

# Tables and Forms in DataEase for Windows

Lawrence Fox, ComputerWizard Consulting  
©ComputerWizard Consulting, 2000

## Introduction

One of DataEase for DOS's (DEDOS) greatest strengths as a RAD (rapid application development) tool is that the table *is* the form, so that when one is creating the place to store the data, you're also creating the user interface. I refer to this a "FormTable" construct, because it is both simultaneously.

I'd like to propose a radical change in approach when doing development in DataEase for Windows (DfW). Instead of the one-step, unitary approach of DEDOS, I suggest that in DfW you **separate table creation and management from the user interface**.

Create one FormTable to manage the table, (add/modify or delete fields) but create as many forms that use this table definition as you need for the application. Let users interact with these forms. I call the former "table-defining" or "table-owning" forms and the others "table-using" forms.

It's radical approach for those used to the DEDOS model, (but not for those used to some other DBMSs) and is not discussed at all in Sapphire's documentation for DfW, which follows the DEDOS model.

Think of the table—the place where the data is stored—as a different entity than the forms which you use to enter, modify or view (browse) the data. This is an important concept to master, because it leads to thinking about things "at the table level" vs "at the form level"—which I've found to be one of the secrets of successful DfW development. DataEase, however, uses the form metaphor throughout, so there isn't a separate table-definition or modification facility as there is in Access or some other products. You use the table-defining form to manage the table, and other forms to enter, modify or view the data.

There are two main reasons why I argue that this is a good approach:

- Stability
- User Interface

## A Tutorial

Before going any further, a little tutorial is in order. Start DfW and create a new database called "Forms Paper". First, let's create a table-defining form:

- 1) Select **F**ile|**N**ew|**F**orm from the drop-down menu.
- 2) In the New Document Dialogue box, click on <New Table>.
- 3) Type **tblCustomers** in the **Select a Database Table** box.

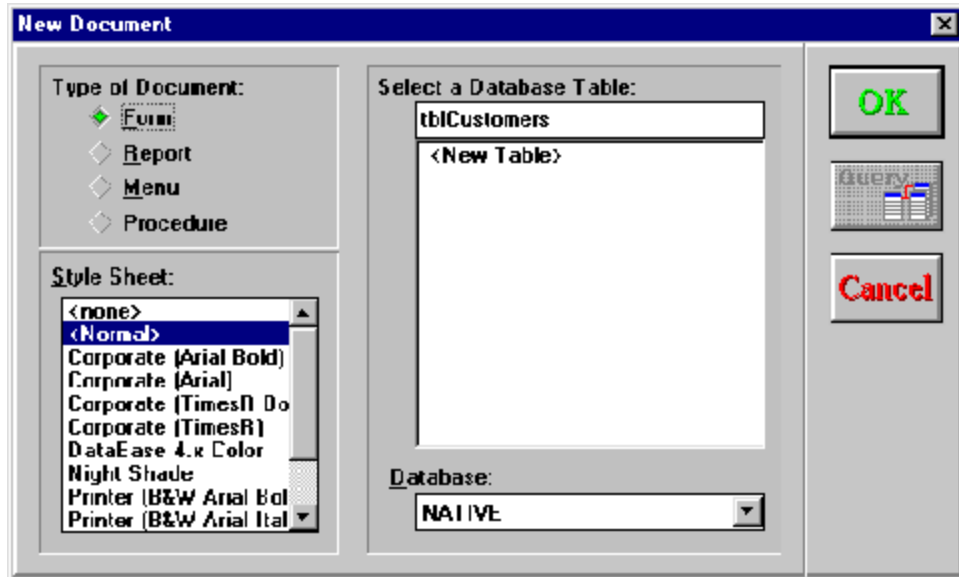


Figure 1: The “New Document” Dialogue Box, when you start creating a “table-defining form”.

- 4) Click on the **OK** button.
- 5) DfW will then display another dialogue box (the **New Table** dialogue box—not shown). Click on the **OK** button.

You now have a blank canvas onto which you can paint the table-defining form. Change the form’s background colour to cyan. Add two fields:

- CustomerCode (text, 5 characters)
- CustomerName (text, 40 characters)

