

Creating Object Views

With Client/Server technology, it is increasingly important that data access is encapsulated in an object-oriented manner. In addition to enabling simple data access to be switched from one computer to another, this modularization offers the following benefits:

- Each access must be coded only once.
- An access operation, when available and tested, can be eliminated as a source of error.
- All object operations are located in a clearly defined place
- Database changes result in minimum modification in easily locatable places.
- An access operation can be used wherever it is required within a system.
- When naming conventions are adhered to, required access operations are easily found.

The following topics are covered below:

- Concepts
 - Natural Objects Associated with an Object View
 - Implementing Single-Object Access
 - Implementing Multiple-Object Access
 - Implementing Access to Preliminary Copies
 - Object View Implementation
-

Concepts

All database accesses are encapsulated in object views. The set of object views for an application builds the access layer for the application. The associated "object orientation" makes it possible to distribute data using a Client/Server approach.

In an object view, all database operations, for example, Store, Update, etc. are implemented for an object on which they can be used.

The term "Object" refers to a group of business data which logically belongs together. An object corresponds, in its simplest form, to an entity identified during requirement analysis.

Complex objects can, however, represent a structure of entities which have relationships with each other.

A simple example of this is the object *Order* composed of the object *Orderheader* and the object *Order position*:

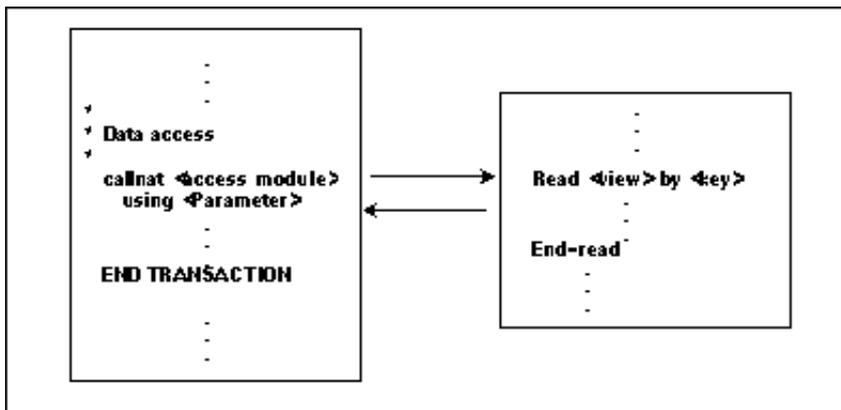
Order



An operation on an object is not implemented in a program directly as a database access. Rather, an access module associated with the object view is called with the appropriate parameters. The database access is then executed by the access module, and the results of this operation are returned to the executing program.

The activation and confirmation of data modifications is performed by an application subprogram (activation module). This module performs the transfer of preliminary data to the original data as well as the confirmation of the data update.

For further information, see section Transaction Logic.



The code of the activation module confirms successful modification with an END TRANSACTION statement or in case of an error condition, backs-out the transaction with the BACKOUT TRANSACTION statement. For distributed databases, it must be ensured that these statements are also applied to all relevant databases.

Besides database access, all necessary validations for data consistency and security are carried out by the access modules of the object views. This is done before any access to a database takes place. This is to ensure that these checks are always carried out regardless of the origin of the call.

Due to the different interfaces (with a single object only one object is passed back, while with multiple objects, several objects are passed back) used for processing single and/or multiple objects, at least two access modules are created for each object, i.e. one for single- and another for multiple-object processing.

Access modules are implemented as subprograms, and required data is supplied in parameter data areas.

Natural Objects Associated with an Object View

When an object view is created using the frame gallery, the following Natural objects are produced:

Object View Info

Contains information collected during the process of creating the object view, which is used later when creating dialogs which use the object view. The data is held in an internal format and can be viewed when the Info button is selected in the "Frame Gallery" dialog window.

Object View LDA

Contains the Natural view of the object and a start key definition used by the multiple object access module.

Constants LDA

Contains constants defining (1) the number of blocks of A100 used in the single object parameter data area; (2) the number of objects to be retrieved by each call to the multiple object access module; (3) a position number for each field in the object view local data area; (4) optionally, additional object-specific operation codes.

These are not automatically included but can be added manually. For further information, see Starting the Implementation.

Single Object PDA

Contains the call interface for the single object access module.

Multiple Object PDA

Contains the call interface for the multiple object access module.

Preliminary Copies Copycode

Contains code for use by dialogs which use preliminary copies.

Single Object Subprogram

Encapsulates operations for accessing and modifying single objects.

