The IBM VisualAge Generator Newsletter

VisualAge® Generator

A Powerful New Vision of Programming™

Volume 3, Number 3
August 1998

Contents

VisualAge Generator Version 3.1 2

Nikon Optical Develops a New Sales/Logistics System with VisualAge Generator 3

Building Visual Basic and PowerBuilder Clients 4

Oracle Support in Version 3.1 9

Making It Easier to Find and Display Parts 11

Understanding Garbage Collection 13

Enforcing VAGen Part Naming Conventions 14

Changing the Edit Policy of a CDV Column 16

Math Function Words and Floating Point Arithmetic 21

Dare to Develop with a RAD Methodology 24

ASRA Abends Happen! 26

Object Connection—Partners in Development 30

Year 2000 Support—Fact and Fiction 31

VisualAge Generator Version 3.1

by Barry Stevenson, Manager, VisualAge Generator Development

Welcome to another issue of the VisualAge Generator Newsletter! You'll find this issue packed with information to get you started using the recently shipped VisualAge Generator V3.1, as well as other hints and tips.

Worldwide use of VisualAge Generator technology continues to rapidly grow, providing users with powerful visual development tools to build applications that satisfy the most demanding enterprise application scenarios. This month's issue highlights one such customer's experience.

VisualAge Generator V3.1 continues an exciting tradition of delivering advanced capability for high-end application development, helping you build, deploy, and multi-tier client/server applications across a diverse group of platforms (Windows NT, Windows 95, Windows 3.11, OS/2, OS/400, AIX, HP-UX, VSE, VM, and OS/390(MVS)).

The following are V3.1 features at a glance:

- Java Gateway capability has been added for the MVS and AIX platforms. Your Java clients can directly invoke VisualAge Generator server programs in CICS on OS/390 (MVS) and VisualAge Generator server programs on AIX.
- Oracle database support adds relational database middleware options when designing applications. VisualAge Generator server programs can access Oracle V7 databases natively (directly) on Windows NT, AIX, HP-UX, and OS/2.
- Non-VisualAge Generator clients, such as Java or Visual Basic, can call the Interactive Test Facility for VisualAge Generator server programs, providing a single-system environment for debugging during application development
- Built-in functions for handling mathematical operations, such as floating point arithmetic, exponentiation, and logarithms
- ODBC support on the HP-UX platform. This enables VisualAge Generator server programs to access non-IBM relational databases
- Various usability and performance improvements, which include performance improvements for MVS batch programs accessing VSAM data
- VM shared saved segments support. VisualAge Generator server runtime components, as well as generated COBOL application programs, can be loaded into VM shared saved segments, reducing working set sizes and virtual storage requirements

With the collaborative effort between IBM VisualAge Generator Development and Planetworks, you can create Visual Basic or PowerBuilder GUIs capable of calling VisualAge Generator transaction server programs. In this issue of the newsletter, you'll find an excellent article outlining the setup associated with this powerful new enhancement.

Commercial software development around the VisualAge product family continues to drive new choices and expanded capabilities for our VisualAge Generator customers. We are pleased that this partnership continues to accelerate tool development for VisualAge products with regard to reusable software components.

Given the never-ending Year 2000 discussion in the news, this month's issue features an article that outlines VisualAge Generator, VisualGen, and CSP considerations in light of this pressing issue.

Inherent within VisualAge Generator technology is the concept of rapid application development. Given the interest on this topic, we thought you'd appreciate some discussion on how best to use this significant paradigm in software development. And finally, there are some great articles on VAGen parts management, Smalltalk garbage collection, changing CDV edit behavior, and CICS debugging (ASRA abends).

Send us your comments and thoughts for future articles or, better yet, submit an article on a topic of interest based on your experiences. Echoing previous authors of this space, we encourage you to write an article to publish in the newsletter (see the Comment Form for details). We are anxious to get your feedback as you experience the VisualAge Generator "powerful new vision of programming."

A Customer Success Story

Nikon Optical Develops a New Sales/Logistics System with VisualAge Generator

by Sue Royer, VisualAge Generator Sales and Technical Sales Support

Nikon Optical is a retailer of eyeglass lenses and frames. Since 1987, Nikon has increased its revenues to 20 billion yen. The company saw further opportunities ahead if it could fine-tune its sales and logistics processes. So, Nikon Optical asked IBM Global Services to provide a complete end-to-end solution, including the development of a new sales/logistics system using VisualAge Generator.

Nikon Optical had used distributed systems in the past, but felt they would need a new system to support the amount of change needed. In addition, system maintenance costs would remain high in the old environment. Nikon Optical chose a complete, integrated client/server solution from IBM, including everything from developing new systems to outsourcing systems maintenance.

An Open Solution Cuts Costs and Improves Customer Service

Using VisualAge Generator, Nikon Optical created a system called the NEWTON. This system consists of a number of applications, including the following:

- Order entry
- Shipping
- Order request
- Stock management
- Billing and collections
- Accounting
- Product planning support
- Database inquiries

The NEWTON went hand in hand with an effort to streamline Nikon Optical's business processes. Nikon Optical constructed a new, centralized business centre model for ordering and shipping. All orders through this business centre are now processed using the NEWTON system. The addition of bar-coding technology for order picking in the business centre has likewise increased the speed and quality of order fulfillment.

To support the ordering and shipping business centre, Nikon Optical also redeveloped the infrastructure of the platform that connected to the order-entry system for glasses retailers and large sellers. In addition, to reduce systems maintenance, Nikon Optical centralized the server function on IBM RS/6000 SP. The result is a powerful, open solution that fully addresses Nikon Optical objectives.

"For example, we've decreased our space requirements for inventory and increased the efficiency of our business processes. Previously, when we received the order or inquiry from the customer, we had to find the memos or ask the person who is savvy with it. Now it is totally different. The new system enables us to give the correct answer to our customers quickly, since we are able to find the accurate data anytime, anywhere," says the Nikon Optical official.

A Simple, Powerful Interface Increases User Productivity

One of the main challenges in building the NEWTON was to create an effective graphical user interface (GUI) for order entry. The goal was to enable users to enter a customer order with the minimum number of keystrokes. In addition, for order-entry applications, user response time had to be within one to three seconds, each calling a server program. To achieve both these goals using VisualAge Generator, the SI team had made an effort to tune the GUI, transferring data from client to server and server applications.

VisualAge Generator is a workstation-based visual programming environment for rapidly developing high-quality enterprise applications. It enabled the team of about 30 IBM VisualAge developers to complete the GUI, which included some 350 modules and an AIX/CICS application with over 1,000 modules, all in just months. The entire sales/logistics system and the newly reengineered business processes were in operation by October 1997.

VisualAge Generator Rapidly Develops High-Quality Enterprise Applications

"Before this new system was implemented, our order-entry process required a product expert," says the Nikon Optical official. "The new system, with its functions for easy input, enables even an inexperienced user to enter the order. In addition, we can now use the data more effectively to support our marketing activities."

He adds, "For the future, we plan to extend the system to the new network computing area. This will help us maximize our business opportunities by enabling real-time order processing for a wide variety of customers via the Internet."

Building Visual Basic and PowerBuilder Clients

by Roger Newton and Paul Hoffman, VisualAge Generator Development

VisualAge Generator and Interspace by Planetworks have teamed up to provide an exciting and powerful new offering for enterprise customers. Using the Interspace development framework and middleware, developers can create Visual Basic or PowerBuilder GUIs that call VisualAge Generator's robust, scalable, transaction server programs to access enterpise data.

In this article, we describe how to build a Visual Basic client for the VisualAge Generator sample server program STFLIST using Interspace. The article assumes you are already familiar with the material in Chapter 15 of the *Client/Server Communications Guide* for VisualAge Generator Version 3.1.

Source code for the sample files can be obtained from the VisualAge Generator ftp site at URL:

```
ftp://ps.software.ibm.com/ps/products/visualagegen/
info/v3.1
```

Sample files are:

STFLIST.cat

Service and service interface defined by Interspace

STFLIST.esf

STFLIST service code ESF file generated by Interspace

STFAPP.esf

STFAPP ESF file modified from STFLIST example

STFLIST.bas

STFLIST wrapper for Visual Basic program generated by Interspace

STFLISTM.bas

A Visual Basic program to initialize the project

STFPROJ.vbp

Visual Basic project properties

STFLOGIN

A Visual Basic userid/password authentication form

STFGUI

A Visual Basic GUI form

Defining the Server Program Interface

Developing an Interspace-enabled client begins with Interspace's Service Interface Painter. This tool is used to define the data passed between the client and server, to perform test calls to the server, and to generate the objects that call the server program. We used the Interspace Painter to define a service interface for the sample program, STFLIST, which is shipped with VisualAge Generator Developer.

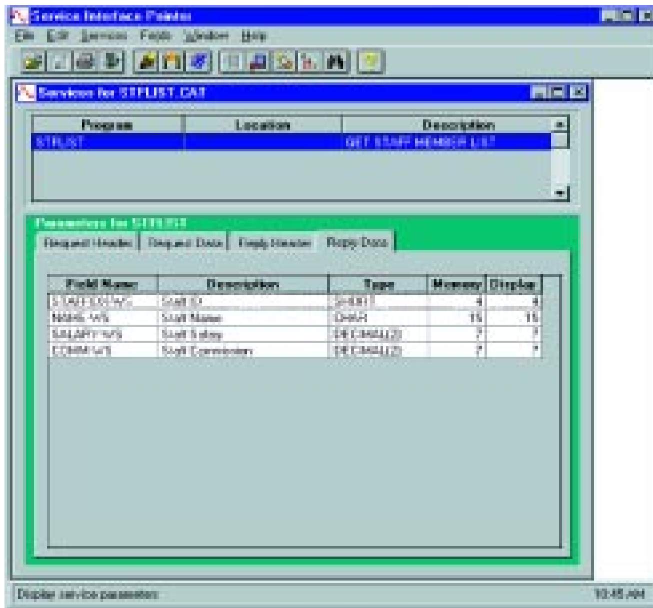
The steps we followed were:

1. Define the fields used in the service interface. Fields are equivalent to VisualAge Generator data items. The fields we defined are located in the STFLIST.cat file.

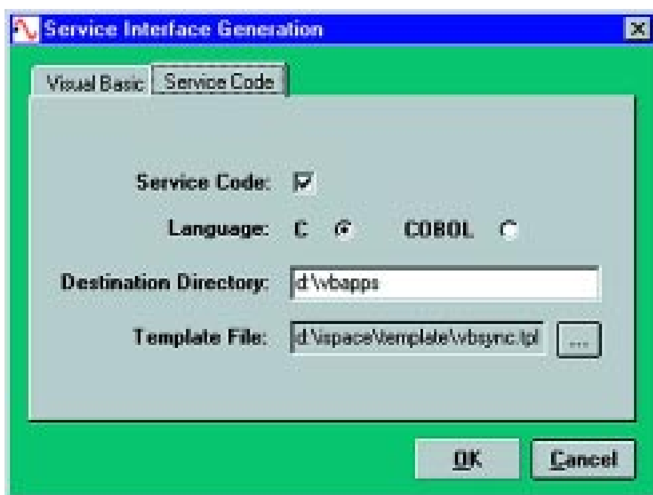
   Note that numbers were defined using type short or integer for integers or decimal for numbers with decimal places.

2. Define the service and service interface. The service is the equivalent of a VisualAge Generator server (remote called batch) program. The service interface is defined as a set of request data flowing to the server, and reply data returned from the server.

