November, 2005

Create Test Data Easily

Use these classes and a little code to set up more realistic test data

By Tamar E. Granor, technical editor

A while back, I needed a test database of people and contact information containing more than a handful of records. Since I didn't want all the records to be for John Smith of 1234 N. Main St., nor did I want to create 1,000 or so people and 5,000-10,000 contact items (addresses, phone numbers, etc.) manually, I decided to write code to handle the task for me.

I wanted my data to be at least somewhat realistic. Thanks to the Internet, I was able to work with actual names, zip codes, area codes and so forth. (See the sidebar for guidance on finding this kind of information and getting it into tables.)

My basic approach was to have tables containing raw data from which to generate my actual data. For example, I found a website listing the top girls' names, boys' names and surnames based on census data. I grabbed 200 of each and massaged them into three tables: GirlsNames, BoysName, and LastNames.

Similarly, I found a list of US area codes online and transformed it to a table (AreaCode), so I can use real area codes when generating phone numbers. I found several other such lists online. In some cases, I simply made up the raw data—for example, I have a table called Domains.DBF that contains words to use in creating domain names for email addresses and URLs.

The one item I couldn't find for free online was city, state and zip code (CSZ) information. For my purposes, I didn't need a wide range of locations, so I used the US Postal Service zip-code look-up online to get the full set of zip codes for five cities of varying sizes and put those into a table.

I didn't want all the people to have matching sets of contact information. I wanted some people to have work addresses and others not to, some to have email addresses and others to have none, and so forth. To handle this problem, I assigned each possible piece of information a probability and used VFP's RAND() function to determine whether a particular record had a particular type of data.

By the time I generated the data I needed, I realized that it wouldn't be hard to make the process more generic, so it could be adapted to whatever test data I needed. I ended up with two abstract classes that structure the process; they can be subclassed for particular data sets. I also created subclasses of them to handle much of the information you want to generate for people—it can be used as is or subclassed for employees, students, and so forth.

The Big Picture

Creating a test data set involves two processes, generating the data and storing it in tables. It's quite possible for the same data to be stored in several different ways, so I chose to separate the two processes. (One of the things this decision enables is testing different database designs on the same data.)

To handle the two tasks, I created two abstract classes. MakeDataSet is a template for generating an entire data set and storing the data. MakeRecord is a template for generating a single record; its driver method returns an object with the data for that record stored in properties. Each subclass of MakeDataSet uses a subclass of MakeRecord.

Creating a Data Set

MakeDataSet (found in MakeData.PRG on this month's Professional Resource CD) is fairly simple. It's subclassed from Session (so that it works in a private data session) and has four custom properties:

- cGeneratorClass is the name of the MakeRecord subclass used to create individual records;
- cGeneratorClassLib is the name of the class library containing the MakeRecord subclass;
- nSetSize indicates how many records to create;
- oRecordGenerator holds an object reference to the MakeRecord subclass.

The only built-in methods containing code are Init and Destroy. Init has just two lines:

```
This.oRecordGenerator = NEWOBJECT(This.cGeneratorClass, ;
   This.cGeneratorClassLib)
This.OpenTables()
```

Destroy is even simpler:

```
This.CloseTables()
```

The class has six custom methods, most of which are abstract at this level. Table 1 lists the custom methods.

Table 1. Custom methods—MakeDataSet uses these custom methods to create a set of test data.

Method	**Purpose**
AfterMakeSet	Code to run after all records have been added. Abstract.
CheckLookup	Checks whether a particular value has already been added to a specified table. If not, adds it. Returns the primary key of the record.
CloseTables	Closes tables opened by this class. Abstract.
MakeSet	The main method of this class. Calls on the record generator class to create a set of records and saves them.
OpenTables	Opens tables needed by this class. Abstract.
SaveRecord	Saves a record returned by the record generator into the appropriate tables. Abstract.

