

# Practical Tips for Working with Existing Code

*Session Number*

*Tamar E. Granor, Ph.D.  
Tomorrow's Solutions, LLC  
8201 Cedar Road  
Elkins Park, PA 19027  
Voice: 215-635-1958  
Fax: 215-635-2234  
Email: [tamar@tomorrowssolutionsllc.com](mailto:tamar@tomorrowssolutionsllc.com)*

*Sooner or later, almost every developer has to take over an existing application. This session looks at tools and techniques for understanding how such applications work, improving the data model, dealing with non-developers who wrote the original code, and more.*

Over the last few years, much of my work has been with existing applications. No matter what my role is in these projects, there are some principles that apply and some tools that make the job

easier. In this paper, I'll look at issues from dealing with the client to understanding how the application currently works to handling non-normalized data and much more.

## **Getting started**

When an existing application lands on your desk, there are a number of things to do before you start changing existing code or writing new code. In some situations, you need to address legal issues. In others, you may need to find the source code. You also want to examine any documentation that exists and take a first look at the code to get a feeling for what faces you.

### ***Legal issues***

Let me preface this section by stating that I'm not a lawyer, nor am I offering legal advice here. This discussion is meant to alert you to possible legal issues in working with existing code, so that you can get appropriate legal advice.

The big concern with any existing code is the matter of ownership. If you work for the company that wrote the code, this is not an issue. But if you are an independent consultant or an employee of a software development company, it's a concern on every project involving an existing application. In-house developers may face legal concerns regarding software purchased by their companies.

Before you modify any code, you need to know that you have the legal right to do so. For that to be the case, the person asking you to make the changes must have appropriate intellectual property rights. Ask for documentation of those rights, either proof that your employer/client owns the copyright to the code or a release from the copyright owner, giving your employer/client the right to modify the code. Get this in writing, so that you don't open yourself up to accusations of copyright infringement. If you have any concerns on this front, speak to a lawyer who focuses on Intellectual Property rights.

### ***Get the code***

Once you're sure you may modify the code, the next step is ensuring that you actually have the code to be modified. Sometimes the problem is having no source code at all, while other times, it's a question of figuring out what actually constitutes the source.

If all you have is executable code, you need a decompiler to create source code. For VFP, the decompiler of choice is Refox (<http://www.refox.net/>). Refox can take an EXE and create the source code files, including forms, class libraries, and so forth. Of course, the files so generated do not have comments, so getting source code from the client is a better choice.

When the client does provide source code, often there's a lot of extraneous material, such as test code, code that was replaced, code related to distribution of the application, and so forth. It's not unusual for the client to give you several sets of source with differences for different users.

Cleaning up this mess and figuring out what's actually used can go a long way toward helping you understand the application. A first step in this direction is to create a new project and add the main program to it. Then, build the project to pull in code that's used. While this isn't infallible, it

gives you a start at figuring out which code is in use. In [The Project](#) later in this paper, I'll look at some utility code to take this process further.

### ***Get documentation***

Ask the client for any documentation that exists for the application. This includes original design documents, developer documents, coding standards and user documentation. Most often, there will be little or nothing.

Even when documentation exists, it tends to be out of date. On one project, I was given database documentation. This was a reasonably well-designed document, listing each table, with the reason for its existence, and listing each field with its explanation. Unfortunately, it had been created soon after the application was first written (nearly 10 years before I took it over), and the only updates were a few scribbles about additional values for some fields.

### ***Run the application***

Once you have the code and documentation in hand, your next step is to run the existing application so you can get a feel for whether it works and how it works. You can do a first quick evaluation of the user interface, and see whether you're dealing with a true Windows application, a FoxPro 2.x Windows application or a DOS application (even if runs in Windows).

This first run isn't meant to be exhaustive. You're not trying to explore every corner of the application (unless you've been hired to evaluate it in that way). Find out from the client what the key pieces of functionality are and try those. See whether they work and if not, what kinds of problems you run into.

### ***Look under the hood***

The next step is to open up the code and the data and take a look. Examine the structure and content of the data. Check whether tables are normalized and have primary keys defined. Look for appropriate indexes (especially if one of the reasons you've been brought in is poor performance).

Look at the main program to get a feel for the architecture of the application. Look for signs that it uses a framework, whether homegrown or commercial.

Open a few forms and examine them. Ask questions like:

- Are forms based on a form class (or a hierarchy of form classes) or is all the code in every form?
- Do all the forms use the same approach to data handling? Is there any indication of n-tier design?
- What classes are the controls based on? In particular, are there classes in use or are all controls based on the VFP base classes.
- Do the forms look (visually) like they belong to the same application?

- Are controls on the form organized in some logical manner?

If the application includes class libraries, use the Class Browser to get a feel for the hierarchies involved.

Examine the code for coding standards. Keep in mind that you're not checking whether the original authors used your personal standards, just whether they used any standards.

Are there comments that actually contribute to your understanding of the code?

Is there a naming convention in use?

Is code appropriately indented?

### ***Consider a more formal audit***

In some situations, you may want to perform a more formal audit of the existing system that results in a report to the client. (I charge at my usual hourly rate for producing such an audit report.) An audit asks all the questions above, but also collects data about the system.

Ted Roche has a paper on his website (<http://www.tedroche.com/Present/2000/E-MAINTDoc.htm>) that contains a template for an audit report. There's also a tool (linked at <http://www.tedroche.com/papers.php>--look for the "Software Maintenance" Source) that scans a project and produces statistics about the number of files, lines of code and comments. These statistics can be very helpful both for you to get the big picture and to show the client the magnitude of his system.

## **Dealing with the client**

From here on out, I'll refer to the person asking for changes as "the client" even though it may be your employer or another division of your company.

After you've run the application and examined the code, you're in a position to talk to the client about what he wants and to determine (if you're dealing with an outside client, not an employer) whether you want to take this project on. Don't be afraid to run away from a project that you think has no chance of success. Also, don't be afraid to tell a client that an application isn't worth changing.

Exactly what you say to the client depends on a number of factors, but a big one is who wrote the original code. If the client or one of his relatives wrote the code, you need to be far more tactful than if it was written by a former employee. Even in the latter case, you're better off choosing your words carefully. Telling the client "whoever wrote this code was an idiot" raises questions about the intelligence of the person who paid for the code. Saying something like "the code doesn't follow best practices" is likely to meet a better reaction. Often, of course, the problem is that the code was designed and written for an earlier version of FoxPro, so that what was appropriate no longer is. Clients can generally understand that the state of the art has changed; after all, they've seen it with other applications.

It's very important at this stage to set expectations appropriately. If the code is a mess and the application doesn't work very well, the best advice you can give the client is to replace it rather

than spend any more money on it. In my experience, many small companies and organizations are running homegrown applications that could be easily replaced with off-the-shelf software. In such cases, your job may turn out to be evaluating existing packages and making a recommendation.

That's exactly what happened when I was called in to a non-profit organization a few years back. They had a homegrown FoxPro application for managing their member/donor database. The original developer had maintained the code for them for more than a decade, but was moving on to other ventures. They hoped I would take over the tasks he'd been doing for them, such as changing the underlying report every time they wanted to send a form letter out. It was immediately apparent to me that keeping this application running wouldn't serve their needs anywhere near as well as investing in software specifically designed for non-profit organizations like theirs.

In some cases, things work well enough and the code is good enough that you can move forward with it, but the client needs to understand the consequences of doing so. That may be dated-looking forms, a less than intuitive user interface, or limited flexibility. Make sure the client knows what these issues are before you start working on the code.

## Tread softly and leave breadcrumbs

One of the first things medical students are taught is the Latin maxim "Primum non nocere," which translates to "First, do no harm." The same idea applies to modifying working applications. As you work with existing code, your first goal should be to avoid breaking what works. It's very tempting when presented with bad code to start making it better, but until you get the lay of the land, making anything but the most minor of changes is a bad idea. If the code works, even if it's dreadful, don't change it without putting a plan in place to ensure that it still works when you're done.

When you do have to make changes, initially keep them small. Do only the minimum to fix the problem until you really understand the consequences of your changes.

Whenever you change code in an existing system (meaning any application that has been deployed), leave a change comment that includes the date and your name or initials, so others can see what you've done. In VFP, it's easy to add such comments and make them uniform. You can use either an IntelliSense script or a text scrap in the Toolbox. I use an IntelliSense script, triggered by the string TEGMOD, that adds a comment in this format:

```
* Modified 20-February-2007 by TEG  
*
```

The script leaves the cursor on the second line ready for me to add an explanation of what I've changed. My code for this script is adapted from a script created by Doug Hennig and looks like this:

```
lparameters toFoxCode  
local lcReturn  
  
if toFoxCode.Location <> 0  
    toFoxCode.ValueType = 'V'  
    lcReturn = GetText()
```

```

endif toFoxCode.Location <> 0
return lcReturn

function GetText
local lcText, nDay, cMonthName, nYear, dToday
dToday = date()
nDay = day(dToday)
cMonthName = cmonth(dToday)
nYear = year(dToday)
text to lcText textmerge noshow
* Modified <<nDay>>-<<cMonthName>>-<<nYear>> by TEG
* ~
endtext
return lcText

```

You can find some additional scripts along these lines at

[http://fox.wikis.com/wc.dll?Wiki~IntelliSenseCustomScripts~VFP#Contents\\_20S0VIYW2](http://fox.wikis.com/wc.dll?Wiki~IntelliSenseCustomScripts~VFP#Contents_20S0VIYW2).