Multiple Object Subprogram

Encapsulates retrieval operations for retrieving multiple objects.

Preliminary Copies Subprogram

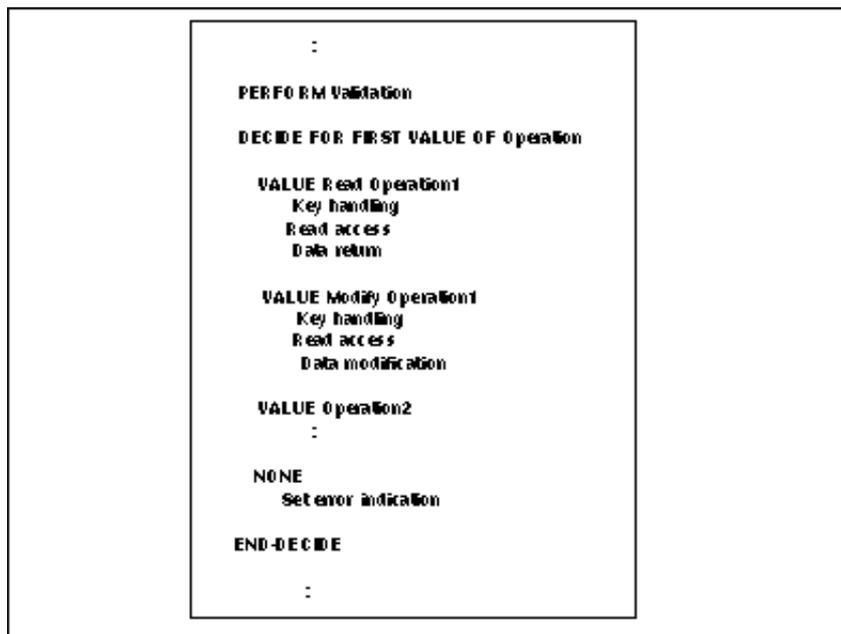
Transfers updates made to preliminary copies to the database.

Implementing Single-object Access

All database operations that read or change an object's data are implemented in one of the access modules belonging to that object. This object data can be composed of fields from various DDMs, which together build a logical object. Before the execution of an operation in an access module, the required consistency checks are executed.

Access Module Structure

The structure of a single object access module is shown below:



The frame gallery produces a single-object access module which includes standard operations for READ, GET, STORE, UPDATE, DELETE and CHECK_EXISTENCE. Additional operations can be added manually using the subprogram editor.

Creation of Consistency Checks

The single-object access module produced by the frame gallery includes checks to ensure that parameters required by the standard operations are passed. Any further consistency checks and validations must be coded manually.

All required validations for data consistency checks are executed before the database modifications take effect.

These checks are collected in an inline subroutine similar to the following:

```
DECIDE ON EVERY VALUE OF Operation

    VALUE Operation1, Operation2, ...
        Consistency checks

    VALUE Operation1, Operation3, ...
        Consistency checks

    VALUE Operation5, ...
        :
        :
        :
        :
    NONE
        Set error indication

END-DECIDE
```

Checks for multiple operations can be combined through the use of the statement `DECIDE ON EVERY VALUE`.

If the consistency checks contain database accesses to other objects (e.g. external key checks), then the access module calls an access module belonging to the other object's object view to carry out the database access. This procedure is necessary to allow data distribution.

To minimize communication overhead with distributed data storage, it may be desirable to perform certain checks during dialog processing. Nevertheless, these checks should also be included in the object view.

The checks do not, however, have to be coded more than once.

You can stipulate that only validation, but no modifying database accesses are executed by the access module, with the switch `PZ_AS_VALID_ONLY`.

It is also possible for certain field-specific checks, for example; check for numerical content, to be copied from the access module and directly coupled with the corresponding input field in for example the 'Change' event.

Application Program/Object View Interface

The interface between application programs and access modules contains various types of data:

- object independent data;
- object dependent data.