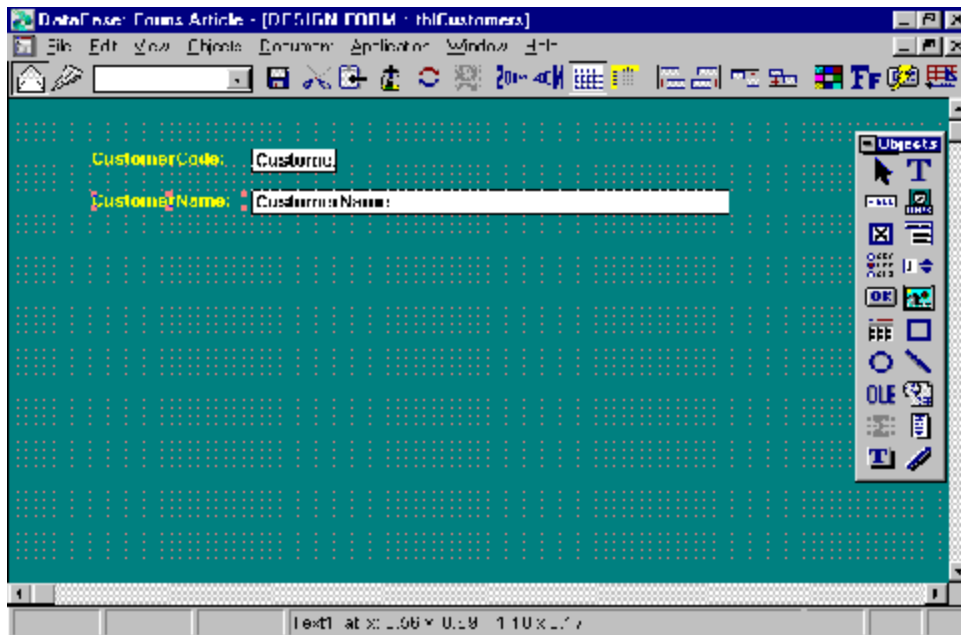


Figure 2: The table-defining form (in design mode) with two fields defined

You have now created a table-defining form!

Close, and then reopen the table (OK, the “table-defining form”), then click on Document|Properties. Look in the upper left-hand corner of the Document Properties sheet and you’ll see that this form is the one that defines the table. (See the closeup in Figure 3). (When in doubt whether a form uses or defines a table, you can always check in here for this information).

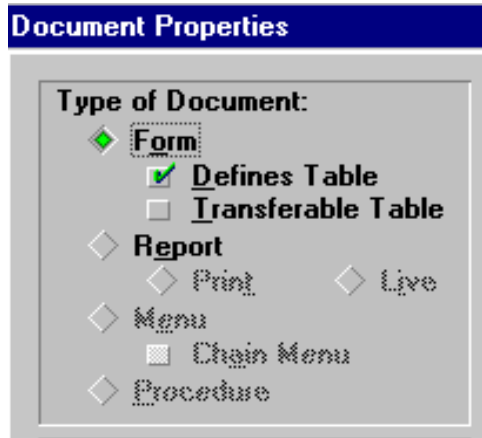


Figure 3: Table-ownership displayed in Document Properties

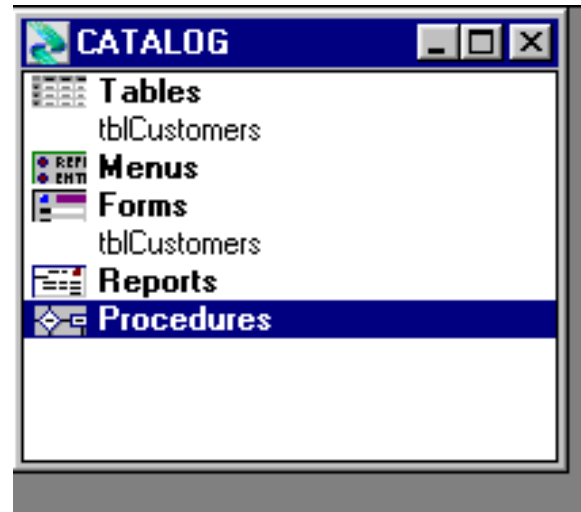


Figure 4: The catalogue with “Tables” expanded

Figure 4 shows a close-up of the catalogue; the table-defining form listed under both “Tables” and “Forms” (normally the “Tables” listing is collapsed and the list of tables is not displayed; to see the list double click on the word “Tables”.)

At the file level you have now created **three** files—**tblCxaaa.tdf**, **tblCxaaa.frm**, and **tblCxaaa.dbm**—where *x* is the database letter and *aaa* may be any three letters; for example, in the database that I created while writing this paper, creating the table-defining form generated **tblCfaaa.tdf**, **tblCfaaa.frm** and **tblCfaaa.dbm**. (Look at the database directory with Windows Explorer to see the files.)

The **tdf** file is the table-definition file, whereas the **frm** file is the form and the **dbm** file holds the data. This set of files is created any time you create a table-defining form. You cannot, however, address the **tdf** file directly—you must interact with it through the intermediary of the table-defining **frm** file.

Now we’re going to create a table-*using* form “over top” of this table. To do this,

- 1) Select File|New|Form from the drop-down menu.
- 2) In the New Document Dialogue box, click on **tblCustomers** (instead of <New Table>), then click on the OK button.

Before clicking on the OK button, you can specify a different style sheet for this form than the for the table-defining form (look at the selection box on the lower left-hand side of the form (in Figure 5); I’ve created a few styles of my own, with one for the table-defining forms (WizardDataTableStyle) and one for the “user” forms (WizardFormStyle1). You can create as

many different style sheets as you want! (Style sheets can be changed at any time by opening the Document Properties form and selecting a new style; I just tend to do it at the same time I create the form).

