Best Practices for Vertical Application Development

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

While a vertical market application shares many of the same goals as a custom application, the bar is higher because the audience is both larger in volume and broader in scope, plus you usually have much less direct contact with the customer. This document examines some of the things to consider when designing and implementing a vertical market application.

Introduction

There are three types of software applications: custom, horizontal, and vertical. Custom applications are developed for a single client, and are highly specific to their business. Horizontal, or commercial, applications have a broad range of features, and are sold to all types of customers; examples include word processors, spreadsheets, operating systems, and so forth. Vertical applications fit a specific niche or industry, such as medical office, real estate, or law practice management software.

Over my career, I've done both custom and vertical application development, and I much prefer the latter. Here are some of the reasons:

- With custom software development, you trade your hours for dollars (or your currency of choice). Since we humans have a finite lifespan, our inventory of hours is limited. With vertical software, you write the application once and (hopefully) sell it many times over. There isn't a direct relationship between hours and income.
- You can leverage the efforts of other people. For example, one custom project Stonefield did had two developers working full-time. If one got sick or went on vacation, productivity (and therefore billables) dropped in half. With a vertical package, I can go on vacation after the software ships and our sales people and support people continue to do their jobs, so revenue doesn't stop flowing.
- Custom software development tends to be local (that is, your customers are usually in the same geographical area as you are). If your local economy takes a dive, as mine did in the 1990's, your business suffers. With vertical software, geography isn't as much of an issue, and if your software is localizable, may not be an issue at all.

Lest you think vertical development is complete Nirvana, here are some of the downsides:

- Unlike custom development, where you interim bill for the work done so far, income doesn't start until the software starts selling. That means you need a source of funds up-front. There are a variety of sources for funding: the usual ones (your savings, friends and family, angel investors, banks, etc.), as well as the revenue other parts of your business generates and even the old shoestring budget (you have a full-time job to pay the bills and develop your vertical application at night).
- You're working on spec: you're taking a risk that the software you're building will have customers. Of course, a lot of vertical packages start with a single customer who's willing to pay for the development; in other words, it started life as a custom application, with a vision to turn it into a vertical one. This is obviously the ideal approach, one we've taken advantage of, but we've also developed applications where there was no client up-front.
- The 30-70 rule: in my estimation, when you're done writing code, you're 30% of the way to having a vertical market application. There are a lot of other things to do, especially marketing and sales.
- Speaking of marketing and sales, unless you are one of the few people who are not only great developers but also good at marketing and sales, you're going to need other people to help out. Marketing a vertical application is nothing like finding clients for custom development.
- Although you may have some competitors for custom application development, there may be a lot of experienced, established, well-funded competitors in the vertical market space you've chosen.

One other thing to be aware of is that you'll likely have a looser relationship with your customers. With custom development, you meet regularly with clients: requirements gathering, prototyping, testing, implementation, post-implementation support, etc. With vertical applications, you're selling a finished product to a customer, so you may only talk to the customer during pre-sales and post-sales support. If your application sells over the Internet, you may have no personal contact with the customer at all. One of the benefits of having a personal relationship with a customer is that they're often willing to forgive small mistakes or glitches in the software. With vertical applications, you usually only have one chance to make a first impression, so it better be flawless.

Best practices for vertical application development is a huge area, one that could easily fill several volumes. In this document, I'll focus on the following topics:

- Application licensing and activation: the pros and cons of licensing your software, licensing and activation tools, and online activation.
- Error reporting: reporting errors to support or development staff with enough information to reproduce or fix the bug quickly.
- Application maintenance models: the pros and cons of subscription vs. pay-as-you-go updates, how to decide who is eligible, subscription management.
- Version update mechanisms: manual vs. automatic update notification and installation.
- Support policies: the pros and cons of different types of support delivery and policies.

As supporting background, here are some of the vertical applications I've been involved in:

- Tourism Information System: used by government tourism departments to manage and track tourism information requests. It was originally developed as a custom application, but was then sold to governments of several other Canadian provinces.
- Pharmacy Partner: a retail pharmacy application. Again, it started as a custom application, but was sold to 35 other pharmacies in Saskatchewan and Alberta.
- Stonefield Data Dictionary, Stonefield AppMaker, Stonefield Database Toolkit: these tools were sold to thousands of FoxPro developers worldwide.
- Stonefield Query: this initially started as a FoxPro add-on tool, but was quickly remarketed as custom reporting tools for a variety of commercial applications, including the Sage Accpac ERP, Sage Pro ERP (formerly known as SBT), Sage BusinessVision, Simply Accounting, and AccountMate accounting systems, GoldMine, ACT, and SalesLogix customer relationship management (CRM) systems, and HEAT help desk software. This application is sold to end-users of these systems, plus the Stonefield Query SDK (formerly called the Developer's Edition) is sold to software developers (some of whom are FoxPro developers).

While I discuss Stonefield Query in this document, this isn't intended as an extended commercial or sales pitch for our product. It just happens to be one that uses many of the practices I discuss in this document, and I decided it's better to show a real-world application rather than some made-up example.

Application Licensing and Activation

Some people estimate software piracy costs software developers billions of dollars. Don't think it can happen to you? I assumed so too, and over the years was disappointed to find even fellow FoxPro users, who as software developers should have been more sensitive to piracy than the average consumer, were not above ripping off my creations. I discovered some companies that bought a single license and deployed it on dozens of machines. As a result, I've come to believe in software protection and licensing.

There are two ideas here. Application activation means installing the software isn't enough to be able to run it; it also needs to be activated. Application licensing means not everyone can run the software; the customer needs to purchase one license for every user who uses the application.

Although these are different concepts, they are related. For example, if the user purchases two licenses, how do you prevent them from installing it on ten computers?

I discussed this topic in some detail in my June 2000 FoxTalk article, "May I See Your License?" See that article if you want code that implements these ideas; in this document, I discuss the concepts in detail rather than showing a lot of code.

Application Licensing

Licensing your application requires two things: knowing how many licenses a customer has purchased and a license manager responsible for knowing how many licenses are in use at one time.

In our case, when the user purchases a number of licenses of Stonefield Query, we provide an activation code (I'll discuss application activation in the next section) that has the number of licenses encrypted within it. You can store this value in a number of locations, including a table or some other type of file or the Windows Registry. In our

case, we decided to create a license file named SFQuery.LIC. This file contains binary, encrypted information about each of the licenses the user purchased, and allows multiple licenses and multiple license types. For example, the user may initially purchase two professional licenses, so we provide a two-user professional license activation code. Later, they may purchase five runtime licenses, so we provide a second activation code for this license; the new license information is appended to the license file so it contains information about both. Regardless of where you store the license information, it must be encrypted to prevent the user from changing the information to give them more or different types of licenses than they've paid for.

At application startup, our application object instantiates user and license manager objects that collaborate. The license manager reads the contents of the license file so it knows how many licenses of what type are available. The user is asked to log into the application; after they do, we check how many users are currently logged in, and if the number of licenses has been exceeded, we give them a warning message and terminate the application.

There are a variety of mechanisms for tracking logged-in users. A common one is to set a logical field in the appropriate record of a users table to .T. to indicate the user is logged in, and set it to .F. when they exit. The thing I don't like about this mechanism is the out-of-sync situation caused when a user's system crashes; the field isn't updated, so the system thinks they're still logged in. Instead, we use a LOGINS table and lock the record for the user. To count how many users are currently logged in, we count the number of locked records. When the user closes the application or if the application crashes, their lock is released so there's never an out-of-sync situation.

Here's the code from the CheckUserCount method of the license manager. It returns .T. if there are unused licenses available:

```
local lnMaxUsers, ;
  lnSelect, ;
  lnReprocess, ;
  lnUsers, ;
  llReturn
lnMaxUsers = This.GetLicenses()
lnSelect   = select()
select 0
use LOGINS shared
lnReprocess = set('REPROCESS')
set reprocess to 0.2 seconds
lnUsers = 0
scan
  if rlock()
    unlock
  else
    lnUsers = lnUsers + 1
  endif rlock()
endscan
use
set reprocess to lnReprocess
select (lnSelect)
llReturn = lnUsers < lnMaxUsers
return llReturn
```

Here's code taken from the user manager that checks whether there are licenses available by calling the method above and if so, locks the appropriate record in the LOGINS table. If it can't, that means another user is logged in with this name, so the application gives a warning and terminates. Notice that LOGINS is left open for the life of the application so the lock is maintained.

```
do case

* Other cases here.

* If the number of concurrent users is OK, lock a record in the logins table.
* If we can't, give an error and return.

  case oLicense.CheckUserCount()
    use LOGINS in 0 shared
    if not seek(padr(This.cUserName, len(LOGINS.USER)), 'LOGINS', 'USER')
      insert into LOGINS values (This.cUserName)
    endif not seek(padr(This.cUserName ...
    if not rlock('LOGINS')
```

```
* display error to user
      use in LOGINS
      llReturn = .F.
    endif rlock('LOGINS')

* We've hit the maximum number of concurrent users.

  otherwise
* display error to user
    llReturn = .F.
endcase
```

Application Activation

There's a spectrum of software protection mechanisms:

- At the low end, there's burning the customer's name into the software (or more likely, a resource such as the Registry or some hidden file) and displaying it on the opening splash screen and other locations. In practice, this doesn't work well: I've seen the names of other companies in the "licensed to" section of the splash screen of Microsoft Word and other applications at clients' offices. Also, this doesn't help the case where a company buys one copy and deploys it to many users; all the splash screens show the same company name.
- At the high end, applications require the presence of a dongle (a hardware device that plugs into one of the system's ports) to function. Buy one license, get one dongle. The downside of this approach is that dongles get lost or broken, may interfere with the system in some ways, and need to be physically shipped to the customer, which makes Internet sales more complicated.
- In the middle, there's application activation. When the application is installed, it looks for something that makes the system it's installed on unique (hard drive serial number, processor serial number, MAC address of the network card, etc.) and uses that to activate the program. Buy one license, get one system-specific activation code. If someone installs the software on another system and tries to activate the software using the same code as the other machine, it'll fail. The downside of this approach is the inconvenience caused if the user has to change the unique features the activation is tied to (their hard drive is replaced, they get a new system, etc.). In that case, they need a new activation code.

