIdent can be used to specify a particular RowIdent value to reposition to. Normally these are not used; see the standard documentation and code header for more information. The Next flag signals whether the database query should be positioned to the next row before starting to load the temp-table. When retrieving the first (or only) batch of rows, this flag should be set to FALSE. And the RowsReturned OUTPUT parameter tells the caller how many rows were actually returned (in case RowsToBatch was greater than the number of rows in the dataset). See the standard documentation for more information on this call.
- **serverCommit** – (internal procedure, INPUT-OUTPUT TABLE RowObjUpd, OUTPUT cMessages, OUTPUT cUndolds) – This call returns a temp-table to the server which holds all of the rows which were added, deleted, or modified. Any error messages generated by the Commit are returned to the caller (cMessages), along with a list of RowIdent fields for the row or rows which generated the errors (cUndolds).

```

/*
  Sample test program for running a SmartDataObject directly using
  an AppServer.
*/

/* This is what the temp-table looks like:
* DEFINE temp-table RowObject
* FIELD CustNum LIKE Customer.CustNum
* FIELD Name LIKE Customer.Name
* FIELD City LIKE Customer.City
* FIELD RowNum AS INT
* FIELD RowIdent AS CHAR
* FIELD RowMod AS CHAR.*/

```

```

DEFINE temp-table RowObject
  {CustSDO.i}
  {src/adm2/robjflds.i}.

/* This temp-table holds changes to be sent back to the server. */
DEFINE TEMP-TABLE RowObjUpd LIKE RowObject
  FIELD ChangedFields AS CHARACTER.

DEFINE VARIABLE iRowsReturned AS INTEGER.
DEFINE VARIABLE p AS HANDLE.
DEFINE VARIABLE cnt AS INTEGER.
DEFINE VARIABLE cMessages AS CHARACTER.
DEFINE VARIABLE cMessage AS CHARACTER.
DEFINE VARIABLE cUndoIDs AS CHARACTER.
DEFINE VARIABLE iCustNum AS INTEGER.
DEFINE VARIABLE appServer AS WIDGET-HANDLE.
DEFINE VARIABLE ok AS LOGICAL.

/* First connect to the AppServer using the default service */

CREATE SERVER appServer.
ok = appServer:CONNECT("-H ithaca") . /* Modify for your host name */
IF ok AND NOT ERROR-STATUS:ERROR THEN
  MESSAGE "Connected to the AppServer!" VIEW-AS ALERT-BOX.
ELSE DO:
  MESSAGE "Error connecting to the AppServer" VIEW-AS ALERT-BOX.
  RETURN.
END.

/* First start up the SDO, initialize it, and open the query. */

RUN CustSDO.w ON SERVER appServer persistent SET p.
RUN initializeObject IN p.

/* Now ask the SDO to move up to 100 rows of data into the temp-table.
*/

RUN serverSendRows IN p
  (0 /* no particular start row */,
  "" /* or starting RowId */,
  no /* no need to do a NEXT after getting the first row */,
  100 /* maximum number of rows to batch*/,
  OUTPUT iRowsReturned /* actual number of rows returned */,
  OUTPUT TABLE RowObject).

/* Display the data we got back. */
MESSAGE "Number of rows fetched: " iRowsReturned VIEW-AS ALERT-BOX.
MESSAGE "About to display rows fetched" VIEW-AS ALERT-BOX.
FOR EACH RowObject:
  DISPLAY RowObject.CustNum
    RowObject.name format "x(20)"
    RowObject.city format "x(12)"
    RowObject.rowmod format "x"
    RowObject.rowident format "x(12)".
END.

/* Now modify each customer from 1 to 4: add "xx" to the city field,

```

```

    or if the field already ends with "xx", remove it. */
FOR EACH RowObject WHERE RowObject.CustNum < 5:
    /* Create an Update row with the unchanged copy of the record,
       but listing those fields which we are updating. This ChangedFields
       list goes into the before copy of the record.
    */

    CREATE RowObjUpd.
    BUFFER-COPY RowObject TO RowObjUpd
    ASSIGN RowObjUpd.RowMod = ""
           RowObjUpd.ChangedFields = "City".

    /* Now create an "after" copy of the record; set RowMod to "U"update
       and change the "city" field. */

    CREATE RowObjUpd.
    BUFFER-COPY RowObject TO RowObjUpd
    ASSIGN RowObjUpd.RowMod = "U".
    cnt = INDEX(RowObject.City, "xx").
    IF cnt = 0 THEN
        RowObjUpd.City = RowObject.City + "xx".
    ELSE
        RowObjUpd.City = SUBSTR(RowObject.City, 1, cnt - 1).
END.

/* Now Add a new customer, using the next available CustNum > 100.
   Set RowMod to "A"dd to indicate that this is a new record,
   and give it a high RowNum to make sure it doesn't conflict with
   any other row in the table of data we received. */

FIND LAST RowObject WHERE RowObject.CustNum > 100 AND RowObject.CustNum
< 1000
NO-ERROR.
iCustNum = IF AVAILABLE RowObject THEN RowObject.CustNum + 1 ELSE 101.

CREATE RowObjUpd.
ASSIGN RowObjUpd.CustNum = iCustNum
       RowObjUpd.Name = "Cust " + STRING(iCustNum)
       RowObjUpd.City = "Bedford"
       RowObjUpd.RowMod = "A"
       RowObjUpd.RowNum = 100001
       RowObjUpd.ChangedFields = "CustNum,Name,City".

/* Now Delete the first record with a CustNum > 1000, if there is one.
   Set RowMod to "D" to indicate the the row is to be deleted. */

FIND FIRST RowObject WHERE RowObject.CustNum > 1000 NO-ERROR.
IF AVAILABLE RowObject THEN
DO:
    CREATE RowObjUpd.
    BUFFER-COPY RowObject TO RowObjUpd
    ASSIGN RowObjUpd.RowMod = "D".
END.

/* Now pass the changes back to be written to the database. */
MESSAGE "About to send updates to the AppServer" VIEW-AS ALERT-BOX.

```

```

RUN serverCommit IN p (INPUT-OUTPUT TABLE RowObjUpd, OUTPUT cMessages,
  OUTPUT cUndoIDs).

/* The output parameter cMessages will contain any error messages.
  These messages are a delimited string. */

IF cMessages NE "" THEN
DO cnt = 1 TO NUM-ENTRIES(cMessages, CHR(3)):
  cMessage = ENTRY(cnt, cMessages, CHR(3)).
  MESSAGE ENTRY(1, cMessage, CHR(4)) VIEW-AS ALERT-BOX.
END.

/* Finally, redisplay records which have been added/updated/deleted */

MESSAGE "About to show summary of added/updated/delete records" VIEW-AS
ALERT-BOX.
FOR EACH RowObjUpd:
  DISPLAY RowObjUpd.CustNum RowObjUpd.name RowObjUpd.city.
END.

```