## Digging into code

When you're ready to start actually making changes, you need to find your way around the application in order to figure out who does what, where it's done and how. There are several tools available to help you do this kind of exploration.

### *The Documenting Wizard*

The first place to look for help is VFP's Documenting Wizard. It analyzes a project and creates a number of reports; it also formats code, handy if the original developer had no standards for this.

For understanding a project, the most useful items created by the Documenting Wizard are the cross-reference listing and the tree listing. The cross-reference listing shows you every variable, function, procedure, property, method, etc., in the project, and shows where each is referenced in the project. The Documenting Wizard puts the same information in a table (FDXREF.DBF), so you can manipulate it with code as well.

The tree listing shows the hierarchy of calls in the project. (Be sure to choose FoxFont to view this file, so the lines it creates look like lines.) It also provides a hierarchy of classes.

Unfortunately, the Documenting Wizard is slow, somewhat clunky to use, and somewhat inflexible. As a result, I use it much less than other tools.

### *The Debugger*

VFP's Debugger is a good way to get a feeling for how a project works. Figure out which is the main program and run it in the Debugger, then step through to see what happens. Obviously, you won't want to step through the entire application, but walking the code this way is a good way to get a grasp on the basic architecture of the application. Use the Locals and Watch windows to examine the variables and objects of the application.

While you're stepping through code, take advantage of the VFP IDE. Use the Command window to query the application and the Data Session window to see what tables are open when and what data they contain.

The Coverage Logger, which is available through the Debugger creates a log file showing every line of code executed while a program is being used. The log can be useful for figuring out the order in which things are happening. Together with the Coverage Profiler, it can also help you find sections of code that aren't being executed at all, or sections that are particularly slow.

When you start actually working with the code, the DEBUGOUT command can help you understand what's really going on. DEBUGOUT accepts a list of expressions and displays their current values in the Debugger's Debug Output window. Use it to trace code execution and the values of things as the code runs.

## ***The Project***

VFP's projects can be explored in two ways. First, a PJX file is simply a DBF with a special extension, so you can open it as a table and examine its contents. A somewhat easier approach is to use the Project object that's automatically created when you open a project in the IDE. The Project object has a Files collection with one item for each file in the project. You can use this collection to explore the project.

As noted in [Get the Code](#) earlier in this document, a good way to start working with an old application is to create a new project, add the main program and choose Rebuild project. That pulls in most of the files used by the project. However, it misses any that are used indirectly (through macros or data-driven approaches).

Once you have the new project, there are several directions to go. First, attempting to build and run an EXE is a good way to start figuring out what's missing, as well as to see whether the code actually compiles. Make sure to choose Rebuild All in the Build Options dialog.

In my experience, the files that clients send me for existing applications always include extraneous material such as little utility programs, programs that are no longer being used, temporary tables, and the like. The ability to work with the project as an object makes it easier to clean up this kind of mess. After building a new project, you can run a little code that builds a list of files in the project directories that aren't used in the project. Listing 1 shows a program that accepts a project name (with path), and a list of folders (comma-separated or semicolon-separated), and fills a cursor with a list of all the files in those folders that aren't included in the project.

**Listing 1. Getting a list of files in the project folders that aren't used in the project is a first step toward cleaning them up.**

```
* CheckForUnusedCode.PRG
LPARAMETERS cProject, cPath
* Look for unused code

LOCAL oProject, nCounter, oFile
LOCAL nDirs, nDir, aDirs[1]

MODIFY PROJECT (m.cProject) nowait

oProject = _VFP.ActiveProject
* First, make a list of all files in project
LOCAL aProjFiles[oProject.Files.Count]

nCounter = 0
```

```

FOR EACH oFile IN oProject.Files
  nCounter = m.nCounter + 1
  aProjFiles[m.nCounter] = UPPER(oFile.Name)
ENDFOR

CREATE CURSOR Unused (mFileName M)

* Now traverse directories
* First, make a list of directories
nDirs = ALINES(aDirs, m.cPath, 1, ";", ",")
CREATE CURSOR DirsToCheck (mDirName M)

FOR nDir = 1 TO m.nDirs
  INSERT INTO DirsToCheck VALUES (aDirs[m.nDir])
ENDFOR

LOCAL aFiles[1], cOldDir, cFile, nFilesToCheck, cExt

cOldDir = SET("Default") + CURDIR()

SCAN
  IF DIRECTORY(mDirName)
    CD ALLTRIM(mDirName)
    nFilesToCheck = ADIR(aFiles, "*. *")
    FOR nFile = 1 TO m.nFilesToCheck
      cFile = aFiles[m.nFile, 1]
      cExt = JUSTEXT(m.cFile)
      IF INLIST(cExt, "PRG", "SCX", "MNX", "FRX", "VCX", "QPR")
        IF ASCAN(aProjFiles, FORCEPATH(m.cFile, ALLTRIM(mDirName)), ;
          -1, -1, 1, 7) = 0
          INSERT INTO Unused VALUES ( ;
            FORCEPATH(m.cFile, DirsToCheck.mDirName))
        ENDIF
      ENDIF
    ENDFOR
  ENDIF
ENDSCAN

CD (m.cOldDir)

RETURN

```

An alternative way to handle the problem of extra files in the project's directories is to create a new directory tree and copy over only the files that are actually used. That has the added benefit of preserving a copy of the project as it existed before you started. (Of course, backing up the original project is an important step, no matter how you do it.)

The easiest way to do this is to create a new project in the existing project directory, add the main program and use Rebuild project to get the list of files, and then move the new project (PJX and PJT) to the new folder. Then, code like Listing 2 (CopyProject.PRG in the session materials) can copy all the files used in the project into the new directory tree. It even creates the directories as needed.

**Listing 2. This program copies all the files in a project from one directory tree to another.**

```

* CopyProject.PRG
LPARAMETERS cProject, cOriginalPath, cNewPath

LOCAL oProject, oFile, cNewName, cNewFilePath

```



```

MODIFY PROJECT (cProject) nowait
oProject = _vfp.ActiveProject

* Copy all files to appropriate directories
FOR EACH oFile IN oProject.Files
  cNewName = cNewPath + STREXTRACT(oFile.Name, cOriginalPath,"",1,3)
  cNewFilePath = JUSTPATH(cNewName)
  IF NOT FILE(cNewName)
    IF NOT DIRECTORY(cNewFilePath)
      MD (cNewFilePath)
    ENDIF
    COPY FILE (oFile.Name) TO (cNewName)
  ENDIF
  IF JUSTEXT(cNewName) = "scx" AND NOT FILE(FORCEEXT(cNewName, "SCT"))
    COPY FILE (FORCEEXT(oFile.Name, "SCT")) TO (FORCEEXT(cNewName, "SCT"))
  ENDIF
  IF JUSTEXT(cNewName) = "vcx" AND NOT FILE(FORCEEXT(cNewName, "VCT"))
    COPY FILE (FORCEEXT(oFile.Name, "VCT")) TO (FORCEEXT(cNewName, "VCT"))
  ENDIF
  IF JUSTEXT(cNewName) = "mnx" AND NOT FILE(FORCEEXT(cNewName, "MNT"))
    COPY FILE (FORCEEXT(oFile.Name, "MNT")) TO (FORCEEXT(cNewName, "MNT"))
  ENDIF
  IF JUSTEXT(cNewName) = "frx" AND NOT FILE(FORCEEXT(cNewName, "FRT"))
    COPY FILE (FORCEEXT(oFile.Name, "FRT")) TO (FORCEEXT(cNewName, "FRT"))
  ENDIF
  IF JUSTEXT(cNewName) = "lbx" AND NOT FILE(FORCEEXT(cNewName, "LBT"))
    COPY FILE (FORCEEXT(oFile.Name, "LBT")) TO (FORCEEXT(cNewName, "LBT"))
  ENDIF
ENDIF
ENDFOR

```

Another thing you might want to do is compare two versions of a project and get a list of files included in the original that aren't in the new version, so you can check to see whether these are used indirectly. Listing 3 shows a program (ListMissingFiles.PRG in the session materials) that performs this comparison.