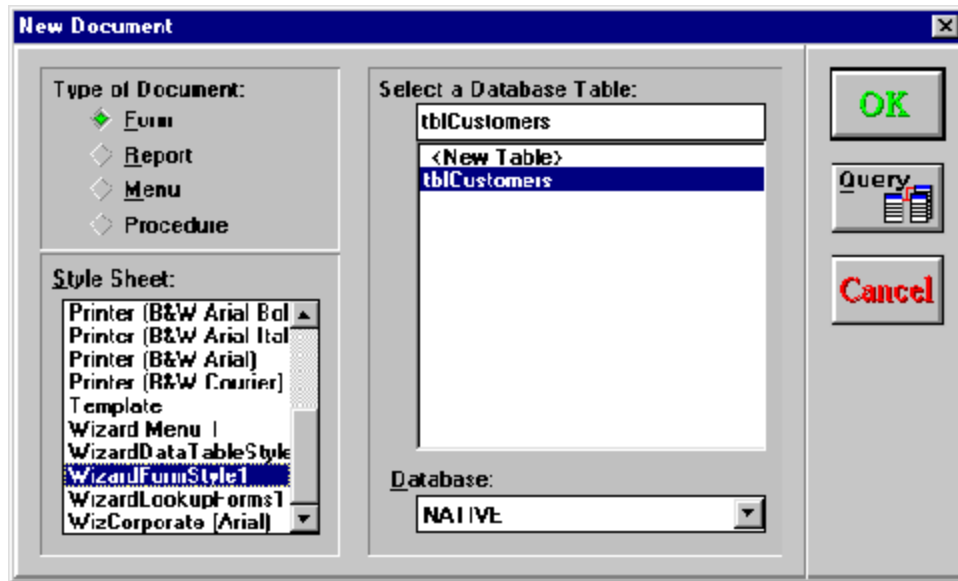


Figure 5: The “New Document” dialogue, but this time we’re creating a form that **uses** tblCustomers, not defines it. Look at the Style Sheet selection box—<Normal> is **not** selected this time

- 3) DataEase will then display the **Layout Options** dialogue box (not shown) instead of the New Table dialogue):

The default layout choice is “Original Form” which means that the table-using form will (at first) look like the table-defining form.

Of course, you’ll want to make it look nicer later on!

Check the on-line documentation for more information on the other layout options.

Click OK to accept this layout and go on to the next step.

- 4) DataEase will now create a form that will look like the original form. Fields will be in the same place, and, if you've used the same style sheet, everything will look the same. Select File|Save As from the menu and enter **frmCustomers** as the form's name in the Document Save As dialogue box, then click on OK.  
(a **Update\_CDF\_Path** that's displayed in Figure 6 is a DQL that's in the my version of this application and has nothing to do with this paper; but notice that DfW mixes forms and DQLS in this dialogue box!)

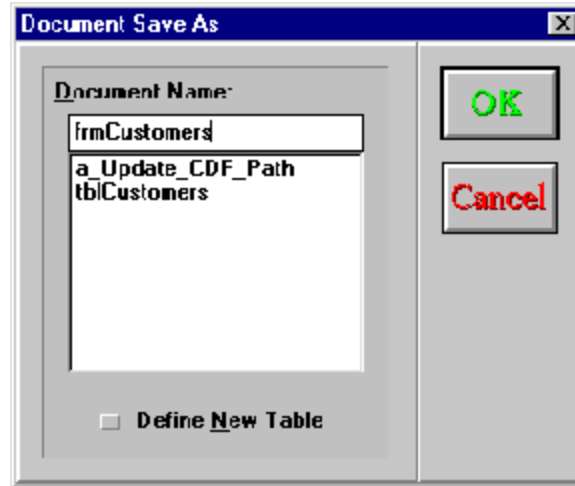


Figure 6: Document Save As Dialogue Box with the form's name

Then I made some changes to the form and saved it again. This is the form that the users will see.

