A Generic Import Utility, Part 1

Doug Hennig

Data entry forms aren't the only way your users want to put data into their applications. Often, important data is stored in other applications, so you need a way to import that data into your application. This article, part 1 of a two-part series, presents a generic import utility you can add to your applications to provide import capabilities from a variety of data sources.

I've written custom file import routines a hundred times. Sometimes I need to import a text file into a customer table, other times it's an Excel file into an invoices table. Getting tired of doing the same thing over and over, I decided it was time to write a generic import utility.

In thinking about such a utility, I came up with the following requirements:

- Separate engine and user interface components. Sometimes I want to import without any user interface, other times I want the user to specify the import settings. Also, it makes it easy to replace the user interface if you want it to more closely resemble the rest of your application.
- Support for a wide variety of file types, include comma-delimited, fixed-length text files, Excel, and even DBF files (especially handy for converting from an older application to a new one with different data structures).
- Ability to specify which fields in the import file map to which fields in the table being imported into, including data conversion functions.
- Support, via subclasses, for duplicate checking and other data validation.

In this issue's article, we'll look at the engine classes. We'll examine the user interface classes in the next issue.

Figure 1 shows the hierarchy of classes in the engine components. (In case you're wondering, I created this diagram using the Class Designer in Visual Studio 2005.) In this diagram, a line with an open arrow indicates association (for example, instantiating SFMappingImportCollection into the oMapping property of SFImport) while a line with a closed arrow indicates subclassing (SFImportFileSDF is a subclass of SFImportFileText, which is a subclass of SFImportFile). A class indicated as "MustInherit" is an abstract class, one that isn't used directly, but must be subclassed.

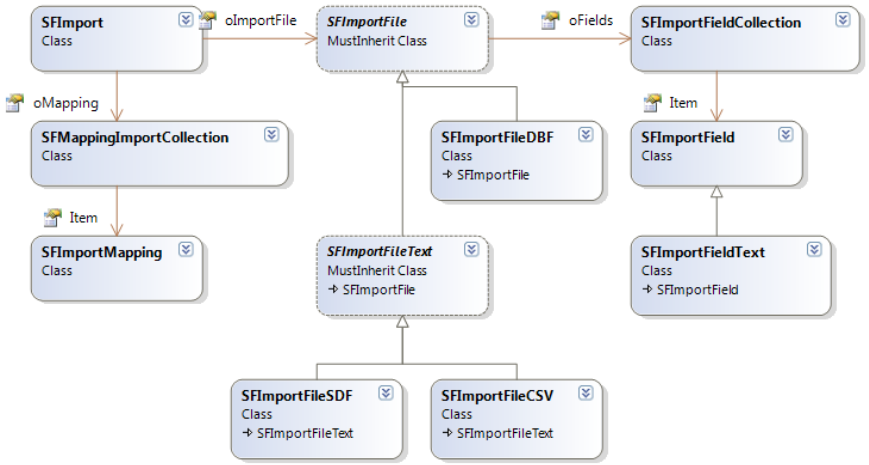

Figure 1. Class diagram for the generic import utility engine classes.

SFImport

SFImport is the main import engine class. Like the other classes in this article, it's defined in SFImport.VCX. It's a subclass of SFCustom, my Custom base class defined in SFCtrls.VCX. It has the following custom public properties:

- cFilter: a filter to use against imported records. This is normally blank but you can set it to the same expression you'd used in a SET FILTER TO command to limit which records are imported.
- cTableName: the name of the table to import records into.
- lCancelOnError: .T. if the import process should stop when an error occurs or a record is rejected.
- lContinue: .T. if we should continue processing. This property could be set to .F. when the user presses Esc or clicks a Cancel button to stop the import.
- nDuplicateRecords: the number of duplicate records encountered.
- nFilteredRecords: the number of records not passing the filter.
- nImportRecords: the total number of records in the import file.
- nInvalidRecords: the number of invalid records encountered.
- nRecordsImported: the number of records imported.
- nRecordsProcessed: the number of import records processed.
- oImportFile: an object reference to an SFImportFile object; we'll look at SFImportFile and its descendents later.
- oMapping: an object reference to an SFImportMappingCollection object.

SFImport doesn't have many public methods:

- Import: performs the import.
- LoadProfile: loads settings from a named profile.
- SaveProfile: saves the current settings to a named profile.
- StatusUpdate: a method provided so SFImport can have a StatusUpdate event.

Performing an import

To perform an import, instantiate SFImport and either set its properties appropriately for the import task or load the settings for a previously saved import, called a profile, using the LoadProfile method.