Our activation and licensing mechanism is shown in **Figure 1**. Here's how it works:

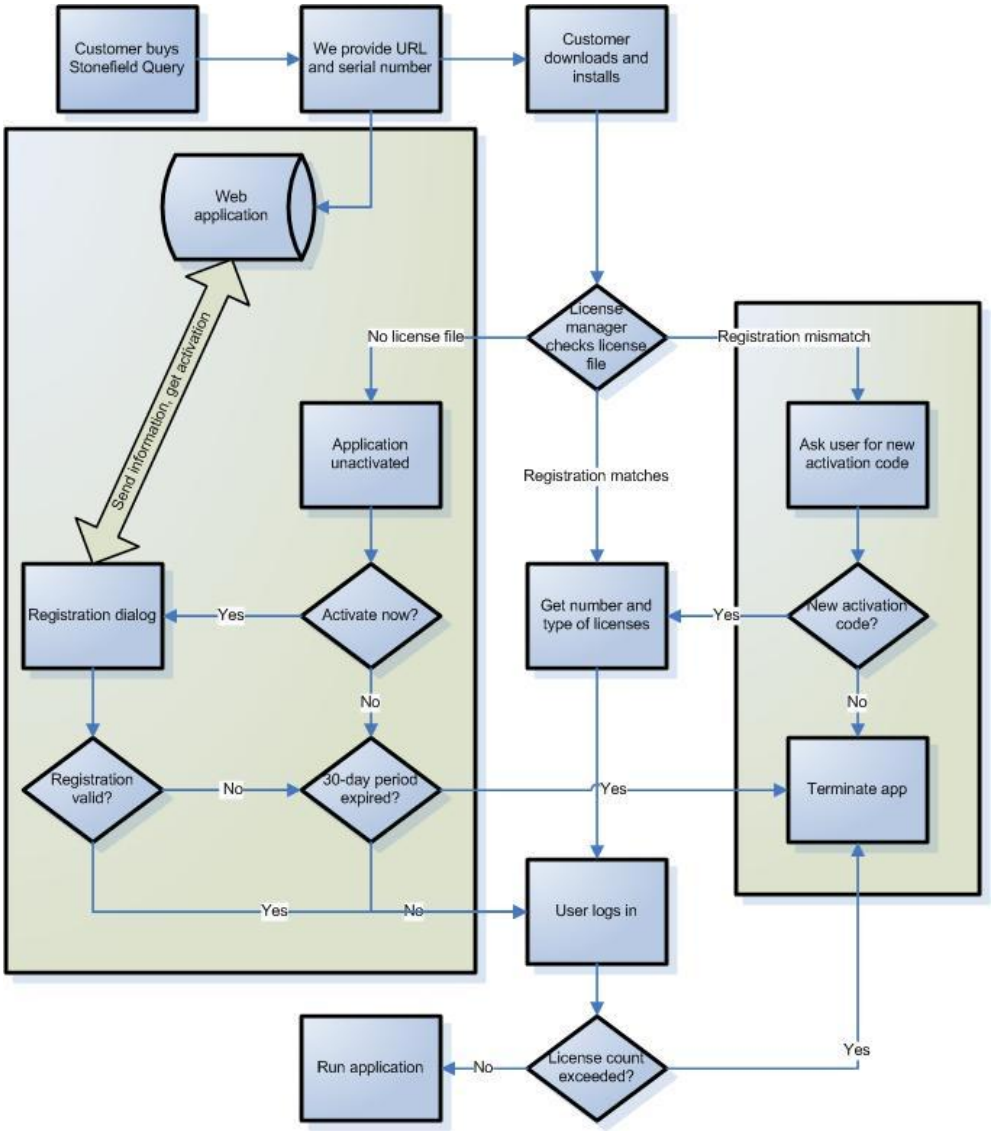

Figure 1. This flowchart shows the activation and licensing mechanism Stonefield Software uses.

- The user purchases Stonefield Query. We provide them with a link to download it from our Web site and a unique serial number.
- The user downloads Stonefield Query from our Web site and installs it.
- When they run it, our application object instantiates a license manager object. The license manager sees there's no license file (SFQuery.LIC) on disk, so the program runs in "unactivated" mode. This used to mean limited functionality—the user couldn't print or output reports to file—but we recently changed it to a 30-day period. If the application hasn't been activated within 30 days, the user can no longer run the software until they do. Why so draconian? To prevent piracy: we had cases where a customer gave someone else our SETUP.EXE. Without activating it, the user of the pirated copy could create and preview reports. Sure they couldn't print them, but some people saw that as a fair tradeoff for saving hundreds of dollars.
- Every time they run the program in unactivated mode, the dialog shown in **Figure 1** appears. They're reminded the application is unactivated, told how many days they have left (or if the 30-day period has expired), and asked if they want to activate it now. If they choose No and the 30-day period hasn't expired, the program runs as normal. If they choose No and it has expired, the application terminates.

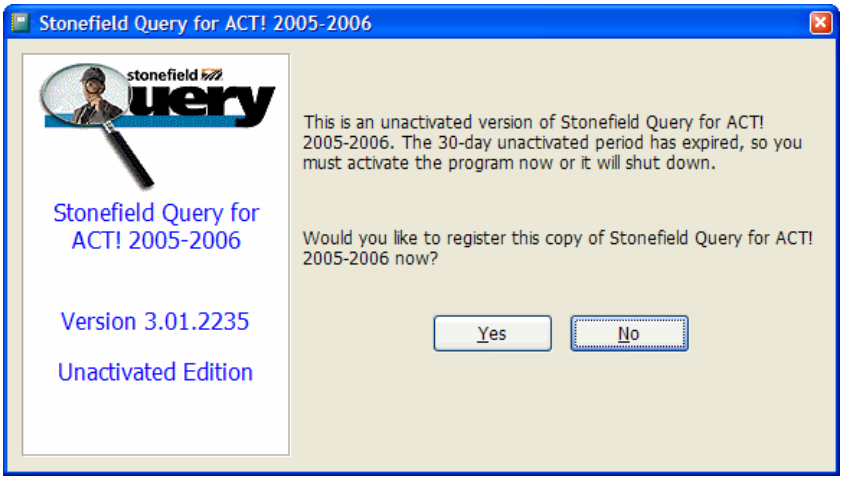

Figure 1. This dialog appears every time the user runs the unactivated application.

- If the user chooses Yes, a registration dialog (**Figure 2**) appears. In the final step, a registration number is displayed, along with textboxes where they can enter the serial number they were provided when they bought it and an activation code. The registration number is tied to their particular computer (I'll discuss how in a moment). The activation code is based on four things: the registration number, the serial number, the number of licenses purchased, and the type of license (we have both professional and runtime licenses). So, a given activation code won't work on another system. If a pirate enters the serial number and activation code assigned to someone else, the Finish button won't enable because the program ensures the values match each other.