In Interspace 5.1, you must define the request data and reply data exactly the same for calling VisualAge Generator servers. If repeating data is defined in the interface, the equivalent of an array in VisualAge Generator, you must also define and use Interspace control fields. These fields should be defined only to the request header and should be defined as the first three fields in the header. The service we defined was named STFLIST. The interface is as follows:



3. Use the Generate function from the Service Interface Painter to generate External Source Format for the program and parameter record member for VisualAge Generator. Notice that Interspace 5.1 generates a single parameter record in the ESF representing both the request data and reply data, rather than two different records as stated in the *Client/Server Communications Guide*. Interspace uses the term “service code” to refer to the ESF file. You can see the generated service code in file STFLIST.ESF.



4. Import the the external source format file into the VisualAge Generator library and code the remainder of the server program using VisualAge Generator Developer. We took most of the code from the existing STFLIST sample program and reworked it to work with the Interspace-generated parameter record. The modified source code is in file STFAPP.ESF.

Test the Server Program

After you define the service interface for the VisualAge Generator server program, you can test the service using the Test function of the Interspace Service Painter as described in the *Client/Server Communications Guide.*

Building Visual Basic GUIs

After the service interface and server program were tested to our satisfaction, we followed the following steps to build the Visual Basic GUI.

1. Generate the Visual Basic wrapper module for the STFLIST service
2. Copy STFLIST module in a Visual Basic project
3. Add modules to the project
4. Build the forms for the GUI
5. Test the Visual Basic program

Generating Visual Basic Functions that Call the Server

Interspace uses the *service definitions* and the *service call template* to generate the appropriate Visual Basic structures and functions. The generated code contains functions that invoke Interspace functions for moving data back and forth between Visual Basic structures and middleware communication buffers. The generated code is written to a Visual Basic file with a “.BAS” extension and a filename equal to the service name.

The *vbsync.tpl* template should be used when generating the service for a VisualAge Generator called server program, since these are synchronous calls. The generator generates several Visual Basic structures and functions that include the name of the service.

Predefined Interspace Functions

The most important function of these generated functions is called *receive_<service>_sync,* where "*<service>*" is the service name. This function invokes other predefined and generated functions to interface with the Interspace middleware services. These predefined and generated functions make up what is called the GUI-Enabling Layer (GEL), which is the top layer. These predefined functions are contained in DCIGEL.BAS. The Distributed Processing Layer (DPL), the middle layer, contains functions that provide additional middleware services. These functions are contained in dcidpl.dll and are declared in DCIDPL.BAS.

Developing A Visual Basic GUI for the STFLIST Server Program

Before a Visual Basic GUI can call a VisualAge Generator server program using the Interspace function, the Interspace environment must be initialized. The function *dcifx_init()*, which initializes the environment, along with other predefined functions, are contained in DCIGEL.BAS, as mentioned above. When terminating the GUI, the Interspace environment should be cleaned up by using the *dcifx_exit()* function. The sample code shows you where these functions and other Interspace functions should be coded in your Visual Basic application.

Visual Basic Modules

The steps for developing a Visual Basic application are as follows:

1. Add the modules (DCIGEL.BAS and DCIDPL.BAS) containing the predefined Interspace functions to the Visual Basic project.

2. For each service (server program) that will be called by the GUI, add the module that was generated from Interspace to the Visual Basic project. For our example, the generated module, STFLIST.BAS, was added to the Visual Basic project.

Adding a Main Subroutine

We created another Visual Basic module called STFLISTM.BAS to keep from modifying the Interspace-generated modules in the event that the service interface changes and the service has to be regenerated. In the STFLISTM.BAS module, we defined a Main subroutine for our Visual Basic GUI and additional variables. The script for the Main subroutine is the first code to be executed and is used to control the initial flow of the GUI.

Included in the additional variables section is a copy of all the service interface parameters for each service call. These are the actual parameters that should be used when making the function calls. The naming convention we adopted for naming the variables was to remove the underscore from the original name. For instance, *STFLIST_request_data* was defined with the name *STFLISTrequestdata*.

```
Public STFLISTrequestdata As FLIST_request_data
Public STFLISTreplydata As STFLIST_reply_data
Public STFLISTrequestmsg As STFLIST_request_msg
Public STFLISTreplymsg As STFLIST_reply_msg

Public pword$      'user password
Public user$       'user name

Public retcode%     'global error code
Public NumofButtons%        'used for the dcifx_show_error

Public Const DistributedEnvironment = "VISGEN"       'environment you are connecting to
Public Const AppToConnectTo = "STFLIST"              'the app connecting to
Public Const Title = "Interspace Reported Error"     'used for the dcifx_show_error

Public Sub Main()
  STFGUI.Show
  stfLogIn.Show 1
End Sub
```

Coding the Visual Basic Forms

The Visual Basic GUI created for our server program contains two forms: a login form (STFLOGIN.frm) and the main form (STFGUI.frm), which controls the interaction with user.

The function, dcifx_init(), which initializes the Interspace environment, is included in the login form (STFLOGIN.frm).

```
Private Sub getListCmd_Click()
  'Initialization section
  Dim stfname As String * 15
  Dim STFListEntry As String * 50

  'Use STARTING_ID entered by the user
  STFLISTrequestmsg.header.starting_id = starting_id.Text

  'Allocate space in memory for the following arrays
  ReDim STFLISTrequestmsg.data(1) 'very important
  ReDim STFLISTreplymsg.data(1) 'very important

  getListCmd.Enabled = False 'Disable until successful server call

  'The parameters are messages:
  '1.STFLISTrequestmsg - data flowing to the server program
  '2.STFLISTreplymsg - data being returned from the server program
  ' In VB terms:  A user defined type. These two were defined in the STFLIST.BAS file.
  ' Their definition is based on the repository file for this service.
  ' The STFLIST.BAS file was generated using the Interspace painter.

  retcode = receive_STFLIST_sync(STFLISTrequestmsg, STFLISTreplymsg)

   'You always use error handling with an Interspace enabled application
   ' The dcifx_show_error provides information about the error.

     If retcode <> 0 Then                              'zero is Interspace success value
       If retcode = -1 Then
       Call dcifx_show_error(NumofButtons, Title)   'Displays Interspace provided error msg
       retcode = 0
       End If
     End If

  getListCmd.Enabled = True 'Call to server was successful
  stfListBox.Clear    'Clear the list box before populating again

  Select Case STFLISTreplymsg.row_count  'using this user defined type to see
                                         'how many rows were returned by the service

  Case Is > 0                            'At least one row was returned
    For i = 1 To STFLISTreplymsg.row_count   'Number of rows read from the database
       stfname = Format(STFLISTreplymsg.data(i).NAME_WS, "@@@@@@@@@@@@@@@")
       STFListEntry = Format(Str(STFLISTreplymsg.data(i).STAFFIDX_WS), "@@@@@") + "| " + stfname +
            "| " + Format(Str(STFLISTreplymsg.data(i).SALARY_WS), "@@@@@") + "| " +
             Format(Str(STFLISTreplymsg.data(i).COMM_WS), "@@@@@") + "|"
       stfListBox.AddItem STFListEntry, (i - 1)
    Next i

  Case Is = 0 'No entries found
  End Select

End Sub
```

Testing the Visual Basic Program

Visual Basic provides robust test facilities for running and debugging the Visual Basic program. Use these facilities to set breakpoints and watch points as you are testing and debugging your program.

You have flexibility in how you test your client and server programs together. You can test the Visual Basic program calling the server program in the VisualAge Generator Test Facility. If you are satisfied with the server program, generate the server program, and test it with Visual Basic calling the generated server program. Once the complete client/server application has been thoroughly tested, the Visual Basic program can be compiled into an executable (EXE).

Deploying The Visual Basic Application

After an executable has been created for your Visual Basic application and is ready to be distributed to the end-users, it is important that all the necessary files are distributed with the executable and that the middleware runtime components are properly installed on each machine. When distributing the Interspace-enabled Visual Basic application for VisualAge Generator middleware, ensure that all the files in the Interspace runtime subdirectory (x:\ispace\runtime) are distributed to the client machine. This directory should contain the Interspace runtime DLL, DCIDPL.DLL, and the DIL DLL for VisualAge Generator, VGENDIL.DLL.

In addition, you need to distribute the latest version of the Visual Basic runtime DLL. Currently, these DLLs are vb40016.dll and vb40032.dll, depending on whether your application is 16 or 32 bit. If your application uses any OCX objects, you also need to distribute the runtime DLLs for these objects.

The runtime DLLs for Interspace, Visual Basic, and OCX (if it's used) should be located in a directory that is defined in the DOS Path of the client's machine. In addition, the Interspace control file, ISPACE.INI, and the repository file containing the service definitions should be copied to the same location.

To complete the setup, you need to install and properly configure VisualAge Generator runtime services and the underlying middleware (CICS Client, Client Access/400, and so on). Refer to the product installation guide for information on the installation and configuration of these products.

You also need a runtime linkage table on the client to specify the location of the server to the PowerServer middleware.

Moving the Server Program to Other Platforms

Our example put the server program on an MVS CICS system. To put the server program on any of the other VisualAge Generator server platforms (for example, IMS, VSE CICS, OS/400, or AIX), regenerate the server program for the new environment and modify the linkage table on the client system to point to the new server. No change is required in the server application code or to the GUI client program.

Using A Different GUI Development Tool

The same server program can be used with PowerBuilder, Java, and ActiveX clients. Use Interspace to generate PowerBuilder DataWindow objects, Java classes, or ActiveX controls that encapsulate the call to the server program, then use the appropriate GUI development tool for the type of object generated. No change is required to the server program.

Summary

The Interspace and VisualAge Generator integrated solution provides customers with a development framework for developing distributed applications using popular UI tools like Visual Basic and PowerBuilder, where the customer prefer these tools.

General Notes

To use the new Callable Test Facility feature introduced in VisualAge Generator V3.1, fixpak1 should be applied to your image.

Interspace 5.1 includes the VisualAge Generator support. We recommend that you pick up the latest fixes for this support. For more information on the Interspace product, visit the Planetworks web site at:

`www.planetw.com.`

Oracle Support in Version 3.1

by Chuck Proffer and Roger Newton, VisualAge Generator Development, and Susan Lafera, VisualAge Generator Information Development

Do you need to access Oracle databases from your VisualAge Generator server application? In previous versions of VisualAge Generator, you might have accessed Oracle using ODBC or DataJoiner. In Version 3.1 of IBM's VisualAge Generator Server, you can directly access Oracle databases without the overhead of ODBC or DataJoiner. Oracle support is available for C++ programs generated for AIX, HP-UX, and Windows NT. Look for support for OS/2 in a future fixpak.

Defining and Testing Your Programs

Oracle databases are not directly accessible from the VisualAge Generator Definition Facility or Test Facility, so the ODBC interface must be used when defining and testing your programs. For more information on using the ODBC interface, refer to the *VisualAge Generator Design Guide.*

Generating Your Programs

After testing your programs, you can generate them to directly access Oracle databases at runtime. When generating your SQL programs, you must specify the database management system you want to use: DB2, Oracle, or ODBC. If you are using the HPTCMD GENERATE batch command, the /DBMS generation option is used to specify the database management system you want to use. If you are generating from the user interface, the database management system is specified in the generation options under the **Validation** tab. If you do not specify a DBMS generation option, the default is DB2.

Running Your Programs

When running VisualAge Generator programs that access Oracle databases, the same environment variables are used to specify the database name as are used when accessing DB2 or ODBC; EZERSQLDB and FCWDBNAME_<progname>. EZERSQLDB enables you to globally specify the database name for all programs and FCWDBNAME_<progname> enables you to specify the database name for a specific program.

Special Considerations

If you are writing new VisualAge Generator programs, you should use VisualAge Generator's Retrieve SQL function to create the SQL row record. This will ensure that the SQL row record data item definitions match the columns in the Oracle table. Alternatively, you can define the SQL row record by specifying the SQL column data types using the DB2 native SQL data codes. Refer to the *VisualAge Generator Programmer's Reference* for the SQL data codes.

