CenturaPro™

Formerly Gupta Pro

Hot Ideas for Centura® Developers

Five Habits of Highly Effective Class Design

Tony Vinayak

Some build 'em, some buy 'em. If you belong to the former sect of class library aficionados, read on. This article, paraphrasing one, Mr. S. Covey, proposes the most widely used features of an application design that are best incorporated within the base class hierarchy of a SQLWindows/Centura application.

OOP has been around for awhile now; it's pretty hard to imagine a software development project starting development without making use of class libraries these days. Careers have been made out of designing base classes to support construction of big (and small) SQLWindows applications. In fact, most large projects often designate a couple of experienced developers as the exclusive custodians of class libraries. They're the ones preoccupied with cleverly, proactively anticipating the common coding requirements of other developers on the team. They need to come up with an effective design, a sound framework of classes that is:

- Easy to use.
- Easy to enhance and override.
- Able to reduce other developers' headaches; encapsulate any low-level, hard-to-code functionality.
- Able to deliver built-in functionality that's common to most parts of the application.

Designing classes, some say, is a state of mind. I hope this article will get you into that mood by enlisting the top empirical features that should find

Continues on page 5

July 1996

Volume 1, Number 6

PRO Publishing

Less is More

Mark Hunter

Last month, the editorial was entitled "Dollar Signs." Just after it went to press there were some unusually favorable dollar signs from Menlo Park. For the first time in two years, Centura Software posted a small quarterly profit. This will soothe the nerves of some long-suffering business partners and customers, not to mention the stockholders! The company has lost some "paper" value due to writedowns related to their reorganization, but their cash position remains strong, with about $15 million in the bank.

I'm just back from the User Conference in Palm Desert, and as usual there's a lot to think about. In this extended editorial I'll tackle just a few of the more interesting topics. If you see references to unfamiliar product names, skip ahead to the table of announced products for some details.

For the first time, Centura Software publicly stated their goal for complete Java compatibility by early 1997, in conjunction with the major language enhancements collectively known as COOL. Java will have a dramatic impact on how developers deploy their work: Centura applications run on many hardware platforms, act as distributed objects, become easier to deploy, and offer excellent runtime performance as just-in-time compilers for Java become more widespread and efficient.

Even more important than what Centura is doing is what they're *not* doing. They're not writing their own new interpreter or compiler–and that's good news. Centura Software has too much to do already, and not enough resources to do everything quickly. Less development responsibility, while maintaining quality, means more opportunity to accomplish their remaining tasks. Despite the difficult initial work of becoming Java-compliant, they'll experience fewer maintenance burdens and more marketing openings in the future.

At the conference I attended a session on SApplication Builder, the SAP R/3 interface product, and came away even more impressed than before. This is another excellent technical product, available by the time you read this, which can produce powerful marketing advantages. And, again, Centura was able to license the product instead of building it with their own engineers. Good move. Less resource consumption and more opportunity for the company and its customers.

Editor Mark Hunter, **Publisher** Dian Schaffhauser, **Business Manager** Shelley Doyle, **Page Manager** Paul Gould, **Bone Manager** Mocha

Centura Pro is published monthly (12 times per year) by Pro Publishing, PO Box 18288, Seattle, WA 98118–0288.

POSTMASTER: Send address changes to *Centura Pro*, PO Box 18288, Seattle, WA 98118–0288.

Subscription information: To order, call Pro Publishing at 206-722-0406. Cost of domestic subscriptions: 12 issues, $129; 24 issues, $199. Canada: 12 issues, $144; 24 issues, $229. Outside North America: 12 issues, $159; 24 issues, $259. Individual issues cost $12 ($15 outside the U.S.). All funds must be in U.S. currency.

Centura technical support: Call Centura Corp. at 415-321-4484.

If you have questions, ideas, Voodoo PC horror stories, or would just love to chat about what you're doing with Centura products, contact us via one of the means at right.

Call the Mothership!

Editorial Command Post
Phone: 818-249-1364
Fax: 818-246-0487
E-mail: 71460.3142@compuserve.com

Subscription Headquarters
Phone: 206-722-0406
Fax: 206-760-9026
E-mail: 71333.2142@compuserve.com

Mail
Pro Publishing
PO Box 18288
Seattle, WA 98118-0288

Source Code on CompuServe
GO CENTURA, Library 10

Compiler suffers from Java overdose

The Centura Object Compiler is moving simultaneously toward obsolescence and increased use. In the 32-bit world, Centura is devoting very few resources to enhancing the compiler, and it will probably never reach the 100% compilation level. This is entirely due to the planned inclusion of the Java Virtual Machine in the Tomahawk release of Centura Team Developer. Just-in-time compilers for Java are already common and practical; by the time Tomahawk is released, runtime support for Java byte code should be superb, and Centura can capitalize on that. In such an environment, a compiler that works on limited operating systems and processors just can't compete.

At the same time, you can expect the C++ compiler technology for SQLWindows to get *better.* Java features are not coming to SQLWindows, and as part of Centura's continued commitment to SQLWindows, the company will provide an improved compiler in the Cyclone release in the fourth quarter.

This provides some explanation for the slow pace of compiler progress in the past year, which upset some customers. According to Centura engineers, since they began seriously considering Java in 1995, the prospect of moving to that language acted as a constraint against aggressively improving the compiler. Let's hope that Centura Software now recognizes the potential of its Java initiative and moves aggressively to implement it.

Centura applications will run on many hardware platforms, act as distributed objects, become much easier to deploy, and offer excellent runtime performance . . .

Softening the impact

Europe is moving toward 32-bit desktops even more slowly than the United States is. In response to this and to other pressures, Centura announced that the Patriot version of Centura Team Developer, due in the third quarter, will permit CTD developers to deploy their applications on Windows 3.1 desktops. Some features that are deeply linked to 32-bit platforms, like Internet QuickObjects, won't work on Windows 3.1. And source code won't move backward from Centura to SQLWindows (although some third parties may change that).

Small steps toward the Web

Before complete Java compatibility, the Centura Web Developer product will provide Web browsers with the ability to connect to Centura applications. The Web Developer is the latest incarnation of the Web Data Publisher discussed in previous months.

Running in conjunction with the Centura Application Server (the three-tier enabler for Centura applications), Web Developer allows apps to create dynamic HTML at runtime instead of displaying windows. So a user with a Web browser would see an HTML page that looks similar to the top-level window displayed by a conventional Centura application. All this is accomplished without any changes to the application source code; the "JAWS" server handles the translation.

Naturally, only visual objects supported by HTML version 3 are generated. So if your table window has columns with checkboxes and drop-down lists, it's not going to translate very well. Simple table windows will, though.

Centura Software pointed out some likely uses for this product, such as a SQLBase "data mart" located outside the corporate firewall, with a Centura application to fetch rows from the database and format them in HTML.