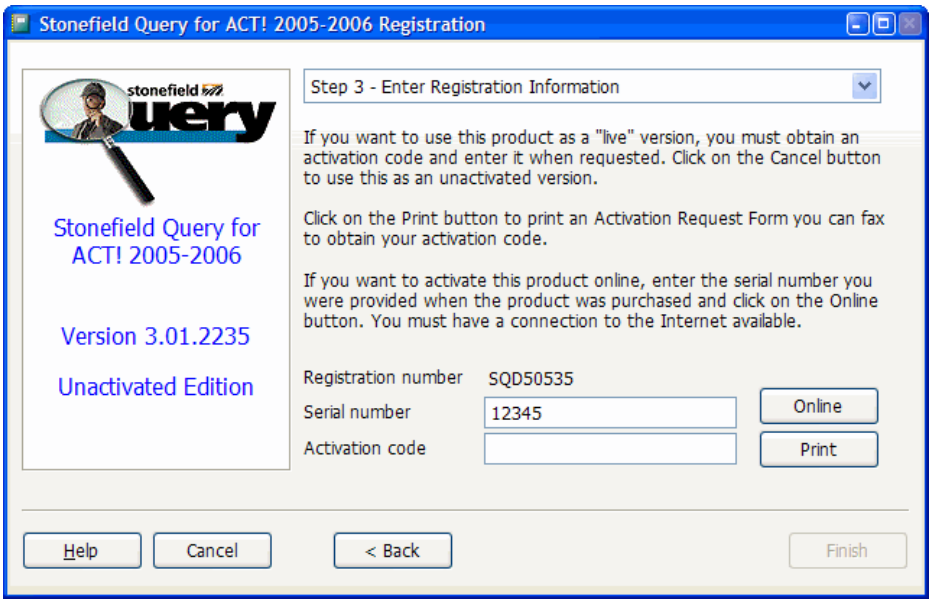

Figure 2. The Registration dialog allows the user to activate the application.

- The user can obtain the activation code in one of two ways. They can contact us by email, fax (the Print button prints an Activation Request Form they can fax), or phone, providing the registration number displayed in the dialog, and we'll provide the matching activation code. This mechanism isn't very convenient for the user or for us, because it means someone must be available to provide the activation code, which prevents activation at night and on weekends and holidays. The second mechanism is much more convenient: the user clicks the Online button and after a few seconds, the activation code is filled in and the Finish button enabled. No human intervention is required.

- Once the program is activated, the activation information is written to the license file. Every time Stonefield Query starts, it calculates the registration number for this computer and compares that to the activation information in the license file. If the values don't match, the user is warned of the problem and told they need to contact us for a new activation code. This is intended to stop people from giving a copy of their license file to another user. Unfortunately, it also has the consequence that legitimate users who change computers need to obtain a new activation code as well. It's not an ideal situation, but one we haven't come up with another solution for yet.

The registration number is a concatenation of a product code (because we have different versions of Stonefield Query, we distinguish them with a code like "SQG" for GoldMine and "SQH" for HEAT) and an algorithm that processes the serial number of the hard drive the program is running from. For obvious reasons, I won't show you the algorithm we use, but here's the code used to obtain (among other things) the hard drive serial number.

```
*============================================================================
* Method:           GetVolumeInformation
* Status:           Public
* Purpose:          Returns information about the specified volume
* Author:           Doug Hennig, from code taken off MSDN
* Last revision:    02/02/2006
* Parameters:       tcVolume - the volume to get information for
*                   taArray  - an array to hold information about the volume:
*
*          Column  Information
*          ------  -----------
*           1    Volume name
*           2    Volume serial number
*           3    File system name (FAT, NTFS, etc.)
*           4    .T. if the file system supports case-sensitive filenames
*           5    .T. if the file system preserves case of filenames
*           6    .T. if the file system supports Unicode filenames
*           7    .T. if the file system preserves and enforces ACLs (NTFS
*                only)
*           8    .T. if the file system supports file-based compression
*           9    .T. if the volume is compressed
*          10    maximum filename length
*
* Returns:          .T.
* Environment in:   none
* Environment out:  none
*============================================================================

lparameters tcVolume, ;
  taArray
local lcVolume, ;
  lcVolumeName, ;
  lnVolumeNameLen, ;
  lnVolumeSerialNumber, ;
  lnMaxFileNameLen, ;
  lnFileSystemFlags, ;
  lcFileSystemName, ;
  lnFileSystemNameLen, ;
  lcFileInfo, ;
  lcFileName, ;
  laFiles[1], ;
  lnFiles, ;
  lnI, ;
  lcFile, ;
  lnHandle

* Define constants.

#define cnFS_CASE_SENSITIVE           0
#define cnFS_CASE_IS_PRESERVED        1
#define cnFS_UNICODE_STORED_ON_DISK   2
#define cnFS_PERSISTENT_ACLS          3
#define cnFS_FILE_COMPRESSION         4
#define cnFS_VOL_IS_COMPRESSED       15
```

```
#define ccNULL                      chr(0)

* Declare the API function.

declare GetVolumeInformation in Win32API ;
  string lpRootPathName, string @lpVolumeNameBuffer, ;
  integer nVolumeNameSize, integer @lpVolumeSerialNumber, ;
  integer @lpMaximumComponentLength, integer @lpFileSystemFlags, ;
  string @lpFileSystemNameBuffer, integer nFileSystemNameSize

* If the path wasn't specified, use the current drive. Otherwise, get the
* drive for the specified path, handling UNC paths specially.

do case
  case vartype(tcVolume) <> 'C' or empty(tcVolume)
    lcVolume = addbs(sys(5))
  case left(tcVolume, 2) = '\\'
    lcVolume = addbs(tcVolume)
    lcVolume = left(lcVolume, at('\', lcVolume, 4))
  case len(tcVolume) = 1
    lcVolume = tcVolume + ':\'
  otherwise
    lcVolume = addbs(justdrive(tcVolume))
endcase
if empty(lcVolume)
  lcVolume = addbs(sys(5))
endif empty(lcVolume)

* Create the parameters for the API function, then call it.

lcVolumeName         = space(255)
lnVolumeNameLen      = len(lcVolumeName)
lnVolumeSerialNumber = 0
lnMaxFileNameLen     = 0
lnFileSystemFlags    = 0
lcFileSystemName     = space(255)
lnFileSystemNameLen  = len(lcFileSystemName)
GetVolumeInformation(lcVolume, @lcVolumeName, lnVolumeNameLen, ;
  @lnVolumeSerialNumber, @lnMaxFileNameLen, @lnFileSystemFlags, ;
  @lcFileSystemName, lnFileSystemNameLen)

* Convert the serial number if necessary.

if lnVolumeSerialNumber < 0
  lnVolumeSerialNumber = 4294967296 + lnVolumeSerialNumber
endif lnVolumeSerialNumber < 0

* Put the information into the array.

dimension taArray[10]
taArray[ 1] = left(lcVolumeName, at(ccNULL, lcVolumeName) - 1)
taArray[ 2] = lnVolumeSerialNumber
taArray[ 3] = left(lcFileSystemName, at(ccNULL, lcFileSystemName) - 1)
taArray[ 4] = bittest(lnFileSystemFlags, cnFS_CASE_SENSITIVE)
taArray[ 5] = bittest(lnFileSystemFlags, cnFS_CASE_IS_PRESERVED)
taArray[ 6] = bittest(lnFileSystemFlags, cnFS_UNICODE_STORED_ON_DISK)
taArray[ 7] = bittest(lnFileSystemFlags, cnFS_PERSISTENT_ACLS)
taArray[ 8] = bittest(lnFileSystemFlags, cnFS_FILE_COMPRESSION)
taArray[ 9] = bittest(lnFileSystemFlags, cnFS_VOL_IS_COMPRESSED)
taArray[10] = lnMaxFileNameLen

* If the serial number is 0 (which happens with Win95/98 systems on remote
* drives), open a file on the drive and get the file information for it.

if lnVolumeSerialNumber = 0
  declare GetFileInformationByHandle in Win32API ;
    integer lnHandle, string @lcFileInfo
  declare integer CreateFile in Win32API ;
    string @lcFileName, integer dwDesiredAccess, ;
    integer dwShareMode, integer lpSecurityAttributes, ;
    integer dwCreationDisposition, integer dwFlagsAndAttributes, ;
```

```
    integer hTemplateFile
  declare CloseHandle in Win32API ;
    integer lnHandle
  lcFileInfo = space(255)

* If a file was specified for the volume name, use it. Otherwise, find some
* file we can open.

  if file(tcVolume)
    lcFileName = tcVolume
  else
    lnFiles    = adir(laFiles, lcVolume + '*.*')
    lcFileName = ''
    for lnI = 1 to lnFiles
      lcFile   = lcVolume + laFiles[lnI, 1]
      lnHandle = fopen(lcFile)
      fclose(lnHandle)
      if lnHandle >= 0
        lcFileName = lcFile
        exit
      endif lnHandle >= 0
    next lnI
  endif file(tcVolume)
  if not empty(lcFileName)
    lnHandle = CreateFile(@lcFileName, 0, 0, 0, 3, 0, 0)
    if lnHandle >= 0
      GetFileInformationByHandle(lnHandle, @lcFileInfo)
      CloseHandle(lnHandle)
      lnVolumeSerialNumber = ctobin(substr(lcFileInfo, 29, 4), '4RS')
      taArray[2] = lnVolumeSerialNumber
    endif lnHandle >= 0
  endif not empty(lcFileName)
endif lnVolumeSerialNumber = 0
return .T.
```

Online activation works as follows:

- When the user purchases the software, our license managing program assigns them a unique serial number and sends an update message to an ASP.NET application on our Web server, specifying the license information and serial number.
- When the user clicks the Online button in the registration dialog, an activation request is sent to the same application on our Web server, providing the user information, serial number, and registration number. This application performs a number of tests:
  - Was a valid password provided (prevents someone from validating pirated software if they discover how to call the online application)?
  - Does the serial number exist in the registration table (prevents a pirate from typing any value they want for the serial number)?
  - Has the serial number already been activated (prevents a pirate from using a legitimate user's serial number)?
- Assuming these tests pass, the application determines the activation code, updates the registration table to indicate that the specified serial number has been activated, and returns the code to the calling program.
- The registration dialog puts the return value into the activation code textbox and enables the Finish button if online activation succeeded, or displays the error message returned by the Web application if not.