If you are migrating existing VisualAge Generator programs that were written for DB2, there are a couple of areas of consideration. The first is data type differences between DB2 and Oracle. The following table shows the Oracle data type and the equivalent DB2 data type.

Oracle Data Type	DB2 Data Type
CHAR(n), n <= 254	CHAR(n)
CHAR(255)	VARCHAR(255)
VARCHAR2(n), n <= 2000	VARCHAR(n)
LONG, up to 32700 bytes	LONG VARCHAR
NUMBER(p,s), p <= 4 and s = 0	SMALLINT
NUMBER(p,s), 4 < p <= 9 and s = 0	INTEGER
NUMBER(p,s), p <= 31 and 0 <= s <= p	DECIMAL
FLOAT(p)	FLOAT
RAW(n), n <= 254	CHAR(n) FOR BIT DATA
RAW(255)	VARCHAR(255) FOR BIT DATA
LONG RAW, up to 32700 bytes	LONG VARCHAR FOR BIT DATA
DATE	TIMESTAMP

Be aware that when you use the DATE data type, there are differences between DB2 and Oracle. In Oracle, the DATE data type contains both date and time information. The corresponding VisualAge Generator data item must be defined as character and must be at least 19 bytes to accommodate the returned data. You can use the VisualAge Generator string functions to separate the date and time information. The default date format is specified by the Oracle Server initialization parameter NLS_DATE_FORMAT and contains a string such as 'DD-MON-YY'. This default can be changed on the server or overridden on the client by specifying the desired format with the following environment variables: NLS_LANG and NLS_DATE_FORMAT.

Another area requiring consideration is functions. If your SQL statements contain functions, you might need to modify the SQL statements if the corresponding function is not available in Oracle. All of the DB2 column functions (AVG, COUNT, MAX, MIN, and SUM) are supported by Oracle; however, only a limited set of the DB2 scalar functions are available in Oracle.

In Conclusion

The VisualAge Generator product continues to respond to its customers by adding new functionality. With the addition of Oracle support, you can now access your Oracle databases directly with improved performance.

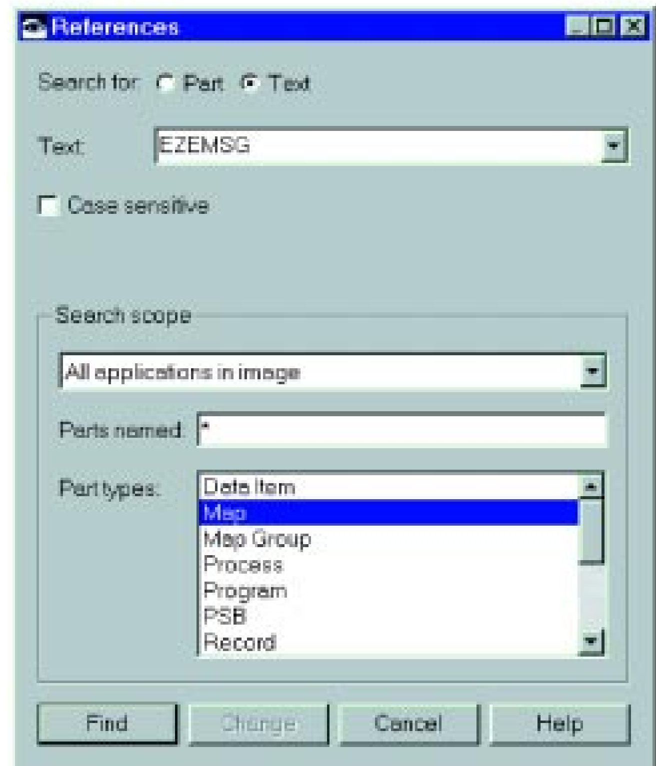
Making It Easier to Find and Display Parts

by Jay Cagle, VisualAge Generator Development

In FixPak 1 of VisualAge Generator Version 3.1, several enhancements were made to make it easier to find and work with **VAGen Parts**. First, there are enhancements to the References utility. The utility now offers two means of searching parts, as well as more control over which parts are searched. You can either search for references to a part of a given name and type, or you can search parts for an arbitrary text string. For example, you can use the text search to find all parts that use a particular EZE word. The text search actually searches the part's ESF for the text string. Knowing this, you can find all programs that allow implicit data items by searching for the string "implicit = Y".

Additionally, the References utility was enhanced to enable greater control in specifying which parts to search. You can indicate which parts are to be searched by specifying a part name pattern and a set of part types. This enables you to narrow and speed up your search. Also, with FixPak 1, you can search parts that are not loaded in your image. The last two options in the search scope drop-down list are **Selected applications** and **Selected configuration maps**. If you choose either of these, a prompter displays enabling you to select multiple application or configuration map editions. The editions you select can be loaded in the image, but the editions don't have to be loaded. If you search selected configuration maps, all applications and subapplications within the maps are searched.

The References utility is available from the **VAGen Parts** menu of the **VisualAge Organizer** window, and from the **Parts** menu of the **VAGen Parts Browser**. The search example in the following **References** window locates all Maps in the image that use EZEMSG.



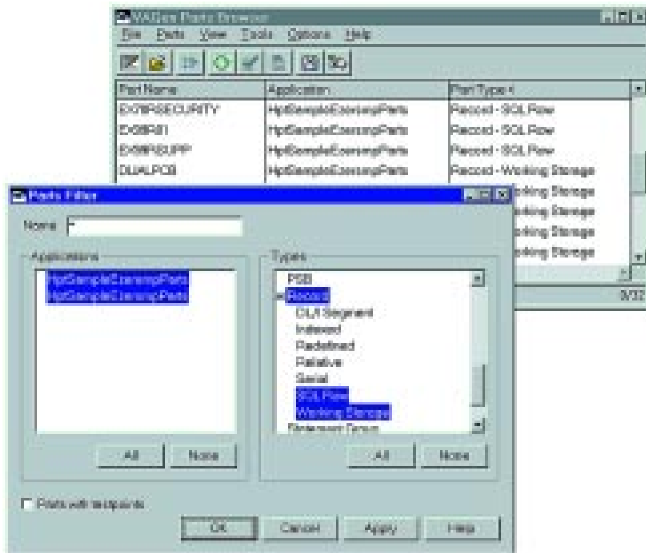
Also in FixPak 1 of Version 3.1, are enhancements to the **VisualAge Organizer** and **VAGen Parts Browser**, which enables you to view and filter by part subtype. Four VAGen part types have subtypes: processes, programs, records, and tables. The subtypes are the process option, the program type, the record type (organization), and the table type. You can toggle which columns are displayed in the **VisualAge Organizer** by selecting columns from the **VAGen Parts->View** cascade menu. If you choose, the subtype is displayed in a column by itself. In the **VAGen Parts Browser**, columns are configured using the **Reorder Columns** window, which you can display by selecting Reorder Columns from the **Parts** menu. You can choose a column that displays the type only or a column that displays both type and subtype. Select **Save as Defaults** from the **View** menu to make your column choices the default for the **VAGen Parts Browser**.

In both the **VisualAge Organizer** and **VAGen Parts Browser**, if the subtype is displayed and you sort the list by type, the list is sorted by type, subtype, then part name. If the subtype is not displayed and you sort by type, the list is sorted by type only, then by name. Displaying the subtype, and sorting by type when subtype is displayed, will be slower than not displaying the subtype or sorting by something other than type. As an alternative, in the **VAGen Parts Browser**, you can display the type and subtype in the status bar instead of in a column. Select **Reorder Status Bar Text** from the **View** menu to configure the status bar information.

The **VAGen Parts Browser** also filters by subtype. From the **Parts Filter** window, you can expand the process, program, record, and table types to display their subtypes. When the types are expanded, you can select only the subtypes you want displayed. If a type is not expanded, all its subtypes are displayed.

Another minor enhancement has been made to the **VAGen Parts Browser,** in which the status bar now displays the number of parts currently selected and the total number of parts in the list.

The following figure shows the **Parts Filter** window with the record type expanded and the SQL row and working storage subtypes selected. The **VAGen Parts Browser** window behind the **Parts Filter** window shows the resulting list, configured to display the type and subtype.



Understanding Garbage Collection

by Jim Eberwein, VisualAge Generator Development

"Garbage Collection" refers to the Smalltalk process of the virtual machine periodically identifying unreferenced objects and deallocating their memory. It is important to understand that just destroying an object does not make that object eligible for garbage collection. Instead, you must ensure that the object is no longer referenced by any other object in order for the system to reclaim the storage allocated by the destroyed object. GUI classes are objects that are frequently destroyed and not garbage collected. This usually occurs when the GUI class was previously stored in an ordered collection and not removed during the destroy processing. However, instances of destroyed GUI classes can remain in storage when another GUI class contains a variable part that references the destroyed object. To ensure that the GUI class is deallocated, the class should be removed from the Order Collection prior to being destroyed, and variables that point to the class should be modified to point to another class object.

Using VisualAge Generator 3.0, it is easy to determine the status of the instances of GUI classes throughout the Smalltalk environment. All you need to do is execute some Smalltalk code. But before you start examining the current state of your Smalltalk environment, you should first understand the class hierarchy of the GUI classes defined in the system. By default, when a new GUI class is defined, the class is specified as a subclass of the VisualAge Smalltalk class *AbtAppBldrView*. An examination of this class will show all of your defined GUI classes as subclasses of *AbtAppBldrView*. You should also notice that there are additional classes defined as subclasses of *AbtAppBldrView*. These subclasses are most likely provided with VisualAge Smalltalk or VisualAge Generator. Since we are not interested in seeing any instances of these subclasses, the logic used to evaluate all subclasses of *AbtAppBldrView* should be smart enough to ignore these classes. This should be easy enough to write since these classes will have names that begin with common prefixes.

The following example stores all of the Non-VisualAge subclass instances of the class *AbtAppBldrView* into an ordered collection. Each element in the ordered collection will be an array that contains all of the instances of a specific GUI class. This ordered collection can then be inspected to see what GUI classes have instances in the system. To execute the code and inspect the output, select the code in a work space area and choose **execute** from the **context** menu.

```
| allguis nonVAInstances a1 |

System globalGarbageCollect.

allguis := OrderedCollection new.

nonVAInstances := OrderedCollection new.

allguis := AbtAppBldrView allSubclasses.

allguis do: [ :element |

   ((( (element name ) asString)
        indexOfSubCollection: 'Abt' startingAt:
        1) = 1)

   ifFalse: [ ((( (element name ) asString)
        indexOfSubCollection: 'Hpt' startingAt:
        1) = 1)

   ifFalse: [ ((( a1 := element basicAllInstances)
        size ) = 0 ) ifFalse: [nonVAInstances add:
        a1  ] ]] ].

nonVAInstances inspect.
```

Upon completion of the above logic, an **OrderedCollection Inspector** is displayed. You should be able to verify that each element is an array that contains the number of instances for a particular non-VisualAge Smalltalk or VisualAge Generator subclass of *AbtAppBldrView* that exists in your virtual machine. By examining each element in these arrays, you can determine if an instance has been destroyed but, for some reason, is not available for garbage collection. Each instance in the array has the attribute *abtIsDestroyed*. If this value is "true", then the instance has been destroyed, but still is referenced by another object. This is true because the example code performed a system garbage collection, which frees the memory for any destroyed objects that no longer has any references to them.

An understanding of the relationship between the process of an object being destroyed and garbage collection should help developers better manage the virtual memory requirements of their application suite. This article hopefully addresses this relationship. Adventurous developers can use the above code as a springboard in developing tools that they can incorporate into their applications to help determine the state of their system during runtime.

