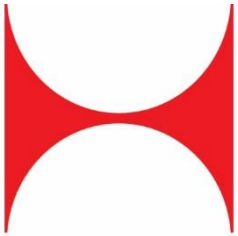
OO Database Modeling - Theory and Practice

A research report by

Tim Kuehn



TDK CONSULTING

www.tdkcs.com

Disclaimer…

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

Where it all began..

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

I have a dream…

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 c
static or dynamic. An example of a static underlying buffer is one you define at c
time by using the DEFINE BUFFER statement, or by implicitly referencing a tabl
ABL construct such as `Customer.CustNum`. An example of a dynamic underlying
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

ADM-DATA attribute	AFTER-BUFFER attribute	AFTER-ROWID attri
AMBIGUOUS attribute	ATTACHED-PAIRLIST attribute	AUTO-DELETE attri
AUTO-SYNCHRONIZE attribute	AVAILABLE attribute	BATCH-SIZE attribu
BEFORE-BUFFER attribute	BEFORE-ROWID attribute	BUFFER-GROUP-ID attribute

Methods

(2 of 2)

BUFFER-GROUP-NAME attribute	BUFFER-PARTITION-ID attribute	BUFFER-TENANT-ID attribute
BUFFER-TENANT-NAME attribute	CAN-CREATE attribute	CAN-DELETE attribute
CAN-READ attribute	CAN-WRITE attribute	CRC-VALUE attribute
CURRENT-CHANGED attribute	CURRENT-ITERATION attribute (Data Objects)	DATASET attribute
DATA-SOURCE attribute	DATA-SOURCE-COMPLETE-MAP attribute	DATA-SOURCE-MODIFIED attribute
DATA-SOURCE-ROWID attribute	DBNAME attribute	DYNAMIC attribute
ERROR attribute	ERROR-STRING attribute	FILL-MODE attribute
HANDLE attribute	HAS-LOBS attribute	INSTANTIATING-PROCEDURE attribute
IS-MULTI-TENANT attribute	IS-PARTITIONED attribute	KEYS attribute
LAST-BATCH attribute	LOCKED attribute	NAME attribute
NAMESPACE-PREFIX attribute	NAMESPACE-URI attribute	NEW attribute
NEXT-SIBLING attribute	NUM-CHILD-RELATIONS attribute	NUM-FIELDS attribute
NUM-ITERATIONS attribute (data objects)	NUM-REFERENCES attribute	ORIGIN-ROWID attribute
PARENT-RELATION attribute	PRIMARY attribute	PRIVATE-DATA attribute
QUERY attribute	RECID attribute	RECORD-LENGTH attribute
REJECTED attribute	ROWID attribute	ROW-STATE attribute
SERIALIZE-NAME attribute	TABLE attribute	TABLE-HANDLE attribute
TABLE-NUMBER attribute	TYPE attribute	UNIQUE-ID attribute
XML-NODE-NAME attribute		

(1 of 2)

ACCEPT-CHANGES( ) method	ACCEPT-ROW-CHANGES( ) method
APPLY-CALLBACK( ) method	ATTACH-DATA-SOURCE( ) method
BUFFER-COMPARE( ) method	BUFFER-COPY( ) method
BUFFER-CREATE( ) method	BUFFER-DELETE( ) method
BUFFER-FIELD( ) method	BUFFER-RELEASE( ) method
BUFFER-VALIDATE( ) method	DETACH-DATA-SOURCE( ) method
DISABLE-DUMP-TRIGGERS( ) method	DISABLE-LOAD-TRIGGERS( ) method

(2 of 2)

EMPTY-TEMP-TABLE( ) method	FILL( ) method
FIND-BY-ROWID( ) method	FIND-CURRENT( ) method
FIND-FIRST( ) method	FIND-LAST( ) method
FIND-UNIQUE( ) method	GET-CALLBACK-PROC-CONTEXT( ) method
GET-CALLBACK-PROC-NAME( ) method	GET-CHANGES( ) method
GET-CHILD-RELATION( ) method	GET-ITERATION( ) method (Data Objects)
INDEX-INFORMATION( ) method	MARK-NEW( ) method
MARK-ROW-STATE( ) method	MERGE-CHANGES( ) method
MERGE-ROW-CHANGES( ) method	RAW-TRANSFER( ) method
READ-JSON( ) method	READ-XML( ) method
READ-XMLSCHEMA( ) method	REJECT-CHANGES( ) method
REJECT-ROW-CHANGES( ) method	SAVE-ROW-CHANGES( ) method
SERIALIZE-ROW( ) method	SET-CALLBACK( ) method
SET-CALLBACK-PROCEDURE( ) method	SYNCHRONIZE( ) method
WRITE-JSON( ) method	WRITE-XML( ) method
WRITE-XMLSCHEMA( ) method	–

AFTER-FILL event	AFTER-ROW-FILL event
BEFORE-FILL event	BEFORE-ROW-FILL event
FIND-FAILED event	OFF-END event
ROW-CREATE event	ROW-DELETE event
SYNCHRONIZE event	–

For information on these events, see the "ProDataSet events" section on page 2045.

Buffer-field object handle, CREATE BUFFER statement, DEFINE BUFFER statement, ProDataSet object handle, Query object handle, Temp-table object handle

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
      GET: RETURN(THIS-OBJECT:BufferHandle:SERIALIZE-NAME).
      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:

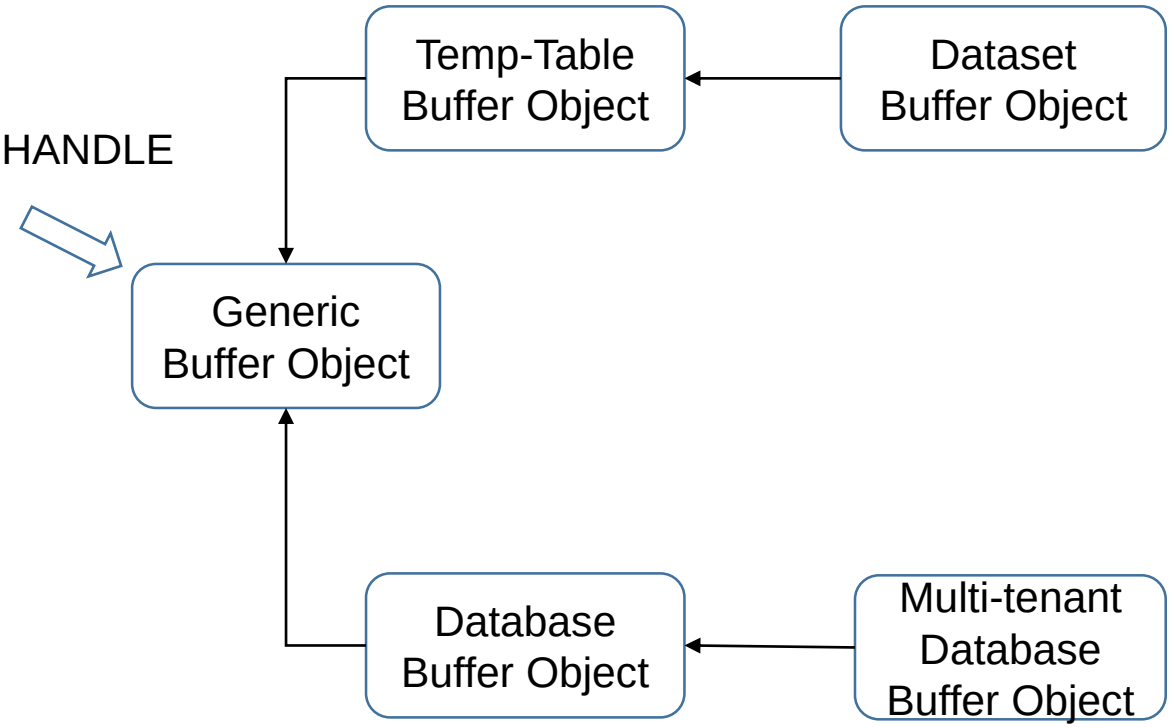
```
METHOD PUBLIC LOGICAL BufferCompare(hSrcBuffer AS HANDLE):
RETURN(THIS-OBJECT:BufferHandle:BUFFER-COMPARE(hSrcBuffer)).
END METHOD.

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”:

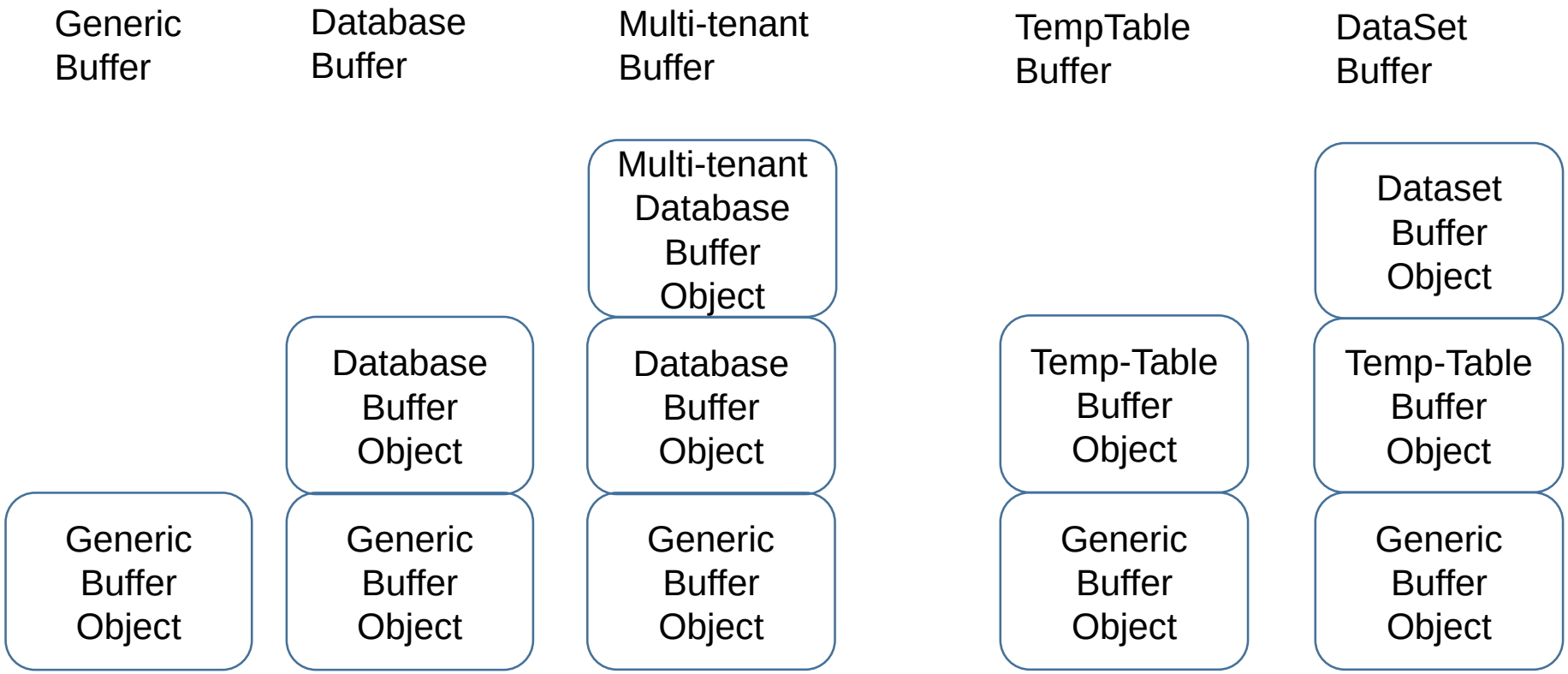
BUFFER TableName:BUFFER-HANDLE



Or, to look at it another way…

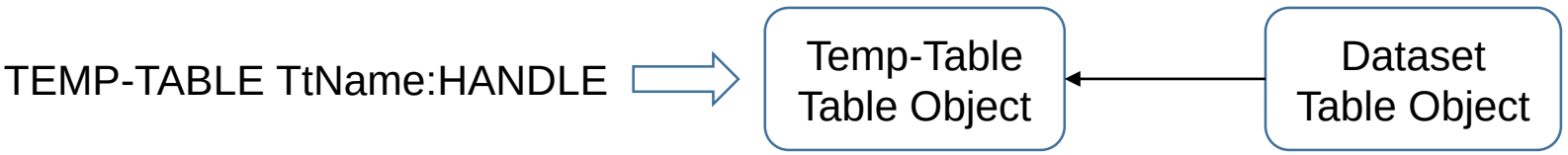
Buffer Object Hierarchy

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



The things you find…

A hierarchy of “Temp Table Objects”:



Creating Concrete Classes

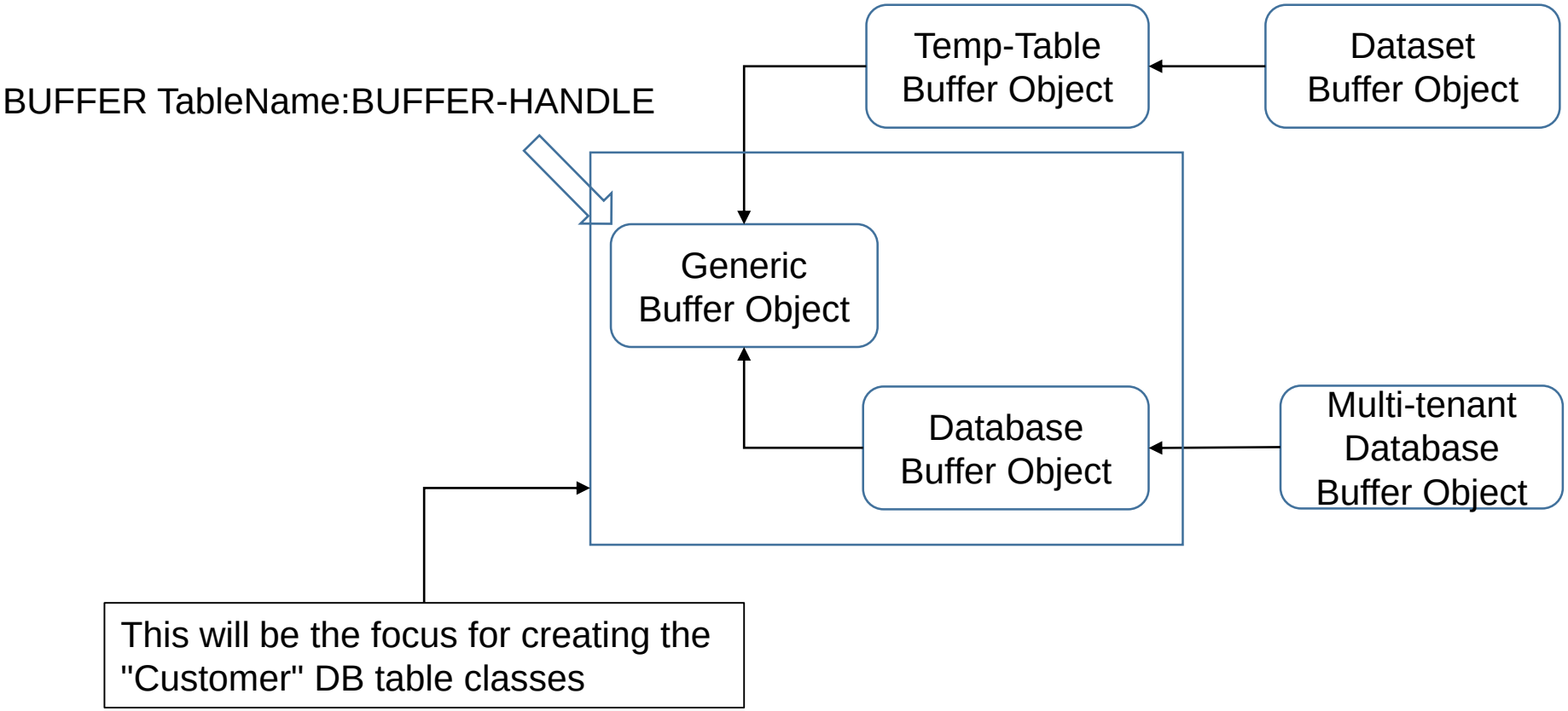
Accomplished:
Developed a set of classes that maps DB and temp-table buffer handles to a set of *generic* OO buffer object constructs.

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.

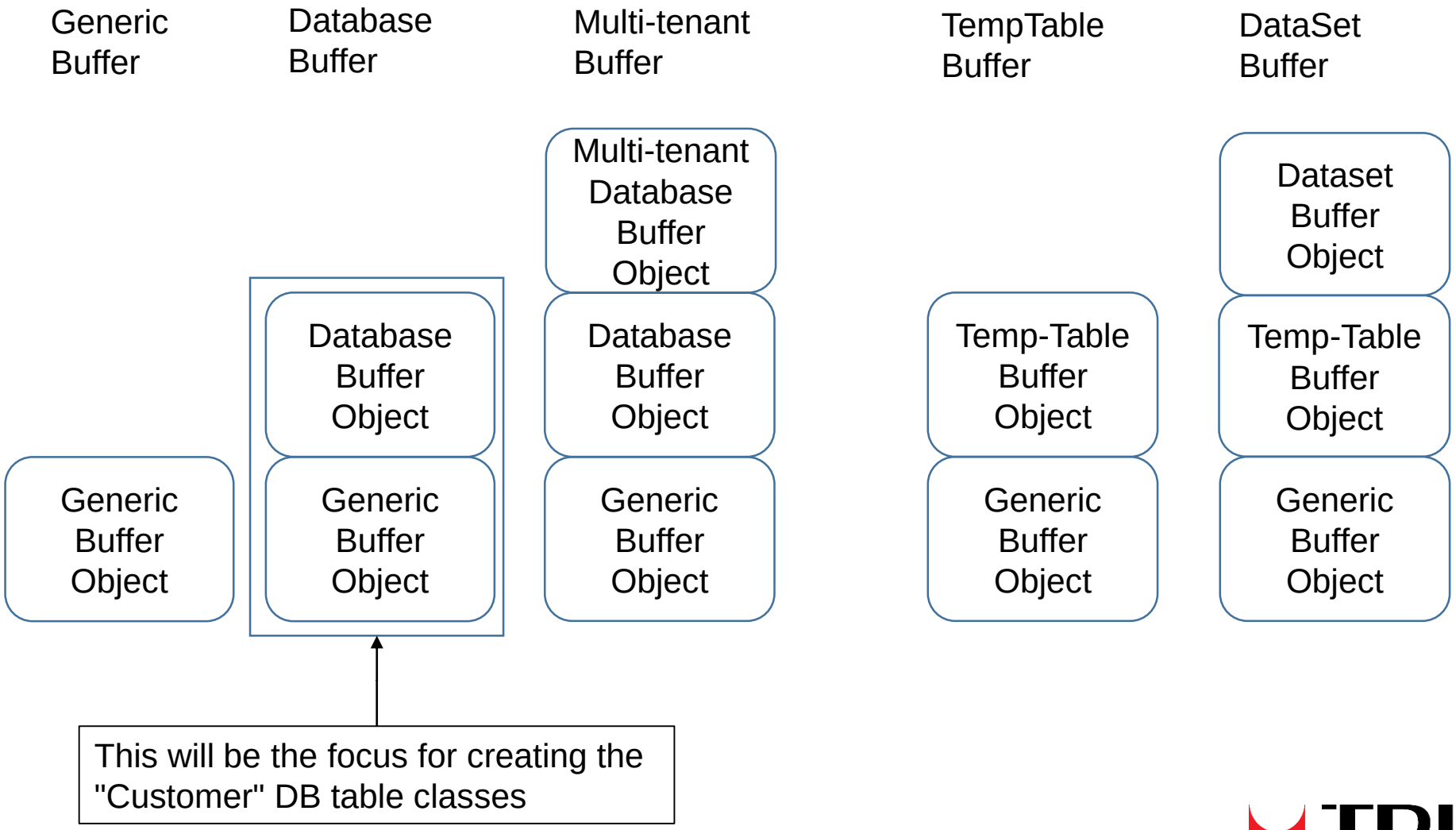
DB Buffer Objects

A hierarchy of “buffer objects”:



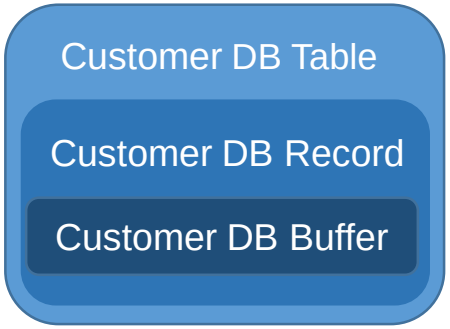
Buffer Object Hierarchy

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