Internet QuickObjects, now in beta, are almost the mirror image of the Web Developer. The principal purpose of this feature is to provide Web browsers that are operated from within Centura applications. Apps may read and parse HTTP data with either a visible or invisible browser. Some developers who had been awaiting this feature were hoping to see low-level Windows Sockets capabilities included, but they're not in the first release. Michael Alessio of Centura confirms that it's a widely requested feature and at the top of the list for inclusion in the next version, due with Tomahawk.

Mixed messages from SQLBase

In the final question-and-answer session of the conference, Centura management politely but firmly vetoed a suggestion by a customer that SQLBase should be unleashed to run on very high-performance platforms and large databases. Management emphasized again that they don't want to play in the fiercely competitive world of high-end relational databases, and prefer that SQLBase be known for reliability, flexibility, and small "footprint" in mobile and departmental servers.

However, they also announced upcoming SQLBase support for symmetrical multi-processor machines and improved threading capabilities on multi-threading operating systems. Sounds like a stealth project to grow SQLBase when no one's looking.

Several customers voiced concerns about stability in SQLBase 6.0.1, and an impromptu session to discuss their experiences was held after the conference ended. PTF 7, available now for SQLBase 6.0.1, may solve some of the reported problems.

SQLBase Voyager, due in the fourth quarter, has some very interesting external interfaces: Stored procedures will be able to call virtually anything that can be contained in a DLL.

And since we're on the subject, did you know that roughly half of Centura Software's revenues continue to come from SQLBase? I'm sometimes chided by readers who use other back ends and don't like "wasting" article space on SQLBase. Based on the revenues, we're not devoting *enough* space to SQLBase. But I admit that the front-end programs generate more need for education and productivity improvements, and we'll continue to focus on those tools. However, it's your newsletter, too. Let us know what mix of topics you'd like to see. One of the major issues confronting us at Pro Publishing is how to divide coverage between SQLWindows and Centura. Fortunately, many articles apply to both tools. Centura is timely, but most of us are still using SQLWindows, and still need tips and support there.

Your vote for most esoteric session?

That's a hard choice for me. But one candidate was Jim Tierney's talk on designing tools and wizards for Centura Team Developer. One featured topic was the "Wizard Wizard," which turns ordinary executables and DLLs into wizards that are available at design time. To be fair, it's a useful and interesting idea, and Jim's sample code on Centura's CompuServe forum is worth a look. But, it gets me to thinking about "Wizard Wizard Wizards" and "Wizard Wizard Wizard Wizards"... The mind boggles.

People-watching

The conference at Palm Desert attracted a smaller crowd than last year. Whether that was due to the confusion that has surrounded Centura in this year of transition, or because it was the third year in a row in a climate where lunchtime temperatures exceed 100 degrees Fahrenheit, is open to question. Vik Chaudhary of Centura noted that news of a return to profitability came too late to have much impact on mindshare, but he'll be curious to see what months of continued profits will do to attendance at the European conference in Paris in October. And it's already been decided that next year's U.S. conference will be somewhere other than Palm Desert, although the exact location hasn't been decided.

Notes from the back

What can I say? Robert X. Cringely of *InfoWorld* already has a "Notes from the Front" column. Besides, these are notes from the back of the session rooms at the conference; some customers were kind enough to offer their opinions on the high and low spots. Well received: the COOL enhancements to SAL, the Internet QuickObjects, Ranger replication, the Application Server, the overall organization of the conference, accessibility and openness of Centura staff, the videos (!), and everything about the Java Virtual Machine. Disappointments included insufficient OLE support in the first version of Centura, lower turnout of vendors and customers, sessions dominated by Centura speakers rather than outsiders, and Master Classes that didn't really live up to their name.

What happened to *Gupta Pro*?

Although Pro Publishing retained the Gupta name in its newsletter for a few months after Gupta Corporation changed its name to Centura Software Corporation, we eventually negotiated the right to use Centura in the newsletter's name. And if you're as sick of hearing the word "formerly" as we are, you'll be pleased to know that we'll use it sparingly! **CP**

Mark Hunter, Editor of *Centura Pro* (and beloved among the staff), runs Hunter Software, Inc., a software development consulting firm in Southern California. Reach him at 71460.3142@compuserve.com or (818) 249-1364.

Table 1. Upcoming Centura releases.

Product	Timing	Notable for
SApplication Builder	June 1996	Interface to SAP R/3: hot!
Centura for Solaris	Q3 1996	No other UNIX versions planned for now
SQLHost 4.0	Q3 1996	Also improved CICS and RPC wizards
Centura Web Developer	Q3 1996	Centura apps with HTML output; requires 3-tier; JAWS server
Centura Application Server	Q3 1996	Easy deployment of Centura apps in the third (fourth, fifth...) tier
Centura Patriot	Q3 1996	Optional app deployment on Windows 3.1, Internet QuickObjects
SQLBase Voyager	Q4 1996	Internal enhancements, NetWare TCP/IP, better replication, multiple connections, stored procedure improvements
SQLWindows Cyclone	Q4 1996	"5.5" version; multiple error messages, dynalibs, compiler improvements, DB Explorer
Centura Tomahawk	Q1 1997	Java deployment; app partitioning; much better ActiveX support; dynamic instantiation; garbage collection; much more

Five Habits . . .

Continued from page 1

their way into most class library designs. This checklist, in fact, may also assist you in evaluating third-party class libraries.

1. Give your classes a facelift

One of the challenges that developers of base classes face is getting their users (that is, the other developers using those classes) to instantiate and implement those classes in a consistent manner. Furnishing exhaustive documentation on the classes is a logical remedy that, alas, isn't the most effective. During those crunch hours to meet deadlines, programmers are often likely to dismiss those documented details as aesthetic rather than pragmatic, and "just do it." Familiar? Consider the following child table window class, designed with the good intention of facilitating its population from the database:

```
♦ Child Table Class: cls_ctblAutoPopulate1
  ◊ Description: Base class for child table window
  ◊ Derived From
  ◊ Contents
  ◊ Class Variables
  ♦ Instance Variables
    ◊ String: sTableName
    ◊ Number: nPopulateMode
  ♦ Functions
    ♦ Function: Populate
      ◊ Description: Function for populating
            the child table window.
            Uses instance variables
            for table name and
            population technique.
            Uses global sql handle.
      ♦ Returns
        ◊ Boolean:
      ◊ Parameters
      ◊ Static Variables
      ◊ Local variables
      ♦ Actions
        ◊ Return SalTblPopulate
          ( hWndForm, ghSql_Select1,
          'SELECT * FROM ' || sTableName,
          nPopulateMode )
  ♦ Message Actions
    ♦ On SAM_Create
      ◊ ! get the values for instance variables
      ◊ Call SalSendMsg ( hWndForm,
        PM_InitVars, wParam, lParam )
      ◊ ! and populate now
      ◊ Call Populate ()
```

