OO Database Modeling - Theory and Practice

A research report by
Tim Kuehn

www.tdkcs.com
The material in this presentation is derived from an ongoing research project undertaken by TDK Consulting and is for informational purposes only.
The Code...

DEFINE BUFFER Customer FOR Customer.

DEFINE VARIABLE hCustomer AS HANDLE NO-UNDO.

CREATE BUFFER hCustomer
    FOR TABLE "Customer".

Issues and Opportunities....
• Static declaration for buffers and handles
• Scoped to the procedure, internal procedure, or function
• “Harder” to create and track additional buffers
• Requires manual object lifecycle management
• No OO capabilities
• Not encapsulated
The dream..... an OO-Enclosed Buffer:

CLASS CustomerDbRecord:
DEFINE PUBLIC PROPERTY CustNum AS INTEGER NO-UNDO
   GET: RETURN(Customer.CustNum).
   END GET.
   SET(iCustNum AS INTEGER):
      ASSIGN Customer.CustNum = iCustNum.
   END SET.

/* Code to setup and control */
/* the record buffer... */

END CLASS.

The advantages.....
• Fully Encapsulated!
• Strong typing
• No passing issues w/in a session
• Can add functionality as needed

The challenges.....
• Fully Encapsulated!
• Different conceptual thought process
• Adding functionality to the right layer
• Scoping generic functionality for a table, record, field, and buffer

Usage:
DEFINE VARIABLE oCustomerDbRecord AS CustomerDbRecord NO-UNDO.
oCustomerDbRecord = NEW CustomerDbRecord().
The mission....

Design an OO(ABL) data access structure to provide:

- generic control of a table, record, and buffer,
- a set of OO constructs which encapsulate the ABL "data access" language elements

Required for Implementation:

- Create a set of generic OO base classes for data access and management
- Use the resulting OO objects in an application

That doesn’t sound so hard.....
Buffer Objects...Oh My!

Buffer object handle

A handle to a buffer object, corresponding to an underlying ABL buffer, which can be static or dynamic. An example of a static underlying buffer is one you define at compile time by using the DEFINE BUFFER statement, or by implicitly referencing a table ABL construct such as Customer. An example of a dynamic underlying buffer is one you create at run time with the CREATE BUFFER statement.

Syntax

```
buffer-handle [ :attribute | :method ]
```

**buffer-handle**

An item of type HANDLE representing a handle to a buffer object.

**attribute**

An attribute of the buffer object.

**method**

A method of the buffer object.

Attributes

<table>
<thead>
<tr>
<th>Buffer object attribute</th>
<th>Buffer object method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM-DATA</td>
<td>AFTER-BUFFER</td>
</tr>
<tr>
<td>AMBIGUOUS</td>
<td>ATTACHED-PAIRLIST</td>
</tr>
<tr>
<td>AUTO-SYNCHRONIZE</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>BEFORE-BUFFER</td>
<td>BEFORE-ROWID</td>
</tr>
</tbody>
</table>

What have I gotten myself into?
Mapping Buffer Attributes

Mapping a read-only attribute to an OO property:

```
DEFINE PUBLIC PROPERTY BufferHandle AS HANDLE NO-UNDO GET. PRIVATE SET.
```

Invariant properties are set in the class constructor

Mapping a read/write attribute to an OO property:

```
DEFINE PUBLIC PROPERTY SerializeName AS CHARACTER NO-UNDO

END GET.

SET(chString AS CHARACTER):

    ASSIGN THIS-OBJECT:BufferHandle:SERIALIZE-NAME = chString.
END SET.
```

(make sure to get INTEGER and INT64's right!)
Mapping Buffer Methods

Mapping a buffer method to an OO method:

METHOD PUBLIC LOGICAL BufferCompare(hSrcBuffer AS HANDLE):
  RETURN(THIS-OBJECT:BufferHandle:BUFFER-COMPARE(hSrcBuffer)).
END METHOD.

METHOD PUBLIC LOGICAL BufferCopy(hSrcBuffer AS HANDLE):
  RETURN(THIS-OBJECT:BufferHandle:BUFFER-COPY(hSrcBuffer)).
END METHOD.
The things you find...

