The Why and How of Test Data

Session 26

Tamar E. Granor
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Fax: 215-635-2234
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

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. According to www.Tech-Encyclopedia.com, test data is:

> *A set of data created for testing new or revised applications. Test data should be developed by the user as well as the programmer and must contain a sample of every category of valid data as well as many invalid conditions as possible.*

The Glossary at www.austin.cc.tx.us/audit/ offers this definition instead:

> *Data that is run through a computer program to test the software. Test data can be used to test compliance with controls in the software.*

www.Answers.com gives this definition:

> *A set of data developed specifically to test the adequacy of a computer run or system; the data may be actual data that has been taken from previous operations, or artificial data created for this purpose.*

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 Tech Encyclopedia 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 have a tendency 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 on 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. There's been plenty of coverage of personal data being leaked to the world over the last few years. 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 25 characters, having test data that's only 1 character as well as test data that fills all 25 characters lets you see whether forms and reports handle short as well as long values. While "John 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 (make sure 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.

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.

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

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 designed to generate test data.

The School database

I've created an example database to make it possible to actually test the various alternatives. The database, School.DBC in the session materials, represents a very simple student and courses set-up for a school running on semesters and is shown in Figure 1.

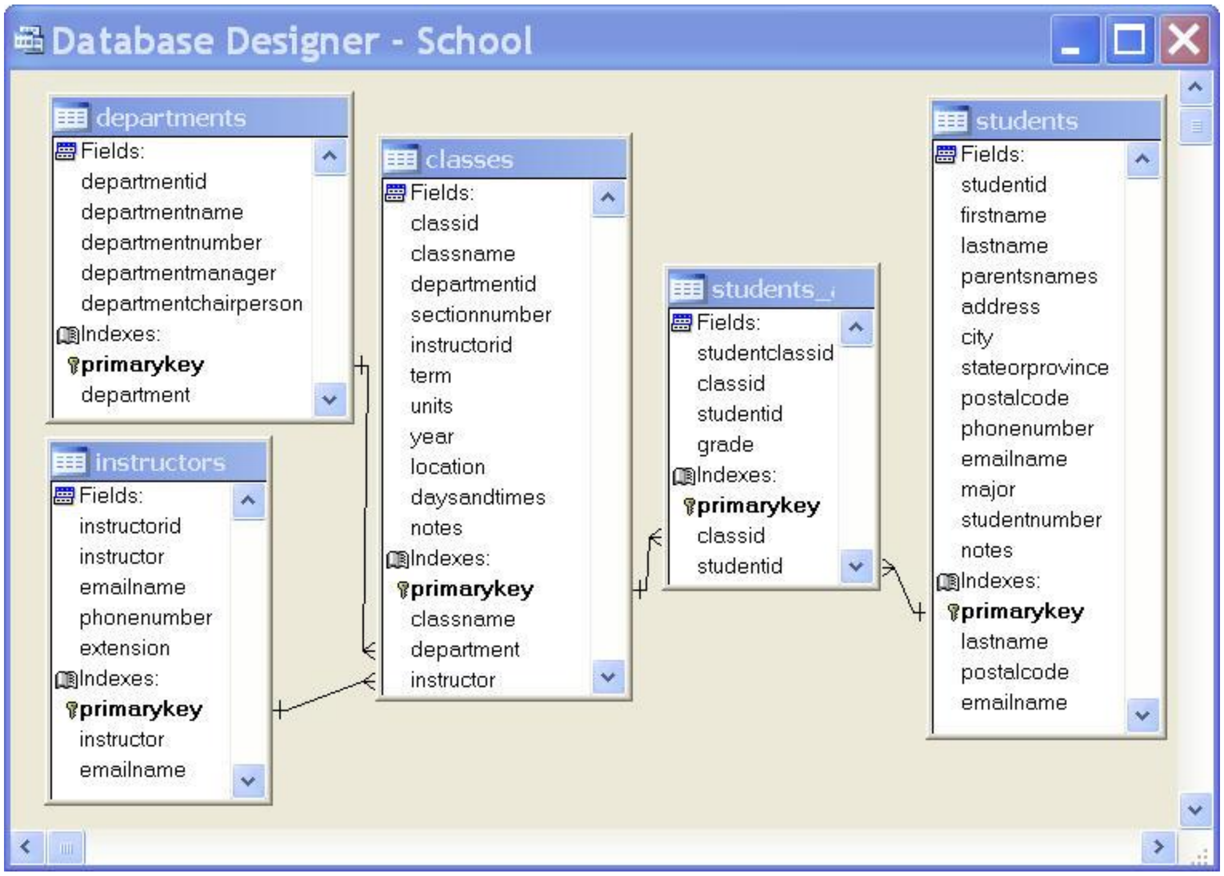


Figure 1. This database represents students and courses for a school. It was created using the VFP Database Wizard.

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.

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 opportunity 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. In addition, Visual Studio 2005 Team Edition includes a test generator; if you're using Visual Studio, this tool may be your best bet.

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 School database generated by each (except for GS Data Generator, which doesn't let you actually generate the data with the test version) is included in the session materials.

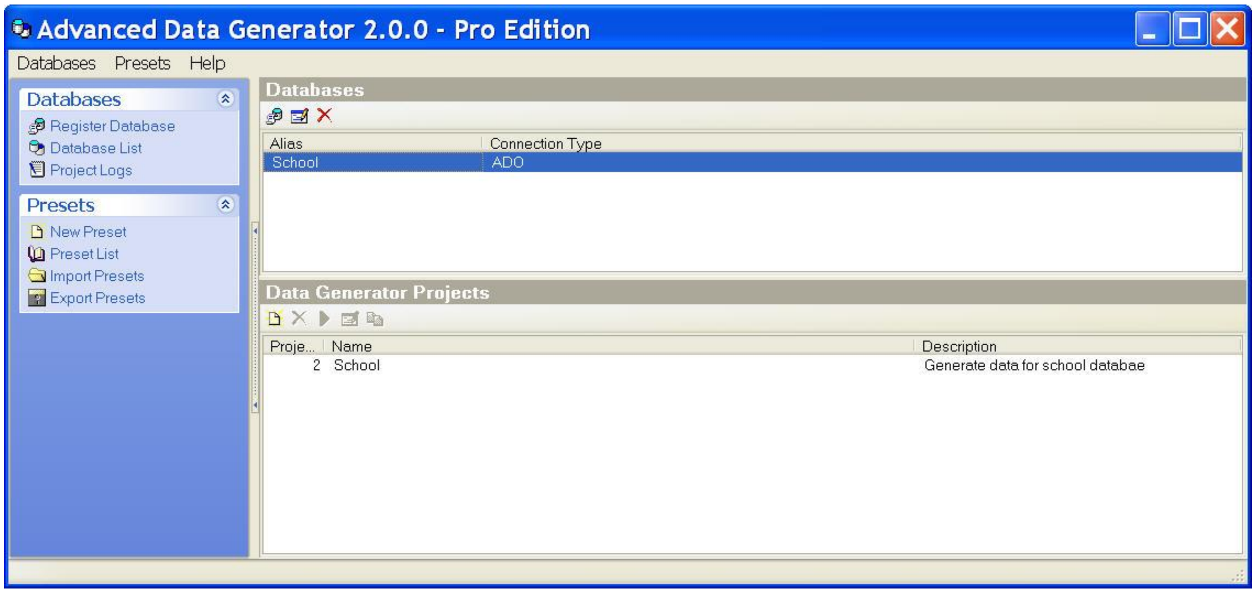
Advanced Data Generator

Advanced Data Generator (ADG) from Upscene Productions (www.upscene.com) can work with any ODBC or ADO data source. The Pro edition (which is the regular product) costs $188 per license for up to 4 licenses. An unlimited site license is $881. Upscene also offers less expensive versions that work with only a single database; currently, they have such versions for InterBase ($74 for one developer; $339 for a site license), Firebird ($62/$276) and MySQL ($49/$226). (All prices are as of March, 2007.) A free trial version is available at their website. The trial version has a 30-day time limit.

ADG is organized in 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.

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.

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. Figure 2 shows the main workspace in ADG, with databases at the top and projects on the bottom.

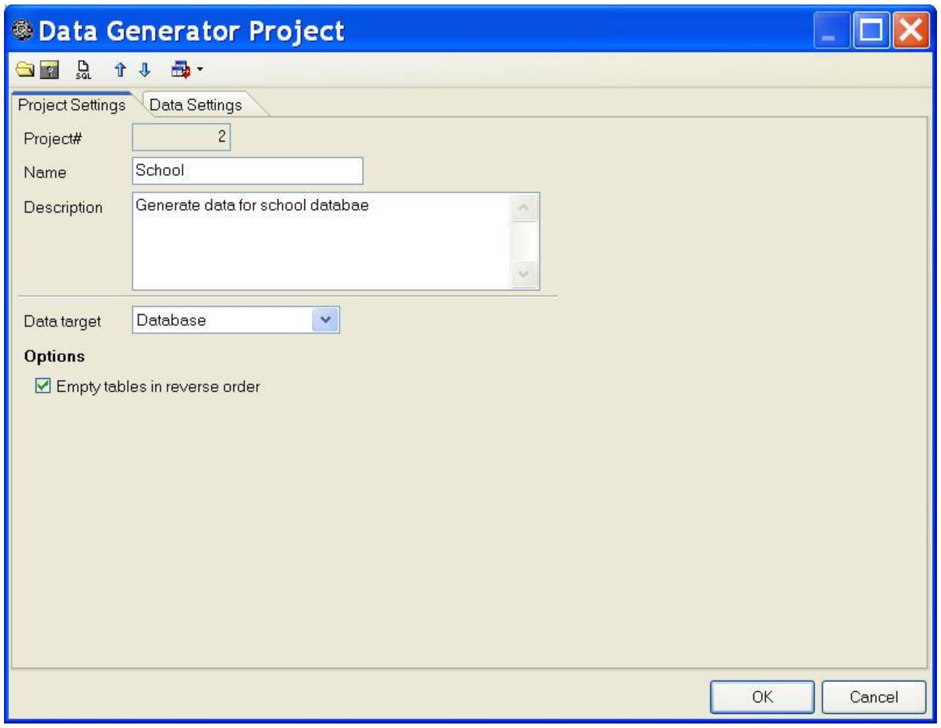


Alias	Connection Type
School	ADO

Proje...	Name	Description
2	School	Generate data for school databae

Figure 2. Advanced Data Generator looks at the world in terms of databases and projects.

To create a project for the selected database, click the New button in the Data Generator Projects section. The Data Generator Project dialog opens, set to the Project Settings tab (Figure 3). Use this tab to specify a name and description for your project. There are two additional options here. First, you can specify where the generated data goes. The default is to put it right in the database, but you can also create SQL scripts and CSV files, as well as a couple of more esoteric formats.

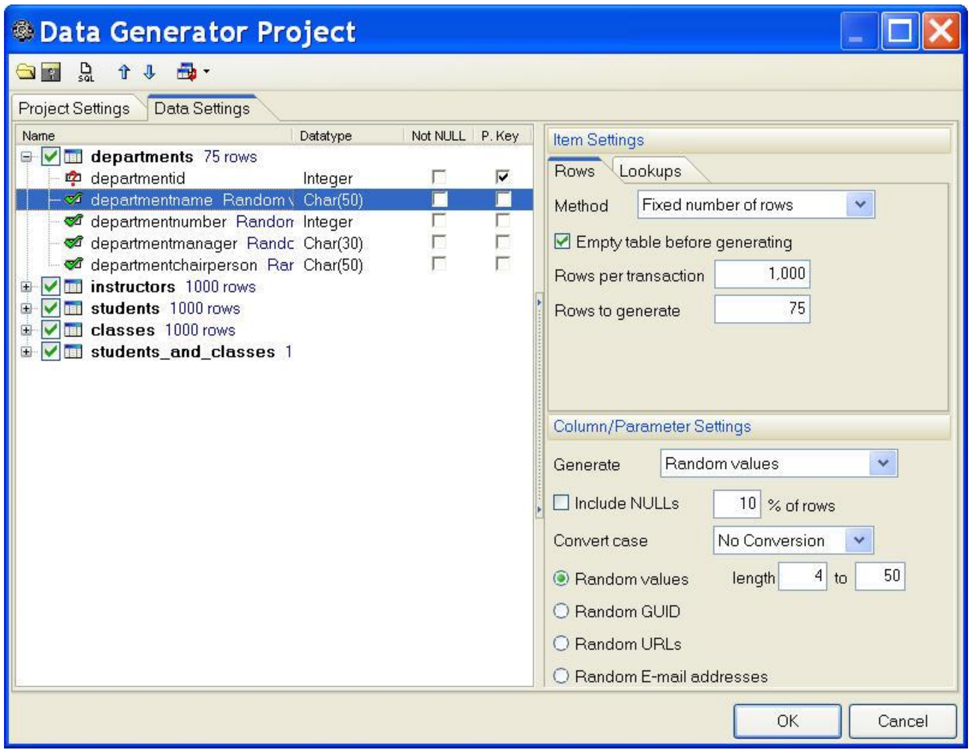


Project# 2
Name School
Description Generate data for school databae
Data target Database
Options
☑ Empty tables in reverse order

Figure 3. You start creating a project in ADG by specifying a name 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.

Once you specify the project-level information, you use the Data Settings tab (Figure 4) to specify how to actually generate the data. For each table in the database, you can determine whether to generate data and how many records to generate. There are a number of choices for generating data for each field; the list varies based on the data type.