The *prescribed* way of putting this class to use would be:

```
♦ Child Table: tblGuests
  ◊ Contents
  ◊ Functions
  ◊ Window Variables
  ♦ Message Actions
    ♦ On PM_Initialize
      ◊ Set sTableName = 'GUEST'
      ◊ Set nPopulateMode = TBL_FillAll
```

Six months elapse. The class designer has gone looking for greener pastures. Speedy Gonzales has been hired to add new modules to the system. He has a quick look at the code for cls_ctblAutoPopulate1, and, swiftly on a Friday evening, codes up another child table window as:

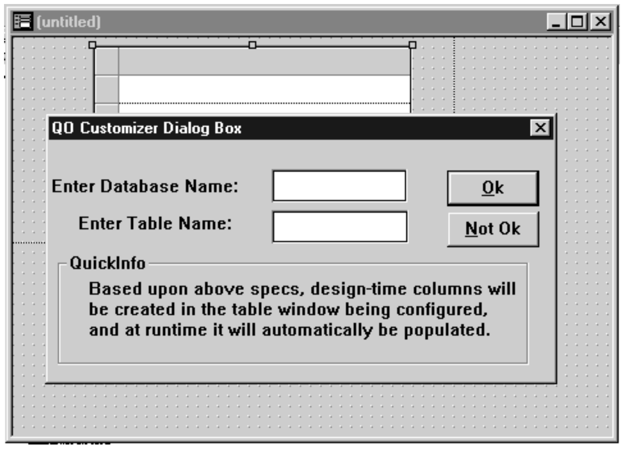
```
♦ Child Table: tblGuests
  ◊ Contents
  ◊ Functions
  ◊ Window Variables
  ♦ Message Actions
    ♦ On SAM_Create
      ◊ Set sTableName = 'GUEST'
      ◊ Set nPopulateMode = TBL_FillAll
      ◊ Call Populate ()
```

The new code is surely going to work just fine on that Friday evening and the days following. But clearly, the inconsistency in approach has induced a time bomb in the application, which is going to detonate as soon as the base class code in cls_ctblPopulate1 is modified.

This, in reality, is quite a severe problem that afflicts large SQLWindows projects. After all, how many lines of code can the QA police inspect to ensure coding consistency ? The best solution would be to get the base class code itself to enforce implementation uniformity by turning the class code into a *QuickObject*.

For example, once the class cls_ctblAutoPopulate1 is turned into a QuickObject, every time a programmer drops an instance of this class onto the design window, a wizard could pop up to capture the values corresponding to instance variables sTableName and nPopulateMode and automatically plug in the required lines of SAL code in the outline. The programmer won't need to touch SAL at all, eliminating the possibility of introducing inconsistency.

To build such QuickObjects of your own, which generate lines of code in the outline (or manipulate *named properties*), you'll need to use the Component Developers' Kit (CDK) from Centura Software. The benefits of this approach are twofold:



Based upon above specs, design-time columns will be created in the table window being configured, and at runtime it will automatically be populated.

Figure 1. The QuickObject interface simplifies development.

- You get an easy-to-use graphical interface atop the classes that makes them easier to be instantiated by programmers at design time.

- The QuickObject wizard generates the same, consistent SAL code every time it's used.

Have a look at the sample application HABIT1.APP. It contains a QuickObject class (ctblQO) that, when dropped on a design window, pops up a little dialog box to capture the database and table name information. The dialog box then goes on to generate design-time columns in the child table window and adds the necessary logic to populate it at runtime. How the QuickObject really works is beyond the scope of this article.

2. Remember: Unobtrusive and flexible

The development framework provided by the class library should be such that the programmers using it shouldn't feel constricted or the need to be overly cautious. The base classes need to intercept certain events during the life cycle of an object: creation, editing, validation, destruction, etc. SQLWindows provides SAM messages for each of these events–SAM_Create, SAM_AnyEdit, SAM_Validate, SAM_Destroy, and so on.

If the class library intercepts some of these message handlers for its own internal usage, the programmers will need to know the alternatives. For example, the class might use up SAM_Create and post a PM_Initialize message to the instance of the object. That's where the programmer is supposed to provide any additional initialization logic. But then, SAM_Create comes so naturally to any SAL programmer. Chances are, someone, somewhere will inadvertently code the SAM_Create message handler in the object instance (and be oblivious about SalSendClassMsg) and nullify all the good work being done in the base class. From a programmer's perspective, having to remember which SAM messages are or aren't available to be coded is a bit of a snarl.

Most SAM messages have an underlying native Windows message (WM_) counterpart: WM_CREATE for SAM_Create, WM_DESTROY for SAM_Destroy, and the like. To stay out of the programmers' way, the class library can intercept these WM_ messages for their internal processing, keeping the SAM messages free for use at the instance level.

In fact, if you look closely at the Windows SDK documentation, you'll find a bunch of other Windows messages that don't have a SAM counterpart, yet are fully available to be trapped by your class code. For example, a form window, as part of its creation, receives the following messages, among others, *before* getting the SAM_Create message: WM_CREATE, WM_NCCREATE, WM_NCCALCSIZE, WM_SETFONT. This offers ample opportunities for trapping the creation of the form window in the class code without having to break into SAM_Create.

For example, the class code might need to perform a runtime security check on the window to determine the access level, or to ensure ample Windows system resources (say, at least 20%) to clinch the window creation. There are, however, going to be some exceptions when there's no real alternative to trapping the SAM messages in the class code–notably the table window. The table window really is a custom control invented by Centura Software. Not being a standard window control, it's bereft of standard windows messages. It doesn't, for instance, get a WM_CREATE message.

Once the programmers have a good understanding of the division of labor of messages, they look for ways and means of enhancing the functionality provided by the class code. To make their lives easier, SQLWindows provides this wonderful OOP feature: late-bound function calls. Despite the hype associated with the sluggishness of late-binding, they normally work *faster* than messages (see the February 1996 issue of *Gupta Pro* for a fuller treatment of the topic).

If you examine the class libraries developed by the Centura Software engineers (for example, QuickObjects and QuickTabs), you'll notice late-bound functions being put to good use, easily allowing extensibility of the class framework. A very good example of that is the QuickObject DVC (Data-source, Visualizer, Commander) framework. It doesn't do QBE for you, but it does provide an easy hook into the class code: The data source (cQuickTable) provides a late-bound function called *GetSelectWhere,* which allows you to specify your own WHERE clause for the SQL being used by the data-source for population. Such kinds of functions provide developers easy hooks into the class code. The trick, of course, is to anticipate ahead, whilst designing the class library.