A hierarchy of “buffer objects”:

BUFFER TableName:BUFFER-HANDLE

Temp-Table Buffer Object

Dataset Buffer Object

Generic Buffer Object

Database Buffer Object

Multi-tenant Database Buffer Object

Or, to look at it another way… “Innovation you can build on.”
Buffer Object Hierarchy

Mapping the “buffer object” hierarchy to OO buffer classes:

- Generic Buffer
- Database Buffer
- Multi-tenant Buffer
- TempTable Buffer
- DataSet Buffer

Diagram:

- Generic Buffer Object
- Database Buffer Object
- Multi-tenant Database Buffer Object
- Temp-Table Buffer Object
- Dataset Buffer Object
The things you find…

A hierarchy of “Temp Table Objects”:

TEMP-TABLE TtName:HANDLE ➔ Temp-Table Table Object ➔ Dataset Table Object
Creating Concrete Classes

Accomplished:
Developed a set of classes that maps DB and temp-table buffer handles to a set of generic OO buffer object constructs.

Next Goal:
Use these generic OO buffer object constructs to create a set of classes that model concrete DB tables, records, and buffers.

First Step:
Map a Sports2000 “Customer” DB table to a set of concrete OO classes.
A hierarchy of “buffer objects”:

- BUFFER TableName:BUFFER-HANDLE
- Generic Buffer Object
- Temp-Table Buffer Object
- Dataset Buffer Object
- Database Buffer Object
- Multi-tenant Database Buffer Object

This will be the focus for creating the "Customer" DB table classes
Buffer Object Hierarchy

Mapping the “buffer object” hierarchy to OO buffer classes:

This will be the focus for creating the "Customer" DB table classes
The Final Objective

An overall view of the final object model:

Database = DB Type IV Structure
Customer Table Records = DB Type III Structure
Customer Buffer Record / Fields = DB Type II Structure
Customer Buffer Handle = DB Type I Structure

Each type value corresponds to an encapsulation layer starting from the inner-most layer and working to the outer layers.
"Type" Class Common Structure

Structure Types

- Customer DB Table
- Customer DB Record
- Customer DB Buffer

Implementation Structure

- "Type" Class
  - Implementation for a specific table / db
  - Custom ABL logic for the table(s)
  - "Type" super classes
    - Implements logic common to the type structure (buffer, record, table, db)
  - Lesser type class instance as a class member
DB Type I Structure

Customer DB Table
Customer DB Record
Customer DB Buffer

Customer Buffer Handle = DB Type I Structure
Structure:
- Customer DB Buffer class *inherits* the Database Buffer Object class
- Customer DB Buffer class holds business logic specific to the Customer table buffer
- Database Buffer Object methods & properties are *exposed*
- Customer DB Buffer Object methods & properties are *exposed*
- Customer DB Buffer can reference a local-to-the-class “Customer” buffer handle, or it can be a Customer table buffer-handle passed in via the constructor

Properties & Usage:
- Used for controlling a Customer table buffer
- Appropriate for linking an OO Buffer to other OO(ABL) constructs (Queries, Data-Sources, etc.)
CLASS CustomerDbBuffer
  INHERITS DataBaseBufferObject:

CONSTRUCTOR CustomerDbBuffer():
  SUPER(BUFFER Customer:HANDLE).
END CONSTRUCTOR.

CONSTRUCTOR CustomerDbBuffer(hBufferParm AS HANDLE):
  SUPER(hBufferParm).
END CONSTRUCTOR.

END CLASS.
Extend the Class

Extend CustomerDbBuffer to “FIND” a Customer

Add to CustomerDbBuffer:

METHOD PUBLIC LOGICAL FindCustomerNL( iCustnum AS INTEGER):
RETURN(THIS-OBJECT:FindUnique(SUBSTITUTE("WHERE &1.custnum = &2",
    THIS-OBJECT:Name, STRING(iCustnum)),
    LockWaitStatic:NoLock)).
END METHOD.
DB Type II Structure

Customer Buffer Record / Fields = DB Type II Structure

Customer Buffer Handle = DB Type I Structure
Customer DB Record Design Overview - Structure

**Structure:**
- Customer DB Record class *inherits* Database Record Class
- Customer DB Record class *contains a private* Customer DB Buffer Object
- Customer DB Buffer Object references the Customer DB Record Customer buffer
- Database Record logic is *exposed via methods*
- Customer DB table *fields are exposed via properties*
- Customer DB Record *logic is exposed via methods*
- Customer DB Buffer Object reference can be obtained via a *method API*
Properties & Usage:
- Class properties mimic Customer table fields
- Record position and control done directly on the local Customer buffer, or indirectly using the Customer DB Buffer object
- Eliminates name-space collision between Customer DB Buffer Object properties and Customer DB table field names
- Appropriate location for implementing record-level CRUD business logic
- Used to link with other OO(ABL) constructs (Queries, Data-Sources, etc.)
OO DB Modeling - Theory and Practice

PUG Challenge Americas 2018

"Innovation you can build on."