Although the MakeSet method is the driver for the whole process, the code is pretty simple:

```
LOCAL nRecord, oRecord

FOR nRecord = 1 TO This.nSetSize
  oRecord = This.oRecordGenerator.GenerateRecord()

  This.SaveRecord( oRecord )
ENDFOR

This.AfterMakeSet()
```

The code in CheckLookup is a little more complex. It receives five parameters: the value to look for, the alias of the table, the index to use for the search, the name of the field in which to put the value if it's not found, and the name of the primary key field to return.

CheckLookup lets you store look-up data as you store the rest of the data, as well as create links to look-up data.

```
PROCEDURE CheckLookup(cValue, cTable, cKey, cField, cPKField)

LOCAL uReturn, cReturnField

IF NOT SEEK(UPPER(cValue), cTable, cKey)
  INSERT INTO (cTable) (&cField) ;
    VALUES (cValue)
ENDIF

cReturnField = cTable + "." + cPKField
uReturn = EVALUATE(cReturnField)

RETURN uReturn
```

CheckLookup can be called from SaveRecord in a subclass.

Creating a Record

MakeRecord provides basic tools that make writing subclass code to generate records easier. It includes methods for choosing random values from a range of number or letters. A number of its methods are abstract at this level.

MakeRecord has three custom properties:

- oData is a collection holding the list of tables (such as the CSZ table) to be opened for generating the record. Once the tables have been opened, the collection also contains the number of records in each of these tables;
- oMethods is a collection of methods to call in order to generate the record;
- oRecord is an object reference to the record being created.

Like MakeDataSet, the only built-in methods containing code are Init and Destroy, but they do a little more work here than in MakeDataSet. Init seeds VFP's random number generator and then calls several methods that do the actual work of setting things up:

```
RAND(-1)

This.oData = CREATEOBJECT("Collection")

This.SetProbabilities()
This.SetMethods()
This.SetData()
This.OpenData()
```

Destroy cleans up:

```
This.CloseData()
This.oRecord = .null.
```

MakeRecord has 11 custom methods, listed in Table 2.

Table 2. Generating records—MakeRecord's custom methods help to generate random data.

Method	**Purpose**
AddData	Adds an item to the oData collection. Pass the name and alias of the table as parameters.
AddMethod	Adds an item to the oMethods collection. Pass the name of the method as a parameter.
CloseData	Closes data tables opened by this class. Uses oData to determine what to close.
GenerateRecord	The driver method for record generation.
GetDataCount	Returns the number of records in a specified data table.
OpenData	Opens data tables used by this class. Uses the information in oData.
RandInt	Returns a random integer between specified values.
RandLetter	Returns a random letter of the alphabet.
SetData	Sets up the list of tables to open. Abstract.
SetMethods	Sets up the list of methods to call to generate the data. Abstract.
SetProbabilities	Sets up the probabilities used to decide what data to generate for a given record. Abstract

SetData is an abstract method to be specified at the subclass level. It's meant for populating the oData collection with the list of tables used for generating random values. For example, for a person, you'd

include the tables of boys' names, girls' names and surnames, as well as the CSZ table and the table of area codes.

AddData is a wrapper for the Add method of the oData collection. It lets you add items to the collection without worrying about its internal structure:

```
PROCEDURE AddData(cTable, cAlias)
LOCAL oDataObject

* Make sure the collection exists
IF VARTYPE(This.oData) <> "O"
  This.oData = CREATEOBJECT("Collection")
ENDIF

* Create the data object
oDataObject = CREATEOBJECT("Empty")
ADDPROPERTY(oDataObject, "Table", m.cTable)
ADDPROPERTY(oDataObject, "Alias", m.cAlias)
ADDPROPERTY(oDataObject, "Count")

* Add the object to the collection,
* using the alias as the key
This.oData.Add(oDataObject, m.cAlias)

RETURN
```

OpenData loops through the oData collection, opening the specified tables. For each table it opens, it stores the number of records in the appropriate member of the oData collection. The code is fairly straightforward:

```
LOCAL oTableInfo, lReturn

lReturn = .T.
FOR EACH oTableInfo IN This.oData
  TRY
    cAlias = oTableInfo.Alias
    USE (oTableInfo.Table) ALIAS (m.cAlias) IN 0
    oTableInfo.Count = RECCOUNT(m.cAlias)

  CATCH
    MESSAGEBOX("Cannot open table: " + oTableInfo.Table)
    lReturn = .F.
  ENDTRY
ENDFOR

RETURN lReturn
```