3. Build frameworks

"Framework n. 1. A skeletal structure for supporting, shaping, or enclosing something; a frame. 2. A basic arrangement, form, system, or set of relationships."–American Heritage Dictionary, *Houghton Mifflin Company, 1981.*

What a framework means depends on what you do for a living. For our purposes, a framework is an environment that facilitates the development of a family of applications (or components) with similar underlying behavior or mechanics. The DVC QuickObjects is the best example of a framework. It consists of a class library, but it isn't just a class library. Visual Toolchest is a class library, but it isn't a framework. The difference is that in a framework, all the components of the constituent class library are interrelated and work together as a team to deliver predefined functionality. Various classes within the Visual

Toolchest, on the other hand, can work quite independently of each other. You can use the drop-down calendar control, for example, without having to worry at all about the Explorer-style listbox control.

One good thing about frameworks is that they provide you a set of well-defined services that alleviate the need for doing menial chores. Some examples of services are error handling, transaction management, and events notification. Even though building a framework requires intense analysis and design, it pays itself back quite quickly by boosting productivity. Frameworks provide a reliable, predictable development environment for creating a set of applications with a consistent look and feel.

When building a large system, it's easy to identify candidates for frameworks: Does your system use MDI windows ? Then probably you're looking for a framework that provides you notifications every time a child window is being created or destroyed, or when the user changes focus to a different child window—or a framework for reading and writing from the database, perhaps.

Make sure, when designing a framework, that it's flexible enough to allow the programmer an easy interface for changing default behavior or even adding new functionality. For example, in the DVC framework, you could subclass the cqoDataSource functional class to create a new datasource that knows how to communicate with SAP R/3 data (what an idea!) and also provides a QBE interface. All this, without having to worry about the data commanders and visualizers—framework's default functionality takes care of that!

4. Don't worry; let the hSql jukebox handle it

How many SQL handles does the application need ? How long does it take to make a connection to the database ? When is a good time to relinquish the connection ? How can you tell whether or not the global SQL handle *ghSql2* is available to be re-used right now ? What should be the lock-time-out setting for this cursor ? Fret not; give the class library a chance to handle all these and other SQL handle-related questions for you. The class can worry about:

- Determining the number of SQL handles that should be connected to the database as part of the application startup.

- The minimum number of handles that should be connected to the database at any given point of time in the runtime life-cycle of the application.

- Maintaining an array of "available" and "checked-out" SQL handles. This safeguards against the infamous "no compiled command" error in table window population, by making sure that another part of the application can re-use the cursor presently in use by the table window.

- Ensuring optimal cursor re-use.

- Setting default properties for the cursors, like: lock-time-out value; cursor context preservation; shared/distributed transaction mode; isolation level; result set mode, and the rest of them.

Cursor management becomes all the more important against non-SQLBase back-ends (like Oracle or Sybase), where each cursor connection can be an expensive operation–in terms of performance, licensing implications, and network resources (for example, the number of TCP/IP sockets required for supporting all those connections), and server resources (the amount of memory required on the server-side for each connection).

5. Approach error handling with structure

Errors happen. A significant portion of the application is often dedicated to handling those runtime errors– frequently stemming from database interaction. A class library should be accompanied by a global error handler that, at the least, gets rid of SQLWindows' default error message dialog box. An *effective* class library, on the other hand, will take care of the following essentials:

- Logging the errors to a file, for the benefit of help desk personnel. In fact, if you want to get fancy, you may even consider triggering an email message or sending out a beeper message.

- *Never, ever* show the user the error text found in ERROR.SQL file. The class library should have access to a database of translated, user-friendly error messages that are presented to the user when things go wrong. Avoid hard-coding the message text in the application–have them in flat files or a database, to facilitate their maintenance without the need for recompiling the application.

Database errors aren't the only kind that can disrupt the functioning of an application. What about the infamous Windows free system resources (FSR) problem, which is particularly acute under Win16? It's a widespread practice that, when creating a top level window within the application, the code doesn't check for the return value, for example, a call to SalModalDialog returns -1 if the dialog box couldn't be created.

Instead of coding "Call SalCreateWindow (…)", it's probably a good idea to code "If *not* SalCreateWindow (…)" to ensure the sanctity of subsequent code. The class library should be proactive enough to handle such

Continues on page 15

Solve DLL and Memory Problems *Dynamically*

Ram Ramakrishnan

SQLWindows applications load *all* the Dynamic Link Libraries (DLLs) declared in the External Functions section of the application at application startup. To optimally manage memory, a DLL should be loaded only when the application invokes one of the DLL's functions. Irrespective of whether or not the functions in these DLLs are invoked during the session, loading all DLLs at application startup doesn't lend to efficient memory management. Often, SQLWindows applications using a large number of DLLs may not even execute due to insufficient memory. In this article I explain how to overcome this limitation of SQLWindows.

Memory management challenges most client/server projects. Often, SQLWindows applications that use a large number of DLLs refuse to load or execute properly due to a lack of memory. Dynamically loading and unloading DLLs provides a solution to this problem.

Ways of loading DLLs

You can load DLLs in one of three ways: implicitly through the make file, explicitly through the module definition (DEF) file, or dynamically using SDK functions that support dynamic loading. Through dynamic loading of DLLs, you can control the use of memory. DLLs loaded dynamically can also be unloaded when desired, freeing up memory. Through this "dynamic" loading and unloading, you can manage DLLs and memory from your SQLWindows applications.

An example

As an example, let's look at a document imaging SQLWindows application. This application usually provides different functions such as scan, fax, print, image manipulation, full text retrieval, and optical character recognition. Most of these functions are usually coded in separate DLLs. Often, developers license third-party DLLs and offer the functionality through their application.

The nature of the application could be such that not all of the DLLs will be needed at the same time. For example, at any given moment, the user is either going to fax an image or scan an image, but not fax and scan simultaneously. Some users may never fax images. Nevertheless, all these DLLs end up getting loaded into memory. Further, when third-party DLLs are used, sometimes you end up loading large DLLs into memory for the entire duration of the application, even though you're interested in only using a couple of functions from that DLL.

The shell game

As a first step towards dynamically loading DLLs, you must build a "shell" DLL. The shell DLL will alias only the functions in the "original" DLLs that your application intends to use. For example, let's say your original DLLs are FAX.DLL, SCAN.DLL, and IMGCTRL.DLL. You're interested in two functions from each DLL—FaxAPage and BuildCoverPage from FAX.DLL, BeginScan and EndScan from SCAN.DLL, and ZoomImage and RotateImage from IMGCTRL.DLL.

