The Why and How of Test Data

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com

A realistic test data set provides a variety of advantages to developers, testers and end users, yet most applications don't include one. In this session, we'll look at the reasons for supplying test data and examine ways to generate it. Since this session was first presented in 2007, the world of test data generation has changed quite a bit; the session will reflect those changes.

Introduction

Testing is a key element of application development. Yet, far too often, developers test their work against a toy data set or against live data. While a small and unrealistic data set lets you find obvious bugs, it doesn't stress test the application and may miss subtle bugs. The problems of testing against live data include damaging the data and violating privacy policies.

The best solution is to have a test data set that is realistic in both size and content. This provides a safe and robust test environment. Some developers even provide test data to their customers; customers can use the test data for training and practice and to try to reproduce problems without damaging their real data.

This paper explores the reasons for creating test data, examines what constitutes good test data, and looks at ways to create test data.

What is test data?

Since the term "test data" may have different meanings for different people, let's start with some definitions. The International Software Testing Qualifications Board Glossary (https://glossary.istqb.org/en_US/home) defines it as:

Data needed for test execution.

K2View (https://www.k2view.com/what-is-test-data/) defines it this way:

Test data is a compliant dataset used by development and QA teams to ensure that software applications perform as expected while maximizing test coverage.

In fact, there are at least two kinds of test data. One kind is a set of realistic data meant to provide a test bed for an application. It includes a substantial number of records ordered in a way that reflects the normal operation of the application.

The second kind of test data (hinted at by the K2View definition) is data designed to test specific aspects of an application. This data might include tests for boundary conditions, application limits and so forth. This kind of test data is often used in conjunction with automated testing or a test-driven approach to development.

This session is focused more on the first kind of test data. While the testing process needs to ensure that invalid data can't get into the database and that the application handles unusual situations properly, a test database is assumed to be data that the application has already validated and accepted.

Why is test data important?

Many developers test their applications by entering a few records manually and then checking that those records can be properly edited, processed and reported on. I tend to start with "John Smith" of "1234 N. Main St." I've seen Mickey Mouse, George Washington, and all kinds of other names in sample data. I've also encountered applications where bug fixes and updates are tested against a copy of the actual application data (or even, occasionally, against the live data itself).

What's wrong with these approaches? To turn the question around, what does a test data set offer that other techniques don't?

Test without damaging live data

This is the most obvious benefit of test data. Obviously, testing with live data is incredibly risky. Taking the chance of damaging a company's mission-critical data is foolhardy and unprofessional.

This benefit also makes the argument for providing users with a test data set. If the people using an application can easily switch between live data and a robust test set, they can try application features in a low-risk environment, replicate errors without further damage to their live data, and train new employees safely.

Test without privacy loss

Many applications deal with personal data, such as social security numbers, health information, salaries, and so forth. It's unlikely that privacy policies permit releasing that data to the company's software developers. Even if it's not specifically prohibited, the fewer people with access to sensitive data, the better. Over the last couple of decades, there has been plenty of coverage of personal data being leaked to the world. A test data set avoids this issue.

Test many situations and unusual cases

A good test data set provides a range of good data that tests the limits and boundaries of the application. For example, if a character field can hold up to 35 characters, having test data that's only 1 character as well as test data that fills all 35 characters lets you see whether forms and reports handle short as well as long values. While "Jane Smith" may look great in a report, "Maximilian Alexander-MacDougall" may turn up some formatting issues.

Having null values where they're permitted tests the null-handling features of your application. Processing large numeric values tests whether fields holding summary results are wide enough.

The more acceptable cases you include in your test data, the more chance of turning up subtle errors while testing.

Test in a known state

It's a good idea to not just include test data, but to keep two copies, one to work on and one clean copy so you can restore your test data to a known state. Then, you can use automated testing tools with known results to do regression testing (that is, ensuring changes don't break anything).

If you provide your users with two sets of test data, you can have them test against known data to ensure they see the same results you do.

Replicate bugs more easily

If everyone working with your application has access to the same test data set, then one person can replicate a bug and provide steps to the person who needs to fix it. If your users have access to the test data set, when they encounter a problem, they can try the same process with the test data and see if they can replicate the problem. If they can, testing and fixing the bug is a lot easier than if you need to have their exact data in the state it was in when the problem occurred.

Stress test

When you create test data manually, you're likely to include just a few records, nothing like the volume actually expected by the application. Some developers create a handful of records and then duplicate them to provide realistic volume, but that results in an unrealistic data set.

With a suitably large realistic data set, you can test that your application performs appropriately under the expected load. Doing this kind of testing yourself will keep you from having to say "But it wasn't slow in my office."

What does good test data look like?

By now, I hope you're convinced that creating a test data set is a good idea. But what should test data look like? Test data should be both realistic and extreme. Test data should also avoid pitfalls.

Good test data is realistic

Realistic test data reflects the application in both values and order of the data. Make sure that the values you include are like the actual data users will enter. For example, if an application must deal with customers around the world, don't include only North American addresses and phone numbers.

If some data may occur more than once, make sure you include some duplicate values in the test data. For example, an application for a bookstore or library has to handle several different books with the same title.

The order of data matters, too. It's common to create test data in sorted order. If new records will be entered in order in the application, that's fine. For example, sales orders will probably be naturally sorted by order number. But if records will actually be entered randomly (like customer records), make sure they're not ordered alphabetically in your test data set.

Good test data includes extreme values

In production, applications run into extreme values, missing values and more. Make sure your test data includes records that reflect this. If an invoice can include anything from 1 to 100 line items, make sure your test data has some of each and some in between.[1]

Create a realistic volume of test data. If users will have 10,000 customer records at the end of a year, don't test on 100. More accurately, don't do all your testing on 100.

Although a test data set is important, make sure you test your application with no data as well. What will happen if the user tries to enter an invoice before entering any customers?

Make sure your test data addresses idiosyncrasies (or potential idiosyncrasies) of the data. Include text values with apostrophes, double quotes and parentheses, if these can occur in real data. Include all acceptable characters.

Good test data avoids pitfalls

There are a few things to avoid when creating test data. The first is obvious from the previous discussion. Don't use a small set of data, repeated many times. As noted earlier, repeated data is a good idea, if it reflects the reality of the application. But setting up 10 customers and replicating them 100 times to create 1000 customer records is a poor choice.

Don't use sample values that make customers question your seriousness about their application. While it's easy to enter Mickey Mouse and Donald Duck while you're testing a piece of code, putting them in a test data set that a user will see may lead your client to suspect you consider their application a toy.

Much more importantly, stay away from foul or inappropriate language in test data (and, of course, in messages the user will see). Even if you think your test data will never be seen outside your office, it's better to stay away from anything that might offend customers. A staff member for one well-known VFP product created a whole series of "interesting"

[1] In fact, I recently fixed a bug in a client's vertical market application for auto repair shops that showed up only when a service order had 100 or more lines in the invoice.

names; the company was very embarrassed when these records were accidentally included with the sample data for one version of the product.

When you're dealing with records that represent people, make your data set diverse, as well. Do so both because a diverse set of names is more likely to test the full range of acceptable characters and lengths, and because customers using the test data set will notice if people like them aren't represented.

Good test data gets updated

I've been using VFP's Northwind database as the basis for demos since it first came along. But increasingly, using it makes my demos look dated, because the data in Northwind is from the last 1990s.

On the other hand, one of my clients maintains a test database for his vertical market application; he has created a mechanism that allows him to add data every month, so that the data remains fresh. Among other things, that makes it possible to test changes in things like year-end routines.

Good test data is easily accessible

Once you have a solid test data set, you want to make it easy for you to work with it, and if you're giving it to customers, for them to do so as well. That might take the form of multiple desktop shortcuts for your application, or a facility in the application to switch data sets.

One client uses an INI file to list all available data sets, with a way of specifying which one is the default. In addition, the application itself contains a utility for switching between the available data sets.

Where does test data come from?

You have three basic choices for creating a test data set: copy or convert existing data, buy test data, or create test data. The right answer depends on the situation.

When updating or replacing an existing application, copying or converting data makes a lot of sense, especially because it probably has to be done at some point anyway. For new applications, that's not generally an option.

The second choice is to use a product that generates test data. I'll look at some of these below.

Finally, you can create test data either manually (which I don't recommend) or by writing code to create it for you. I'll show a set of VFP classes I designed to generate test data and a VFPX project that lets you generate test data.

In this session, I'll focus only on products that let you generate test data for Visual FoxPro databases. If you're looking for test data for SQL back-ends, there are a lot more choices.

The simplified Northwind database

To test the various alternatives, I created a simplified version of the Northwind database that comes with VFP. I eliminated several tables that weren't populated in the supplied data, as well as the Category table. I also eliminated all the views and RI (referential integrity) code from the original. I also added an Integer primary key to OrderDetails, and removed the tag based on a compound key. The result is 7 tables with a number of connections among them; the database structure is shown in **Figure 1**.

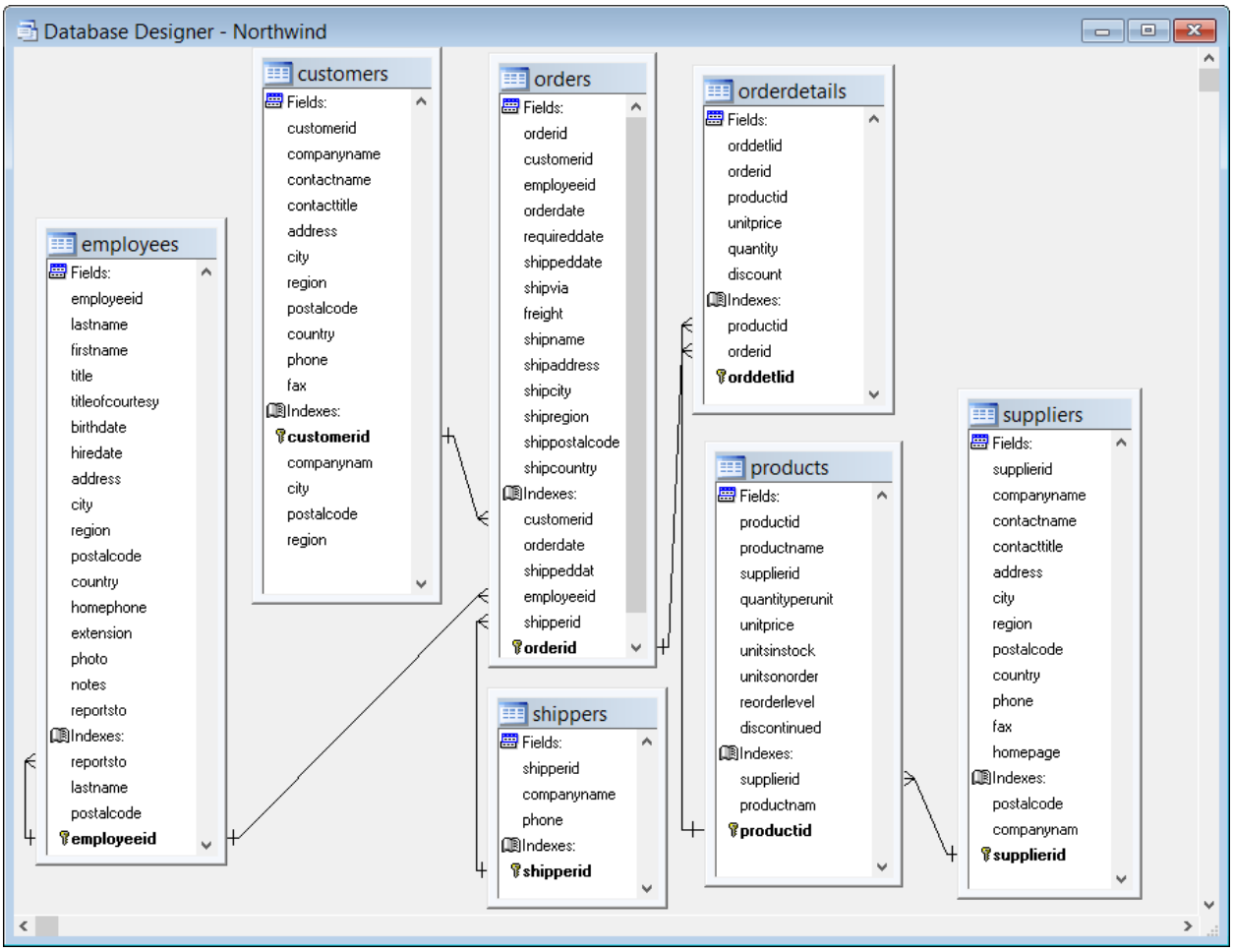

Figure 1. This database is a simplified version of the Northwind database that comes with VFP. We'll use it to test the various ways of generating test data.

This database includes many of the data items you'll typically want to generate, including names, addresses and phone numbers, as well as foreign keys that need to link to actual data in other tables. Code to generate the database is included in the downloads for this session as NorthwindSimplified.PRG.

The downloads for this session have a separate folder for each product I tested containing any relevant code. In addition, each of those folders contains a Northwind folder with the

data generated using that product. There's also a Northwind folder containing an empty set of the tables for the simplified Northwind database.

Creating test data from existing data

If your application is an update to or replacement for an existing application, a substantial data set should already exist. For minor updates, you can probably test using a copy of this data. For more significant updates or for replacement applications, the data structures are likely to change, but you can write code to convert the existing data to the new format.

There are both pros and cons to taking this approach to creating test data. On the plus side, you certainly get a realistic data set; what could be more realistic than the customer's actual data? In addition, creating your test data from existing data forces you to consider the data conversion process early on and gives you lots of opportunities to test that process.

The negatives need to be considered as well, though. First, conversion duplicates any problems in the existing data. If bad data, such as orphaned records or duplicated primary keys, has crept in over time, your test data set will include those problems. While your new code might prevent them from occurring, you'll need a way to deal with them in existing data. (Of course, dealing with this in creating your test data is a good thing, since it likely will lead to dealing with such problems for the users, as well.)

Converting existing data may also fail to pick up unusual cases. After all, what makes them unusual is that they don't occur very often. You may need to enhance the existing data with some records to test extreme cases.

Perhaps the biggest issue is that converting existing data exposes items that should be kept private, such as social security numbers, health information, salaries and so forth. However, there are products available that allow you to deal with these issues by "scrambling" sensitive data. You can also write some code to obfuscate the private data while keeping the rest.

Overall, for updates and replacements, using existing data as the basis for test data is probably the best choice.

Buying test data

A number of companies offer test data generator products. Some of them are actually open-source, so "buying" isn't really the right verb here, though of course, learning to use whatever product you choose will cost time.

When I first created this session, I found a small number of commercial products, and tested several of them. (I know I limited my tests to products that offered free trials and to those that could connect to databases via ODCB or OLE DB, but beyond that, I don't remember what criteria I used.)

Today, there are far more such products, some existing online only and others that require installation. As before, I will limit my tests to those with free trials.

It appears that test data generators can be divided into three broad categories: open-source products where you clone/download and install, and then write scripts to get what you need; online products where you do everything on a website; and commercial products where you download and install, and work through a user interface. To my amazement, two of the three commercial products I evaluated in 2007 still exist today.

The open-source tools I turned up generally seemed focused on specific development environments, and looked like they would require significant set-up. Since one of the points of this session is to make this easy, I decided not to test any of them.

The test data generator products vary widely in price, ability and interface. What follows is an overview of a few of them that make test versions freely available. In each case, the product has additional, more advanced capabilities. The simplified Northwind database generated by each is included in the session materials.

Online test data generators

I tried several of the online test data generators. To use these, you work in a browser and the configuration you set up is stored in an online account.

I didn't get far with most of them. In a couple of cases, they could only handle a single table at a time, which meant there was no way to set up foreign keys. Another looked far more promising, but the set of data types available was too limited; among other things, it had no was to generate postal codes.

The only online test data generator I looked at that I could imagine actually using was Mockaroo, which turned out to be powerful and flexible.

Mockaroo

Mockaroo (https://www.mockaroo.com/) is an online test data generator that's free for up to 1000 records per table. If you need more than that, Mockaroo offers several price levels. For $60 a year, you can have up to 100,000 records per table. $500 per year buys you 10 million records per table. Finally, the Enterprise level at $7500 per year is unlimited and allows you to host Mockaroo on your own server or private cloud. According to the website, the paid levels generate data 8 times faster than the free service.

Mockaroo has an API that allows you to call it from web applications. The free service is limited to 200 calls per day, while the paid versions have increasing access. I didn't test the API because it's not relevant for the task of generating data for VFP.

In order to save anything in Mockaroo, you have to create an account. All that's required for a free account is an email address and a password. Unless you sign up for a paid account, Mockaroo collects no private data.

The website is oriented toward working with a single table (called a *schema*) at a time. However, you can group schemas into *projects*. Projects can also contain datasets (data you've already generated), as well as *APIs* and *scenarios*; I didn't test APIs and scenarios. **Figure 2** shows the Mockaroo project I created for the simplified Northwind database. For each table, there's a schema and a dataset.

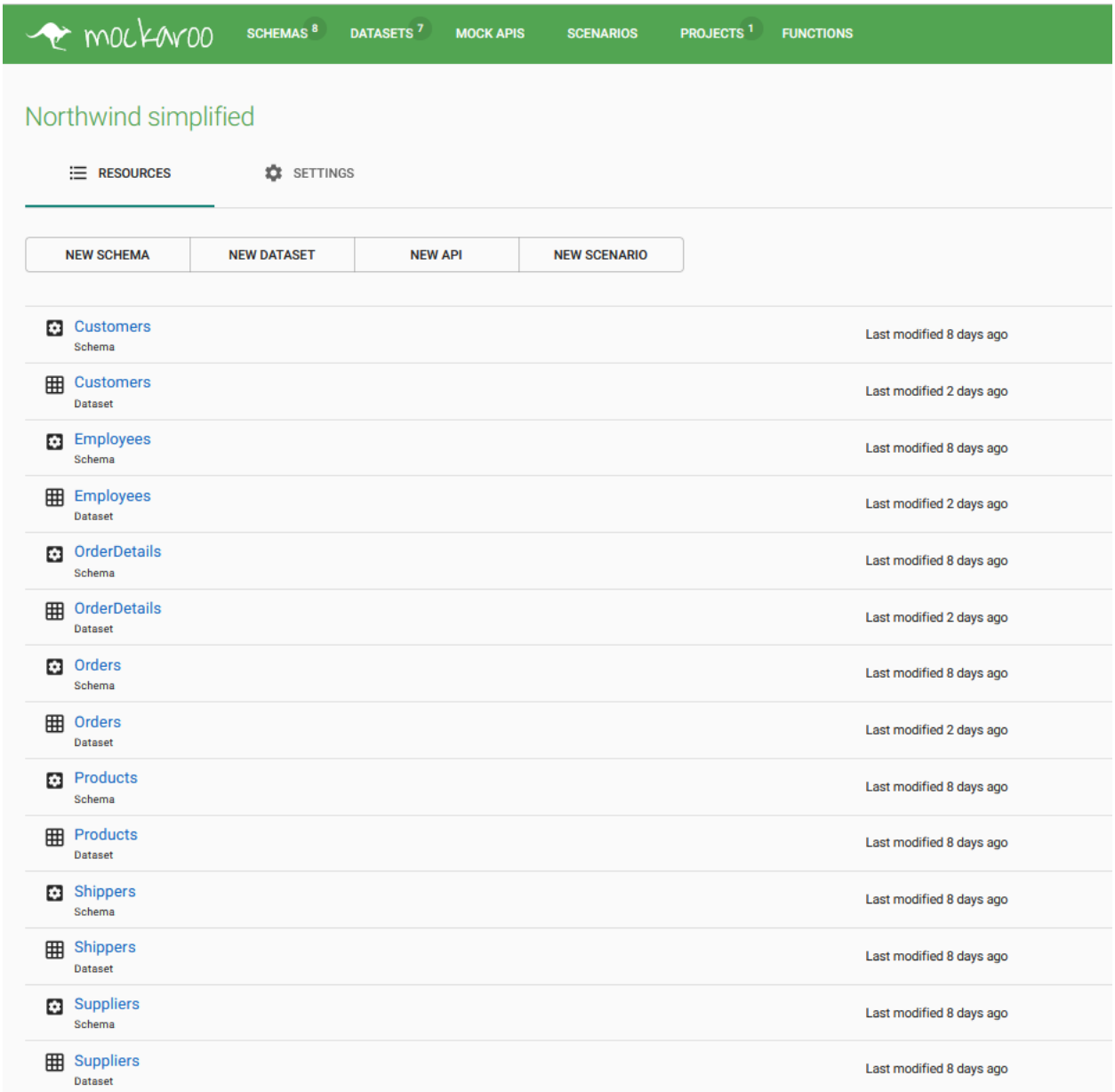

Figure 2. For my simplified version of Northwind, I created this project in Mockaroo.