Object Independent Data

Parameters that are the same for every object are included here, and are defined in the parameter data area ZXAS000A. In particular, the following data is included:

- IN: Operation code (PZ_AS_OPERATION)
The standard operation codes used by the application shell are contained in the local data area ZXA0000L. Additional operation codes can either be hard coded in the program or defined as constants in a local data area. Suggested code is included in the constants local data area, produced for an object view by the frame gallery.
- IN: Validation flag (PZ_AS_VALID_ONLY)
Indicates that **only** validation is to be carried out.
- IN: Language position (PZ_AS_LANG_POS)
- IN/OUT: ISN of the record that will be or was read (PZ_AD_ISN)
- OUT: Error information
 - Response code (PZ_AS_RSP)
 - Existence of record. If the record exists the flag is set to 1 (PZ_AS_REC_EXIST)
 - Message number (PZ_AS_MSG_NUM)
 - Additional message information (PZ_AS_MSG_FILL)
 - Position number of the field in error in the view (PZ_AS_FLD_POS)
 - Index of the field in error for arrays (PZ_AS_FLD_OCC)
- OUT: Runtime information
 - Natural error code number ((PZ_AS_NAT_ERROR)
 - Natural program line in which error occurred (PZ_AS_NAT_LINE)
 - Natural object name (PZ_AS_NAT_PROG)

When using an SQL access module, an additional standard local data area ZXA000QL is required.

Object Dependent Data

For the single-object access module, three sets of object-dependent data are required and are produced by the frame gallery during object view creation:

- an local data area containing the Natural view.
- a single object parameter data area which includes a copy of the Natural view, as a group containing all or a subset of the fields of a complete view. It differs from the Natural view structure in that all structure definitions are omitted.

Only elementary fields are defined. Higher level fields are excluded.

This is done for the following reasons:

- The danger that errors will occur on parameter transfer is eliminated.
- Type conversion would otherwise not function properly with distributed data storage.
- Only essential fields are transferred.

This parameter data area is used both in the access module and in the application programs.

Additional object dependent data, for example constants for field positions and object specific operation codes are also defined in a local data area (the constants LDA).

Additional Object Dependent Data

Any additional object dependent fields and parameters should be defined in a separate parameter data area.

Error Handling

For error handling, the following applies:

- A position number is assigned as a constant for each field of the object view local data area. The necessary code is generated in the constants LDA.
- Each input field is assigned a field position within the dialog.
- If multiple views are affected, a view ID must be used to identify the view containing errors.
- If an error occurs in the access module, then the error number and the position number of the view field in error is returned. For arrays the indices for these marked errors are also returned.

If, for example, when deleting an object, additional reference checks are necessary, it is recommended to include in the error text the name of the object in which the object to be deleted is a foreign key.

Implementing Multiple-Object Access

Note:

This functionality is automatically generated.

In principle, all database operations which read more than one object record are implemented in an access module belonging to the object.

These "records" can be composed of fields from various DDMs, which together form a logical object.

The object view for multiple-object processing does not, as a rule, contain database accesses that change the data.

The access module returns a definable number of objects. The maximum number of supplied objects is defined in the index of the arrays of object fields found in the interface local data area and/or parameter data area of the application program.

If a block of records is returned, then the starting value of the next block to be read is also returned automatically. As a result, unlimited numbers of records can be read sequentially.

Because no accesses change data, the execution of consistency checks is not required.

Structure of the Multiple-Object Access Module

The structure of a multiple-object access module is as follows:

```

DECIDE FOR FIRST VALUE OF Operation
  VALUE Read Operation1
    Key handling
    Read access
    Data return
  VALUE Read Operation2
    Key handling
    Read access
  VALUE Read Operation3
    :
    :
    :
  NONE
    Set error indication
END-DECIDE
    
```

The frame gallery produces a multiple-object access module which includes standard operation for LIST and GET_SET. Additional operations can be added manually using the subprogram editor.

Checking

In a multiple-object access module, checking is restricted to the required input parameters for control of the access module.

Application Program/Object View Interface

The area of communications between application programs and access modules contains various types of data:

- Object Independent Data
- Object Dependent Data

Object Independent Data

Parameters that are the same for every object are defined in the parameter data area ZXAM000A. The following is included:

- IN: Operation code (PZ_AM_OPERATION)

The standard operation codes used by the application shell are defined in the local data area ZXAM000L. Additional operation codes can either be hard coded in the program or defined as constants in a local data area. Suggested code is included in the constants LDA, produced for an object view by the frame gallery.

- IN: Number of required objects (PZ_AM_CNT_REC_NEEDED)
- IN/OUT: Number of objects found (PZ_AM_CNT_REC_FOUND)
- OUT: Status information
 - Response code (PZ_AM_RSP)
 - EOD flag, is set, when END OF DATA (PZ_AMEND_OF_DATA).
- OUT: Runtime information
 - Natural error code number ((PZ_AM_NAT_ERROR)
 - Natural program line in which error occurred (PZ_AS_NAT_LINE)
 - Natural object name (PZ_AM_NAT_PROG)

Object Dependent Data

For the multiple-object access module, two sets of object-dependent data are required and are produced by the frame gallery during object view creation:

- a local data area containing the Natural view (also used by the single-object access module). This local data area also includes a definition for a single start key.
- a multiple-object parameter data area which includes an array containing a copy of the Natural view, as a group which includes all or a subset of the fields in the Natural view. This definition differs from the Natural view structure in that all structure definitions are omitted. Only elementary fields are defined. Higher level fields are excluded.