Profiles are handy when you run the same import process several times; you don't have to write the many lines of code necessary to set up the SFImport object. Profiles are stored in individual tables; the name of the table is the name of the profile.

Listing 1 is an example that shows both techniques. It imports from a fixed-length text file into a table called Test.DBF. The first five characters of each record in the import file are imported into a field named Field2 in Test and the second five characters are imported into Field1.

Listing 1. This example shows how to import a file by either setting up the import object programmatically or using a profile.

```
lparameters tlLoadProfile

* Create a table we'll import records into.

close databases all
create table Test (Field1 C(5), Field2 C(5))

* Create the import object.

loImport = newobject('SFImport', 'SFImport.vcx')
```

```
* If we're loading a profile, do so.

if tlLoadProfile and file('profile.dbf')
  loImport.LoadProfile('profile')

* Set up the import object.

else
  loImport.cTableName = 'test.dbf'

* Create the import file object.

  loImport.oImportFile = newobject('SFImportFileSDF', ;
    'SFImport.vcx')
  loImport.oImportFile.cFileName = 'Import.txt'

* Load the import file definition.

  loField = loImport.oImportFile.oFields.Add()
  loField.nStartPos = 1
  loField.nFieldLen = 5
  loField = loImport.oImportFile.oFields.Add()
  loField.nStartPos = 6
  loField.nFieldLen = 5

* Load the mappings.

  loMap = loImport.oMapping.Add('Field1')
  loMap.uImportField = 2
  loMap = loImport.oMapping.Add('Field2')
  loMap.uImportField = 1

* Save the profile so we can use it next time.

  loImport.SaveProfile('profile')
endif tlLoadProfile ...

* Do the import.

llSuccess = loImport.Import()
if llSuccess
  messagebox(transform(loImport.nRecordsImported) + ;
    ' records of ' + transform(loImport.nImportRecords) + ;
    ' imported')
  select Test
  browse
else
  messagebox(loImport.cErrorMessage)
endif llSuccess
```

This sample starts by creating a table to import into; a real program would likely use an existing table, but I wanted this demo to be self-contained. Next it instantiates an SFImport object. If the tlLoadProfile parameter is .T. and Profile.DBF exists, LoadProfile loads the profile used last time the program was run. Otherwise, the code sets the cTableName property to Test.DBF, since that's the table we'll import into.

SFImport collaborates with an object subclassed from SFImportFile, which we'll look at later. This object, which is instantiated into the oImportFile property, is responsible for managing the import file, including opening and closing the file and reading in records. There are several subclasses available, each of which deals with a different type of import file. The code in Listing 1 uses SFImportFileSDF, which handles fixed-length text files, sometimes referred to as SDF (simple data format) files. The code sets the cFileName property, which specifies the name of the import file, to Import.TXT.

The next block of code defines the structure of the import file. The oFields member of SFImportFile contains a reference to an SFImportFieldCollection object, which is ultimately a subclass of Collection. The Add method instantiates and returns an SFImportField object or one of its subclasses; in the case of SFImportFileSDF, it's actually an SFImportFieldText object. The code then sets properties of each

SFImportFieldText object, defining the starting position and length of each field in each record of the text file.

Next, the code defines a mapping between the fields in the import file and the table being imported into. The oMapping property contains a reference to an SFImportMappingCollection object, another Collection-based class. This code specifies that Field1 in Test.DBF maps to the second field in the import file and Field2 maps to the first one.

The sample program then saves the import settings to a profile called Profile, which you can use the next time you run the program by passing .T. to the program, and calls the Import method to perform the import. If the import process fails for some reason, Import returns .F. and cErrorMessage contains the message of the error.

This program is pretty simple: it doesn't deal with things like duplicate records or data conversion. Let's look at a more complex example.

Handling duplicates and data conversion

The code in Listing 2 uses a subclass of SFImport called TestImport that does duplicate checking. This class overrides the IsDuplicate method, ensuring the record about to be added to the table doesn't already exist by trying to match on the contact name and zip code. If IsDuplicate returns .T., the record isn't imported. We'll see where this method is called from later.