A schema comprises a list of fields and instructions to generate data for each. Mockaroo provides nearly 200 different kinds of data, along with ways to fine-tune many of them. (Mockaroo calls these *types*.) **Figure 3** shows the schema I set up for the Northwind Customers table. Although every field in the VFP table is character, you can see that I've chosen many different kinds of data here.

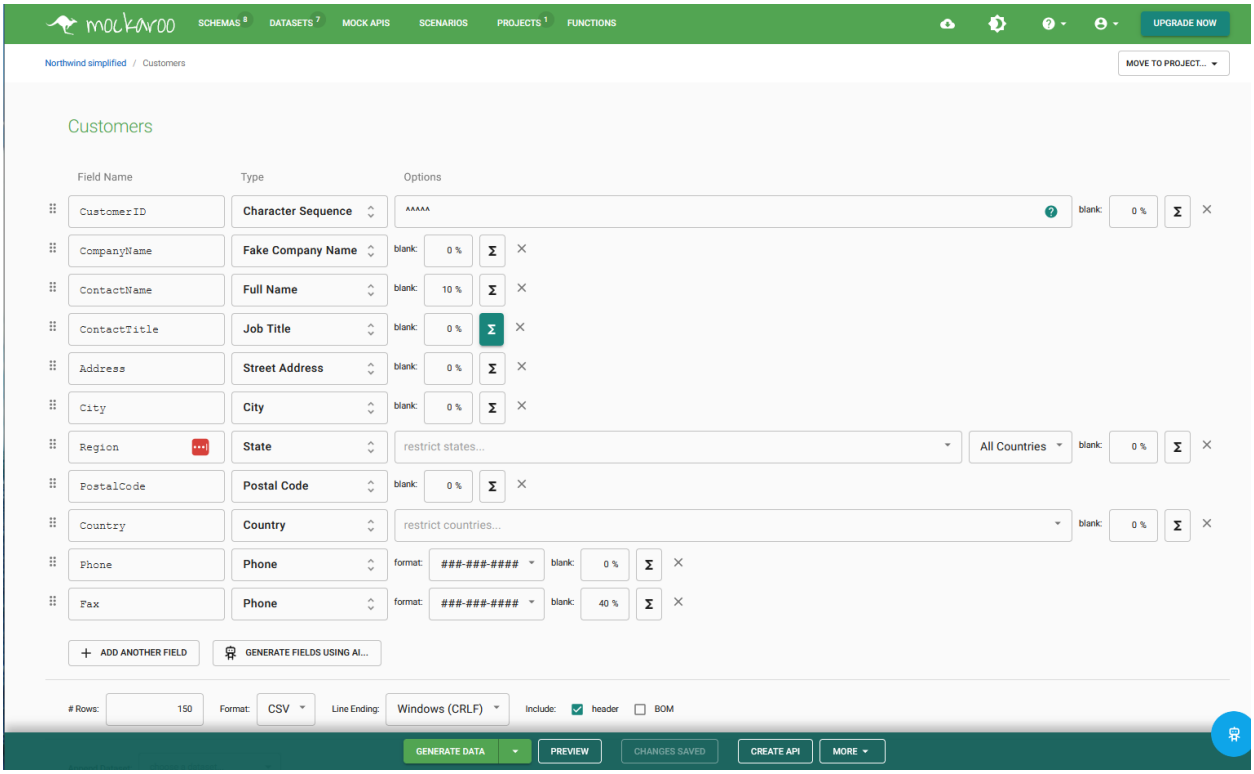

Figure 3. For each schema (table), you set up a list of fields and the instructions for generating each one.

The dialog shown in **Figure 4** appears when you click into the Type column for a field. You can use the categories on the left or start typing into the search box at the top to narrow the list down. For example, when I type “nu” (without the quotes) into the search box, I see only the 5 types shown in **Figure 5**, while clicking on the Location category shows the types in **Figure 6**.

The City, State, Country, and Postal Code types in the Location category are linked, so that if you use more than one of them in a schema, the result will make sense. However, the Phone type you can see in Figure 3 isn’t linked to those location types. You can, however, choose the format in which phone numbers are generated. (I chose the typical North American format and omitted the country code.)

Mockaroo includes some interesting and useful types. For example, it has both real company names and fake company names. There are sets of car makes and models, and even VINs. Types also include GUIDs and ISBNs and colors. Overall, it wasn’t hard to find an appropriate type for most of the Northwind data.

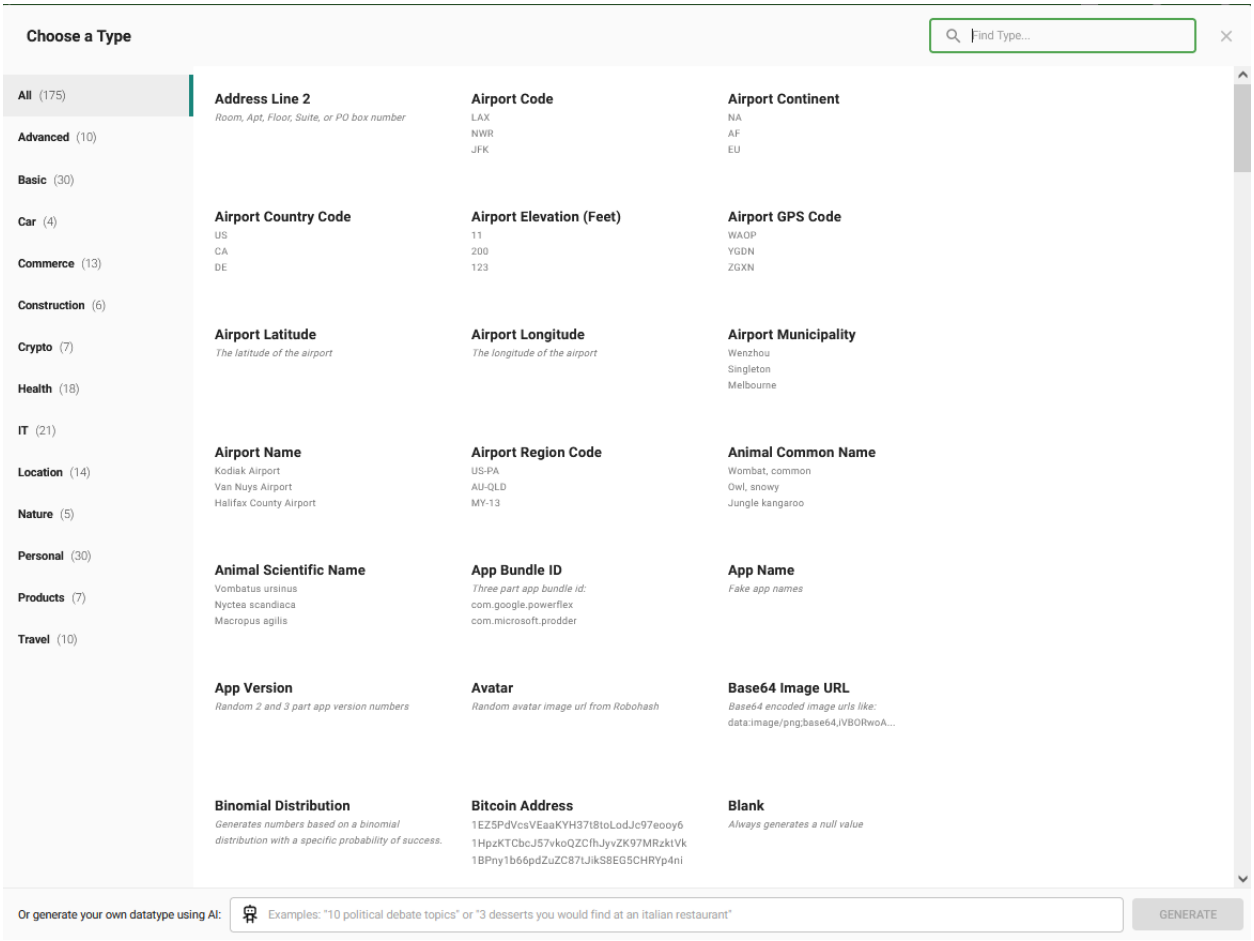

Figure 4. When you click into the Type column for a field, this dialog opens to choose the kind of data to generate for the field.

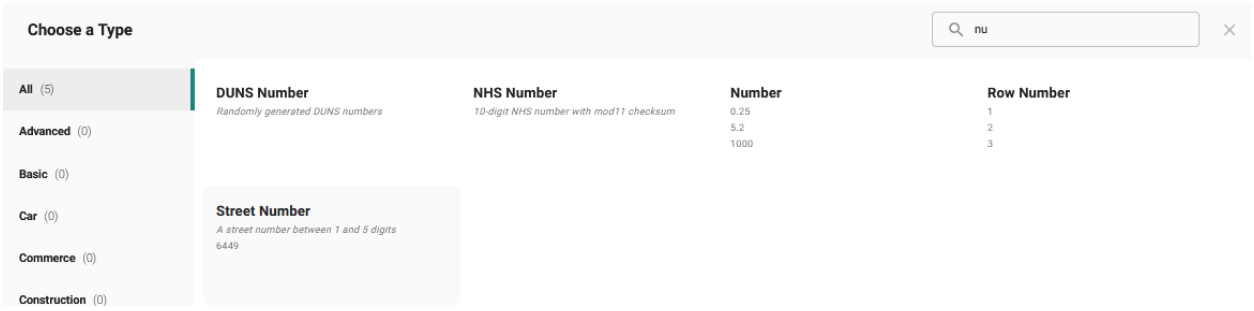

Figure 5. These are the data types that appear when you filter the list on "nu."

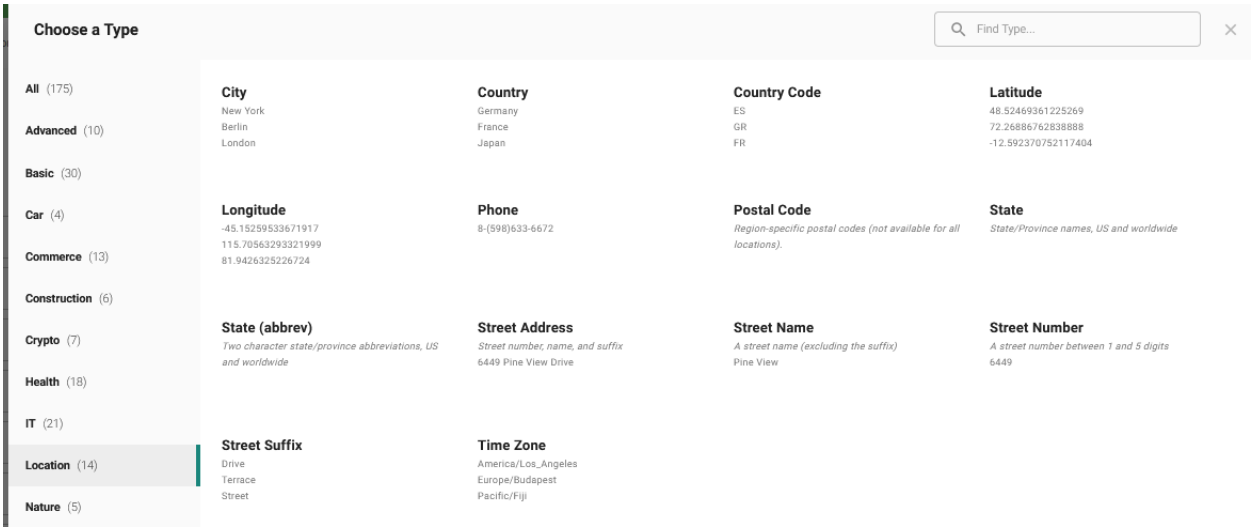

Figure 6. The Location category contains a variety of data related to locations, including a couple of ways to generate addresses.

Depending on the type you choose, you may need to supply additional information. For the 5-character sequence I specified for the CustomerID column (which in the original Northwind database is an abbreviation of the actual company name), I specified a template using a set of characters which I could see by clicking the question mark at the end of the Options column for that field. **Figure 7** shows the panel that pops up on the left; very cleverly, you can leave it open as you work, so you don't have to memorize the choices. (Looking at this, it's clear that the Phone type is a special case of a character sequence.)

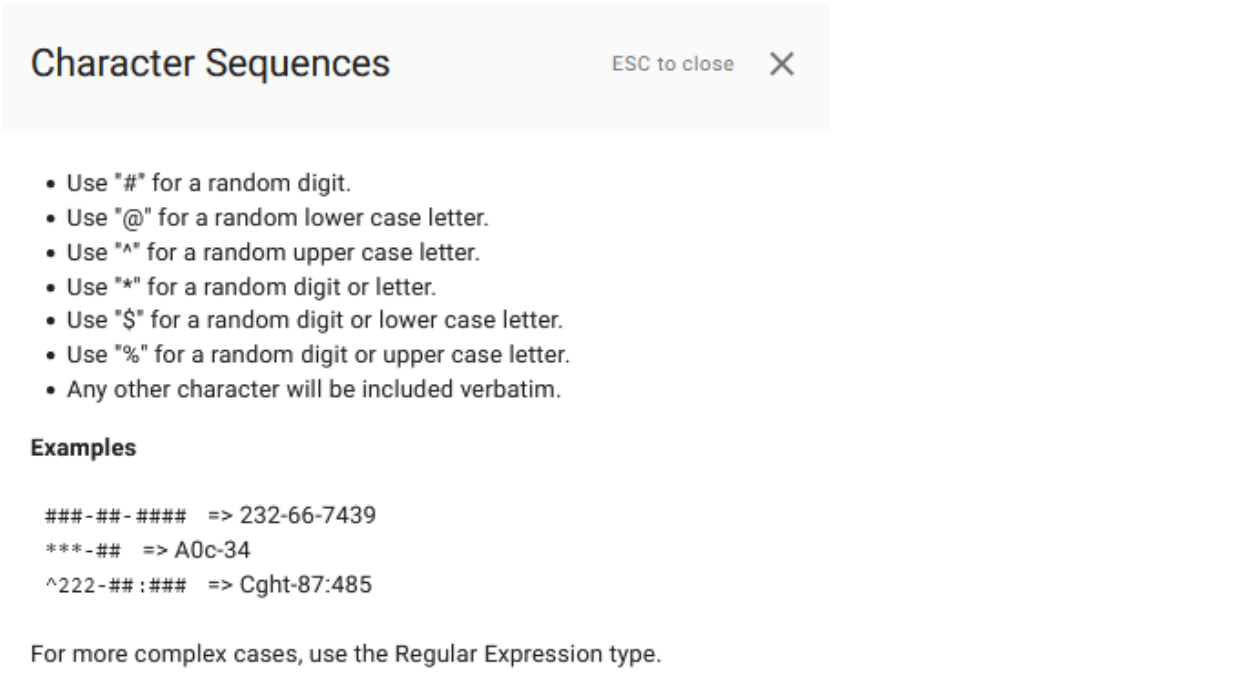

Figure 7. You provide a template for the Character Sequence type using these characters.

Mockaroo also lets you write code to populate a field. Code uses Ruby syntax and functions, but despite never having worked with Ruby, the guidance provided allowed me to write some simple bits. To write code for a field, you click the sigma (sum operator) for the relevant field. Once you've added code, the sigma appears with a green background.

For example, I specified that 10% of the customer records should have no contact. (You can see that in the third row in **Figure 3**.) When there was no contact, I didn't want to specify a ContactTitle. **Figure 8** shows the Formula editor and the code I wrote for this case.

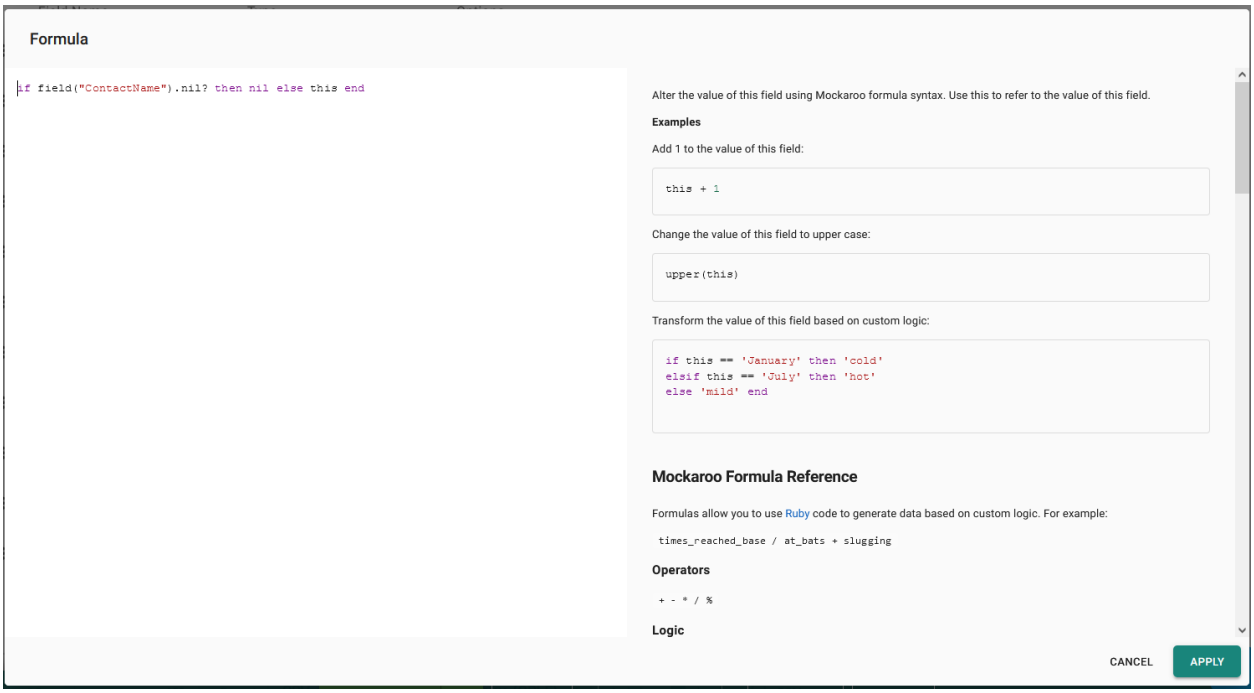

Figure 8. The code here says that if the ContactName field is empty ("nil"), then this field should be empty, too. Otherwise, use whatever was generated ("this").

It turns out that Ruby expects lower-case field names. Since I'd used mixed-case, I had to put my field names inside the field() function for Mockaroo to understand them.

You can specify fields meant only for helping generate other fields by preceding their names with an underscore (that is, with names like "_fieldname"). I didn't test this capability, but I can how it would be useful.

The biggest challenge I faced was dealing with primary keys and foreign keys. Most of the Northwind tables use auto-incrementing integer primary keys. But if I left those fields blank (which is an option), I wouldn't have been able to fill the foreign key fields that depended on those.

So I chose to let Mockaroo generate those fields (using the Row Number data type), knowing that when I imported the data, I'd need to turn off autoincrements for those fields, and then turn it back on afterward, plugging the appropriate value in using the NEXTVALUE clause of ALTER TABLE. (I'll show code for all this a little later in this section.)

As I noted in "The simplified Northwind database" earlier in this paper, I changed the primary key of the OrderDetails table to an auto-generated integer rather than using the combination of OrderID and ProductID. That was, in part, because I couldn't see how to ensure that the combination of the two fields would be unique across all the generated records. (In fact, I think generated surrogate keys are a better practice than meaningful compound keys, so this wasn't a loss, in my view.)

In order to actually generate foreign keys, I used Mockaroo's ability to generate and keep datasets, and to specify that a field should contain data from a dataset. You can choose the output format Mockaroo produces (using the Format dropdown shown the bottom of **Figure 3**), but if you want to create a dataset, you have to choose either CSV or JSON. Since VFP can import CSV fairly easily, I chose CSV. (This raised another issue I'll discuss later in this section.)

To generate a database, click the arrow adjacent to the Generate Data button at the bottom of the page, shown in **Figure 9**. Then click Create Dataset. Once you've created a given dataset once, the item changes to Update Dataset. After some time (that depends on how many records you're creating), you'll see a little of the generated data (as in **Figure 10**). You can also download the generated file by clicking the filename on that page.