A pair of start/thru values and a minimum value are defined for the selected search key for the object view. By default, these are used to read a range of records. For a composite key, you can include additional code to treat each component as containing independent start/thru values.

Additional start/thru values for further search keys can be added manually.

Additional selection criteria can also be defined in the parameter data area. With distributed data storage, it may be desirable to carry out all selections on the computer which holds the data.

This parameter data area is used as both in the access module as well as locally in application programs.

Error Handling

Errors will only occur if an invalid parameter (unknown operation code, invalid/ missing start value) or a call in the multiple-object access module to the single-object access module involving database updates is detected.

Error information passed by the single-object module to the multiple-object module must be passed on by the multiple-object module to the application program.

Error handling is performed in the same way as outlined above for the single-object access module.

Implementing Access to Preliminary Copies

The frame gallery produces two Natural modules to handle access to preliminary copies for an object view. A copycode contains the object type specific access to the preliminary records. An activation module transfers the preliminary data to the original data.

For further information on the use of preliminary copies, see Preliminary Copies.

Copycode for Access to Preliminary Copies

For access to preliminary copies, a copycode is created for each object. This copycode is issued in the maintain dialogs and subdialogs, as well as in the corresponding activation module.

A modified version of this copycode is produced if the single-object parameter data area contains more than 4000 bytes.

Activation Module

The preliminary copies subprogram produced by the frame gallery can be modified as required.

If you do not want to use the predefined operation codes in the object view, you can specify the desired operation code using the following MOVE statement subsequent to the call of subroutine

Z_INIT_PARMS_ACCESS_ORIGINAL:

```
MOVE <operation-code> TO PZ_AS_OPERATION
```

Object View Implementation

Starting the Implementation

The object view which builds the access layer should be implemented as early as possible. The earliest possible point of time is upon completion of the database design.

Undoubtedly, not all operations can be implemented so early. Experience has shown that requests for additional operations arise during implementation. However, it should be possible to implement all standard operations.

The frame gallery automatically generates an object view which includes the actions READ, GET, STORE, UPDATE, DELETE, CHECK_EXISTENCE, LIST and GET_SET.

Object View Creation

Procedure

The procedure described here assumes that the access modules do not need to call access modules for further object views.

If this is not the case, see Object View Creation for Complex Objects.

Create the object view using the frame gallery (see Creating an Object View).

When object-specific operations are required complete the following:

Add all object-specific operation codes, which are individually valid for this object, and are not already in the standard local data area for operation codes to the local data area.

Assign initial values for operation codes in ascending order, beginning with 51 in the above mentioned local data area with the object-specific contents.

Implement the individual operations in the available suggested code DECIDE structure in the multiple- or single-object access modules.

Code the appropriate VALUE statement, and insert the essential coding.

For reasons of clarity, it is recommended that individual operations are placed after standard operations.

Test the object view.

Use:

for single-object access, the skeleton program ZXFAS00P

for multiple-object access, the skeleton program ZXFAM00P

If later, during the implementation phase, additional operations are required, then check that the operation codes have already been defined. If not, define the new operation codes, and implement them in the access modules.

Additional Information Concerning Multiple-object Access

With multiple-object access, three repositioning parameters can be used to delimit which records are to be read. These are:

- **Start-value.** Specifies the first record to be accessed (with the first read). When the access module has read as many objects as are to be returned by a single call to the module, it sets the start key to the next object to be read. When the access module is next called, it starts from this new start value.
- **Thru-value.** Specifies the last record to be accessed. Not set by dialogs produced using the frame gallery.
- **Minimum-value.** Not set by dialogs produced using the frame gallery.

A minimum value field is included in the multiple-object parameter data area but the access module does not include suggested code and the minimum key field is not used by dialogs generated using the frame gallery. You can add code to use the minimum value as follows.

The minimum value can be set to indicate that the components of a composite key are to be treated as independent search criteria. Before the first call to the access module, the minimum value is set to the start value.

For each key component, all objects with a value lower than that specified as the minimum value are ignored.

Example

Start-value:

Client ID	Customer ID	Retailer ID
BB	Carey	Bell

End-value:

Client ID	Customer ID	Retailer ID
DD	Zackery	McCalls

The first access will begin with the Start-value. The first 10 occurrences are placed in the parameter data area and the 11th occurrence is marked as the next start-value. If additional objects are required, the next access will begin with the 11th occurrence. This procedure is repeated until either the end-value is reached or no additional objects are required.