**Listing 3. This program compares two projects and fills a cursor with a list of all the files in the first project, but not the second.**

```

* ListMissingFiles.PRG
LPARAMETERS cOldProject, cNewProject

LOCAL oOld as VisualFoxpro.IFoxProject, oNew as VisualFoxpro.IFoxProject

MODIFY PROJECT (m.cOldProject) NOWAIT
oOld = _VFP.ActiveProject

MODIFY PROJECT (m.cNewProject) NOWAIT
oNew = _VFP.ActiveProject

CREATE CURSOR Missing (mFile M)

LOCAL oFile, oNewFile, cFileName

FOR EACH oFile IN oOld.Files
  * Look for each file from the old project in the new project.
  * The filename without path is the key in the collection.
  cFileName = JUSTFNAME(oFile.Name)

  TRY

```

```

oNewFile = oNew.Files[m.cFileName]

CATCH
* Used in old, not in new
INSERT INTO Missing VALUES (oFile.Name)

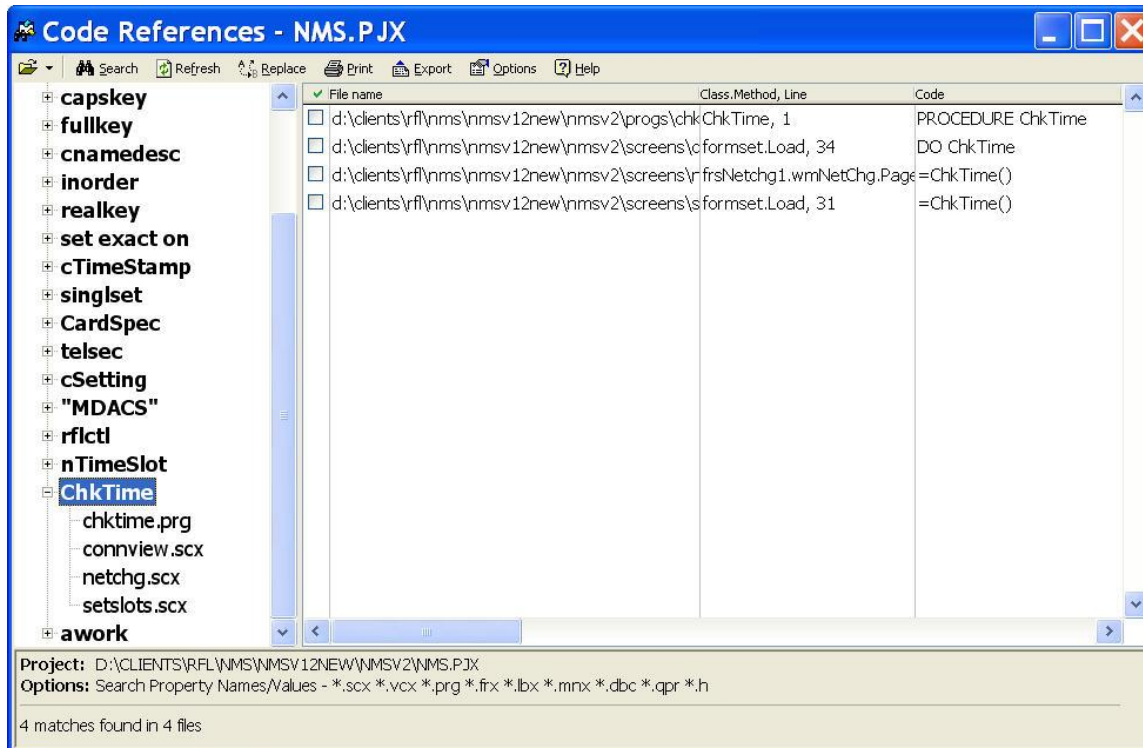
ENDTRY
ENDFOR

```

## Code References

The Code References tool (added in VFP 8) is your best friend when you're working with unfamiliar code. It lets you search an entire project for occurrences of a string. It also lets you search through a directory hierarchy.

To search in a project, open the project before opening Code References. Once you've done a search, the Code References window shows the results, with the list of files on the left and the lines of code on the right, as in Figure 1. When you click on a file name in the left pane, only the matches within that file are shown in the right pane.



**Figure 1. The Code References tool lets you find all occurrences of a string in a project.**

Double-click on any match in the right pane to open the file with the matched string highlighted. Code References can also perform search-and-replace within a project, though there are some limits on that capability.

You can begin a search by choosing Lookup Reference from the shortcut menu in any code editing window, so you don't have to explicitly open the tool.

I use Code References in several ways. First, it provides a way to "trace" backwards. That is, if I'm trying to understand how a particular function or method is used, I can search for it in the project, as I did for the procedure ChkTime in Figure 1. Then I can look at the calling code to get some context and, if necessary, search for calls to the calling code, and continue working my way upwards through a calling chain. ("Calling chain" is a little misleading since this is a development-time activity, but it mimics working backward through a calling chain at runtime.)

I've also used Code References to check whether a table or field is used in a project, so that I could remove it. In one recent project, I did some major restructuring of the data tables, adding primary and foreign keys, and removing repeated data. I then needed to adapt all the code that used the fields I was removing. Code References let me find all those places easily.

Code References has another face that's handy for walking upward through code. In any code editing window, the shortcut menu includes View Definition. When you choose that item, VFP searches for the place where whatever is currently highlighted is defined. You can use this facility for variables, constants, methods, functions, properties of code classes (classes in PRGs), and classes. If there's a project open, that project is searched. Otherwise, the search casts a pretty wide net and you may find matches you didn't expect. If a single match is found, it's highlighted, opening the file containing the definition, if necessary. If multiple matches are found, a window (Figure 2) appears to let you decide which one to open.

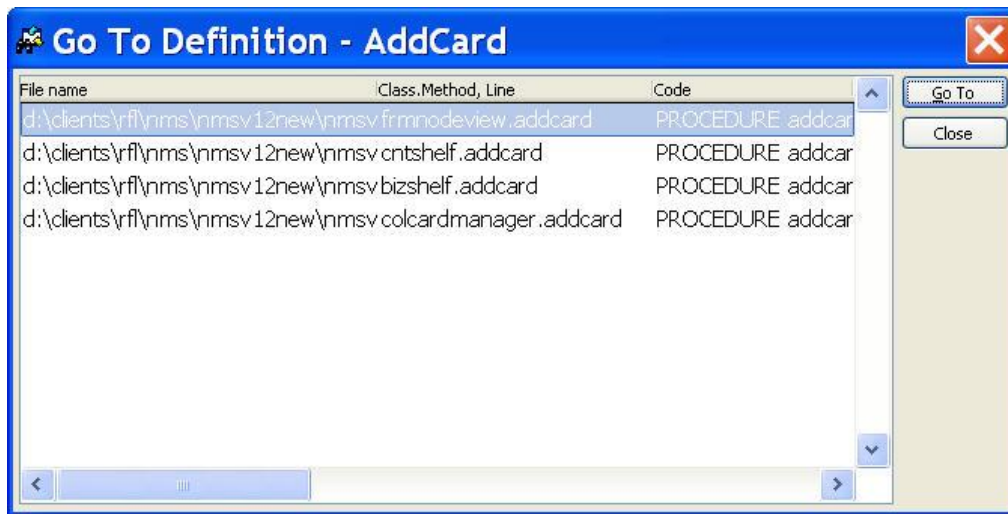


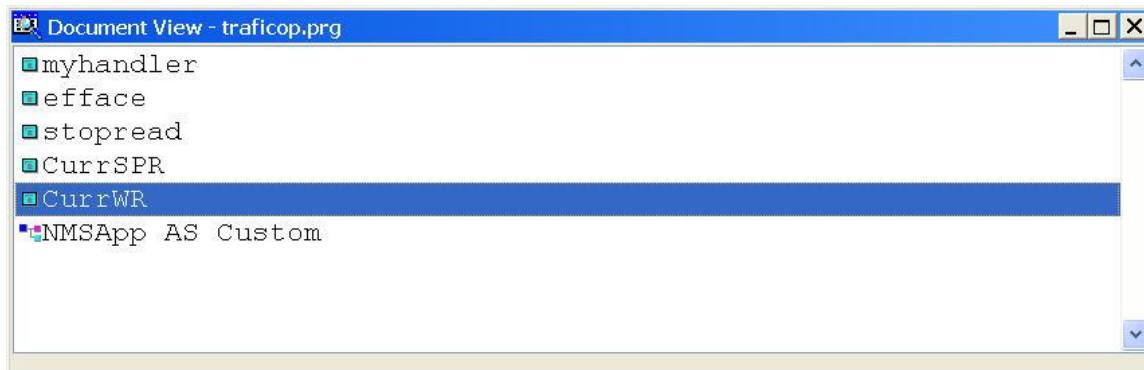
Figure 2. When you choose View Definition from the shortcut menu in any code editing window, and multiple definitions are found, this window lets you decide which to open.

## Document View

Older applications often use procedure files and program files that contain functions called by the principal program in the file. The Document View tool (added in VFP 7) offers an easy way to navigate through these code files. (The tool is also valuable for dealing with classes written in code rather than in the Class Designer.)

Document View lists functions, class definitions, methods, and constant definitions in the active "code source." A "code source" can be a PRG file, a class or a form. The tool's shortcut menu lets you control display of constant definitions and preprocessor directives (like #INCLUDE). Figure

3 shows Document View for the main program of a project I updated, after I'd added an application class, but before I'd added any methods to that class.



**Figure 3.** The Document View tool shows you what definitions are included in the current code source. You can jump right to any item listed by clicking on it.

## Dealing with Data

The data design for many of the applications that cross my desk leaves a lot to be desired. Frequently, data is not normalized and often, the design actually seems haphazard. I think there are two factors at work here.

First, many older applications were originally designed by non-developers, who didn't know the rules of normalization. Often, these people approached database design as if they were working with a spreadsheet, resulting in data being repeated across multiple tables, and in repeating fields (several columns for a single kind of data instead of using a one-to-many or many-to-many relationship) within a single table.

Second, often, the needs for the application changed over its lifetime. When that happened, more often than not, rather than redesigning the database and the code to neatly accommodate the changes, new abilities were tacked on wherever they could be without a major impact on existing code.