CloseData loops through the oData collection, closing the tables:

```
LOCAL oTableInfo
```

```
FOR EACH oTableInfo IN This.oData
  cAlias = oTableInfo.Alias
  TRY
    USE IN (m.cAlias)
    This.oData.Remove(oTableInfo)
  CATCH
  ENDTRY
ENDFOR

RETURN
```

Both OpenData and CloseData use TRY-CATCH to avoid errors if tables can't be found. Because this class is a developer tool, the error handling is fairly simple—just a messagebox.

VFP's RAND() function returns values between 0 and 1. (In fact, it never returns exactly 1.) RandInt and RandLetter convert the value returned by RAND() into something a little more useful. RandInt returns an integer between specified bounds:

```
PROCEDURE RandInt (nMin as Integer, ;
                   nMax as Integer) as Integer
* Return a random integer between
* the specified min and max

LOCAL nRand, nResult

IF VARTYPE(nMin)<>"N"
  nMin = 0
ENDIF

IF VARTYPE(nMax)<> "N"
  nMax = 1
ENDIF

nRand = RAND()
nResult = INT((nMax - nMin + 1) * nRand) + nMin

RETURN nResult
```

RandLetter returns a random uppercase letter:

```
PROTECTED PROCEDURE RandLetter
* Return a randomly selected letter of the alphabet

LOCAL nRand, cLetter

nRand = This.RandInt(1, 26)
cLetter = CHR(64 + nRand)

RETURN cLetter
```

Neither of these methods is called by code in MakeRecord; they're provided to be used in code added to subclasses. I'll show examples later in the article.

SetProbabilities and SetMethods are both abstract at this level. In subclasses, SetProbabilities is used to set up probabilities for various attributes. In most cases, corresponding properties are added in the subclass and SetProbabilities gives them appropriate values.

SetMethods is provided to populate the oMethods collection with the list of methods to call in order to generate the actual data. The methods themselves are added at the subclass level, as well.

AddMethod is a wrapper for the oMethods collection's Add method. The code is analogous to that in AddData.

GenerateRecord is the main routine for this class. It loops through the list of methods in the aMethods array, calling each in turn:

```
LOCAL oMethod, cMethod

This.oRecord = CREATEOBJECT("Empty")

FOR EACH oMethod IN This.oMethods
  cMethod = "This." + oMethod.Name
  &cMethod
ENDFOR

RETURN This.oRecord
```

GenerateRecord creates an empty object; it's up to the methods it calls to add appropriate properties to hold the data.

Generating People

A fairly common need is generating people and their addresses, phone numbers, emails, and so forth. So the first subclasses of MakeDataSet and MakeRecord perform this task. I'll look at the MakeRecord subclass first, then show how it's used by the MakeDataSet subclass. Both classes are contained in MakePeople.PRG, which is included on this month's PRD.

The MakeRecord subclass is called MakePerson. It has a number of additional custom properties, each of which controls either the range of data for a particular item or the probability of an item. They're listed in Table 3. The array properties are filled in the SetProbabilities method.

Table 3. Controlling record generation—These custom properties of MakePerson determine the values permitted or the likelihood of a record having a particular data value.

Property	**Purpose**
aAddress[1,2]	The probability that the person has each type of address. Column 1 is the type. Column 2 is the probability.
aEmails[1,2]	The probability that the person has each type of email. Column 1 is the type. Column 2 is the probability.
aPhones[1,3]	The probability that the person has each type of phone number. Column 1 is the type. Column 2 is the location. Column 3 is the probability.
aWeb[1,2]	The probability that the person has each type of web address. Column 1 is the type. Column 2 is the probability.
dOldest	The earliest permitted birth date.
dYoungest	The last permitted birth date.
nDates	The number of days between dOldest and dYoungest.
nDomainWordMax	The maximum number of words to use in creating a domain name.
nHasLetter	The probability that a street address includes a letter after the digits.
nHighHouseDigits	The maximum number of digits in a street address.
nLowHouseDigits	The minimum number of digits in a street address.
nMale	The probability that a record should be male.