The Final Objective

An overall view of the final object model:

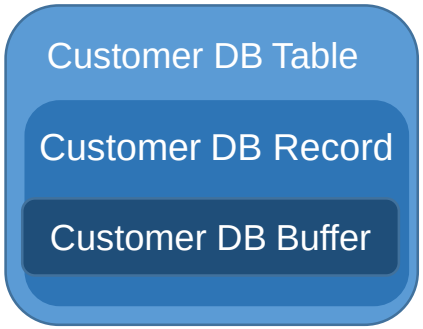


Database	= DB Type IV Structure
Customer Table Records	= DB Type III Structure
Customer Buffer Record / Fields	= DB Type II Structure
Customer Buffer Handle	= DB Type I Structure

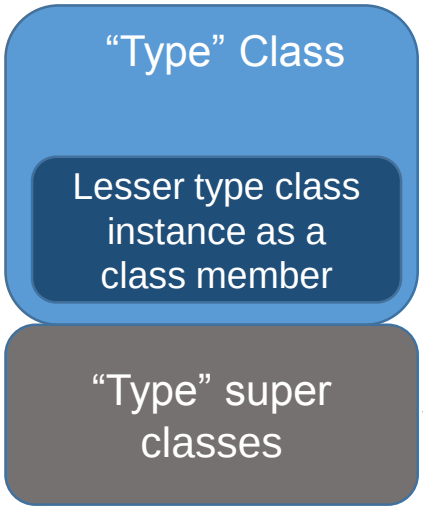
Each type value corresponds to an encapsulation layer starting from the inner-most layer and working to the outer layers

"Type" Class Common Structure

Structure Types



Implementation Structure

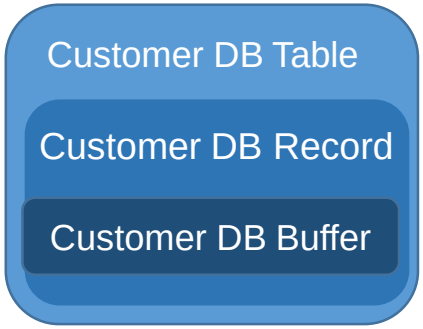


Implementation for a specific table / db

Custom ABL logic for the table(s)

Implements logic common to the type structure (buffer, record, table, db)

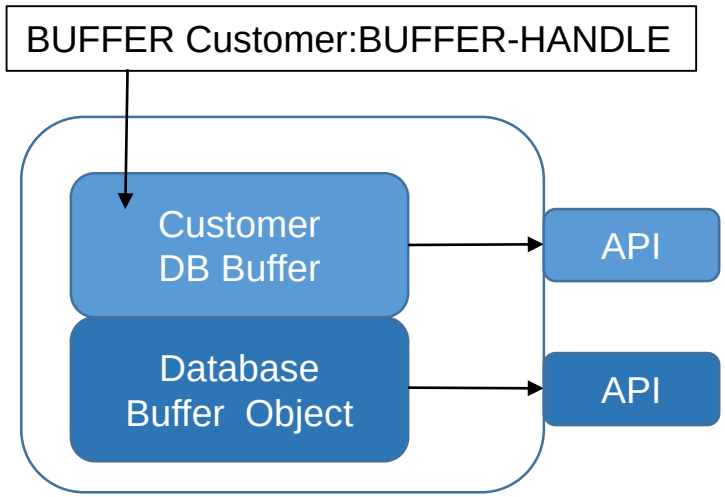
DB Type I Structure



Customer Buffer Handle = DB Type I Structure

DB Type I Structure

Customer DB Buffer Design Overview



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.)

DB Type I Structure

Customer DB Buffer Implementation Overview