Doug Hennig tells what may be the ultimate story about adding new data and abilities without modifying the database:

One particular, very popular, commercial application I've worked with has very unusual data structures because of the unwillingness of the developers to add additional fields to the table structures as the business requirements changed. For example, when email addresses became an important value to store about 15 years ago, rather than adding an email address field to the company table, the developers instead decided to add a new record type to a child table and reuse existing fields for different values. The RecType contains a "P," the Contact field contains "E-mail Address," and the email address goes into the City field. However, as email addresses got longer and longer, the 25-character city field didn't cut it, so now the first 25 characters of the email address goes in the City field and the rest in the Notes field. Of course, someone could have more than one

email address, so to distinguish the primary email address record from secondary ones, the second character of the Zipcode field contains a "1."

Web sites are the same except the Contact field contains "Web Site" instead.

Now try to imagine what a SQL statement looks like to display the primary email address and primary Web site for a given company. It requires multiple joins to the child table, concatenation of the City and Notes fields, and some unusual looking WHERE clauses.

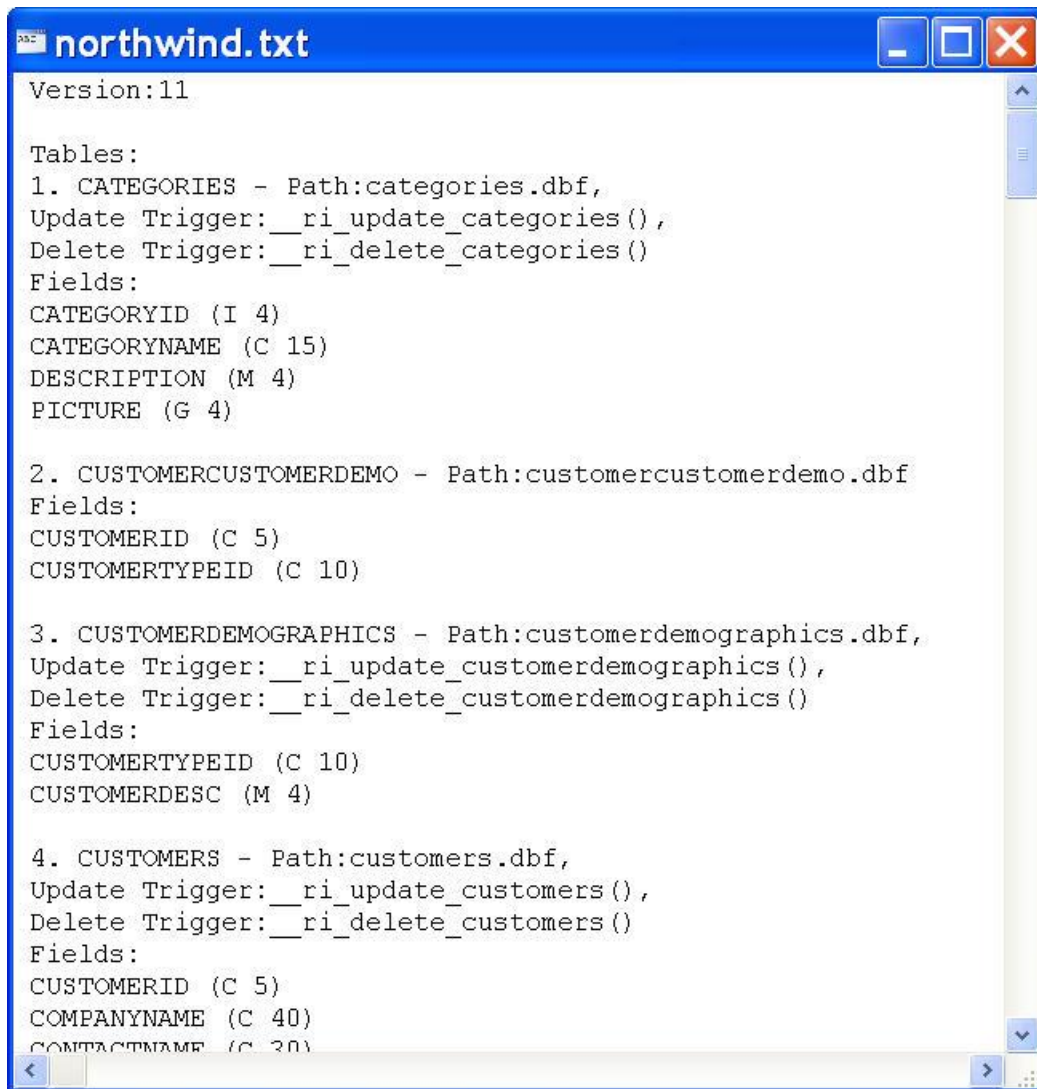
As with code, the first thing to do when dealing with bad data design is the bare minimum. While it's tempting to come in and clean it up, realize that changes to the database almost always mean major changes to code. Any changes you make need to be carefully planned.

### ***Document the database***

The first step is to see what you actually have by creating documentation. Even if you've been given documentation, chances are it's not up-to-date. (See my story in [Get Documentation](#) earlier in this document.) There are several ways to create documentation for a VFP database.

One way to document a database is by running GENDBC against it. This tool, which comes with all versions of VFP, creates a program that can recreate the structure of the database. While the output is very useful, though, it's probably not the best way to get documentation to help you understand the data. In addition, it works only for databases, not free tables; in my experience, free tables are more common for older applications.

Sedna introduces a new database documenting tool as part of the Data Explorer, which itself was new in VFP 9. The context menu for any VFP database now contains Document Database. When you choose it, a text file is created and displayed, showing the structure of the database. Figure 4 shows part of the output for the example Northwind database.



```
Version:11

Tables:
1. CATEGORIES - Path:categories.dbf,
Update Trigger:__ri_update_categories(),
Delete Trigger:__ri_delete_categories()
Fields:
CATEGORYID (I 4)
CATEGORYNAME (C 15)
DESCRIPTION (M 4)
PICTURE (G 4)

2. CUSTOMERCUSTOMERDEMO - Path:customercustomerdemo.dbf
Fields:
CUSTOMERID (C 5)
CUSTOMERTYPEID (C 10)

3. CUSTOMERDEMOGRAPHICS - Path:customerdemographics.dbf,
Update Trigger:__ri_update_customerdemographics(),
Delete Trigger:__ri_delete_customerdemographics()
Fields:
CUSTOMERTYPEID (C 10)
CUSTOMERDESC (M 4)

4. CUSTOMERS - Path:customers.dbf,
Update Trigger:__ri_update_customers(),
Delete Trigger:__ri_delete_customers()
Fields:
CUSTOMERID (C 5)
COMPANYNAME (C 40)
CONTRACTNAME (C 20)
```

**Figure 4.** The new Document Database option in the Data Explorer creates a text file listing all tables, fields, views and relations in the specified database.

You'll find the code that generates the documentation in the Options dialog of the Data Explorer (choose Manage Menus, click on the Document Database item and choose the Script to Run tab), so if you don't like the output it generates or want to add additional information (such as index tags), you can do so without starting from scratch.

However you create documentation of what's actually included in the application's data, the harder part is understanding what each table and field represents. If a previous developer is available to help you with this, take advantage of it. Otherwise, you have to use a combination of looking through the application code and asking the client; both strategies tend to leave holes in your knowledge. The Code References tool can be helpful here.

### ***Improving data structures***

One of the biggest issues I see with older databases is a failure to provide single-field primary keys for each table. Often, these tables have several fields drawn from a parent table to provide a

primary key. In addition, it's not unusual for older applications to repeat data rather than use separate look-up tables.

At some point in the maintenance process, it can make sense to fix these problems. But doing so can be time-consuming so it's not an automatic decision.

There are three data design problems (all related to normalization) you may want to fix:

1. No primary key or primary key has meaning or uses multiple fields.
2. Data is duplicated in multiple tables. This is usually a consequence of problem #1.
3. Repeating fields are used, that is, a single table has multiple fields for the same kind of data.

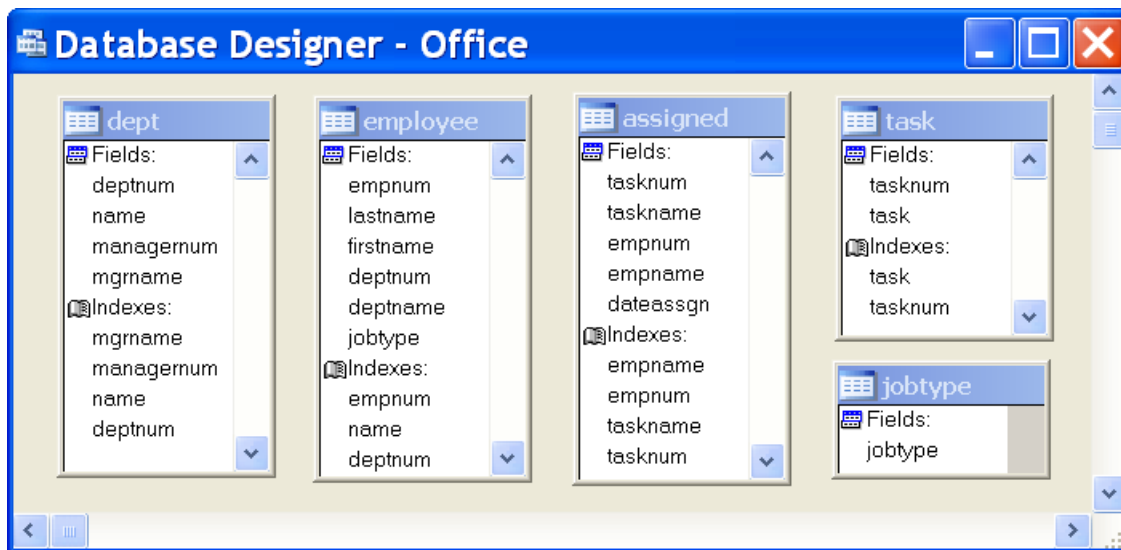
Fixing any of these problems has two consequences. First, you have to fix the code that depends on the current data structure so that it works with the new data structure. The Code References tool can be very useful for tracking down references to fields and tables you're changing.