Listing 2. This more complex example handles data conversion and duplicate checking.

```
* Create a table we'll import records into.

close databases all
create table Customers (Contact C(40), Address C(40), ;
  City C(20), State C(2), ZipCode C(10))

* Create the import object and specify the table to import
* into.

loImport = createobject('TestImport')
loImport.cTableName = 'Customers.dbf'

* Create the import file object and specify the table to
* import from.

loImport.oImportFile = newobject('SFImportFileDBF', ;
  'SFImport.vcx')
loImport.oImportFile.cFileName = 'Names.dbf'

* Load the import file definition.

loField = loImport.oImportFile.GetFileStructure()

* Load the mappings.

loMap = loImport.oMapping.Add('Contact')
loMap.uImportField = 'NAME'
loMap.cExpression  = 'proper(VALUE)'
loMap = loImport.oMapping.Add('Address')
loMap.uImportField = 'ADDRS1'
loMap.cExpression  = 'proper(VALUE)'
loMap = loImport.oMapping.Add('City')
loMap.uImportField = 'CITY'
loMap.cExpression  = 'proper(VALUE)'
loMap = loImport.oMapping.Add('State')
loMap.uImportField = 'SOURCE1'
loMap = loImport.oMapping.Add('ZipCode')
loMap.uImportField = 'POSTCODE'

* We only want customers in a certain range of zip codes.

loImport.cFilter = "between(Customers.ZipCode, " + ;
  "'10000', '39999')"
```

```
* Do the import and show the results.

llSuccess = loImport.Import()
if llSuccess
  messagebox('Records processed: ' + ;
      transform(loImport.nRecordsProcessed) + chr(13) + ;
    'Records imported: ' + ;
      transform(loImport.nRecordsImported) + chr(13) + ;
    'Duplicate records: ' + ;
      transform(loImport.nDuplicateRecords) + chr(13) + ;
    'Filtered records: ' + ;
      transform(loImport.nFilteredRecords) + chr(13) + ;
    'Invalid records: ' + ;
      transform(loImport.nInvalidRecords), ;
    0, 'Import Results')
  select Customers
  browse
else
  messagebox(loImport.cErrorMessage)
endif llSuccess

define class TestImport as SFImport of SFImport.vcx
  function IsDuplicate(toRecord)
    local lnSelect, ;
      loRecord, ;
      llDuplicate
    lnSelect = select()
    select (This.cAlias)
    locate for Contact = toRecord.Contact and ;
      ZipCode = toRecord.ZipCode
    llDuplicate = found()
    select (lnSelect)
    return llDuplicate
  endfunc
enddefine
```

This program starts by creating a Customers table and setting the cTableName property of a newly instantiated TestImport object to that table. It then instantiates an SFImportFileDBF object because the file we'll import from, Names.DBF, is another table. Rather than coding the structure of the import file, the code uses the GetFileStructure method of SFImportFileDBF to fill the oFields collection itself. It then sets up the import mappings. Notice in this case, the mappings are set by field name rather than field number, and that in some cases, an expression is specified for data conversion purposes. In these expressions, VALUE is a placeholder for the actual value to convert; in this case, we'll use PROPER() on the values. This code also specifies a filter so only certain records are imported. It then calls the Import method and displays the results, shown in Figure 2. Notice most records aren't imported; they're either filtered out or are duplicates.

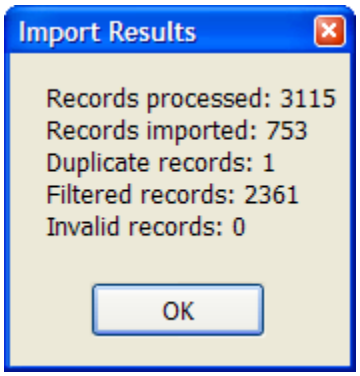

Figure 2. Running the second sample program gives these results.

SFImport: the code

Now that we've seen SFImport in action, let's look at how it's implemented.

Import, shown in Listing 3, is the key method in SFImport. This method is well-commented, so I won't describe it in detail here. Notice the use of the IsDuplicate, ValidateRecord, BeforeSaveRecord, and AfterSaveRecord hook methods. These methods are blank in this class but could be filled in in a subclass to implement whatever behavior you desire.