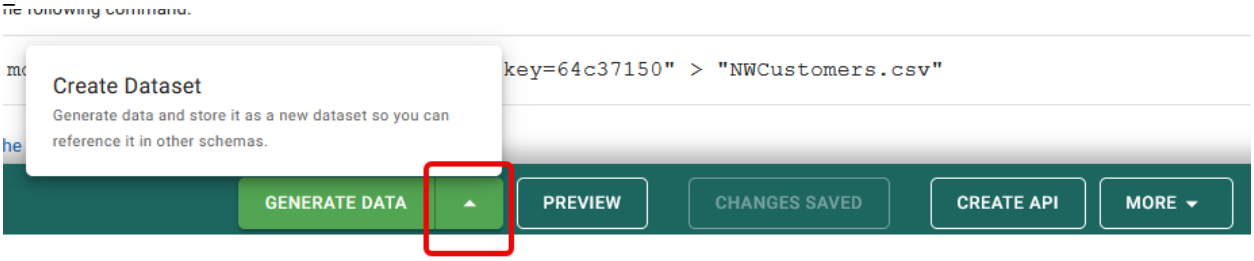

Figure 9. To generate a dataset that you can use to supply foreign keys to other tables, click the Generate Data arrow and then click Create Dataset.

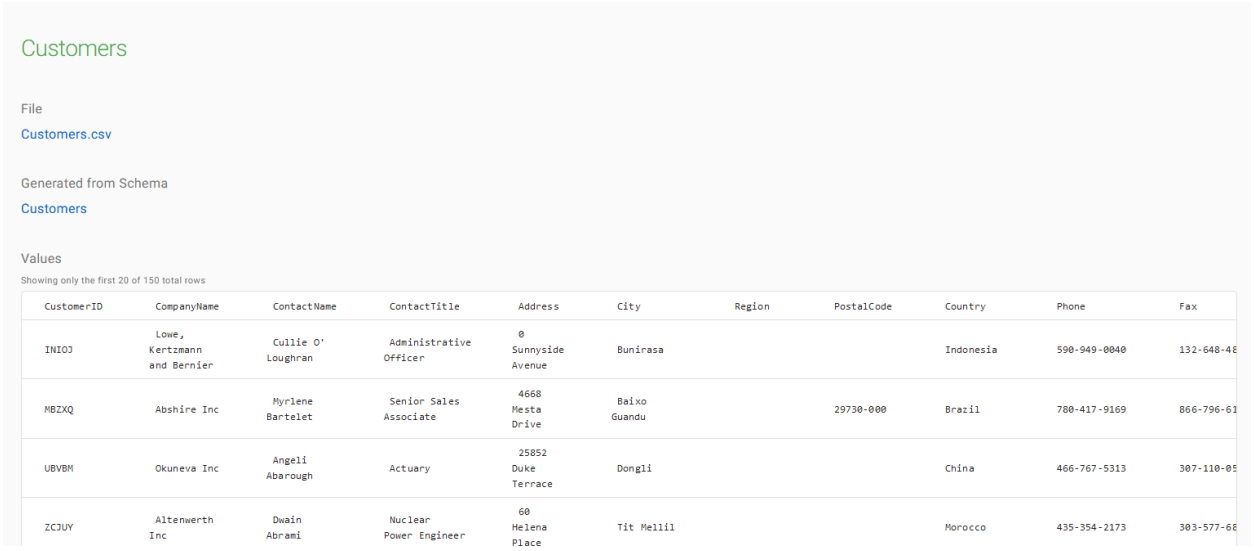

Figure 10. Once you've generated data, you can see a subset of it in Mockaroo.

To use a field from one dataset in another table, specify Dataset Column as the Type and then choose the table and column you want. **Figure 11** shows the schema for the Orders table (with a little bit cut off on the right), including foreign key fields for the CustomerID, EmployeeID, and ShipVia fields.

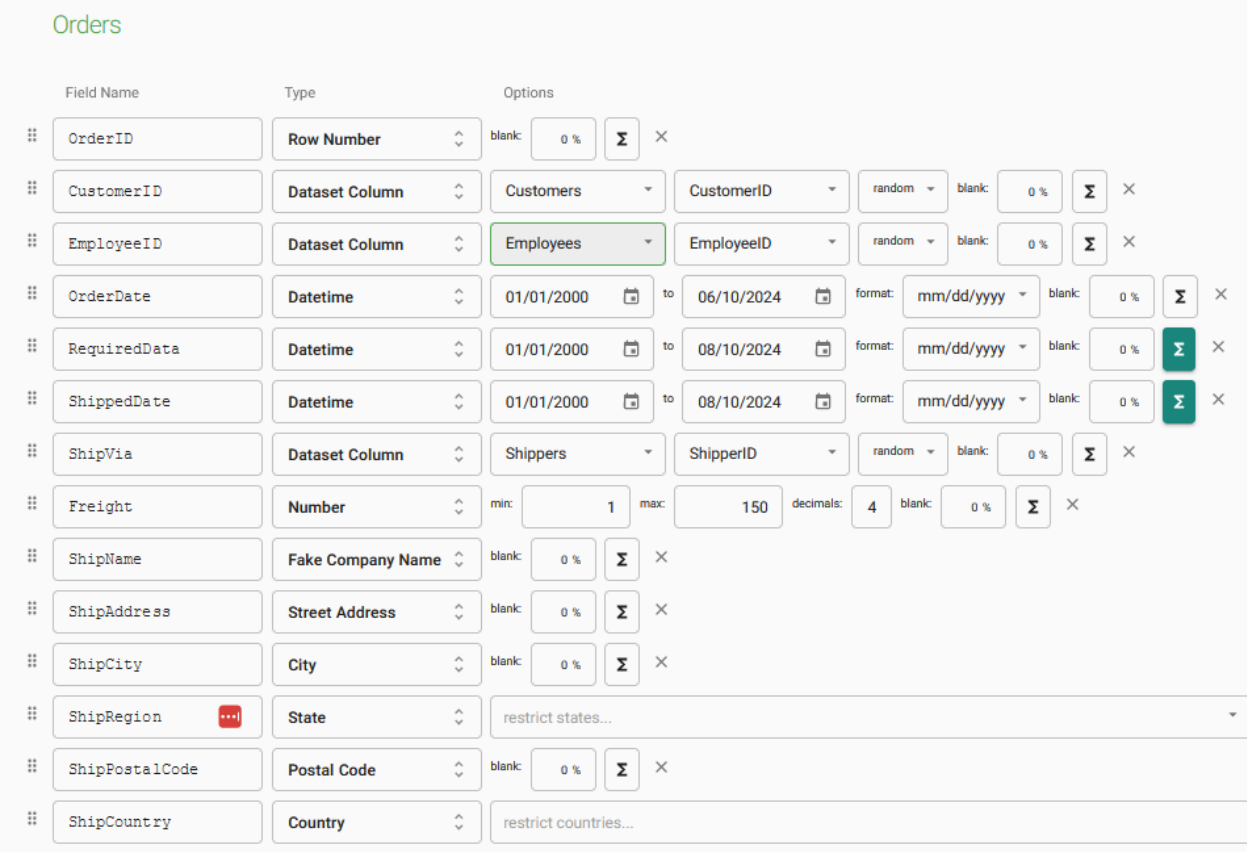

Figure 11. The Orders table has foreign keys to the Customers, Employees, and Shippers tables, each specified using the Dataset Column type.

Of course, in order to generate data for a table that includes Dataset Columns from other tables, you need to have created datasets for those other tables first. Similarly, if you modify the schema for a table and generate a new dataset for it, you also need to go back and generate new datasets for any tables that use columns from the first table.

To get the Mockaroo data into my simplified Northwind database, I downloaded all the datasets, and I wrote a little code to do the actual import. I needed the code to do several things: modify all the auto-incrementing Integer fields to be regular Integer fields, populate each table from the corresponding CSV file, and then modify the Integer fields back to auto-increment, specifying the next available value.

The Employees table has a Notes field, which is defined as Memo. Mockaroo has a Paragraphs data type that supplies random text. (I limited it to a single paragraph to simplify the import.) APPEND FROM can't handle memo fields, so I needed extra code to collect that data from the CSV file and populate the Notes field. **Listing 1** shows the import

code; it's included in the Mockaroo folder of the downloads for this session as ImportMockaroo.PRG. The generated CSV files are in the same folder; the file for each table has the same filestem as the table.

Listing 1. This code imports the CSV files downloaded from Mockaroo into the simplified Northwind database.

```
* Import Mockaroo data

OPEN DATABASE Northwind/Northwind

* Step 1: Turn off AutoInc keys and remove PKs
ALTER TABLE Employees ALTER COLUMN EmployeeID I NOT NULL
SELECT Employees
DELETE TAG EmployeeID
ALTER TABLE Orders ALTER COLUMN OrderID I NOT NULL
SELECT Orders
DELETE TAG OrderID
ALTER TABLE OrderDetails ALTER COLUMN OrdDetlID I NOT NULL
SELECT OrderDetails
DELETE TAG OrdDetlID
ALTER TABLE Products ALTER COLUMN ProductID I NOT NULL
SELECT Products
DELETE TAG ProductID
ALTER TABLE Shippers ALTER COLUMN ShipperID I NOT NULL
SELECT Shippers
DELETE TAG ShipperID
ALTER TABLE Suppliers ALTER COLUMN SupplierID I NOT NULL
SELECT Suppliers
DELETE TAG SupplierID

* Step 2: Import data in appropriate order and grab last ID
CREATE CURSOR csrLastIDs (cTable C(12), cPK C(10), nLastID I)
LOCAL lnLastID

SELECT 0
USE Customers

APPEND FROM Customers.csv TYPE csv

SELECT Employees
APPEND FROM Employees.csv ;
   FIELDS EmployeeID, LastName, FirstName, Title, ;
          BirthDate, HireDate, ;
          Address, City, Region, PostalCode, ;
          Country, HomePhone, Extension ;
   TYPE csv

LOCAL lcEmps, laEmpLines[1], lnLines, lcNotes, lnStartPos
lcEmps = FILETOSTR("Employees.CSV")
lnLines = ALINES(laEmpLines, m.lcEmps)
GO TOP IN Employees

* Skip first line because it's headings
```

```
FOR lnLine = 2 TO m.lnLines
   lnStartPos = AT(',', laEmpLines[m.lnLine], 14) + 1
   lcNotes = SUBSTR(laEmpLines[m.lnLine], m.lnStartPos)
   REPLACE Notes WITH m.lcNotes IN Employees
   SKIP 1 IN Employees
ENDFOR

CALCULATE MAX(EmployeeID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('Employees', 'EmployeeID', m.lnLastID)

SELECT Suppliers
APPEND FROM Suppliers.csv TYPE csv
CALCULATE MAX(SupplierID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('Suppliers', 'SupplierID', m.lnLastID)

SELECT Shippers
APPEND FROM Shippers.csv TYPE csv
CALCULATE MAX(ShipperID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('Shippers', 'ShipperID', m.lnLastID)

SELECT Products
APPEND FROM Products.csv TYPE csv
CALCULATE MAX(ProductID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('Products', 'ProductID', m.lnLastID)

SELECT Orders
APPEND FROM Orders.csv TYPE csv
CALCULATE MAX(OrderID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('Orders', 'OrderID', m.lnLastID)

SELECT OrderDetails
APPEND FROM OrderDetails.csv TYPE csv
CALCULATE MAX(OrdDetlID) TO m.lnLastID
INSERT INTO csrLastIDs VALUES ('OrderDetails', 'OrdDetlID', m.lnLastID)

* Step 3: Turn autoinc back on
LOCAL lcTable, lcPK
SELECT csrLastIDs
SCAN
   lcTable = ALLTRIM(csrLastIDs.cTable)
   lcPK = ALLTRIM(csrLastIDs.cPK)
   lnLastID = csrLastIDs.nLastID

   ALTER TABLE (m.lcTable) ALTER COLUMN (m.lcPK) I AUTOINC NEXTVALUE m.lnLastID + 1
NOT NULL PRIMARY KEY
ENDSCAN
```

You can save Mockaroo schemas to (JSON) files. The Mockaroo schemas I created for the simplified Northwind database are included in the Mockaroo\Schemas folder of the downloads for this session.

There were a few things I wanted to do that I either couldn't do or couldn't figure out.

In the original Northwind database, the primary key for Customers is a unique five-character field based on the company name, but tweaked to guarantee uniqueness. I was able to grab the first five characters of the generated CompanyName field (though it meant putting the CustomerID field after CompanyName in the schema), but I couldn't figure out how to ensure that it was unique in the dataset. Ultimately, I generated a random five-character string instead.

In the original Northwind data, the Products table's QuantityPerUnit field contains strings like "48 - 6 oz jars" and "16 kg pkg." and so forth. I didn't find a way to generate such strings, but I suspect it's possible using custom datasets of measurements and packages, along with some code. (In fact, I didn't generate data for this field with any of the products I tested.)

I also didn't figure out how to specify a distribution of values other than random, as I wanted for the Products table's Discontinued field. I would have liked to set it to make, say, 10% true and the rest false. I suspect there's a way to do this.

Similarly, I wanted to indicate that for most Orders records, the shipping fields should come from the related Customers record, but that some percentage of them should be newly generated. This is another case I suspect is possible.

Mockaroo provides a few tutorials videos. There are also community forums where Mockaroo users can help each other.

Overall, Mockaroo was quite useable, with a lot of flexibility. The biggest weakness from my perspective was the need to generate one table at a time. For a large database, that could be quite tedious.

Installed products

Two of the three commercial products I wrote about in 2007 were still available in 2024. A major advantage of these products is that you can connect them directly to your database structure, which simplifies the specification of the test data, and makes it easier to handle primary and foreign keys.

When I began testing, I found that one of the two products, DTM Data Generator, crashed when I tried to connect it to the simplified Northwind database using an OLE DB connection. I reached out to the company, but they hadn't provided a solution by the time I completed this paper. So, I wasn't able to test that product.

Advanced Data Generator

Advanced Data Generator (ADG) from Upscene Productions (www.upscene.com) can work with any ODBC or ADO data source. That version is €259 and includes updates for one year; additional maintenance can be purchased after the first year, but the price isn't shown on the website. Upscene also offers less expensive versions that work with only a single database; each costs €119. Currently, they have such versions for InterBase,

Firebird, MySQL, and Access. (All prices are as of July, 2024.) A free trial version is available at their website. The trial version is time-limited, and you are limited to no more than 10,000 rows per table.

ADG is organized into databases and projects. A database is any ODBC or ADO data source. You register it with ADG and it appears in a list of databases. There's a wizard for registering a database; the wizard includes a link to the Windows applets that let you create new datasources.

The functionality of ADG hasn't changed a lot since I tested it in 2007, but the UI has changed and some of the terminology has, as well.

Getting started with ADG

You can work with VFP databases through either ODBC or ADO, but if the VFP database includes any features added after VFP 6 (such as auto-increments or blob fields), you have to use ADO. **Figure 12** shows the main workspace in ADG, with the list of registered databases shown.

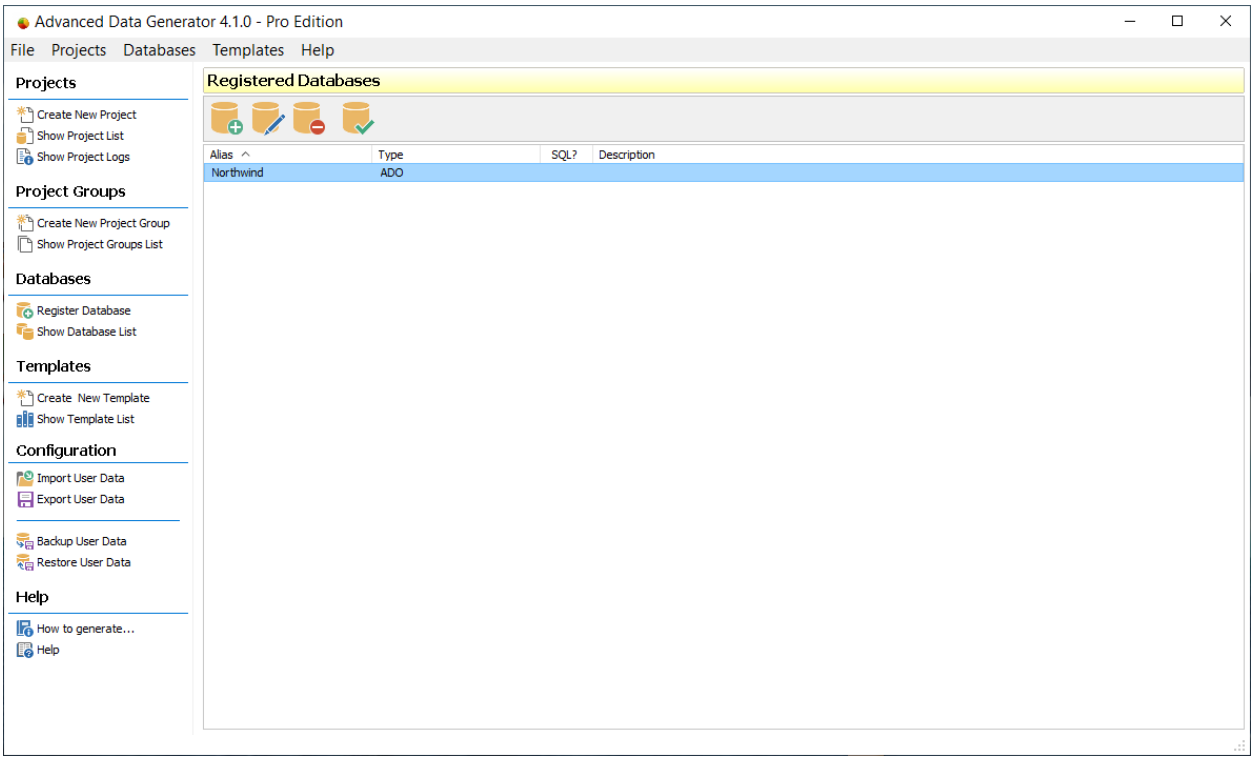

Figure 12. The first step in working with Advanced Data Generator is to register a database. After you do so, this view is available.

Once a database is registered, you can create projects for it. A single database can have multiple projects, so you don't have to create all the test data for a database in one shot. An individual project can populate one or more tables.

To create a project, click the Create New Project in the left pane. You're prompted to select a database from the list of registered databases, as in **Figure 13**.

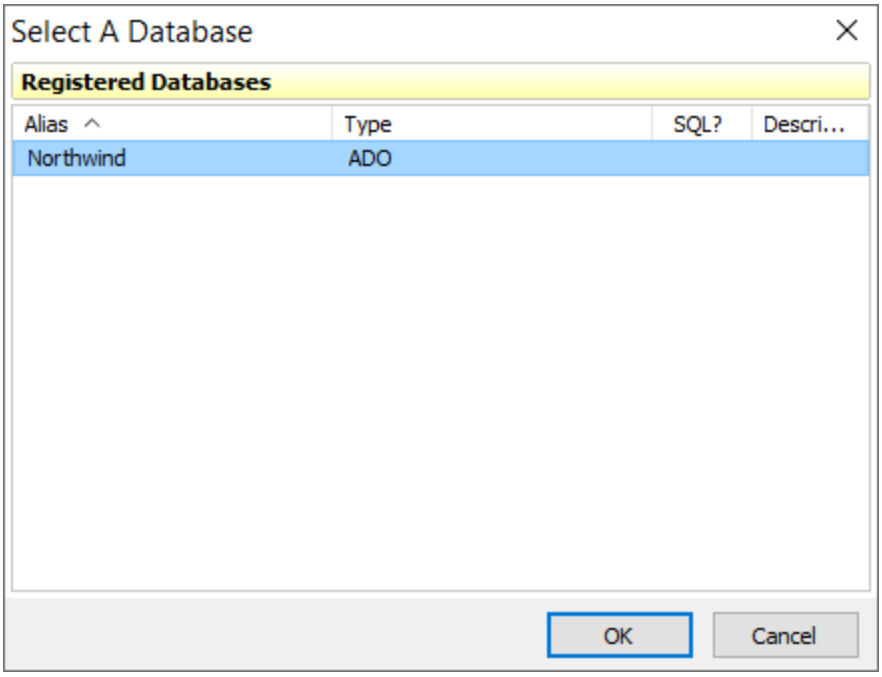

Figure 13. To create a project, you first select a database.

After you select a database, the Data Generator Project dialog opens, set to the Project Settings tab (**Figure 14**). Use this tab to specify a name and description for your project. You also specify where the generated data goes. The default is to put it right in the database, but you can also create SQL scripts or CSV or JSON files, as well as a couple of formats aimed at specific languages.