Second, you have to provide a way for current users to update their existing data to the new structure without losing anything. If you use Stonefield Database Toolkit to manage databases, you may be accustomed to having that tool handle database updates for you. However, SDT can't handle these changes because you need to actually modify data, not just data structures. You can do the conversion as part of the installation process (in a post-Setup executable) for the new version, or the first time the new version runs.

## **Why change the database?**

Figure 5 shows a database (included in the session materials as Office.DB. Full disclosure: I created this database, using VFP's Database Wizard, to demonstrate these problems. This is not a database I encountered in the wild) that has the first two data problems. Rather than using surrogate primary keys, meaningful data is used to link tables together. For example, the EmpNum field of Employee, representing the Employee number, is the acting primary key. Though you can't tell it from the design, there's a nastier version of this problem in the Task table. The TaskNum is created by concatenating the department number with two digits. So if a task is reassigned to another department, either its number should change or it can no longer be properly interpreted. Of course, any change to TaskNum requires changes to all the records in Assigned for that task.

In addition, even with data available to link records in different tables, some additional fields are repeated. For example, in Dept, the department manager is indicated by both ManagerNum (which is drawn from Employee.EmpNum) and MgrName (drawn from Employee.LastName and Employee.FirstName). Assigned contains both the task number and its description.



**Figure 5.** This database needs surrogate primary keys and has data duplicated across tables.

This database also shows a more subtle form of the second problem. The JobType table is available to populate a combobox of job descriptions, but rather than having a primary key that then links Employee to JobType, the actual job description is included in Employee.

You may wonder why storing the manager's name or copying the job description is a problem. Inevitably, this data will change. The manager will get married or divorced. The description used for the job will be reworded. With the data scattered through several tables, changes like this have to be propagated from the original table to the others. Using the record's primary key to point back to it instead means that changes are needed only when the actual connection changes, for example, when the manager leaves and someone else is appointed manager of that department or when an employee changes jobs within the company. This is why the rules of database normalization call for storing each item of data once and only once.

## Adding surrogate primary keys

The best practice for primary keys is to use a field that has no other meaning, called a *surrogate key*. While surrogate keys can be character or numeric, VFP makes using integer keys quite easy. Starting with VFP 8, you can use the Integer (AutoIncrement) type for a primary key and VFP will automatically populate the field for each new record.

Because auto-incrementing integer fields are read-only in VFP, adding them to an existing table is a three-step process. You first add the new field as an integer, then populate it, and finally change the new field to AutoIncrement, setting the next available value appropriately. Listing 4 shows a function (AddPK.PRG in the session materials) that handles the task. The function also creates an index tag for the field, setting it as the primary key if the table is in a database and as a candidate key for free tables.

**Listing 4.** This function adds a primary key to an existing table, ensuring that every record gets a unique key.

```
*PROCEDURE AddPK
LPARAMETERS cTable, cField

IF FILE(FORCEEXT(cTable, "DBF"))
```



```

SELECT 0
USE (cTable) EXCLUSIVE ALIAS __AddPK
IF TYPE(cField) <> "N"
  * Add it
  ALTER TABLE (cTable) ADD (cField) I

  * Populate it
  REPLACE ALL (cField) WITH RECNO()

  GO BOTTOM
  STORE EVALUATE(cField) TO nLastID

  * Make it auto-increment
  IF NOT EMPTY(CURSORGETPROP("Database", "__AddPK"))
    ALTER TABLE (cTable) ;
    ALTER (cField) I AUTOINC NEXTVALUE nLastID+1 PRIMARY KEY
  ELSE
    ALTER TABLE (cTable) ;
    ALTER (cField) I AUTOINC NEXTVALUE nLastID+1 UNIQUE
  ENDIF

ENDIF

USE IN __AddPK
ENDIF

RETURN

```

To use this function, just pass the name of the table and the name for the new primary key field. For example, to add a primary key to the Employee table of the Office database shown in Figure 5, call the function like this:

```
AddPK("Employee", "iID")
```

This adds an autoincrementing primary key field called iID (my standard name for primary key fields) to Employee.

## Eliminating duplicated fields

Once you've added a primary key to a particular table, you can use that field in other tables to point into the original table. A field in one table that contains the primary key of another table is called a *foreign key*. When one table has a foreign key to another, there's no reason to include any other fields from the second table in the first.

For example, in the Office database, adding a foreign key to Employee to the Dept table means we can remove the ManagerNum and MgrName fields from Dept. That way, if the manager's name or employee number changes, nothing has to change in Dept. (Why would an employee number change? The company might find that it needs more digits, or might merge with another company and have to adjust employee numbers.)

Listing 5 shows AddAndPopulateFK, a function that creates a new foreign key and populates it appropriately. The function takes seven parameters, shown in Table 1. The key idea here is that one table (the "foreign key table") contains fields that relate in some way to another table (the "primary key table"). The function adds a field to the foreign key table that contains primary keys

from the primary key table. It determines what values to assign to the new field based on an expression that relates the two tables.

**Listing 5. This function adds a foreign key a table and removes fields duplicated from the parent table while preserving the relationships between the two.**

```
* PROCEDURE AddAndPopulateFK
* Add FK to specified table and populate it, based on existing data
LPARAMETERS cFKTable, cFKField, cPKTable, cPKField, cPKDataTag, ;
           cFKRelExp, aDropFields

LOCAL cPKFieldAliased, cDropClause

IF FILE(FORCEEXT(cFKTable, "DBF"))
  SELECT 0
  USE (cFKTable) EXCLUSIVE ALIAS __FKTable
  IF TYPE(cFKField) <> "N"
    ALTER TABLE (cFKTable) ADD (cFKField) I

    IF NOT EMPTY(cPKDataTag)
      USE (cPKTable) ORDER (cPKDataTag) IN 0 ALIAS __PKTable
      SET RELATION TO EVALUATE(cFKRelExp) INTO __PKTable
      cPKFieldAliased = FORCEEXT("__PKTable", cPKField)
      REPLACE ALL (cFKField) WITH EVALUATE(cPKFieldAliased) IN __FKTable
      SET RELATION TO
    ELSE
      * No index for desired tag. Use specified expression instead
      USE (cPKTable) IN 0 ALIAS __PKTable
      * Replace aliases in expression
      cFindValue = STRTRAN(STRTRAN(cFKRelExp, cPKTable + ".", ;
                                "__PKTable.");
                        , cFKTable + ".", "__FKTable.")
      REPLACE ALL (cFKField) WITH EVALUATE(cFindValue) IN __FKTable
    ENDIF
    USE IN __PKTable

    * Index on new FK
    INDEX ON &cFKField TAG (cFKField)

    * Remove extraneous fields, taking tags along
    ATAGINFO(aTags)
    cDropClause = ""
    FOR nField = 1 TO ALEN(aDropFields,1)
      IF ASCAN(aTags,aDropFields[m.nField],-1,-1,1,7) > 0
        DELETE TAG (aDropFields[m.nField])
      ENDIF
      cDropClause = m.cDropClause + " DROP COLUMN " + aDropFields[m.nField]
    ENDFOR

    IF NOT EMPTY(m.cDropClause)
      ALTER TABLE (cFKTable) &cDropClause
    ENDIF

  ENDIF

  USE IN __FKTable
ENDIF

RETURN
```

**Table 1. The parameters to AddAndPopulateFK indicate which table to change, which table to point to, and what fields can be eliminated.**

Parameter	Meaning
cFKTable	The name of the table to which the foreign key is being added. The "foreign key table."
cFKField	The name to give the new foreign key field.
cPKTable	The name of the table that contains the primary key, that is, the table to which the foreign key table should point. The "primary key table."
cPKField	The name of the primary key field in the primary key table.
cPKDataTag	The name of the tag in the primary key table to which existing data in the foreign key table corresponds. Empty if no such tag exists.
cFKRelExp	When cPKDataTag is not empty, an expression that can be used to set a relation between the two tables.  In cases where there is no appropriate index tag in the primary key table (so cPKDataTag is empty), an expression that can be used to look up a particular record from the foreign key table in the primary key table.
aDropFields	An array listing the fields currently in the foreign key table that can be removed after adding and populating the foreign key.