To create realistic people and contact data, I used the tables described at the beginning of this article. These provide a group of names, streets, area codes and so forth. They're all listed in the SetData

method, which uses the AddData method to populate the oData collection:

```
PROCEDURE SetData

WITH This
  .AddData("LastNames", "LastNames")
  .AddData("BoysNames", "BoysNames")
  .AddData("GirlsNames", "GirlsNames")
  .AddData("StreetNames", "Streets")
  .AddData("CSZ", "CSZ")
  .AddData("AreaCode", "AreaCode")
  .AddData("Domains", "Domains")
  .AddData("TLDs", "TLDs")
ENDWITH

This.nDates = This.dYoungest - This.dOldest + 1

RETURN

RETURN
```

Although the list of possible birth dates isn't stored in a table, SetData uses the end dates provided to compute the number of birth dates available.

SetProbabilities fills in the likelihood that the person has various types of data. For example, it sets the chance of a home (personal) address to 90%, but there's only a 40% chance of a work (business) address and a 20% change of a school address.

Only a portion of the method is shown here. The rest is analogous, populating the rest of the aPhones array and resizing and populating the aEmails and aWeb arrays.

```
WITH This
  DIMENSION .aAddresses[3,2]
  .aAddresses[1,1] = "Personal"
  .aAddresses[1,2] = .9
  .aAddresses[2,1] = "Business"
  .aAddresses[2,2] = .4
  .aAddresses[3,1] = "School"
  .aAddresses[3,2] = .2

  DIMENSION .aPhones[8,3]
  .aPhones[1,1] = "Personal"
  .aPhones[1,2] = "Voice"
  .aPhones[1,3] = .9
  .aPhones[2,1] = "Personal"
  .aPhones[2,2] = "Fax"
  .aPhones[2,3] = .3
```

SetMethods lists the methods to be called in the order in which they should be called, calling AddMethod to populate the oMethods collection:

```
WITH This
  .AddMethod("GetName")
  .AddMethod("GetBirthdate")
  .AddMethod("GetAddresses")
  .AddMethod("GetPhones")
  .AddMethod("GetEmails")
  .AddMethod("GetURLs")
  .AddMethod("GetSSN")
ENDWITH

RETURN
```

The real work is done in all the Getxxx methods listed in SetMethods. Each one creates one type of data. GetBirthdate is the simplest, but demonstrates most of the basic ideas:

```
LOCAL nRand

nRand = This.RandInt(1, This.nDates)
ADDPROPERTY(This.oRecord, "dBirthdate", ;
            This.dOldest + nRand - 1)

RETURN
```

RandInt returns a number between 1 and the number of days specified. The second line adds a property called dBirthdate to the record and sets its value to the specified date (the day nRand-1 days after the starting date).

GetName generates a first name and last name and also sets the record's gender. It uses the BoysNames, GirlsNames and LastNames tables. The method calls RandInt to return a number between 1 and the number of surnames. It uses that value as a record number and grabs the surname at that position. Next, it generates a random number and checks it against the probability that the person is male. Depending on the result of that check, either a boy's name or a girl's name is chosen, using the same approach as for the surname. cFirst and cLast properties are added and set to the names chosen. In addition, a cGender property is added and set to either "M" or "F".

```
LOCAL nRec, nRand

* Choose a last name
nRec = This.RandInt(1, This.GetDataCount("LastNames"))
GO nRec IN LastNames
ADDPROPERTY(This.oRecord, "cLast", ;
```

```
            ALLTRIM(LastNames.cName))

* Determine male or female and get first name
nRand = RAND()
IF nRand <= This.nMale
  nRec = This.RandInt(1, This.GetDataCount("BoysNames"))
  GO nRec IN BoysNames
  ADDPROPERTY(This.oRecord, "cFirst", ;
              ALLTRIM(BoysNames.cName))
  ADDPROPERTY(This.oRecord, "cGender", "M")
ELSE
  nRec = This.RandInt(1, This.GetDataCount("GirlsNames"))
  GO nRec IN GirlsNames
  ADDPROPERTY(This.oRecord, "cFirst", ;
              ALLTRIM(GirlsNames.cName))
  ADDPROPERTY(This.oRecord, "cGender", "F")
ENDIF

RETURN
```