Enforcing VAGen Part Naming Conventions

by Jay Cagle, VisualAge Generator Development

In VisualAge Generator Version 3.0 FixPak 4, a set of Smalltalk APIs were introduced that enable users to enforce their own naming conventions for VAGen parts. This article describes those APIs and provides sample code showing how to use them. Also introduced in FixPak 4 is a set of APIs for a number of **VAGen Part** utilities. These APIs enable users to create new and customized part utilities. The utility APIs will not be covered by this article; however, all of the VAGen Smalltalk APIs are documented in the *VisualAge Generator Programmer's Reference* in the Appendix. The online version of the manual was updated with FixPak 4.

To create your own custom naming conventions, you must create a Smalltalk method that performs the name validation. This method is called before a **VAGen Part** is created, but after the part name has already passed the standard VAGen naming conventions. The method must take three arguments: part name, part type, and ENVY application. It must return a boolean value; true indicates the part name is valid.

The ENVY application will be nil if it is not known when the part name is validated. For example, this can happen when adding a main process to a program. The process name is validated at this time, but the application for the process is not known until the process is actually created. Due to this situation, if you use the Envy application in your validation routine you should first check whether it is nil.

The following code shows a sample validation method, and another method used by the validation method.

```
validate: partName type: partType app: app
  "Validate the part name has the following format:      XX##T##
  'XX' must be the same two characters as the application
  the part is being created in.  '##' are any two digits.  'T'
  is a single character indicating the part type."

  "Length must be at least 7"
  partName size < 7 ifTrue: [ ^false ].

  "First two characters must be letters."
  ( (partName at: 1) isLetter and: [ (partName at: 2) isLetter ] )
    ifFalse: [ ^false ].

  "Third, fourth, sixth, and seventh characters must be digits."
  ( (partName at: 3) isDigit
      and: [ (partName at: 4) isDigit
        and: [ (partName at: 6) isDigit
         and: [ (partName at: 7) isDigit ] ] ] )
          ifFalse: [ ^false ].

  "Fifth character must correspond to part type."
  (partName at: 5) = (self characterFor: partType)
  ifFalse: [ ^false ].

  "First two characters must be same as application."
  app isNil
  ifFalse: [
    (partName copyFrom: 1 to: 2) = (app name copyFrom: 1 to: 2)
      ifFalse: [ ^false ] ].

  ^true

characterFor: partType
  "Given a part type, return a single character which
  represents the type:"
  ^( Dictionary new
    at: PartTypeProgram put: $A;
    at: PartTypeProcess put: $P;
    at: PartTypeStatementGroup put: $S;
    at: PartTypeMapGroup put: $G;
    at: PartTypeMap put: $M;
    at: PartTypeRecord put: $R;
    at: PartTypeTable put: $T;
    at: PartTypePsb put: $B;
    at: PartTypeDataItem put: $I;
     yourself )
       at: partType
```

Once you have written your validation method, you must register it so it will be called. The *HptPartValidationHandler* class provides two registration methods: one for adding a validation routine, and another for removing it. Assuming you created the sample methods above as class methods in a class called *VAGenTools*, you would register the method using the following code:

```
HptPartValidationHandler
 addCallbackFor: VAGenTools
  selector: #validate:type:app:
```

To remove the validation routine, you would do:

```
HptPartValidationHandler
  removeCallbackFor: VAGenTools
```

If you place the class containing the validation methods into a utility application, you can set it up so that when the application is loaded into an image the validation routine will be registered. Do this by performing *addCallbackFor:selector:* in the *loaded* class method of the *application* class. The *loaded* method of an *application* class is automatically called when the application is loaded into an image. Likewise, perform the *removeCallbackFor:* method in the *removing* class method of the application class. The *removing* method is called when the application is unloaded from an image.

Naming conventions can help you manage systems with a large number of parts. With these APIs and a little Smalltalk code, it is now easy to enforce your **VAGen part** naming conventions.

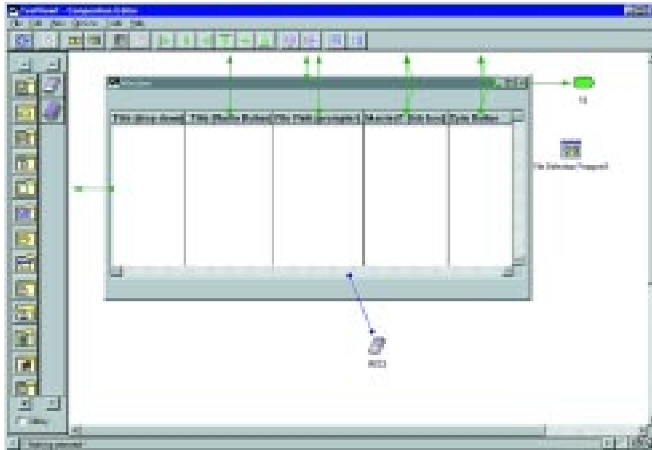
Changing the Edit Policy of a CDV Column

by Guy Slade, VisualAge Generator Development, and Thatcher Robinson, VisualAge Generator Consulting Services

This article is in response to customers interested in how to change the edit behavior of a Container Details View (CDV) cell. Two examples of this are the VisualAge Generator Version 3.0 **Record Editor** and the **properties table view** of any visual parts. This article describes one of many ways to change the edit policy of a CDV column and achieve the same effects as shown in the two examples below. With the power of Smalltalk, you will see that this is a fairly easy task. This example shows you how to apply the following edit policies:

- Combo box
- Radio box
- Prompter
- Toggle button
- Spin button

The following GUI is used as an example:



REC1 is a working storage record with the following definition:

Name	Occurs	Type	Length	Decimals	Scope
TOP	20	Char	69	0	Local
A	1	Char	10	0	Local
B	1	Char	10	0	Local
C	1	Char	40	0	Local
D	1	Char	5	0	Local
E	1	Num	4	0	Local

S1 in the previous GUI is a **Statement Group** and is executed when the *aboutToMapWidget* event occurs. It primes the record REC1 with data.

The *aboutToMapWidget* event also triggers a Smalltalk method called *setColEditPolicies*. This piece of script contains most of what we are about to discuss.

The setColEditPolicies Method

This method contains the following Smalltalk script:

```
setColEditPolicies
   "Since the edit policies for all cells with a particular column are going to be the same we can set
       them up before showing the UI"
   " Set a Combo Box edit policy on the first column "
   (self subpartNamed: 'Title Drop Down') editPolicy:
              ((EwComboBoxEditPolicy on: (self subpartNamed: 'Cell Edit Policy Examples')
                           parentingWidget) items: #('Mr' 'Mrs'); □value: ' ').

    " Set a Radio Button edit policy on the second column "
       (self subpartNamed: 'Title Radio Button') editPolicy:
              ((AbtEwObjectRadioBoxEditPolicy on:
                    (self subpartNamed: 'Cell Edit Policy Examples')  parentingWidget)
                           items: #('Mr' 'Mrs')).

     " Set a Prompter edit policy on the third column "
       (self subpartNamed: 'File Prompter') editPolicy:
              ((AbtEwObjectPrompterEditPolicy on:
              (self subpartNamed: 'Cell Edit Policy Examples')  parentingWidget)
                           editable: true;
                           prompter:(self subpartNamed: 'File Selection Prompter1') ;
                           buttonLabelString: 'Select';
                           buttonAlignment:              XmALIGNMENTEND ).

       " Set a Toggle Button (tick box) edit policy on the four column "
       (self subpartNamed: 'Married Tick Box') editPolicy:
              ((EwToggleButtonEditPolicy on:
                    (self subpartNamed: 'Cell Edit Policy Examples')  parentingWidget)
                           labelString: 'Married?').

       "Set a Spin Button edit policy on the fifth column "
       (self subpartNamed: 'Spin Button') editPolicy:
              ((EwxSpinButtonEditPolicy on:
                    (self subpartNamed: 'Cell Edit Policy Examples') parentingWidget)
                           increment: 10;
                           minimum: 0;
                           maximum: 300;
                           itemType: XmSBNUMERIC;
                           value: 20).
```

As you can see, this method is changing the edit policy for each of the five CDV columns. Let's examine in detail the code that changes the first column.

```
" Set a Combo Box edit policy on the first column "
(self subpartNamed: 'Title Drop Down') editPolicy:
        ((EwComboBoxEditPolicy on: (self subpartNamed: 'Cell Edit Policy Examples')
                              parentingWidget)
        items: #('Mr' 'Mrs');
        value: ' ').
```

The code "(self subpartNamed: 'Title Drop Down')" above identifies the CDV column on which we are changing the edit policy. In this case, we have renamed the *first column* of the CDV from its default to "Title Drop Down."

The code "items: #('Mr' 'Mrs');" is specifying an Ordered Collection of strings that we want to show in the drop down list. In this example, we have hard coded the Ordered Collection. The code *"value: ' ' "* specifies the value to show as selected in the drop down. You will see that we override this later on, so it doesn't matter what you assign at this point. However, It is important to note that the value clause must be set here (even though it is overridden later). If you want the user to be able to type in a value other than the ones displayed in the drop-down list, you must also add the following line of code:

```
"editable: true;"
```

The *second column* is set to use a radio button widget. In this example, the column will display two radio buttons, one for 'Mr' and one for 'Mrs'.

The *third column* is the prompter example. When the user selects a cell in column 3, the current value is displayed in a text box on the left-hand side of the cell and a button is displayed on the right-hand side. If the user clicks the button, the specified prompter window displays. Let's take a closer look at this code.

```
" Set a Prompter edit policy on the third column "
(self subpartNamed: 'File Prompter') editPolicy:
        ((AbtEwObjectPrompterEditPolicy on: (self subpartNamed: 'Cell Edit Policy
                              Examples')  parentingWidget)
        editable: true;
        prompter:(self subpartNamed: 'File Selection Prompter1') ;
        buttonLabelString: 'Select';
        buttonAlignment: XmALIGNMENTEND ).
```

The code "editable: true;" enables the user to type a new value into the cell rather than having to press the prompter button. The code "prompter:(self subpartNamed: 'File Selection Prompter1') ;" specifies the prompter part we want to display. In this case, we dropped a File Selection Prompter onto the free-form surface and named it File Selection Prompter1.

The *fourth column* is set to use a toggle button widget. In the example, when you click on the cell, the text 'Married?' is shown beside the toggle button.

The *fifth column* in this example shows how to have a spin button appear when editing the cell. The parameters in this example are pretty clear. Note that the edit policy is called *EwxSpinButtonEditPolicy*. This edit policy is not loaded into your image by default. To get this edit policy, you must load the application EwExamples.

To get a better understanding of the methods being used above, take a look at the edit policy classes. Open a **Hierarchy Browers** on the class *EwEditPolicy* (From the **System Transcript —> Tools —> Browse Hierarchy**...). By using this browser, you can investigate the various methods we have used above.

To recap, the *setColEditPolicies* method sets the edit policies of each CDV column to something other than the default. Now let's take a look at how we prime the new widget with the value held in the working storage record so that it appears when the CDV cell is selected. Each CDV column has the following connection:

aboutToBeginEdit: (event) <—> *beginEditCellX*.

X represents either A,B,C,D, or E, depending on what column was selected. The following is the code in the method *beginEditCellA:* (a cell in the first CDV column)

```
beginEditCellA: callData


        " Set the initial value that gets displayed in the Combo Box "
        (self subpartNamed: 'Title Drop Down')  editPolicy value: callData value.
```

The *callData* parameter is automatically generated by the *aboutToBeginEdit*: event. One of the values held in *callData* is the value currently displayed in the CDV cell. This method passes the callData value through to the new edit Policy widget. If we don't pass this value, a blank is displayed as the initial value in the cell.