For example, you can call the function as follows to add a foreign key to Employee to the Dept table:

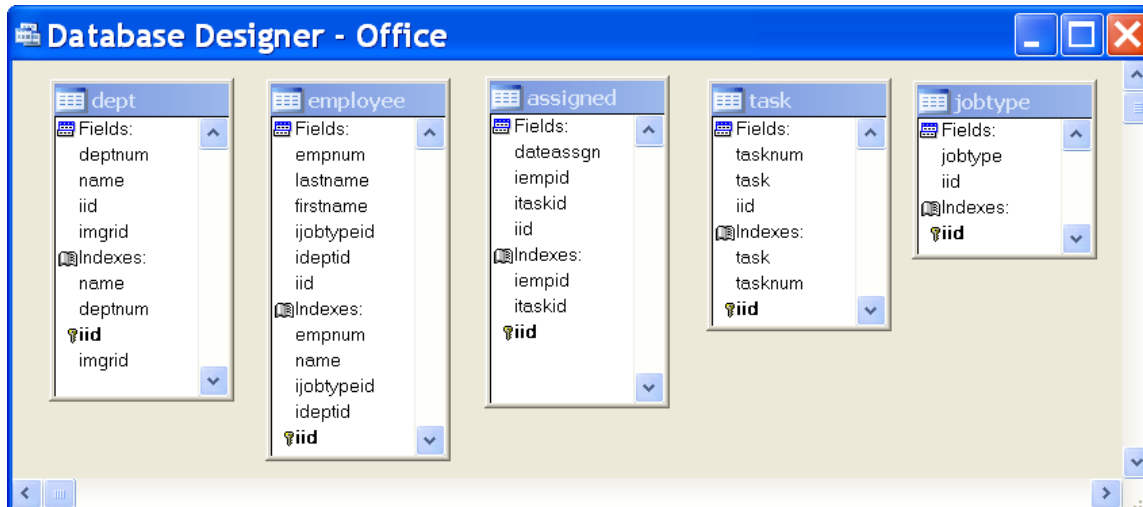
```
DIMENSION aDropFields[2]
aDropFields[1] = "ManagerNum"
aDropFields[2] = "MgrName"

AddAndPopulateFK("Dept", "iMgrID", "Employee", "iID", ;
                "EmpNum", "ManagerNum", @aDropFields)
```

The function call can be read: Add a field called iMgrID to Dept. Populate it from the iID field of Employee by looking up Dept.ManagerNum in the EmpNum index of Employee. Then, remove the ManagerNum and MgrName fields from Dept.

The function also drops any index tags based on the fields being removed.

In addition to the two functions, AddPK and AddAndPopulateFK, the session materials include the original Office database and the updated version, and a program, UpdateOffice.PRG, that makes the changes. Figure 6 shows the Office database after performing the complete set of updates.



**Figure 6.** After adding primary keys to each table, and adding and populating foreign keys, the Office database is a lot easier to work with.

## Eliminating repeating fields

Putting repeating fields (such as phone1, phone2, phone3) into a table is a common mistake for inexperienced database designers. Especially in cases where only a certain number of items are permitted, having one field or set of fields can seem quite logical.

For example, one application I've worked on handles medical billing. A particular claim is composed of up to seven services. The original designer of the application created a table where each claim is one record and there are seven sets of fields for tracking the services that are part of this claim.

There are two problems with this kind of design. The first is that it can result in very clunky code, as you have to address each of the repeated fields to do things like get the total for a claim. The second and more serious problem is that the limit can change. What happens if the rules change and a claim can contain 10 services? You have to change both the database design and the underlying code.

A better design is to store the repeated items in a separate table linked to the record to which they correspond. That is, use a one-to-many relationship. If there is a limit on the number of items that can be associated with a particular record, it can be enforced in code (though it's best to put the actual limit in data and look it up, so that changes to the limit don't require changes to the code).

Figure 7 shows a table (included in the conference materials as Original Data\CourseLoad.DBF) that might be used to track the courses being taken by each student in a high school or college. It's already on its way to being properly normalized, with a primary key and foreign keys to the student record and the course records. However, each student is limited to a maximum of five courses per semester. In addition, creating reports like class lists can be pretty ugly, since you have to check each of the five course columns. Listing 6 (ClassListOriginal.PRG in the session materials) shows a query to create a class list for a particular course in a particular semester.

Iid	Istudentid	Csemester	Cyear	Icourseid1	Icourseid2	Icourseid3	Icourseid4	Icourseid5
1	478	Spring	2007	54	867	27	94	67
2	8579	Spring	2007	854	34	665	22	0
3	4738	Spring	2007	45	235	342	634	34
4	4738	Fall	2006	82	112	233	432	0

**Figure 7.** This table shows the courses taken by a student in a specified semester. It's limited to five courses per student, and creating class lists and other reports requires some unwieldy code.

**Listing 6.** Creating a class list with the original CourseLoad table means looking is each of the iCourseIDn fields.

```

LPARAMETERS iCourseID, cSemester, cYear

SELECT iStudentID ;
FROM CourseLoad ;
WHERE cSemester = m.cSemester ;
AND cYear = m.cYear ;
AND (iCourseID1 = m.iCourseID ;
OR iCourseID2 = m.iCourseID ;
OR iCourseID3 = m.iCourseID ;
OR iCourseID4 = m.iCourseID ;
OR iCourseID5 = m.iCourseID) ;
INTO CURSOR CourseStudents

```

If one student gets special permission to take six courses in a given semester, this data structure and the code will fail.

As with introducing primary keys and foreign keys, eliminating repeating fields requires both changing the code that depends on them (in most cases, simplifying it) and writing code to preserve existing data. I haven't yet developed generic code (like the AddPK and AddAndPopulateFK functions) for making this kind of change, but writing the code for a particular case isn't hard. Listing 7 (MoveCoursesToChildTable.PRG in the session materials) shows code to create a new table with one record for each course taken. In this example, the original table is probably no longer needed; more often, the original table contains additional data other than the repeating fields. The resulting tables are included in the conference materials in the Transformed Data folder.

**Listing 7.** Converting repeating fields to records in a child table is straightforward.

```

* Create the many side of a one-to-many
* table with one course taken per record

CREATE TABLE CourseTaken ;
(iIID I AUTOINC UNIQUE, iStudentID I, ;
cSemester C(10), cYear C(4), iCourseID I)

SELECT 0
USE CourseLoad ORDER iStudentID

SCAN
IF NOT EMPTY(iCourseID1)
INSERT INTO CourseTaken ;
(iStudentID, cSemester, cYear, iCourseID) ;
VALUES ;
(CourseLoad.iStudentID, CourseLoad.cSemester, ;

```

```

        CourseLoad.cYear, CourseLoad.iCourseID1)
ENDIF

IF NOT EMPTY(iCourseID2)
    INSERT INTO CourseTaken ;
        (iStudentID, cSemester, cYear, iCourseID) ;
    VALUES ;
        (CourseLoad.iStudentID, CourseLoad.cSemester, ;
        CourseLoad.cYear, CourseLoad.iCourseID2)
ENDIF

IF NOT EMPTY(iCourseID3)
    INSERT INTO CourseTaken ;
        (iStudentID, cSemester, cYear, iCourseID) ;
    VALUES ;
        (CourseLoad.iStudentID, CourseLoad.cSemester, ;
        CourseLoad.cYear, CourseLoad.iCourseID3)
ENDIF

IF NOT EMPTY(iCourseID4)
    INSERT INTO CourseTaken ;
        (iStudentID, cSemester, cYear, iCourseID) ;
    VALUES ;
        (CourseLoad.iStudentID, CourseLoad.cSemester, ;
        CourseLoad.cYear, CourseLoad.iCourseID4)
ENDIF

IF NOT EMPTY(iCourseID5)
    INSERT INTO CourseTaken ;
        (iStudentID, cSemester, cYear, iCourseID) ;
    VALUES ;
        (CourseLoad.iStudentID, CourseLoad.cSemester, ;
        CourseLoad.cYear, CourseLoad.iCourseID5)
ENDIF

ENDSCAN

* Add tags to new table
SELECT CourseTaken
INDEX on iStudentID TAG iStudentID
INDEX on UPPER(cYear + cSemester) TAG Semester

* Remove newly extraneous fields
ALTER table CourseLoad ;
    drop iCourseID1 ;
    drop iCourseID2 ;
    drop iCourseID3 ;
    drop iCourseID4 ;
    drop iCourseID5

USE IN CourseLoad
USE IN CourseTaken

RETURN

```

Once you've made this change, creating a class list for a particular course is easy, with code like Listing 8 (ClassListNew.PRG in the session materials).

**Listing 8. When you eliminate repeating fields, queries get much easier.**

```
LPARAMETERS iCourseID, cSemester, cYear
```

```
SELECT iStudentID ;
FROM CourseTaken ;
WHERE cSemester = m.cSemester ;
AND cYear = m.cYear ;
AND iCourseID = m.iCourseID ;
INTO CURSOR CourseStudents
```

## **Build conversion code as you go**

Once you decide to make changes to the existing database, plan to build the code to convert the data to the new format as you make the changes to the application. That is, at the same time that you modify the application to use a primary key for a particular table, put the code to actually add the primary key (even if it's just a call to my AddPK function) into a program that will be called either as part of installation or the first time you the new version of the application runs.

Building the conversion code along the way lets you test as you go, and ensures that you don't end up rushing to do this when the new version is ready and the client is impatiently waiting for it.