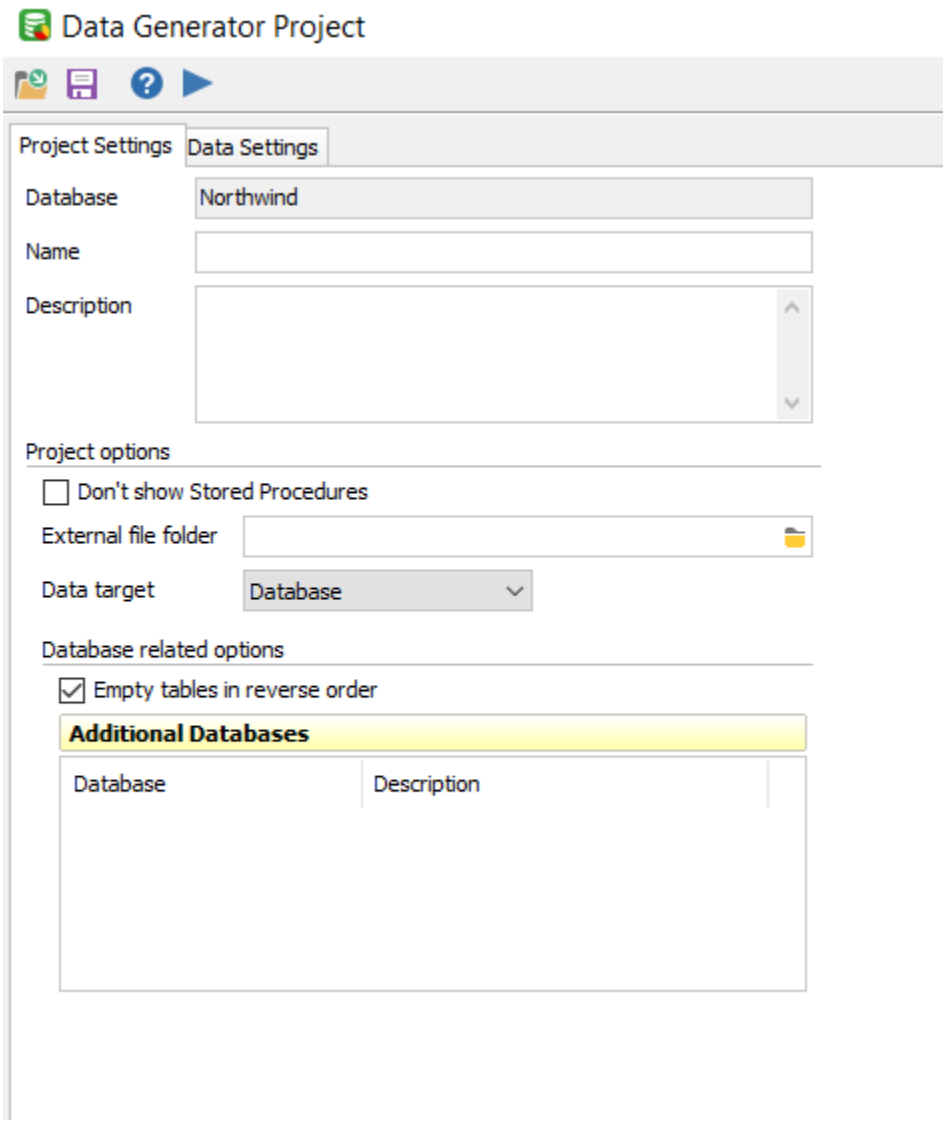

Figure 14. You start creating a project in ADG by specifying a name and description for it.

You may want any existing data to be deleted before generating new data. In that case, if you have referential integrity rules in place, deleting data in the right order is necessary to avoid RI problems. The Empty tables in reverse order checkbox tells ADG to start deletion from the bottom of the list of tables for which data is to be generated.

Specifying fields

Once you specify the project-level information, you use the Data Settings tab (**Figure 15**) to specify how to generate the data. For each table in the database, you can determine whether to generate data and how many records to generate.

One warning before going any farther. As you'd expect, clicking OK to close this dialog saves your work to your project and clicking Cancel throws away any changes made in this session. But I was surprised to find that closing the dialog with the Windows close button

(the "X" in the upper-right corner) discarded my changes. I recommend sticking with the buttons.

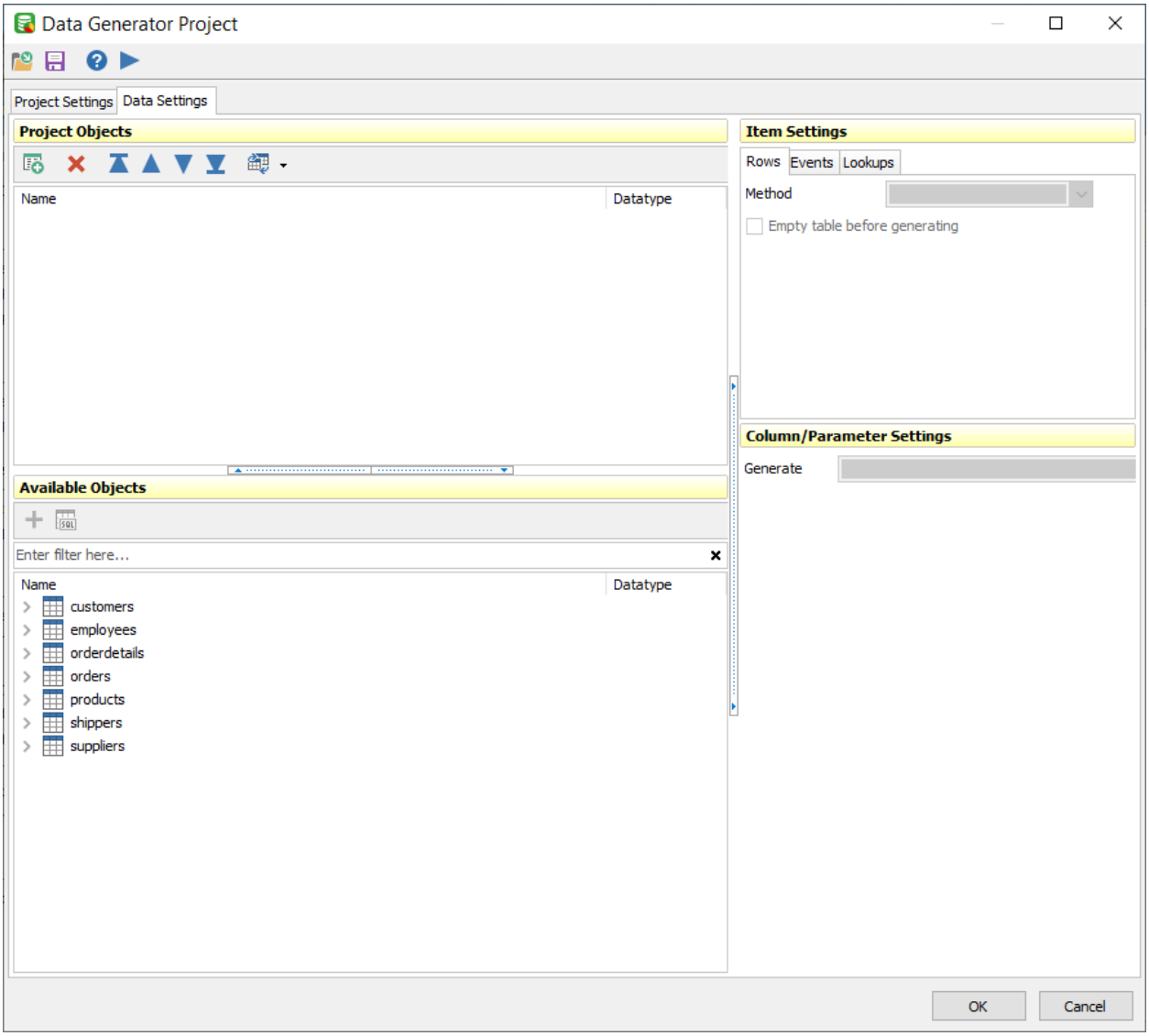

Figure 15. ADG's Data Settings tab lets you indicate which tables to generate data for, the order of data generation, and the data to be generated for each field.

All of the tables in the database are shown in the Available Objects list. To add a table to the project, double-click it and it's added to the Project Objects list. Each field and its type is shown, as in **Figure 16**. Use the Item Settings tab to specify how many rows to generate. The method dropdown there (shown in **Figure 17**) gives you multiple ways to specify the number of rows. That's also where you indicate whether the table should be emptied before generating data.

When you generate data, tables are filled in the order they're shown in the Project Objects pane. The toolbar there lets you change the order of the tables, so you can make sure everything can be generated.

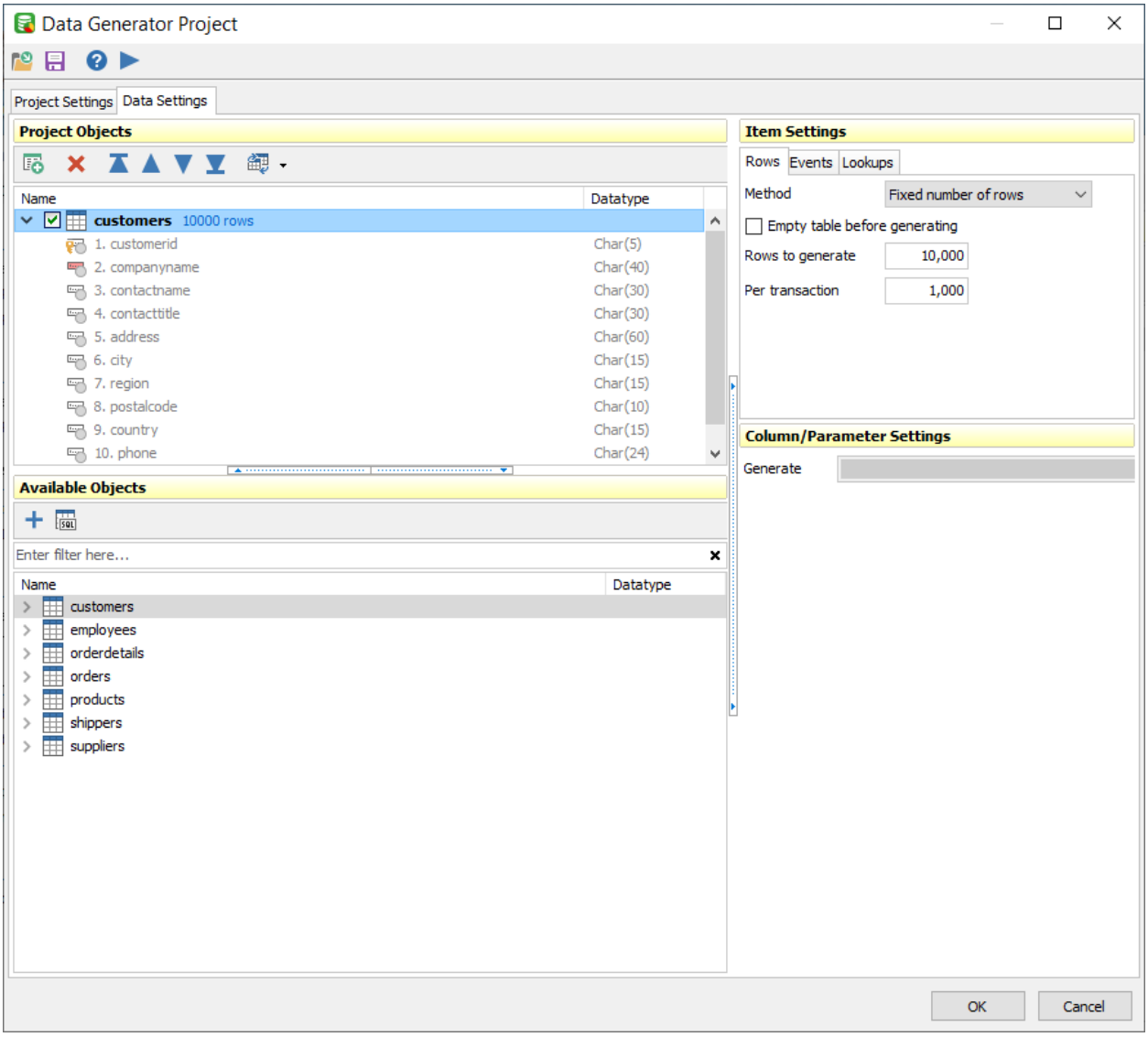

Figure 16. Double-click a table in the Available objects list to add it to the project. Use the Item Settings tab to specify how many records to generate for the table.

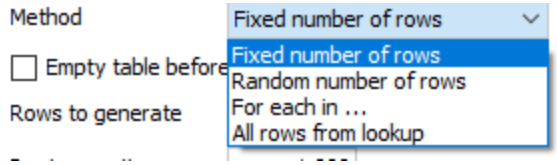

Figure 17. You can specify a fixed number of records for a table, a random number between a lower and upper bound, or base the number on data in other tables.

There are a number of choices for generating data for each field; the list varies based on the data type. When you click on a field, the Column/Parameter Settings pane on the right shows the options for populating that field. Oddly, it assumes you don't want to generate data for that field and the Generate dropdown is set to "[nothing]," as in **Figure 18**.

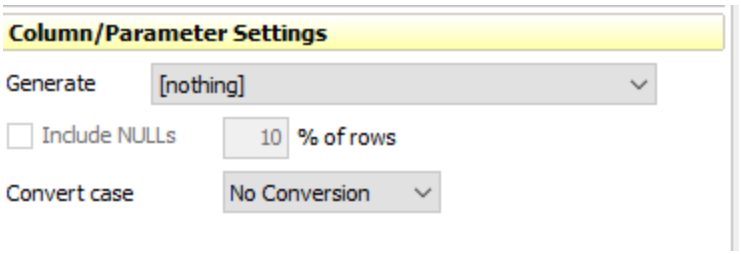

Figure 18. When you choose a field that isn't yet specified, ADG starts out with no data generation.

The Generate dropdown, shown in **Figure 19**, lets you choose what method of data generation to use. Once you make a choice, the rest of the pane shows your options for that data generation method.

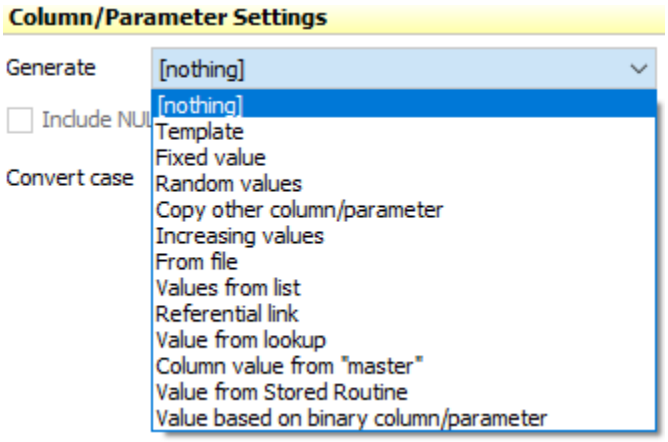

Figure 19. To start specifying data generation for a field, choose one of these methods. Then the rest of the pane will show your options for that method.

In setting up the Northwind data, I used Random values most, with a few Fixed values, some Templates, some Values from list, and some Referential links. (My set-up for the simplified Northwind database is included in the AdvancedDataGenerator folder of the materials for this session as NorthwindProject.dgp.)

Character fields offer the widest range of options. The various Random items, shown in **Figure 20**, are mostly self-explanatory. (It is worth noting that the "Random addresses" item produces a street address in the format shown, with the street name first, so isn't suitable for US and Canadian addresses.) I used a random 5-character string for the

CustomerID field of Customers, and the Random full names option for the ContactName field.

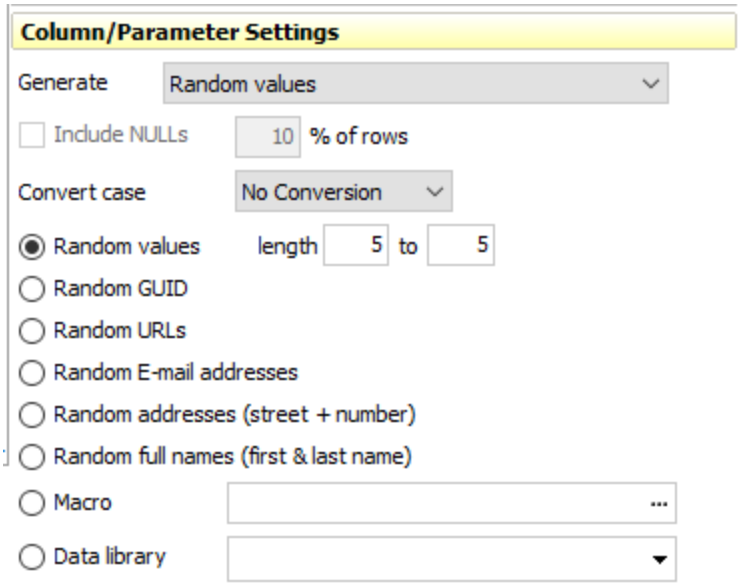

Figure 20. Character fields can be based on a wide variety of sources in ADG. For the CustomerID field of Customers, I specified a random 5-character string.

The Macro option lets you specify a format to which items must conform (analogous to a regular expression). ADG offers a dialog to build these, but you can also just write them directly. **Figure 21** shows the macro I built for the ProductName field of Products; it specifies a string starting with a capital letter and followed by anywhere from 3 to 7 more lower-case letters.

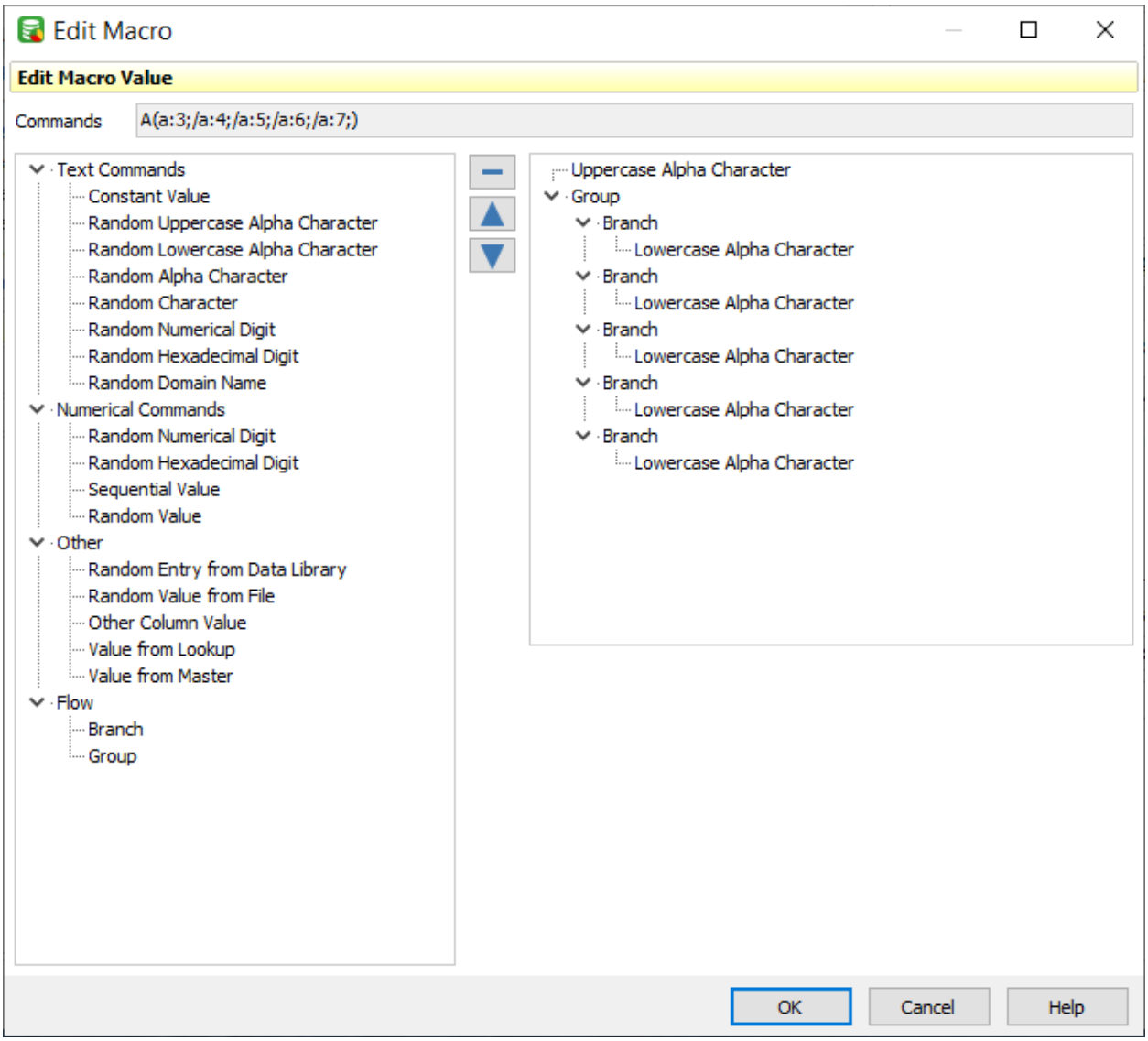

Figure 21. Macros are like regular expressions. You provide instructions for how to construct the string you want.