Data Generator Project

Project Settings | Data Settings

Name	Datatype	Not NULL	P. Key
✓ **departments** 75 rows			
departmentid	Integer	☐	✓
departmentname Random v	Char(50)	☐	☐
departmentnumber Randon	Integer	☐	☐
departmentmanager Randc	Char(30)	☐	☐
departmentchairperson Rar	Char(50)	☐	☐
✓ **instructors** 1000 rows			
✓ **students** 1000 rows			
✓ **classes** 1000 rows			
✓ **students_and_classes** 1			

Item Settings

Rows | Lookups

Method: Fixed number of rows

✓ Empty table before generating

Rows per transaction: 1,000

Rows to generate: 75

Column/Parameter Settings

Generate: Random values

☐ Include NULLs 10 % of rows

Convert case: No Conversion

● Random values length 4 to 50

○ Random GUID

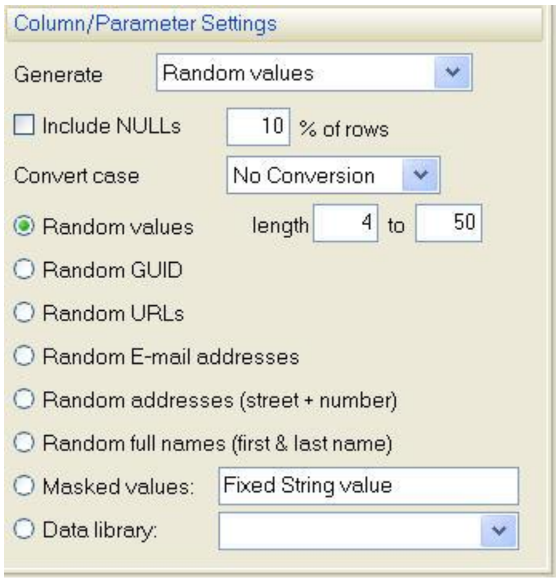
○ Random URLs

○ Random E-mail addresses

OK | Cancel

Figure 4. ADG's Data Settings tab lets you specify the order of data generation and the data to be generated for each field.

Character fields offer the widest range of options and the list in Figure 4 is misleading (though the figure shows the way this dialog opens every time). Expanding the dialog shows additional options (Figure 5). The various Random items are self-explanatory. The Masked values option lets you specify a format to which items must conform (analogous to an inputmask). The Data library option lets you use random items from provided lists of first and last names (both English and Spanish, with the option of using only male or only female), city names or street names from a number of countries, US states, countries and even companies.


Column/Parameter Settings

Generate: Random values

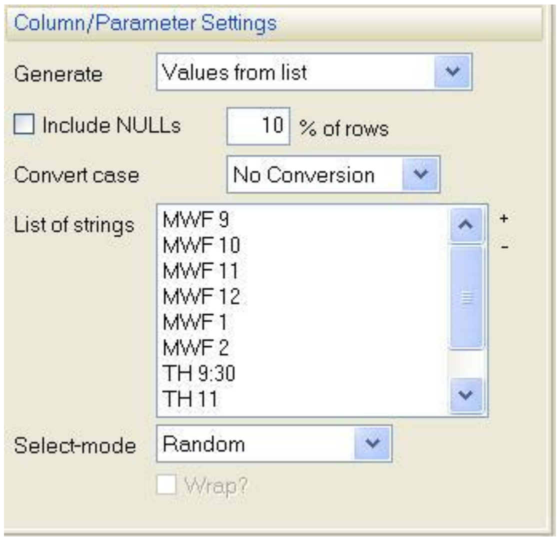
☐ Include NULLs 10 % of rows

Convert case: No Conversion

● Random values length 4 to 50
○ Random GUID
○ Random URLs
○ Random E-mail addresses
○ Random addresses (street + number)
○ Random full names (first & last name)
○ Masked values: Fixed String value
○ Data library:

Figure 5. Character fields can be based on a wide variety of sources in ADG.

The Generate dropdown lets you gather data from other than random sources, including a fixed list of items (Figure 6). You specify the contents of the list and how the list should be applied to your new records. Another choice in the Generate dropdown is a referential link; you specify the table and field the data comes from and whether links should be random, sequential or one-to-one. Yet another option is to draw the values from a text file.


Column/Parameter Settings

Generate: Values from list

☐ Include NULLs 10 % of rows

Convert case: No Conversion

List of strings:
MWF 9
MWF 10
MWF 11
MWF 12
MWF 1
MWF 2
TH 9:30
TH 11

Select-mode: Random

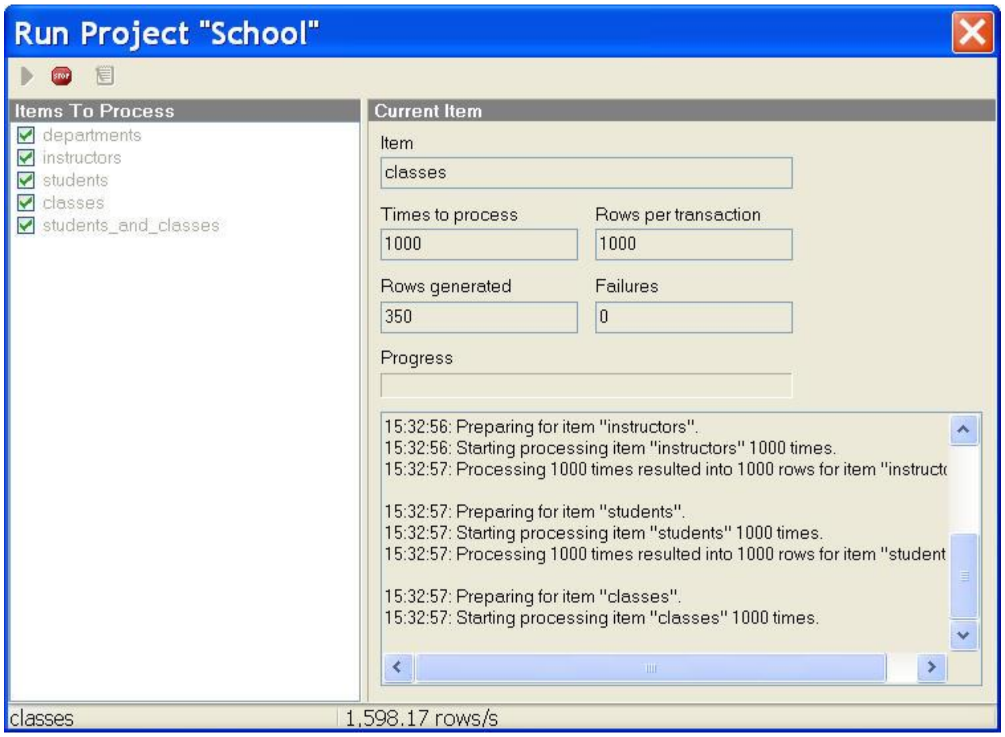
☐ Wrap?

Figure 6. Rather than generating field values randomly in ADG, they can be drawn from a specified list.

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.

One attractive feature of ADG is the ability to define "presets" for field types you use repeatedly. A preset is like any other field specification, but it's stored and named, so you can apply it repeatedly. ADG comes with a few presets, including one for US zip codes.

Once you've specified how data is to be generated, you run the project to actually generate the type of output indicated. The Run Project dialog (Figure 7) appears and shows progress as the data is generated.


Run Project "School"

Items To Process
- ✓ departments
- ✓ instructors
- ✓ students
- ✓ classes
- ✓ students_and_classes

Current Item
Item: classes
Times to process: 1000
Rows per transaction: 1000
Rows generated: 350
Failures: 0
Progress

```
15:32:56: Preparing for item "instructors".
15:32:56: Starting processing item "instructors" 1000 times.
15:32:57: Processing 1000 times resulted into 1000 rows for item "instructo

15:32:57: Preparing for item "students".
15:32:57: Starting processing item "students" 1000 times.
15:32:57: Processing 1000 times resulted into 1000 rows for item "student

15:32:57: Preparing for item "classes".
15:32:57: Starting processing item "classes" 1000 times.
```

classes 1,598.17 rows/s

Figure 7. ADG's Run Project dialog gives you feedback as the data is generated.

The biggest issues with ADG are in the user interface. As noted above, the Data Generator Project dialog opens at a size that doesn't show all the choices. Even after you've resized it, the next time you open it, it's back to the old size. The interface is unhelpful in other ways. For example, though the main form remembers the size you set, it doesn't remember what database and project you last selected.

In addition, 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.

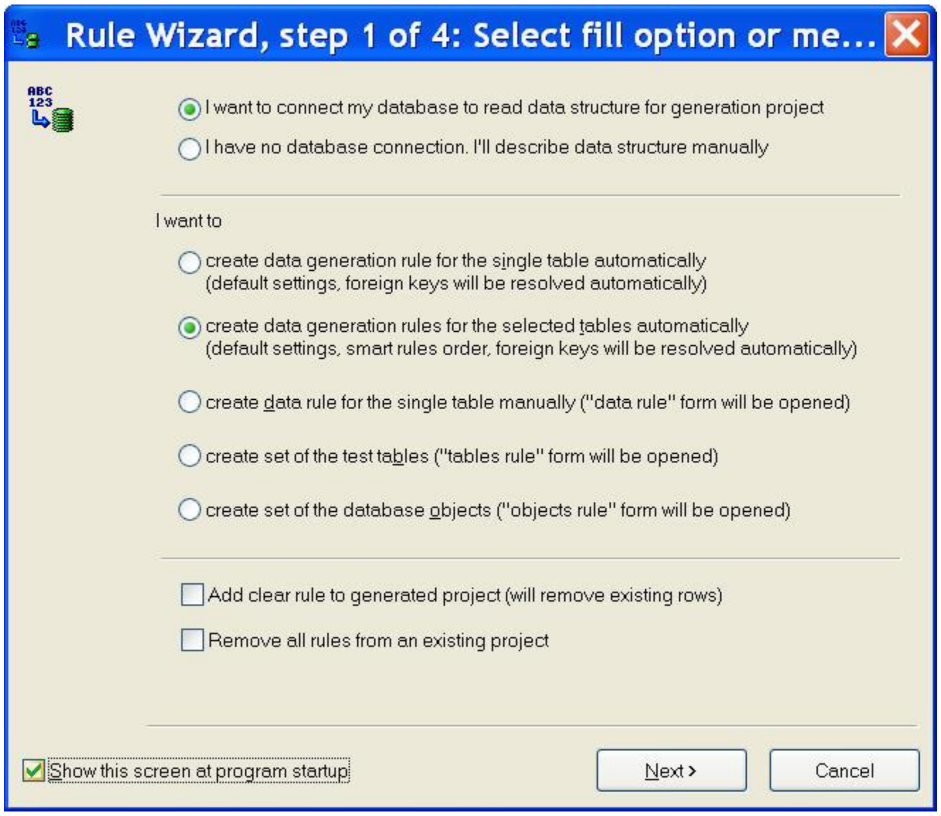
From a data standpoint, the most significant weakness is that random string values use the full character set and have no notion of words. It would be useful to be able to generate random strings of letters only or random strings of words.

DTM Data Generator

DTM Data Generator (DTM) from DTM soft (www.sqledit.com) supports ODBC, OLE DB and OCI (Oracle Call Interface) databases. There are two versions, Standard and Professional. For the Standard version, pricing starts at $159 for a single developer license, and includes a 3- license pack ($299), a 5-license pack ($449), a 10-license pack ($799), a site license ($1199) and a world license ($1999). For the professional version, a single developer license is $239. Bulk licenses run $449 for three and $599 for five. An annual support subscription, including updates, is $60 after the first year. A demo version is available for download; it works like the full product, but is limited to generating 10 records per table.

DTM is organized into projects and rules. A project refers to a particular database; you can create multiple projects for a single database, but only one project can be open at a time. Within a project, a series of rules determine how data is generated.

When you first open DTM, the Rule Wizard appears; Figure 8 shows the first page of the wizard. Here you specify whether there's a connection to the database available and where rule generation should start. If you indicate that a connection is available, the Connect to Database dialog (Figure 9) appears.


Rule Wizard, step 1 of 4: Select fill option or me...