These functions are aliased as ShellFaxAPage, ShellBuildCoverPage, and the like, in the shell DLL. These alias functions will have the same number of parameters with the same datatypes as the parameters of the original functions. The alias functions' return type will also be the same as the original function. In this example, since you're interested in only six functions, the shell DLL will contain just those six functions and will replace the three DLLs. If you use all the functions in the original DLL, then you'll have to alias all of them in the shell DLL.

The SQLWindows application accesses the functions in the original DLL through the alias functions in the shell DLL. At application startup, only the shell DLL will be loaded (see **Listing 1** on the next page).

Using the shell DLL

When the SQLWindows application wants to execute the FaxAPage( ) function, it calls the ShellFaxAPage( ) function. This, in turn, loads the appropriate original

DLL. As soon as the function executes, the original DLL is unloaded. Similarly, other functions aliased in the shell DLL load and unload the appropriate original DLLs. Thus, memory is effectively used.

The sample SQLWindows code in **Listing 2** demonstrates how the shell DLL is invoked.

Listing 1. Sample C code for a shell DLL

```
#include <windows.h>

int FAR PASCAL LibMain(HANDLE, WORD, WORD, LPSTR);
int FAR PASCAL WEP(int);

/* Define the alias functions */
/* The parameters are the same as the original function */
/* The return type is the same as the original function */

HANDLE FAR PASCAL ShellFaxAPage(LPSTR,int,BOOL);


int FAR PASCAL LibMain(HANDLE hInst, WORD wDataSeg, WORD wHeapSize, LPSTR
lpszCmdLine)
{
  if(wHeapSize > 0)
    UnlockData(0);
  return(1);
}


int FAR PASCAL WEP(int nParam)
{
  return(1);
}

/************************************************************
 *  ShellFaxAPage aliases the FaxAPage function in FAX.DLL.*
 *  The original function is declared as:                 *
 *         HANDLE FaxAPage(LPSTR,int,BOOL)                 *
 ***********************************************************/

HANDLE FAR PASCAL ShellFaxAPage(LPSTR file, int type, BOOL bdialog)
{
  HINSTANCE hInstance;
  /* typecast the function pointer */
  HANDLE (FAR PASCAL *lpfnFaxAPage) (LPSTR, int, BOOL);
  HANDLE hResult;

  /* load the library */
  hInstance = LoadLibrary( "FAX.DLL" );

  if (hInstance > HINSTANCE_ERROR)  {
    /* get the function address from the library */
    (FARPROC) lpfnFaxAPage = GetProcAddress( hInstance, "FaxAPage");
    if (lpfnFaxAPage != NULL)  {
      /* execute function */
      hResult = (*lpfnFaxAPage) ( file, type, bdialog);
      /* unload the library  */
      FreeLibrary( hInstance );
      return (hResult);
    }
    else  {
      /* function failed.. unload library  */
      FreeLibrary( hInstance );
      /* add error handling here  */
      return (HANDLE)0;
    }
  }
  else  {
    /* failed in loading DLL */
    /* add error handling here */
    return (HANDLE)0;
  }
}

/* define rest of the functions here */
```

Pros and cons

You can build a shell DLL that's small and replace a number of other larger DLLs. Since the SQLWindows application loads only the small shell DLL at startup, you'll find that your applications load faster—an advantage of this approach.

Dynamically loading and unloading DLLs add a small performance penalty associated with reading the DLL from disk and loading to memory. Therefore, you ought to be careful in selecting the DLLs to be dynamically loaded.

If you have a small DLL whose functions are frequently called by the application, it may not be a good candidate for dynamic loading. Consider this technique for large DLLs whose functions are infrequently called. (My users haven't indicated any noticeable performance degradation.) I've used this technique with memory-bound SQLWindows applications to manage memory and DLLs effectively. **CP**

Ram Ramakrishnan is a Technical Director of USI at Chantilly, Virginia. He specializes in the design and development of enterprise-wide client/server systems. He can be reached through email at 72607.3443@compuserve.com.

Listing 2. Code for invoking the shell DLL.

```
♦Library name: SHELL.DLL
  ♦Function: ShellFaxAPage
    ◊ Description:
    ◊ Export Ordinal: 2
    ♦ Returns
      ◊ Window Handle: HWND
    ♦ Parameters
      ◊ String: LPSTR
      ◊ Number: INT
      ◊ Boolean: BOOL
  ♦Function: ShellBuildCoverPage
  ♦Function: ShellZoomImage
  ♦ ! Declare other aliases here..

♦Form Window: frmMain
  ♦ Pushbutton: pbFax
    ♦ Message Actions
      ♦ On SAM_Click
        ◊ Set strFile = dfFaxImage
        ◊ Set hWndFaxQueue =
          ShellFaxAPage
          ( strFile, 0, FALSE )
```

Create Your Own Application-Level Messages

R.J. David Burke

The Application Actions section of Global Declarations is used to define message handlers for application level messages. SQLWindows provides the following application level messages:

- SAM_AppStartup, sent when an application begins execution, before any windows are created.
- SAM_AppExit, sent when an application is terminated, after all windows are destroyed.
- SAM_SqlError, sent when a SQL error occurs.

Have you ever wanted to implement your own application level message? SQLWindows provides the ability to define messages (as numeric constants) and the ability to process messages (On <message>). But how do you transmit messages to the Application Actions section? SalSendMsg( ) and SalPostMsg( ) both require a window handle as the recipient of the message. What is the window handle of the application?

Windows API to the rescue! By calling the function PostAppMessage( ), a message can be placed in the message queue of an application, by specifying the task handle of the application. The Windows API function GetCurrentTask( ) can be used to get the task handle of the current application.

The application code in **Listing 1** shows how to add this functionality to your applications.

The Windows SDK documentation lists GetCurrentTask as returning a task handle and the first parameter of PostAppMessage has a task handle, type. Since SQLWindows doesn't provide task handle data types, I've used window handle data types. In 16-bit Windows environments, window handles and task handles are both implemented as 16-bit unsigned words. By taking advantage of this compatibility, I can provide some protection to task handles (prevent using them in arithmetic expressions). The alternative would be to define these items using a Number: WORD external data type. **CP**

This tip provides a powerful tool for managing communications not only within an application, but among multiple Windows applications.

For the sample application, download APPACT.APP (GO CENTURA, Library 10).