Other Licensing Ideas

The mechanism we use may not work for you, depending on your needs. Fortunately, there are lots of alternatives. For example, in the January 2006 issue of FoxTalk, Barbara Peisch wrote an article titled "Are You Registered" that addresses this topic. Here's her approach:

- In addition to discussing the volume serial number, she presented some other ideas to obtain unique information about a computer using Windows Management Instrumentation (WMI), including the processor ID and network ID.
- Her application has multiple modules, and the customer may not purchase all of them. To indicate which modules are available to a particular customer, she sets certain bits in a numeric value. Each bit position represents a different module, so 23 (binary 00010111) might indicate the customer has purchased only the Job Costing, Accounts Payable, General Ledger, and Human Resource modules.
- She combines the two values—the unique identifier of the machine and the modules-purchased code—using an algorithm described in her article to produce a registration number.
- At application startup, her application reads the registration number from the Registry (which is where she stores it), validates that the machine identifier portion matches that for the machine, and uses BITTEST(ModulesPortion, BitNumberForDesiredModule) to determine which modules the user can access.

If you don't feel like rolling your own code, there are a number of commercial packages you can purchase that provide this for you, including SoftwareKey (www.softwarekey.com), PC Guard (www.sofpro.com), and SoftwarePassport (www.siliconrealms.com). InstallShield also has a service related to this.

I looked at SoftwarePassport (**Figure 3**) fairly closely and liked it a lot. Unfortunately, it didn't quite fit our model, so we didn't implement it, but under different circumstances I definitely would. SoftwarePassport compresses and encrypts your EXE and adds a wrapper around it that handles all of the licensing requirements. It has features such as:

- Control over how the machine is identified: hard drive size and serial number, size of physical memory, network card MAC address, and others. By combining several of these, you can avoid needing to assign a new license to a user who changes one thing.
- Detection of license-breaking attempts, including date rollback (attempting to keep a trial period going by changing the system date to an earlier value), using memory dump programs to view the application's memory, and using memory patching programs that change the application's code after it's been loaded into memory.
- Configurable key expiration. This allows you, for example, to create a 30-day demo of your program without writing a single line of code.
- Stolen code tracking, which prevents someone from entering a known stolen code to activate the program.
- Keys can include custom information, such as the module bits Barbara's mechanism uses.
- Configurable (including localizable) license dialogs, such as expiration warnings and purchase dialogs.
- It has an API so you can do things like assigning keys from within your own in-house application rather than using the SoftwarePassport user interface.

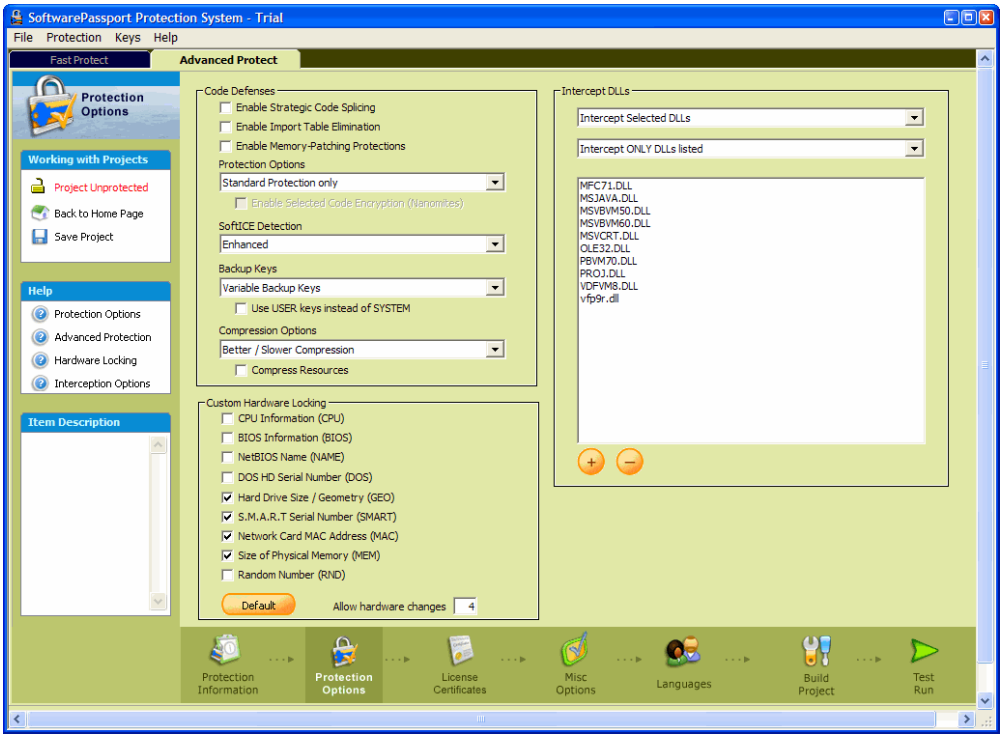

Figure 3. SoftwarePassport provides complete application licensing and protection services to applications.

In addition, SoftwarePassport can tie into the Digital River marketing system, which provides complete software distribution services including a Web store, software downloads, and online activation. Of course, there's a fee for that above and beyond the cost of SoftwarePassport itself, but for a small company with little infrastructure for all of the chores involved in software distribution, it might be well worth it.

Error Reporting

Error handling is a key component in any application. The services a good error handling mechanism must provide are:

- Logging the error to some persistent storage, such as a table or text file.
- Displaying a useful message to the user. Errors are usually scary things to users, so information such as the cause of the error and what the user can do about it is helpful.
- Recovering from the error. "Recovery" may include retrying the command that failed (for example, a resource was temporarily unavailable but it may be available now), staying in the application but not returning to the point of error, or terminating the application.

I've written extensively about error handling and won't go into detail on the basics here. For background, please see the Error Handling white paper available at http://www.stonefield.com/techpap.html.

However, for vertical applications, error handling is even more important than it is in custom applications. If a bug exists in your application, you don't have one customer with a problem; you have hundreds. Since you often have limited contact with your customers, and we know users rarely report problems themselves, your error handler needs to provide as much information as possible to your support and development staff so they can resolve the problem quickly. This means providing a way to communicate this information to your company.

Stonefield Query's Error Handling

Stonefield Query handles errors in several ways:

- As much as possible, we try to anticipate "hot spots," places in the program more likely to cause errors than others. Opening a table, writing to file, performing a query, and evaluating an expression entered by the user are examples of known places that can fail. We wrap these sections with TRY blocks and try to deal with the error with a descriptive warning message, such as that shown in **Figure 4.**

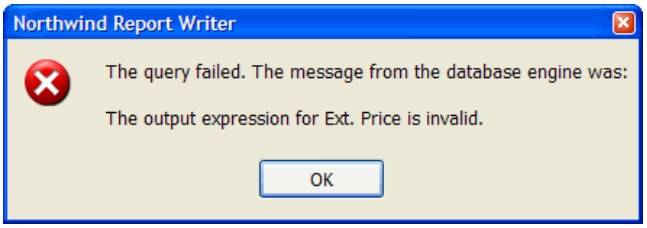

Figure 4. A warning dialog displayed when anticipated problems occur.

- Queries can fail for a variety of reasons without necessarily causing an error in VFP: the query returns no records, the SQL statement is invalid for the database engine being used (some databases require joins in the WHERE clause, for example), the user doesn't have access to the tables being queried on, and so forth. Again, we display a descriptive warning message in these cases.
- Unexpected errors are those that don't happen commonly and we didn't anticipate (although we try to learn from them and handle them more gracefully in the future). These aren't necessarily programmer bugs (for example, there may be a corrupted file), but they did trigger our error handler. These aren't a lot of fun, because unlike the previous types, we have less information about the environment in which the error occurred and therefore our recovery options are more limited.