The Data library option lets you use random items from provided lists of first and last names from various countries (or the whole set), with the option of using only male or only female; city names or street names from various countries; US states; countries; and companies. Unlike Mockaroo, however, there's no connection between streets, cities, and states. (Also, note that even after choosing an item from the Data library dropdown, I had to be sure to click the Data Library option button to keep my selection.)

The absence of what Northwind calls "regions" (states, provinces, cantons, etc.) for other countries led me to restrict the data to US addresses. So, I used the Fixed value method (shown in **Figure 22**) to populate the Country field of the various tables.

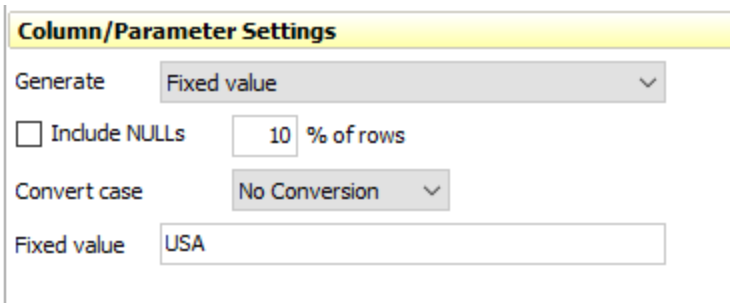

Figure 22. You can specify an exact value for a field. Because I had no way to specify the Region field for countries other than the US, I used this option for the Country field in several Northwind tables.

The Values from list method lets you specify a fixed list of items; as shown in **Figure 23**, I used it for the TitleOfCourtesy field in the Employees table. While this is convenient, if you'll need the same list for multiple fields, it's better to create a template, as described a little later in this section.

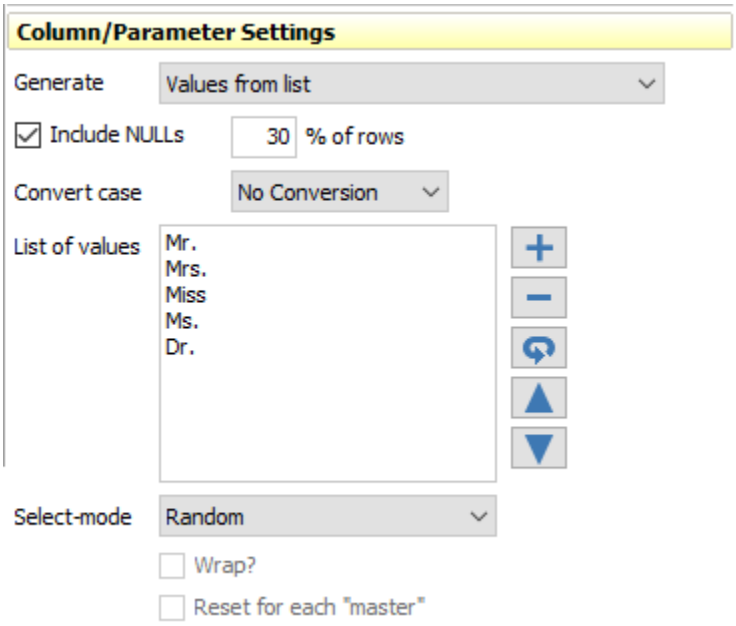

Figure 23. Rather than generating field values randomly in ADG, they can be drawn from a specified list.

The Referential link method lets you specify the table and field the data comes from and whether links should be random, sequential or one-to-one. It's presumably intended primarily for foreign keys but can be used for other data. I used it for all the foreign keys in Northwind (like the SupplierID field in Products, as shown in **Figure 24**), except for the ReportsTo field of Employees. To use a referential link, the source field must have already been populated (which means that the order of the tables in the project matters). Because ReportsTo is self-referential, I couldn't populate it this way.

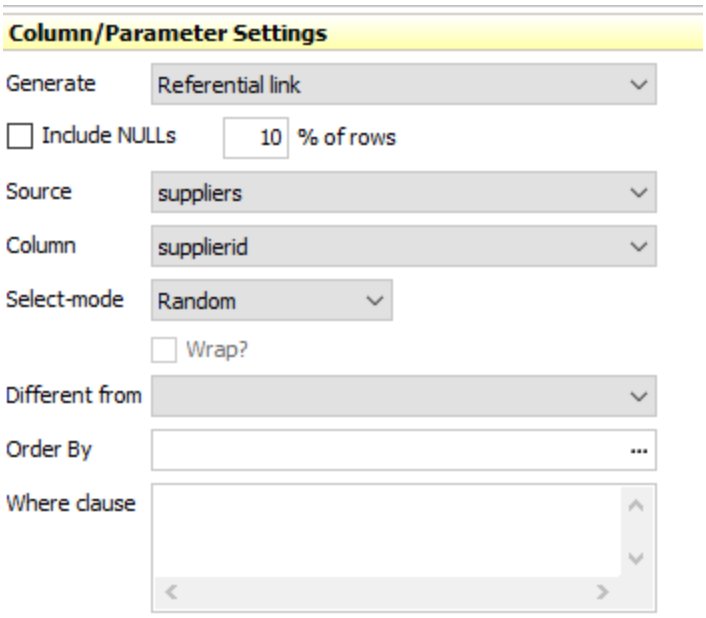

Figure 24. Use a Referential link to pull data from one table into another. The source table must be higher in the list of Project Objects than the table you're defining, so that its data is generated first.

There are several other options for character fields, but other than templates, I didn't try them.

The choices for fields seen as "Text" (a VFP memo field) are more limited. These fields are expected to contain paragraphs and all the choices are oriented toward that goal. In particular, though you can generate URLs for Character fields, there's no way to do so for Text fields.

There are almost as many Generate methods for numeric fields as for character fields, though of course, the options once you make a choice are oriented toward numeric values. For Random values, you can specify a low value, a high value, and the number of decimals. For non-integer numeric types (like Currency, which ADG labels "Money"), you can indicate whether only whole numbers should be generated. **Figure 25** shows my specification for the UnitPrice field of Products.

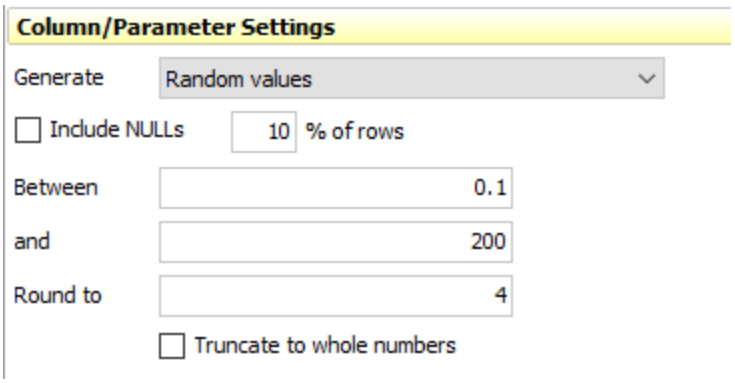

Figure 25. For numeric fields, when you choose Random values, you indicate the lowest value, highest value and number of decimals ("Round to").

Date and datetime fields also offer a lot of generation options. For Random values, you specify start and end dates and whether the time portion should be fixed or also random. **Figure 26** shows the specified I used for the OrderDate field of Orders.

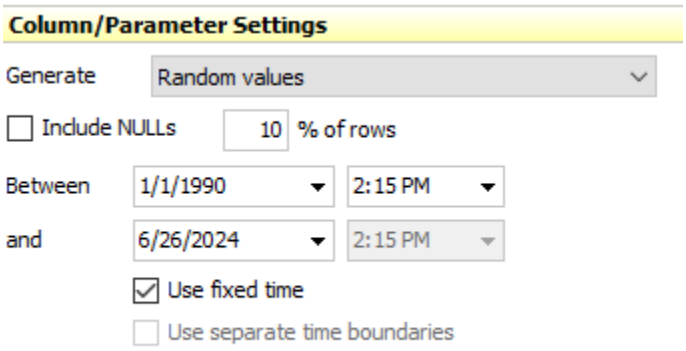

Figure 26. For random dates and datetimes, you specify start and end dates and times.

You can also specify that a date or datetime field should be based on a different field in the same table. Once you choose a field to start with, you specify a range of time units to randomly add to the original value. In **Figure 27**, which is for the RequiredDate field of Orders, I specified between 0 and 30 days after the OrderDate field.

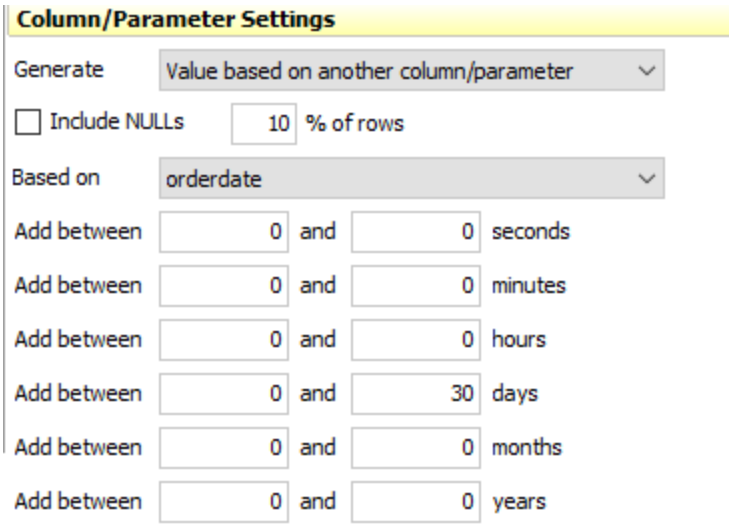

Figure 27. When basing one date or datetime field on another, you can specify how much time to add for the new value.

Creating reusable templates

One attractive feature of ADG is the ability to define *templates* for field types you use repeatedly. A template is like any other field specification, but it's stored and named, so you can apply it repeatedly. ADG comes with a dozen or so presets, including postal codes for a number of countries, including the US; and US phone numbers. Most of the supplied templates use macros. Be warned that you can delete the supplied templates; they're in a list together with the ones you define and there's no special warning before deleting a macro that comes with ADG.

To see the available templates, choose Show Template List in the left pane. The right pane shows the list of templates at the top and the details of the currently selected template at the bottom, as in **Figure 28**.

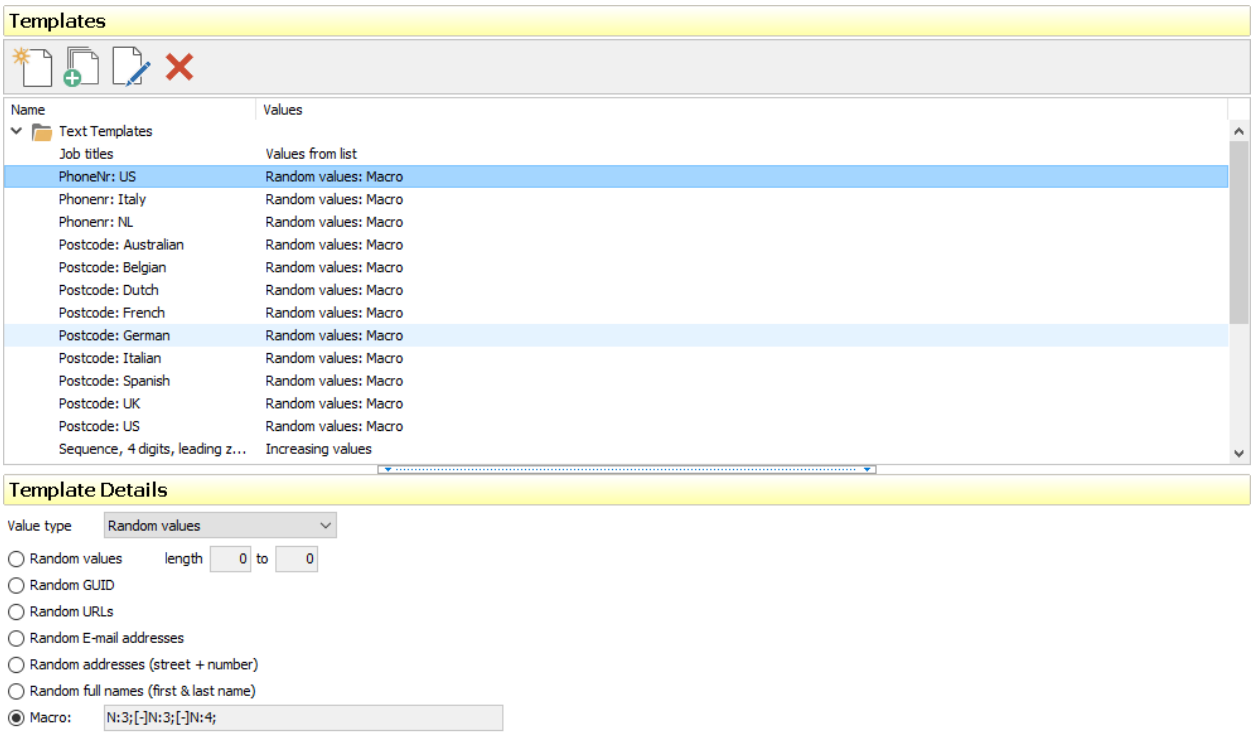

Figure 28. Templates let you define additional data types you can use over and over.

To add a new template, either click the New Template button in the Templates pane or click Create New Template in the left pane. The Create Data Template dialog opens for a brand-new template, as in **Figure 29**. Use the dropdown to specify the data type and then click Next. The dialog changes to Template Details, where you specify a name for it and which method you want to use to generate the value. **Figure 30** shows the dialog with the method dropdown open.

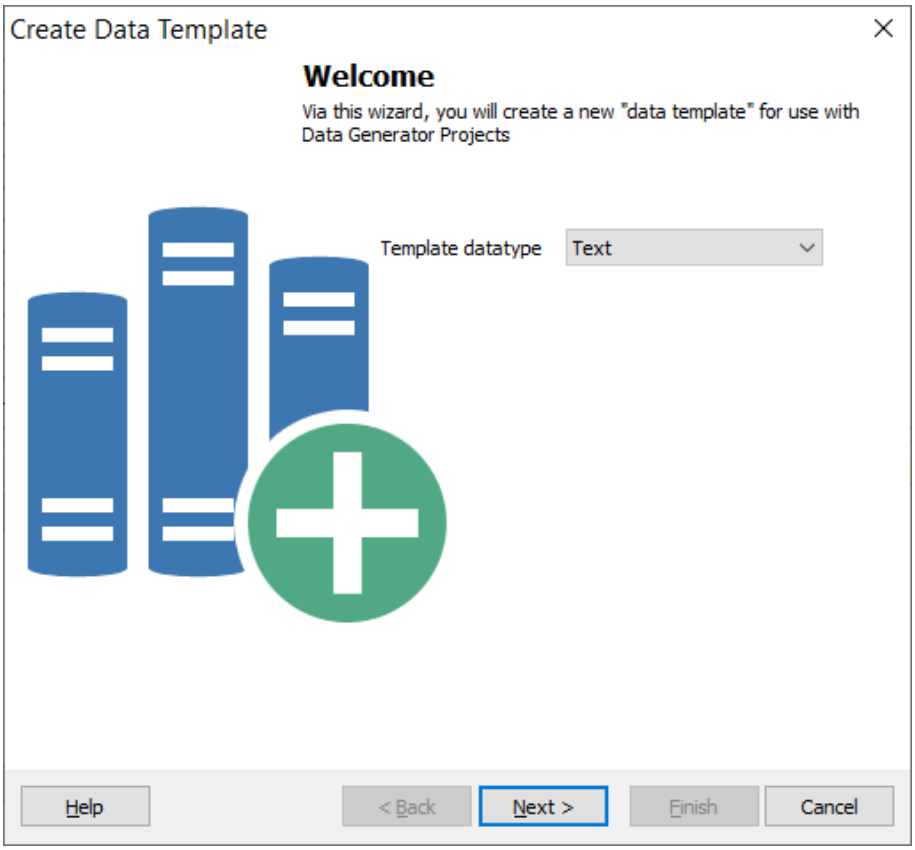

Figure 29. This dialog lets you start building an entirely new template.

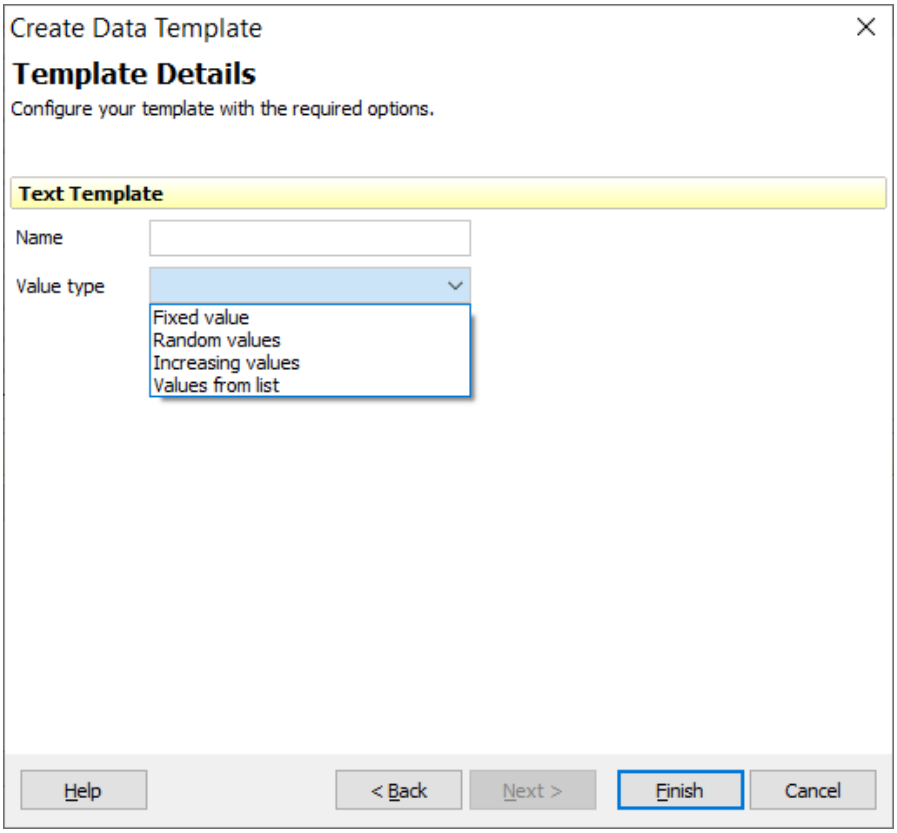

Figure 30. After choosing the template type, you specify a name and the method for generating data.

Click Next to update the Template Details page to let you specify the template. One template I created is for US street addresses, in the usual format with the house number first; it's shown in **Figure 31**. I also created a template for job titles, because several Northwind tables needed them. I based that template on a hard-coded list of values. (I was disappointed to find that, having first simply defined a field based on a list of values, there was no way to copy that list so I could put it into a template instead.)

I'd hoped to use a template to work around the issue with URLs, but templates for Large Text/Memo fields don't offer any more options than for specific fields of that type.

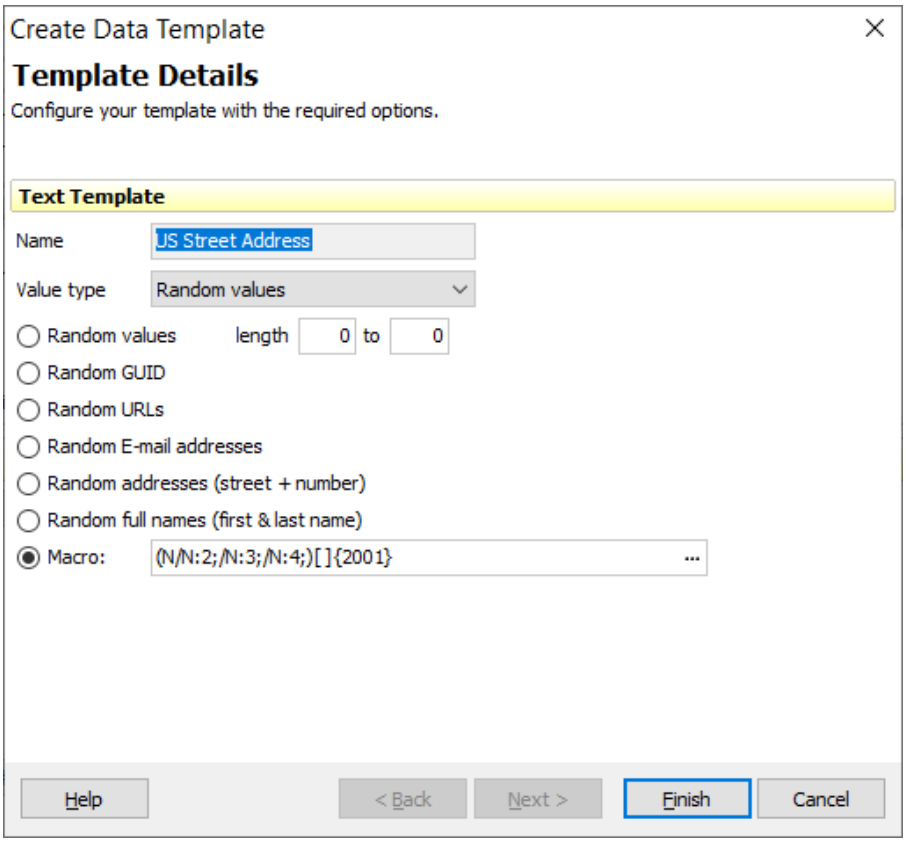

Figure 31. I created this template for US street addresses. It provides one to four digits in front of a street name from the list of US street names in the data library.