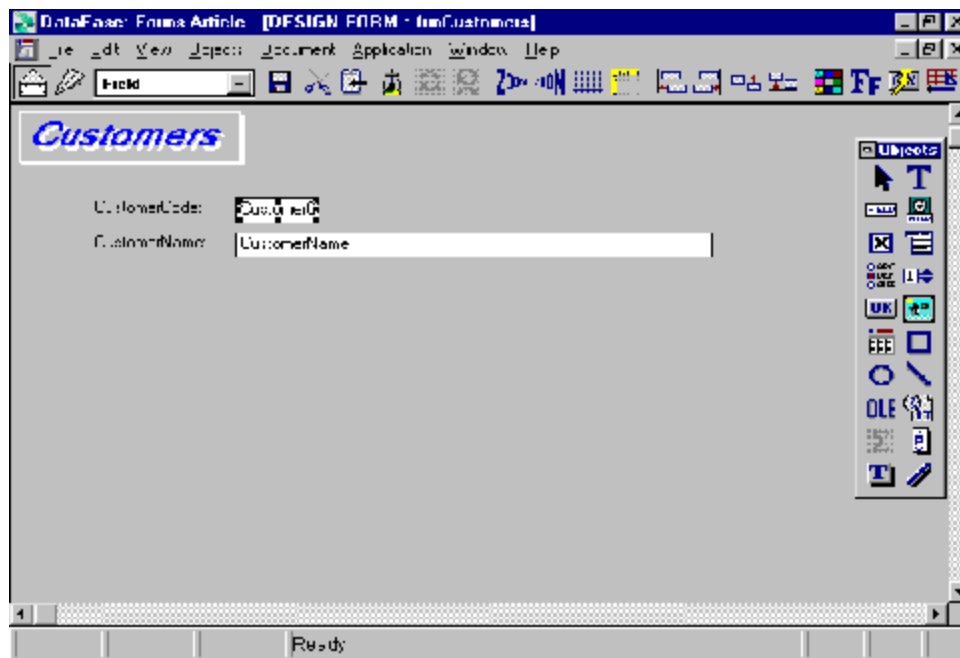


Figure 7: A table-using form ready for adding more user-interface widgets

Now if you look at the catalogue, you'll see three items—a table, a table-defining form, and a table-using one. Notice that both forms cite **tblCustomers** as the primary table:

| Tables       |               |
|--------------|---------------|
| tblCustomers |               |
| Menus        |               |
| Forms        |               |
| frmCustomers | PRIMARY TABLE |
| tblCustomers | tblCustomers  |

Figure 8: Close-up of part of the Catalog showing the table-owning and table-using forms

If you were to open **frmCustomers** in design mode and check the Form Properties form, you'd find that the "Defines Table" option box is "greyed out" and cannot be checked.

Go ahead and enter a few records via **tblCustomers**—then close it, and open **frmCustomers**—and you'll see that all the records are there. Now enter a few via **frmCustomers** (or edit them). Close **frmCustomers** and open **tblCustomers**—and they're there.

If you were to look at the application directory now using Windows Explorer, you'd see a file named **frmCxaaa.frm**—the table-using form that we've just created. Notice that there's only one file created for a table-using form, since it can access the fields in the appropriate data files using fields defined in the **tdf** files.

## A Word about naming conventions

I tend to call my table-defining forms **tblsomething** and my table-using forms **frmsomething** as in the tutorial above. I also preface virtual fields with a "v" (e.g., **vTotalFoobars**) and projected fields (discussed below) with a "p" (e.g., **pMyField**). I don't insist that **you** do the same—just that you pick some kind of naming convention and stick to it. (I'm not in favour of the overwhelmingly baroque system of prefixes and suffixes that Microsoft uses for some of its programming languages!)

## Big deal—what have I accomplished?

As I said earlier, there are two main reasons for this approach:

- Stability
- User Interface

Let's now take a closer look at all of them.

## Stability

DfW **does** GPF or otherwise hang. It's not perfect and it *is* a Windows application. It's also a bit of a resource hog when in design mode, which can also cause a lockup sooner or later. If DfW "goes south" when you've got a table-defining form open in design mode, or if you are using it for data-entry, you can lose the table definition and/or the associated data.

On the other hand, using table-using forms for data-entry seems to reduce resource use and crashes. The same goes for viewing/browsing or modifying or deleting records—all which can be done with the same form. (I don't suggest that you need one form for each activity—unlike some other products!) And even if DfW *does* go south, you might lose the table-using form, but NOT the table-defining one or the data.

Until I made this change, I regularly had problems with DfW.

I have to admit that when discussing this approach to using DfW, the stability issue is the most anecdotal and takes up the least amount of space—but I do think it's an important consideration.

Table-defining forms don't have to be pretty; I generally leave them in a "field per line" format, and simply add any new fields to the bottom. Let's say that in DEDOS you had a table with a bunch of fields in it, and needed to add a new one. Let's also say that the new field should be located between the second and third fields already there in the FormTable. Well, all you'd have to do is press F8 to insert a line, then create the field and save the table. DEDOS would *insert* the field as a new column in the table (in table view), renumbering all of the other fields and related indexes (indices).

In DfW adding a field merely appends it to the table's column order; position on the table-defining form is not relevant (you can see this when looking at a field-picklist when in design-mode—table, form, QBM or DQL). Moving around large groups of fields in order to make the table-defining form look pretty is often a source of GPFs (especially when you've been working on the form for hours and hours; I think it's the universal Imp of the Perverse deciding to strike).

John Middleton posted (on the DataEase Forum at <http://www.plmconsulting.com>) that:

I've now concluded that the CDFs which seem to give the most problems in long term usage (ExecuteFile(), etc) seem to be less problematic (but not cured, i.e., they will function for longer) provided they are fired from a form that does not define a table.