Because each person can have multiple addresses, phone numbers, email addresses and websites, the methods that generate that information all work similarly. Each first adds a property to the person record pointing to an empty collection. Then it loops through the corresponding probability array, and for each item, uses RAND() to determine whether this person should have an item of the specified type. If so, the method creates an empty object to hold the new item. Then, it uses appropriate techniques (calls to RandInt andRandLetter, calls to RAND(), look-ups in the right tables) to create the data for that item and add properties to the new object to hold the data. Finally, it adds the newly created object to the collection. GetAddresses is typical:

```
LOCAL nAddr, nRand, oAddress
LOCAL nHouseNumber, cHouseLetter, nHigh, nLow

ADDPROPERTY(This.oRecord, "oAddresses", ;
            CREATEOBJECT("Collection"))

FOR nAddr = 1 TO ALEN(This.aAddresses, 1)
  nRand = RAND()
  IF nRand <= This.aAddresses[ m.nAddr, 2]
    * Generate this one
    oAddress = CREATEOBJECT("Empty")
    ADDPROPERTY(oAddress,"cType", ;
                This.aAddresses[m.nAddr, 1])

    * Get a house number. First, figure out how
    * many digits, then choose a random value with
    * that many digits. This approach is used
    * because choosing randomly over the whole range
    * results in too many longer values.
```

```
    nRand = This.RandInt(This.nLowHouseDigits, ;
                         This.nHighHouseDigits)
    nLow = 10^(nRand-1)
    nHigh = 10^nRand - 1
    nHouseNumber = This.RandInt(m.nLow, m.nHigh)
    * Check whether to add a letter
    nRand = RAND()
    IF nRand <= This.nHasLetter
      cHouseLetter = This.RandLetter()
    ELSE
      cHouseLetter = ""
    ENDIF
    cHouseNumber = TRANSFORM(m.nHouseNumber) + ;
                   m.cHouseLetter

    * Get a street
    nRand = This.RandInt(1, This.GetDataCount("Streets"))
    GO nRand IN Streets
    cStreet = Streets.cDir -(" " + Streets.cStreet) - ;
              (" " + Streets.cType)

    * Get a city, state, zip combination
    nRand = This.RandInt(1, This.GetDataCount("CSZ"))
    GO nRand IN CSZ

    ADDPROPERTY(oAddress,"Street", m.cHouseNumber + ;
                " " + ALLTRIM(m.cStreet))
    ADDPROPERTY(oAddress,"City", CSZ.cCity)
    ADDPROPERTY(oAddress,"State", CSZ.cState)
    ADDPROPERTY(oAddress,"Zip", CSZ.cZip)

    * Now add the new address to the collection
    This.oRecord.oAddresses.Add(m.oAddress)
  ENDIF
ENDFOR

RETURN
```

MakePerson also includes GetPhones, GetEmails and GetURLs. Email addresses and URLs have two components in common, the domain name and the top-level domain (COM, EDU, ORG, etc.). So the class includes GetDomainName and GetTLD methods, which generate those randomly.