DB Type II Structure

Customer DB Record Implementation Overview

CLASS CustomerDbRecord INHERITS DatabaseRecord:

DEFINE VARIABLE oCustomerDbBo AS CustomerDbBuffer NO-UNDO.

CONSTRUCTOR CustomerDbRecord():
    THIS-OBJECT(NEW CustomerDbBuffer(BUFFER Customer:HANDLE)).
END CONSTRUCTOR.

CONSTRUCTOR PRIVATE CustomerDbRecord(oCustomerDbBoParm AS CustomerDbBuffer):
    SUPER(oCustomerDbBoParm).
    ASSIGN oCustomerDbBo = oCustomerDbBoParm.
END CONSTRUCTOR.

/* Class Properties defined in next slide */

END CLASS.
Customer DB Record Implementation Overview – Field Properties

**DEFINE PUBLIC PROPERTY** CustNum AS INTEGER NO-UNDO

**GET:** RETURN(Customer.CustNum).
END GET.

**SET**(iCustNum AS INTEGER):
ASSIGN Customer.CustNum = iCustNum.
END SET.

**DEFINE PUBLIC PROPERTY** PostalCode AS CHARACTER NO-UNDO

**GET:** RETURN(Customer.PostalCode).
END GET.

**SET**(chTmp AS CHARACTER):
ASSIGN Customer.PostalCode = chTmp.
END SET.
Extending the Classes

Task: Extend CustomerDbRecord to “FIND” using a Customer Number

Already added to CustomerDbBuffer:

METHOD PUBLIC LOGICAL FindCustomerNL( iCustnum AS INTEGER):
RETURN(THIS-OBJECT:FindUnique(SUBSTITUTE("WHERE &1.custnum = &2",
THIS-OBJECT:Name, STRING(iCustnum)),
LockWaitStatic:NoLock)).
END METHOD.

Now add to CustomerDbRecord:

METHOD PUBLIC LOGICAL FindCustomerNL(iCustNum AS INTEGER):
RETURN(oCustomerDbBo:FindCustomerNL(iCustNum)).
END METHOD.

Facade which passes the call to the CustomerDbBuffer API
Extending the Classes

Task: Extend CustomerDbRecord to “FIND” using a Customer Number

CustomerDbRecord call:

METHOD PUBLIC LOGICAL FindCustomerNL(iCustNum AS INTEGER):
  RETURN(oCustomerDbBo:FindCustomerNL(iCustNum)).
END METHOD.

Doing the same thing with a static buffer reference:

METHOD PUBLIC LOGICAL FindELNW(iCustNum AS INTEGER):
  FIND Customer
    WHERE Customer.CustNum = iCustNum
    EXCLUSIVE-LOCK
    NO-WAIT.
  RETURN(AVAILABLE(Customer)).
END METHOD.

Benefit – does lookup directly instead of via the OO buffer handle
Cost – would not implement logic in OO buffer object
**DB Type III Structure**

Customer DB Table

- **Customer Table Records** = DB Type III Structure
- **Customer Buffer Record / Fields** = DB Type II Structure
- **Customer Buffer Handle** = DB Type I Structure

Customer DB Record

Customer DB Buffer
Customer DB Table Design Overview:
Table = Container for 1 or more records

Structure:
- Customer DB Table class inherits Database Table class
- Customer DB Record is private to the Customer DB Table class
- Database Table class APIs are exposed
- Customer DB Table class APIs are exposed
- Customer DB Record class exposed via an API

Properties & Usage:
- Appropriate location for Customer DB Table scoped business logic
- Customer DB Table can be updated to have more than one Customer Record DB class instance
- Appropriate place for implementing table-level CRUD business logic
- Used to link with other OO(ABL) constructs (Queries, Data-Sources, etc.)
**DB Type III Structure**