Because they're harder to deal with, let's discuss the unexpected error scenario in more detail. Here's how our error mechanism works:

- As described in my white paper, all of our base classes have code in the Error method that escalates the error using a Chain of Responsibility design pattern. Errors bubble up the class hierarchy first: subclasses can handle the error or DODEFAULT to move up until the base class code is executed. If the error hasn't been handled, it then goes up the containership hierarchy: each object's Error method calls its parent's Error method, until the top of the containership tree is reached. At that point, if the error still hasn't been handled, the application's global error handler is called.
- Not described in the white paper (since it was written several years ago): because error information can be lost in all the delegation, the Error methods of our base classes retrieve and store as much information as possible before passing the error on to something else. There are four special things this method does:
  - It uses AERROR() as early as possible. Some commands and functions, most notably TYPE(), can affect the error information, so we need to capture it before it's potentially altered.
  - If the code causing the error used a TRY block but threw the resulting Exception, the Error method would catch the error, but the error would be "unhandled structured exception" rather than the actual error. So, before using THROW, our code stores the Exception object in a custom oException property. The Error method sees that oException contains an object, and replaces the information in the array filled by AERROR() with the correct information in the oException object, then nulls oException so we don't inadvertently use an old object again next time.

```
lparameters tnError, tcMethod, tnLine
* Some code omitted from here
lnError  = tnError
lcMethod = tcMethod
lnLine   = tnLine
lcSource = message(1)
aerror(laError)
with This
  if vartype(.oException) = 'O'
```

```
    lnError  = .oException.ErrorNo
    lcMethod = .oException.Procedure
    lnLine   = .oException.LineNo
    lcSource = .oException.LineContents
    laError[cnAERR_NUMBER]  = .oException.ErrorNo
    laError[cnAERR_MESSAGE] = .oException.Message
    laError[cnAERR_OBJECT]  = .oException.Details
    .oException = .NULL.
  endif vartype(.oException) = 'O'
endwith
```

- To ensure the global error handler has the most accurate information possible, the Error method of the object calls the SetError method of the error handler, passing it the information it retrieved. This information is stored in properties of the error handler, which are then used by the various error handling methods for logging and possibly display purposes.

```
if pemstatus(oError, 'SetError', 5)
  oError.SetError(lcMethod, lnLine, lcSource, @laError, lcProperties)
endif pemstatus(oError, 'SetError', 5)
```

- Now that the error handler object is set up properly, it's time to do its thing. First, it logs the error to a table called ERRORLOG.DBF. In addition to the usual information (error number, method, date and time, name of the user, and so forth), it logs a memory variable dump using LIST MEMORY TO FILE and object property dump using LIST OBJECTS TO FILE. Because this lists all variables and all objects, including those of the error handler itself, that file is read into a string and lines containing known variables and other information we don't need are removed. The string is then written to a memo field of ERRORLOG.DBF.
- The error handler then displays the dialog shown in **Figure 5**. The user can enter comments, such as what they were doing when the error occurred. Print runs a report (ERROR.FRX) that includes the user's name, phone and fax numbers, and email address (this comes from configuration information the user fills out), and information about the error, including the variable and object dump. Email creates a text file with all of the error information and attaches it to an email sent to our support email address. Under certain conditions, we also attach other things to the email, such as a copy of the report the user was trying to run, which is very helpful when trying to reproduce the problem. Save creates a text file with the error information and opens it for viewing. The Continue and Quit buttons allow the user to stay in or exit the application, respectively.

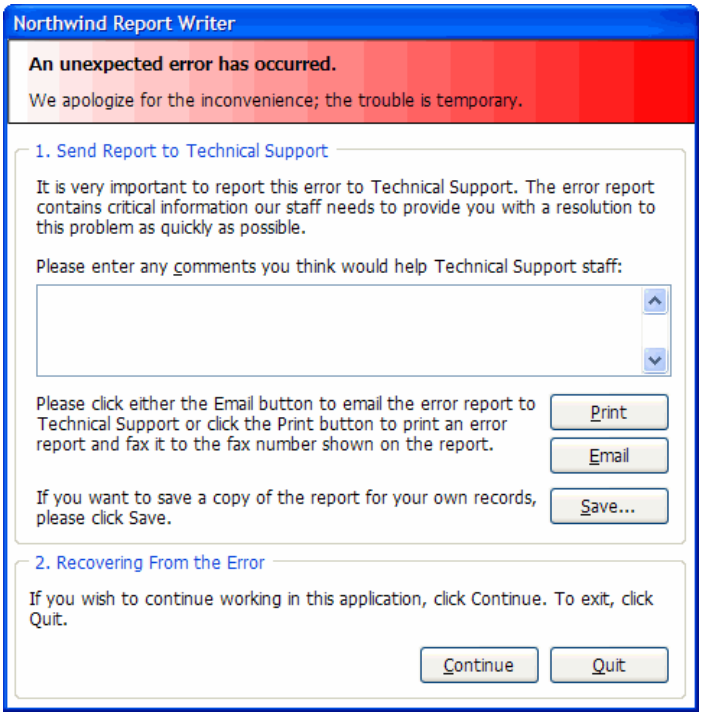

Figure 5. This dialog appears when the global error handler catches an error.

- This dialog doesn't appear if the user is running in "developer" mode. I don't necessarily mean within the IDE; we have a special mode that indicates the application should act differently when an error occurs. In this case, this dialog is replaced with a simpler dialog (**Figure 6**) with options to cancel, continue, retry, quit, or bring up the debugger. The latter is usually the course of action I take because it's the quickest way to track down the bug.

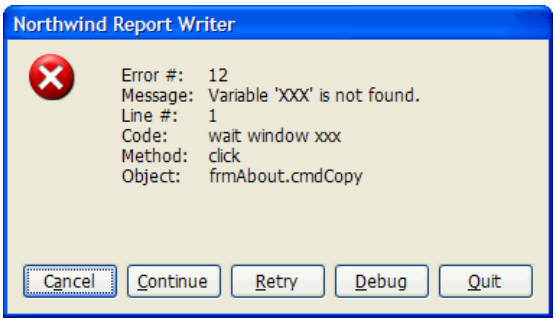

Figure 6. This dialog is displayed instead of that in Figure 5 when running in "developer" mode.

- Dealing with the error is trickier in VFP 8 and later than before. In earlier versions, Continue would RETURN TO the method of our application object containing the READ EVENTS statement for the application. This prevented returning to the original method where the error occurred, because that would likely result in more errors. Quit would terminate the application. The problem with newer versions of VFP is TRY structures: you can't use RETURN inside a TRY structure. No problem, you're probably thinking; if the error occurred within a TRY structure, it'd be handled locally. Unfortunately, that isn't necessarily the case. If a method of an object is called within a TRY block, an error occurs in that method, and the object has code in its Error method, the Error method is called. Because of our error handling mechanism, execution may eventually end up with the global error handler. We can't use RETURN TO there because there's still a TRY structure on the call stack. Conditional code to check whether a TRY is in effect by checking SYS(2410) doesn't work because unfortunately that function doesn't return the proper value in

this particular case. The solution is somewhat kludgy: before each TRY statement, the code sets oError.lInsideTRY to .T. and resets it to .F. after the ENDTRY. The error handler checks lInsideTRY; if it's .F., the user is given the option to continue using the application and RETURN TO is used. Otherwise, the user is informed the application will exit once they close the error dialog.

Diagnostic Logging

In addition to error reporting, we've also instrumented Stonefield Query to log diagnostic information about the application's processes. This is sort of like how some modern cars (especially the expensive ones) keep a history of engine performance in the on-board computer, so a technician can quickly diagnose problems even if they don't occur while he's observing the car.

In my October 2003 article in FoxTalk, titled "But it Works for Me," I provided an instrumentation object I use to log various milestones of an application's life. (Lisa Slater Nichols discussed a similar class in her December 2005 article "Log4Fox: A Logging API for Visual FoxPro.") SFLogger is actually a very simple class: its LogMilestone method simply writes the specified message, along with the number of seconds elapsed since the last time it was called, to a text file if logging is turned on. We don't want logging turned on all the time, so we use a simple mechanism to flag that logging should occur: the presence of a file called LOG.TXT. If users are experiencing problems, whether an error occurs or not, we ask them to create an empty LOG.TXT file, run the application, and then email us the resulting diagnostic file. We haven't instrumented the entire application, just those sections prone to problems (opening ODBC connections, performing a query, evaluating calculated field expressions, etc.).

Here's an example that logs the performance of a SQL SELECT statement. Note that we can't necessarily use SYS(3054) (although we do if it's native VFP data we're querying on), but we can show the SQL SELECT statement used, the number of records retrieved, and the time to execute the statement.

```
lcLogMessage = 'SFDataEngine.PerformQuery' + chr(13) + 'Retrieving data'
oLogger.LogMilestone(lcLogMessage)
oLogger.StartProcess()
* execute a SQL SELECT statement
lcLogMessage = transform(reccount()) + ' records' + chr(13) + ;
  'tcSelect = ' + tcSelect
oLogger.LogElapsedMilestone(lcLogMessage)
```

With our error handling and diagnostic mechanisms, we have yet to encounter a bug that wasn't quickly tracked down and fixed.

Other Error Handling Ideas

As well as our error handler works, I've recently come across some other interesting ideas.

Rick Strahl of West Wind Technologies has a great tool for creating help files called HTML Help Builder. When an error in Help Builder occurs, the dialog shown in **Figure 7** appears.