R. J. David Burke is a Senior Consultant with Centura Software Corporation. 102336.171@compuserve.com.

Listing 1. A method for establishing the window handle of an app.

```
♦ Global Declarations
  ♦ Window Defaults
  ♦ Formats
  ♦ External Functions
    ♦ Library name: KERNEL.EXE
      ♦ Function: GetCurrentTask
        ◊ Description:
        ◊ Export Ordinal: 0
        ♦ Returns
          ◊ Window Handle: HWND
        ◊ Parameters
    ♦ Library name: USER.EXE
      ♦ Function: PostAppMessage
        ◊ Description:
        ◊ Export Ordinal: 0
        ♦ Returns
          ◊ Boolean: BOOL
        ♦ Parameters
          ◊ Window Handle: HWND
          ◊ Number: WORD
          ◊ Number: WORD
          ◊ Number: DWORD
  ♦ Constants
    ◊ System
    ♦ User
      ◊ Number: PM_AppLevelTest = SAM_User + 0
  .
  .
  .
  ♦ Application Actions
    ♦ On PM_AppLevelTest
      ◊ Call SalMessageBox( 'Processing App-Level Message',
             'App-Level', MB_Ok )
♦ Form Window: frm1
  ◊ Description:
  ◊ Named Menus
  ◊ Menu
  ♦ Tool Bar
  ♦ Contents
    ♦ Pushbutton: pb1
      ♦ Message Actions
        ♦ On SAM_Click
          ◊ Call PostAppMessage( GetCurrentTask(  ),
              PM_AppLevelTest, 0, 0 )
  ◊ Functions
  ◊ Window Parameters
  ◊ Window Variables
  ◊ Message Actions
```

Everything You Ever Wanted to Know About Dynalibs But Were Afraid to Ask

Doug Mitchell

This article explores Centura's new and somewhat enigmatic dynamically linked libraries (dynalibs). With Centura Team Developer's capability to generate *three* different types of libraries, you have the somewhat confusing task of determining when to use which library type. This article provides an in-depth look at dynalibs and compares the three library types to help you take maximum advantage of this new feature when scaling up your applications.

Before jumping headfirst into dynalibs, let's review the features, strengths, and limitations of the two other available library types: Application Libraries (APLs) and Dynamic Linked Libraries (DLLs).

Background: APLs

The oldest available library type is the APL, which allows you to decompose your application into numerous code modules. APLs provide you with two distinct features: code reuse and parallel development. First, generic, reusable code can be centralized into sharable libraries, which allows multiple applications to take advantage of the pre-built code. Second, APLs allow large applications to be decomposed by business function so that multiple developers can work simultaneously on the same project. Both features are critical for large-scale application development.

One of the APL's greatest strengths is its flexibility. For example, almost any section in the application outline can be contained in an APL. As a result of this flexibility, an APL allows for module design and application decomposition. Another important aspect is that an APL is included at design-time. While this aspect offers powerful debugging capability for you to step through the code at design-time, it does present one significant drawback. Because the APL's code is included at design-time, it needs to be compiled along with the application including it. For an application making extensive use of pre-built APLs, this drawback can result in a significant productivity hit.

Background: DLLs

While DLLs have been around since Microsoft Windows was first introduced, SQLWindows/Centura Team Developer's ability to create them has not. The SAL-to-C compiler makes it possible, with help from a third-party C compiler, to convert an APL into a DLL. This capability has obvious advantages over APLs: speed, speed, and more speed. The created DLL is a true DLL and, as such, executes with blinding speed. Another advantage is that the DLL is loaded at runtime instead of design-time and, therefore, does not have to compiled over and over.

Of course, debugging DLLs is another story. On the downside, only a very limited number of outline sections can be compiled. This limitation has a significant impact on its flexibility and, therefore, its utility. The most recent release of Centura Team Developer has increased the number of outline sections that can be compiled, although it's still a long way off from compiling the entire application. Even within the allowed sections, there are significant restrictions on the code for it to be compiled to a DLL.

Dynalibs: What are they anyway?

Dynalibs, or APD for short, are specially *compiled* Team Developer APLs that other Team Developer applications can dynamically load at *runtime*. Unfortunately, this "compiled" state doesn't result in any performance gains relative to ordinary APLs because the code still remains interpreted by the Team Developer runtime engine. Another way to think of a dynalib is as a cross between a traditional application library and a Dynamic Link Library (DLL). However, this description is somewhat simplistic because dynalibs are neither as flexible as APLs, nor as fast in execution speed as DLLs.

How do I create a dynalib?

A dynalib begins with a standard Team Developer APL as its source. The first step to converting an APL into a dynalib is to identify exactly what is to be exported, or exposed for importation, into a Team Developer application. Unfortunately, dynalibs weren't well integrated into the new Team Developer user interface. Consequently, the export identification takes the form of a comment with an embedded pragma (compiler directive), for example:

```
Function: fnMyFunction ! __Exported
```

Without this pragma, the dynalib objects won't be exported and, thus, can't be referenced by another application, including the dynalib. Once the desired dynalib objects have been identified, the next step is to compile the source APL file into a dynalib file using the Team Developer's *Project/Build* menu option.

How do I use them?

A dynalib, like an APL, can be included in another APL or APP file via the library section in the application's outline. Also like an APL, the included objects appear in the host's outline and can be referenced like any other includable object. There are two fundamental differences, however. First, the code is dynamically loaded at runtime, instead of at design time. Second, the included object's code (in other words, its action section) is *not* included in the host's outline.

Dynalib's public interface

Instead, on the entire object's code, only its public interface or stub is visible in the host's outline. For a function, its action section won't be visible in the host's outline. For a top-level window, you can't even view its layout! Examples of the imported items might look like this:

```
Variables:
  Dynalink String: strExported    ! __exported

Internal Functions:
  Dynalink Function: fnFun ! __exported
    Description:
    Returns
      Boolean
    Parameters
      String: strParm1
      Receive Number: numParm2

Dynalink Form Window: frmMyForm ! __exported
  Description:
  Functions
    Function: fnMyWinFun
    ...
  Window Parameters
    String: strWinParm1
```

Important to note is that the public interface of top-level windows *doesn't* include inherited functions and variables. That is, inherited functions and variables aren't visible or directly accessible by an application importing the dynalib window. Also note that the child controls aren't imported, so you can't directly manipulate them. Although there are several workarounds for these limitations, all of them require some extra effort.

Dynalib limitation workarounds

For inherited functions that need to be invoked by the importing application, you can declare "wrapper" or pass-through functions in the dynalib window object; these, in turn, invoke the appropriate inherited function. Another alternative is to define a message-based, rather than function-based, public interface at the class level. While not always practical, a message-based interface can always be accessed without requiring additional code at the object level.

Inherited variables and child controls are less of a problem because, as a rule, they shouldn't be externally referenced without an accessor function. An accessor function at the object level can provide an interface to manipulate either the variable or child control. However, these accessor functions are often defined within the classes. In this case, redundant pass-through functions are required at the object level to provide access to an imported object's variables or child controls.

How do I edit a dynalib?

Typically, the compiled dynalib has the same name as the source file, except with an APD extension. It's good practice to maintain the same file name when you build the dynalib. By keeping the same name, you can launch from the host application a Team Developer session to edit the source file of an included dynalib using the *Component/Libraries/Go To Item* menu item. It's imperative that the name of the two files remain the same because there is no explicit link between the source file and the

Centura Tip!

