Virtualize your data structure

Presented by: **Brent Raymond**

INTRODUCTION

This session explores a development strategy we at the Art Institute of Chicago call the Virtual Structure. This approach helps you to centralize data query and retrieval, control application access, and leverage generic code which results in reduced codebase maintenance and better performance while satisfying complicated functionality requirements. Virtualize your data model and define its access characteristics using 4D Objects to deliver a cohesive and scalable application.

BACKGROUND

We are all familiar with the 4D database Structure. This is the most intimate view of the inner workings of a data intensive application. Most of us developers can 'see' in our minds exactly where a given table is in the structure view, its relations, similar tables, and recall at least some of its development history. An understanding of your application at this level and being able to visually map it in this kind of view is invaluable and, in fact, altogether necessary to develop its features at macro scales and even right down to the simplest design decisions.

However, what can a database structure tell us on its own about the functionality of an application? What innate functional characteristics of your data structures… tables, fields, blobs, objects… what information can they impart that would link them directly to the business end of your application? For that sort of perspective, you would need some serious local knowledge… like knowing the highways, back roads, subdivisions and region of a city (like Portland or Paris). You would need to know the code, naming conventions, have diligent commenting history, be familiar with the user interface and all of the paths that it can take you down, or you would at least need to know the right person to call to ask (if they can still remember what they were thinking when they created that environment. Good luck with that one!)

Now, not to minimize the relative value of the 4D Structure data entry controls (like Mandatory and Allow Choice List), Comments, and being able to color code tables in the Inspector and not to say that you do not need the aforementioned knowledge and conditions, but in sophisticated applications, things get complicated quick. Smart architecture decisions scale up and can have huge importance as application load increases. Unfortunate design pitfalls scale up to a bunch of constant maintenance costing you the opportunity to add new features among other improvements. So you better have a good map and a strategy on how to manage data access and its functional role!

Developers at the Art Institute of Chicago (AIC) have leveraged a strategy we call the Virtual Structure (VS) to organize their custom application for managing the information and activities of the museums extensive art collection. I have been a (captive) software architect there for over seven years now, so I like to think of the application, named CITI, as my baby now. However I owe the majority of the design and code architectural decisions to the original developers, Mike Blossom and Stefan Lucke, both of whom have mysteriously moved on from 4D development work… oddly to non-programming careers even. The thought of it! Yet the museum owes most of the application design of CITI to the likes of David Adams and his writings on good practices for code and data architecture. As recently as last year, check out his posts on the iNug on how to implement a custom REST API in 4D applications (http:4d.1045681.n5.nabble.com/Publishing-REST-Services-from-4D-td5741604.html). I like to think we have taken the ideas in some other interesting directions though.

DESCRIPTION

The Virtual Structure is a general programming concept, not specific to 4D or any other language. It is the strategy of creating an abstracted layer of data structure and access definition over the actual 4D structure. At its most base element, a Virtual Structure is a list of tables to access and what the fields are that constitute the records in the tables. None of tables and fields actually need to have any direct link or representation in the 4D Structure, hence the 'virtual' part of the name.

However, the strategy involves more than just creating a sort of data dictionary. A virtual structure catalog not only characterizes fields and tables but also references other properties describing how the data can be classified and accessed. Properties of virtual structure entities could include:

- Label (can depend on application mode, field/ table manner of exposure)
- Functionally where it appears in the user interface
- How it is queried/ search comparators
- From what data/ formula is it created
- Relations and control lists
- Security restrictions/ parameters
- Business rules
- Properties of actual interface rendering

See the Case Study section below for actual examples from our CITI application.

Those properties alone provide interesting opportunities for creating generic code that can be leveraged widely throughout the application for data query and retrieval, interface event handling, access control, and API design. Once you have cataloged your data and application flow sufficiently, often patterns emerge that can be taken advantage of. Certain functionalities become easier to isolate into a module with shared methods. Recurrent coding structures can be refined into fully generic code that draws field specific values and handling from the Virtual Structure registry.

Furthermore once the functionality is clearly defined for these groups of similar data structures and standards in the usage of certain 'types' of methods are established, the developer can effectively start to be able to control the function of their application and create new features and data types application wide simply from flipping switches in the Virtual Structure definition code. Take the idea one step further and define application modes in which sections of structure registration settings get included or not, activated or deactivated, or even tweaked depending on circumstance, and you can deploy an agile codebase to a variety of functional roles without maintaining several different 4D structure files.

Again, see the case study section below for actual examples from our CITI application.

ANALYSIS

Let's look at the Virtual Structure strategy by utilizing a common project management technique called SWOT analysis. From Wikipedia, "A SWOT analysis (alternatively SWOT matrix) is a structured planning method used to evaluate the strengths, weaknesses, opportunities and threats involved in a project or in a business venture." It is pretty easy to understand from the following graphic:

Figure 1. SWOT diagram from Wikipedia.

Strengths

- Naming and data constructs are independent from the 4D structure. Tables do not need to be tables; fields do not need to be fields. It is ultimate separation of the data layer from the UI layer.
  David Adams: "Allows for changing names of 4D structure fields and tables without altering or interrupting the structure that the user sees in the interface/ API."
- Allows for easier user navigation of complicated multi-table relation chains.
- Limits "structure bloat" and number of fields in each table (physical data storage vs. logical).
- Unifies data presentation in complex/ rich interfaces. Field rendering is controlled from one location instead of in each occurrence in the interface/ API.
- Simplifies data access in the code ultimately. Query and retrieve table data as if all fields are in one table when, in fact, they may not be or even be fields at all.
- Facilitates access of data from external sources/ databases by keeping a unified interface in the front end while obscuring back end data sources and business logic.
  David Adams: "The data source can be a calculation, function call, result of a call to a remote system, etc. That's up to you inside 4D and it makes no difference (and is invisible) to the outside world. Good and wholesome design."
- Self-documenting for API access due to registration process. The Virtual Structure can be used to expose a data structure API to external access (SOAP, REST).
  David Adams: "If you store your dictionary in data, then you can publish it. So, your (virtual) Customer table can be presented in live docs with whatever attributes you want to share. I love having docs that are driven by the same data as the system because then they're always right."