"Resource Problems & Cacheman, 19-Sep-2000"

## User Interface (UI) Issues

Here's where the split between storage and usage really starts to shine. You can create as many forms as you need. Windows is a much more visual medium than DOS, and you often have to spend *more* time on the UI than you do with DOS. When you separate the database structure from the user interface, you can spend more time on getting **both** just right.

You also get fewer reorganizations of tables while designing your user interface. In fact, the only time a table is reorganized is when you add or delete a field to the table-defining form, not when you move one around on the table-using form.

(When you start designing reports in DfW, you'll also find that the model (structure or schema) of your database is much more important than in DEDOS. But that's an issue for another paper, I think.)

Add all of your user interface "widgets"—command buttons, custom menus, (custom toolbars when and if they become available), logos and bitmaps to frmCustomers, not tblCustomers.

There is no automatic synchronization between the table-defining and table-using forms, so if you add fields to the table, you have to add them manually to the form (if you wish to see them).

Let's look at couple of UI issues in detail.

### **Multiple Forms over the Same Table(s)**

For most simple (and even some complex) applications, it isn't necessary to have more than one arrangement of a table's fields that users can see and interact with—but there are some times where it's invaluable. In the first case, you'll find that you end up creating one table-owning form and one table-using form, and it feels like you've done twice the work than you have to (or did do with the DOS version).

But in the latter cases, where you start creating more than one view of a table's fields, the possibilities are endless and add all sorts of nifty things. And don't forget, DOS isn't Windows and DfW isn't DEDOS—you often have to rethink your approach to a problem if you want to create an application that “fits” into the Windows paradigm.

I had a client that was using DfW to generate quotations. We created two tables (tblQuoteHeader and tblQuoteDetails) and a couple of forms that used them, plus a series of procedures to enter data and generate reports. After the quote was created, the users needed to view SOME of the fields and only update a few (but weren't allowed to change the Quote #). With DEDOS, the only way to do this was via a procedure with a data-entry form. For DfW we created a special form with the subset of the fields from tblQuoteHeader and they could search for records and update them as necessary.

In the DEDOS world, a main form/subform construct is a “permanent” thing—both tables are permanently joined together on screen and cannot be viewed separately (unless you “F10” into the subform). In DfW, the tables exist independently and are only joined if and when you need or want them to be.

On the other hand, one of DfW's biggest failings (compared to the DOS product) is it's treatment of the F10 Multi/GoToForm and Ctrl-F10 Dynamic Lookup function keys. This is remedied, however, by the ability to create “lookup forms”. (For more information on this, see **Better Control (F10)**, Smith, Bolton, et al, *Dialogue Magazine*, Dec95/Jan96 issue. Contact Sapphire regarding back issue availability.)

As an aside, the splitting of storage and presentation is the “current rage” in database design. It's the basis of the “n-tier” model so favoured by Microsoft and various SQL vendors. It can't hurt to start thinking this way and moving away from the unitary “FormTable” approach of DEDOS.

### **Make the Forms “Match” the Source Documents**

Often, people are entering information into a database from some paper forms. Often, it's the same form over and over again. (Unless they're doing data-entry into an accounting system, especially Accounts Payable—it seems that every invoicing package puts invoice numbers in different places!)

If you're in this kind of situation, you can make the fields in the data-entry forms "match" the flow of the data on the paper form—you can even make the screen **look** like the paper form if you want, but arrange the fields differently in the underlying table.

Why should this matter? In DataEase for DOS, the "column number" (or field number of a field when you printed out the FormTable definition) of an indexed field had an impact on processing speed of some queries, especially when you were selecting on two or more fields. Generally speaking, no matter how the clauses in a select statement were arranged, indexes on fields with lower column numbers were reviewed first.

I suspect that DfW is no different in this respect, and that the "more selective" or unique items in a table should be added to table definition first. For example, in an "InvoiceHeader" table, InvoiceNumbers (which should be unique) should come before CustomerNumbers, since (hopefully) you'll have many invoices to the same Customer.

However, when it comes time to layout the actual data-entry form, you may need (for one reason or another) to change this order.

## Screen Styles and Colours

In DEDOS you created a screen style (set of colours) and assigned it to a user and this would control the colours of the fields and the form. In DfW, the style is assigned to the form—which means if you want Gail to see one set of colours and fonts, and Lawrence to see a different set, you need to create two identical forms with different sets styles or colours, then control which one the user gets via their menu tree.

(This one, admittedly, is *more* work with DfW than DEDOS).

## Hidden Fields

When creating any DataEase database (DEDOS or DfW), there are often fields that developers need to add to a table that users don't need to see or have access to—mainly artificial keys, but there are others. In DEDOS, you generally make them prevent-entry and "Highlight 3" in colour (black on black). Of course, the first time a user presses Alt-F5 and turns on search mode, the cursor pops into that field and they ask "Hey what's going on here?"