**Customer DB Table Implementation Overview**

```plaintext
CLASS CustomerDbTable
    INHERITS DatabaseTable:

DEFINE VARIABLE oCustomerDbRecord AS CustomerDbRecord NO-UNDO.

CONSTRUCTOR CustomerDbTable():
    THIS-OBJECT(NEW CustomerDbRecord()).
END CONSTRUCTOR.

CONSTRUCTOR PRIVATE CustomerDbTable(oCustomerDbRecordParm AS CustomerDbRecord):
    SUPER(oCustomerDbRecordParm).
    oCustomerDbRecord = oCustomerDbRecordParm.
END CONSTRUCTOR.

METHOD PUBLIC CustomerDbRecord GetCustomerDbRecord():
    RETURN(oCustomerDbRecord).
END METHOD.
END CLASS.
```

- **Class definition**
- **Customer Record DB variable**
- **Create a new table using a default Customer DB Record object**
- **Remember the Customer DB Record object used to make this instance**
- **API to get the Customer DB Record instance**
Overall OO Structure

Customer DB Table

- Customer DB Record
- Customer DB Record
- Customer DB Record
- Customer DB Record

Customer DB Record

- Custnum
- City
- PostalCode

Corresponds to Customer DB Record Fields

Corresponds to the Customer DB Table Record

Corresponds to Customer DB Table Buffer Handle

"DB Table" holds one or more Customer DB Record instances

"Innovation you can build on."

"Innovation you can build on."

"Innovation you can build on."

"Innovation you can build on."

"Innovation you can build on."

"Innovation you can build on."
Sports DB Design Overview (not implemented)

Structure:
- Sports DB class *inherits* the Database class
- Sports DB *contains* all DB table classes as private instances
- Database class attributes are *exposed*
- Sports DB attributes are *exposed*
- SportsDB table classes are *exposed* via APIs

Properties & Usage:
- Contains all table classes.
- Appropriate location for all inter-table Data Access BL
- Appropriate place for DB-level CRUD BL
Putting theory to use
The Target Application

A read-only set of TT and DB objects to encapsulate data taken from the Progress _file, _field, etc. tables.
From Theory to Practice – My Experience

- Currently on the 3rd iteration
- A significant amount of detail to manage
- Mapping relationships is a challenge
- Will be hard to fully implement by hand
- Good candidate for a code generator
An overall view of the final object model:

- Database = DB Type IV Structure
- Customer Table Records = DB Type III Structure
- Customer Buffer Record / Fields = DB Type II Structure
- Customer Buffer Handle = DB Type I Structure

Each type value corresponds to an encapsulation layer starting from the inner-most layer and working to the outer layers.
The Model – From Theory to Application

Database = DB Type IV Structure
Customer Table Records = DB Type III Structure
Customer Buffer Record / Fields = DB Type II Structure
Customer Buffer Handle = DB Type I Structure

- A database is a collection of 0 or more tables
- A table is a collection of 0 or more records + 1 or more indexes
- A record is a collection of 1 or more fields
- A buffer is a pointer to a record
- Tables have relationships

Storage
Run Time
Architecture
Implementation Overview

- Possible to completely separate the interface from the implementation
  - All data access structures programmed to an interface
  - Able to swap between a TT and a DB as needed!
- Some encapsulation "pros" and "cons"
  - Abstracts some common operations into the object
  - Limits what you can do with a table to what's in the object
  - Cloning a TT object requires duplicating the TT data
- A code generator may be required for certain implementation models.
- Performance?
Overall OO Structure Review

- "DB Table" holds zero or more Customer DB Record instances
- "DB Table" corresponds to the Customer DB Table
- Customer DB Record:
  - Custnum
  - City
  - PostalCode
- Customer DB Buffer Handle

- Too much copying data
- Would be hard to use
- Probable performance issues
- Needs to be "flattened" out a bit

"Innovation you can build on."
A Flattened Out OO Structure

"Table" holds zero or more records

Corresponds to a Table.

Table

Record

Record

Record

Record

Static Table Buffer

Query

Find

Corresponds to Record Fields

Field1

Field2

Field3

Points to a Record

"Innovation you can build on."
The Target Application

Each record will have:

- Fields that are "key" (DbRecordID) and "not key" (everything else)
- A set of "key" fields is an "identifier"
  - "DbIdentifier" = DbRecordId
  - "FileIdentifier" = DbRecordId, FileRecordID
- Keys, Identifiers, and their access members a specified in their own interfaces
- Every object implements its own interface
- Interface inheritance is only done in an interface, never a class
- All APIs are written to an interface

Some terminology:

- Schema  Parent name for all the tables FieldSchema, FileSchema, etc.
- Base  Base class in an inheritance chain for a given of abstraction
- List  Item pertains to a "list" of things – records, lines, etc.
- TT, DB  Temp Table, Database
Building the Object Structure