[● I want to connect my database to read data structure for generation project]
○ I have no database connection. I'll describe data structure manually

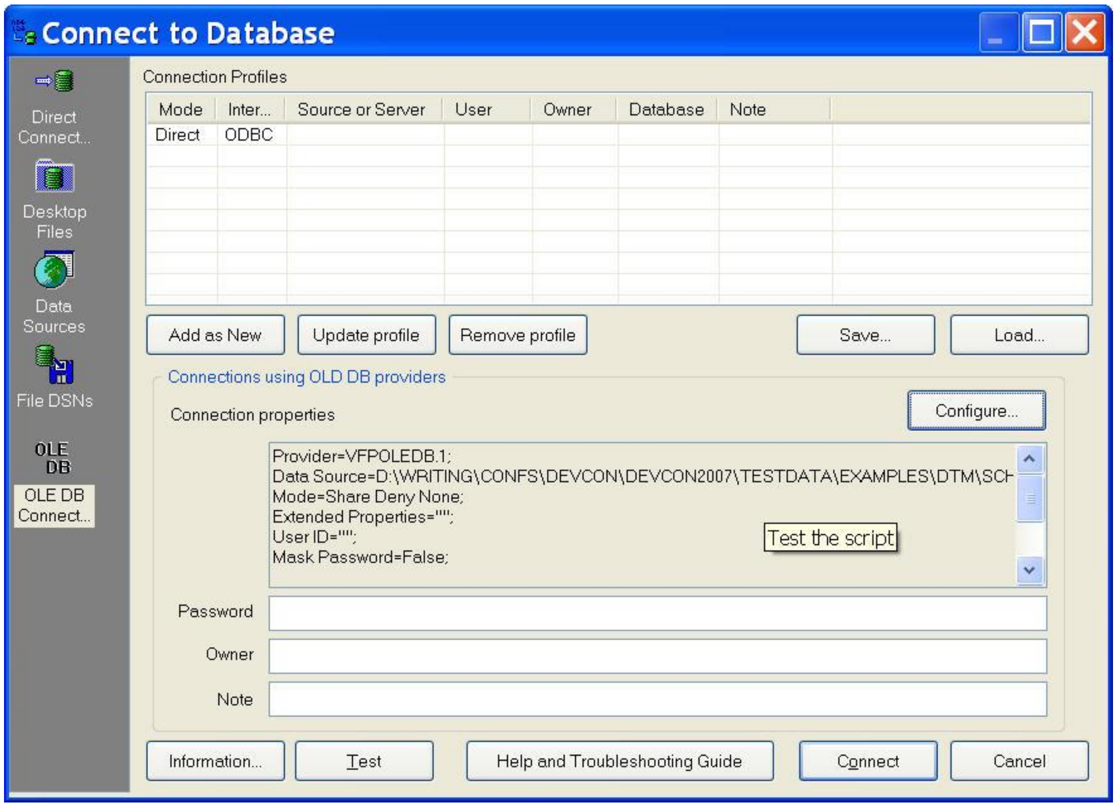
I want to

○ create data generation rule for the single table automatically (default settings, foreign keys will be resolved automatically)
[● create data generation rules for the selected tables automatically (default settings, smart rules order, foreign keys will be resolved automatically)]
○ create data rule for the single table manually ("data rule" form will be opened)
○ create set of the test tables ("tables rule" form will be opened)
○ create set of the database objects ("objects rule" form will be opened)

☐ Add clear rule to generated project (will remove existing rows)
☐ Remove all rules from an existing project

[✓ Show this screen at program startup]

Figure 8. The DTM Rule Wizard starts by determining how to connect to a database and where to start the generation process.



Mode	Inter...	Source or Server	User	Owner	Database	Note
Direct	ODBC					

```
Provider=VFPOLEDB.1;
Data Source=D:\WRITING\CONFS\DEVCON\DEVCON2007\TESTDATA\EXAMPLES\DTM\SCH
Mode=Share Deny None;
Extended Properties="";
User ID="";
Mask Password=False;
```

Figure 9. This dialog lets you set up a connection to a database in DTM. The bottom half of the dialog varies based on the selection in the left pane.

What appears once you specify a database connection depends on your choice in the middle part of Step 1. If you choose the default setting (shown in Figure 8), the second page of the Wizard (Figure 10) appears, to let you choose which tables will have data generated.

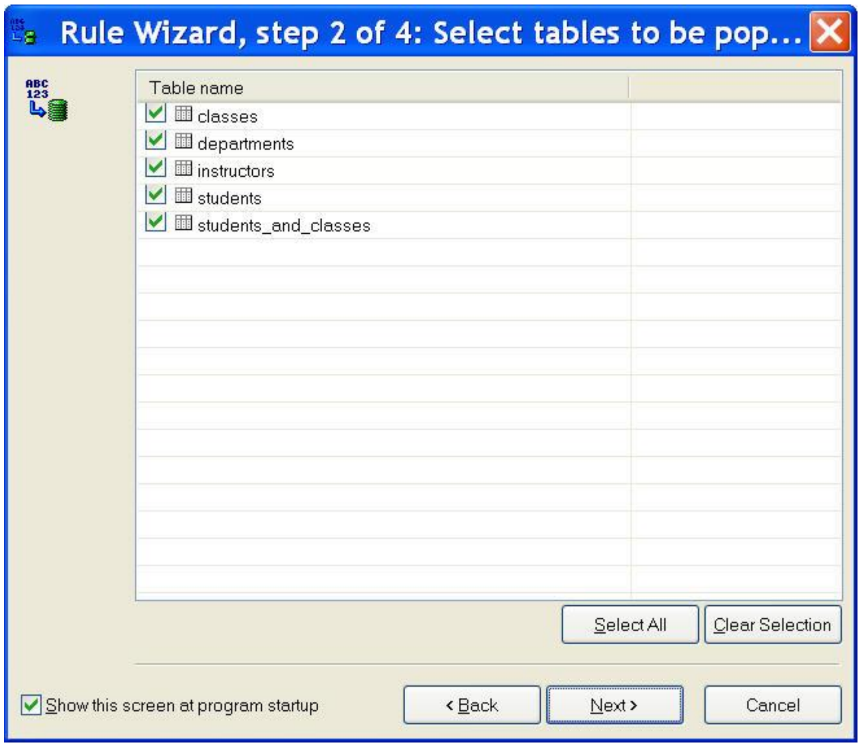

Rule Wizard, step 2 of 4: Select tables to be pop...
Select All
Clear Selection
Show this screen at program startup
< Back
Next >
Cancel

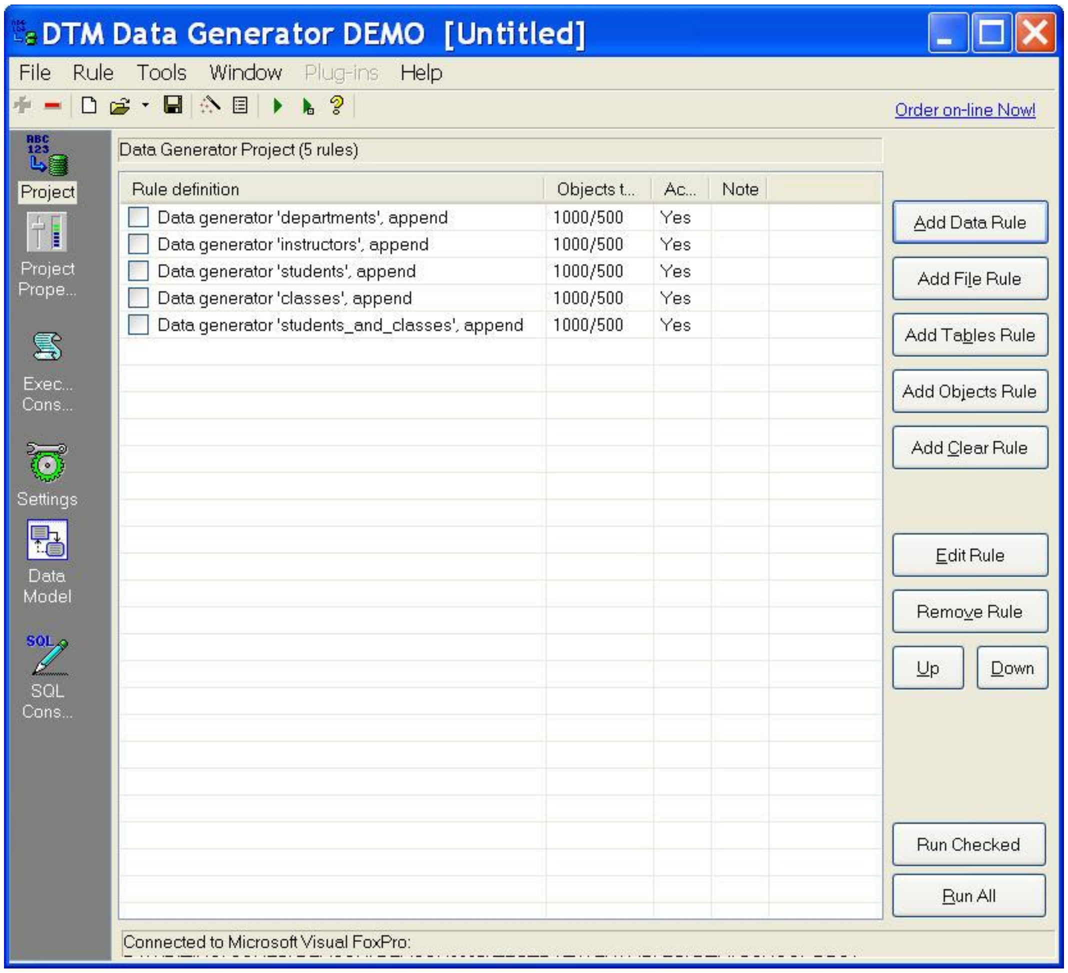
Table name	
✓ classes	
✓ departments	
✓ instructors	
✓ students	
✓ students_and_classes	

Figure 10. This step of the DTM's Rule Wizard lets you specify which tables in the database should have test data generated.

In the third step of the Wizard, you specify key settings, including the number of rows to generate for each table. (This is a default; you can change it for individual tables later.)

In the final step, you determine what should be generated. The choices include putting data directly into the tables, generating INSERT statements (a great way to ensure that you can recreate your test data at any time), generating text files and generating XML.

When you choose Finish, the wizard makes an attempt to generate the appropriate data rules, based on the field names and types in your database, as well as the relationships between tables. It does a remarkably good job. For the sample School database, it put the tables in the right order, so that records are created before their primary keys are needed as foreign keys. Figure 11 shows the results for the School database.


DTM Data Generator DEMO [Untitled]

File Rule Tools Window Plug-ins Help

Order on-line Now!

Data Generator Project (5 rules)

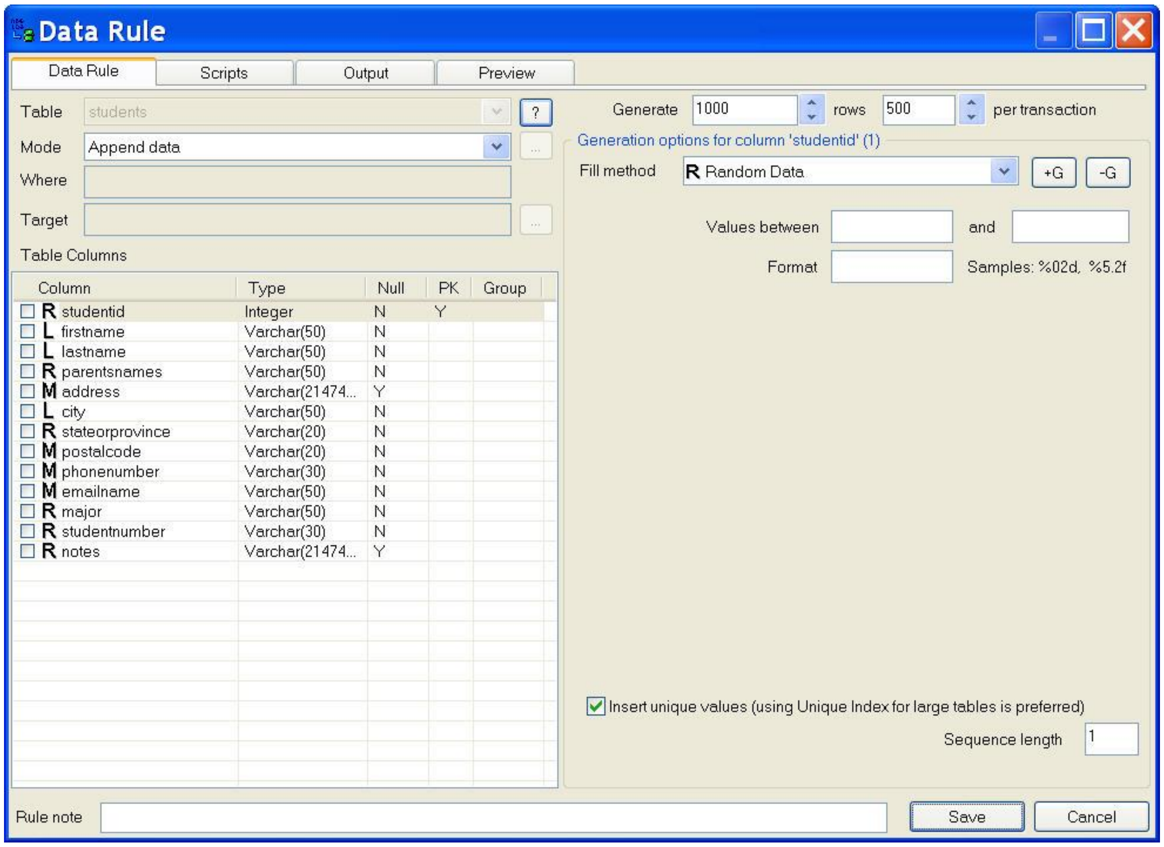
Rule definition	Objects t...	Ac...	Note
Data generator 'departments', append	1000/500	Yes	
Data generator 'instructors', append	1000/500	Yes	
Data generator 'students', append	1000/500	Yes	
Data generator 'classes', append	1000/500	Yes	
Data generator 'students_and_classes', append	1000/500	Yes	

Connected to Microsoft Visual FoxPro:

Figure 11. DTM's Rule Wizard generates an initial set of rules and attempts to put the tables in the appropriate order, using field names, types and relations as clues.

At this point, you can edit the generated rules and add your own. One rule you're likely to want is a "Clear Rule" that indicates that existing data should be deleted before generated new data. When you choose Add Clear Rule, a dialog lets you specify the order in which to empty the tables to avoid RI problems. As with the generation order, DTM guessed this order right for the School database.