```
CLASS CustomerDbBuffer
    INHERITS DataBaseBufferObject:


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


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


END CLASS.
```

Class Definition

Create object using a local customer buffer

Create object using an external customer buffer reference

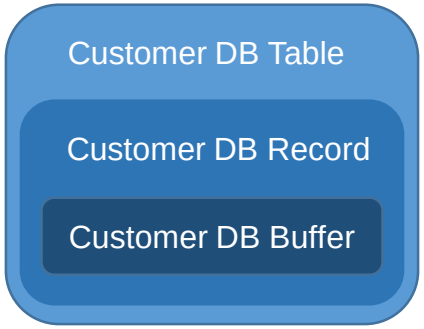
Extending the Class

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

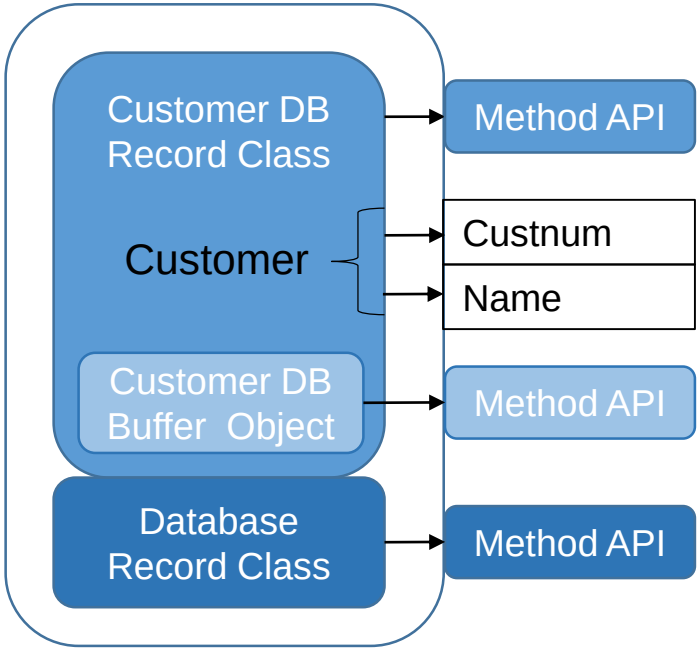


Customer Buffer Record / Fields = DB Type II Structure

Customer Buffer Handle = DB Type I Structure

DB Type II Structure

Customer DB Record Design Overview - Structure

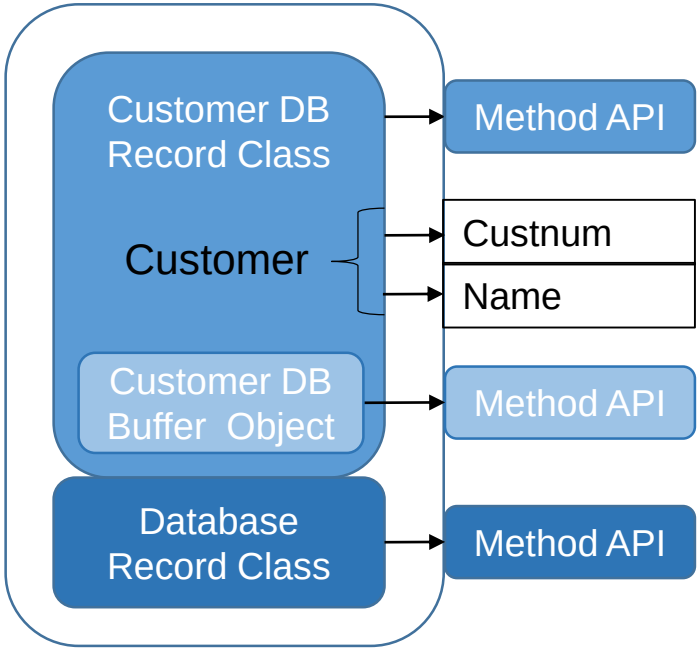


Structure:

- 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

DB Type II Structure

Customer DB Record Design Overview - Properties



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.)

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.
```