You can export and import lists of templates. My list of templates is included with the downloads for this session as MyTemplates.adgx; you'll need to import it if you want to open my project for the simplified Northwind database.

Running data generation

Once you've specified how data is to be generated, you run the project to actually generate the type of output indicated. You can run the project from the main toolbar in the Data Generator Project dialog, from the toolbar in the main ADG window when the list of projects is displayed, or by right-clicking on a project and choosing Run Project. The Run Project dialog appears (with a confirmation prompt that you can eliminate for future runs) and shows progress as the data is generated. Once you confirm, the dialog shows progress as data is generated, as in **Figure 32**.

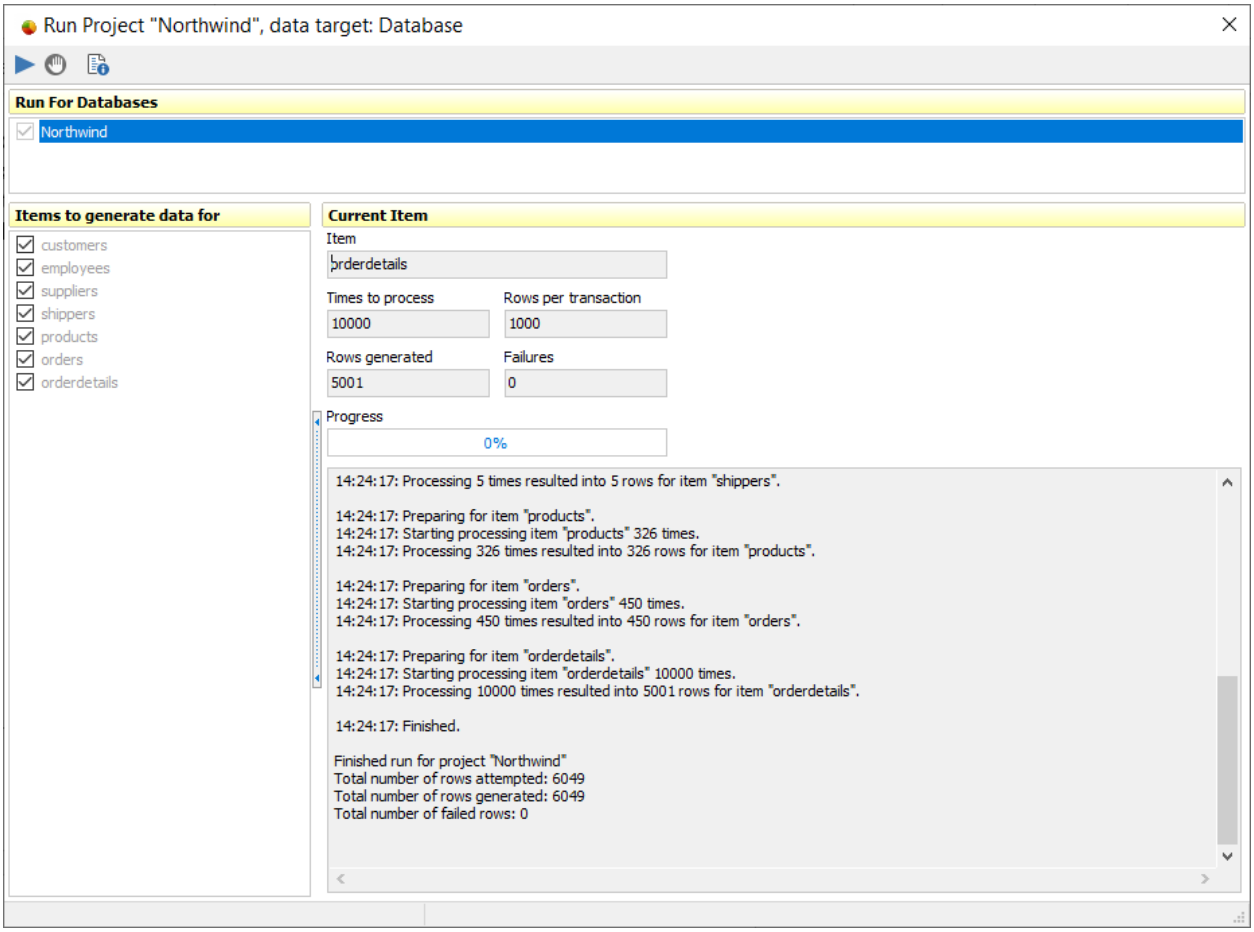

Figure 32. ADG's Run Project dialog gives you feedback as the data is generated.

Final thoughts

ADG has some advanced features I didn't test. For example, you can specify queries to run before and after items in the project. It's not clear to me whether you can specify queries only for the project as a whole, for each table, or in fact, for each field. In addition, you can specify Lookups into outside tables or CSV files. I suspect one of these approaches might solve the problem of the self-referential foreign key in Employees.

I found the behavior of the mouse wheel in ADG confusing. Once I'd clicked into the Generate combobox, using the mouse wheel scrolled that combobox even if the mouse was positioned in a different pane. I had to click elsewhere to change what the mouse wheel affected.

The error messages displayed when something goes wrong in data generation are difficult to interpret. I suspect ADG simply repeats the error returned by the ODBC driver or OLE DB generator.

From a data standpoint, the most significant weakness is that random string values use the full character set and have no notion of words. It would be useful to be able to generate random strings of letters only or random strings of words. It may be possible to create a macro for random strings of letters, but it wasn't easily apparent to me how to do that without specifying the number of letters involved.

ADG provides documentation on their website. You can get to it from the Support link on their home page. Help is available directly from the ADG user interface with a context-sensitive button in the Data Generator Project dialog and a Help item on the main menu. Oddly, when you request help from anywhere inside the product, you land not on the Help pages of ADG's website (https://www.upscene.com/documentation/adg4/), but on a file-based copy of those pages (file:///C:/ProgramData/Advanced%20Data%20Generator%204%20Pro/Documentation/index.htm).

Generating test data with VFP code

The final alternative for generating test data is to use VFP code. The advantage of this approach is that you can tailor it exactly to your needs. The downside, of course, is writing and debugging the generation code. I'll present two options here that reduce that cost. I wrote a set of generic VFP classes to handle test data generation. To use them, you have to write only a small number of mostly simple methods. In addition, a VFPX project called FoxFaker provides a class to generate test data. To use it, you have to instantiate it and write code to call its methods as needed.

My custom test data generator

Like the commercial products, my test data generator has some basic data to draw from, including lists of last names; male and female first names; street names; city, state, and zip code combinations; and area codes. Most of these lists (which are included with the session materials) were created by finding an appropriate list somewhere on the Internet and converting it to a VFP table.

Overall structure

Creating a test data set involves two processes, generating the data and storing it. 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 for the same data. It also enables saving code to generate the data, such as a set of INSERT commands, rather than saving the data itself.)

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. Both classes are in MakeDataV2.PRG in the Generator folder of the session materials.

Both class hierarchies use a class that puts a wrapper around the RAND() function. RandFunctions seeds RAND() in its Init method, and includes three functions that use RAND():

- RandInt returns a random integer between specified values.
- RandLetter returns a random letter of the alphabet.
- RandRecord chooses a record at random from a specified table and returns the value of a specified field.

This data generator is organized around data sets and types. A data set is the whole set of test data to be generated. A data type is a particular kind of record to generate. Though data types usually correspond to individual tables, a data type could, in fact, include data aimed at multiple tables. (See "Generating People" below.) The generator lets you specify the number of records to generate for each data type.

Creating a data set

MakeDataSet is fairly simple. It's subclassed from Session (so that it works in a private data session) and has five 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;
- oRand holds an object reference to a RandFunctions object;
- oRecordGenerator holds an object reference to the MakeRecord subclass;
- oDataToGenerate holds a collection indicating the types and numbers of records to generate.

The only built-in methods containing code are Init and Destroy. Init is brief; it's shown in **Listing 2**.

Listing 2. The Init method of MakeDataSet sets things up for the process.

```
This.oRand = NEWOBJECT("RandFunctions","RandFuncs.PRG")
This.oRecordGenerator = ;
   NEWOBJECT(This.cGeneratorClass, This.cGeneratorClassLib)
This.oDataToGenerate = CREATEOBJECT("Collection")
This.OpenTables()

This.SetData()
```

Destroy is even simpler. It just closes the tables opened in Init by calling the custom CloseTables method.

The class has 11 custom methods, many 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
About	Documentation for this class.
AddDataType	Add a data type to the collection of types to generate.
AfterMakeSet	Code to run after all records have been added. Abstract.
AfterMakeType	Code to run after creating all records of one type. Pass the type as a parameter. 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.
GetRandRecord	Calls the RandRecord method of the RandFunctions object.
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.
SetData	Sets up the collection of data types to create. Abstract.

AddDataType adds a data type to the collection of types to generate. It accepts two parameters: the name of the data type and the number of records to generate. The code, intended to be called from SetData in subclasses, is straightforward; it's shown in **Listing 3**.

Listing 3. The AddDataType method adds a data type to the collection of types to generate.

```
PROCEDURE AddDataType(cName, nCount)
LOCAL oDataObject

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

* Create the data object
oDataObject = CREATEOBJECT("Empty")
ADDPROPERTY(oDataObject, "Type", m.cName)
ADDPROPERTY(oDataObject, "RecordCount", m.nCount)

* Add the object to the collection, using the type as the key
This.oDataToGenerate.Add(oDataObject, m.cName)

RETURN
```

Although the MakeSet method (shown in **Listing 4**)is the driver for the whole process, the code is pretty simple. The method goes through the list of types to generate and creates the specified number of records for that type.

Listing 4. MakeDataSet is the main routine for generating a data set.

```
LOCAL oDataType, nRecord, oRecord

FOR EACH oDataType IN This.oDataToGenerate FOXOBJECT
   FOR nRecord = 1 TO oDataType.RecordCount
      oRecord = This.oRecordGenerator.GenerateRecord(oDataType.Type)
      This.SaveRecord(oRecord, oDataType.Type)
   ENDFOR

   This.AfterMakeType()
ENDFOR

This.AfterMakeSet()

RETURN
```

CheckLookup (**Listing 5**) lets you store look-up data as you store the rest of the data, as well as create links to look-up data. CheckLookup can be called from SaveRecord in a subclass. 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.

Listing 5. The CheckLookup method lets you populate look-up tables as you're saving the records that reference them.

```
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
```

Creating a record

MakeRecord provides basic tools that make writing subclass code to generate records easier. A number of its methods are abstract at this level.

MakeRecord has four custom properties:

- oData is a collection holding the list of tables (raw data tables, such as the list of surnames) 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 each record type;

- oRand is an object reference to a RandFunctions object.
- 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, shown in **Listing 6**, calls several methods that do the actual work of setting things up.

Listing 6. The Init method of MakeRecord sets things up for generating a single record.

```
This.oRand = NEWOBJECT("RandFunctions","RandFuncs.PRG")
This.oData = CREATEOBJECT("Collection")
This.oMethods = CREATEOBJECT("Collection")

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

Destroy cleans up; it's shown in **Listing 7**.

Listing 7. The Destroy method cleans up from record generation.

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

MakeRecord has 13 custom methods, listed in **Table 2**.

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

Method	Purpose
About	Documentation method.
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 and the data type as parameters.
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 by calling the RandInt method of the RandFunctions object.
RandLetter	Returns a random letter of the alphabet by calling the RandLetter method of the RandFunctions object.
RandRecord	Chooses a record at random from a specified table and returns the value of a specified field by calling the RandRecord method of the RandFunctions object.
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 (city/state/zip) table and the table of area codes. In concrete subclasses, SetData is likely to be a series of calls to AddData.

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; it's in **Listing 8**.

Listing 8. AddData makes it easy to add to the oData collection of raw data tables.

```
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, "RecordCount")

* 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 in **Listing 9**.

Listing 9. The OpenData method opens the raw data tables needed for record generation.

```
LOCAL oTableInfo, lReturn

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

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

```
RETURN lReturn
```

CloseData, shown in **Listing 10**, loops through the oData collection, closing the tables.

Listing 10. The CloseData method closes the tables opened by the record generator.

```
LOCAL oTableInfo

FOR EACH oTableInfo IN This.oData FOXOBJECT
  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.

The three RandX methods aren't called by code in MakeRecord; they're provided to be used in code added to subclasses.

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 to generate the actual data for each data type. The methods themselves are added at the subclass level, as well. In subclasses, the code in SetMethods likely to be a list of calls to AddMethod.

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. Shown in **Listing 11**, it loops through the list of methods in the aMethods array, calling those that apply to the specified data type:

Listing 11. The GenerateRecord method drives the process of creating each record.

```
PROCEDURE GenerateRecord(cRecordType)

LOCAL oMethod, cMethod

This.oRecord = CREATEOBJECT("Empty")

FOR EACH oMethod IN This.oMethods FOXOBJECT
```

```
   IF oMethod.cGroup == m.cRecordType
      cMethod = "This." + oMethod.Name
      &cMethod
   ENDIF
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 I created 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 MakePeopleV2.PRG, which is included in the Generator folder of the session materials.

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. 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.
nHasExtension	The probability that a phone number includes an extension.
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 raw data tables (which are included in the Generator\RawData folder of the materials for this session). These provide a group of names, streets, area codes and so forth. They're all listed in the SetData method, shown in **Listing 12**, which uses the AddData method to populate the oData collection.

Listing 12. This code in the SetData method tells the generator to open a bunch of tables containing raw data.

```
PROCEDURE SetData

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

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

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, shown in **Listing 13**, fills in the likelihood that the person has various types of data. For example, the chance of a home (personal) address is set to 90%, but there's only a 40% chance of a work (business) address and a 20% chance 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.

Listing 13. The SetProbabilities method populates a set of arrays that indicate how likely a given person is to have a given type of contact information. Only a subset of the method is shown here.

```
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 (**Listing 14**) lists the methods to be called in the order in which they should be called, calling AddMethod to populate the oMethods collection. :

Listing 14. The SetMethods method is where you indicate what methods to call to generate each data type.

```
PROTECTED PROCEDURE SetMethods

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

RETURN
```

The real work is done in all the GetXXX methods listed in SetMethods. Each one creates one kind of data. GetBirthdate is the simplest and demonstrates the most basic ideas; it's shown in **Listing 15**.

Listing 15. The GetBirthdate method is one of a number of GetXXX methods that do the actual work of generating the test data.

```
PROTECTED PROCEDURE GetBirthdate
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, shown in **Listing 16**, generates a first name and last name and sets the record's gender. It uses the BoysNames, GirlsNames and LastNames tables. The method calls RandRecord to return a surname. 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". (This code was written nearly 20 years ago and needs updating to reflect other gender options.)

Listing 16. The GetName method generates a first name, last name, and gender.

```
PROTECTED PROCEDURE GetName
LOCAL nRec, nRand

* Choose a last name
ADDPROPERTY(This.oRecord, "cLast", ;
```

```
   ALLTRIM(This.RandRecord("LastNames","cName")))

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

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 object 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, RandLetter and RandRecord, calls to RAND(), etc.) 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 (shown in **Listing 17**) is typical.

Listing 17. The GetAddresses method creates a collection of addresses for an individual, using the probabilities to decide which ones to create.

```
PROTECTED PROCEDURE GetAddresses
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
      * Use method to move to correct record, but need to
      * retrieve multiple fields
      This.RandRecord("Streets","cStreet")
      cStreet = Streets.cDir -(" " + Streets.cStreet) - (" " + Streets.cType)

      * Get a city, state, zip combination
      This.RandRecord("Cities","cCity")

      ADDPROPERTY(oAddress,"Street", ;
         m.cHouseNumber + " " + ALLTRIM(m.cStreet))
      ADDPROPERTY(oAddress,"City", Cities.cCity)
      ADDPROPERTY(oAddress,"State", Cities.cState)
      ADDPROPERTY(oAddress,"Zip", Cities.cZip)
      ADDPROPERTY(This.oRecord, "AreaCode", Cities.cACode)

      * 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 random Social Security number and shown in **Listing 18**. The code follows the basic rules for the structure of a US Social Security number. 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.

Listing 18. GetSSN applies the rules for generating a US Social Security number and ensures we haven't already generated the same value.

```
PROTECTED PROCEDURE GetSSN
LOCAL cSSN, nDigit1, nDigit2, nDigit3, nMiddle, 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 in SetMethods.

Generating a set of people

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

Listing 19 shows 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.

Listing 19. This code from OpenTables opens tables if they exist and creates them otherwise. If the Person table already exists, it creates a cursor of social security numbers that are already in use.

```
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
```

SetData just adds the Person data type to the oDataToGenerate collection, with a count of 5000.

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 generating test data was designed specifically to test a new approach to storing contact information; it put all contact items into a single table and maintained a pair of look-up tables to indicate the item type and location. The version shown in **Listing 20** 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.

Listing 20. The SaveRecord method stores the data you've generated to actual records.

```
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 FOXOBJECT
    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 FOXOBJECT
    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 FOXOBJECT
    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 FOXOBJECT
    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.

Generating data for the simplified Northwind database

When I wrote the original version of this paper back in 2007, I created another set of subclasses to create test data for a simple version of a college database, with students, instructors, departments, and courses. While I'm not going to cover that example in this paper, I've included the code in MakeSchoolDataV2.PRG and the tables from that example in the Generator\School folder of the downloads for this session.

Instead, we'll explore classes to generate test data for the simplified version of the Northwind database described earlier in this paper. As with the previous cases, we need to subclass both MakeDataSet and MakeRecord. Both subclasses are in MakeNorthwind.PRG in the Generator folder of the downloads for this session.

Northwind is a much more complex database with a lot more kinds of data than either the tables used for person data or the School database, but it uses all the raw data tables used for generating people, as well as two more that I added (again by finding data online and scraping it). The JobTitles table contains almost 200 job titles, while ProductNames contains about 300 grocery items. So, the SetData method is the same as in MakePerson, plus the two additional tables.

MakeNorthwind has quite a few custom properties that determine either minimum and maximum values for an item to be generated or specify the probability of a particular choice. The portion of the class definition that sets them is shown in **Listing 21**.

Listing 21. These properties set at the top of MakeNorthwind determine the range of values for an item or how often a given option occurs.

```
* Ranges
nDomainWordMax = 3 && Maximum number of combined words for domain name
                   && Use for company name, too

** Birthdates
dOldest = DATE(1954,1,1)
dYoungest = GOMONTH(DATE(), -18 * 12)

** Company dates, used for hire dates and order dates
dStarted = DATE(2000, 1, 1)

** Addresses
nLowHouseDigits = 1 && # of digits
nHighHouseDigits = 5 && # of digits
nHasLetter = 0.3 && is there a letter after the number

* Order dates
nReqWithin = 30 && orders required within 30 days
nMaxLate = 60 && late orders shipped within 60 days of required

* Product price
nLowPrice = 0.05
nHighPrice = 19.99
```

```
* Stock quantities
nLowQty = 0
nHighQty = 1000

* Freight charges
nMinFreight = 0.01
nMaxFreight = 1111

* Lines per order
nMinLines = 1
nMaxLines = 25

* Order quantities
nMinItems = 1
nMaxItems = 100

* Discounts (in %)
nMinDiscount = 1
nMaxDiscount = 25

* Probabilities
nMale = 0.5 && Determines what percent of result records are male
nHasExtension = 0 && No phone extensions
nDiscontinued = 0.05 && What percent of products are discontinued?
nOrderLate = 0.05 && What percent of orders ship after the required date?
nHasDiscount = 0.3   && What percent of line items have a discount?
```

There are seven tables in the simplified Northwind database, but I divided them into six data types. All of the tables except Orders and OrderDetails can be seen as look-up tables for those last two, so each of those five (Customers, Employees, Shippers, Suppliers, and Products) is a separate data type. I combined Orders and OrderDetails into a single data type, generating all the line items together with the header. **Listing 22** shows the code in SetMethods, to set up generation of each type.