Below is the code for the methods *beginEditCallB:*, *beginEditCellC:*, and *beginEditCellE:.* As you can see, the code for the methods are basically the same as above.

```
beginEditCellB:callData
        " Set the initial radio button "
        (self subpartNamed: 'Title Radio Button')  editPolicy value: callData value.

beginEditCellC: callData
        " Set the initial value that gets displayed "
        (self subpartNamed: 'File Prompter')  editPolicy value: callData value.

beginEditCellE: callData
        " Set the initial value that gets displayed in the Drop Down list "
        (self subpartNamed: 'Spin Button')   editPolicy value: callData value.
```

The method *beginEditCellD:* is different, though. The data item in the record that corresponds to the fourth CDV column contains the string value 'TRUE' or 'FALSE'. This CDV column has a toggle button policy. To set the toggle button on or off, we have to use a Boolean value. The *beginEditCellD:* method needs to convert the TRUE/FALSE string into a Boolean value.

```
beginEditCellD: callData
        | initial |
        " Decode the value held in the record to a true or false Boolean that dictates whether the
        toggle box is selected or not "
        callData value = 'TRUE' ifTrue: [ initial := true] ifFalse: [ initial := false].

        (self subpartNamed:'Married Tick Box')  editPolicy set: initial.
```


The last piece to this puzzle are the two extra connections from the fourth and fifth CDV columns. The connections are *AboutToEndEdit*(event) <—> *endEditCellD*: (and E). Again, the *aboutToEndEdit* event automatically generates the *callData* parameter that is accepted by the *endEditCellD:* (and E) method. Let's take a look at the *endEditCellD*: method first.

```
endEditCellD: callData


        " Decode the boolean value held in the toggle box to a string representation held in the record "
        callData newValue = true
                  ifTrue: [ callData newValue: 'TRUE']
                  ifFalse: [ callData newValue: 'FALSE']
```

This method is doing the opposite of the *beginEditCellD:* method. It is taking the TRUE/FALSE Boolean value and converting it to a TRUE/FALSE string value, then passing that value back to the CDV cell (and hence the working storage record).

Finally, let's look at the *endEditCellE:* method.

endEditCellE: callData

```
callData newValue:          callData newValue asString.
```

This is a strange one. Without going into any detail, this method is overcoming a limitation of a converter method that runs somewhere between the cell edit, finishing, and the new value arriving in the CDV cell. This converter must have the new value in a string format and the *endEditCellE:* method is doing just that.

It is also possible to have a single CDV cell displaying different edit policies. As an example, we can make the edit policy dependent on the cell value: for example, if the cell contains the value 'A', a drop-down edit policy is used; and if the cell contains the value 'B', a radio button edit policy is used. Rather than just setting the initial value in the *beginEditCellX:* method, what you can do is have some logic that tests the current value and sets the edit policy of the cell accordingly. This will have the additional overhead of setting the edit policy of the column each time the user clicks on a cell in that column, but give the flexibility of multiple edit policies in the same CDV column.

Great Lakes Area User-Group Meeting Held in RTP

by Rusty Edmister, VisualAge Generator Sales Support

The Great Lakes Area VisualAge Generator User-Group meeting was held in Research Triangle Park, North Carolina, on July 16 & 17! Over 40 companies were represented and more than 100 people from organizations either using VisualAge Generator or thinking of using VisualAge Generator heard presentations from VisualAge Generator developers and managers, IBM Business Partners, and other customers.

Attendees were also invited to vote for new additions to be included in future releases of the product by investing a mythical $200 and choosing from functions in a shopping list of possible future functions. This was a survey conducted by the RTP Lab to ensure that the future product delivers what its current and future users need and expect.

Networking among attendees was the rule at breaks, before and after the day-and-a-half session, and during the picnic, which featured eastern North Carolina barbecue on Thursday evening. Also at the picnic, attendees were served cake commemorating VisualAge Generator's 4th birthday!

Steve Gilkerson, representing Highlights for Children and the Great Lakes Area User-Group, spoke to the audience about organizing a national VisualAge Generator User's Group. Plans for such a group were incomplete at the conclusion of the meeting. But as information becomes available, we will be providing it to you in future editions of this newsletter as well as on the VisualAge Generator web page at:

www.software.ibm.com/ad/visgen

Math Function Words and Floating Point Arithmetic

by Paul Hoffman, VisualAge Generator Development

In VisualAge Generator Version 3.1, new special function words enable you to use the mathematical functions in the C runtime library from VisualAge Generator programs. Since the C functions operate on double precision floating point numbers, the new function words also support basic floating point arithmetic operations and conversion between VisualAge Generator numeric data types and floating point format.

Coding Math Function Calls

To invoke a math function, code the function name followed by the function arguments and an optional (REPLY option. For example:

```
EZEMAX NUMERICITEM1, NUMERICITEM2,
            RESULT (REPLY ;
```

All the arguments are numeric data items.
The term “numeric data item” refers to any of the following:

- Any data item with the type NUM, NUMC, PACK, PACF, or BIN
- A 4-byte HEX item. The item is assumed to be a single precision, 4-byte floating point number native to the runtime environment. The number is precise to a maximum of 6 digits.
- An 8-byte HEX item. The item is assumed to be a double precision, 4-byte floating point number native to the runtime environment. The number is precise to a maximum of 15 digits.

The (REPLY option indicates how exception conditions are handled by the function. If the (REPLY option is specified, exception codes are returned in EZERT8. If not, the program ends with an error message when an exception is detected.

Math function exception codes are:

8 Domain error; argument is not in a valid range to be operated on by the function.

12 Range error; intermediate or final result cannot be represented as a double precision floating point number, or with the precision of the result parameter.

16 C math function exception.

Floating Point Math Functions

The following math functions are floating point functions. All input parameters are converted to double floating point numbers in the format appropriate for the machine on which the program is running.

EZESIN NUMERICITEM, RESULT ;

Return RESULT = sine of NUMERICITEM.

EZECOS NUMERICITEM, RESULT :

Return RESULT = cosine of NUMERICITEM.

EZETAN NUMERICITEM, RESULT ;

Return RESULT = tangent of NUMERICITEM.

EZEASIN NUMERICITEM, RESULT ;

Return RESULT = arcsine of NUMERICITEM in the range -pi/2 to +pi/2. Domain error if NUMERICITEM not in range of -1 to 1.

EZEACOS NUMERICITEM, RESULT ;

Return RESULT = arccosine of NUMERICITEM in the range 0. pi. Domain error if NUMERICITEM not in range of -1 to 1.

EZEATAN NUMERICITEM, RESULT ;

Return RESULT = arctangent of NUMERICITEM in the range -pi/2, +pi/2.

EZEATAN2 NUMERICITEM, NUMERICITEM2, RESULT ;

Return RESULT = theta component of the polar coordinate (r, theta) corresponding to the rectangular coordinate (NUMERICITEM1, NUMERICITEM2). The result is in the range -pi to pi.

EZESINH NUMERICITEM, RESULT ;

Return RESULT = hyperbolic sine of NUMERICITEM.

EZECOSH NUMERICITEM, RESULT ;

Return RESULT = hyperbolic cosine of NUMERICITEM.

EZETANH NUMERICITEM, RESULT ;

Return RESULT = hyperbolic tangent of NUMERICITEM.

EZEEXP NUMERICITEM, RESULT ;

Exonential function. Return RESULT = e to the power NUMERICITEM.

EZELOG NUMERICITEM, RESULT ;

Return RESULT = natural logarithm of NUMERICITEM. Domain exception if NUMERICITEM <= 0.

EZELOG10 NUMERICITEM, RESULT ;

Return RESULT = base 10 logarithm of NUMERICITEM. Domain exception if NUMERICITEM <= 0.

EZEPOW NUMERICITEM1, NUMERICITEM2, RESULT ;

Return RESULT = NUMERICITEM1 to the NUMERICITEM2 power. Domain exception if NUMERICITEM1 = 0 and NUMERICITEM2 <= 0, or if NUMERICITEM1 < 0 and NUMERICITEM2 is not an integer.

EZESQRT NUMERICITEM, RESULT ;

Return RESULT = square root of NUMERICITEM, NUMERICITEM => 0.

EZELDEXP NUMERICITEM1, N, RESULT ;

Return RESULT = NUMERICITEM1 *(2 to the power of N). N is a 4-byte BIN integer item.

EZEFREXP NUMERICITEM1, N, RESULT ;

Splits NUMERICITEM into a normalized fraction in the range 1/2 to 1, which is returned in RESULT, and a power of 2, which is stored in N. If NUMERICITEM is zero, both returned values are zero. N is a 4-byte BIN integer item.

EZEMODF NUMERICITEM1, NUMERICITEM2, RESULT ;

Splits NUMERICITEM into integral and fractional parts, each with the same sign as NUMERICITEM1. It returns the fractional part in RESULT, and the integral part in NUMERICITEM2.

Floating Point Arithmetic Functions

The following math function words can be used to convert between VisualAge Generator numeric data types and floating point numbers and to perform arithmetic operations with floating point numbers. The double floating point result is converted with rounding to the format of the result parameter.

EZEFLSET NUMERICITEM, RESULT ;

Set RESULT to the value of NUMERICITEM.

EZEFLADD NUMERICITEM1, NUMERICITEM2, RESULT ;

Add NUMERICITEM to NUMERICITEM2 and return the sum in r.

EZEFLSUB NUMERICITEM1, NUMERICITEM2, RESULT ;

Subtract NUMERICITEM2 from NUMERICITEM and return the difference in r.

EZEFLMUL NUMERICITEM, NUMERICITEM2, RESULT ;

Multiply NUMERICITEM by NUMERICITEM2 and return the product in r.

EZEFLDIV NUMERICITEM1, NUMERICITEM2, RESULT ;

Divide NUMERICITEM by NUMERICITEM2 and return the quotient in r. Domain exception if NUMERICITEM2 is 0.

EZEFLMOD NUMERICITEM1, NUMERICITEM2, RESULT ;

Return RESULT = floating point remainder of NUMERICITEM NUMERICITEM2, with the same sign as NUMERICITEM. Domain exception if NUMERICITEM2 is 0.

18-Digit Functions

The following math functions do not require conversion of parameters to double floating point numbers and operate with a full 18 digits of precision. The result is converted with rounding to the format of the result parameter.

EZENCMPR NUMERICITEM1, NUMERICITEM2, N, RESULT ;

Compare NUMERICITEM to NUMERICITEM2 and return n, where n is 1(NUMERICITEM > NUMERICITEM2, 0(NUMERICITEM = NUMERICITEM2, or -1(NUMERICITEM < NUMERICITEM2. N is a 4-byte BIN integer item.

EZEMIN NUMERICITEM1, NUMERICITEM2, RESULT ;

Return RESULT , minimum of NUMERICITEM and NUMERICITEM2.

EZEMAX NUMERICITEM1, NUMERICITEM2, RESULT ;

Return RESULT , maximum of NUMERICITEM and NUMERICITEM2.

EZEROUND NUMERICITEM1, N, RESULT ;

Round NUMERICITEM to the Nth power of 10 and return the result in r. N is a 4-byte BIN integer item.

EZECEIL NUMERICITEM, RESULT ;

Return RESULT = smallest integer not less than NUMERICITEM

EZEFLOOR NUMERICITEM, RESULT ;

Return RESULT = largest integer not greater than NUMERICITEM

EZEABS NUMERICITEM, RESULT ;

Return RESULT = absolute value of NUMERICITEM.

Query functions

The following functions enable a program to determine the precision and length of math function parameters.

EZEPRCSN NUMERICITEM, N, RESULT ;

Return the maximum precision (N) in decimal digits for a numeric data item of this type. For floating point numbers (4-byte or 8-byte HEX item), the precision is the maximum number of decimal digits that can be represented in the number for the system on which the program is running. N is a 4-byte BIN integer item.

EZEBYTES PARAMETER, BYTES RESULT

Return length in BYTES of PARAMETER; BYTES is a 4-byte BIN integer item.