- Centralized data access allows optimizations to have system wide impact.

Weaknesses

- Built-in 4D user facing interfaces such as Query Editor, Quick Report, Labels no longer applicable. The tables and fields are not referring directly to your Virtual Structure.
- Direct external SQL access to your 4D database also circumvents the Virtual Structure. It would no longer be useful for user facing data access.
- Data retrieval tends toward per field basis which can have some performance cost.

Opportunities

- Facilitates ability to run the application in different modes controlled by switches/ environment.
- Organizes the application for more advanced security features like an access control system/ field and record based privileges per user.
- Enables easier interface exposure of data stored in Object fields. In the user interface an Object field property can be exposed as a standard field to simplify the interface in the front end, but optimize data architecture decisions in the back end.
- Allows for generic coding approach to data access and shared code usage in similar modules.
- Scales well with data requirements. Effectively reduces workload due to centralized data access and leveraging of Virtual Structure properties in creating generic code.
- Use 'Execute On Server' method property to easily handle all data query and retrieval on 64 bit 4D servers.
- Inherently better data security due to abstraction of data access. A Virtual Structure does not expose actual data sources/ field and table names.

Threats

- Additional work to implement beyond simply exposing 4D structure and needs diligence to maintain.
- A steeper learning curve for new developers. Due to centralized data query and retrieval, developers will utilize your custom framework more than direct 4D language calls. Also, a thorough understanding of how to use pointers is a must for working with generic code.
- Is an advanced concept that can be difficult to explain to those with less systems engineering experience. In fact, it can be a challenge to explain to anyone without some discussion!

The strategy is not right for every developer or 4D application, but I believe that it has much to offer if one chooses to go down this route.

CASE STUDY AT ART INSTITUTE OF CHICAGO

For a better understanding of what the Virtual Structure approach has to offer, let us take a look at the CITI application at the Art Institute of Chicago. Our main business is art. So one might assume our artwork Objects table to be the common behemoth of a table where developers keep stuffing information instead of making new tables. In fact, our Objects table only has 63 fields, most of which are only used for a special purpose in one of our application modes. However to the user and API, it appears that the Object table has 347 fields. Instead of seeing 231 tables, users only are aware of 9. Instead of asking the user to navigate through complicated relations or have them write formulas for queries and sorting, we use the Virtual Structure to mask the complexity of the data structure.

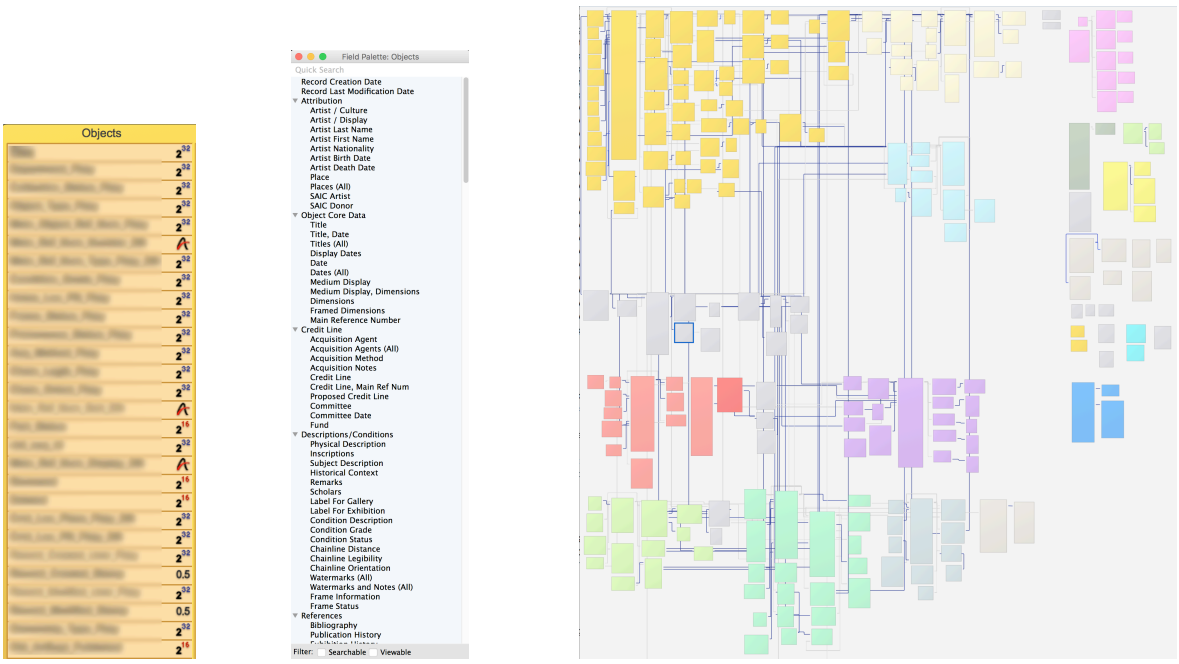


Figure 2. 4D Objects table (63) vs. VS Objects table (347) vs database structure

As you might imagine from the figure above, the database is heavily normalized. Normalization has multiple functional benefits for databases, but it can be tedious and confounding for a user to understand. Using a Virtual Structure, you can mask the glorious yet confounding database design with a human friendly, flat user interface.