To modify a rule, you highlight it and click Edit Rule (or simply double-click the rule). The Data Rule dialog opens. Figure 12 shows the initial set of rules for the Students table. The boldface symbol next to each field shows the fill method chosen; "R" indicates random data, "L" uses a "Values library" (a predefined list of values), "M" is a masked value, along the lines of a regular expression, and "I" says to ignore the field (necessary for VFP's auto-increment fields). You can also base a field on another table (in the same or another database), allowing you to create referential links, as well as generate values in order, and compute values based on constants or other fields.



Data Rule

Data Rule | Scripts | Output | Preview

Table: students

Mode: Append data

Where:

Target:

Table Columns

Column	Type	Null	PK	Group
☐ R studentid	Integer	N	Y	
☐ L firstname	Varchar(50)	N		
☐ L lastname	Varchar(50)	N		
☐ R parentsnames	Varchar(50)	N		
☐ M address	Varchar(21474...	Y		
☐ L city	Varchar(50)	N		
☐ R stateorprovince	Varchar(20)	N		
☐ M postalcode	Varchar(20)	N		
☐ M phonenumber	Varchar(30)	N		
☐ M emailname	Varchar(50)	N		
☐ R major	Varchar(50)	N		
☐ R studentnumber	Varchar(30)	N		
☐ R notes	Varchar(21474...	Y		

Generate 1000 rows 500 per transaction

Generation options for column 'studentid' (1)

Fill method: R Random Data

Values between [ ] and [ ]

Format [ ] Samples: %02d, %5.2f

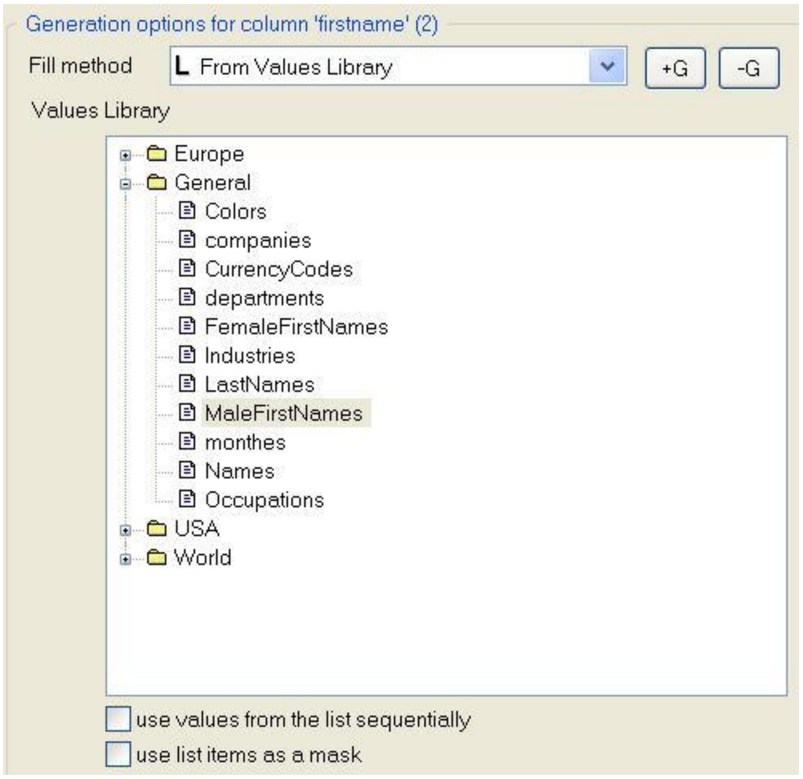
✓ Insert unique values (using Unique Index for large tables is preferred)

Sequence length: 1

Rule note

Figure 12. The rules created by DTM's wizard for the Students table include getting first name, last name and city from predefined lists, and using masked values for address, postal code, phone number and email.

Like ADG, DTM provides predefined lists for a number of items. Figure 13 shows some of the choices. Other sections include US states, US and Canadian postal codes (in separate lists), cities, and countries, among other things. You can also build your own Values Libraries.


Generation options for column 'firstname' (2)

Fill method: From Values Library

Values Library

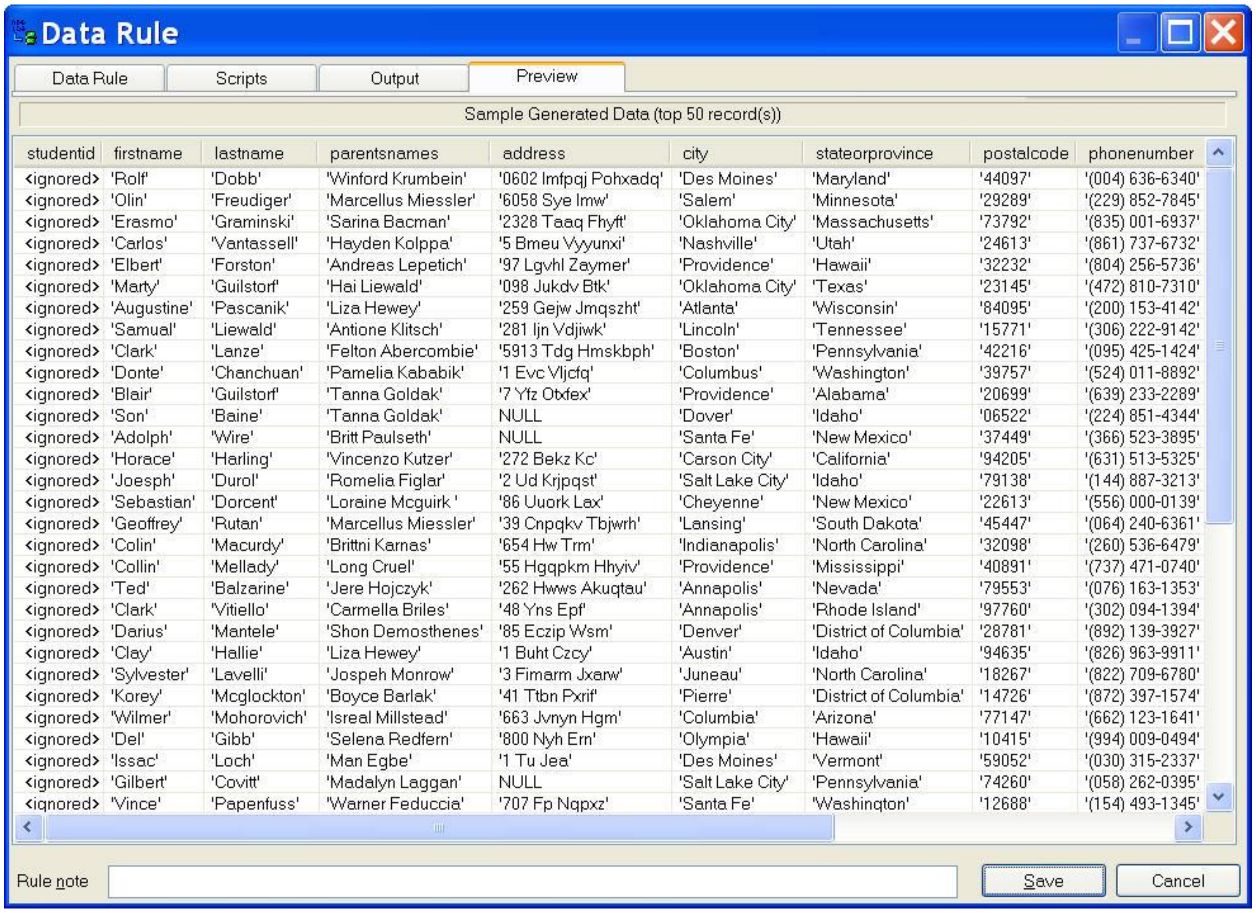
- Europe
- General
  - Colors
  - companies
  - CurrencyCodes
  - departments
  - FemaleFirstNames
  - Industries
  - LastNames
  - MaleFirstNames
  - monthes
  - Names
  - Occupations
- USA
- World

[ ] use values from the list sequentially

[ ] use list items as a mask

Figure 13. You can base the data in a field on a Values Library in DTM, a predefined list. One weakness here is that there's no genderless list of first names; you have to choose either male or female.

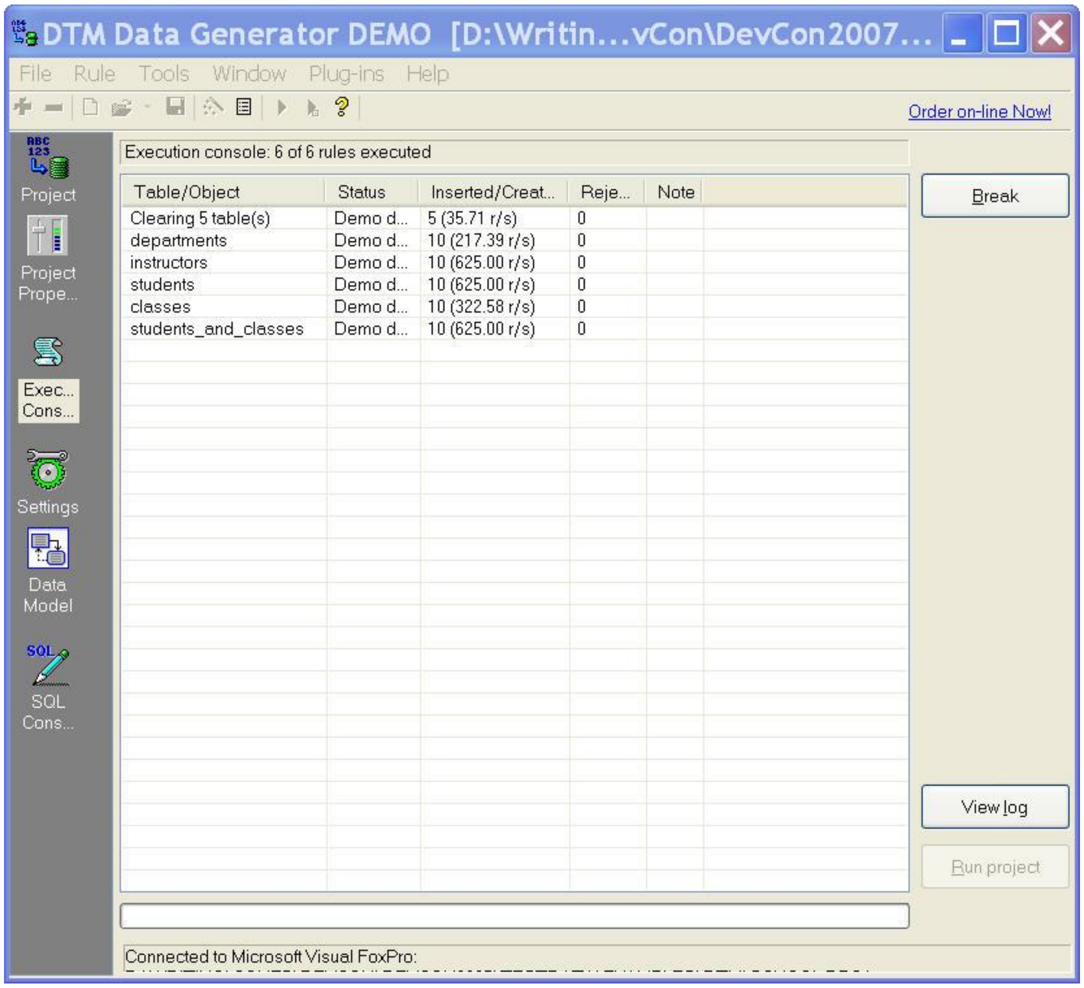
While you're building your rules, you can see what the results for that table will look like by switching to the Preview tab (Figure 14). This ability seems especially useful for masked values, where you may need to adjust the mask several times before you get the desired result.