Dare to Develop with a RAD Methodology!

by Henry Jicha, VisualAge Generator Consulting Services

It would be unthinkable to build a skyscraper without a means of prioritizing the work and making sure that dependencies are satisfied for each successive step. For example, a skyscraper would require extensive rework if the steel were to be erected before the foundation work was properly completed. This also shows how important it is to be sure that the most efficient, and thereby least expensive path is taken when moving towards one's goal. In software development, this path to success is called a methodology.

Rapid Application Development (RAD) is a methodology designed to get the maximum amount of function implemented in the shortest possible time. This means that in today's rapidly changing world, a system can be implemented before the requirements change, and it becomes obsolete. Using traditional waterfall approaches, only 3 percent of projects lasting three years or more are successfully implemented!

RAD relies on two things for success. The first is infrastructure, and the second is iterative, incremental development, which includes prototyping. VisualAge Generator is ideally suited to this powerful approach. VisualAge Generator offers extensive screen painting/template-based application generation techniques, as well as interactive testing capabilities. You can use these facilities for design prototyping and subsequent user functional validation during the incremental development process.

Infrastructure

For RAD to be most effective, there must be a strong infrastructure in place at the start of the project. Infrastructure elements include naming conventions, data definitions, screen standards, and a robust architectural model. Several means can be found for achieving this infrastructure. The easiest is relying on the infrastructure already in place. This is particularly applicable to projects that add function to existing systems. With existing systems, a stable set of data definitions are usually already in place to support production. Likewise, screen standards and naming conventions have generally been established by precedent within existing systems, and will need to be followed for consistency with the existing infrastructure.

Another way to build an infrastructure, particularly in new application areas, is through the use of template-based development. Using templates, VisualAge Generator can build nearly complete applications using only the data definitions. Here, flexibility in the screen and architecture is traded for the ability to absorb changes in the data model. If the data model changes, all you need to do is re-create the application using the new data definitions with the templates. The application architecture is provided by the templates, as are the screen standards. An additional advantage of the new template technology in VisualAge Generator Version 3.0 is its ability to provide a high level of flexibility along with the ability to absorb change in the data definitions that underly the application. This is achieved by allowing screen layout modifications to be preserved during regeneration of the application through the use of templates if the data definitions change. Additional flexibility is provided to modify the templates to achieve new architectures or conventions to suit specific requirements. You can extend this to include existing manually coded infrastructure in new applications through reverse engineering of existing applications into templates that can then be used to create new application functions.

Data is the most critical infrastructure component. Changes in data can be accommodated with template-based development to avoid expensive human rework. However, if custom screens or GUIs are required, stable data definitions are needed before extensive application definition work begins. Then, using a consistent architectural model facilitates incremental development in an orderly fashion. This avoids rework to make components developed in the future connect with those developed previously. Naming conventions also contribute to this by assuring that developed parts can be easily located and reused where possible. In addition, naming conventions help prevent duplicate development of the same functions.

Incremental Development

A cube can be used as a model to look at the dimensions of the application complexity. Each dimension of the cube represents a different metric. On the face of the cube, the horizontal metric is application breadth, which can be measured in terms of different functions or functional areas of the application. The vertical dimension on the face is the depth of functionality in each functional area. The last dimension, depth, is then the robustness of the application, starting with prototyping, and finishing with full error handling for all possible scenarios.

When developing applications, it is often useful to break up the effort

based on the cube model. For example, different groups can be assigned to different application areas so that parallel efforts are possible without dilution of subject area knowledge. With the appropriate infrastructure in place, these efforts can later be easily combined.

Likewise, the cube is useful for looking at iterative development techniques for code creation. The robustness dimension is divided into three tiers: 1) screen interaction and prototype, 2) logic development and validation, and 3) error handling and recovery, help, and online documentation. This allows an iterative development process to be implemented based on incremental development of these functional layers.

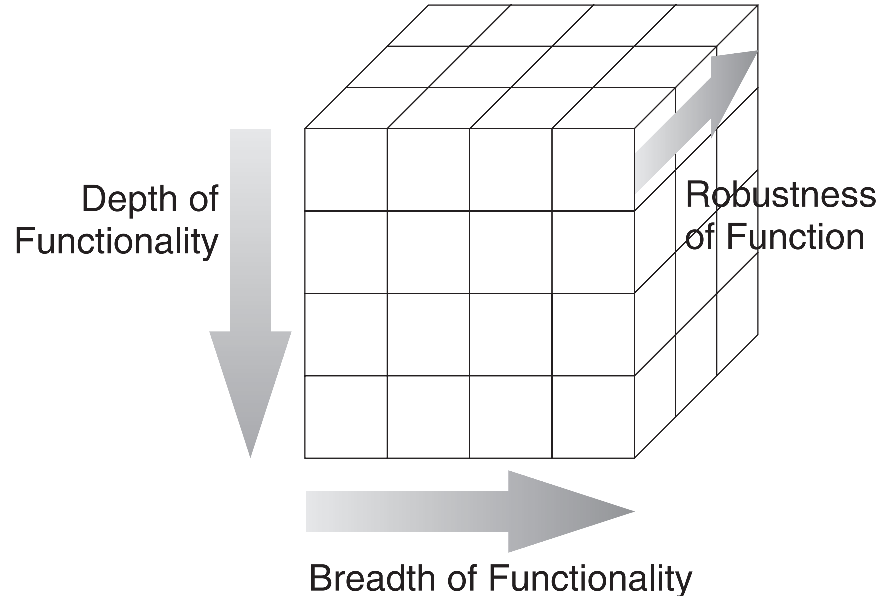
First, define screens and user interface elements, along with basic navigation. You can then have the user review this using ITF capabilities to validate the specifications. By doing this, expensive coding of functions that do not meet the user needs is avoided, and you have an opportunity to fine-tune requirements early on with the advantage of real screens to help visualize the functionality. This becomes the first round of the iteration.

You can now proceed to code the application logic. You now have a good specification of the flow and the requirements have been validated. Reusable logic parts can also be identified and coded as parts to be included in the overall coding effort.

Templates can also be modified (if desired), or used directly to create a working application that performs all of the functions described in the user requirements. You can again have the user validate this, and make the necessary adjustments. The advantage with templates like these in VisualAge Generator is that much of the additional coding for error handling and help and the like does not need to be done, eliminating rework if the specifications are adjusted based on the user evaluation of the system's functionality.

Finally, when the functional model has been approved, and the second phase of iteration is completed, error handling, help, and additional functions can be coded with the assurance that this work is not being wasted on a function that will not survive in the final product.

Dimensions of Application Development



Summary

In summary, RAD offers a practical approach to development that you can adapt to a variety of circumstances. With proper planning and guidance, RAD can address the creation of enhancements to existing applications, or new development in areas that have never been automated. Be aware, however, that the implementation of methodologies such as RAD is not trivial, and often involves significant cultural changes in an organization that take the time and effort to make it happen. Implementation of a RAD tool like VisualAge Generator can greatly facilitate this process, but it is not the total solution.

Proper use of the cube model allows for easy decomposition of a project into manageable parts by functional areas, functions within the areas, and levels of implementation of the functions. This can be useful in developing a project plan. Additionally, the cube model can be used as a guide to help develop techniques to implement an iterative development approach. The iterative approach assists user acceptance, and provides for multiple signoffs of specifications at points in the development cycle designed to maximize development efficiency. Additionally, the three-phase iterative approach maximizes use of the VisualAge Generator strengths in prototyping and interactive testing capabilities. When coupled with a strong infrastructure based on data, standards, and/or templates to guide development, the productivity results can be extraordinary!

ASRA Abends Happen!

by Theresa Smit, Heather Albright, and Mark Evans, VisualAge Generator Consulting Services

Sometimes developing applications can be mind-boggling—you never know how they'll turn out!

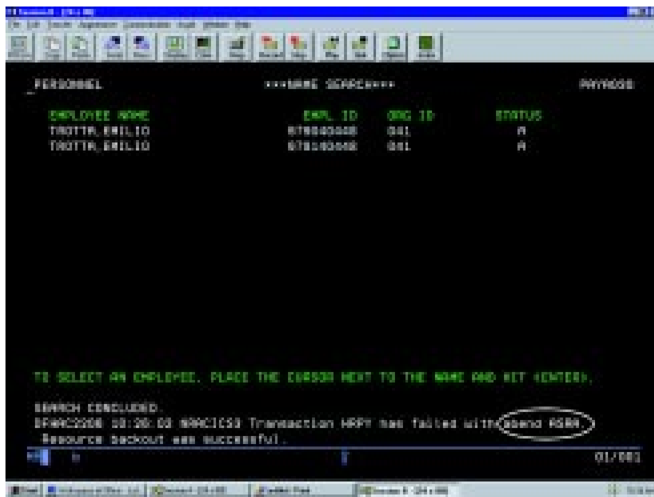


Figure 1

Whether the application is a TUI or a GUI calling a CICS server, an ASRA abend might occur.

The presentation might be displayed or formatted differently from Figure 1, but you still need to gather the same type of information from various resources. You need to gather the following information:

- What statement in the generated COBOL program was executing when the ABEND occurred?
- What VisualAge Generator script statement generated the COBOL statement?
- What are the data elements, values, and types involved in the failing statement?

This article discusses three ways to get information about the ASRAs:

- The VisualAge Generator error log temporary storage queue.
- The CICS execution debug facility (CEDF).
- The Interactive Test Facility

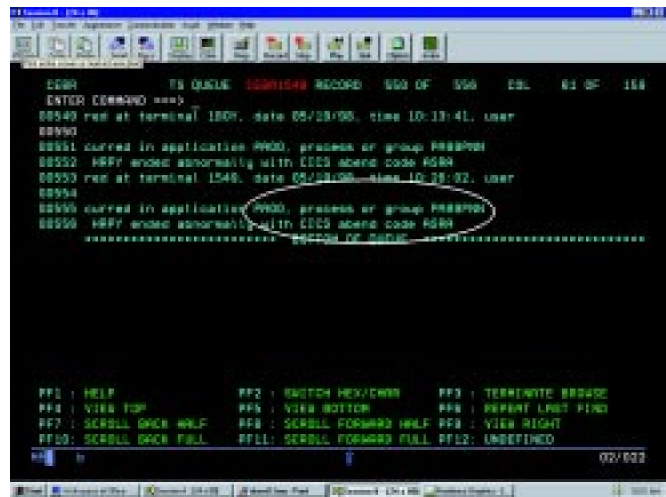


Figure 2

VisualAge Generator keeps a temporary storage queue called ELAD. You can use the CICS transaction, CEBR, to view the temporary storage queue contents. To view the queue, do the following:

1. Enter GET ELAD on the command line.
2. Press **PF5** to display the contents at the bottom of the queue where you will find a message containing the name of the application and the process where the ASRA occurred.
3. Use **PF9** to scroll right to show the rest of the message as shown in Figure 2.

In the Figure 2 example, VisualAge Generator Host Services has detected the abend in process PA99PNN of Program PA00. The information log remains until CICS is recycled or you purge the queue by typing PURGE on the command line.

The application name in the log will always be correct, but there are times that the failure actually occurs in a statement group whose name is not specified. Use the information in the log as a starting source.

If you have any of the commercially available COBOL debuggers and your VisualAge Generator generated source code, it will be much easier to find the abending statement. However, if the CICS Execution Debugging Facility is all you have, start it by entering CEDF on the command line. If you are debugging a Client/Server called application, you need to determine the terminal ID or connection ID that the called transaction is using. To do this, do the following:

1. Enter CEMT I CONN at the ready prompt on the host.
2. Append this ID to the debugging facility transac-

tion ID (that is, CEDF term/conn-id).

3. Once the message ‘THIS TERMINAL: EDF MODE ON’ displays, press the **Clear** key and re-create the call to the server, or enter the transaction ID for the TUI transaction. The CICS Execution Debugging Facility then takes over to display each CICS command before and after its execution.