Automating the Dynalib Build Process

The build process can be automated in a batch file by using the "-y" or "-m" Team Developer command line parameters:

```
CBI10.EXE -y DYNLIB.APL.
```

generated APD file. Therefore, Team Developer has to "guess" at the source file's name. It first checks for the APD's file name with an APL extension. If the file name with an APL extension isn't found, it then checks for the APD's file name with an APP extension.

Code restrictions

Code is significantly less restrictive with regard to dynalibs than that compiled to a DLL using Team Developer. In fact, the only real code restriction is that semi-qualified references (for example, hWndForm.dfName) aren't permitted. Given the nature of dynalibs, this limitation is understandable; without global knowledge at compilation time, the compiler can't enforce the consistency of this late-bound reference in any other way except to reject the reference. This restriction is easy to avoid by using fully qualified references instead.

Nested dynalibs

Another important usage feature is that dynalibs can be nested as simply as APLs. For example, an application can import objects from two dynalibs, A and B. Dynalib A can also import objects from Dynalib B. In doing so, the application is both directly and indirectly using Dynalib B. See **Figure 1**.

In the event of this multiple reference, only one instance of the dynalib is loaded into memory per application. This feature provides additional scalability by allowing you to build library hierarchies that include smaller generic dynalib components.

Comparing the features of the various library types

The key to maximizing the use of Team Developer's libraries is understanding their strengths and weaknesses. **Table 1** summarizes the features of the three libraries that Team Developer can generate.

Initially, it looks like dynalibs have all of the features of APLs and DLLs. However, as I said before, a dynalib is neither as flexible as an APL, nor as quick as a DLL. Further examination of these features will provide a better understanding of the exact advantages and disadvantages of the dynalib.

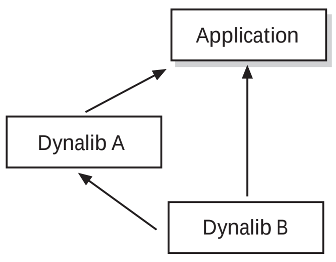


Figure 1. You can nest dynalibs as simply as APLs.

Table 1. Features by Team Developer library type.

Library Feature	APL	DLL	APD
Ease of Use	√+	√-	√
Include Flexibility	√+	√-	√
Reuse	√+	√-	√
Modular Design	√		√
Application Decomposition	√		√-
Incremental Upgrades		√	√
Reduced Deployment Footprint		√	√
Runtime Load upon Demand		√	√
Compilation Speed	√-	√+	√+
Source Code Security		√	√+
Execution Speed	√-	√+	√-

Compilation speed

A dynalib's ability to load at runtime has some obvious advantages and some more subtle disadvantages. My favorite advantage is the ability to avoid compiling the code every time you compile your application. In many of my projects, I've made use of generic and application-specific function APLs that rarely change, but are always included. When I compile my application, these static APLs always require recompilation. Dynalibs offer the perfect solution to this problem because they avoid unnecessary and redundant compilation.

Source code security

Another somewhat esoteric advantage of dynalibs is their ability to hide source code. This allows third-party companies to provide developer add-ons without having to distribute the source code. This source code security feature might make building developer add-ons by third-party companies more attractive, since these companies will be able to keep their proprietary code confidential.

Incremental upgrades

Dynalibs also provide the capability of performing incremental upgrades. That is, an application in production can have a dynalib patch applied without having to recompile and redeploy the entire application. This feature can be useful with widely-distributed applications; however, care must be exercised to avoid introducing unintentional side effects.

Reduced deployment footprint

Another dynalib deployment feature is the reduced disk space requirements (footprint) when deploying multiple applications using the same dynalibs. Since dynalibs are loaded at runtime, the same dynalib file can serve multiple applications. While an APL allows for the source code to be centrally located, its code is ultimately replicated in each Team Developer executable including the APL. A dynalib, like a DLL, not only centralizes the code at design-time, but also centralizes it at runtime.

While the runtime load feature might save you disk space during a deployment, don't assume it will save you memory. For every application that imports an object from a dynalib, a separate instance of the dynalib is loaded into memory. However, this is consistent with how DLLs work in a 32-bit operating system such as Windows 95 and NT.

Runtime load upon demand

An important scaling up feature for large-scale application development is the ability dynalibs have to load on demand at runtime, rather than at startup. Like a DLL, a dynalib will only be loaded into memory when an imported object is referenced. This feature can significantly shorten the startup time of large applications.

Dynalib flexibility

Some of the disadvantages dynalibs have relate to their current limitations. Like APLs, dynalibs promote reuse, modular design, and application decomposition. Unfortunately, not every object allowed in an APL can be imported from a dynalib. As a result, dynalibs are less flexible than APLs. Thus, a combination of dynalibs and APLs are required to maximize Team Developer's reuse and decomposition capability. The objects that can be imported from a dynalib are internal functions, primitive global variables (no user-defined variables), and top-level windows. **Table 2** compares the outline sections each library type can include.

While a dynalib is limited in what objects it can export, it isn't limited to what those objects can reference. Therefore, a form window can be derived from a user-defined class, reference constants, and call external functions. This is the critical aspect of dynalibs that really makes them a powerful and viable library type.

Table 2. Included sections by Team Developer library type.

Outline section	APL	DLL	APD
Global functions	Yes	Yes, but with significant restrictions	Yes
Classes	Yes	General Window Classes only	
Top-level windows	Yes		Yes
Global variables	Yes		
Primitives only			
Named menus	Yes		
Constants	Yes	Ext. function declarations	Yes
Formats			

Dynalib dangers

Pre-compiled libraries, such as dynalibs, open up the Centura developer to a whole new set of potential pitfalls. Since a dynalib can make use of classes, it's interesting to note that even if a dynalib requires an APL for its compilation, an application importing that dynalib doesn't require the same APL. An example will help clarify this point.

Suppose I create a dynalib called BROWSER.APD containing a form called "frmBrowser," created using a form class called "cls_frmBrowser." This class is contained in an APL called CLS_BROW.APL. The library, CLS_BROW.APL, is required to create the BROWSER.APD, but our application importing BROWSER.APD *doesn't* need to include CLS_BROW.APL in order to compile. For this to be possible, the class definitions in CLS_BROW.APL must be included in BROWSER.APD, but not exported or exposed.

Now, if the application needed CLS_BROW.APL for a different form, the application would have two definitions for the same class loaded redundantly in memory at runtime. Even worse, if cls_frmBrowser were modified *after* the dynalib was created, then the application at runtime would have two definitions for the same class, but the definitions would be different!