studentid	firstname	lastname	parentsnames	address	city	stateorprovince	postalcode	phonenumber
<ignored>	'Rolf'	'Dobb'	'Winford Krumbein'	'0602 Imfpqj Pohxadq'	'Des Moines'	'Maryland'	'44097'	'(004) 636-6340'
<ignored>	'Olin'	'Freudiger'	'Marcellus Miessler'	'6058 Sye Imw'	'Salem'	'Minnesota'	'29289'	'(229) 852-7845'
<ignored>	'Erasmo'	'Graminski'	'Sarina Bacman'	'2328 Taaq Fhyft'	'Oklahoma City'	'Massachusetts'	'73792'	'(835) 001-6937'
<ignored>	'Carlos'	'Vantassell'	'Hayden Kolppa'	'5 Bmeu Vyyunxi'	'Nashville'	'Utah'	'24613'	'(861) 737-6732'
<ignored>	'Elbert'	'Forston'	'Andreas Lepetich'	'97 Lgvhl Zaymer'	'Providence'	'Hawaii'	'32232'	'(804) 256-5736'
<ignored>	'Marty'	'Guilstorf'	'Hai Liewald'	'098 Jukdv Btk'	'Oklahoma City'	'Texas'	'23145'	'(472) 810-7310'
<ignored>	'Augustine'	'Pascanik'	'Liza Hewey'	'259 Gejw Jmqszht'	'Atlanta'	'Wisconsin'	'84095'	'(200) 153-4142'
<ignored>	'Samual'	'Liewald'	'Antione Klitsch'	'281 Ijn Vdjiwk'	'Lincoln'	'Tennessee'	'15771'	'(306) 222-9142'
<ignored>	'Clark'	'Lanze'	'Felton Abercombie'	'5913 Tdg Hmskbph'	'Boston'	'Pennsylvania'	'42216'	'(095) 425-1424'
<ignored>	'Donte'	'Chanchuan'	'Pamelia Kababik'	'1 Evc Vljcfq'	'Columbus'	'Washington'	'39757'	'(524) 011-8892'
<ignored>	'Blair'	'Guilstorf'	'Tanna Goldak'	'7 Yfz Otxfex'	'Providence'	'Alabama'	'20699'	'(639) 233-2289'
<ignored>	'Son'	'Baine'	'Tanna Goldak'	NULL	'Dover'	'Idaho'	'06522'	'(224) 851-4344'
<ignored>	'Adolph'	'Wire'	'Britt Paulseth'	NULL	'Santa Fe'	'New Mexico'	'37449'	'(366) 523-3895'
<ignored>	'Horace'	'Harling'	'Vincenzo Kutzer'	'272 Bekz Kc'	'Carson City'	'California'	'94205'	'(631) 513-5325'
<ignored>	'Joesph'	'Durol'	'Romelia Figlar'	'2 Ud Krjpqst'	'Salt Lake City'	'Idaho'	'79138'	'(144) 887-3213'
<ignored>	'Sebastian'	'Dorcent'	'Loraine Mcguirk '	'86 Uuork Lax'	'Cheyenne'	'New Mexico'	'22613'	'(556) 000-0139'
<ignored>	'Geoffrey'	'Rutan'	'Marcellus Miessler'	'39 Cnpqkv Tbjwrh'	'Lansing'	'South Dakota'	'45447'	'(064) 240-6361'
<ignored>	'Colin'	'Macurdy'	'Brittni Karnas'	'654 Hw Trm'	'Indianapolis'	'North Carolina'	'32098'	'(260) 536-6479'
<ignored>	'Collin'	'Mellady'	'Long Cruel'	'55 Hgqpkm Hhyiv'	'Providence'	'Mississippi'	'40891'	'(737) 471-0740'
<ignored>	'Ted'	'Balzarine'	'Jere Hojczyk'	'262 Hwws Akuqtau'	'Annapolis'	'Nevada'	'79553'	'(076) 163-1353'
<ignored>	'Clark'	'Vitiello'	'Carmella Briles'	'48 Yns Epf'	'Annapolis'	'Rhode Island'	'97760'	'(302) 094-1394'
<ignored>	'Darius'	'Mantele'	'Shon Demosthenes'	'85 Eczip Wsm'	'Denver'	'District of Columbia'	'28781'	'(892) 139-3927'
<ignored>	'Clay'	'Hallie'	'Liza Hewey'	'1 Buht Czcy'	'Austin'	'Idaho'	'94635'	'(826) 963-9911'
<ignored>	'Sylvester'	'Lavelli'	'Jospeh Monrow'	'3 Fimarm Jxarw'	'Juneau'	'North Carolina'	'18267'	'(822) 709-6780'
<ignored>	'Korey'	'Mcglockton'	'Boyce Barlak'	'41 Ttbn Pxrif'	'Pierre'	'District of Columbia'	'14726'	'(872) 397-1574'
<ignored>	'Wilmer'	'Mohorovich'	'Isreal Millstead'	'663 Jvnyn Hgm'	'Columbia'	'Arizona'	'77147'	'(662) 123-1641'
<ignored>	'Del'	'Gibb'	'Selena Redfern'	'800 Nyh Ern'	'Olympia'	'Hawaii'	'10415'	'(994) 009-0494'
<ignored>	'Issac'	'Loch'	'Man Egbe'	'1 Tu Jea'	'Des Moines'	'Vermont'	'59052'	'(030) 315-2337'
<ignored>	'Gilbert'	'Covitt'	'Madalyn Laggan'	NULL	'Salt Lake City'	'Pennsylvania'	'74260'	'(058) 262-0395'
<ignored>	'Vince'	'Papenfuss'	'Warner Feduccia'	'707 Fp Nqpxz'	'Santa Fe'	'Washington'	'12688'	'(154) 493-1345'

Figure 14. Use DTM's Preview tab to see the form of the generated data for a table.

Once you've tweaked the rules as needed, click Run All from the main window to generate the data in the specified format(s). The Execution Console (Figure 15) appears and shows progress.



Execution console: 6 of 6 rules executed

Table/Object	Status	Inserted/Creat...	Reje...	Note
Clearing 5 table(s)	Demo d...	5 (35.71 r/s)	0	
departments	Demo d...	10 (217.39 r/s)	0	
instructors	Demo d...	10 (625.00 r/s)	0	
students	Demo d...	10 (625.00 r/s)	0	
classes	Demo d...	10 (322.58 r/s)	0	
students_and_classes	Demo d...	10 (625.00 r/s)	0	

Connected to Microsoft Visual FoxPro:

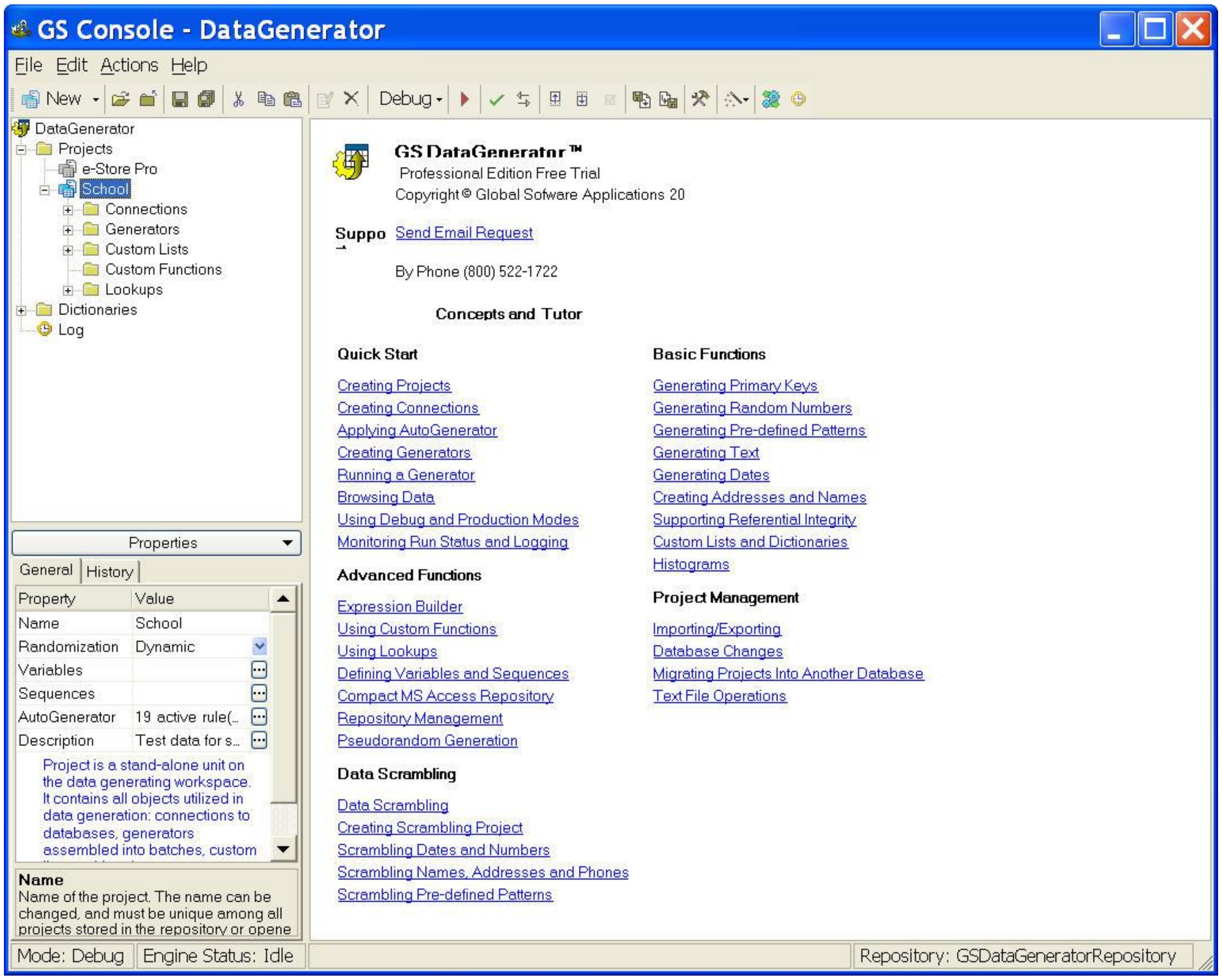
Figure 15. The DTM Execution Console shows progress while generating data.

Overall, DTM does a good job and is fairly easy to work with. The User Interface is unclear in some places. (My guess is that English is not the first language of the developer.) The inability to generate a mixed set of male and female names is also a negative; however, the raw data for the Values Libraries is available from DTM, so you could create a mixed library.

GS Data Generator

GS Data Generator (GSDG) from Global Software Applications (www.gsapps.com) works with a specific list of databases, which includes SQL Server, Oracle and FoxPro (through OLE DB), among others. It comes in standard and professional editions, priced at $595 and $1950 for a single license, respectively. There are volume discounts for multiple licenses: 10% for 2-5, 15% for 6-10, 20% for 11-15 and 25% for 16 or more. A free trial version is available, but does not permit you to actually store data in a database. Instead, you can see the generated data inside GSDG. The trial version lets you test either the standard or professional edition; my testing was done with the professional edition.

GSDG is organized into projects, which contain connections, batches and more. The GS Console (Figure 16) gives you an overview.


GS Console - DataGenerator

File Edit Actions Help

DataGenerator
- Projects
  - e-Store Pro
  - School
    - Connections
    - Generators
    - Custom Lists
    - Custom Functions
    - Lookups
- Dictionaries
- Log

Properties

General | History

Property	Value
Name	School
Randomization	Dynamic
Variables	
Sequences	
AutoGenerator	19 active rule(…
Description	Test data for s…

Project is a stand-alone unit on the data generating workspace. It contains all objects utilized in data generation: connections to databases, generators assembled into batches, custom

Name
Name of the project. The name can be changed, and must be unique among all projects stored in the repository or opene

GS DataGenerator ™
Professional Edition Free Trial
Copyright © Global Sofware Applications 20

Suppo Send Email Request

By Phone (800) 522-1722

Concepts and Tutor

Quick Start
Creating Projects
Creating Connections
Applying AutoGenerator
Creating Generators
Running a Generator
Browsing Data
Using Debug and Production Modes
Monitoring Run Status and Logging

Advanced Functions
Expression Builder
Using Custom Functions
Using Lookups
Defining Variables and Sequences
Compact MS Access Repository
Repository Management
Pseudorandom Generation

Data Scrambling
Data Scrambling
Creating Scrambling Project
Scrambling Dates and Numbers
Scrambling Names, Addresses and Phones
Scrambling Pre-defined Patterns

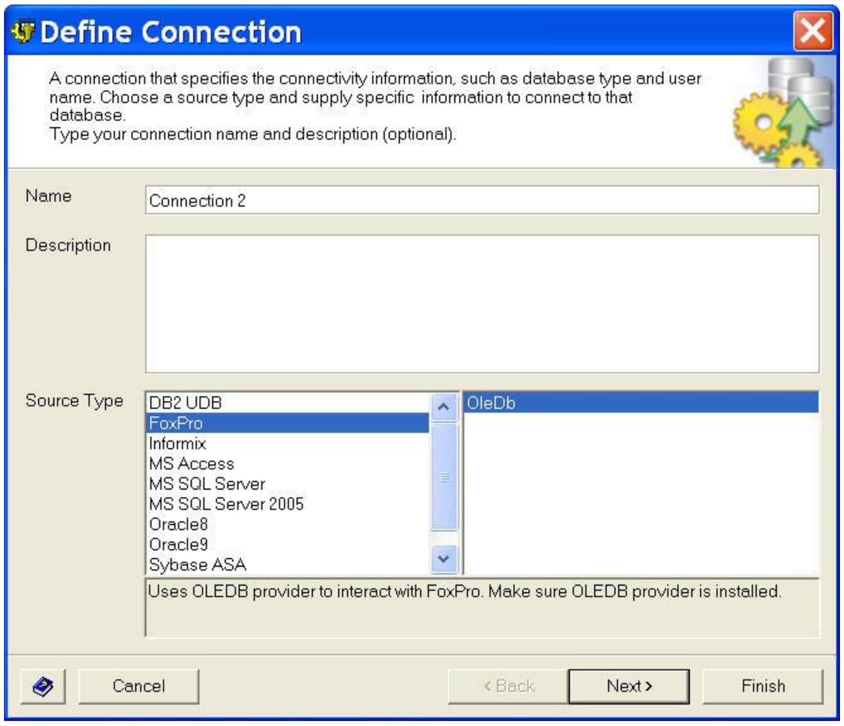
Basic Functions
Generating Primary Keys
Generating Random Numbers
Generating Pre-defined Patterns
Generating Text
Generating Dates
Creating Addresses and Names
Supporting Referential Integrity
Custom Lists and Dictionaries
Histograms

Project Management
Importing/Exporting
Database Changes
Migrating Projects Into Another Database
Text File Operations

Mode: Debug | Engine Status: Idle | Repository: GSDataGeneratorRepository

Figure 16. GSDG is organized into projects. Each project contains connections, batches of generators, lookups, and more.

When you create a new project, a wizard walks you through the initial setup. The Define Connection page (Figure 17) comes first. The next step varies, depending on the type of database you choose. For VFP, the Data Link properties dialog lets you set up an OLE DB connection to a database.



A connection that specifies the connectivity information, such as database type and user name. Choose a source type and supply specific information to connect to that database.
Type your connection name and description (optional).