Listing 22. The SetMethods code shows the Northwind data divided into six different data types.

```
PROTECTED PROCEDURE SetMethods

WITH This
   * Customers
   .AddMethod("GetCompanyName", "Customer")
   .AddMethod("GetName", "Customer")
   .AddMethod("GetJobTitle", "Customer")
   .AddMethod("GetAddress", "Customer")
   .AddMethod("GetPhone", "Customer")
   .AddMethod("GetFax", "Customer")

   * Employees
   .AddMethod("GetName", "Employee")
   .AddMethod("GetJobtitle", "Employee")
   .AddMethod("GetBirthdate", "Employee")
   .AddMethod("GetHiredate", "Employee")
   .AddMethod("GetAddress", "Employee")
   .AddMethod("GetPhone", "Employee")
```

```
*!*      .AddMethod("GetNotes", "Employee")

   * Shippers
   .AddMethod("GetCompanyName", "Shipper")
   .AddMethod("GetPhone", "Shipper")

   * Suppliers
   .AddMethod("GetCompanyName", "Supplier")
   .AddMethod("GetName", "Supplier")
   .AddMethod("GetJobTitle", "Supplier")
   .AddMethod("GetAddress", "Supplier")
   .AddMethod("GetPhone", "Supplier")
   .AddMethod("GetFax", "Supplier")
   .AddMethod("GetURL", "Supplier")

   * Products
   .AddMethod("GetProductName", "Product")
   .AddMethod("GetSupplier", "Product")
   .AddMethod("GetPrice", "Product")
   .AddMethod("GetStockUnits", "Product")
   .AddMethod("GetDiscontinued", "Product")

   * Orders
   .AddMethod("GetCustomer", "Order")
   .AddMethod("GetEmployee", "Order")
   .AddMethod("GetOrderDates", "Order")
   .AddMethod("GetShipper", "Order")
   .AddMethod("GetFreightCharge", "Order")
   .AddMethod("GetOrderLines", "Order")

ENDWITH

RETURN
```

The GetName method is the same as the one used in MakePerson, and the GetAddress method is a simplified version of MakePerson's GetAddresses method, creating a single address rather than a set of addresses.

It quickly became clear that Northwind required multiple kinds of phone numbers, some with extensions and some never having them, so I created a GetPhoneNumber method (**Listing 23**) to generically create and return a phone number object. Among other things, GetPhoneNumber lets you use an existing area code (as you might have from the GetAddress method).

Listing 23. GetPhoneNumber creates an object containing the components of a phone number and returns it.

```
PROTECTED PROCEDURE GetPhoneNumber(tlHasExtension)
* Generate a phone number. Used by multiple routines

LOCAL nPhone, nRand, oPhone

* Generate this one
oPhone = CREATEOBJECT("Empty")
```

```
* Create a phone number. Use the area code already saved,
* if there is one.

IF PEMSTATUS(This.oRecord, "AreaCode", 5)
   cAreaCode = This.oRecord.AreaCode
ELSE
   cAreaCode = This.RandRecord("AreaCode","NPA")
ENDIF

nFirstDigit = This.RandInt(2, 9)
cNumber = TRANSFORM(m.nFirstDigit)
* Loop to get rest of digits.
FOR nDigit = 2 TO 7
   cNumber = m.cNumber + TRANSFORM(This.RandInt(0,9))
ENDFOR

ADDPROPERTY(oPhone,"AreaCode", m.cAreaCode)
ADDPROPERTY(oPhone,"Number", m.cNumber)

IF m.tlHasExtension
   nRand = This.RandInt(2, 4) && number of digits
   nLow = INT(10^(nRand-1))
   nHigh = INT(10^nRand-1)
   nRand = This.RandInt(nLow, nHigh)
   cExt = PADL(nRand, 4, "0")
ELSE
   cExt = ""
ENDIF
ADDPROPERTY(oPhone, "Extension", m.cExt)


RETURN m.oPhone
```

Then, I created separate methods for phone numbers (GetPhone, which might have an extension, is shown in **Listing 24**) and faxes (GetFax, which don't have extensions).

Listing 24. GetPhone uses GetPhoneNumber, passing a parameter to indicate whether to add an extension.

```
PROTECTED PROCEDURE GetPhone
* Get a phone number

LOCAL oPhone, lAddExtension

nRand = RAND()
lAddExtension = (m.nRand <= This.nHasExtension)

oPhone = This.GetPhoneNumber(m.lAddExtension)

ADDPROPERTY(This.oRecord, "Phone", m.oPhone)

RETURN
```

It also quickly became apparent that I'd need a number of different dates, using different date ranges, so I created a generic GetDate routine, based on the GetBirthdate method from MakePerson. Shown in **Listing 25**, it accepts parameters for the earliest and latest dates and returns the generated date.

Listing 25. GetDate accepts first and last date parameters and generates a random date between them.

```
PROTECTED PROCEDURE GetDate(dFirstDate, dLastDate)
* Return a random date between the specified dates

LOCAL nRand, nRangeSize, dDate

nRangeSize = m.dLastDate - m.dFirstDate + 1

nRand = This.RandInt(1, m.nRangeSize)

dDate = m.dFirstDate + m.nRand - 1

RETURN m.dDate
```

Three routines use GetDate: GetBirthdate, GetHireDate, and GetOrderDates. GetBirthdate is the simplest; it passes the properties that specify the earliest and latest possible birthdates. GetHireDate (shown in **Listing 26**) uses the generated birthdate for the record if there is one to set up the range of possible hire dates. If no birthdate has been generated, it uses the property dStarted, which represents the start date for the company.

Listing 26. GetHireDate tries to make the date of hiring make sense with the person's age or the company's start date.

```
PROTECTED PROCEDURE GetHireDate
* Generate a hire date for this person.
* Make sure it's at least 18 years after person's
* birthdate, if we have a birthdate.

LOCAL nRand, dBirth, dEarliest, dHire

IF PEMSTATUS(This.oRecord, "Birthdate", 5)
   dBirth = This.oRecord.Birthdate
ELSE
   dBirth = {}
ENDIF

IF EMPTY(m.dBirth)
   dEarliest = This.dStarted
ELSE
   dEarliest = MAX(GOMONTH(m.dBirth, 12 * 18), This.dStarted)
ENDIF

dHire = This.GetDate(m.dEarliest, DATE())
ADDPROPERTY(This.oRecord, "Hiredate", m.dHire)

RETURN
```

GetOrderDates generates the date for an order and then sets the required date and shipping date based on the order date. It calls GetDate three times, passing different parameters for each. It's shown in **Listing 27**.

Listing 27. GetOrderDates calls GetDate three times to set up the order date, required date, and shipping date for an order.

```
PROTECTED PROCEDURE GetOrderDates
* Get the order date, required date and shipped date
* for an order.
LOCAL dOrder, dRequired, dShipped
LOCAL nRand

dOrder = This.GetDate(This.dStarted, DATE())

nRand = This.RandInt(1, This.nReqWithin)
dRequired = This.GetDate(m.dOrder, m.dOrder + m.nRand)

* Was it late?
nRand = RAND()
IF m.nRand <= This.nOrderLate
   dShipped = This.GetDate(m.dRequired, m.dRequired + This.nMaxLate)
ELSE
   dShipped = This.GetDate(m.dOrder, m.dRequired)
ENDIF

ADDPROPERTY(This.oRecord, "OrderDate", m.dOrder)
ADDPROPERTY(This.oRecord, "ReqdDate", m.dRequired)
ADDPROPERTY(This.oRecord, "ShipDate", m.dShipped)

RETURN
```

GetCompanyName uses the same list of words as MakePerson's GetDomainName, but strings one or more together with some punctuation to make a company name. It's shown in **Listing 28**.

Listing 28. GetCompanyName chooses a few words and combines them to create a company name.

```
PROTECTED PROCEDURE GetCompanyName
* Get a company name by combining data
* from the Domains table

LOCAL nWords, cCompName, nWord, cWord

* How many words to use?
nWords = This.RandInt(1, This.nDomainWordMax)

cCompName = ""
FOR nWord = 1 TO m.nWords
   cWord = ALLTRIM(This.RandRecord("Domains", "cWord"))
   DO CASE
   CASE m.nWords = 1 && only one word in name
      cCompName = m.cWord
```

```
   CASE m.nWord = m.nWords-1 && next-to-last, add and
      cCompName = m.cCompName + m.cWord + ", and "

   CASE m.nWord = m.nWords
      cCompName = m.cCompName + m.cWord

   OTHERWISE
      cCompName = cCompName + m.cWord + ", "

   ENDCASE
ENDFOR

ADDPROPERTY(This.oRecord, "CompanyName", m.cCompName)

RETURN
```

GetDomainName and GetTLD are the same as in MakePerson, but I modified GetURL so that if we've already generated a company name, it extracts the words from the company name and strings them together for the DomainName. It's shown in **Listing 29**.

Listing 29. GetURL checks whether there's already a company name and uses the words there as the domain name, if possible. If not, it generates the domain randomly.

```
PROTECTED PROCEDURE GetURL
* Generate URL. If we already have
* a company name, mash it together.
* Otherwise, generate randomly.

LOCAL cURL

IF PEMSTATUS(This.oRecord, "CompanyName", 5)
   LOCAL nCommas, cCompName, nWord
   cURL = ''
   cCompName = This.oRecord.CompanyName
   nCommas = OCCURS(',', m.cCompName)

   FOR nWord = 1 TO m.nCommas + 1
      cWord = ALLTRIM(GETWORDNUM(m.cCompName, m.nWord, ','))
      IF LEFT(m.cWord, 4) = 'and '
         cWord = SUBSTR(m.cWord, 5)
      ENDIF
      cURL = m.cURL + m.cWord
   ENDFOR
ELSE
   * Create randomly
   cURL = This.GetDomainName()
ENDIF

cURL = "www" + "." + m.cURL
cURL = ALLTRIM(m.cURL - "." - This.GetTLD())


* Add it
ADDPROPERTY(This.oRecord, "URL", m.cURL)
```

```
RETURN
```

GetJobTitle is a simple method that just picks a random record from the JobTitles table; it's shown in **Listing 30**. GetProductName is analogous, choosing randomly from the ProductNames table.

Listing 30. GetJobTitle just chooses a random record from the JobTitles table.

```
PROTECTED PROCEDURE GetJobTitle
* Generate a random job title

LOCAL nRand, cJobTitle

cJobTitle = This.RandRecord("JobTitles","Title")

ADDPROPERTY(This.oRecord, "JobTitle", m.cJobTitle)
RETURN
```

For products, we need to generate three values related to how many we have or will have: the number in stock, the number on order, and the reorder level. I created a single method to generate all three. For the number in stock and the number on order, we just call RandInt, passing properties for low and high quantity. Reorder level seemed like it should be a fraction of the maximum we can order, so I did that in two steps, as shown in **Listing 31**.

Listing 31. Products need three quantities: in stock, on order, and reorder level. The first two are simple, but I chose to set reorder level to a random value between 10% and 50% of the maximum quantity you can order.

```
PROTECTED PROCEDURE GetStockUnits
* Generate number in stock, number on order, and reorder level

LOCAL nInStock, nOnOrder, nReorderLevel

nInstock = This.RandInt(This.nLowQty, This.nHighQty)
nOnOrder = This.RandInt(This.nLowQty, This.nHighQty)

* Reorder level should never be 0, and should be significantly lower
* than the high qty. Use a random value between 10% and 50% of high qty.
LOCAL nScaleReorder

nScaleReorder = This.RandInt(10,50)/100

nReorderLevel = This.RandInt(1, m.nScaleReorder * This.nHighQty)

ADDPROPERTY(This.oRecord, "InStock", m.nInstock)
ADDPROPERTY(This.oRecord, "OnOrder", m.nOnOrder)
ADDPROPERTY(This.oRecord, "ReorderLevel", m.nReorderLevel)

RETURN
```

A property indicates what percent of Products should be marked as discontinued. The GetDiscontinued method (**Listing 32**) uses that property.

Listing 32. To see whether a product is discontinued, we generate a random value and compare it to the property that specifies what fraction of products should be so marked.

```
PROTECTED PROCEDURE GetDiscontinued
* Generate a discontinued flag

LOCAL nRand, lDiscontinued

nRand = RAND()
lDiscontinued = (m.nRand <= This.nDiscontinued)
ADDPROPERTY(This.oRecord, "Discontinued", m.lDiscontinued)

RETURN
```

Several different kinds of money amounts are needed, so I created a generic GetMoneyAmount method (**Listing 33**) that accepts low and high values as parameters and returns a random amount between them. It generates dollars and cents separately, in case either the high or low value isn't an even dollar amount.

Listing 33. GetMoneyAmount generates a dollars and cents number between low and high values and returns it.

```
PROTECTED PROCEDURE GetMoneyAmount(tnLow, tnHigh)
* Generate a random money amount

LOCAL nDollars, nLowCents, nHighCents, nCents, nAmount

nDollars = This.RandInt(INT(m.tnLow), INT(m.tnHigh))

nLowCents = 0
nHighCents = 99

IF m.nDollars = INT(m.tnLow)
   nLowCents = MOD(m.tnLow, 100) * 100
ENDIF

IF m.nDollars = INT(m.tnHigh)
   nHighCents = MOD(m.tnHigh, 100) * 100
ENDIF
nCents = This.RandInt(m.nLowCents, m.nHighCents)

nAmount = m.nDollars + m.nCents/100

RETURN m.nAmount
```

Two methods use GetMoneyAmount. GetPrice passes the nLowPrice and nHighPrice properties to generate the price of a product, while GetFreightCharge passes the nMinFreight and nMaxFreight properties to generate the shipping cost.

We need to connect Products to Suppliers and Orders to Customers, Employees, and Shippers. For each of those look-up tables, we have a Get method that calls RandRecord to grab the primary key from a random record from the specified table. GetShipper is shown in **Listing 34**; the others are analogous.

Listing 34. GetShipper is one of a set of methods that adds the primary key from a look-up table to the record we're building.

```
PROTECTED PROCEDURE GetShipper
* Choose a random shipper
LOCAL nRecs, cShipperID

nRecs = This.GetDataCount("Shippers")

cShipperID = This.RandRecord("Shippers", "ShipperID")

ADDPROPERTY(This.oRecord, "ShipperID", m.cShipperID)

RETURN
```

The most complex Get method in this class is GetOrderLines, shown in **Listing 35**, which creates a collection of lines for an order. First, it decides how many detail lines to generate for this order (using the nMinLines and nMaxLines properties). Then, for each, it chooses a random product, grabs the price of that product, generates a random quantity (using the nMinItems and nMaxItems properties) and decides whether a discount applies and, if so, how much, based on the nHasDiscount, nMinDiscount, and nMaxDiscount properties. The code then creates a record and adds the necessary properties for the detail line, and then adds that detail line object to the collection. After the whole set of detail lines has been generated, it adds the collection to the record.

Listing 35. GetOrderLines builds a collection of detail lines for an order and adds the collection to the record.

```
PROTECTED PROCEDURE GetOrderLines
* Generate order lines for an order and put them
* into a collection.

LOCAL oLines, nLines, nLine
LOCAL iProdID, nPrice, nQty, nDisc
LOCAL nRand

nLines = This.RandInt(This.nMinLines, This.nMaxLines)

oLines = CREATEOBJECT("Collection")

FOR nLine = 1 TO m.nLines
   iProdID = This.RandRecord("Products", "ProductID")
   * RandRecord leaves us on the record, so grab unitprice
   nPrice = Products.UnitPrice
   **** TO DO: 15-July-2024 by TEG
   * Vary prices?

   nQty = This.RandInt(This.nMinItems, This.nMaxItems)
```

```
   nRand = RAND()
   IF m.nRand <= This.nHasDiscount
      nDisc = This.RandInt(This.nMinDiscount, This.nMaxDiscount)/100
   ELSE
      nDisc = 0
   ENDIF

   oLine = CREATEOBJECT("Empty")
   ADDPROPERTY(m.oLine, "iProdID", m.iProdID)
   ADDPROPERTY(m.oLine, "nPrice", m.nPrice)
   ADDPROPERTY(m.oLine, "nQty", m.nQty)
   ADDPROPERTY(m.oLine, "nDisc", m.nDisc)

   oLines.Add(m.oLine)

ENDFOR

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

RETURN
```

I could have chosen to wait to generate the detail lines in the AfterMakeSet method of MakeNorthwindSet, but this felt more natural and allowed Orders and OrderDetails to be treated as a single data type.

MakeNorthwindSet has a lot less custom code in it than MakeNorthwind. The OpenTables method opens the Northwind database and the tables we want to populate. SetData (**Listing 36**) sets up our six data types and specifies how many records to create for each.

Listing 36. MakeNorthwindSet's SetData method calls AddDataType for each of the six data types we want to create.

```
PROCEDURE SetData

This.AddDataType("Customer", 95)
This.AddDataType("Employee", 20)
This.AddDataType("Shipper", 5)
This.AddDataType("Supplier", 37)
This.AddDataType("Product", 115)
* The Order data type will generate both header and line data
This.AddDataType("Order", This.nSetSize)

RETURN
```

There are two methods that assist in saving the generated records to the tables. For historical reasons, the primary key for the Customers table is a 5-character string that, ideally, is related to the company name (rather than an auto-generated integer). The GetCompanyID method (see **Listing 37**) generates that 5-character string, ensuring that it isn't already used in the records already in the table. It first tries using the first 5 alphabetic characters from the company name. If that fails, it generates a random 5-letter string and repeats that until it finds a unique choice.

Listing 37. GetCompanyID attempts to create a unique 5-character string from the company name, but just creates a random 5-character string if that doesn't work.

```
PROTECTED PROCEDURE GetCompanyID(tcCompanyName)
* Generate a 5-char ID for a company. If possible,
* use the first 5 of their name.

LOCAL cCompName, cLetter, cID
LOCAL nLetter, lSuccess

cCompName = m.tcCompanyName

lSuccess = .F.

DO WHILE NOT m.lSuccess
   cID = ''

   IF EMPTY(m.cCompName)
      FOR nLetter = 1 TO 5
         cID = m.cID + This.oRand.RandLetter()
      ENDFOR
   ELSE
      LOCAL nPos
      nPos = 1

      DO WHILE LEN(m.cID) < 5 AND m.nPos <= LEN(m.cCompName)
         cLetter = SUBSTR(m.cCompName, m.nPos,1)
         IF ISALPHA(m.cLetter)
            cID = cID + m.cLetter
         ENDIF
         nPos = m.nPos + 1
      ENDDO
   ENDIF

   IF LEN(m.cID) < 5
      * Not able to get 5 chars from company name. Make it random.
      cCompName = ''

   ELSE
      SELECT CustomerID ;
         FROM Customers ;
         WHERE CustomerID == m.cID ;
         INTO CURSOR csrID

      IF RECCOUNT("csrID") > 0
         * Not unique. Make it random
         cCompName = ''
      ELSE
         lSuccess = .T.
      ENDIF

      USE IN SELECT("csrID")
   ENDIF
ENDDO
```

```
RETURN m.cID
```

I considered doing this work in the GetCompanyName method, but multiple tables in Northwind need company names and only Customers needs this unique character ID. So, it seemed better to do it at save time.

MakeNorthwind's GetPhoneNumber method creates an object with properties for the area code, number, and extension. That provides flexibility, so it can be used whether each is stored separately in a table, or the whole phone number is stored in a single field. Northwind uses a single field for each phone number, so the PhoneObjToNumber method converts the data in a phone number object to a string in the (reasonably) standard North American form "(999) 999-9999"; it's shown in **Listing 38**. (None of the tables in Northwind has room for an extension in the phone number field, so the nHasExtension property of MakeNorthwind is set to 0.)

Listing 38. PhoneObjToNumber assembles the properties in the phone number object created by GetNorthwind into a single character string.

```
PROCEDURE PhoneObjToNumber(toPhoneObj)

LOCAL cPhoneNumber

cPhoneNumber = "(" + toPhoneObj.AreaCode + ") " + toPhoneObj.Number


RETURN m.cPhoneNumber
```

For Northwind, we generate a different record for each data type. So, the SaveRecord method is a giant CASE statement with one case for each data type. In addition to using the GetCompanyID and PhoneObjToNumber method, it does some work to assemble various fields, such as the ContactName in Customers and Shippers.