The final method in MakePerson is GetSSN, used to generate a social security number at random. The code follows the basic rules for the structure of a US social security number (which I found on the web). It also demonstrates the approach to use for items that should be unique in the data set, but can't be specified as AutoIncrement fields. GetSSN maintains a cursor of the social security numbers generated so far. The code is set up so that the calling object (a subclass of MakeDataSet) could create that cursor before calling on MakePerson;

doing so allows MakePerson to add data to an existing test set, rather than only create new test sets. Here's the code for GetSSN:

```
LOCAL cSSN, nDigit1, nDigit2, nDigit3, nLast, lNewNum

IF NOT USED("__SSNs")
  CREATE CURSOR __SSNs (cSSN C(9))
  INDEX on cSSN TAG cSSN
ENDIF

lNewNum = .F.
DO WHILE NOT lNewNum
  * First set of three: 001 to 772
  nDigit1 = This.RandInt(0, 7) && First digit not above 7
  IF m.nDigit1 = 7
    nDigit2 = This.RandInt(0, 7)
    IF m.nDigit2 = 7
      nDigit3 = This.RandInt(0, 2)
    ELSE
      nDigit3 = This.RandInt(0, 9)
    ENDIF
  ELSE
    nDigit2 = This.RandInt(0, 9)
    IF m.nDigit1 = 0 AND nDigit2 = 0
      nDigit3 = This.RandInt(1, 9)
    ELSE
      nDigit3 = This.RandInt(0, 9)
    ENDIF
  ENDIF

  cSSN = TRANSFORM(m.nDigit1) + ;
         TRANSFORM(m.nDigit2) + TRANSFORM(m.nDigit3)

  * Second set of two: 01 to 99
  nMiddle= This.RandInt(1, 99)
  cSSN = m.cSSN + PADL(m.nMiddle,2,"0")

  * Third set of four: 0001 to 9999
  nLast = This.RandInt(1, 9999)
  cSSN = m.cSSN + PADL(m.nLast, 4, "0")

  * Is it unique?
  IF NOT SEEK(m.cSSN, "__SSNs", "cSSN")
    lNewNum = .T.
    INSERT INTO __SSNs VALUES (m.cSSN)
  ENDIF
ENDDO

ADDPROPERTY(This.oRecord, "cSSN", m.cSSN)

RETURN
```

To generate additional data items, create the appropriate Getxxx routine and add the method call to the aMethods array.

Generating a Set of People

To create a set of people, I subclassed MakeDataSet and set nSetSize to 5000, cGeneratorClass to "MakePerson" and cGeneratorClassLib to "MakePeople.PRG". I had to put code in only two methods, OpenTables and SaveRecord.

For OpenTables, I chose to take the "open or create" approach. That is, for each table, the method checks whether it already exists. If so, it opens the table. If not, the method creates the table with the desired structure.

Depending on your needs, you might choose to always create new tables or to always open existing tables. While testing my code, I used a version of OpenTables that created cursors, so that they'd disappear when I was done. In some cases, you might choose to clone all the tables from an existing database—that could provide an easy way to set up a test data set for an application.

Here's a portion of the code in OpenTables. Note that if the Person table already exists, the code creates the cursor of social security numbers and fills it with existing values to ensure the new values are unique.

```
IF FILE("Person")
  USE Person IN 0
  * Grab SS#'s already in use
  SELECT cSSN FROM Person INTO CURSOR __SSNs READWRITE
  INDEX on cSSN TAG cSSN
ELSE
  CREATE TABLE Person (iID I AUTOINC UNIQUE, ;
      cFirst C(15), cLast C(30), cGender C(1), ;
      cSSN C(9), dBirth D)
ENDIF

IF FILE("Address")
  USE Address IN 0
ELSE
  CREATE TABLE Address (iID I AUTOINC UNIQUE, ;
      iPersonFK I, iLocFK I, cStreet c(60), ;
      cCity C(20), cState C(2), cZip C(9))
ENDIF
```