Listing 3. SFImport.Import is responsible for importing the records from a file into a table.

```
local llReturn, ;
  llAdded, ;
  llSuccess, ;
  loMap, ;
  luImport, ;
  luValue, ;
  lcField, ;
  lcFilter
with This

* Open the table we're importing into.

  do case
    case not .OpenTable()
      llReturn = .F.

* Open the import file.

    case not .oImportFile.OpenFile()
      .cErrorMessage = .oImportFile.cErrorMessage
      llReturn = .F.

* Zero the record counters and set nImportRecords to the
* total number of records in the import file.

    otherwise
      store 0 to .nRecordsImported, .nDuplicateRecords, ;
        .nFilteredRecords, .nInvalidRecords, ;
        .nRecordsProcessed
      .nImportRecords = .oImportFile.nRecords

* Import each record until we hit the end of the import file
* or we're forced to stop.

      llReturn   = .T.
      .lContinue = .T.
      do while not .oImportFile.lEOF and .lContinue

* Create an object to hold the values for the fields in the
* current record. Process each field defined in the mapping
* collection. Get the current mapping definition and the
* value to put into that field from the import file.

        loRecord = createobject('Empty')
        for each loMap in .oMapping
          luImport = loMap.uImportField
          luValue  = .oImportFile.GetField(luImport)
          luValue  = .ConvertData(loMap, luValue)
          lcField  = loMap.cFieldName
          addproperty(loRecord, lcField, luValue)
        next loMap
        lcFilter = strtran(.cFilter, .cAlias + '.', ;
          'loRecord.', -1, -1, 1)
        do case

* If this is a duplicate record, increment the duplicate
* records counter.

          case .IsDuplicate(loRecord)
            .nDuplicateRecords = .nDuplicateRecords + 1
```

```
* If there's a filter and the record doesn't match,
* increment the filtered out records counter.

          case not empty(lcFilter) and not evaluate(lcFilter)
            .nFilteredRecords = .nFilteredRecords + 1
            llSuccess = .T.

* If custom ValidateRecord() fails, increment the invalid
* records counter.

          case not .ValidateRecord(loRecord)
            .nInvalidRecords = .nInvalidRecords + 1

* So far, so good. Call the BeforeSaveRecord() hook method,
* then try to save the record. If we succeeded, call the
* AfterSaveRecord() hook method and increment the number of
* records imported.

          otherwise
            llSuccess = .BeforeSaveRecord(loRecord)
            llSuccess = llSuccess and .SaveRecord(loRecord)
            if llSuccess
              .AfterSaveRecord(loRecord)
              .nRecordsImported = .nRecordsImported + 1
            endif llSuccess
        endcase

* We'll continue processing records if we succeeded or we
* failed but lCancelOnError is .F. Move to the next record.

        .lContinue = (llSuccess or not .lCancelOnError) and ;
          .oImportFile.NextRecord()
        llReturn   = .lContinue

* Increment the number of records processed and raise the
* StatusUpdate event.

        .nRecordsProcessed = .nRecordsProcessed + 1
        raiseevent(This, 'StatusUpdate')
      enddo while not .oImportFile.lEOF ...

* Close the import table.

      .oImportFile.CloseFile()
  endcase
endwith
return llReturn
```

Let's discuss some of the methods called from Import. OpenTable is a relatively simple method. It opens the table specified in cTableName and turns on table buffering so we can control when records are written to the table. ConvertData, shown in Listing 4, applies any expression specified in the cExpression property of the mapping object for the current field to the value read from the input table.

Listing 4. ConvertData performs any data conversion specified in the cExpression property of a mapping object.

```
lparameters toMap, ;
  tuValue
local luValue, ;
  lcExpression, ;
  loException as Exception
luValue = tuValue
if not empty(toMap.cExpression)
  lcExpression = strtran(toMap.cExpression, 'VALUE', ;
    'luValue', -1, -1, 1)
  try
    luValue = evaluate(lcExpression)
```

```
  catch to loException
    This.cErrorMessage = loException.Message
    luValue = .NULL.
  endtry
endif not empty(toMap.cExpression)
return luValue
```

SaveRecord (Listing 5) uses INSERT INTO followed by TABLEUPDATE() to save the record in the table. In a subclass, you could use something like SQLEXEC() if you need to update a SQL Server table instead (in that case, you'd also override OpenTable to open a connection to the SQL Server database and CloseTable to close the connection).

Listing 5. SaveRecord adds the record to the table.

```
lparameters toRecord
local llSuccess, ;
  laErrors[1], ;
  lnError
with This
  insert into (.cAlias) from name toRecord
  llSuccess = tableupdate(1, .F., .cAlias)
  if not llSuccess
    aerror(laErrors)
    lnError = laErrors[1]
    if lnError = cnERR_DUPLKEY
      .nDuplicateRecords = .nDuplicateRecords + 1
    else
      .nInvalidRecords = .nInvalidRecords + 1
    endif lnError = cnERR_DUPLKEY
    tablerevert(.F., .cAlias)
  endif not llSuccess
endwith
return llSuccess
```

SFImportFile

We've briefly seen how SFImport uses an instance of an SFImportFile subclass to deal with the import file. Let's look at this class and its subclasses in more detail.