Listing 39. SaveRecord has a case for each data type.

```
PROCEDURE SaveRecord(toRecord, tcType)

DO CASE
CASE m.tcType = 'Customer'
   LOCAL cCustID
   cCustID = This.GetCompanyID(toRecord.CompanyName)

     INSERT INTO Customers ;
      (CustomerID, CompanyName, ContactName, ContactTitle, ;
       Address, City, Region, PostalCode, Country, ;
       Phone, Fax) ;
      VALUES ( ;
       m.cCustID, ;
       toRecord.CompanyName, ;
       toRecord.cFirst - (' ' + toRecord.cLast), ;
       toRecord.JobTitle, ;
       toRecord.Address.Street, ;
       toRecord.Address.City, ;
```

```
       toRecord.Address.State, ;
       toRecord.Address.Zip, ;
       "USA", ;
       This.PhoneObjToNumber(toRecord.Phone), ;
       This.PhoneObjToNumber(toRecord.Fax))

CASE m.tcType = 'Employee'
   INSERT INTO Employees ;
      (LastName, FirstName, Title, ;
       BirthDate, HireDate, ;
       Address, City, Region, PostalCode, Country, ;
       HomePhone) ;
      VALUES ( ;
       toRecord.cLast, ;
       toRecord.cFirst, ;
       toRecord.JobTitle, ;
       toRecord.BirthDate, ;
       toRecord.HireDate, ;
       toRecord.Address.Street, ;
       toRecord.Address.City, ;
       toRecord.Address.State, ;
       toRecord.Address.Zip, ;
       "USA", ;
       This.PhoneObjToNumber(toRecord.Phone))

CASE m.tcType = 'Shipper'
   INSERT INTO Shippers ;
     (CompanyName, Phone) ;
     VALUES ( ;
       toRecord.CompanyName, ;
       This.PhoneObjToNumber(toRecord.Phone))

CASE m.tcType = 'Supplier'
   INSERT INTO Suppliers ;
     (CompanyName, ContactName, ContactTitle, ;
       Address, City, Region, PostalCode, Country, ;
       Phone, Fax, Homepage) ;
      VALUES ( ;
       toRecord.CompanyName, ;
       toRecord.cFirst - (' ' + toRecord.cLast), ;
       toRecord.JobTitle, ;
       toRecord.Address.Street, ;
       toRecord.Address.City, ;
       toRecord.Address.State, ;
       toRecord.Address.Zip, ;
       "USA", ;
       This.PhoneObjToNumber(toRecord.Phone), ;
       This.PhoneObjToNumber(toRecord.Fax), ;
       toRecord.URL)

CASE m.tcType = 'Product'
   INSERT INTO Products ;
     (ProductName, SupplierID, UnitPrice, ;
      UnitsInStock, UnitsOnOrder, ReorderLevel, ;
      Discontinued) ;
```

```
     VALUES ( ;
      toRecord.ProductName, ;
      toRecord.SupplierID, ;
      toRecord.Price, ;
      toRecord.InStock, ;
      toRecord.OnOrder, ;
      toRecord.ReorderLevel, ;
      toRecord.Discontinued)

CASE m.tcType = 'Order'
   * Need to save header and child records
   * Currently not handling ShipName, etc.
   INSERT INTO Orders ;
     (CustomerID, EmployeeID, ;
      OrderDate, RequiredDate, ShippedDate, ;
      Shipvia, Freight) ;
     VALUES ( ;
      toRecord.CustomerID, ;
      toRecord.EmployeeID, ;
      toRecord.OrderDate, ;
      toRecord.ReqdDate, ;
      toRecord.ShipDate, ;
      toRecord.ShipperID, ;
      toRecord.Freight)

   * Grab the orderID
   LOCAL iOrderID
   iOrderId = Orders.OrderID

   FOR EACH oLine IN toRecord.oLines
      INSERT INTO OrderDetails ;
        (OrderID, ProductID, ;
         UnitPrice, Quantity, Discount) ;
        VALUES ( ;
         m.iOrderID, ;
         oLine.iProdID, ;
         oLine.nPrice, ;
         oLine.nQty, ;
         oLine.nDisc)
   ENDFOR

ENDCASE

RETURN
```

As with the other data generators, populating the self-referential ReportsTo field of Employees is a challenge. But the AfterMakeType method is up to it. It receives the data type as a parameter, so we can use IF or CASE to provide different code different data types. For Northwind, Employee is the only type that needs additional handling.

The code in **Listing 40** specifies that each manager should have between 2 and 6 people reporting to them. It chooses an employee at random to be the main boss and puts that employee's ID into a cursor (csrToHandle) and marks their record in Employees so that we

know it's handled. Then, it loops through the records in csrToHandle until the ReportsTo field of every record in Employees has been set. For each record it processes in csrToHandle (we exit before we reach the end), it determines how many "reports" that person should have and then one by one, finds records in Employees that don't yet have a manager (ReportsTo is 0), makes the person we're now handling that employee's manager and adds that employee to csrToHandle. Along the way, we keep a count of the number of employees yet to be assigned a manager, and when that count reaches 0, we stop. Finally, we go back to the main boss and set their ReportsTo field to 0, to indicate they have no manager.

Listing 40. Code in AfterMakeType lets us set up the self-referential ReportsTo field in Employees.

```
PROCEDURE AfterMakeType(tcType)
* Set up manager links for employees
* Easiest path is probably to choose
* a random employee to be the boss,
* then n random employees at the next
* level and so on until all are done.

IF m.tcType = 'Employee'
   LOCAL nMinReports, nMaxReports, nReports, nReport
   LOCAL iEmpID, nToAssign

   nMinReports = 2
   nMaxReports = 6

   CREATE CURSOR csrToHandle (EmployeeID I)

   iEmpID = This.GetRandRecord("Employees", "EmployeeID")
   INSERT INTO csrToHandle VALUES (m.iEmpID)

   * Mark this one as used. We'll come back later and clear it
   UPDATE Employees ;
      SET ReportsTo = -1 ;
      WHERE EmployeeId = m.iEmpID

   nToAssign = RECCOUNT("Employees") - 1

   LOCAL nCurRec

   SELECT csrToHandle
   SCAN WHILE m.nToAssign > 0
      iEmpID = csrToHandle.EmployeeID
      nCurRec = RECNO('csrToHandle')

      DO CASE
      CASE m.nToAssign < m.nMinReports
         * At the end, just assign the rest to whoever's on top
         nReports = m.nToAssign
      CASE m.nToAssign < m.nMaxReports
          nReports = This.oRand.RandInt(m.nMinReports, m.nToAssign)
      OTHERWISE
         nReports = This.oRand.RandInt(m.nMinReports, m.nMaxReports)
```

```
      ENDCASE

      FOR nReport = 1 TO m.nReports
         SELECT EmployeeID ;
            FROM Employees ;
            WHERE EMPTY(ReportsTo) ;
            INTO CURSOR csrLeft

         iReportEmpID = This.GetRandRecord("csrLeft", "EmployeeID")
         UPDATE Employees ;
            SET ReportsTo = m.iEmpID ;
            WHERE EmployeeID = m.iReportEmpID

         INSERT INTO csrToHandle VALUES (m.iReportEmpID)
         nToAssign = m.nToAssign - 1
      ENDFOR

      GO (m.nCurRec) IN csrToHandle
   ENDSCAN

    USE IN SELECT('csrLeft')
    USE IN SELECT('csrToHandle')

   * Now fix big boss
   UPDATE Employees ;
      SET ReportsTo = 0 ;
      WHERE ReportsTo = -1
ENDIF

RETURN
```

This code may seem complex, but it was necessary to create an actual hierarchy of employees with a single boss, everyone else reporting to someone in the hierarchy, and no loops in the hierarchy.

Also worth noting is that this code could also have gone into AfterMakeSet, where it wouldn't have needed to be bracketed with IF. However, since it affects and uses only a single table, it made more sense to me to put it in AfterMakeType. Among other things, that means it won't run if we don't include the Employee data type for some reason.

There are some data items in Northwind I chose not to handle here. That includes putting shipping information into Orders, occasionally setting the item price in a detail line to something other than the price for that item in Products, and providing a "title of courtesy" (Mr., Mrs., etc.) and a phone extension for Employees. That was generally not because my classes couldn't handle those, but because at some point, I had to decide that "shipping is a feature" and I'd done enough to demonstrate that these classes work. I made similar choices with each of the products demonstrated in this paper.

FoxFaker

FoxFaker is a VFPX project (created by Irwin Rodríguez) intended to help create data for unit testing with FoxUnit and FoxMock. However, it can also be used with a little additional code to populate a database. You can get it by going to the VFPX site and downloading from its project-specific page (https://github.com/Irwin1985/FoxFaker).

To use FoxFaker, you instantiate a single object and call its methods. There's an excellent ReadMe file with the project that lists all the methods, showing an example of the kind of data each produces. They're grouped into "providers," related sets of methods.

All the method names begin with "fake" and then name the thing they create data for. For example, fakeState returns the name of a US state, while fakeJobTitle returns a job title.

A few of the methods accept one or more parameters to limit the generated results in some way. For example, the fakeWords method accepts a parameter to specify how many words to generate. One method I used extensively was fakeNumberBetween, which accepts a low value and a high value and returns an integer between the specified values. ("Between" includes both endpoints as possible return values.)

The ReadMe file suggests that the set of methods available and their return values depend on your system locale. That is, I got the results I saw because my computer is set to "US English." However, there's no indication what other languages/locales are supported.

Once you've downloaded and unzipped FoxFaker, getting started is easy. If you're working in a folder other than the one where FoxFaker was installed, SET PATH to that folder, and then instantiate the FoxFaker object, as in **Listing 41**. Note that the documentation is incorrect on this point. It says you need to SET PROCEDURE to the PRG rather than SET PATH to the folder.

Listing 41. To start using FoxFaker, just SET PATH and instantiate.

```
SET PATH TO "D:\FOX\VFPX\FOXFAKER\FOXFAKER-MASTER\"
oFaker = NEWOBJECT("FoxFaker", "D:\Fox\VFPX\FoxFaker\FoxFaker-master\FoxFaker.prg")
```

Once you've instantiated the object, just call methods to return the values you need. **Listing 42** shows a few calls and the values they returned. (Of course, calling those methods again is likely to return different results.)

Listing 42. Each call to a FoxFaker method returns a test date item.

```
oFaker.fakeName('male')   && Lionel Klein
oFaker.fakeAddress()      && 6122 Lebsack Hill Suite 099 Michelshire, SD 84299-3873
oFaker.fakeURL()      && http://orn.net/totam-debitis-voluptatem-earum-ea-
aliquid.html
oFaker.fakeDate()   && 07/19/1979
```

To generate data for the simplified Northwind database, I wrote a program (included in the FoxFaker folder of the materials for this session as FakeNorthwind.PRG) that uses a loop to populate each table.

I generated the tables in the same order I used with each of tools, handling the look-up tables Customers, Employees, Shippers, and Suppliers first, then generating Products records, and finally generating Orders and OrderDetails in parallel. I found that FoxFaker could handle most of my needs with direct calls. But there were a few places where I had to write a little more code.

One field that needed extra attention was Customers' CustomerID. I needed to ensure that each 5-character string was unique. I could have taken an approach similar to what I used with my own generator, attempting to create a string from the first five alphabetic characters in the generated company name, but for simplicity, I just generated a random five-character string (using the fakeRandomLetter method) and then tested it for uniqueness in the table so far.

Listing 43. This code uses FoxFaker to populate the Customers table.

```
LOCAL cCustID, nChar, lIsNew

USE Customers

FOR nRec = 1 TO 95
   lIsNew = .F.
   DO WHILE NOT m.lIsNew
      cCustID = ''

      FOR nChar = 1 TO 5
         cCustID = m.cCustID + oFaker.fakeRandomLetter()
      ENDFOR

      lIsNew = NOT SEEK(m.cCustID, 'Customers', 'CustomerID')
   ENDDO

   INSERT INTO Customers ;
      (CustomerID, CompanyName, ContactName, ContactTitle, ;
       Address, City, Region, PostalCode, ;
       Country, Phone, Fax) ;
      VALUES ;
      (m.cCustID, ;
       oFaker.fakeCompany(), ;
       oFaker.fakeName(), ;
       oFaker.fakeJobTitle(), ;
       oFaker.fakeStreetAddress(), ;
       oFaker.fakeCity(), ;
       oFaker.fakeState(), ;
       ALLTRIM(oFaker.fakePostcode()), ;
       "USA", ;
       oFaker.fakePhoneNumber(), ;
       oFaker.fakePhoneNumber())
ENDFOR
```

The only other complication for Customers is that, for the US, the fakePostcode returns some five-digit zip codes and some 10-digit zips in the Zip+4 format. The method always returns a right-aligned 10-character string, even for a five-digit zip code. Northwind expects left-aligned postal codes, so I wrapped the call to fakePostcode (here and for other tables) with ALLTRIM().

Employees also presented a complication. The fakeDate method can return future dates, not just past dates. Not surprisingly, Employees has a field rule to reject future dates in the BirthDate field. I also wanted to be sure that only people 18 and older were hired (that is, that only birthdates at least 18 years ago were chosen and that the hire date for an employee was at least 18 years after the birthdate).

FoxFaker offered the easiest option for populating the Employees Notes field of anything I tested. The fakeText method generates a string of the specified length. I could have first used fakeNumberBetween to decide how much text to generate for each employee and passed that value to fakeText.

Listing 44 shows the code to populate the Employees table. I chose not to populate the ReportsTo field here, but the code I used for that in my generator could easily be adapted to use here.

Listing 44. The only complication in generating Employees was making sure the company hired only adults.

```
LOCAL dBirthDate, dHireDate

FOR nRec = 1 TO 20
   * Generate dates in advance, so we can screen for sensibleness
   dBirthDate = oFaker.fakeDate()
   DO WHILE dBirthDate > GOMONTH(DATE(), -18 * 12)
      dBirthDate = oFaker.fakeDate()
   ENDDO

   dHireDate = oFaker.fakeDate()
   DO WHILE dHireDate < GOMONTH(dBirthdate, 18 * 12) OR dHireDate > DATE()
      dHireDate = oFaker.fakeDate()
   ENDDO


   INSERT INTO Employees ;
      (LastName, FirstName, Title, ;
       TitleOfCourtesy, BirthDate, HireDate, ;
       Address, City, Region, PostalCode, ;
       Country, HomePhone, Notes) ;
      VALUES ;
      (oFaker.fakeLastName(), ;
       oFaker.fakeFirstName(), ;
       oFaker.fakeJobTitle(), ;
       oFaker.fakeTitle(), ;
       dBirthDate, ;
       dHireDate, ;
       oFaker.fakeStreetAddress(), ;
```

```
       oFaker.fakeCity(), ;
       oFaker.fakeState(), ;
       ALLTRIM(oFaker.fakePostcode()), ;
       "USA", ;
       oFaker.fakePhoneNumber(), ;
       oFaker.fakeText(100))
ENDFOR
```

Both Shippers and Suppliers were straightforward with no complications. Products was the first table that needed a primary key from another table (Suppliers). FoxFaker doesn't offer a method to do this, but I quickly realized I could do it using fakeNumberBetween to choose a random record in the table. Because I needed the same technique for multiple fields in Orders and OrderDetails, I wrote a function to accept a table name and field name, and return the value of the specified field in a randomly-chosen record of the specified table. The function, called GetRandRecord, is shown in **Listing 45**. It opens the specified table and gets its record count. Then, it uses fakeNumberBetween to choose a random record, moves to the chosen record, and returns the value of the specified field.

Listing 45. To populate Northwind, we need a way to get the primary key of a randomly-chosen record.

```
PROCEDURE GetRandRecord(tcTable, tcField)
* Choose a random record in the specified table
* and return the specified field from that record.

LOCAL nRecCount

SELECT 0
USE (m.tcTable) AGAIN ALIAS __Table
nRecCount = RECCOUNT("__Table")

LOCAL nRecNo, uReturn
nRecNo = oFaker.fakeNumberBetween(1, m.nRecCount)

GO (m.nRecNo) IN __Table
uReturn = EVALUATE("__Table." + m.tcField)

USE IN SELECT("__Table")

RETURN m.uReturn
```

To generate product names, I used fakeNumberBetween to decide how many words should be in the name and passed that value to the fakeWords method.

Both Products and Orders need a money amount. I used the same approach for each of them, calling fakeNumberBetween once to get the dollar amount and again, passing 0 and 99, for the cents. I then divided the second value by 100 and added the two together.

Listing 46 shows the code to populate Products.

Listing 46. Generating Products with FoxFaker was a little more complicated than the look-up tables.

```
FOR nRec = 1 TO 115
```

```
   LOCAL nNameWords, nSupplierRec, iSupplierID, nPrice

   * First decide how many words in product name
   nNameWords = oFaker.fakeNumberBetween(1, 3)

   * Now, pick a supplier
   iSupplierID = GetRandRecord("Suppliers", "SupplierID")

   * Build a price with dollars and cents
   nPrice = oFaker.fakeNumberBetween(0, 19) + oFaker.fakeNumberBetween(0, 99)/100

   INSERT INTO Products ;
      (ProductName, SupplierID, ;
       UnitPrice, UnitsInStock, ;
       UnitsOnOrder, ReorderLevel, ;
       Discontinued) ;
      VALUES ;
      (oFaker.fakeWords(m.nNameWords), ;
       m.iSupplierID, ;
       m.nPrice, ;
       oFaker.fakeNumberBetween(0, 1000), ;
       oFaker.fakeNumberBetween(0, 1000), ;
       oFaker.fakeNumberBetween(0, 100), ;
       oFaker.fakeBoolean())
ENDFOR
```

Finally, I handled Orders and OrderDetails together, as with my own generator. Each order has three relevant dates: order date, required date, and shipped date. I used fakeDate to get the order date and then fakeNumberBetween to convert it to required date and shipped date.

I used fakeNumberBetween to decide how many detail lines each order had. After calling GetRandRecord to return a product ID, I used a query to retrieve the unit price for the chosen product.

The code to generate these two tables is shown in **Listing 47**.

Listing 47. While the code to generate Orders and OrderDetails is a little long, it mostly uses methods and techniques we've already seen.

```
LOCAL cCustID, iEmpID, iShipperID, dOrderDate

FOR nRec = 1 TO 1000
   cCustId = GetRandRecord("Customers", "CustomerID")
   iEmpID = GetRandRecord("Employees", "EmployeeID")
   iShipperID = GetRandRecord("Shippers", "ShipperID")

   dOrderDate = oFaker.fakeDate()
   DO WHILE dOrderDate > DATE()
      dOrderDate = oFaker.fakeDate()
   ENDDO

   INSERT INTO Orders ;
```

```
      (CustomerID, EmployeeID, OrderDate, ;
       RequiredDate, ShippedDate, ShipVia, ;
       Freight) ;
      VALUES ;
      (m.cCustID, ;
       m.iEmpID, ;
       m.dOrderDate, ;
       m.dOrderDate + oFaker.fakeNumberBetween(1, 30), ;
       m.dOrderDate + oFaker.fakeNumberBetween(1, 60), ;
       m.iShipperID, ;
       oFaker.fakeNumberbetween(0, 1111) + oFaker.fakeNumberBetween(0, 99)/100)

   LOCAL nLines, nLine, iProdID, nUnitPrice
   nLines = oFaker.fakeNumberBetween(1, 25)


   FOR nLine = 1 TO m.nLines
      iProdID = GetRandRecord("Products", "ProductID")
      SELECT UnitPrice ;
         FROM Products ;
         WHERE ProductID = m.iProdID ;
         INTO CURSOR csrUnitPrice
      nUnitPrice = csrUnitPrice.UnitPrice
      USE IN SELECT("csrUnitPrice")

      INSERT INTO OrderDetails ;
         (OrderID, ProductID, ;
          UnitPrice, Quantity, Discount) ;
         VALUES ;
         (Orders.OrderID, ;
          m.iProdID, ;
          m.nUnitPrice, ;
          oFaker.fakeNumberBetween(1, 100), ;
          oFaker.fakeNumberBetween(0,25)/100)
   ENDFOR

ENDFOR
```

Overall, once I figured out that I needed SET PATH, working with FoxFaker was easy. I'd love to see more methods covering more different kinds of data, but as the examples here show, I was able to generate almost everything in the simplified Northwind database with only a little code beyond calls to FoxFaker's methods.

The Bottom Line

All of the tools explored in this paper are capable of generating most of the kinds of data needed for a solid test data set. The effort required varies both with the tool and with your requirements. While my test data generator likely requires the greatest effort, it also provides the most flexibility and the greatest opportunity to expand the types of data you can generate. But you can't go wrong with any of these products.

Having a realistic set of test data makes almost every aspect of the development process easier. Test data should reflect the real data, including both everyday and extreme cases. Whether you create test data from production data, generate it with a commercial tool, or use VFP to generate it, once you get used to working with a good test data set, you'll wonder how you ever managed without one.