This flat view of the data is of course welcome to the users, but it is also convenient for coding as well. The approach effectively masks the complexity of the data structure and business rule cases from everyday coding. Effectively loading data becomes one custom **Selection To Array** for any actual or logical (calculated on the fly) field.

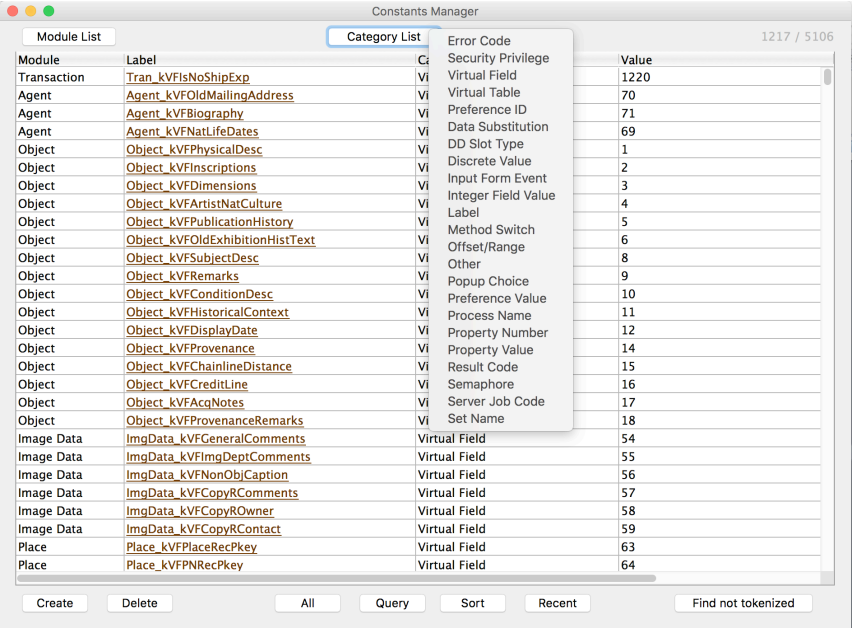


Figure 3. CITI Constants Manager list view.

The best place to get an overview of what fields and tables are in the structure is currently from our Constants Manager. Each of the entities in our Virtual Structure has a constant associated with it that can be used to represent the table or field in code. If combined with a predictable naming convention, constants provide a human readable interface for the Virtual Structure. For more information about how we use 4D user defined constants in CITI, see the demo video at https:www.youtube.com/watch?v=axxLk1HVL14.

However the constants view has been expanded to be more than just a 'database for our database.' The entry window has the following breakdown of what properties have been assigned to each of the Virtual Structure elements.

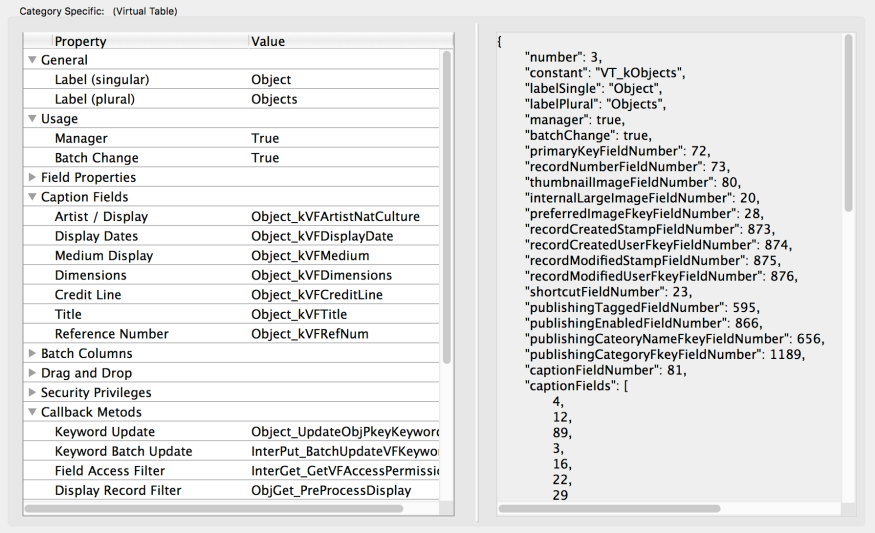


Figure 4. Properties view for a virtual table.

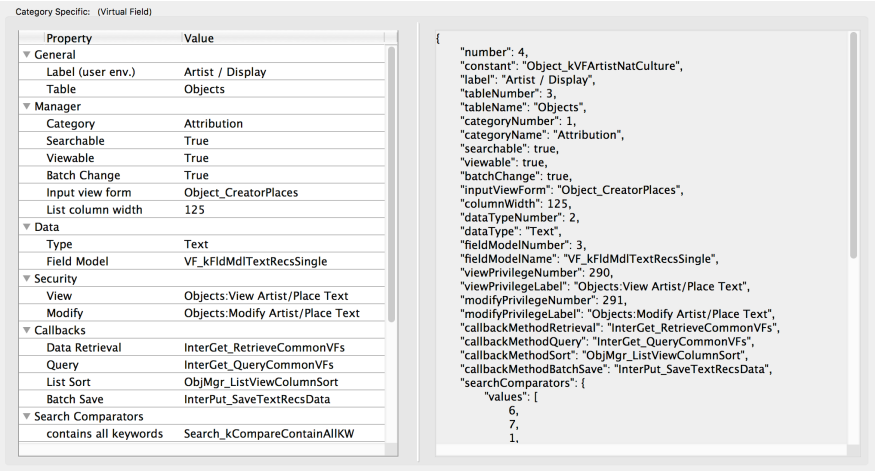


Figure 5. Properties for a Virtual Field

From the incomplete views of the property lists above, you can see that we have found this strategy to be very useful. Most of the properties point to/ are used in a piece of table/ field specific code in the application that has since been made generic. The application pulls functionality logic and interface values from the Virtual Structure properties of each of the structure elements. Out of the 120 separate modules we have in CITI, 44 of them are classified as Generic. Many of those interact with VS to inform their functionality.