## **Working with forms and classes**

While most of the existing applications I've encountered have lots of forms, use of classes is sporadic. Typically, some forms and controls are based on the VFP base classes while others were built using the VFP wizards and others may use a few custom-built classes. Often, this reflects increasing knowledge by the original developer; sometimes, it's the result of several developers having worked on the project over time.

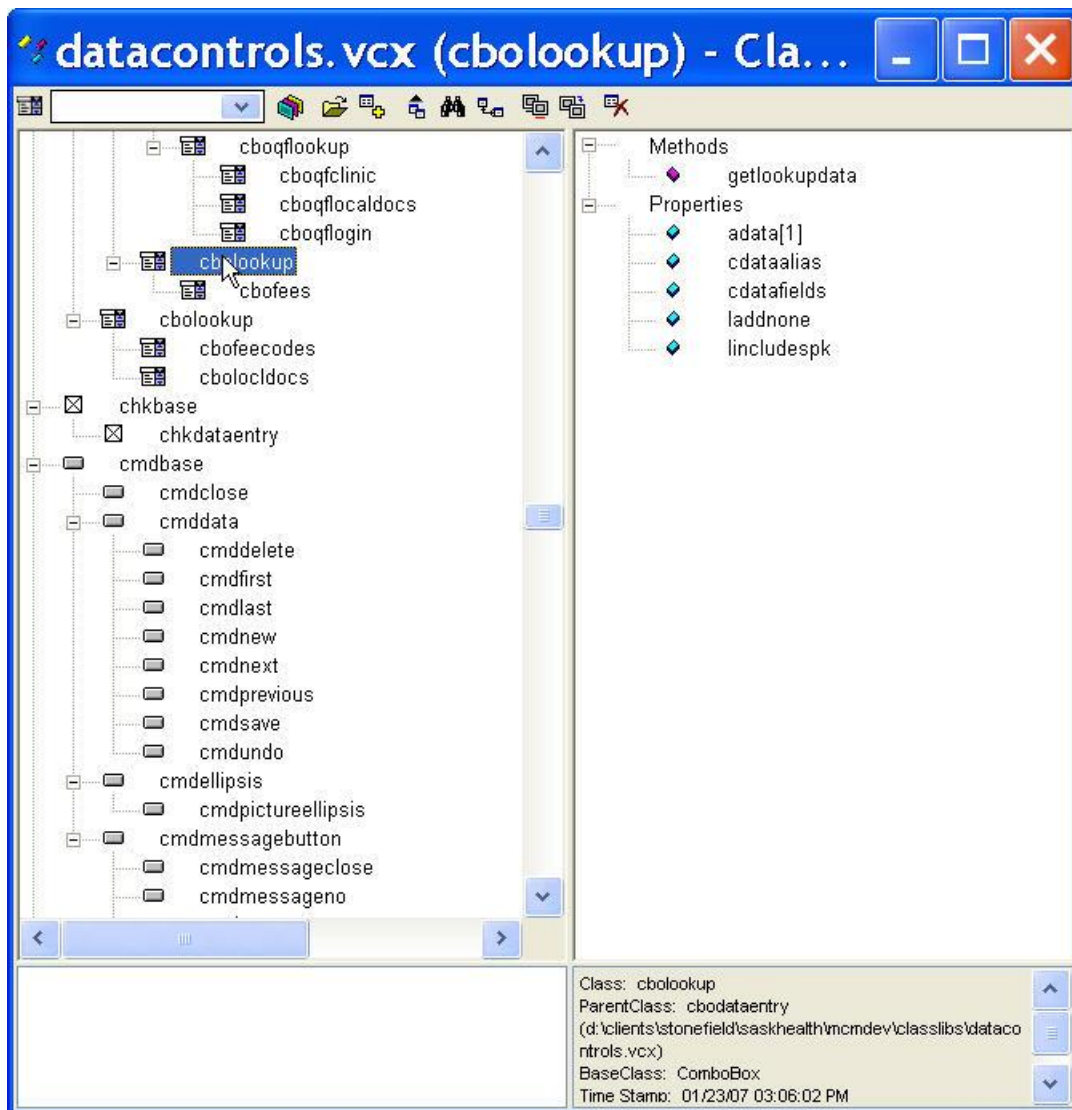
The first thing to do in this regard is to figure out what you have. What classes are present? How are they used? Are there any apparent rules? There are various tools you can use to figure all this out.

### ***The Class Browser***

The Class Browser gives you a way to look at what classes are present and get some sense of what they are without having to open each one. You can also see the class hierarchy and see what class a particular form is derived from.

To explore a particular class library, open it in the Class Browser. To see the relationships among several class libraries, open the first using the Open button, then open the others using the View Additional File button. To see the class hierarchy, make sure the Hierarchical item in the context menu is checked.

You can also open an entire project with the Class Browser. In the Open dialog, change the Files of type dropdown to Project and choose the project you want to see. Figure 8 shows part of the class hierarchy for a project I'm working on.



**Figure 8.** The Class Browser lets you see all the classes and forms in a project at once.

You can open a class directly from the Class Browser, so if something you see needs exploration, you can get there quickly.

## ***Build your own***

While the Class Browser gives you a look at what's there, you may need more information. It doesn't take much code to get some idea of what you have. For one project I worked on, I found forms based on several different form classes as well as the VFP base form class. To assess the situation, I wrote the program in Listing 9 (AuditForms.PRG in the session materials). It creates a cursor with one record for each form, providing the name, class and class library. (At that point in this project, all the files were contained in a single directory, so I didn't have to drill down.) A query on this cursor gave me a list of all the form classes in use, and examination of the cursor let me check whether there was any logic to the choice of class for each form.

**Listing 9.** This little bit of code creates a cursor listing all forms in the current directory, and showing the form class each is based on.



```

* Find out where the forms come from

LOCAL aForms[1], nFormCount, nForm

CREATE CURSOR FormClasses (mFile M, mClass M, mClassLib M)

nFormCount = ADIR(aForms, "*.scx")

SELECT 0
FOR nForm = 1 TO nFormCount
    USE (aForms[nForm, 1]) ALIAS __Form
    GO 3
    DO WHILE NOT (UPPER(__Form.BaseClass)=="FORM")
        SKIP
    ENDDO

    INSERT INTO FormClasses VALUES (aForms[nForm, 1], ;
        __Form.Class, __Form.ClassLoc)
    USE IN __Form
ENDFOR

```

Later in the life cycle of the same project, I needed to categorize all forms based on their type: data entry, dialog, reporting, and so forth, and get a count of each type. For this task, I used the Project object's Files collection. Since there turned out to be no logic to the choice of form class for the forms, I had to make the determination of type for each, but the code in Listing 10 (CountForms.PRG in the session materials) sped the task up considerably.

**Listing 10. This program spins through the forms in a project and lets you indicate the type of each form.**

```

LPARAMETERS cProject
MODIFY PROJECT (cProject) NOWAIT
oProject = _VFP.ActiveProject

DIMENSION aFormTypes[4]
aFormTypes=0

FOR EACH oFile IN oProject.Files
    IF oFile.Type = "K"
        MODIFY FORM (oFile.Name)
        cType = INPUTBOX( ;
            "1 for data entry, 2 for reporting, 3 for message, 4 for other")
        nType = val(cType)
        aFormTypes[nType] = aFormTypes[nType] + 1
    ENDIF
ENDFOR

?"Data entry forms = ", aFormTypes[1]
?"Reporting forms = ", aFormTypes[2]
?"Message forms = ", aFormTypes[3]
?"Other forms = ", aFormTypes[4]

```

As you explore and audit the forms and classes in an existing project, keep in mind that like projects, class libraries and forms are stored in VFP tables, so it's easy to look inside with VFP code.

## Begin to make changes

As you begin to work with the existing forms and classes, you may find that you need to make some kinds of changes that aren't well-supported by the Form and Class Designers. Depending what you want to change, there are a couple of tools at your disposal.

To change the parent class of a form or class, you can use the Class Browser's Redefine capability. Right-click on the object in question and choose Redefine. The Redefine dialog opens and you can choose the new parent class, making sure it has the same base class. This is handy when you need to insert another level in the class hierarchy or move classes from one hierarchy to another.

The Class Browser is also useful when you want to change the name of a class. Make sure that every class or form that refers to the class in question is open in the Class Browser. Then click on the class to change and choose Rename from the shortcut menu. Not only does the class' name get changed, but the references are corrected. Be aware, though, that only references in VCX and SCX data are fixed; the Class Browser can't update code references (such as CreateObject() calls).

HackCX from White Light Computing (shown in Figure 9) offers a way to get inside forms and class libraries. You can look at the structure, properties and methods for each record in a VCX or SCX and make changes to it, even adding properties and methods. When you're done, HackCX recompiles the file. HackCX is available in two versions, a free version and a Professional version; [www.whitelightcomputing.com](http://www.whitelightcomputing.com).