With DfW, you can put these fields into the table-defining forms and NOT display them at all in the user forms (table-using forms). They will still be updated, just not visible to users.

Hiding the implementation logic from the user keeps a nice clean interface while allowing you to do what you have to do.

## Sorted and/or Filtered Record Displays

DfW adds the ability to define a permanent sort on a form, or a permanent filter (or some combination of both). Thus, you can force records to be displayed in alphabetical order by client name as a user scrolls through the form by pressing the "Next Record" or "Previous Record" key. Or you can permanently restrict records to some subset. Either action can be performed in design mode by modifying the Query By Model (QBM) of the table-using form.

These sorts and filters are defined at the form level at design time and cannot be overridden by users (although they can be supplemented by a “temporary”, user-defined sort or filter using the appropriate menus.)

Form-level sorts are one of those things that Access users seem to crow about—so now you can do it with DfW.

## Read-Only Forms

Any form can be flagged by the designer as “read-only”.

If you do this with the table-defining form, than any user who manages to gain access to the forms directly still cannot add, modify or delete data, forcing them to interact with your database only through the forms that you’ve specified for this purpose.

If you do this with table-using forms, you can create a true “look but don’t touch” view of the data, so that no user, no matter what her security level (or the defined security level of the underlying table) can add, modify or delete the data. This just adds one more security level to your “security toolkit” and it is one that I have used in some applications.

## Projected Fields

Fields in a form can be “bound” to, or “attached” to the underlying table. They take on (or inherit) the characteristics of those fields. Thus, if you’ve defined a field as one into which you can enter data, you can enter data into it in a form—you can’t change that field to prevent entry on a table-using form.

However, you can create prevent-entry fields that are bound to the table-using form; they are part and parcel of the **form’s** definition and cannot be referred to by other tables or forms. Adrian Jones calls them “form-only” (or “form-only virtual”) fields; some of the early developers of DfW used the term “projected” for these fields (to differentiate them from “virtual” fields which exist as part and parcel of the table definition) since they “lie upon the surface” of the form, the way an image is projected on a screen.

I like the latter term.

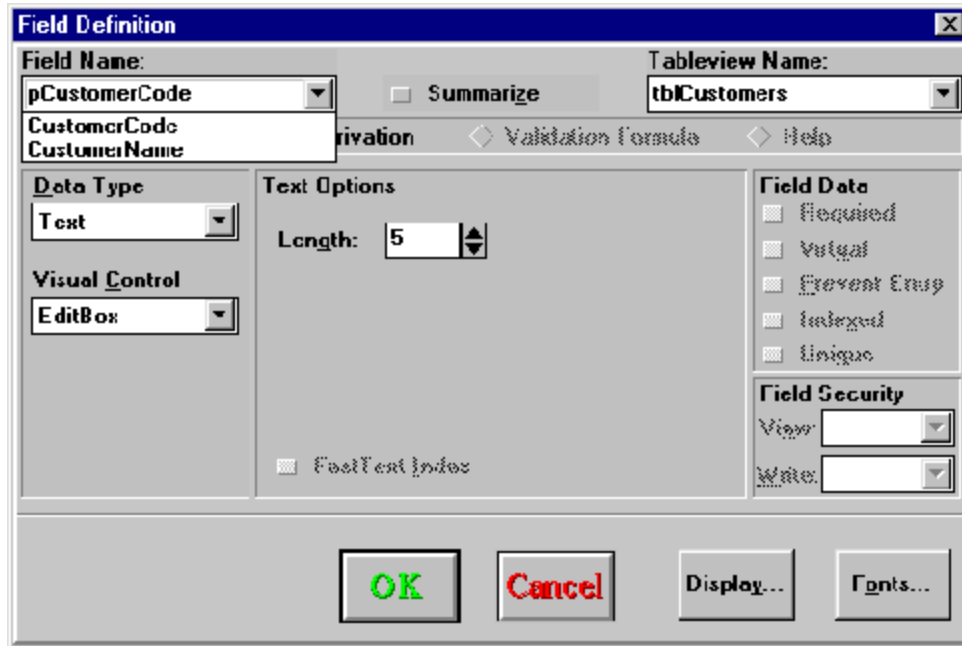
A projected field can do many of the UI things that a virtual field can do or is used for in DEDOS. The major difference is that it exists **ONLY** on a form.

A projected field’s field derivation can be a formula, a CDF call or even just one field. I’ve used them for on-screen prompts that change their text as values change in fields (or for text that appears or disappears as values change in fields). I’ve used them to load values into the array created by the CDFS2 library. (If you create a “hidden” projected field—e.g., one with no border and where the foreground and background colours are the same, Query/Search Mode (Alt-F5) will move the cursor into that field just as in DEDOS, so you haven’t totally eliminated this issue.)

One of my favourite “tricks” is if you make a projected field’s value equal to a field in the underlying table, you will be able to create a read-only (e.g., prevent-entry) version of a field that’s normally defined to allow entry!