Name: Connection 2

Description:

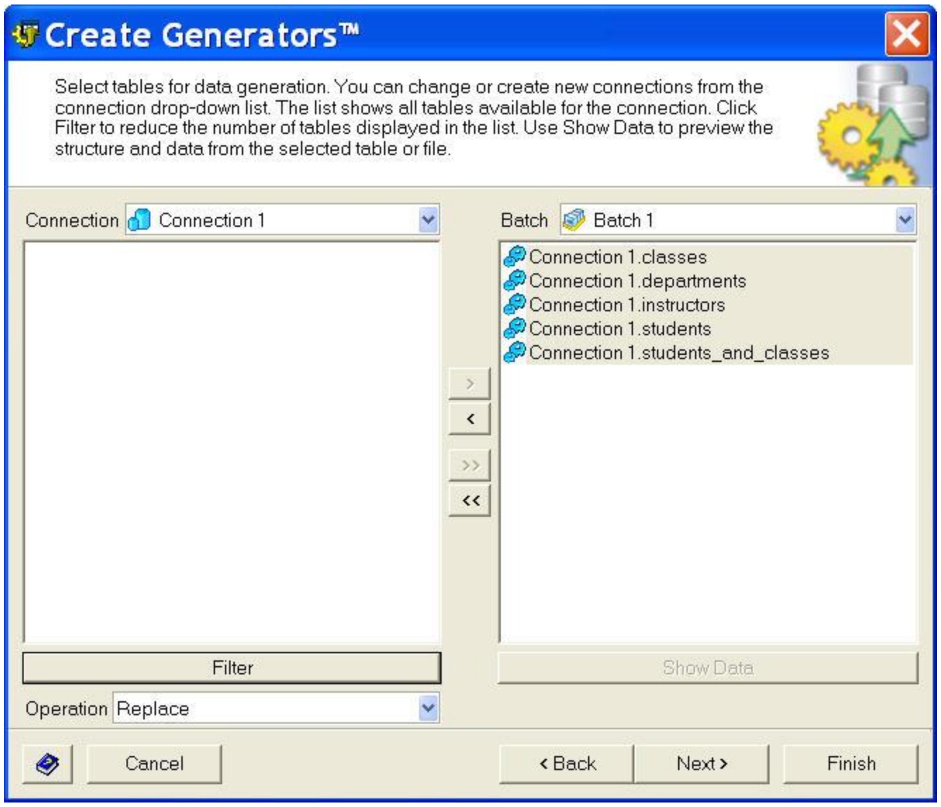
Source Type: DB2 UDB, FoxPro, Informix, MS Access, MS SQL Server, MS SQL Server 2005, Oracle8, Oracle9, Sybase ASA

OleDb

Uses OLEDB provider to interact with FoxPro. Make sure OLEDB provider is installed.

Figure 17. This GSDG page lets you choose the type of database, as well as name and describe the connection.

Once you set up the connection, the Create Generators page (Figure 18) prompts you to set up one or more generators for the database. A generator is a set of rules for generating data for a single table. Generators are grouped into batches. Although the page doesn't make it obvious, you can use it to create several "batches." Initially, it offers you Batch 1; to add another generator, choose <New Batch> from the Batch dropdown.



Create Generators™

Select tables for data generation. You can change or create new connections from the connection drop-down list. The list shows all tables available for the connection. Click Filter to reduce the number of tables displayed in the list. Use Show Data to preview the structure and data from the selected table or file.

Connection: Connection 1

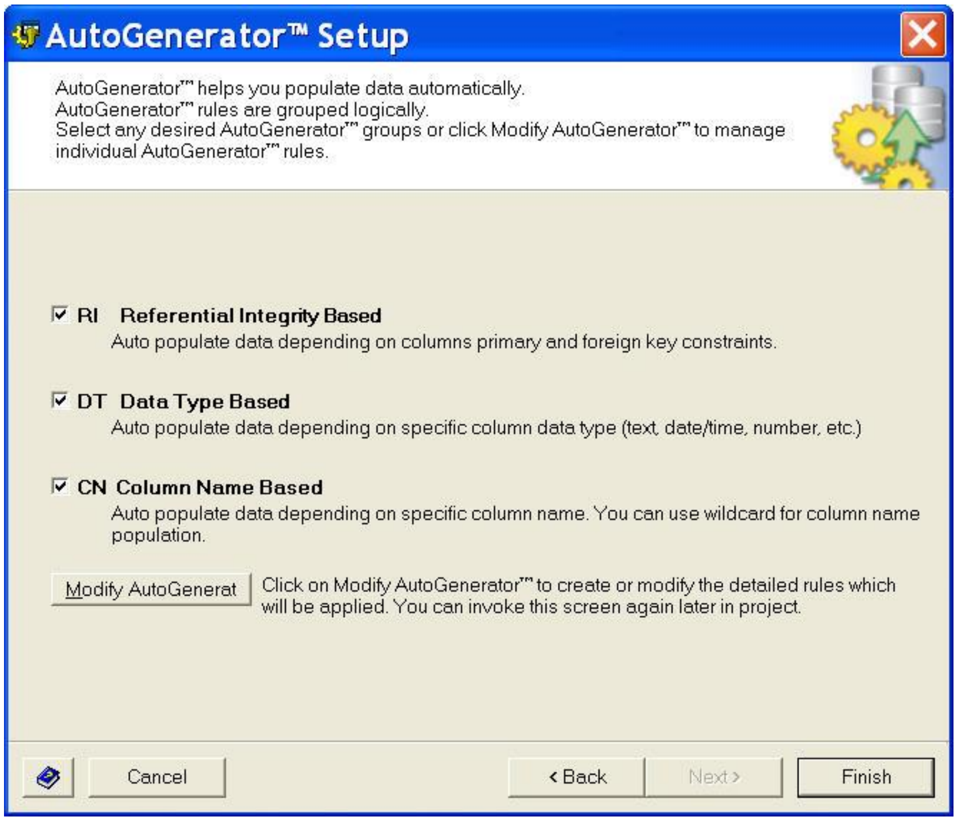
Batch: Batch 1

- Connection 1.classes
- Connection 1.departments
- Connection 1.instructors
- Connection 1.students
- Connection 1.students_and_classes

Operation: Replace

Figure 18. On GSDG's Create Generators page, you decide which tables to generate data for and whether the generated data is added to existing data or replaces it.

When you choose Next from Create Generators, the AutoGenerator Setup page (Figure 19) is displayed. This page lets you specify the rules GSDG applies to the database to create the initial set of generators. Like DTM, GSDG uses the structure of the database to guess how to generate the data. However, in this case, you have significant control over the rules.



AutoGenerator™ helps you populate data automatically.
AutoGenerator™ rules are grouped logically.
Select any desired AutoGenerator™ groups or click Modify AutoGenerator™ to manage individual AutoGenerator™ rules.

☑ RI Referential Integrity Based
Auto populate data depending on columns primary and foreign key constraints.

☑ DT Data Type Based
Auto populate data depending on specific column data type (text, date/time, number, etc.)

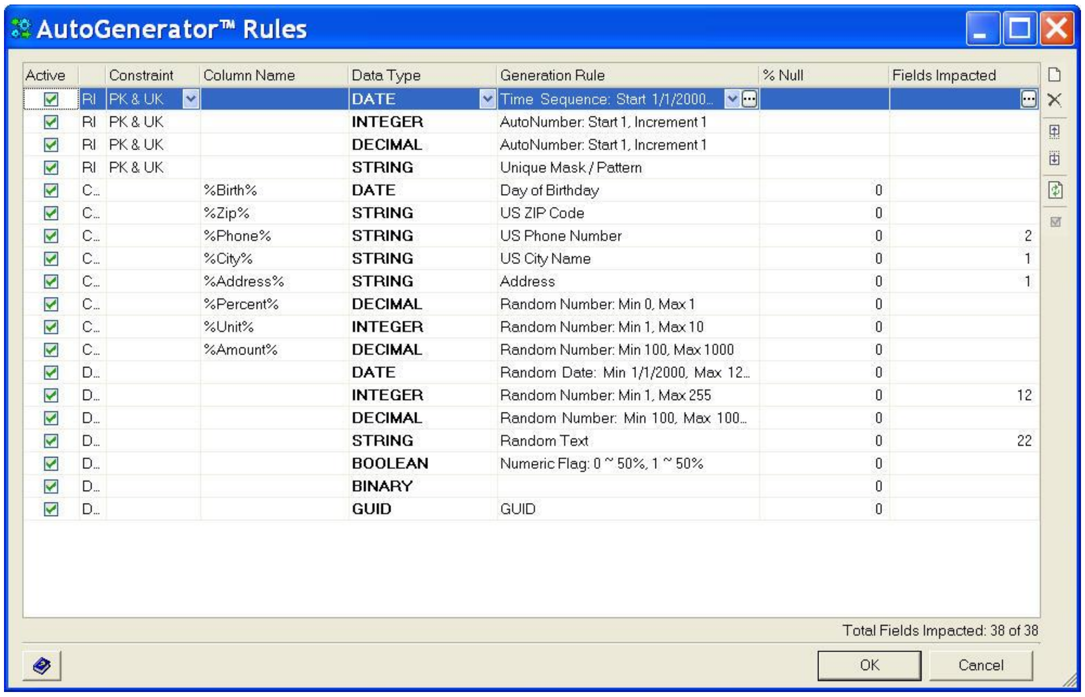
☑ CN Column Name Based
Auto populate data depending on specific column name. You can use wildcard for column name population.

Modify AutoGenerat — Click on Modify AutoGenerator™ to create or modify the detailed rules which will be applied. You can invoke this screen again later in project.

Figure 19. The AutoGenerator Setup page specifies which of GSDG's rules to apply when initially creating generators.

GSDG uses three kinds of rules. The RI rules handle primary, candidate and foreign keys. CN rules provide guidelines for fields with certain strings in their names; for example, by default, a field name containing "zip" is filled with a US Zip Code. DT rules provide defaults for other fields based on their data type. You can turn off any of these sets of rules.

In fact, you can modify or remove the default rules and add your own. Click Modify AutoGenerator to open the AutoGenerator Rules dialog. Figure 20 shows the dialog with the default rules. To modify a rule, just click it and change it. To add a rule, click the New button on the right side of the dialog. A new row is added; you can fill in the fields. (You don't specify the rule type; GSDG figures it out.) Make sure to then move the rule to the right place in the list; it appears that rules are processed in the order shown in this dialog.



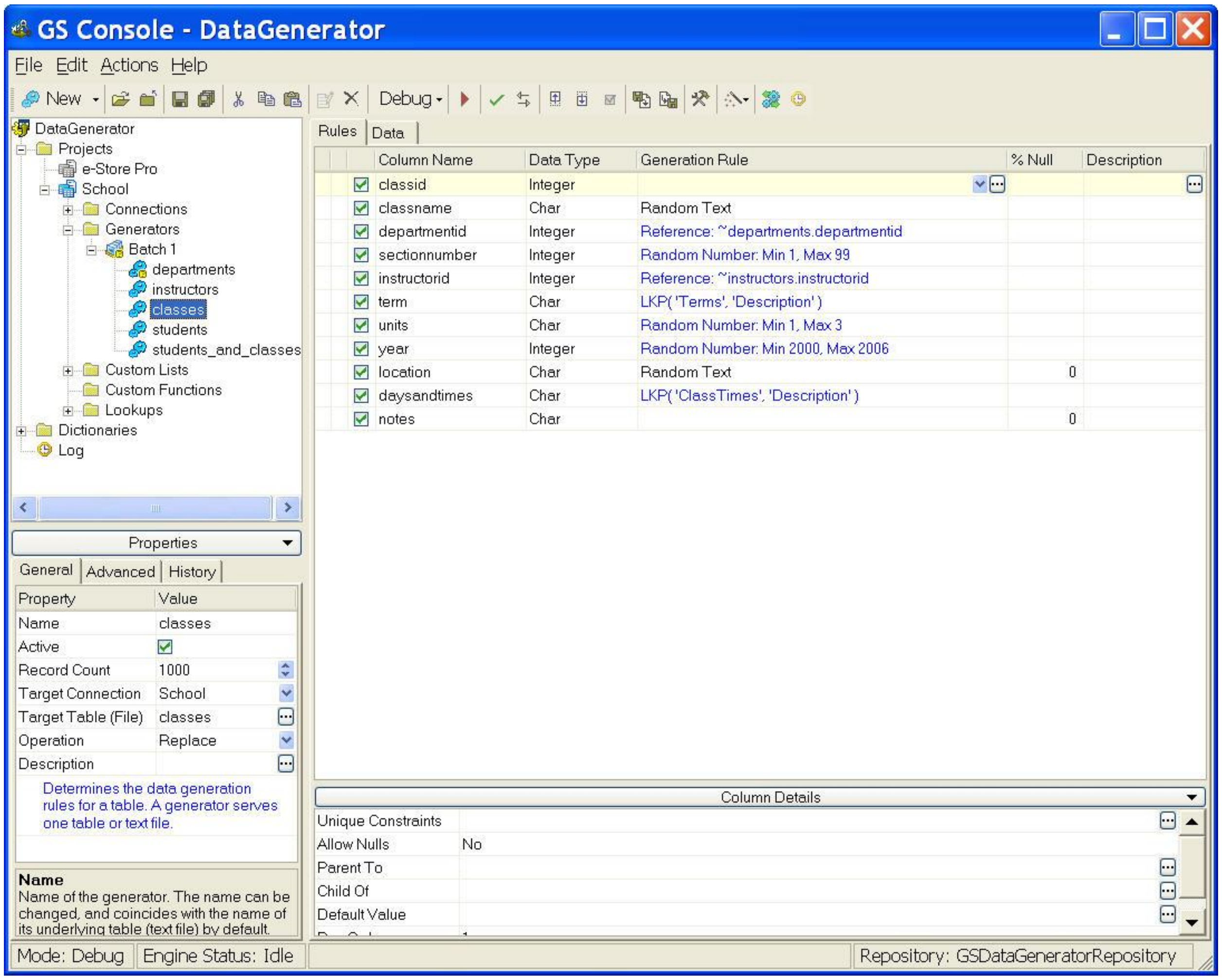
Active		Constraint	Column Name	Data Type	Generation Rule	% Null	Fields Impacted
✓	RI	PK & UK		DATE	Time Sequence: Start 1/1/2000...		
✓	RI	PK & UK		INTEGER	AutoNumber: Start 1, Increment 1		
✓	RI	PK & UK		DECIMAL	AutoNumber: Start 1, Increment 1		
✓	RI	PK & UK		STRING	Unique Mask / Pattern		
✓	C...		%Birth%	DATE	Day of Birthday	0	
✓	C...		%Zip%	STRING	US ZIP Code	0	
✓	C...		%Phone%	STRING	US Phone Number	0	2
✓	C...		%City%	STRING	US City Name	0	1
✓	C...		%Address%	STRING	Address	0	1
✓	C...		%Percent%	DECIMAL	Random Number: Min 0, Max 1	0	
✓	C...		%Unit%	INTEGER	Random Number: Min 1, Max 10	0	
✓	C...		%Amount%	DECIMAL	Random Number: Min 100, Max 1000	0	
✓	D...			DATE	Random Date: Min 1/1/2000, Max 12...	0	
✓	D...			INTEGER	Random Number: Min 1, Max 255	0	12
✓	D...			DECIMAL	Random Number: Min 100, Max 100...	0	
✓	D...			STRING	Random Text	0	22
✓	D...			BOOLEAN	Numeric Flag: 0 ~ 50%, 1 ~ 50%	0	
✓	D...			BINARY		0	
✓	D...			GUID	GUID	0	