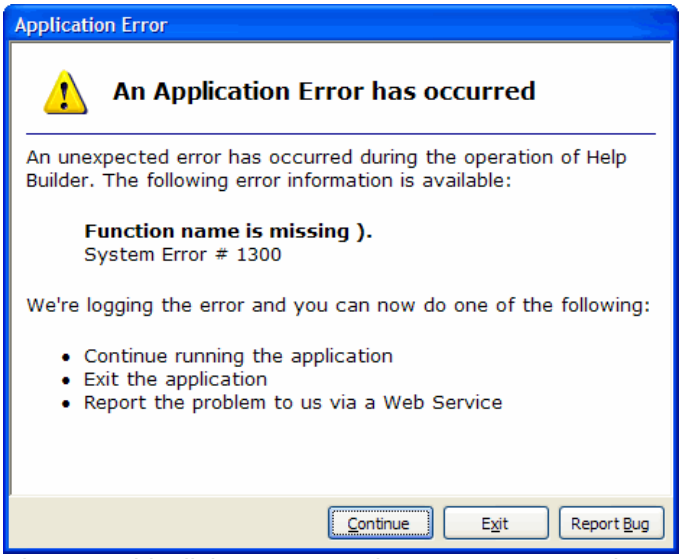

Figure 7. This dialog appears when an error occurs in HTML Help Builder.

The three choices are to continue, exit, or report the bug. As you expect, Exit terminates the application. Continue does something interesting: it actually shuts the application down and then restarts it. This is likely to avoid the TRY problem discussed earlier.

The "report bug" option displays the dialog shown in **Figure 8**, which gives the user a chance to enter as much information as possible about the error, and then submits it to a Web service on the West Wind site for logging purposes. This avoids possible issues with email, such as not having the email settings configured correctly.

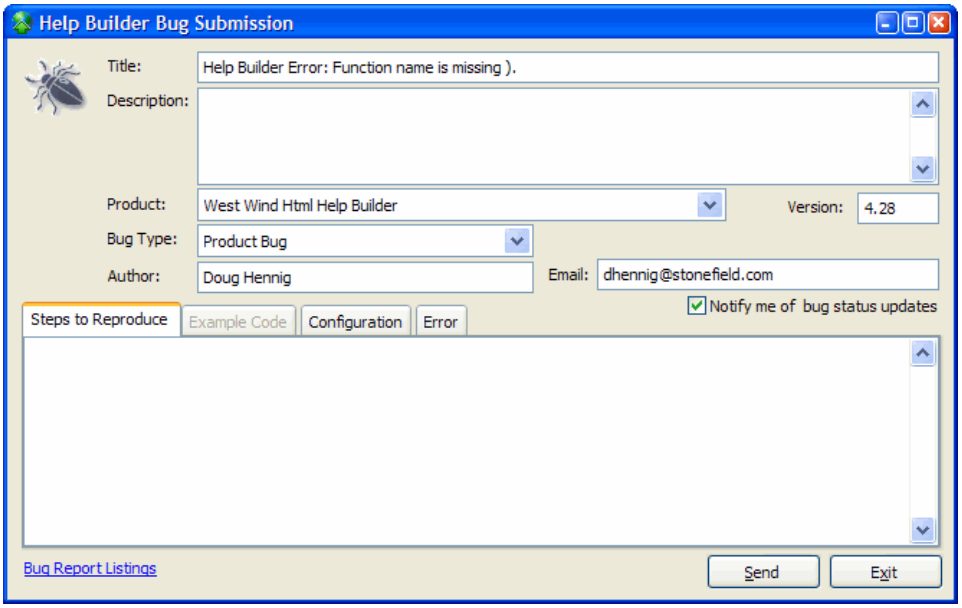

Figure 8. The Help Builder Bug Submission dialog makes it easy for the user to report bugs to West Wind Technologies.

There are also a couple of useful options here: the Notify Me checkbox specifies that you want to be notified about the bug's status by email and the Bug Report Listings link displays logged bugs (**Figure 9**) so you can find out what problems others have experienced with the application (so you can avoid them or at least know that the problem isn't just on your system).

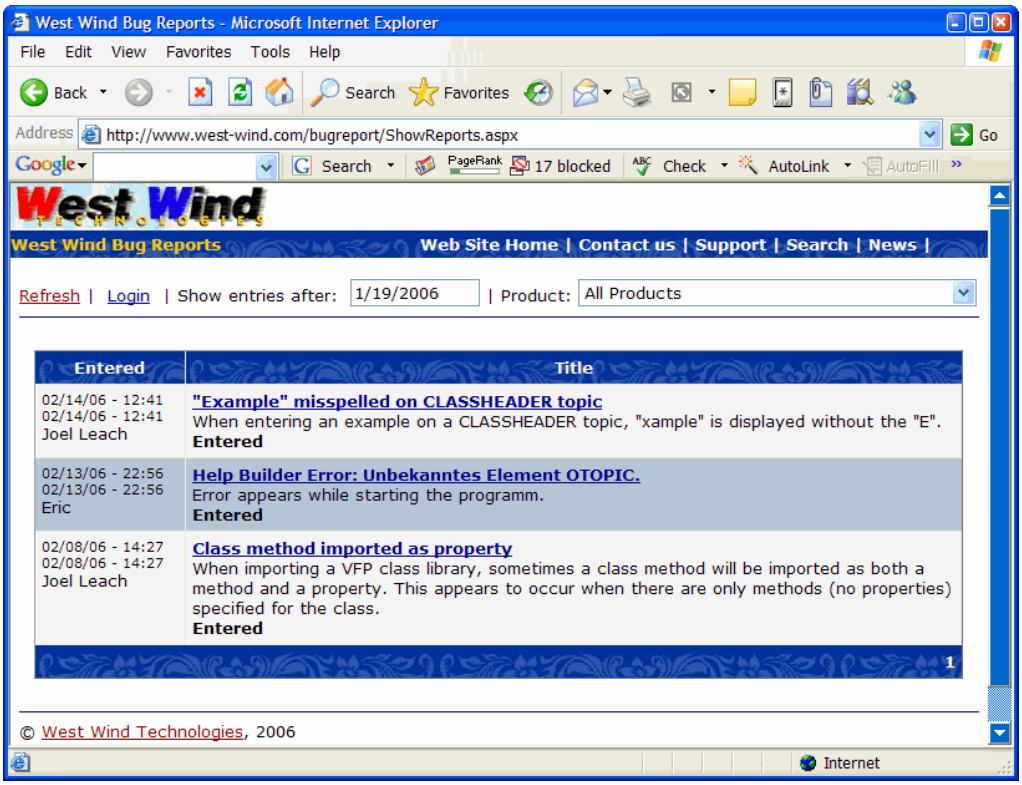

Figure 9. The Bug Report Listings option displays this Web page, showing reported bugs in the application.

At his session on RSS at the 2005 Southwest Fox conference, Rick Borup mentioned a cool use of RSS his company employs: communicating error reports. I won't go into detail about RSS and how it works in this document, but here's what Rick does in a nutshell. When an error occurs, his error handler creates an XML string formatted to the RSS specification and then posts it to his Web site. Rick (or others) can subscribe to the bug report RSS feed on his Web site and receive instant notification about problems in his applications in the RSS reader of his choice. Again, this eliminates possible problems with email. Although he didn't mention it, it's possible he logs fixes to errors in his RSS feed as well, so his staff and even users can quickly see the status of errors, similar to the bug report listings West Wind provides.

One thing we've thought of, but haven't implemented yet, is having the error dialog we display provide links to a knowledge base on our Web site, or even look up the error number and method using a Web service and display some "what to do about this" information in the dialog.

Dealing with the Error

Obviously, the purpose of the error handler reporting problems to you is so you can resolve them. Hopefully, the error information your error handler logged provided enough information for you to quickly discover the source of the error, or reproduce it yourself. Some problems aren't due to bugs, so the resolutions is usually on the customer side: the user deleted a file, so you replace it, or a file became corrupted, so you fix it or they restore from backup. When there's a bug, however, what do you do about it?

There are a couple of trains of thought on this:

- The bug is added to the list of issues to work on in the next release.
- The bug is fixed immediately and a "hot fix" sent to the customer.

I tend to prefer the latter approach, because it resolves the customer's problem and provides goodwill. However, if the bug is obscure, affects a small number of customers, doesn't cause damage to data, or is easily worked around until it's fixed, sometimes we'll use the former approach. One of the issues with shipping a hot fix is you need to be

careful of other things you were working on that aren't complete or fully tested, and deal with dependencies, such as a new feature you just added that requires a new or updated DLL, in which case you have to include that with the hot fix.

Application Maintenance Models

The great thing about software is that it's never done. Lots of things drive change: Users constantly demand new features, government regulations change, new operating systems are released, and so forth.