For example, data security is a very important aspect of CITI given that its job is to manage data about the Art Institutes extensive art collection. Some users have the privilege to see only some fields for only some records. Control of the privilege settings is handled in the Security module.

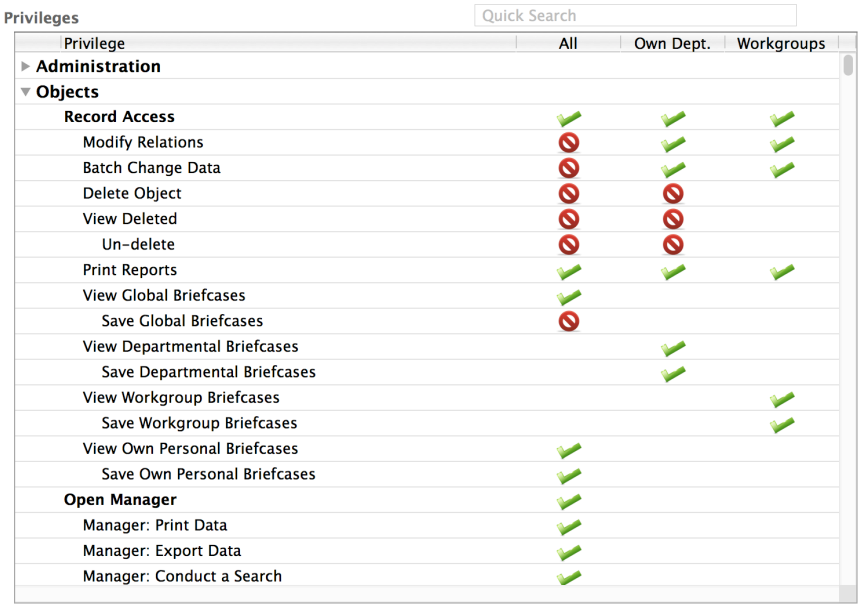


Figure 6. Security Privilege settings.

Each user of the system belongs to a privilege group that has specifically assigned privilege settings according to their role at the museum. Visitor Services will have different access from Curators, which will have different access from the Directors Office, and so on. In fact, the system currently has over 1100 individual privilege settings that can be combined in different ways for different roles. See the following demo video of Welcome to CITI - Users Privileges Workgroups for a more in depth view of the Security module.
https:www.youtube.com/watch?v=kFoz5mVRUDY

So many privileges! How can this possibly be managed without becoming a complete mess? Thankfully the majority of the privilege values are referenced only through table and field registration in the Virtual Structure instead of all throughout the application code. For instance if the application is getting ready to show a selection that includes the Dates field, the first step is to get the 'View' security privilege number from the structure registration. Then call the security module with the current user and the privilege number to determine if the data can be viewed. All of this can happen using generic, non-field-specific code leveraging the Virtual Structure coding strategy.

Here are a few more screen shots of areas in the application that use the structure registration operate generically:

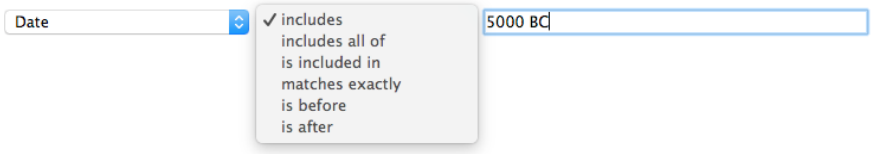


Figure 7. Manager list view search area.

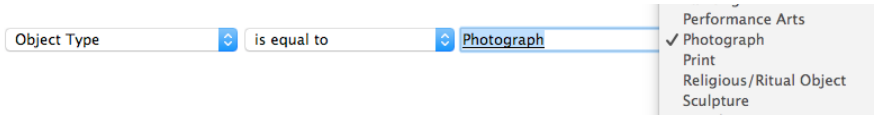


Figure 8. Search using lookup list.

The application retrieves what fields are enabled for search in the current table, checks the security privilege named in VS for viewing this field, pulls a list of registered search comparators from VS for the field, and sets up the entry field according to the field type set in the VS. For fields with a related support table containing normalized field values or a registered list of static choices, the application puts the list to choose from in the entry field and shows the entry as a tokenized value in the interface. When the user finally executes the search, CITI gets the method registered for data query of the selected field and calls it in an execute statement. Normally callback methods that handle a specific task like searching will have a predictable set of standard method parameters, thus allowing various methods to be called interchangeably in Execute statements.

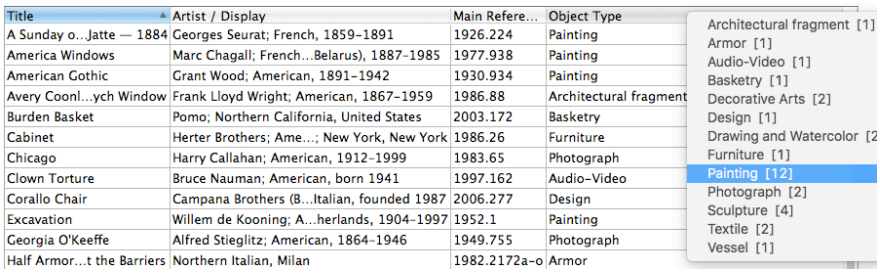


Figure 9. Record list view with filter choice list drop down.