SaveRecord is the most interesting method in this subclass. In this method, you can take the generated data and store it in whatever form meets your needs. The database that got me started on this code was designed specifically to test a new approach to storing contact information; it puts all contact items into a single table, and maintains

a pair of look-up tables to indicate the item type and location. The version included with this article uses a more traditional approach, with separate Address, Phone, Email and Web tables. It also creates a look-up table for location values ("Business", "Personal", "School", etc.) and uses the CheckLookup method to handle those values.

```
PROCEDURE SaveRecord(oRecord)

LOCAL iPerson, iLoc

WITH oRecord
  INSERT INTO Person (cFirst, cLast, cGender, ;
      cSSN, dBirth) ;
    VALUES (.cFirst, .cLast, .cGender, ;
      .cSSN, .dBirthdate)
  iPerson = Person.iID

  FOR EACH oAddress IN .oAddresses
    WITH oAddress
      iLoc = This.CheckLookup(.cType, "Location", ;
        "cLocation", "cLocation")
      INSERT INTO Address (iPersonFK, iLocFK, cStreet, ;
         cCity, cState, cZip) ;
        VALUES (m.iPerson, m.iLoc, .Street, .City, ;
                .State, .Zip)
    ENDWITH
  ENDFOR

  FOR EACH oPhone IN .oPhones
    WITH oPhone
      iLoc = This.CheckLookup(.cLoc, "Location", ;
        "cLocation", "cLocation")
      INSERT INTO Phone (iPersonFK, iLocFK, ;
          cType, cNumber) ;
        VALUES (m.iPerson, m.iLoc, .cType, ;
          ALLTRIM(.AreaCode) + ALLTRIM(.Number))
    ENDWITH
  ENDFOR

  FOR EACH oEmail IN .oEmails
    WITH oEmail
      iLoc = This.CheckLookup(.cType, "Location", ;
        "cLocation", "cLocation")
      INSERT INTO Email (iPersonFK, iLocFK, mEmail) ;
        VALUES (m.iPerson, m.iLoc, .Email)
    ENDWITH
  ENDFOR

  FOR EACH oURL IN .oWeb
    WITH oURL
      iLoc = This.CheckLookup(.cType, "Location", ;
        "cLocation", "cLocation")
      INSERT INTO URL (iPersonFK, iLocFK, mURL) ;
        VALUES (m.iPerson, m.iLoc, .URL)
```

```
    ENDWITH
  ENDFOR

ENDWITH

RETURN
```

By changing the code in OpenTables and SaveRecord, you could even store the same data into two different sets of tables, which would enable you to check which structure works better for a particular application.

Taking it farther

MakePerson and MakePersonSet handle the basic information you want in a person record. I created two subclasses of each. MakeStudent and MakeStudentSet (in MakeStudent.PRG on this month's PRD) simply extend the person structure to include a unique 8-digit student number and limit the birthdates to a reasonable range for students. In addition, MakeStudentSet saves the data in cursors, demonstrating the way to test this code without leaving traces.

MakeEmployee and MakeEmployeeSet (in MakeEmp.PRG on this month's PRD) create a single table containing a multi-level hierarchy. They generate data for a table called Emp with these fields:

- iID – primary key
- cFirst – first name
- cLast – last name
- iSuper – primary key of employee's supervisor

MakeEmployee is very simple. It sets nMale to .5, indicating that approximately half the employees should be male and half female. SetMethods indicates that GetName is the only method to call.

In MakeEmployeeSet, OpenTables and SaveRecord are essentially cut-down versions of their counterparts in MakePerson. The really interesting code in this class is in AfterMakeSet.

Because the iSuper field for each record has to be drawn from the set of primary keys, that field can't be filled in until the entire set has been created. The class has two additional properties, nMinSubordinates and nMaxSubordinates, that determine the minimum and maximum number of direct subordinates for each supervisor.

AfterMakeSet collects all the primary keys and chooses one at random to be the top-level boss. That person's ID is stored in a cursor call AddEmps.