Object Class Inheritance for a File TT object

- FileSchemaTtList
  - Public Object Implementation for FileSchema TT object records
- SchemaListTtBase
  - Implements generic TT record functionality for all Schema List TT objects
- SchemaListBase
  - Implements
    - Generic record functionality for all DB / TT objects
    - SchemaBase members
- SchemaBase
  - Specifies abstract members required by all DB / TT objects
Object **Class** Inheritance for a **File DB** object

- **FileSchemaDbList**
  - Public Object Implementation for FileSchema DB object records
  - Implies
    - Generic record functionality for all DB / TT objects
    - SchemaBase members
- **SchemaListBase**
- **SchemaBase**
  - Specifies abstract members required by all DB / TT objects

now for the interfaces…
Building the Object Structure

Object Interface Inheritance for a File TT object

- iFileSchemaTtList
- iFileSchemaList
- iFileSchema
- iFileSchemaIdentifier
- iFileSchemaTtList
- iSchematicListTtBase
- iSchematicList
- iSchematicListBase
- iSchematicBase
- iGetFileIdentifier
- iGetFileIdentifier
- iGetDbIdentifier
- iGetDbIdentifier
- iDbSchemaIdentifier
- iHolderBase

Same as object classes

Used to indicate what the class holds – DB or TT

TDK Consulting
"Innovation you can build on."
Building the Object Structure

Object **Interface** Inheritance for a **File TT** object

- `iFileSchemaTtList`
- `iFileSchemaList`
- `iFileSchema`
- `iFileSchemaIdentifier`
- `iDbSchemaIdentifier`

**Data Structure & Access**

- `iSchemaListTtBase`
- `iSchemaListBase`
- `iSchemaBase`
- `iHolderBase`

**Record Identification**

**Generic table management**
Building the Object Structure

Object Interface Inheritance for a File Db object

- **iFileSchemaDbList**
- **iFileSchemaList**
- **iFileSchema**

Data Structure & Access

- **iSchemaListBase**
- **iSchemaBase**
- **iHolderBase**

Record Identification

- **iFileSchemalentifier**
- **iDbSchemalentifier**
- **iGetFileIdentifier**
- **iGetDbIdentifier**

Generic table management

"Innovation you can build on."
File Table Access Interface

Object **Interface** to program to use for **File** table references

Program all table access to this interface

- iFileSchemaList
- iFileSchema
- iFileSchemaIdentifier
- iGetFileIdentifier
- iDbSchemaIdentifier
- iGetDbIdentifier
- iSchemaListBase
- iSchemaBase
- iHolderBase
A Code Example

"File" table access object for local use

```
oFileSchemaList:GetFirst().
DO WHILE oFileSchemaList:Available():
   /* do stuff */
   oFileSchemaList:GetNext().
END.
```

Holds an assy of table access objects

Filter to records that match the identifier

Traverse Records

"File" table access object instance in a "holder" object

Get a local copy

"Innovation you can build on."
Converting Fields to Properties

METHOD PUBLIC iPropertyDefinitionList
  GeneratePropertyList(oFieldSchemaList AS iFieldSchemaList, chPropertyVisibility AS CHARACTER)
):

  oFieldPropertyList = NEW PropertyDefinitionList(oGenerateSupport).

  oFieldSchemaList:OpenQuery().
  oFieldSchemaList:GetFirst().

  DO WHILE oFieldSchemaList:Available():

    oFieldProperty = NEW PropertyDefinition(oGenerateSupport).
    oFieldProperty:PropertyName = oFieldSchemaList:_Field-Name.
    oFieldProperty:DataTypeName = oFieldSchemaList:_Data-Type.
    oFieldProperty:NoUndo = YES.
    oFieldProperty:AccessMode:SetAttribute(chPropertyVisibility, YES).
    oFieldProperty:ExtentSpecification = IF oFieldSchemaList:_Extent > 0
                                             THEN "EXTENT " +
                                             STRING(oFieldSchemaList:_Extent)
                                             ELSE ""
                                            .

    oFieldPropertyList:AddProperty(oFieldProperty).
    oFieldSchemaList:GetNext().
  END.

RETURN(oFieldPropertyList).
Questions?
Thank you for your time!

Tim Kuehn
TDK Consulting Services Inc.
www.tdkcs.com

Email: tim.kuehn@tdkcs.com
Ph: 519-781-0081
Twitter: @tdkcs