When recalling a list of records, data retrieval callback methods are executed for each of the displayed fields. The callback methods themselves can utilize Virtual Structure properties to navigate registered related tables. VS field properties are used in the data list view to ensure that the field is indeed enabled for View, to get the width of the column for each field, and in this case to show a list of field values to filter the selection. When a selection of records is dragged, VS is consulted to set the Drag Slot Type in order to determine drop-ability in other areas. Given that the entire record manager is not table specific, the code heavily relies on VS for its functionality. For more information about CITI Managers, see the following demo video https:www.youtube.com/watch?v=7LuN5ANhIno.

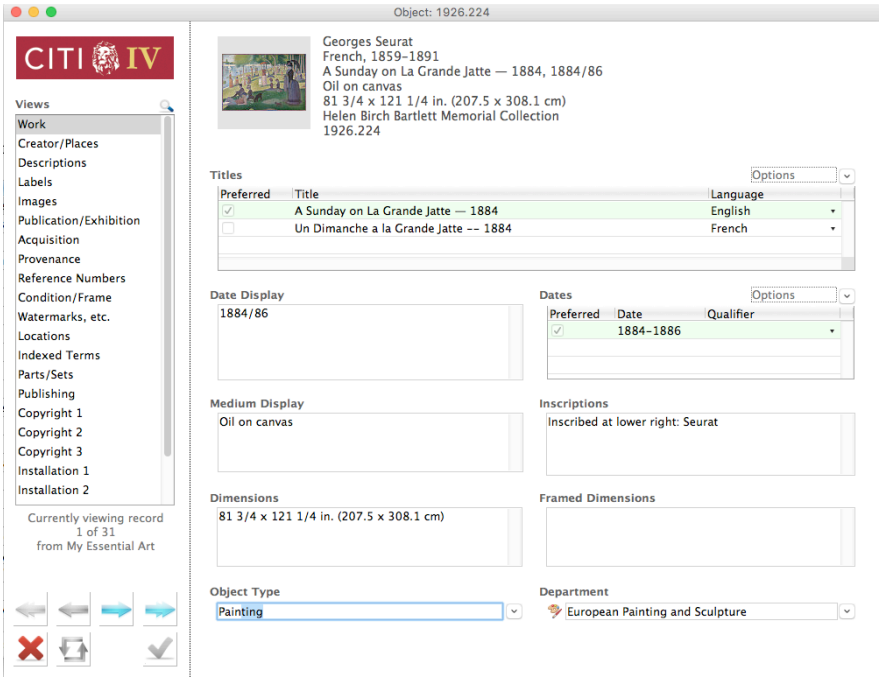


Figure 10. Input window for an art Object.

The record editing window uses VS properties to control a variety of functions generically:

- Checking view security privilege access to form tabs listed in Views section (Work, Creator/Places, Descriptions…)
- Checking view/ modify security privileges for rendering each of the fields on the form. Setting entry objects for fields to read only when appropriate.
- Loading and updating designated caption field for a table (updates automatically when associated constituent fields are updated.)
- Loading and validating choice lists in drop-downs and type ahead functionality.
- Handling business rules for data entry such as mandatory fields and multi-field dependencies.
- Providing a generic 'Go to field' function to load a view with a selected field and position the cursor in that field.

Handing off standard functionality to generic code which accesses Virtual Structure properties allows us to enhance functionality in shared code and implement new features for all fields together without touching all input forms in the application. For more information about CITI edit windows, see the following demo video https:www.youtube.com/watch?v=YnoA4UyrE40.

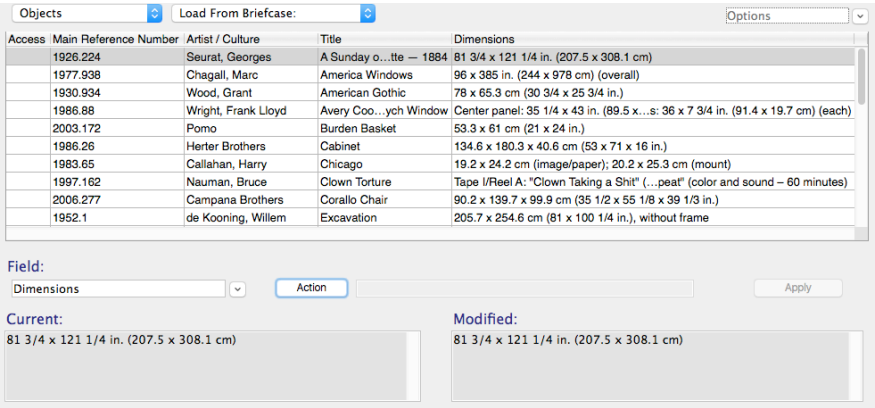


Figure 11. Batch Change.