Let's take a create one of these kind of projected fields using using the form **frmCustomers** we created earlier. Save the form with a different name—like **frmCustomers2**.

Open this form in design mode, then click on the **Field** button on the Object Palette (or press F10-Field, or select **Objects|Field** from the menu). Click anywhere on the form, and DfW will display the Field Definition dialogue box/form:



**Figure 9:** Creating a read-only projected field of the CustomerCode—first step

The fields in the underlying table are displayed in a choice list below the **Field Name** box, but don't select one of them; instead, type the word **pCustomerCode** (I use the "p" prefix to identify projected fields) into the edit box. (CustomerCode happens to be a five-character Text field, so I accepted the default values shown in the figure; but if you are creating a prevent-entry field, make sure that it matches the description (type and length) of the original field.)

Notice that all of the **Field Data** options (middle of the form, far right-hand side) are "greyed out" and cannot be changed.

Now click on the "Derivation" diamond-shaped widget, and the form changes to display the field derivation formula; the formula will be entered into the panel at the bottom of the form, just like any field that you create. You have two options here; you can enter a derivation formula of either one of:

- CustomerCode
- jointext("", CustomerCode)

as the field's derivation.

Both are acceptable, and both will get the job done (we'll discuss the difference in just a second). Click on **OK** to save the field. You could also change the colour (foreground and/or

background) and/or font to indicate to users that this is prevent-entry field. (I have a field style called “Prevent Entry” that I would apply to the field to do this, but that’s just me.)

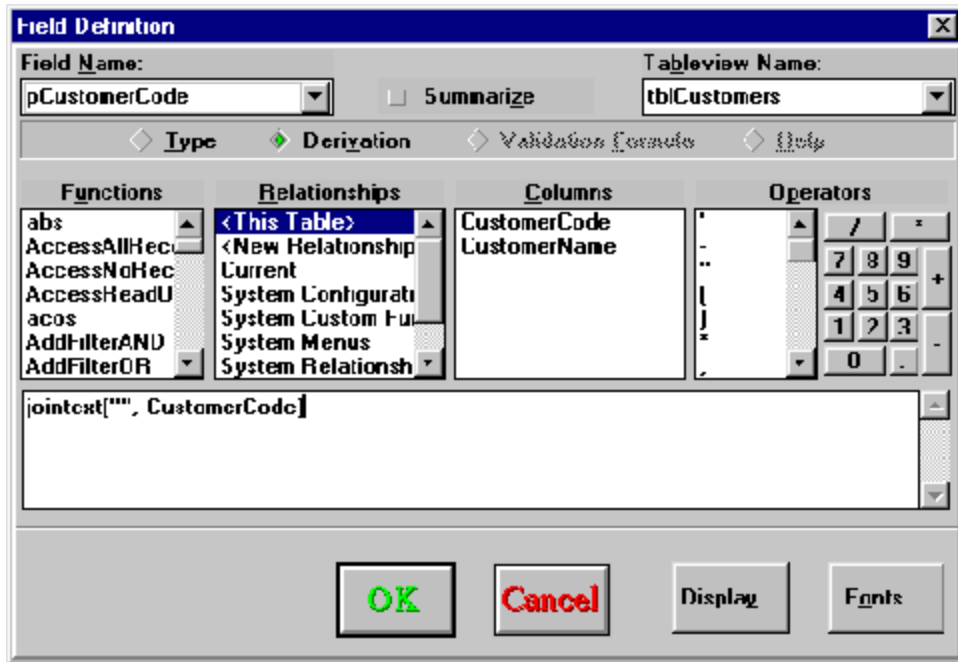


Figure 10: Creating a projected field—Step 2—the field derivation step

You’ve now created a prevent entry field on a form—which may not be prevent entry on other forms. If you’ve created any records, browse through the form, and you’ll see the same value in the **CustomerCode** field as well as the **pCustomerCode** field. Delete the CustomerCode field from this form. You now have one form in which you can enter a CustomerCode and CustomerName (e.g., frmCustomers) and second one in which you can only modify the CustomerName but not change the CustomerCode! Just imagine the possibilities!

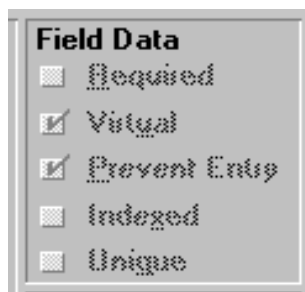


Figure 11:  
Close-up of Field Data  
once field is saved

If you open the form again in edit mode and look at the field’s derivation again, the Field Data attributes are still “greyed-out”, but notice that the **Virtual** and **Prevent Entry** attributes are now checked. Remember, projected fields are prevent-entry and virtual *by definition*. One the field is saved, DfW will automatically add these two attributes.

DfW does not currently offer conditional attributes (e.g., conditional colours, conditional prevent-entry, conditional required) as does DEDOS 5.xx.