The major processing loop goes through AddEmps. For each record, it calls RandInt to choose the number of direct subordinates for this employee and then loops through, choosing employees at random to be this employee's subordinates. Each one is added to AddEmps. The loop keeps track of the number of employees processed and stops when it runs out of people to assign.

```
LOCAL nCount, nRec, nProcessed, nBoss, nNumEmps, nEmpID
LOCAL nEmp, lGotOne, nCount, nHoldRec

* Set up a cursor with all the record numbers
SELECT RECNO() AS nRecNO, iID, .F. AS lUsed ;
  FROM Employee ;
  INTO CURSOR EmpRecs READWRITE

SELECT EmpRecs
INDEX ON nRecNo TAG nRecNo
nCount = RECCOUNT("Employee")
nProcessed = 0
nBoss = 0

* Set up a cursor to handle employees yet to process
CREATE CURSOR AddEmps (iID I)

* Choose the boss
nRec = This.oRecordGenerator.RandInt(1, nCount)
SEEK nRec IN EmpRecs
REPLACE lUsed WITH .T. IN EmpRecs
INSERT INTO AddEmps VALUES (EmpRecs.iID)
nProcessed = 1

DO WHILE nProcessed<nCount AND NOT EOF("AddEmps")
  * Process the current record in AddEmps
  nBoss = AddEmps.iID
  nHoldRec = RECNO("AddEmps")

  * Find out how many employees for this boss
  nNumEmps = This.oRecordGenerator.RandInt( ;
    This.nMinSubordinates, This.nMaxSubordinates)
  nNumEmps = MIN(nNumEmps, nCount-nProcessed)

  FOR nEmp = 1 TO nNumEmps
    * Choose an unused record
    lGotOne = .F.
    DO WHILE NOT lGotOne
      nRec = This.oRecordGenerator.RandInt(1, nCount)
      SEEK nRec IN EmpRecs
      IF NOT EmpRecs.lUsed
        lGotOne = .T.
```

```
        REPLACE lUsed WITH .T. IN EmpRecs
        nEmpId = EmpRecs.iID
      ENDIF
    ENDDO

    * Process it
    SEEK nEmpID ORDER iID IN Employee
    REPLACE iSuper WITH nBoss IN Employee
    INSERT INTO AddEmps VALUES (nEmpID)
    nProcessed = nProcessed + 1
  ENDFOR

  GO (nHoldRec) IN AddEmps
  SKIP
ENDDO

RETURN
```

Putting it all together

To use any of the MakeDataSet subclasses, simply instantiate it, set nSetSize, and call the MakeSet method. For example:

```
oMakeSet = NewObject("MakeEmployeeSet", "MakeEmp.PRG")
oMakeSet.nSetSize = 1000
oMakeSet.MakeSet()
```

When it's done, you'll have a test set that's ready to go. I've found that with this code available, I'm far more likely to create proper test data instead of testing on just a few records with ridiculous values.

The Professional Resource CD for this issue includes all of the classes described here, as well as my tables of data for names, addresses, phone numbers, emails and URLs.

Sidebar: Creating Raw Data Tables

Populating the raw data files took a little work. To find them, I used Google to find sites offering the information I wanted in any format that would be easy to work with.

The male names, female names and surnames came from the US census website. They have long lists of names from the 1990 census at http://www.census.gov/genealogy/names/names_files.html. I highlighted the first 200 in each list and copied them to a text file. Then I used VFP's text processing functions to turn each text file (which contained additional data) into a table with only the names listed.

I found a list of street names for Anchorage, Alaska available in a downloadable Excel workbook at http://webapps1.muni.org/pdpw/addressing/StreetResults.cfm. Converting the spreadsheet to a table was a piece of cake and the nearly 5000 entries seemed to be plenty for my purposes.

As noted in the main article, the CSZ table, containing city, state and zip code combinations, has only a little data in it. (Complete city, state, zip data is available from a number of companies, but I couldn't find it for free.) I collected my data by using the US Postal Service's zip code lookup website. One of the options is to look up zip codes by city (http://zip4.usps.com/zip4/citytown_zip.jsp). I used that choice for five cities. For each, I highlighted the results, saved them to a text file and then used VFP code to put the data I wanted into a table.

For area codes, I found a variety of information online. I downloaded an Access database containing the list of area codes in use and extracted the data I needed from it. I can no longer find the original source, but http://www.nanpa.com/area_codes/ offers a similar (though more complex) table. A search of Google for "area codes" turns up several sites with area code information in text form.

Ten years ago, creating these tables would have been a labor-intensive process. With the Internet plus a little VFP code, I was able to put these tables together fairly quickly.