Class definition

Customer DB Buffer Object

Create a new Customer DB record instance using a local customer buffer

Create a new record instance using a passed-in customer buffer object

Hold the Customer Buffer Object instance reference

DB Type II Structure

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.
```

Façade which passes the call to the CustomerDbBuffer API

Now add to CustomerDbRecord:

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

Extending the Classes

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

Façade which passes the call to the CustomerDbBuffer API

CustomerDbRecord call:

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

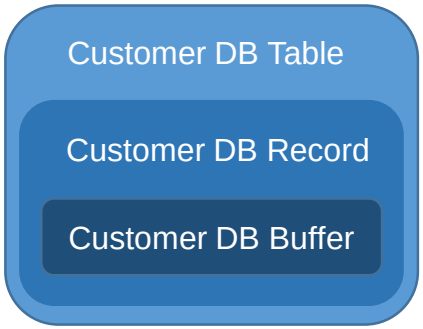
Doing the same thing with a static buffer reference:

```
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

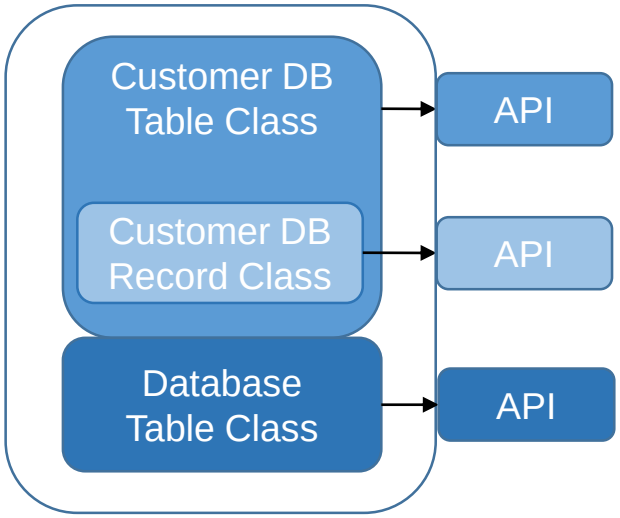
DB Type III Structure



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

DB Type III Structure

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

```
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

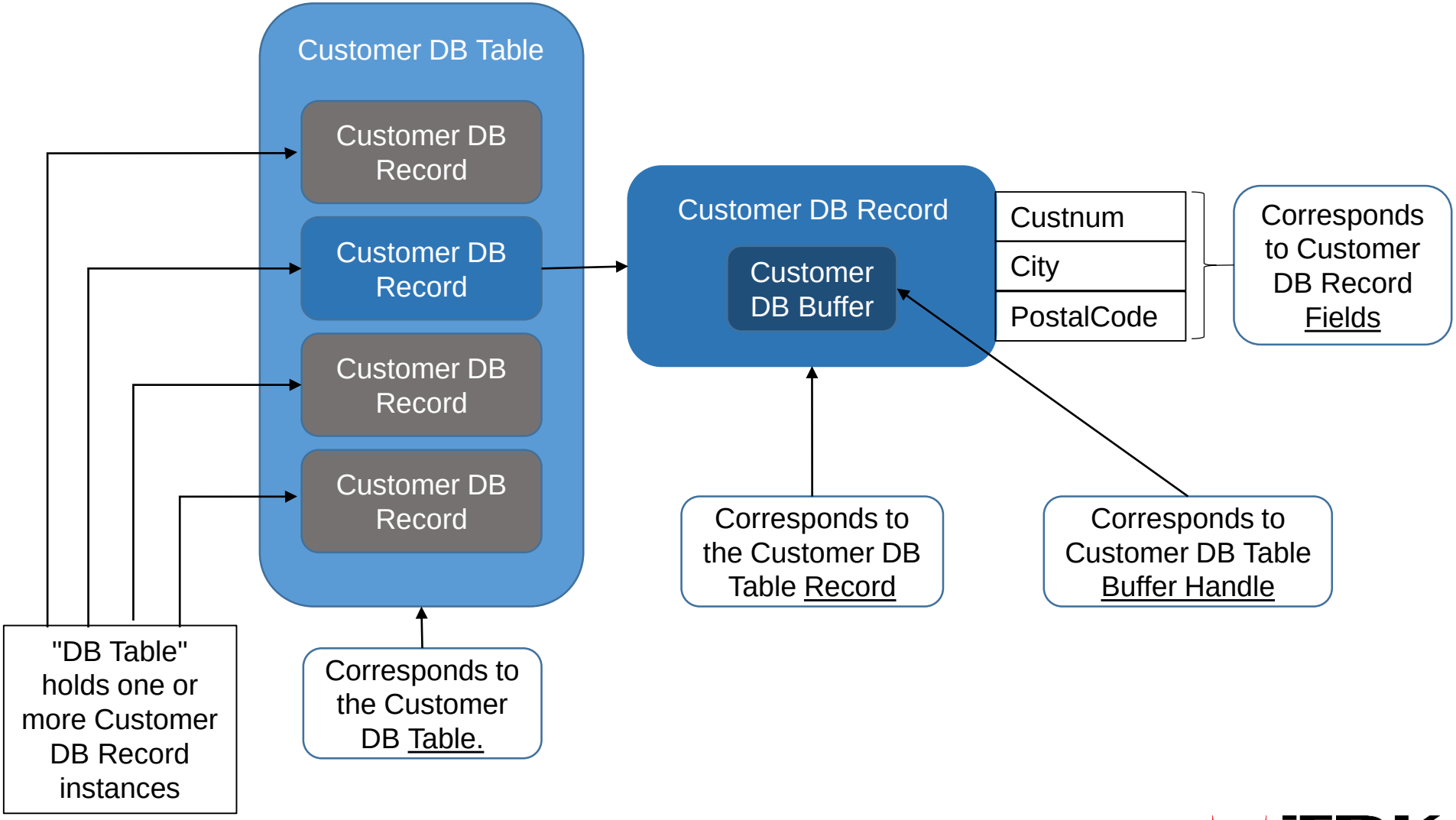