Like SFImport, SFImportFile is based on SFCustom. As I mentioned earlier, its oFields property contains a reference to a collection of SFImportField objects that describe the structure of the import file. Because some types of files, specifically comma-separated (CSV) or SDF files, need additional properties for fields, the cFieldClass and cFieldLibrary properties of SFImportFile specify the class to use for field objects (currently either SFImportField or SFImportFieldText). cFileName contains the name of the import file and cAlias contains the alias of a table the import file was read into. nRecords contains the number of records in the import file and nRecno contains the current record number.

SFImportFile has methods to open the import file (OpenFile), move to the previous and next records (PreviousRecord and NextRecord), fill the oFields collection with objects describing the structure of the import file (GetFileStructure), and return the value of the specified field in the current record (GetField). These methods are either very simple or empty in this class, so we won't look at the code for this class.

SFImportFile isn't used directly; instead, a subclass specific to a certain type of import file is used. SFImport.VCX contains subclasses for CSV files (SFImportFileCSV), fixed-length text file (SFImportFileSDF), VFP tables (SFImportFileDBF), and those file types using the IMPORT command, such as Excel, Lotus 1-2-3, and so on (SFImportFileOther).

Let's look at SFImportFileCSV to give you an idea of how it works. SFImportFileCSV is actually a subclass of SFImportFileText (SFImportFileSDF is also a subclass of that class). The OpenFile method of SFImportFileText, shown in Listing 6, creates a VFP cursor with the structure necessary to import the text file (as specified in the oFields collection) and then reads the text file into the cursor. Since text files may have one or more header records, the nStartRow property contains the starting row number for data records. CreateCursor is implemented in this class but AppendFromFile is blank because the mechanism to use depends on whether we're importing a CSV or SDF file (SFImportFileCSV uses APPEND FROM … DELIMITED while SFImportFileSDF uses APPEND FROM … SDF).

Listing 6. SFImportFileText.OpenFile imports a text file into a VFP cursor.

```
lparameters tcFilter
local llReturn
with This
  llReturn = .CreateCursor()
  llReturn = llReturn and .AppendFromFile(tcFilter)
  if llReturn
    go top in (.cAlias)
    .lEOF = eof(.cAlias)
    if not .lEOF and .nStartRow > 0
      delete next .nStartRow
      skip
    endif not .lEOF ...
    .nRecords = reccount(.cAlias) - .nStartRow
    .nRecno   = recno(.cAlias) - .nStartRow
    llReturn  = .nRecords > 0
    if not llReturn
      .cErrorMessage = 'No records to import'
    endif not llReturn
  endif llReturn
endwith
return llReturn
```

The GetFileStructure method fills the oFields collection by deducing the structure of the import file. In the case of SFImportFileCSV, the code counts the number of commas in the first row and uses that number plus 1 as the number of fields. All fields are assumed to be C(254) with a name of FIELD<n>, but after calling GetFileStructure to do most of the work, you could adjust the members of the field collection as necessary to match the actual structure. Some of the code for this method (error handling and other support code is omitted) is shown in Listing 7.

Listing 7. SFImportFileCSV.GetFileStructure fills the field collection from the structure of the import file.

```
lnHandle = fopen(.cFileName)
if lnHandle >= 0

* Get the first line and see how many commas there are.

  lcText   = fgets(lnHandle)
  lnFields = occurs(',', lcText) + 1
  .oFields.Clear()
  for lnI = 1 to lnFields
    if .nStartRow > 0
      lnStart = iif(lnI = 1, 1, at(',', lcText, lnI - 1) + 1)
      lnEnd   = iif(lnI = lnFields, len(lcText), ;
        at(',', lcText, lnI) - 1)
      lcField = substr(lcText, lnStart, lnEnd - lnStart + 1)
    else
      lcField = 'FIELD' + padl(lnI, 3, '0')
    endif .nStartRow > 0

* Add the field to the fields collection.

    loField = .oFields.Add(lcField)
    loField.cFieldType = 'C'
    loField.nFieldLen  = 254
  next lnI
  fclose(lnHandle)
* Other code here
endif lnHandle >= 0
```

Summary

The classes in SFImport.VCX provide a generic import utility. However, they require setting up programmatically, which makes them a lot of work to use and prevents your users from being able to configure them. In the next issue, we'll look at an import wizard that provides a front end to these classes, making it easy to add an import function to your applications.

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; 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). Web: www.stonefield.com and www.stonefieldquery.com, Email: dhennig@stonefield.com, Blog: http://doughennig.blogspot.com