Total Fields Impacted: 38 of 38

Figure 20. GSDG's default rules do a pretty good job of generating data, but you can tweak them for your situation.

If you're using VFP's auto-increment feature to generate primary keys, you may want to change the rule for the generation of Integer primary and candidate keys to leave them empty.

Once you click Finish on the AutoGenerator Setup page, GSDG creates an initial set of generators. To see the generator for a table, expand the Generators section and the specific Batch in the treeview and click on the table. The right side of the main window shows the generator for that table; you can modify the rules as needed. Figure 21 shows the generator for the Classes table after modification.



DataGenerator
- Projects
  - e-Store Pro
  - School
    - Connections
    - Generators
      - Batch 1
        - departments
        - instructors
        - classes
        - students
        - students_and_classes
    - Custom Lists
    - Custom Functions
    - Lookups
- Dictionaries
- Log

	Column Name	Data Type	Generation Rule	% Null	Description
✓	classid	Integer			
✓	classname	Char	Random Text		
✓	departmentid	Integer	Reference: ~departments.departmentid		
✓	sectionnumber	Integer	Random Number: Min 1, Max 99		
✓	instructorid	Integer	Reference: ~instructors.instructorid		
✓	term	Char	LKP('Terms', 'Description')		
✓	units	Char	Random Number: Min 1, Max 3		
✓	year	Integer	Random Number: Min 2000, Max 2006		
✓	location	Char	Random Text	0	
✓	daysandtimes	Char	LKP('ClassTimes', 'Description')		
✓	notes	Char		0	

Property	Value
Name	classes
Active	✓
Record Count	1000
Target Connection	School
Target Table (File)	classes
Operation	Replace
Description	

Determines the data generation rules for a table. A generator serves one table or text file.

Name
Name of the generator. The name can be changed, and coincides with the name of its underlying table (text file) by default.

Unique Constraints	
Allow Nulls	No
Parent To	
Child Of	
Default Value	

Mode: Debug | Engine Status: Idle | Repository: GSDataGeneratorRepository

Figure 21. You can see and edit the rules for a particular table in GSDG by drilling down in the treeview.

GSDG provides two ways to specify the rule for a particular field. Use the dropdown in the Generation Rule column (Figure 22) to choose from the most common options, including lists of names, cities, states and so forth. The dropdown also includes "reference," which lets you point to a field in another table to create foreign keys.

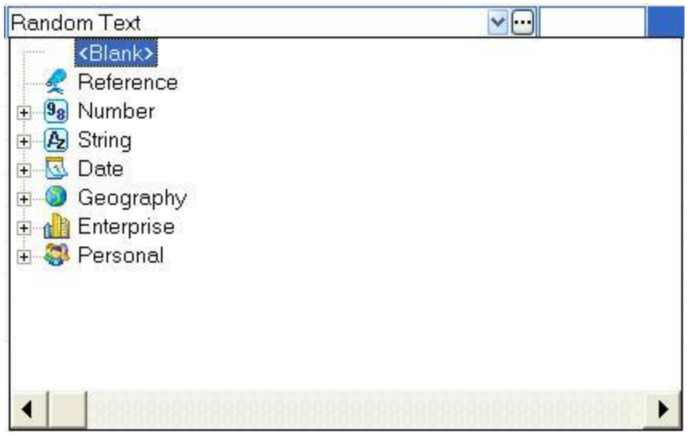
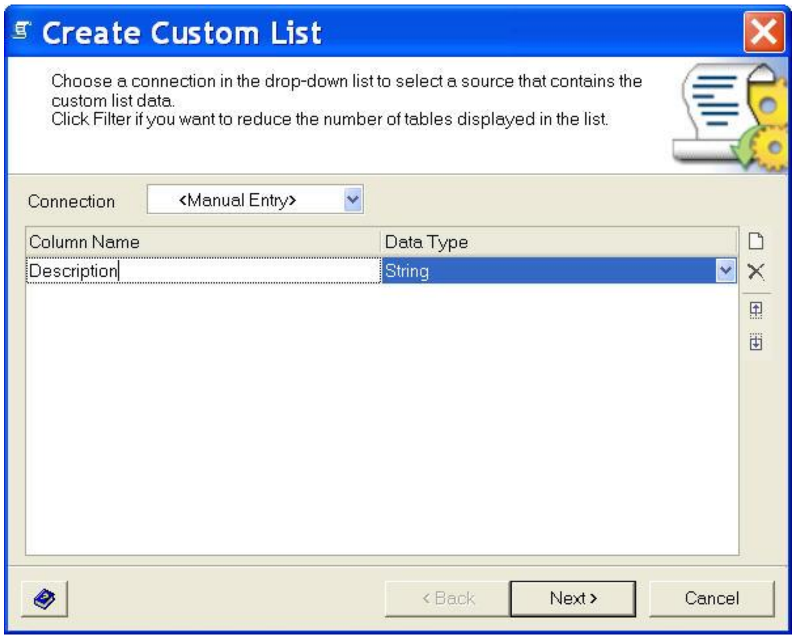


Figure 22. GSDG's Generation Rules dropdown provides quick access to the most common rules.

You can also build more complex rules, by clicking the ellipsis button next to the dropdown. Doing so opens the Expression Builder, where you have access to a wide variety of functions and columns from which you can build the rule.

Using your own list of values is a little harder in GSDG than in other products, but it's also more flexible. To do so, you have to create a Custom List, which is essentially a table. First, you right-click on Custom Lists in the treeview and choose New Custom List. In the Create Custom List dialog, you can either read the structure of the list from a database (including a large list provided with GSDG) or specify the fields yourself, as in Figure 23. Once you've created the structure, choose the new list in the treeview and enter the individual items (Figure 24).



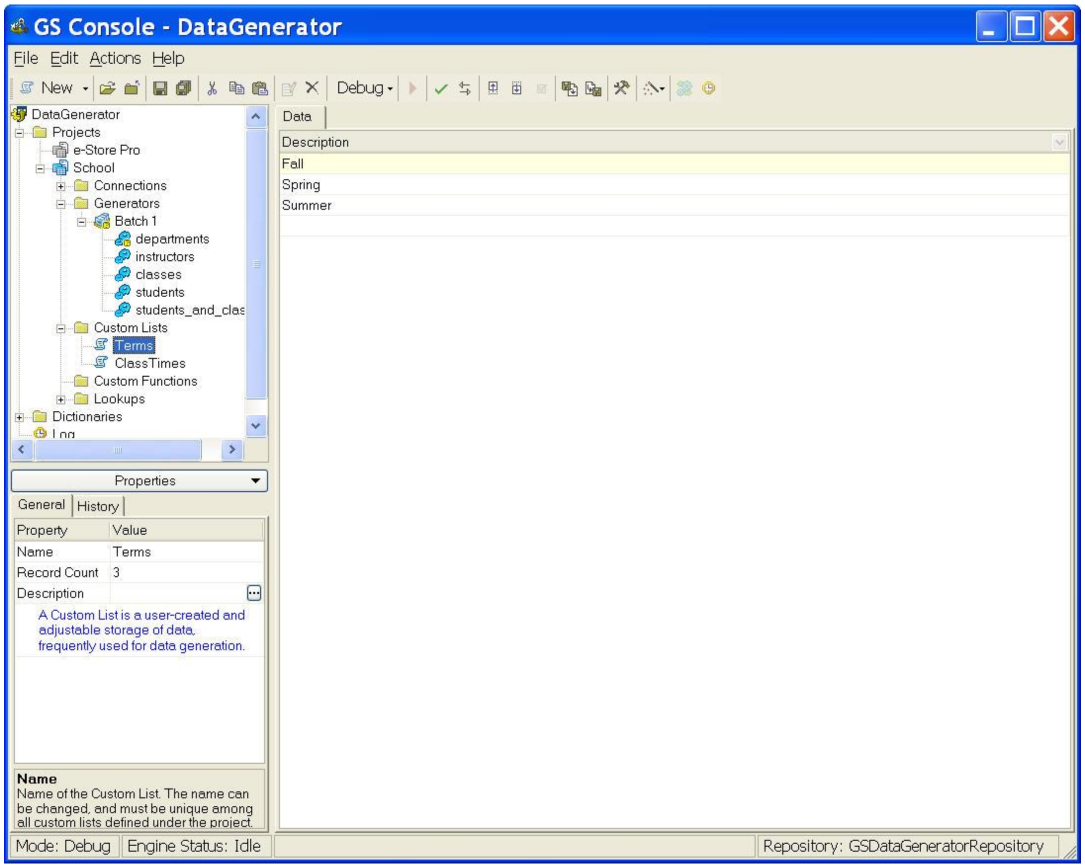
Create Custom List

Choose a connection in the drop-down list to select a source that contains the custom list data.
Click Filter if you want to reduce the number of tables displayed in the list.

Connection: <Manual Entry>

Column Name	Data Type
Description	String

Figure 23. To create a list of values in GSDG, start by defining fields for a table to hold those values.



Description
Fall
Spring
Summer

Property	Value
Name	Terms
Record Count	3
Description	

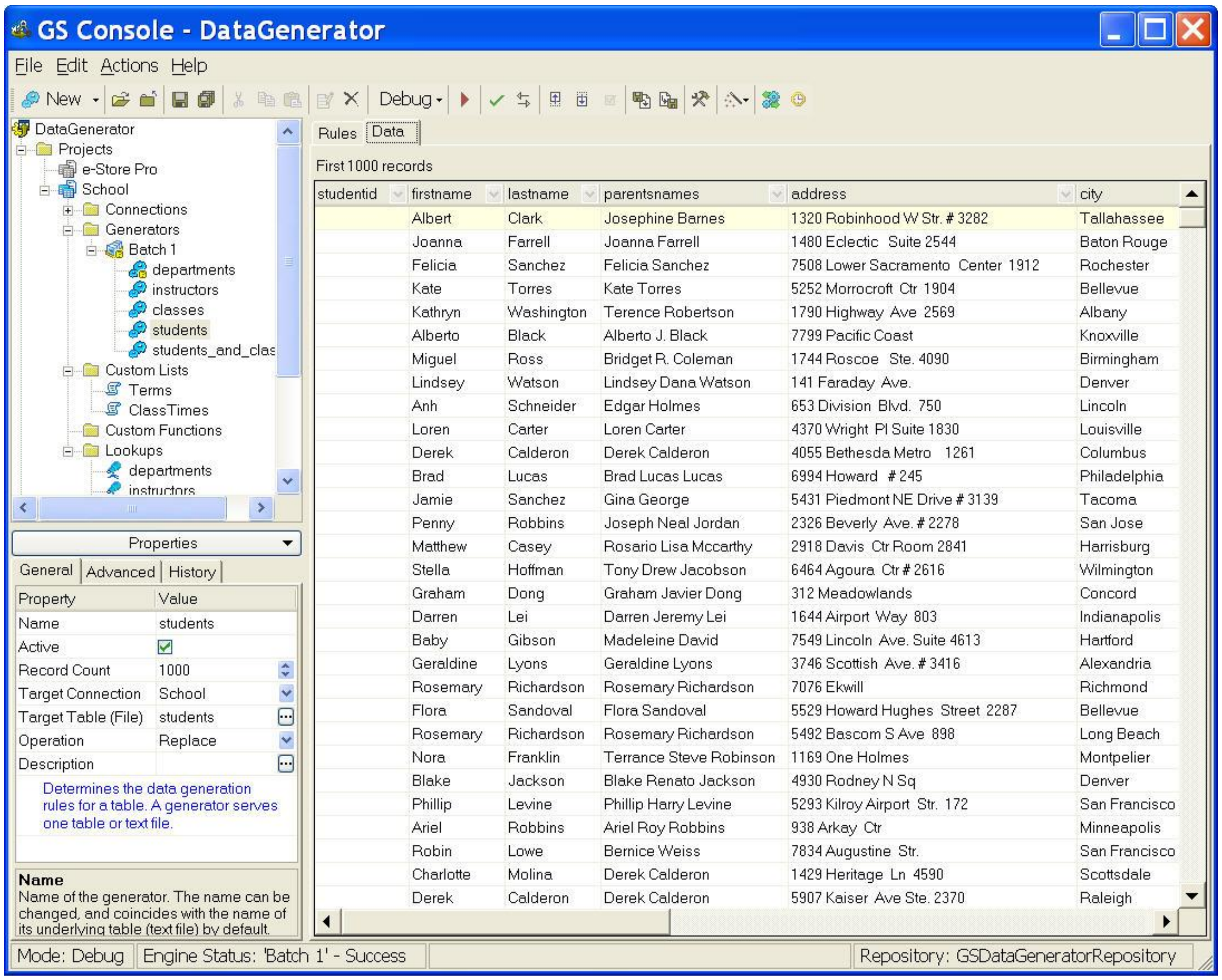
A Custom List is a user-created and adjustable storage of data, frequently used for data generation.