Customer Record DB variable

Create a new table using a default Customer DB Record object

Remember the Customer DB Record object used to make this instance

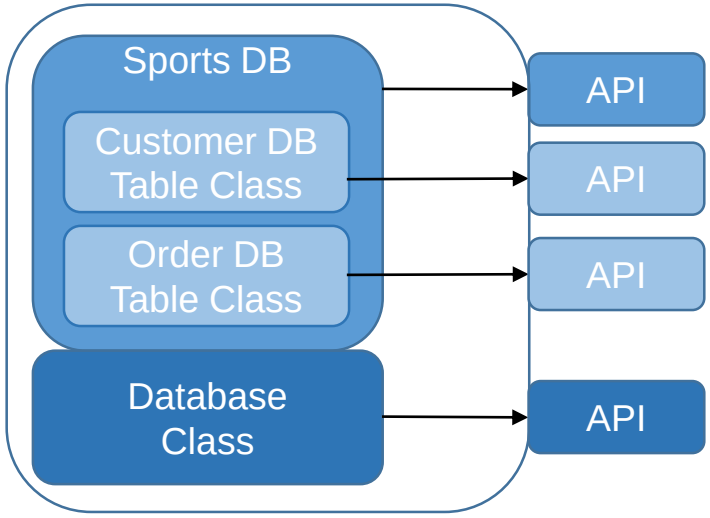
API to get the Customer DB Record instance

Overall OO Structure



DB Type IV Structure

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:

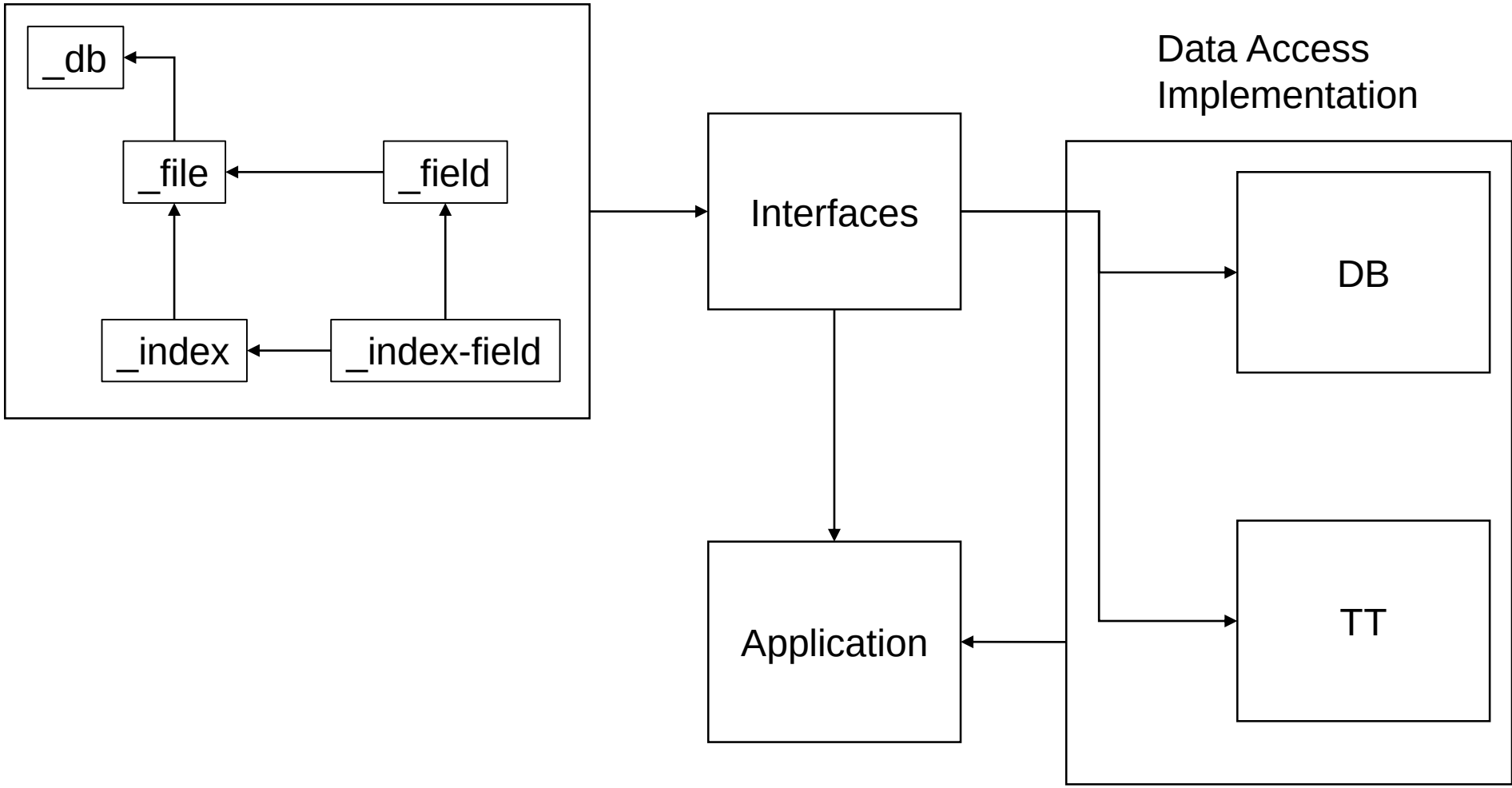
- Contains all table classes.
- Appropriate location for all inter-table Data Access BL
- Appropriate place for DB-level CRUD BL

The Next Step

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.

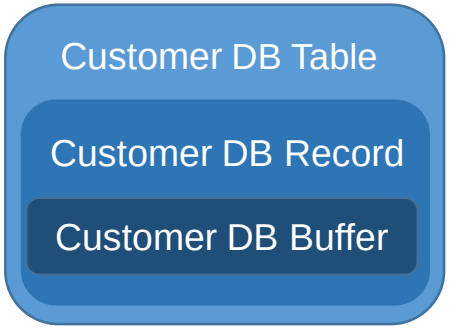


From Theory to Practice – My Experience

- 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

The Model – From Theory to Application

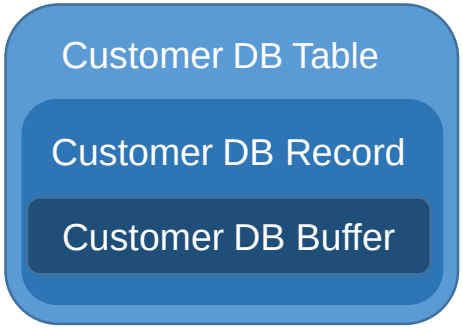
An overall view of the final object model:



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

Storage

- A buffer is a pointer to a record

Run Time

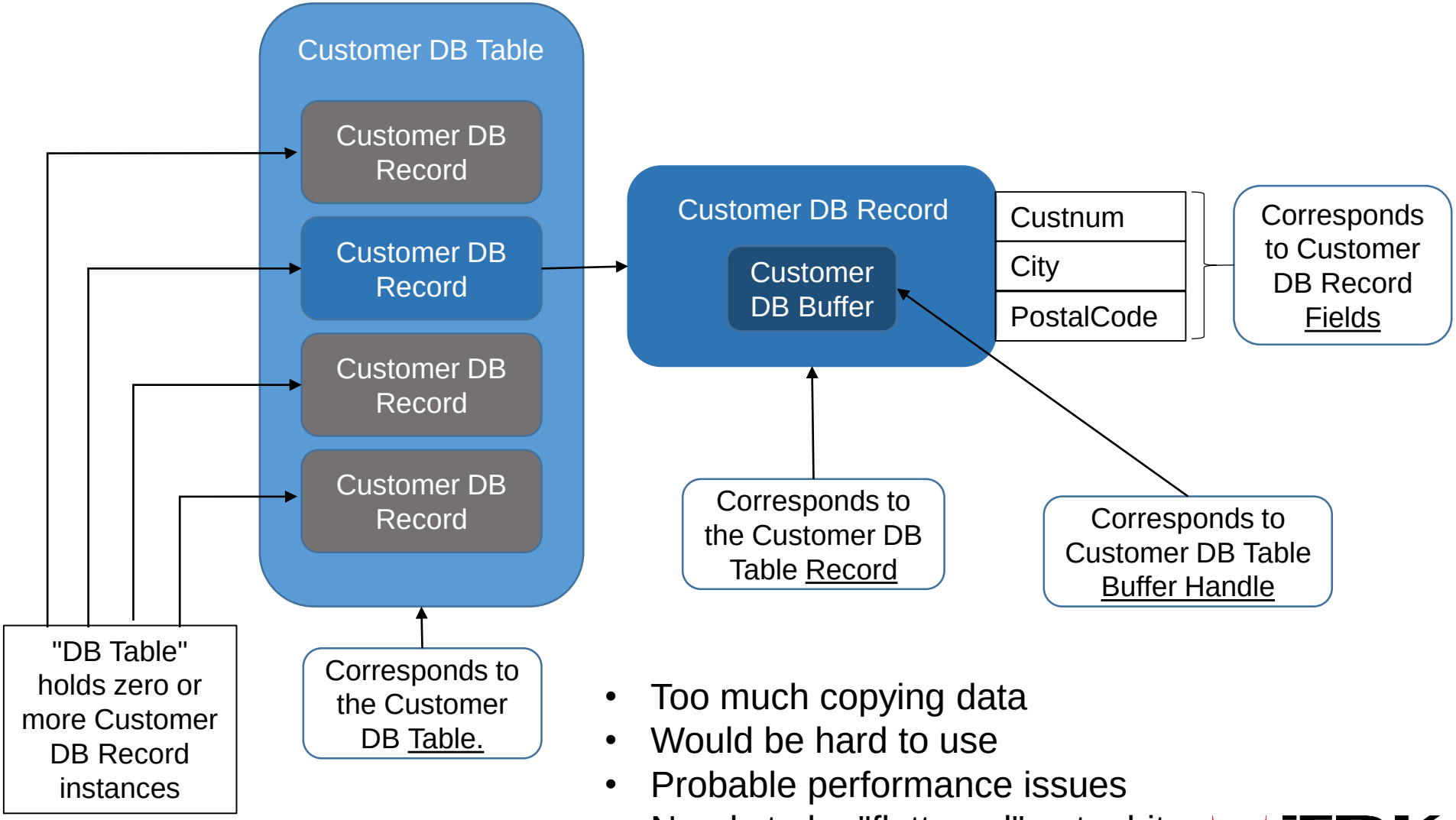
- Tables have relationships

Architecture

Implementation Overview

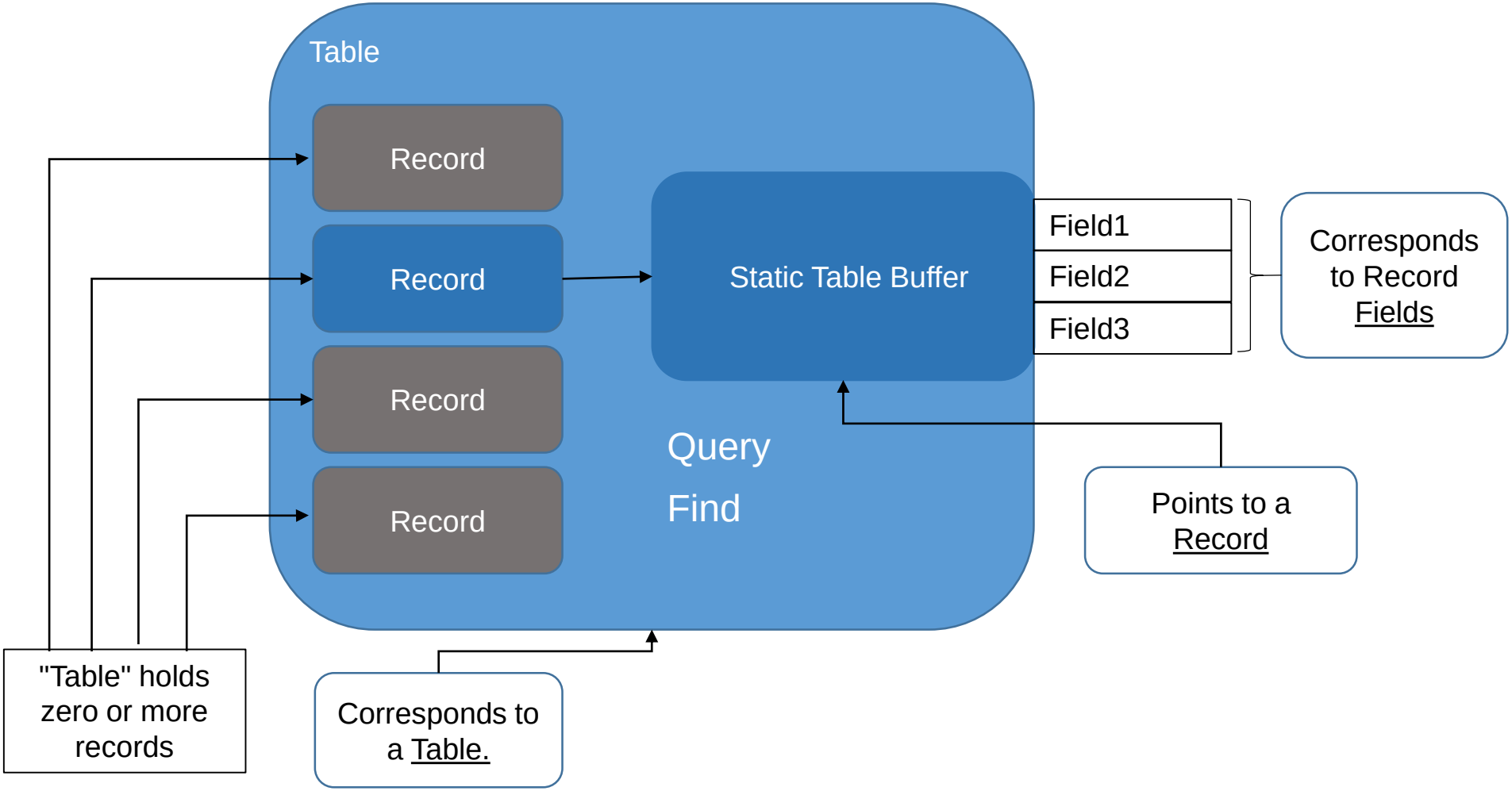