There are two ways you can be compensated for the work invested in a new release: a la carte and subscription. With the first approach, you charge an upgrade fee to customers who get the new version. With the second, customers pay in advance and receive all upgrades during the subscription period. The more vertical an application, the more likely the vendor is to use the second mechanism. For example, with most horizontal software, you pay for the software once, and if you want a newer version, you buy it at either full price or pay a smaller upgrade fee. Most of the vertical applications I've encountered charge an annual fee for "software maintenance," which is just another term for subscription. Note that this is not to be confused with another use of the word "subscription," in which software licenses expire at the end of a subscription and the user can no longer use the software (for example, Microsoft Action Pack).

With the subscription model, the revenue stream is more regular, because subscriptions come up for renewal every month (although some people prefer that all customers renew at the same time every year). It's also more predictable: if you sell a lot of software one month, you expect you'll have a lot of subscription renewal income one year later. With a la carte upgrade fees, it's more of a crap-shoot. You only get upgrade revenue after the release of an upgrade and you hope customers find enough new features in a particular upgrade to make it worthwhile purchasing it.

It's also easier to manage upgrade eligibility with the subscription approach. If someone's subscription is current, they get the upgrade. If not, they don't. With a la carte, there are several approaches:

- If the software is cheap enough, you simply make the customer buy it again. For example, you have to buy tax preparation software again each year, although some vendors throw in some goodies for returning customers.
- Some vendors provide free upgrades within a specific version. For example, anyone who buys SnagIt version 7.0 from Tech Smith gets free upgrades to any new releases in the version 7 series. However, they have to pay for version 8 or later upgrades.
- The most common approach is to charge a certain percentage of the new price as an upgrade fee. Most customers don't expect to pay more than 50%, and unless the upgrade is a major new release, often expect to pay a fraction of the new price. So, setting the price for an upgrade can be as complex as deciding on the full price in the first place. You may decide to offer customers who purchased the software shortly before a new release (such as three months) a free upgrade.

Having used both mechanisms, I prefer the subscription approach. It does require managing subscriptions and notifying customers whose subscriptions are coming up for renewal, plus you have to make sure you release upgrades, even small ones, regularly or else customers will stop renewing. However, in my opinion, the benefits outweigh these issues.

Subscription Management

Let's look in detail at the business processes Stonefield uses for subscription management. I'm not saying these processes are perfect; in fact, we're constantly fine-tuning them as needs change. They just illustrate one way you can manage subscriptions. Different types of businesses, especially those where you have more direct contact with your customers, may need completely different processes.

Customer Purchase

When a customer purchases Stonefield Query, we check in our contact management system (we use GoldMine) whether they're already there (they may already be a customer or may be there because they downloaded a demo version), and add them if not. In addition to the usual information, we also record what type of product was purchased (Stonefield Query for GoldMine, the Stonefield Query SDK, etc.), how many licenses, what type of

license (we offer professional and runtime licenses), their subscription expiry date (a three-month trial subscription is included in the purchase; not all companies do it that way), and the number telephone support minutes they're entitled to (I'll discuss that more later in the Support Policies section). Rather than entering the license into GoldMine, we actually use a custom license manager application to capture the license information; it integrates with GoldMine, adding records as necessary.

The custom application generates a serial number for the customer's license. Why do we need a serial number? That's our link from the customer's purchase to activation. As I mentioned earlier, the serial number identifies a license for activation purposes, and in fact is part of the information used to generate the activation code. The serial number can be generated a variety of ways: a SYS(2015) value or a GUID if you want unique, non-ordered values, the next available value from an auto-incrementing field if you want sequential values, and so on.

After saving the new license, the application sends an update message to an ASP.NET application on our Web server, specifying the license information and serial number. This information is used for online activation of the user's license, as I discussed earlier.

We also record a transaction in our accounting system and process the customer's credit card or check. We then email the customer a serial number, a link to download the software from our Web site, and the password needed for installation.

Note that some of this is manual labor. One of the things we're looking at is an online store, which would automate some or most of this.

Software Activation

As I discussed earlier, the customer can only use Stonefield Query for 30 days without activating it. They can activate it by contacting us, in which case a staff member will look up their license in our license manager application and assign an activation code from the registration number provided by the user and their serial number. However, the preferred mechanism is online activation, since it's a much less manual process. On a regular basis, we run a "web sync" process that downloads newly activated licenses and updates the database on our server with the activation code and date activated.

Subscription Renewal

Our software subscriptions are initially for a three-month period, starting with the purchase of the software. After that, the customer can purchase one-year or longer subscriptions. Some software companies like to synchronize renewals so all customers renew their subscription on the same date (for example, December 31), but we decided that each customer's renewal would be separate (although if they have multiple licenses purchased at different times, we do synchronize all license renewals to a single date).

At the start of every month, the subscription manager (Christine) runs a report (written in Stonefield Query, of course <g>) showing which subscriptions lapse within that month. She generates quotes for subscription renewal and emails them to the main contact at each customer. She then follows up with phone calls about two weeks later; we found that our subscription renewal rate went way up when we started doing that. If the customer renews, she updates the subscription expiry date in the license manager.

Note that this is "push" process; we have to contact the customer. We're moving to a "pull" process, where the customer contacts us. Example of this are ZoneAlarm and Norton Anti-Virus, which remind you when your subscription is almost expired and provide you with an online mechanism for renewal. **Figure 10** shows the dialog that ZoneAlarm displays when your subscription is due. It tells you the advantages of renewing and provides buttons to renew now or remind you later. The key to this type of renewal reminder is having the application know when the subscription expires. One way to do this is to provide a new subscription/activation code with each renewal, with the expiration date encrypted within the code.

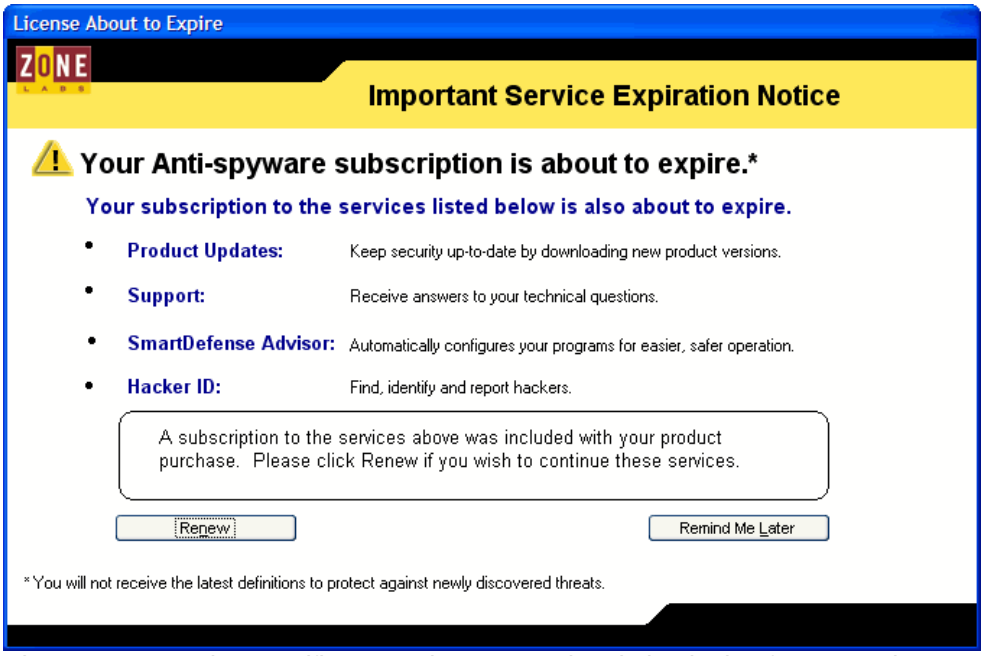

Figure 10. ZoneAlarm notifies you when your subscription is due for renewal.

In our case, although most customers renew, some do not, for a variety of reasons:

- They don't use the application anymore. Perhaps they no longer use ACT, so they have no need for Stonefield Query for ACT. Sometimes the main person using the product left the company and didn't train their replacement about it.
- They feel the subscription isn't worth it because we didn't release enough updates during the past year, or add enough new features.
- Some customers try to use the subscription as leverage: "We'll renew once you add feature X."

It's really important to determine why customers choose not to renew, because it gives you an opportunity to fix a problem you may not know you had.

Other Subscription Issues

Here are some other things to consider:

- Is your subscription renewal mandatory? In other words, the customer must stop using software like the Microsoft Action Pack if they choose not to renew. This is not a very common mechanism but may fit your business model.
- Do you allow someone to renew fewer licenses than originally purchased? While this seems like a reasonable request (perhaps they purchased more licenses than they initially needed, expecting to bring more users online, and then plans changed), it could also be a way for someone to cheat you. For example, if they purchased two licenses but only renew one, what's to stop them from downloading updates and installing it on both machines?
- Do you issue a new activation code upon subscription renewal? This is one way around the previous issue, since the new code will only work for the machine it's issued for. ZoneAlarm works this way: when you renew your subscription, they provide a new code you must enter into the application. Another benefit of this was discussed earlier: the expiry date can be encrypted within the code so the application knows when the subscription is due for renewal.
- Do you allow someone to purchase new licenses if their subscription for existing licenses has expired?