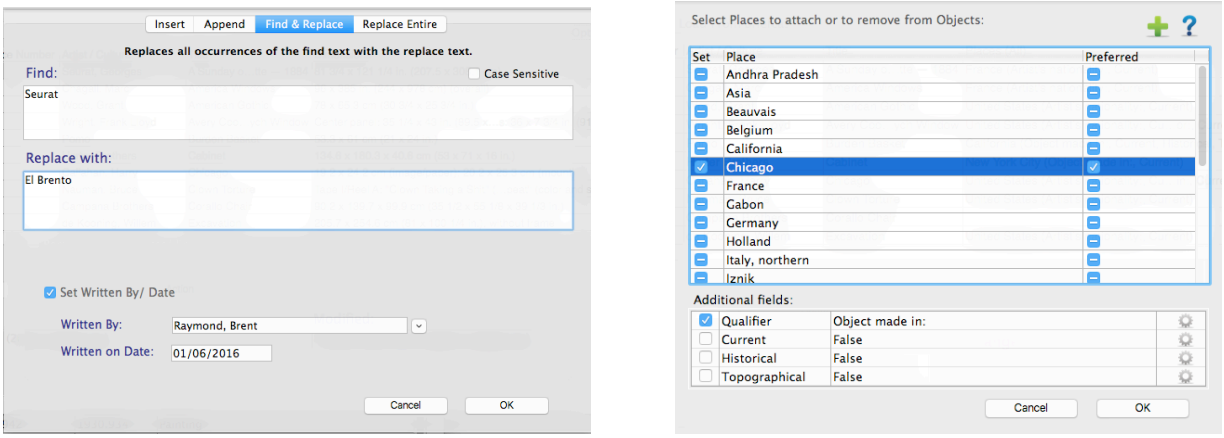


Figure 12. Batch edit setup for text and for a related table.

One of the most powerful applications of the Virtual Structure has been in the development of our Batch Change functionality. CITI offers the ability to set the value for a specific field for a selection of records. This is, of course, a very appreciated feature for a museum with over 200,000 objects in its collection. Properties in the VS can be set to enable a field for use with Batch Change, declare a field model for standardized handling, and set callback methods for setting up a batch change and executing the save. Batch change can handle fields of all data types including complex relations with other virtual tables. All post saving processing code such as updating keywords and publishing data to external repositories is handled generically by leveraging the VS. For more information about CITI Batch Change and Create, see the following demo video https:www.youtube.com/watch?v=QDY5K09VBLA.

Finally lets take a look at application modes. CITI is capable of running in several different runtime modes including:

- Production
- Mirror
- Internal API
- Web Publishing
- Developer
- Test
- Maintenance
- Crash Recovery

The application mode can be determined in most cases automatically by system preference settings and environment. Enabling an application mode not only triggers certain mode specific sections of code, but it can also affect which elements of the Virtual Structure are initialized. For instance, only certain fields are made available

on our external web API, some of which are not available in the Production application mode. All other fields no longer exist as far as the application knows. As well, data normalization can be adjusted depending on mode. Data in the Web Publishing mode is stored 'flatter' than the same data is stored in Production for quicker retrieval. In the mirror and API modes, interface driven data changes are set to be restricted. Use of these application mode settings has enabled CITI to stay a unified code base for multiple different functional roles. Having one 4D structure to take care of is enough!

EXAMPLE DATABASE

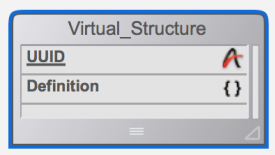
Now for the developer in all of us, let's have a look at some example code to see how the idea can be implemented. Conceptually Virtual Structures fit well with object oriented coding. In older versions of 4D, the Virtual Structure was stored by several synced arrays… one array with the field or table index and several other arrays of varying types to store the property values. Aparajita's Object Tools plugin could have also been used to organize the registration into objects. Since 4D version 14, we have 4D Object variables that offer an opportunity to simplify the registration code and storage of the structure definition within the native language. Whether using synced arrays or Objects, functionally the end result is the same. As the saying goes, 'there are many ways to skin a cat.' What a weird expression, but yea, some ways are easier than others.

Note: Since 4D v14, we also can use SQL Views natively in 4D. The example code does not use this approach, but a combined strategy of SQL Views and a registered Virtual Structure could be a powerful option to explore.

The example database has 3 main modules and 1 utility module for array handling:

- VS – All methods for initializing the Virtual Structure and generic data access
- Manager – Offers a generic interface for viewing table data, search and saving
- Person – A sample main table module that owns the data and non generic data access

VS_Init initializes the Virtual Structure by building 4D Objects that define fields and tables and stores each in the [Virtual_Structure] table.



A sample field registration looks something like this:

```
OB SET($oField;VS_kPropID;Person_kVSFirstName)
OB SET($oField;VS_kPropLabelSingular;"First Name")
OB SET($oField;VS_kPropIsField;True)
OB SET($oField;VS_kPropTableID;Person_kVSTable)
OB SET($oField;VS_kPropDataType;Is text)
OB SET($oField;VS_kPropFieldModel;VS_kFldMdlTextRecsSingle)
OB SET($oField;VS_kPropCBDataRetrieval;"VS_DataGet")
OB SET($oField;VS_kPropCBQuery;"VS_DataQuery")
OB SET($oField;VS_kPropCBSave;"VS_DataSave")
OB SET($oField;VS_kPropCanModify;True)
OB SET($oField;VS_kPropCanSearch;True)
```

As you can see, constants are used to identify each of the properties in code. User defined constants can easily be created and edited using the free 4D Pop component. The above code creates the following 4D Object:

```
{
  "number": 3,
  "labelSingle": "First Name",
  "isField": true,
  "tableID": 1,
  "dataTypeNumber": 2,
  "fieldModel": 8,
  "callbackMethodRetrieval": "VS_DataGet",
  "callbackMethodQuery": "VS_DataQuery",
  "callbackMethodSave": "VS_DataSave",
  "canModify": true,
  "canSearch": true
}
```

Tip: Use the command **METHOD Get path** when setting/ using callback method properties to ensure that the method actually exists. This validation will save many a headache when they are called via the **EXECUTE METHOD** command.