- Possible to completely separate the interface from the implementation
  - All data access structures programmed to an interface
  - Able to swap between a TT and a DB as needed!
- Some encapsulation "pros" and "cons"
  - Abstracts some common operations into the object
  - Limits what you can do with 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



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

A Flattened Out OO Structure



The Target Application

Each record will have:

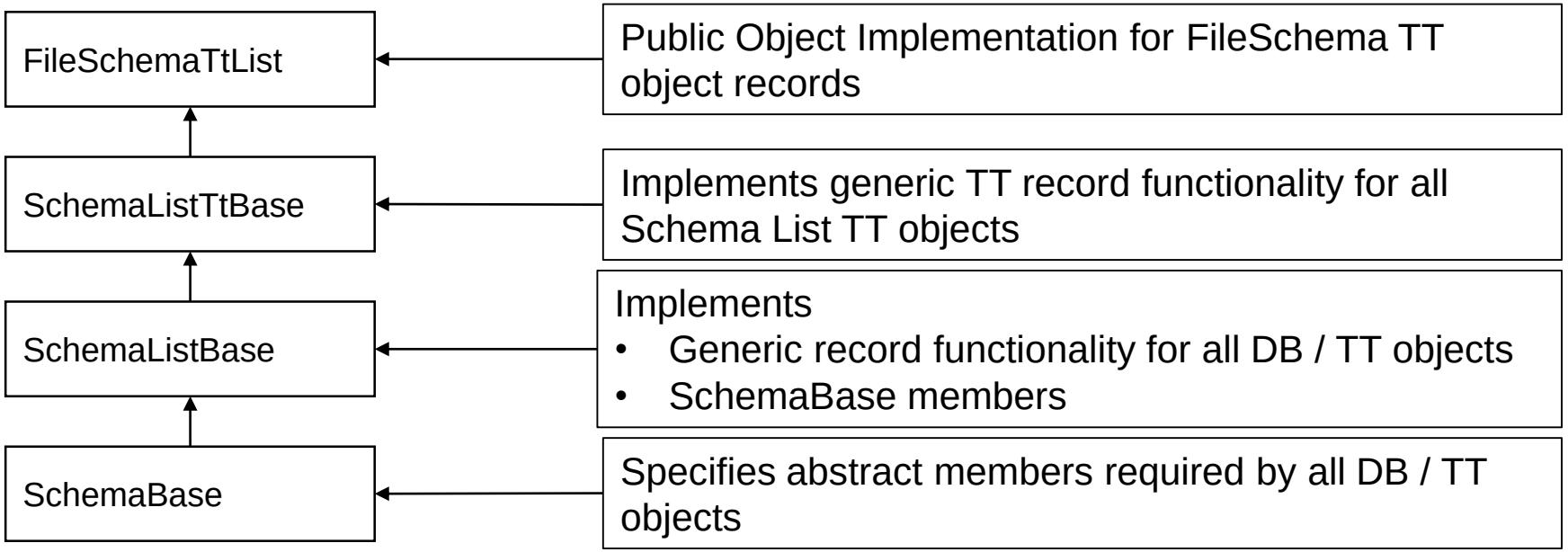
- Fields that are "key" (DbRecordID) and "not key" (everything else)
- A set of "key" fields is an "identifier"
  - "DbIdentifier" = DbRecordId
  - "FileIdentifier" = DbRecordId, FileRecordID
- Keys, Identifiers, and their access members a specified in their own interfaces
- Every object implements its 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

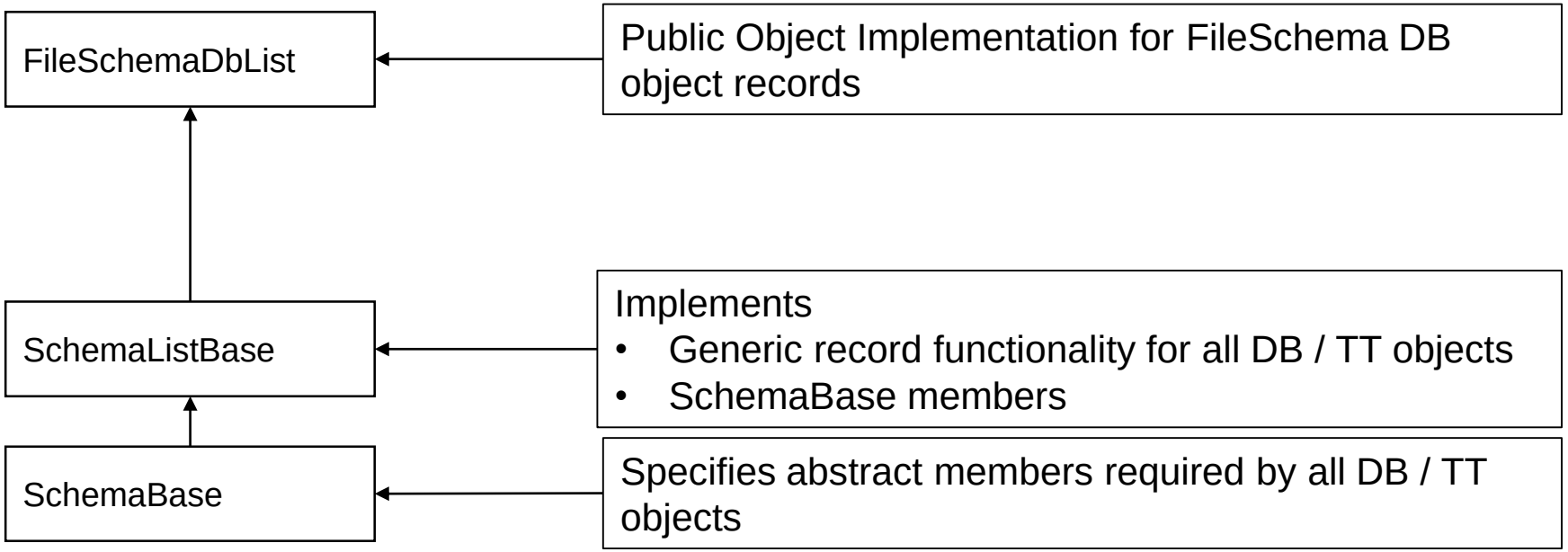
Building the Object Structure

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



Building the Object Structure

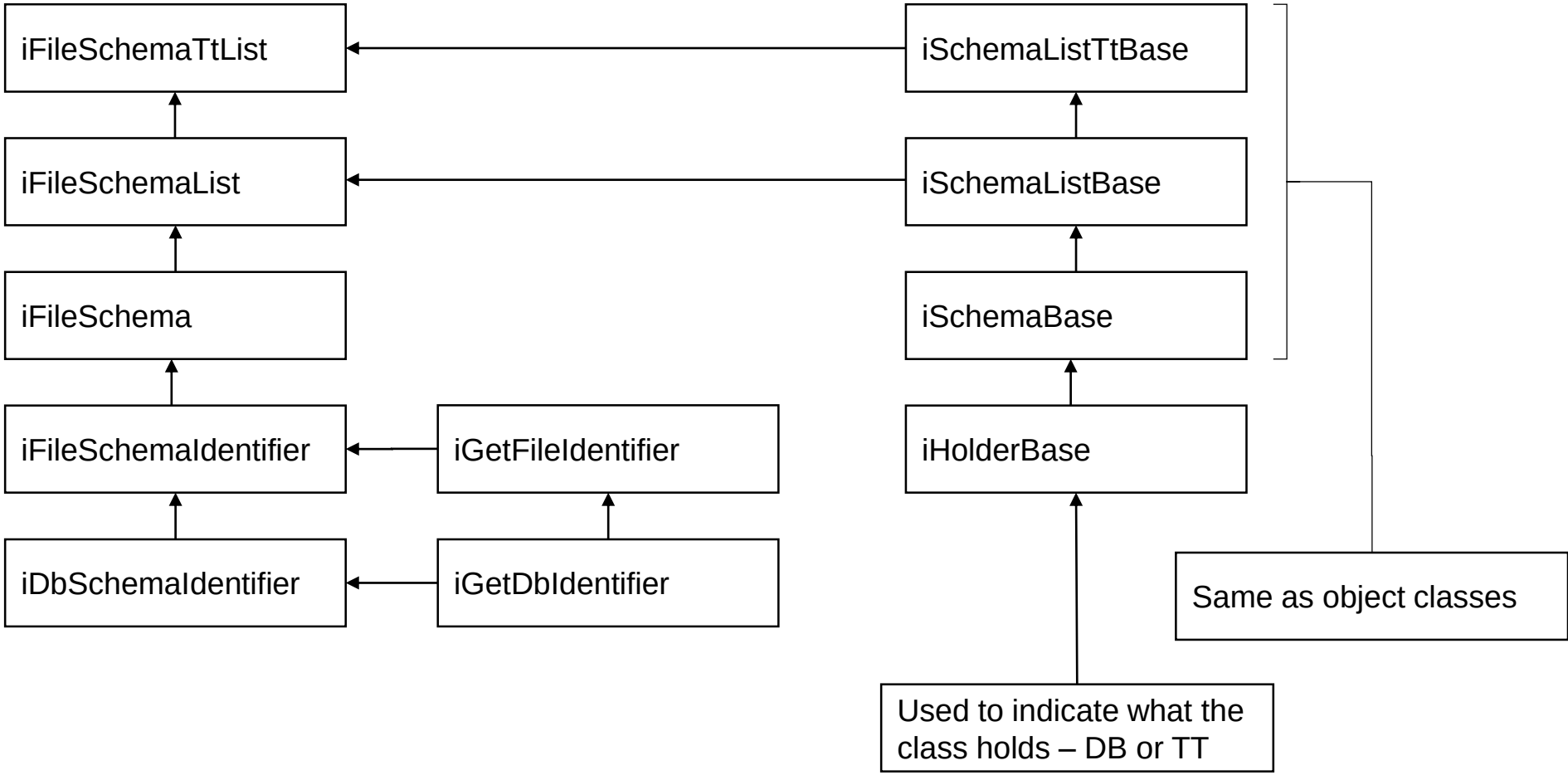
Object **Class** Inheritance for a **File DB** object