   The initial program entry screen displays the starting contents for the COMMAREA. This is especially valuable to client/server programs since this is where the data contents are passed. You can press **PF2** to see the storage address where the complete COMMAREA data is stored.

4. Press **PF5**, “Display Working Storage,” to get to the address and view the data passed, then overlay the beginning of the working storage address with the address of the COMMAREA and press **Enter** to view the contents.

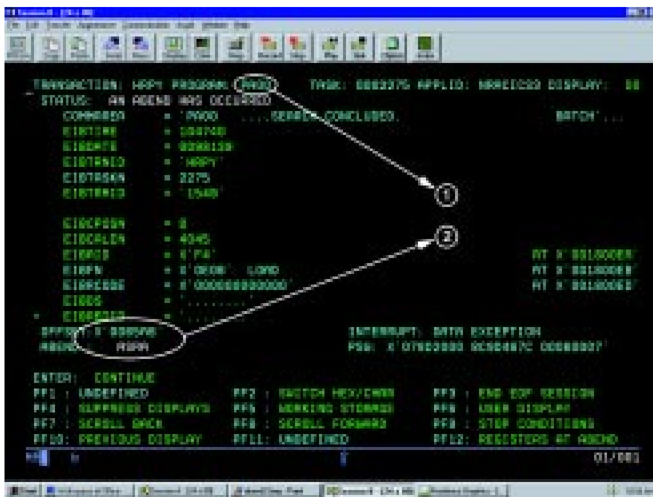


Figure 3

To get to the ASRA quicker, press **PF4** to suppress the intermediate displays. You can also control what CICS command the debugging facility stops on by pressing **PF9**.

Note: CEDF also stops when an abnormal response is encountered (to continue to suppress displays, press **PF4**), and also on program and task termination (respond with **YES** to continue, then press **Enter** to continue debugging).

To minimize the number of debugging screens, suppress them by:

- Using PF9 to specify CICS Stop conditions, rather than specify a specific stop command, specify NO for both normal and abnormal termination.
- Specify RECEIVE as the CICS stop condition. Once you have responded to the map prompt just prior to the ASRA, remove the stop condition and begin trapping each CICS command or set another more specific stop condition.

Remember, PF5 displays the program’s working storage contents. In the generated COBOL program, each record definition generates a header that contains a character field initialized with the record name. So as you are paging through the working storage, you can easily spot the records. You can even change the data in working storage to affect the subsequent execution path.

5. Once you get the ASRA (see Figure 3), make a note of the Program name and Offset address of the program’s abending instruction.

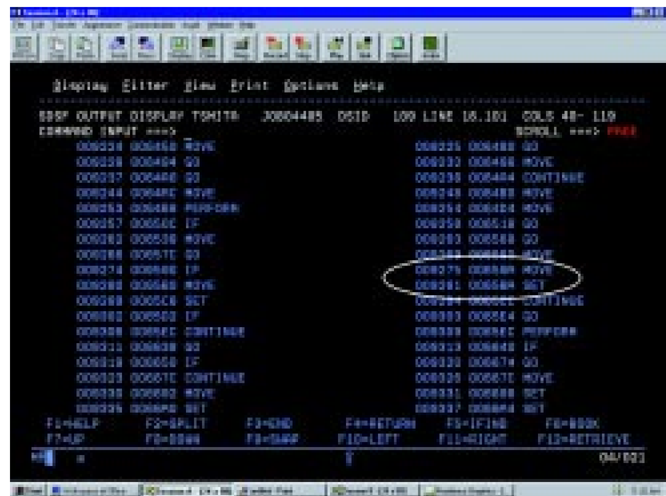


Figure 4

6. Refer back to the compiler listing to find the COBOL statement at the ASRA offset. Note that you must have compiled the program with the OFFSET compile option.

   Ensure that the compiler listing that you are viewing matches the version in which the Abend occurred. To validate this, press **PF5** to display the working storage, the compile date and time is located in the character mapping on the right of the screen. If you are unsure, recompile the program, new copy it into CICS, and repeat the CEDF tasks to get the new offset.

7. To locate the offending instruction:

   a. Go to the bottom of the compiler listing

   b. Search backwards for the word ‘verb’ (F verb prev).

   There will be three sets of three columns that contain the COBOL statement number, the program offset, and the COBOL verb in the statement (scroll right to see the third set).

c. Browse forward through the columns (order is left, then right) until you find the offset closest to being less than or equal to the Abend offset as shown in the circled area in Figure 4. The offset usually falls between two COBOL statements, since multiple Assembler statements are generated for each COBOL statement. The program offset column lists the first statement.

d. Record the COBOL statement number (the number in the first column).

e. Search backwards for this statement number. (You may have to repeat the search for large listings.)

If you generated the program with Comment Level = 4 [/COMMENTLEVEL(Statements)], the VAGen Script statements will appear as comments preceding the COBOL statements that they generated.

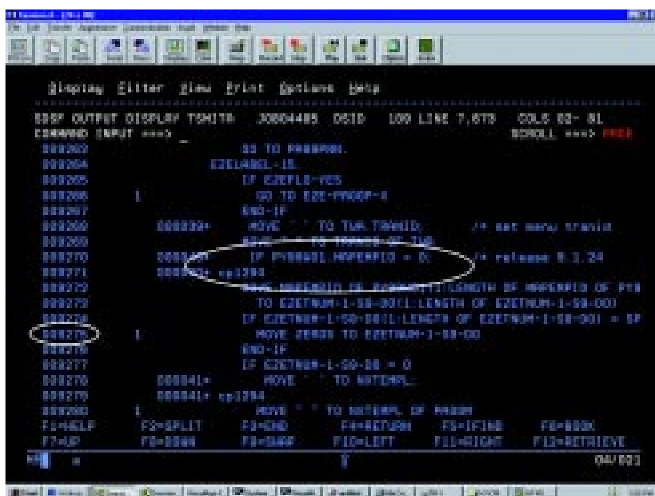


Figure 5

8. Note the VisualAge Generator Script statement and number. Then scroll up in the listing until the process name block is displayed (as shown in Figure 6).

   Now that you have identified the process and the statement where the abend occurred, you will need to determine why the abend occurred.

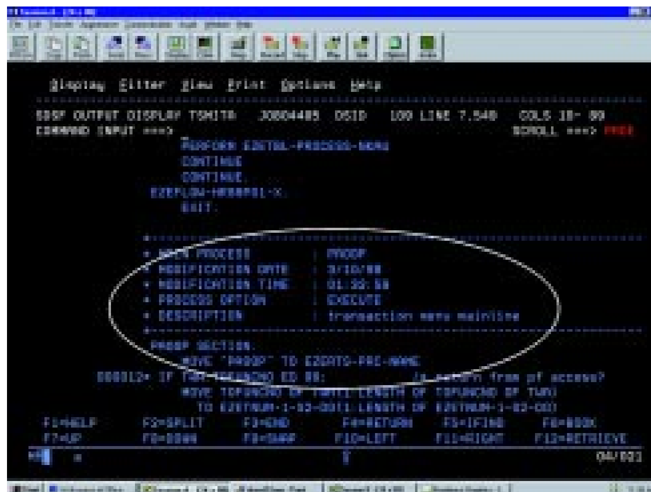


Figure 6

The most likely cause is a numeric field that does not contain numeric data. How can this happen, you ask? The table at the bottom of the page provides some of the possibilities and corrections.

Cause	Correction
1. Uninitialized numeric field.	Use the /INITREC and /INITADDWS generation options or set the record empty at the beginning of the program.
2. Numeric field contains spaces, from parent field that was a character field and had spaces moved to it or a lower-level character field was initialized to spaces.	This can be resolved with the /SPZERO generation option or moving 0 to field, if spaces are present.
3. Numeric field contains non-numeric data not spaces, from field that is used for numeric and character data.	Test a higher level character field for numeric before usage in IF, MOVE, or calculation.

Note that using the /INITREC and /INITADDWS generation options causes a slight performance degradation. The /SPZERO generation option will have an even greater impact on performance. Be selective when using these generation options in applications.

9. Run the program using the Interactive Test Facility (ITF) and observe the contents of the data fields to determine their values. Set breakpoints on the data and process, and watchpoints on the field(s).

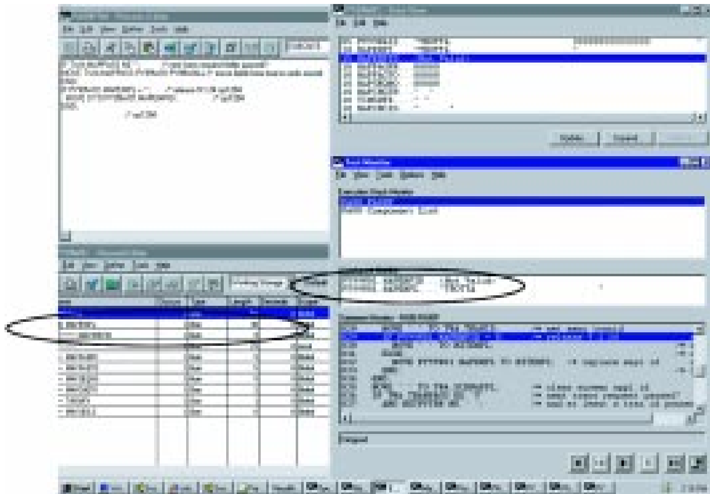


Figure 7

In Figure 7, you can see that MAPEMPID is not valid because MAPEMPID character parent field is set to 'TROTTA'. This is why a data exception occurs when the numeric field MAPEMPID is compared to 0.

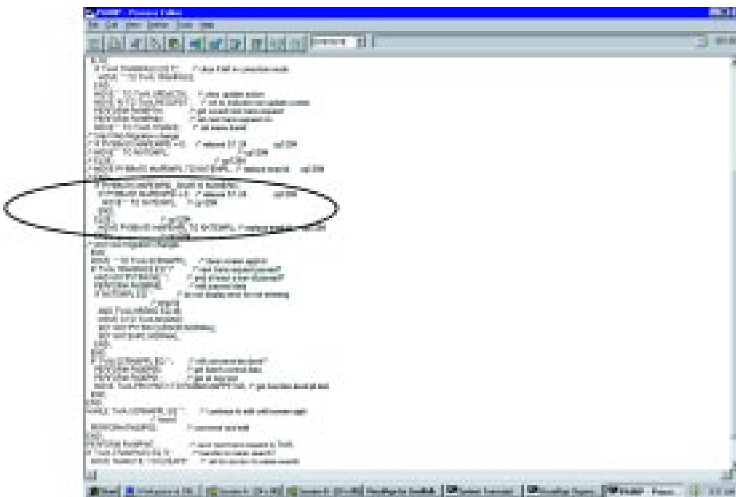


Figure 8

10. Double-click on the process name in the Execution Stack to open the **Process Editor**. You can then change the code to check a new character field (that is a parent of the numeric field) for NUMERIC before the numeric compare. Doing this causes the comparison not to execute if the field contains nonnumeric data.

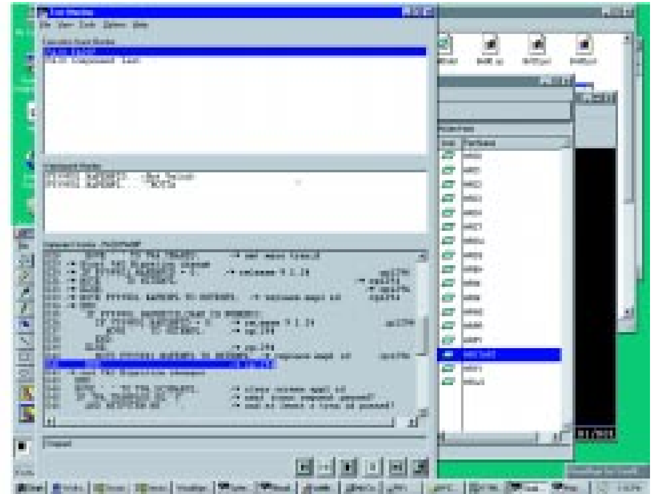


Figure 9

Figure 9 shows that in retesting the program in the ITF, the compare to 0 does not execute. Now the program should not fail on this statement once it has been regenerated, compiled, linked, and new-copied to CICS!