The VS_PropertyList method has a list of the properties that were created for this example database.

General
VS_kPropID
  lonint: unique VS structure ID number (table/ field number)
VS_kPropLabelSingular
  text: singular label for structure object
VS_kPropLabelPlural
  text: plural label for structure object
VS_kPropIsTable
  boolean: is table object
VS_kPropIsField
  boolean: is field object
VS_kPropTableID
  lonint: VS structure ID number for table that field belongs to
VS_kPropPrimaryKeyID
  lonint: VS structure ID number that is the primary key for the table
VS_kPropCanModify
  boolean: VS structure data can be modified
VS_kPropCanSearch
  boolean: VS structure data can be searched

Data
VS_kProp4DTableNumber
  pointer: actual 4D table number if available
VS_kProp4DFieldNumber
  pointer: actual 4D field number if available
VS_kPropDataType
  longint: data type of field (field)

Manager
VS_kPropUsesManager
  boolean: true if table can be used in manager

Callaback methods
VS_kPropCBDataRetrieval
  text: method to call for data retrieval of a field
VS_kPropCBQuery
  text: method to call for query of a field
VS_kPropCBSave
  text: method to call for save of a field

These properties only apply to usage in the example database, however might spark your imagination as to what kind of properties might be useful in other applications. The property list can be easily be extended later on when you think of more interesting values to store.

The VS module offers a few methods for accessing these properties:

VS_GetProp (text; longint; pointer)
  Purpose: Get property value for given virtual structure
  Parameters:
  $1 - (text) property

$2 - (longint) vs id
$3 - (pointer) <- to variable of matching type to receive value

VS_FindVSsWithProperty (text; pointer; pointer)
Purpose: Get a list of VS IDs that have a certain property value.
Parameters:
$1 - (text) property
$2 - (pointer) -> value to search for
$3 - (pointer) <- longint array receiving matching VS IDs

VS_PropertiesToArray (text; pointer; pointer)
Purpose: Retrieve a list of property values for a list of VS IDs
Parameters:
$1 - (text) property
$2 - (pointer) -> longint array of VS IDs
$3 - (pointer) <- value of property for each of the VSs

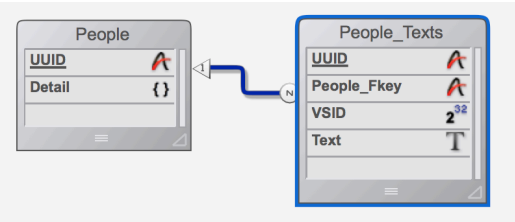
Finally, the VS module offers a few generic data handling methods:

VS_DataQuery (longint; pointer)
Purpose: Search a table on a field with a given value
Parameters:
$1 - (longint) VS field ID
$2 - (pointer) -> to value

VS_DataGet (longint; pointer)
Purpose: Get data for a given VS field
Parameters:
$1 - (longint) VS field ID
$2 - (pointer) <- array to receive data

VS_DataSave (text; longint; pointer)
Purpose: Save data for a given VS field
Parameters:
$1 - (text) record key
$2 - (longint) VS field ID
$3 - (pointer) -> to value

The People module has 2 tables and one DataGet method with the same parameters as VS_DataGet so it can be used interchangeably in the Manager modules generic code.

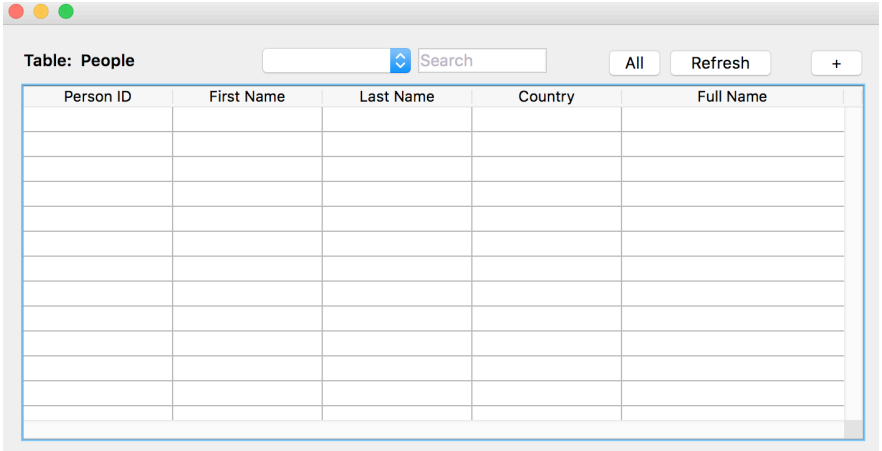


Person_DataGet (longint; pointer)
Purpose: Get data for a Person field
Parameters:
$1 - (longint) VS field ID
$2 - (pointer) <- array to receive data

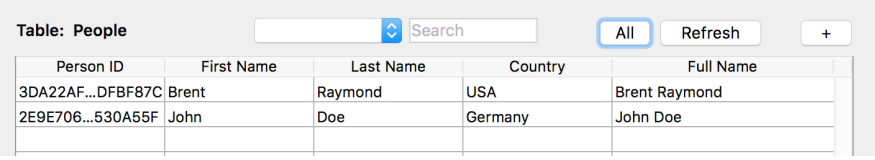
Finally the Manager module and a disclaimer… this module is not meant to be production ready code. It is meant to demonstrate how one can use the Virtual Structure strategy to create and control generic interfaces.