now for the interfaces…

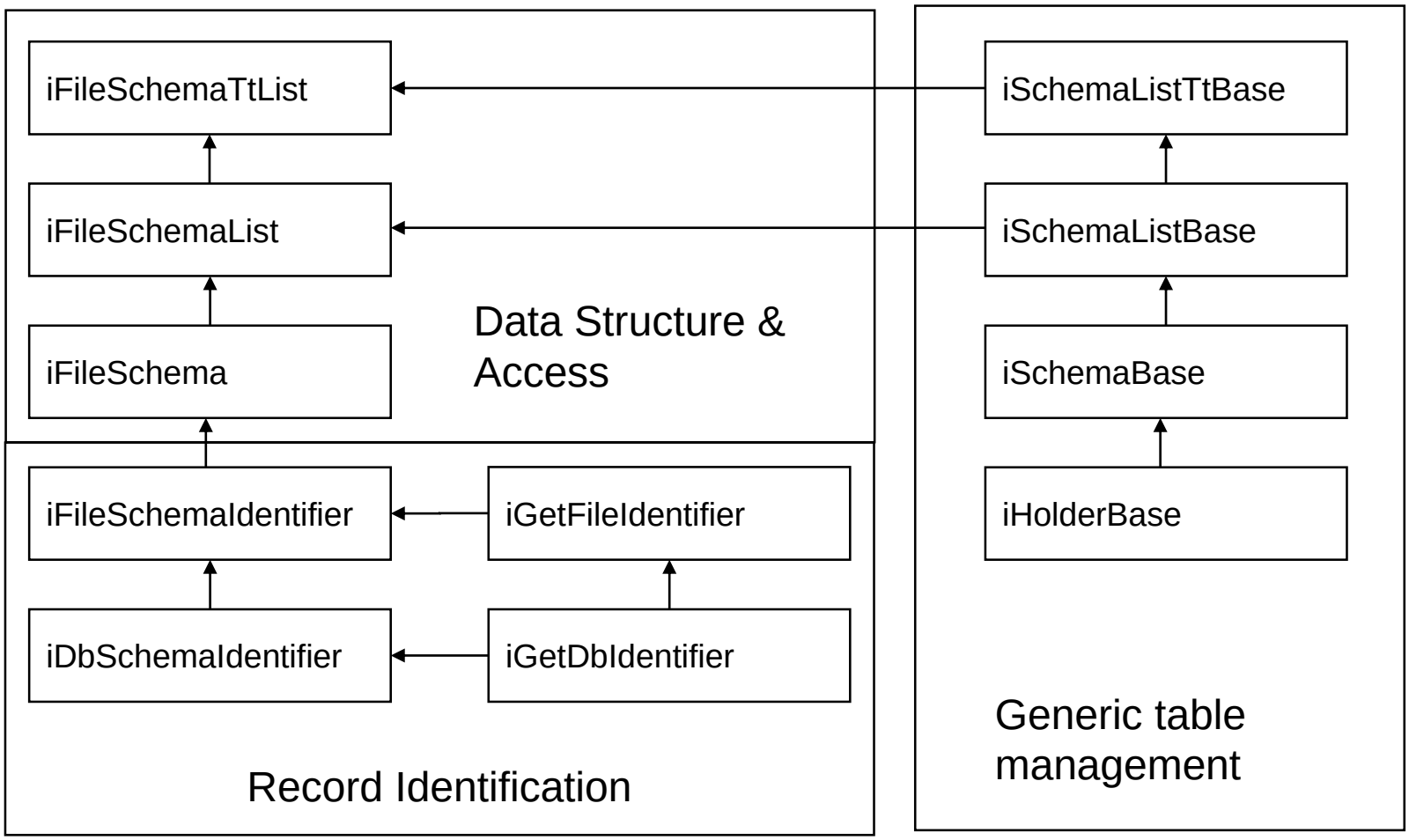
Building the Object Structure

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



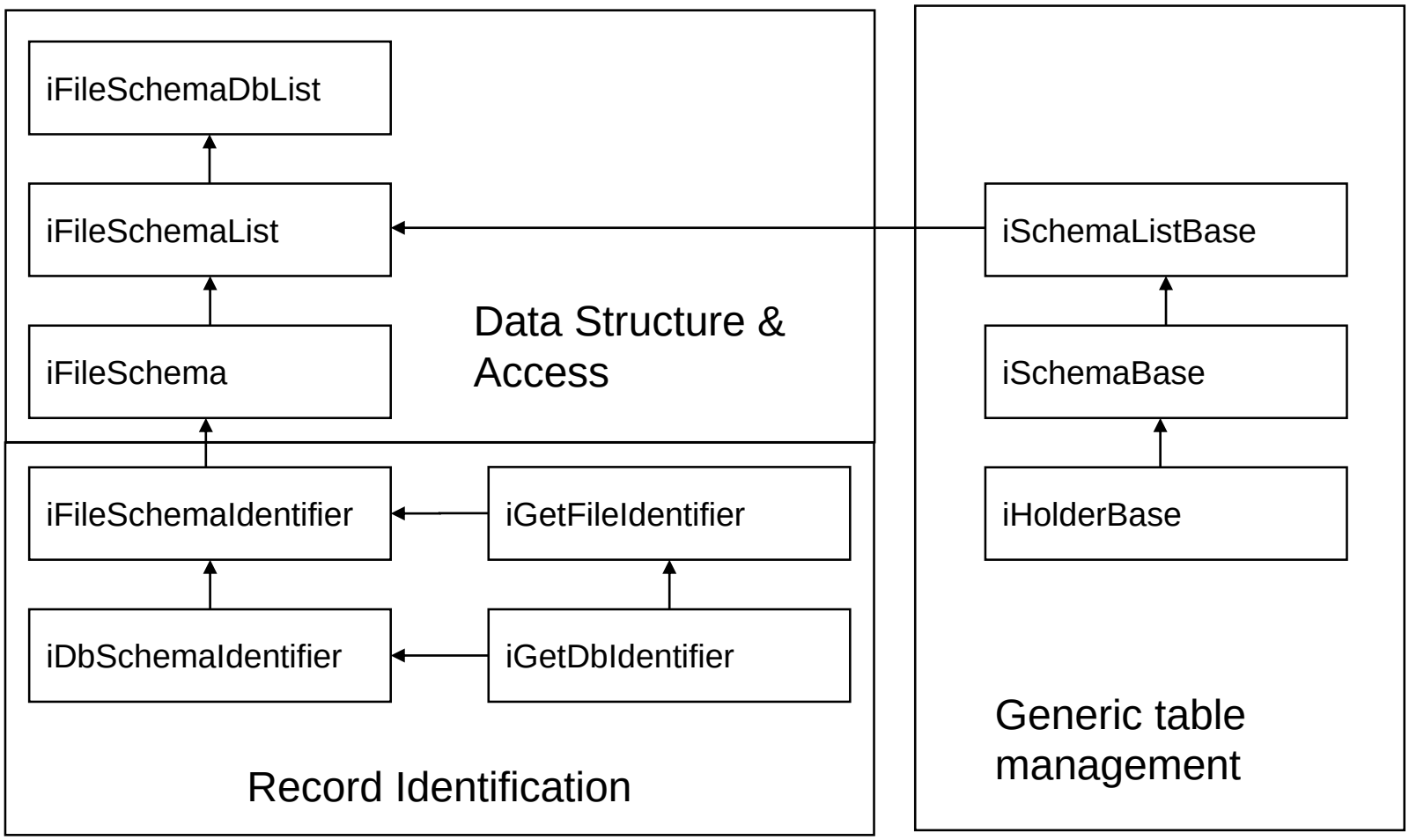
Building the Object Structure

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



Building the Object Structure

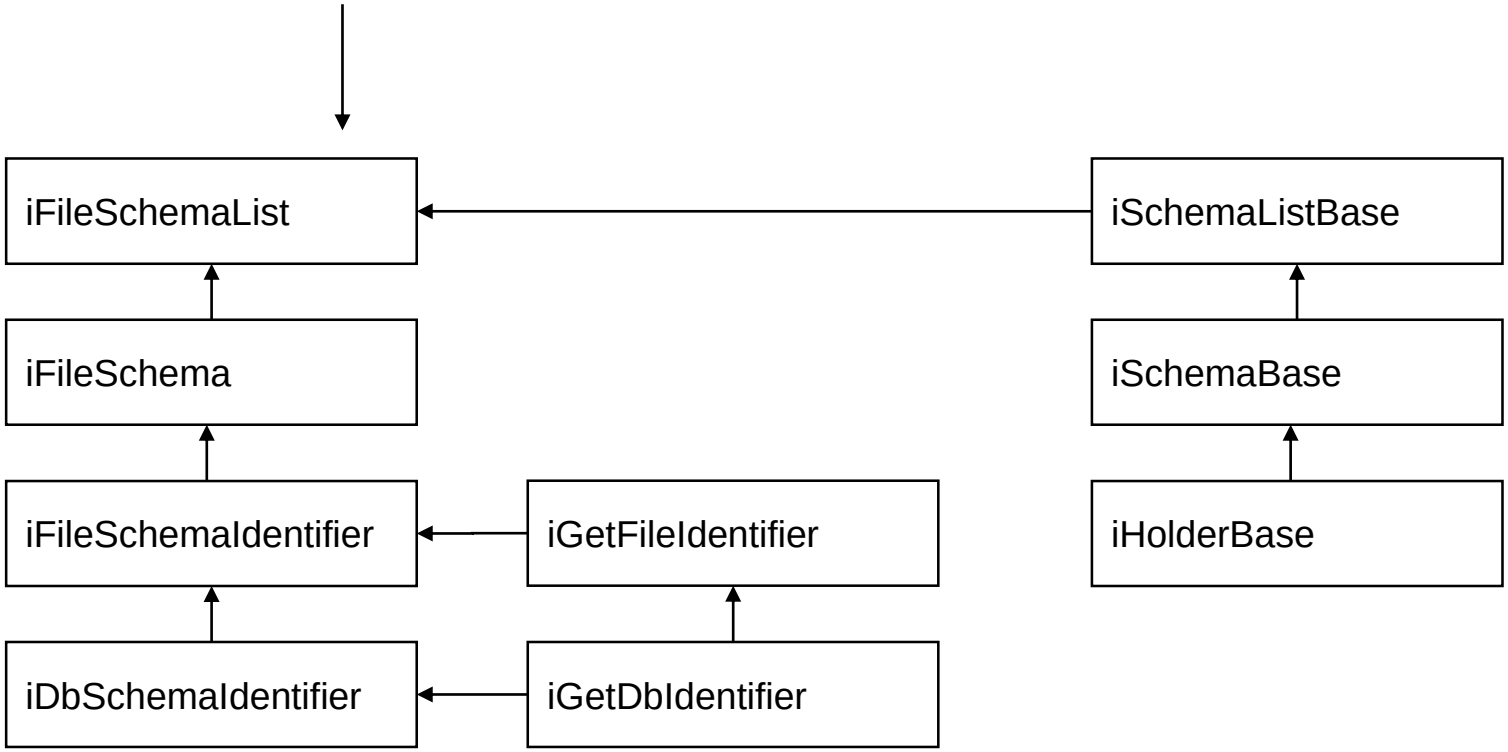
Object **Interface** Inheritance for a **File Db** object



File Table Access Interface

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

Program all table access
to this interface



A Code Example

"File" table access object for local use

Holds an assy of table access objects

Get a local copy

```
oFileSchemaList = THIS-OBJECT:FileSchemaAssemblyHolder:FileSchemaList:Duplicate().
oFileSchemaList:SetFileFilter(THIS-OBJECT:FileSchemaIdentifier).
```

Filter to records that match the identifier

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

```
oFileSchemaList:GetFirst().
DO WHILE oFileSchemaList:Available():

    /* do stuff */
    oFileSchemaList:GetNext().

END.
```

Traverse Records

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