Object Connection—Partners in Development

by Greg Nash, Relationship Manager, Object Connection Program

The Object Connection Program is designed to encourage commercial software developers to explore the business of building software components with VisualAge-enabled and reusable software componentry.

In addition to component development, the Object Connection Program supports software solution providers building VisualAge-complementary tools and end-user applications.

The program provides members with valuable benefits including:

- Loaned VisualAge development licenses
- Technical support for development and enabling with waived fees
- Product listings in the VisualAge Resource Catalog
- Free setup for Internet electronic sales and delivery

IBM also expands the VisualAge Object Connection support program for software solution providers to include VisualAge Generator. Through this program, commercial software developers can obtain IBM's top of-the-line VisualAge application development tools for use in the development of their software products. They also get technical support and several "go to market" programs, including advertising and access to an electronic delivery channel. There is no charge for membership.

This program is part of IBM's larger Solution Developer Program, designed to make all of IBM's product brand technologies easily accessible to the commercial software development community. Together, IBM and the Object Connection members provide a broader and richer set of application development tool solutions than any other development tool provider. To register for the Solution Developer and VisualAge Object Connection Programs, visit the program home page at:

`http://www.developer.ibm.com`

To e-mail us, send your e-mail to:

`objconn@us.ibm.com`

You can contact us by phone at:

`1-800-627-8363 (U.S. and Canada)`

`1-770-835-9902 (Worldwide)`

Our Fax number is:

`1-919-254-0472`

Year 2000 Support—Fact and Fiction

by Lisa Lasher, VisualAge Generator Development and Sue Royer, VisualAge Generator Sales Support

There has been a lot of confusion concerning Year 2000 support and the CSP and VisualAge Generator products. This article will try to set the record straight.

What Does Year 2000 Ready Mean?

According to IBM's Year 2000 Product Readiness database, "Year 2000 ready means that a product, when used in accordance with its associated documentation, is capable of correctly processing, providing, or receiving date data within and between the 20th and 21st centuries..."

This is a rather broad definition that can easily be misinterpreted as it relates to CSP and VisualAge Generator.

First, the definition does not require products to support a 4-digit year format. It just says that, within its documented functionality, a product must behave correctly through the 21st century. This is an important point to understand in regard to CSP 3.3. CSP 3.3 is clearly documented as supporting only 2-digit years; however, it will handle the turn of the century correctly, so it is listed as Year 2000 Ready. Earlier releases of CSP will not accept 00 as a valid year, so they are marked as Not Ready.

Second, the definition does not guarantee that your generated applications are automatically Year 2000 ready. If your applications always use 4-digit year formats, then they are probably Year 2000 ready. But if your applications ever use 2-digit formats, then you will need to assess them to determine if they need to be modified. The considerations are the same as if you were coding in a 3GL: date fields will need to be expanded or you will need to add code to interpret them correctly. For more information, refer to the white paper "VisualAge Generator, CSP, and the Year 2000" by Allan DeLoach at:

`http://www.software.ibm.com/year2000`

In a Nutshell

The table at the bottom of the page summarizes the Year 2000 readiness of the CSP and VisualAge Generator products. Each row in the table refers to the entire product family for that release. For example, CSP V4.1 includes CSP/370AD 4.1, CSP/370RS 4.1, CSP/2AD 1.2, and CSP/2RS 2.1.

What Are the APARs For?

In the table below, several APARs are listed as either optional or required. CSP/AE 3.3 requires the specified APAR in order to process 00 as a valid year. You must install this APAR to make the product Year 2000 ready. The optional APARs in the table correct a usability problem with the MSL member list in CSP/2AD 4.1 and VisualAge Generator Developer 2.x. The MSL member list uses a 2-digit year format for the timestamp. Without the APAR, the year will show as 100, 101, etc., instead of 00, 01, etc. The APARs are marked as optional because the product still interprets the dates correctly, including during a sort.

End Of Support ?!?

Another area of confusion has been over the end of support dates. You might have seen two different dates published, and you may also be worried that the date is too early for you. Early last year IBM made a statement of corporate policy that all products that are Year 2000 ready will remain in support until May 2000.

Product	Year 2000 Ready	Accepts 00 as Year	4-digit Year Format	APARs	End of Support
CSP V3.2.2 and earlier	No	No	No		Not Supported
CSP V3.3	Yes	Yes	No	PN87466 (required)	12/31/98
CSP V4.1	Yes	Yes	Yes	PN91531 (optional)	01/31/2001 or later
VisualGen V1.0, V1.1	Yes	Yes	Yes		Not Supported
VisualAge Generator V2.0	Yes	Yes	Yes	PN91532 (optional)	Not Supported
VisualAge Generator V2.2	Yes	Yes	Yes	PQ00211 (optional)	01/31/2001 or later
VisualAge Generator V3.0, V3.1	Yes	Yes	Yes		01/31/2001 or later

In keeping with this policy, CSP and VisualAge Generator published this end-of-support date. Since then, as we have learned more about the magnitude of the Year 2000 problem, IBM has revised its policy, and has committed to keep these products in support through January 2001. We are in the process of updating this information for CSP and VisualAge Generator, but you might still see references to the earlier date.

If you don't think you will be ready to migrate to a newer version of VisualAge Generator even by 2001, there's no need to panic. CSP 4.1 and VisualAge Generator will remain in support at least through January 2001. The actual date for end of support will be determined by a number of factors, including customer usage and the stability of the product. It is IBM's policy to announce withdrawal from support at least one year in advance. The only product for which we have made such an announcement is CSP 3.3, as indicated in the table above.

What About COBOL?

Many people have wondered whether VisualAge Generator requires a Year 2000 ready version of COBOL, such as LE COBOL. The answer is no. VisualAge Generator uses a windowing technique to determine the 4-digit year for system dates, so it does not require any Year 2000 capabilities in the compiler. So you can still use older versions of COBOL, such as COBOL II. Conversely, with COBOL II going out of support, people have wondered whether older versions of CSP and VisualAge Generator will work with newer versions of COBOL. The answer to that question is yes; you can upgrade your version of COBOL without needing to recompile your generated applications.

For More Information...

Visit the IBM Year 2000 home page for more information about Year 2000 readiness at:

http://www.software.ibm.com

This site includes a wealth of information, including the Year 2000 Product Readiness database. Under the **Tools** link, you can go directly to the white paper referred to above, as well as to the ESFSCAN tool, which can help you analyze your applications for Year 2000 readiness.

VisualAge Generator Web Pages

The VisualAge Generator web address is:

`www.software.ibm.com/ad/visgen`

For IBM's predecessor 4GL, Cross System Product, the web address is:

`www.software.ibm.com/ad/visgen/csp`

Comment Form

Please check any appropriate boxes:

- ☐ I'd like to receive future issues of this newsletter. (You need to check this item only if you have not already responded.)
- ☐ I'd like more information about Version 3.1.
- ☐ I'm interested in writing an article to include in *The VisualAge Generator Newsletter*. Subject:____
- ☐ I'm interested in participating in an AD users' group meeting.
- ☐ I'm interested in participating in a VisualAge Generator users' group meeting.

I have a question I'd like to submit for the Question & Answer section of this newsletter:

Are we putting the type of information you want to see in the newsletter? If not, what would you like to see in the newsletter?

Any comments you'd like to share with us about VisualAge Generator or about this newsletter? (Include your comments or concerns about VisualAge Generator's future directions here.)

Name	Title
Company Name	
Street Address/P.O. Box	
City	State/Province
ZIP/Postal Code	Country
Phone No.	FAX No.

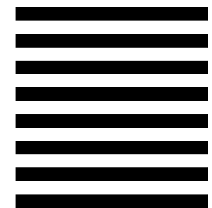
Fold, tape, and mail this page - no postage is required. Or FAX it to (919) 254-0206.

G242-0315-09

Cut or
Fold Along
Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines
The VisualAge Generator Newsletter
Newsletter Editor
TF6B/062/J125
P.O. Box 12195
RTP, NC 27709-2195
USA

Fold and Tape **Please do not staple** Fold and Tape

G242-0315-09

Cut or
Fold Along
Line

A Question From Us To You

Do you have questions? If so, use the Comment Form in this newsletter to send us your questions. Then, in future issues of the newsletter we will provide you with articles in response to your questions.

Acronyms

3GL	third-generation language
4GL	fourth-generation language
AIX	Advanced Interactive Executive
API	Application Programming Interface
AS/400	Application System/400
CAE/2	Client Application Enabler/2
CASE	Computer-aided Software Engineering
CICS	Customer Information Control System
CICS OS2	Customer Information Control System Operating System/2
CPU	central processing unit
CSP	Cross System Product
DB2	Database 2
DBCS	double-byte character set
DBMS	database management system
DCE	distributed computing environment
GUI	graphical user interface
IBM	International Business Machines
IMS	Information Management System
LAN	Local Area Network
MSL	member specifications library
MVS	Multiple Virtual Storage
NT	Notes
OS/2	Operating System/2
OS/400	Operating System/400
RAD	rapid application development
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/ Internet Protocol
VM	Virtual Machine
VSE	Virtual Storage Extended
WWW	World Wide Web

The VisualAge Generator Newsletter

This newsletter is published by the IBM Software Solutions Division, Research Triangle Park Development Laboratory. Letters to the editor are welcome. Please address correspondence to:

The VisualAge Generator Newsletter
Managing Editor
IBM Corporation
Dept. TF6B/062
P.O. Box 12195
3039 Cornwallis Road
RTP, NC 27709-2195
USA
FAX: (919) 254-0206

© Copyright International Business Machines Corporation 1998. All rights reserved. Printed in U.S.A.

The following terms used in this publication are trademarks or service marks of the IBM Corporation in the United States or other countries or both: AIX, AS/400, CICS, CICS OS2, COBOL, Database 2, DataJoiner, DB2, DB2/2, DB2/6000, IBM, IMS, MQSeries, MVS, VM, VSE, Operating System/2, OS/2, OS/400, RISC System/6000, VisualAge, and VisualGen.

The following terms and phrases used in this publication are trademarks or service marks of other companies:

Lotus Notes is a trademark or registered trademarks of Lotus Development Corporation.

Java and JavaBeans are trademarks of Sun Microsystems, Inc.

Oracle is a trademark of Oracle Corporation.

PowerBuilder is a trademark of Sybase, Incorporated.

Planetworks and Interspace are trademarks of Planetworks L.L.C.

Nikon is a trademark of Nikon Corporation.

ENVY is a trademark of Object Technology International, Inc..

HP is a trademark of Hewlett-Packard Company.

Microsoft, Windows, Windows NT, the Windows 95 logo, Visual Basic, and ActiveX are trademarks or registered trademarks of Microsoft Corporation.

Other company, product, and service names may be trademarks or service marks of others.

IBM has made reasonable efforts to ensure the accuracy of the information contained in this publication. However, this publication is presented "as is" and IBM makes no warranties of any kind with respect to the contents hereof, the products listed herein, or the completeness or accuracy of this publication. Customer experiences may be different from those described here. IBM does not warrant any non-IBM programs or products which are described in this newsletter. These articles are for information only, and you should contact the stated company with your questions.

The VisualAge Generator Newsletter
IBM Corporation
Dept. TF6B/062
P.O. Box 12195
3039 Cornwallis Road
RTP, NC 27709-2195
USA

G242-0315-09