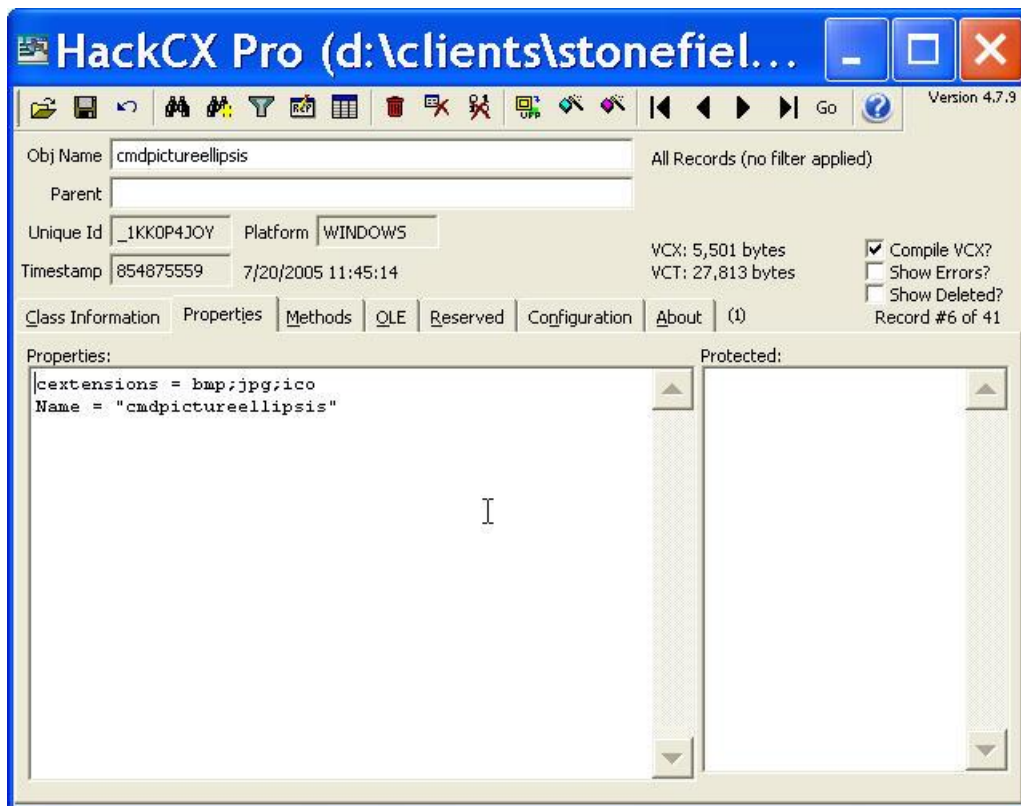


Figure 9. HackCX gives you a way to explore and change class libraries and forms.

## Rename controls for readability

One of the most common problems I find in existing code is that the original developer has left the default names for controls, so that forms are filled with text1, combo1, and so forth. Trying to understand the behavior of a form that has 20, 30, 40 or more controls with meaningless names can be trying, especially when the code isn't very good to start with.

In one application I'm working on, some forms have 20 to 25 buttons, all named command1, command2, and so forth. To make my job easier, I wrote a tool to rename the controls and fix the code. The tool, Control Renamer, can be run independently or as a builder. When you call it with a form or class open in the Form or Class Designer, it presents a list of all the controls in the form or class and lets you specify a new name for each. As you select a control in the list, it's highlighted in the Form or Class Designer, as shown in Figure 10. When you click the Rename button, both the control names and the code in the methods of the form or class is changed to use the new names. (The Control Renamer is included with the session materials; see its readme file to learn how to use it.)

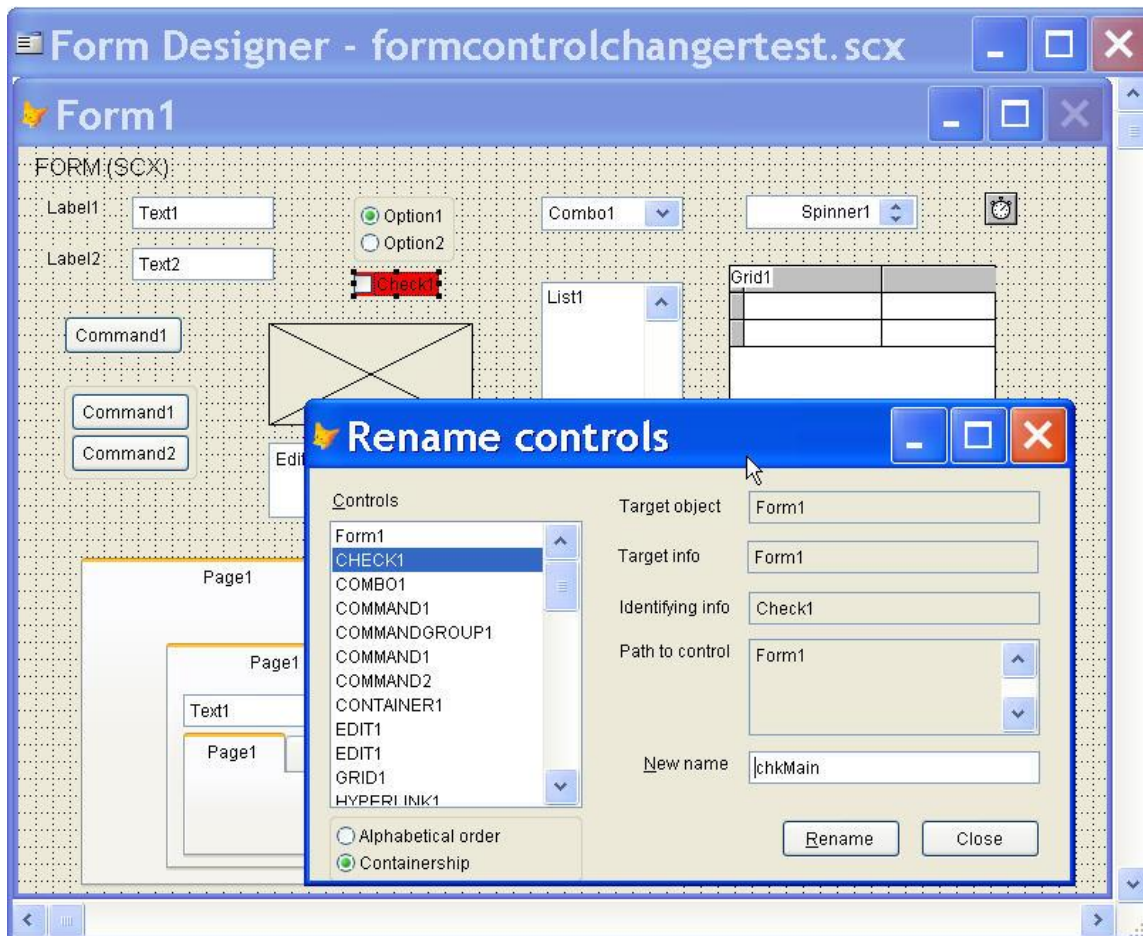


Figure 10. The Control Renamer tool lets you rename the controls in a form or class, fixing the code automatically.

## ***Refactor or replace? Start simple***

One of the hardest choices with existing applications is when to modify existing forms and when to replace them. The tough question is often which approach accomplishes the most at the lowest cost. In my experience, you often start with modifications and move to replacement later.

For example, in one project, I found early in my involvement that every form used to prepare for reporting had one button for Preview and another for Print, and that the Click method of the two buttons were identical except for the REPORT FORM command. The two Click methods each contained the same code to collect the data for the report. In addition, these forms were all based on the base Form class rather than on a subclass.

Since the code mostly worked, replacing 30 or 40 forms at once didn't make sense. But each time I needed to deal with one of these forms (perhaps to add a field to the report involved), I added a method called GetReportData and moved the data collection code there, calling it from the two Click methods.

A little later, the client asked for some new reports. At that point, I created a form class incorporating the Preview and Print buttons and the GetReportData method (along with a few other things designed to keep the form generic). I subclassed my new class for the new reports, and from that point, began replacing existing reporting forms as I needed to deal with them. Over time, I also developed some standard controls (such as to gather begin and end dates for the report) to use on those forms.

If you choose to modify rather than replace, follow the guidelines for safe refactoring so you don't break things as you go. See [www.refactoring.com](http://www.refactoring.com) or Martin Fowler's seminal book "Refactoring: Improving the Design of Existing Code" for details.

Whether you're modifying existing code or replacing it, you may want to introduce unit tests as you go. This is especially useful when you're starting with working code; unit tests will help you ensure that you end up with code that works the same way.

## ***Eliminate extra classes***

It's not unusual to find that existing applications contain a lot of class libraries where only one or two classes are actually used. Often, when developers decide to use a third-party tool (including those in the FoxPro Foundation classes), they simply pull the entire library into the project rather than copy the classes they need into a new class library.

Steven Black's Share.PRG makes it easy to prune away those you don't need. It's an add-in for the Class Browser. When you run it, it lets you point to a class library, and then puts a copy of the currently selected class and its entire inheritance hierarchy into that library. It's available at [www.stevenblack.com](http://www.stevenblack.com).

## **The bottom line**

Taking over an existing application is quite different from building a brand new application. In some ways, it's easier since you're not starting from scratch. But in other ways, it's much more difficult since you need to find a way inside the mind of another developer (or several other developers).

The tools described here are just a start. I'd love to hear what tools you find useful in this process.

The challenge of figuring out how things work and changing it can be a lot fun. So can the reaction of users as you make the application easier to use, more reliable, and more capable.

*Copyright, 2008, Tamar E. Granor, Ph.D..*