Name
Name of the Custom List. The name can be changed, and must be unique among all custom lists defined under the project.

Figure 24. After you create a custom list in GSDG, you add items to it by typing in the right pane.

You can use custom lists either through the Reference choice or by building a Lookup for the item. The Reference choice is easier, but building a Lookup gives you more control. For example, in a Lookup, you can determine whether values are evenly distributed.

When you've set up the generators as you want them, click the Run button to generate data. In the trial version, you get a reminder that you can only run in Debug mode and data won't be stored to the database. Once you've generated the data, click on the table and then on the Data tab to see the results. Figure 25 shows data generated for the Students table. Because StudentID is an auto-increment field left empty by the generator, there's no data shown. That limit makes it hard to check on foreign keys.



Property	Value
Name	students
Active	✓
Record Count	1000
Target Connection	School
Target Table (File)	students
Operation	Replace
Description	

Determines the data generation rules for a table. A generator serves one table or text file.

Name
Name of the generator. The name can be changed, and coincides with the name of its underlying table (text file) by default.

studentid	firstname	lastname	parentsnames	address	city
	Albert	Clark	Josephine Barnes	1320 Robinhood W Str. # 3282	Tallahassee
	Joanna	Farrell	Joanna Farrell	1480 Eclectic Suite 2544	Baton Rouge
	Felicia	Sanchez	Felicia Sanchez	7508 Lower Sacramento Center 1912	Rochester
	Kate	Torres	Kate Torres	5252 Morrocroft Ctr 1904	Bellevue
	Kathryn	Washington	Terence Robertson	1790 Highway Ave 2569	Albany
	Alberto	Black	Alberto J. Black	7799 Pacific Coast	Knoxville
	Miguel	Ross	Bridget R. Coleman	1744 Roscoe Ste. 4090	Birmingham
	Lindsey	Watson	Lindsey Dana Watson	141 Faraday Ave.	Denver
	Anh	Schneider	Edgar Holmes	653 Division Blvd. 750	Lincoln
	Loren	Carter	Loren Carter	4370 Wright Pl Suite 1830	Louisville
	Derek	Calderon	Derek Calderon	4055 Bethesda Metro 1261	Columbus
	Brad	Lucas	Brad Lucas Lucas	6994 Howard # 245	Philadelphia
	Jamie	Sanchez	Gina George	5431 Piedmont NE Drive # 3139	Tacoma
	Penny	Robbins	Joseph Neal Jordan	2326 Beverly Ave. # 2278	San Jose
	Matthew	Casey	Rosario Lisa Mccarthy	2918 Davis Ctr Room 2841	Harrisburg
	Stella	Hoffman	Tony Drew Jacobson	6464 Agoura Ctr # 2616	Wilmington
	Graham	Dong	Graham Javier Dong	312 Meadowlands	Concord
	Darren	Lei	Darren Jeremy Lei	1644 Airport Way 803	Indianapolis
	Baby	Gibson	Madeleine David	7549 Lincoln Ave. Suite 4613	Hartford
	Geraldine	Lyons	Geraldine Lyons	3746 Scottish Ave. # 3416	Alexandria
	Rosemary	Richardson	Rosemary Richardson	7076 Ekwill	Richmond
	Flora	Sandoval	Flora Sandoval	5529 Howard Hughes Street 2287	Bellevue
	Rosemary	Richardson	Rosemary Richardson	5492 Bascom S Ave 898	Long Beach
	Nora	Franklin	Terrance Steve Robinson	1169 One Holmes	Montpelier
	Blake	Jackson	Blake Renato Jackson	4930 Rodney N Sq	Denver
	Phillip	Levine	Phillip Harry Levine	5293 Kilroy Airport Str. 172	San Francisco
	Ariel	Robbins	Ariel Roy Robbins	938 Arkay Ctr	Minneapolis
	Robin	Lowe	Bernice Weiss	7834 Augustine Str.	San Francisco
	Charlotte	Molina	Derek Calderon	1429 Heritage Ln 4590	Scottsdale
	Derek	Calderon	Derek Calderon	5907 Kaiser Ave Ste. 2370	Raleigh

Mode: Debug | Engine Status: 'Batch 1' - Success | Repository: GSDataGeneratorRepository

Figure 25. The Data tab for a table lets you see the results of data generation.

I found it a little harder to figure out how to use GSDG than the other products, but its overall capabilities are impressive. It is missing some features, however, including the ability to talk to any ODBC or OLE DB data source, and has only a limited ability to generate output other than actual data.

Generating test data with custom code

The final alternative for generating test data is to use custom 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've reduced that cost by writing a set of generic VFP classes to handle test data generation. To use them, you have to write only a small number of methods.

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 in tables. It's quite possible for the same data to be stored in several different ways, so I chose to separate the two processes. (One of the things this decision enables is testing different database designs on the same data. It also enables saving code to generate the data 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 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:

```
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:

```
This.CloseTables()
```

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. 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:

```
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, "Count", 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 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.

```
LOCAL oDataType, nRecord, oRecord

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

   This.AfterMakeType()
ENDFOR

This.AfterMakeSet()

RETURN
```

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

```
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 calls several methods that do the actual work of setting things up:

```
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:

```
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 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:

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

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

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

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

RETURN
```

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

```
LOCAL oTableInfo, lReturn

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

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

RETURN lReturn
```

CloseData loops through the oData collection, closing the tables:

```
LOCAL oTableInfo

FOR EACH oTableInfo IN This.oData 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 in order to generate the actual data for each data type. The methods themselves are added at the subclass level, as well. In subclasses, it's 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. It loops through the list of methods in the aMethods array, calling those that apply to the specified data type:

```
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 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. These provide a group of names, streets, area codes and so forth. They're all listed in the SetData method, which uses the AddData method to populate the oData collection:

```
PROCEDURE SetData

WITH This
   .AddData("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 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.

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

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

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

```
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:

```
LOCAL nRand

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

RETURN
```

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

GetName generates a first name and last name and also sets the record's gender. It uses the BoysNames, GirlsNames and LastNames tables. The method calls 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".

```
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 is typical:

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

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

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

      * Get a house number. First, figure out how many digits,
      * then choose a random value with that many digits.
      * This approach is used because choosing randomly over
      * the whole range results in too many longer values.
      nRand = This.RandInt(This.nLowHouseDigits, This.nHighHouseDigits)
      nLow = 10^(nRand-1)
      nHigh = 10^nRand - 1
      nHouseNumber = This.RandInt(m.nLow, m.nHigh)
      * Check whether to add a letter
      nRand = RAND()
      IF nRand <= This.nHasLetter
         cHouseLetter = This.RandLetter()
      ELSE
         cHouseLetter = ""
      ENDIF
      cHouseNumber = TRANSFORM(m.nHouseNumber) + m.cHouseLetter

      * Get a street
      * 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 social security number at random. The code follows the basic rules for the structure of a US social security number (which I found on the web). It also demonstrates the approach to use for items that should be unique in the data set, but can't be specified as AutoIncrement fields. GetSSN maintains a cursor of the social security numbers generated so far. The code is set up so that the calling object (a subclass of MakeDataSet) could create that cursor before calling on MakePerson; doing so allows MakePerson to add data to an existing test set, rather than only create new test sets. Here's the code for GetSSN:

```
LOCAL cSSN, nDigit1, nDigit2, nDigit3, 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 to the aMethods array.

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 that they'd disappear when I was done. In some cases, you might choose to clone all the tables from an existing database—that could provide an easy way to set up a test data set for an application.

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

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

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

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 puts all contact items into a single table, and maintains a pair of look-up tables to indicate the item type and location. The version shown here 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.

```
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 the School database

MakePerson and MakePersonSet handle the basic information you want in a person record. Since the School database contains two kinds of people (Instructors and Students), I subclassed those classes to create MakeSchoolData and MakeSchoolSet (included with the session materials in MakeSchoolDataV2.PRG).

Five methods contain code in MakeSchoolSet. OpenTables and CloseTables open and close the tables in the School database, respectively.

SetData defines four data types to generate:

```
This.AddDataType("Department", 7)
This.AddDataType("Class", 1000)
This.AddDataType("Instructor", 1000)
This.AddDataType("Student", 1000)

RETURN
```

SaveRecord saves the generated data, using a CASE statement based on the data type to determine what to do.

AfterMakeSet uses the GetRandRecord method to create relational links between the tables, including the full process of generating the Students_And_Classes records:

```
* First, link courses to departments and instructors
SELECT Classes
SCAN
   REPLACE DepartmentID WITH ;
             This.GetRandRecord("Departments", "DepartmentId"), ;
```

```
         InstructorID WITH This.GetRandRecord("Instructors", "InstructorID")
ENDSCAN

* Add major to student
SELECT Students
SCAN
   REPLACE Major WITH This.GetRandRecord("Departments", "DepartmentName")
ENDSCAN

* Add student/course links
FOR nRecord = 1 TO 10000
   nStudent = This.GetRandRecord("Students", "StudentID")
   nCourse = This.GetRandRecord("Classes", "ClassID")
   INSERT INTO Students_And_Classes ;
      (ClassId, StudentID) ;
      VALUES ;
      (m.nCourse, m.nStudent)
ENDFOR

RETURN
```

MakeSchoolData has four additional properties and resets the nHasExtension property. Except for nMaxSections, they all control the probability of a particular situation. nMaxSections determines the maximum number of sections for a course.

```
nSingleParent = .5
nFemaleParent = .7
nMultiSection = .5
nMaxSections = 10
nHasExtension = .5
```

Only the three Set methods are overridden or extended. SetData opens some additional raw data tables:

```
DODEFAULT()
This.AddData("Fields", "Fields")
This.AddData("CoursePrefix", "CoursePrefix")
This.AddData("CourseTopic", "CourseTopic")
This.AddData("Terms", "Terms")
This.AddData("ClassDays", "ClassDays")

RETURN
```

SetProbabilities specifies that everyone has one address, one phone and one email address. SetMethods indicates which methods to call for each data type:

```
WITH This
   .AddMethod("GetName", "Student")
   .AddMethod("GetAddresses", "Student")
   .AddMethod("GetParents", "Student")
   .AddMethod("GetPhones", "Student")
   .AddMethod("GetEmails", "Student")
   .AddMethod("GetStudentNumber", "Student")

   .AddMethod("GetName", "Instructor")
   .AddMethod("GetEmails", "Instructor")
   .AddMethod("GetPhones", "Instructor")

   .AddMethod("GetDept", "Department")
   .AddMethod("GetMgr", "Department")
   .AddMethod("GetChair", "Department")

   .AddMethod("GetCourse", "Class")
   .AddMethod("GetTerm", "Class")
   .AddMethod("GetUnits", "Class")
   .AddMethod("GetMeetings", "Class")

ENDWITH
RETURN
```

As in MakePerson, the various Get methods use the raw data and a set of rules to create particular data items. For example, GetParent figures out whether to use a single parent or a couple, and puts together one or two names:

```
LOCAL nRec, nRand, cName

* Decide whether married or single
nRand = RAND()
IF nRand <= This.nSingleParent
   * Single parent--male or female
   nRand = RAND()
   IF nRand <= This.nFemaleParent
      cName = This.GetRandomName("GirlsNames")
   ELSE
      cName = This.GetRandomName("BoysNames")
   ENDIF
ELSE
   cName = This.GetRandomName("BoysNames")
   cName = m.cName + " and " + This.GetRandomName("GirlsNames")
ENDIF

* Choose a last name
cName = m.cName + " " + This.GetRandomName("LastNames")

ADDPROPERTY(This.oRecord, "ParentsNames", m.cName)

RETURN
```

To generate the School data set, instantiate MakeSchoolSet and call the MakeSet method:

```
oMakeSet = NewObject("MakeSchoolSet", "MakeSchoolDataV2.PRG")
oMakeSet.MakeSet()
```

While this approach is more work than using any of the commercial products, the ability to tweak using VFP code means I can get data in exactly the form I want. The session materials include all the classes described here.

The Bottom Line

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.