### What’s the difference between the two versions of the field derivation?

If you simply type **CustomerCode** in the derivation formula (do not choose it from the pick list) and save the field, you can search on the field when in user mode (e.g., press **Alt-F5** and you

can put the cursor into **pCustomerCode**, enter some search criteria and get a result. However, if you later go into form design mode you'll find the derivation is blank (but still valid).

On the other hand, if you use the derivation shown in the figure (e.g., **jointext("", CustomerCode)**), the derivation will always be displayed if you subsequently view the projected field in an edit session. **But** any attempt to search on the field when in user mode will generate an error message!

## Report Formatting Logic

The report writer in DfW has its, ah, quirks. There are some times that I've found that the best (or easiest or only) way to work around them is to add some virtual fields to the tables, hide these fields from the user forms (since they don't need to see them) but use them in QBM reports or DQL procedure reports.

These virtual fields can be treated just like any other virtual field—they are available for any form or report that needs them. They're also no different than the hidden fields discussed above—just different in their usage. You hide them by just *not* displaying them on any table-using forms.

For example, DfW automatically treats all temporary numerical variables as floating point numbers, which cannot be formatted (e.g., I cannot force DfW to display it as "x,xxx.xx"). It also treats any number inside a logical "if" statement in the same way.

I have a client that issues tax receipts for donations for its fund-raising efforts. Canadian tax law requires that they report the amount donated by individuals vs donations received from other charities, e.g., the report would look like:

| Donor            | Individual | Charity  |
|------------------|------------|----------|
| Lawrence         | 1,500.00   |          |
| Ploni Foundation |            | 1,500.00 |
| ...              |            |          |
|                  | -----      | -----    |
|                  | [total]    | [total]  |

The Donation by DonorType Report

In DEDOS, you could write the query as:

```
for tblDonations ;  
  
  list records  
    DonorName in order ;  
    ReceiptNumber in order ;  
    if(DonorType = Individual, Amount, 0) : item sum ;  
    if(DonorType = Charity, Amount, 0) : item sum .  
  
end . -- main loop .
```

Donations by DonorType query (version 1)

and when you created the report layout, you could put the cursor in the “if(DonorType...)” fields, press F10 and format the results to give you comma separators at the thousands, and dollars and cents, etc. (e.g., “fixed” with 5 characters to the left of the decimal, and two characters to the right, along with commas in appropriate places—1,500.00, for example.)

Clicking on the field in DfW won’t let you change the format from float to “fixed”. If it does, the change won’t stick, or leads to an error while printing. If you didn’t need totals, you could wrap the “if(DonorType...)” line in the NumToText() CDF which would format the calculation. If you did need totals, you have to preface the procedure with several “sum of” statements, and then format the results with NumToText().

The easiest way to do this was to add two virtual fields to tblDonations (**vIndividualDonation** with the first formula and **vFoundationDonation** with the second one), formatted with five characters to the left of the decimal and two to the right, and then write the query as:

```
for tblDonations ;  
  
  list records  
    DonorName in order ;  
    ReceiptNumber in order ;  
    vIndividualDonation : item sum ;  
    vFoundationDonation : item sum .  
  
end . -- main loop .
```

Donations by DonorType query (version 2)

The downside to this approach is that your tables may get filled with all kinds of these fields, since logical if’s aren’t the *only* situation where these fields are needed.

I need to stress that was the **easiest** and **quickest** way to do get my report done and in the hands of my client. It’s not the **only** approach; I’m sure that there’s at least two or three more ways to handle it—after all, in DataEase, if you can’t come up with a half-dozen ways to accomplish a task, you’re not trying hard enough!

## Self-Documentation

I like to add notes to my table-defining forms that explain the purpose of the table (if it's an answer table or some kind of summary table that users won't see) or the purpose of a field (if it's not immediately obvious from the name). With several clients and multiple applications—parts of which I might not look at for months at time—referring to these notes is often the **only** way I can continue to keep my ComputerWizardly status in the eyes of my clients. Since the client or her staff don't see the table-owning forms, they don't see the notes either, which keeps the user interface nice and clean.

## Are there any downsides to this “Tables vs Forms” approach?

There are two that I can think of:

- 1) It *does* increase some of your development time with DfW (compared to DEDOS), since you are, in some cases, “doing stuff twice” (creating the table-owning form then creating the table-using one). But it's a minor increase in time, since you're not worrying about making the table-owning form “pretty” or ensuring that it's layout is logical or that it matches the (paper) source documents that users may be working from.
- 2) F10 Multi/GoTo and Ctrl-F10 Dynamic Lookup work in DfW only between table-defining forms. You need to do some “programming around” this issue.

---

*Lawrence Fox is the principal of ComputerWizard Consulting; you can reach him at:*

[computerwizard@compuserve.com](mailto:computerwizard@compuserve.com)

*He welcomes your comments on this paper.*

*The opinions expressed in this paper are his and do not necessarily reflect the views or opinions of the employees of Sapphire International.*