If the following Minimum-value is used, the number of objects returned is further reduced:

Client ID	Customer ID	Retailer ID
BB	Andrews	Bell

For client ID CA, for instance, only objects with a Customer ID greater or equal to Carey and a Retailer ID greater or equal to Bell will be returned.

Object View Creation for Complex Objects

▶ To create object views for complex objects which contain other objects

1. Generate an object view for each individual object.
Make sure that an object view for the lowest object in the hierarchy is implemented first.
2. In each access module that needs access to the lowest hierarchical object, code a call to the access module belonging to this object.

A hierarchical calling structure of access modules, matching the logical structure of the object, results.

Size Problem Solution

Size problems in an application program could be caused by the following:

- too many or too large data areas accessed in the program, resulting in a program which cannot be stowed, or
- total size of data areas is too large at run time due to nested calls to modules.

Because the object view concept suggests only one object view per object, for all fields in that object, problems of size can be solved by reducing the data areas in the application program.

There are two initial solutions:

Solution 1 - Data Areas in the Application Program too Large to Stow

If the data areas in the application program are too large, then the problem can be solved by reducing the size of the data areas in the application program.

▶ To reduce the size of the data area in the application program

1. Encapsulate the call to the access module in an additional subprogram, which can be used as a type of "intermediate layer".
2. Define the interface between the application program and "intermediate module", so that only the data actually used by the application program is passed.

The intermediate module contains the "complete" interface for the object view.

Communication between the application program and the access module is no longer direct, instead the data in the intermediate layer is filtered. Through this method the number of object views for an object is reduced.

Solution 2 - Total Size of Data Areas at Runtime too Large**▶ To reduce the size of the data areas at run time**

- Create object views that include only interface fields which are actually used by the application program.

Take care in particular that not too many object views are created. Experience has shown that a "small" additional object view, containing the most frequently used data is sufficient.

This object view can be used in all application programs and/or access modules which only require the reduced data area.

Calling the Access Modules

Note:

The frame gallery automatically generates these calls.

Access modules are invoked in the following manner:

```
CALLNAT <Access Module Name>
      USING <Standard Parameter>
          <Object View Fields>
          <Additional Parameters> (optional)
```

Standard Parameters	For single-object modules: PZ_XAS000 For multiple-object modules: PZ_XAM000 This enables the control parameters for the access module to be transferred.
Object View Fields	Enables the group name of the view fields from the data area which contains the object fields of the view: P_viewname
Additional Parameters	These parameters contain all information which in addition to the standard parameters and the view fields must be passed to the access module, e.g. selection or sorting criteria. Using additional information is optional.

Error Handling

An error check should be made after calling the access module.

When using a single-object module, it is sufficient to check the variable PZ_AS_RSP.

The check for multiple-object modules is combined with a REPEAT loop.

This REPEAT loop is used when the number of database accesses makes a screen output necessary. It can also be used when the records returned by the access module require further filtering by the application program which must read call the access module more than once, e.g. to display a large number of objects in a list box.

The error check for the multiple-object module takes place within this REPEAT loop.

This call has the following structure:

```
REPEAT
  Call to the Access Module
  Error check
END-REPEAT
```

Single-object Processing with the Multiple-object Module

If the application program already uses multiple-object access for an object, and single-object access is required, it is not necessary that a single-object object view, with accompanying data areas, be additionally implemented in the application program.

A single-object can also be read by a multiple-object module:

Set the parameter `PZ_AM_CNT_REC_NEEDED` to 1, and then activate the multiple-object module in the usual manner.

Due to the large size of the data area for a multiple-object module, this approach is only practical when multiple-object access is already used in the application program.

Note:

The subset of fields available with the multiple-object access module may differ from that available via the single-object access module, depending on how you created the object view using the frame gallery.

Reading Sequentially using the Multiple-object Module

Sometimes further filtering of the records, which are returned from the multiple-object module, is necessary in the application program.

If, this is the case, proceed as follows:

Code the secondary selection check inside the REPEAT loop to call the access module after the error enquiry.

Remove the rejected records from the transfer buffer, and move the remaining records together

Count the number of rejected records.

Subtract the number of rejected records from the value in the variable `PZ_AM_CNT_REC_FOUND`.

Repeat this loop process.

In this way, the rejected records are replaced. The start value for the next call to the access module is set by the access module during its previous call to the record following the last record it read.