Launch a Manager instance for the People virtual table by running Person_OpenManager which simply calls:

```
Manager_Launch (Person_kVSTable)
```



Manager uses the Virtual Structure to check that this table is registered for use with Manager, determine what the table name is, what fields belong to this table, what field is the primary key, what fields are searchable, and what fields are modifiable.



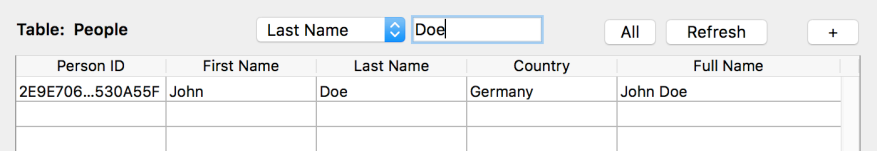
The interface dynamically adds the fields to the listbox with their registered field names and enables editing and saving for those that modify is allowed. Manager uses the registered callback methods for each field for any data retrieval, search, and saving. With the Virtual Structure that was set up in this example, we have 5 fields.

Person ID – A main table field which points to the UUID for the [People] table.
First Name – A text field stored in the [People_Texts] table.
Last Name – A text field stored in the [People_Texts] table.
Country – An Object attribute type field stored in the [People]Detail field.
Full Name – A logical field that is formulated by the values of First Name and Last Name.

Manager_LoadData uses the following code to display the table data in the listbox:

```
$bValue:=True
VS_FindVSsWithProperty (VS_kPropIsField;->$bValue;->Manager_alFields)

VS_PropertiesToArray (VS_kPropLabelSingular;->Manager_alFields;->$atLabels)
VS_PropertiesToArray (VS_kPropCBDataRetrieval;->Manager_alFields;->$atCallbacks)
VS_PropertiesToArray (VS_kPropCanModify;->Manager_alFields;->$abModifies)

$bAddColumns:=(LISTBOX Get number of columns(Manager_abListBox)=0)

For ($i;1;Size of array(Manager_alFields))
  $pDataArray:=Get pointer("Manager_atField"+String($i))
  $pHeader:=Get pointer("Manager_lHeader"+String($i))

  EXECUTE METHOD($atCallbacks{$i};*;Manager_alFields{$i};$pDataArray)

  If ($bAddColumns)
    LISTBOX INSERT COLUMN(Manager_abListBox;$i;"field"+String($i);$pDataArray->;VS_kPropLabelSingular;$pHeader->)
    OBJECT SET TITLE($pHeader->;$atLabels{$i})
    OBJECT SET ENTERABLE($pDataArray->;$abModifies{$i})
  End if

End for
```

Fields that have the property VS_kPropCanSearch set to True are loaded into the pop up list at the top of the interface. When one of these fields is selected and the user types into the Search field, the selection in the listbox below is updated.



Generic Search as you type is accomplished with the following code:

```
: (Form event=On After Keystroke)
If (Manager_atSearchFields>0)
  $lVSID:=Manager_alSearchFields{Manager_atSearchFields}

  VS_GetProp (VS_kPropCBQuery;$lVSID;->$tQueryCallback)

  $tSearchString:=Get edited text
  EXECUTE METHOD($tQueryCallback;*;$lVSID;->$tSearchString)

  Manager_LoadData
End if
```

The All and Refresh buttons are there to conveniently select all records in the table or to refresh the data for the records currently displayed.

While the design and basic functionality of this interface may not impress many, it is important to note that the developer can now simply add new fields and tables to the Virtual Structure definition and they will instantly be fully functional in the Manager. In fact, adding new fields with the field model property (VS_kPropFieldModel) set to VS_kFldMdlTextRecsSingle (stored in the fat text table [People_Texts]) or VS_kFldMdlObjectAttribute (stored in the flexible object field [People]Detail) requires no changes whatsoever to the 4D structure to be usable application wide. For instance, the following code will create a new field stored in the [People]Detail 4D Object field and will be fully functional in the interface after initialization.

```
OB SET($oField;VS_kPropID;Person_kVSZipCode)
OB SET($oField;VS_kPropLabelSingular;"Zip Code")
OB SET($oField;VS_kPropIsField;True)
OB SET($oField;VS_kPropTableID;Person_kVSTable)
OB SET($oField;VS_kPropDataType;Is text)
OB SET($oField;VS_kPropFieldModel;VS_kFldMdlObjectAttribute)
OB SET($oField;VS_kProp4DTableNumber;Table(->[People]Detail))
OB SET($oField;VS_kProp4DFieldNumber;Field(->[People]Detail))
OB SET($oField;VS_kPropAttribute;"zipCode")
OB SET($oField;VS_kPropCBDataRetrieval;"VS_DataGet")
OB SET($oField;VS_kPropCBQuery;"VS_DataQuery")
OB SET($oField;VS_kPropCBSave;"VS_DataSave")
OB SET($oField;VS_kPropCanModify;True)
OB SET($oField;VS_kPropCanSearch;True)
```

In fact, the addition of the above field could simply be handled by enabling different VS initialization depending on Application Mode. Change the application mode of the example database in the On Startup database method from “Normal” to “Super” (of course with user defined constants), restart the demo, and see that the Zip Code field is now available in the manager interface.

CONCLUSION

Using the Virtual Structure approach to application development can deliver some very interesting benefits including performance, flexibility, and peace of mind. It encourages developers to write generic code that can access an ever-expanding list of properties from a centralized catalog stored in 4D Objects. This ability can allow an application to handle vastly different requirements with the same code base and scale gracefully over time. Virtualize your data structure to start taking advantage of the benefits of this development approach.