Version Update Mechanisms

There are a variety of ways you can release updates to your customers, including:

- Manual: send them an update CD.
- Manual online: notify the customer of the new version and they download it from your Web site. We password-protect our SETUP.EXE and only provide the password to current subscribers. Others, such as Eqeus, the maker of XFRX, have a section on their Web site only available to subscribers (who must log in) where downloads are available.
- Automatic: the application "calls home" (i.e. checks something on your Web site) to see if a new version is available (it either does this automatically or prompts the user first) and if so, tells user to download and install it or does that automatically. Many applications, including HTML Help Builder and ZoneAlarm, use this mechanism.

We currently use the second mechanism but are implementing the third. To some extent, it's related to subscription management, because our application needs to know whether the customer's subscription is current. However, that's not to say an update has to be free. For example, SnagIt from TechSmith doesn't use a subscription model, but does check for updates and if one is available, shows the user a list of new features and the cost to upgrade (**Figure 11**). For articles describing how this works, and even VFP code to handle it, see my articles in the September and October 2000 issues of FoxTalk or Rick Strahl's white paper at http://www.west-wind.com/presentations/wwCodeUpdate/codeupdate.asp.

If you don't feel like rolling your own, other mechanisms are available. InstallShield has an update subscription service that can manage this for you. ClickOnce, normally used to deploy .NET applications, can also be used to deploy VFP applications; see Craig Boyd's article in the CoDe Focus on Sedna magazine, available online at http://www.code-magazine.com/Article.aspx?quickid=0703072.

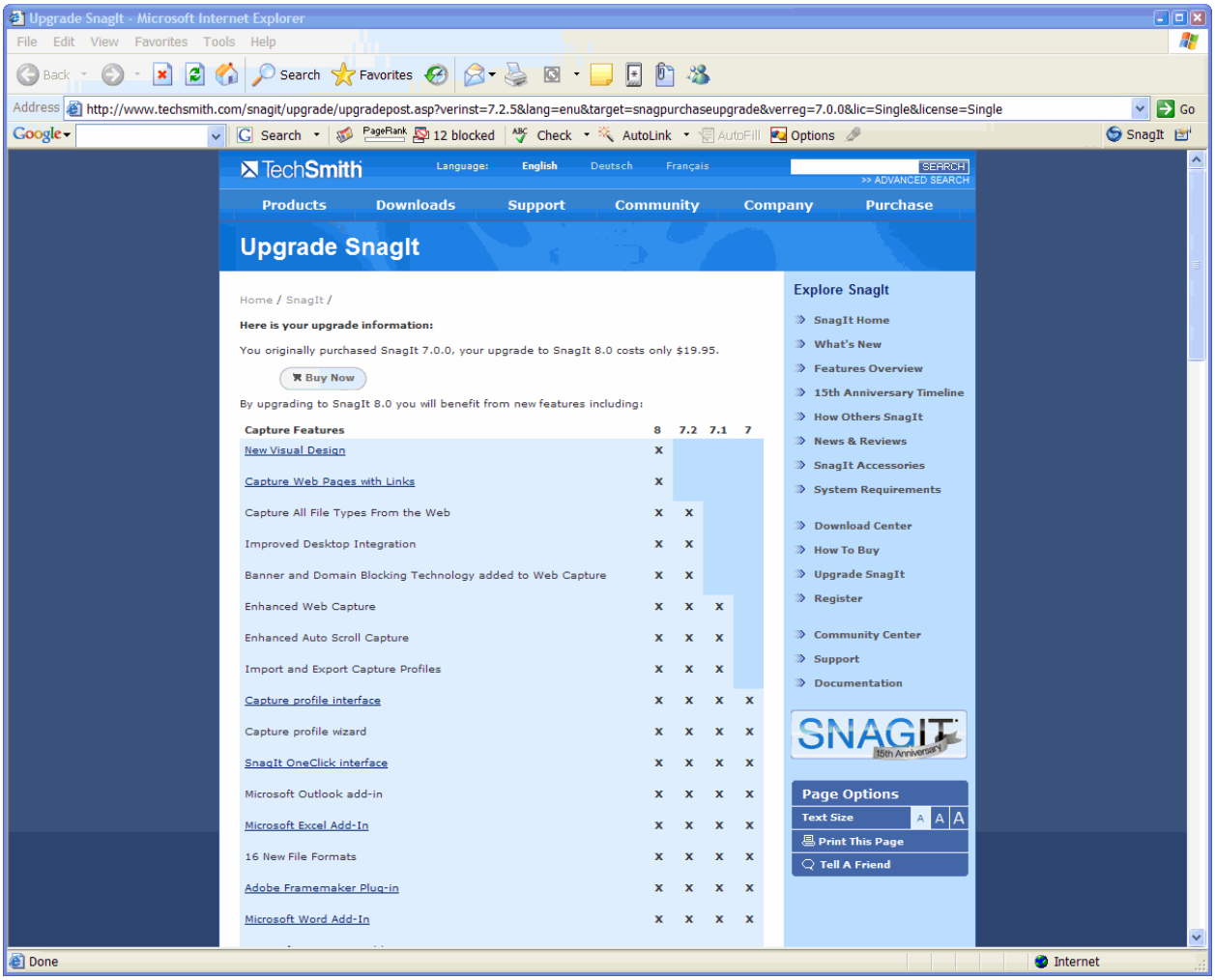

Figure 11. SnagIt tells you when a new version is available and optionally takes you to a Web site where you can see what's new and order the upgrade.

I'm not going to discuss the steps and tasks necessary to release an update; a great book by Rick Schummer, Rick Borup, and Jacci Adams called "Deploying Visual FoxPro Solutions" (Hentzenwerke Publishing) goes into great detail on best practices for updating your applications. One thing to be aware of, however, is that unlike custom applications, where you have one client so you can release updates every day, with vertical applications, you may have hundreds or thousands of customers, so update release management is much more important. If there's an error in the update or the update process, a lot of customers will be affected.

If you choose the second mechanism, where you manually notify customers when an update is available, in addition to contacting current subscribers, you should also contact those whose subscriptions has lapsed. If the list of new features is enticing enough, you may bring some back to the fold.

Support Policies

Regardless of how user-friendly your application is, you're going to have to provide technical support to your customers. Some people need hand-holding during installation, you have to deal with errors, and most people don't read the documentation you spent hundreds of hours writing (I don't sound bitter, do I? <g>) so they call wanting to know how to do something.

It seems like there are as many different support plans as there are software companies. Here are some issues to consider when you create yours:

- Is support tied to subscriptions? We decided to do this for a couple of reasons: as a further incentive (besides updates) for someone to renew and to avoid having to support old versions of the software

someone can't update because they don't have a subscription. However, other companies provide support whether you're on a subscription plan or not, and cut off support for older versions.

- What do you define as technical support? Some things are obvious, like installation issues or errors, but what about the "how do I" questions? We decided that those types of calls are really consulting rather than technical support, and charge extra for them.
- Do you put a limit on the amount of support someone can receive? Some companies, especially those that charge a lot for their applications, provide unlimited support. We used to as well, but economic reality has caused us to scale this back.
- Do customers pay a la carte or in advance? Some companies require you to purchase support contracts (which are sort of like subscriptions but cover only support) or support incidents in advance, which means, of course, that you must manage this. We decided not to, and instead charge on a per-call basis.
- Do you charge by the incident or by the minute? The benefit to the customer of per incident is a ceiling on the cost of resolving a problem, but they also burn up incidents with questions that take 10 seconds to answer.
- Do you charge for some types of support and not others? For example, we provide free email and message board support, but charge for telephone support for SDT. For Stonefield Query, we provide free email and message board support and between 30 and 60 minutes of free telephone support, depending on the version.
- Should you implement incident tracking software? If you have dedicated support staff, the answer is yes, since it allows them to see what work was done for what customer, when, and by whom, and ensure incidents don't fall between the cracks. For smaller organizations, it might be more trouble than it's worth.
- The joys of telephone support: customers can interrupt you at any time (not as big of a problem if you have dedicated support staff, but a hassle if you're trying to write code), you have to decide how to prevent someone from jumping the support queue by phoning, some people want to tell you their life stories, some people would never dream of being abusive in person but seem to feel it's OK over a telephone, etc.
- An online knowledgebase is a great thing to have, but can be a lot of work and expense to set up. We don't currently have one, but have been slowly building an internal one with the idea of making it available to our customers someday soon.

Summary

Vertical software development can be much more rewarding than custom development because it allows you to leverage your time and resources. However, in some ways it's considerably more difficult than custom development, especially when it comes to the marketing and business policy side, so it's not for everyone. In this document, I discussed some of the best practices (in my opinion, anyway) for vertical application development, including application licensing and activation, error reporting, application maintenance models, version update mechanisms, and support policies. Feel free to contact me with any ideas or questions you have in this area.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro, and the My namespace and updated Upsizing Wizard in Sedna. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor and Advisor Guide. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward~VFP).