To prevent this from occurring, it would be necessary to rebuild all included dynalibs every time an application executable was generated–an unfortunate, but necessary step to prevent a dynalib from using a potentially obsolete version of the code.

A bit of good news is that class variables behave properly in dynalibs, so a dynalib's top-level window can share class data with a non-dynalib top-level window. This is somewhat amazing (scary?) considering the fact that the class definitions can be entirely different, as portrayed in the example.

"Select From" Library Feature Decommissioned

With the introduction of dynalibs, a third type of library, Centura has decommissioned the little-used "Select From" application library feature that allows you in SQLWindows to include a specific section/item from a library, rather than the all-or-nothing approach with a "File Include." Centura cited this feature's lack of use and the addition of dynalibs as driving reasons for the removal of this "Select From" feature.–*D.M.*

Large-scale application development using dynalibs

With careful planning, dynalibs can provide the capability for application decomposition required by large-scale development. By decomposing your application into distinct business functions and storing the related screens in dynalibs, an entire application's business function layer can conceivably be composed of dynalib components (see **Figure 2**).

With dynalibs' current limitations, it's more likely that a large-scale application will need to leverage all the Team Developer's library types, as shown in **Figure 3**.

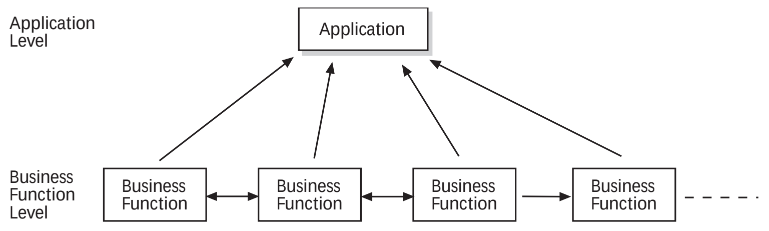


Figure 2. Application decomposition using dynalibs.

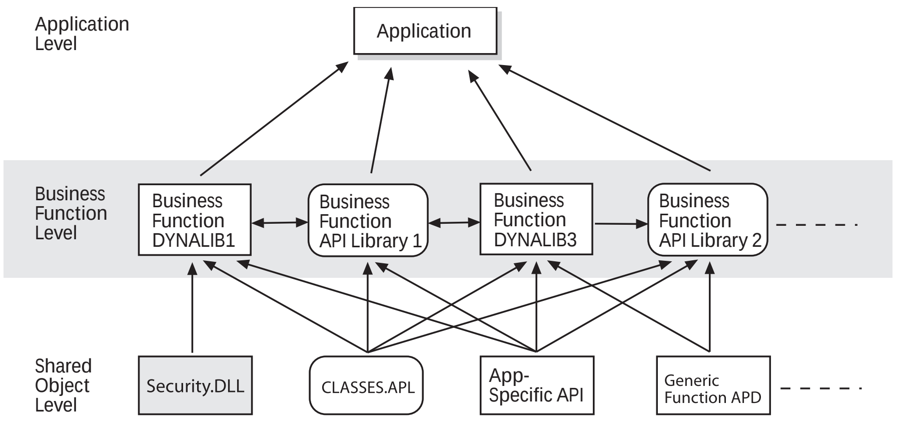


Figure 3. Application decomposition by business function and shared objects.

Final notes

Dynalibs are no panacea; however, when properly used in conjunction with the Team Developer's other library types, dynalibs can be a real boon to developers. In writing this article I found myself wondering about the future role of dynalibs. In the process of achieving Centura's goal of executing a Team Developer application using a Java runtime engine, dynalibs could easily become obsolete. At the recent Centura conference, I took the opportunity to ask this very question. Centura's Don Madison, who is involved in the development of dynalibs, stated that dynalibs will have an increasingly important role in the future because Centura is planning to enhance them to allow more outline sections, such as classes, to be exported. This expanded capability will make dynalibs even more useful when scaling up applications. **CP**

Doug Mitchell, a Principal with American Management Systems (AMS) in Fairfax, Virginia, has been using SQLWindows/Centura to help his clients develop custom client/server business applications for the last five years. He has served as a SQLWindows instructor, presented at the last four Gupta/Centura conferences, and currently presides over the Washington, D.C.-area Centura user group. Internet doug_mitchell@mail.amsinc.com.

Five Habits . . .

Continued from page 7

situations. Windows 3.x is known to become quite unstable once FSR drops below 20%. A good way to avoid this problem would be to check FSR *before* creating each top level window. For example, the following code, in a top level window's base class (or in a general window class, for that matter), will take care of the problem:

```
♦ On WM_CREATE
  ◊ ! GetFreeSystemResources is a function
    ! in USER.EXE
  ♦ If GetFreeSystemResources
    ( GFSR_GDIRESOURCES ) < 20
    ◊ Call SalDestroyWindow ( hWndItem )
    ◊ Return -1
```

A work in progress

I hope the ideas I've explored in this article will lead you to other areas where you can introduce efficiency and effectiveness into your class libraries. There are certainly many more to discover. Keep these class library design habits in mind as you develop habits of your own. **CP**

Download VINAYAK2.ZIP from library 10 of the Centura CompuServe forum.

Tony Vinayak is a Senior Consultant with Centura Software Professional Services. (201) 644-4000, 75144.2043@compuserve.com

OLE, the *Really* Ugly Way

Per-Arne Liljedahl

The standard way to access OLE automation in SQLWindows is to build functional classes using the QuickOLE wizard. There are some holes in this feature, and you may want to know if there's another way to use OLE in SQLWindows.

Perhaps you recall "OLE: the Good, the Bad, and the Ugly" in the May issue of *Centura Pro*. After the article was published, this tip from Per-Arne Liljedahl showed just how ugly OLE can really be. Compared to this, QuickOLE doesn't look so bad . . .

Yes, there is, but it's quite tricky. The OLE wizard uses the typelib to get information about parameters, GUID, function type, and the like. A tool called OLE2VIEW can give you this information even without a typelib. It comes with the OLE Toolkit (and possibly other Microsoft products).

If you use OLE2VIEW and get this information about an object:

```
VOID Delete(
  IN String Name
  )
memid=0x60030001
guid={00025E50-0000-0000-C000-000000000046}
lcid=1033
```

you should end up with the declaration in SQLWindows shown in **Listing 1**.

It can be quite tricky to determine the datatypes in some cases when it's declared as variant. It's usually possible if you have good documentation on the program (and a lot of patience).

The ordinal number of the external function is determined by the function's type and whether it takes parameters or not. Here's a list:

2	STRINGMETHOD
3	STRINGMETHODNOPARMS
4	INTEGERMETHOD
5	INTEGERMETHODNOPARMS
6	LONGMETHOD
7	LONGMETHODNOPARMS
8	FLOATMETHOD
9	FLOATMETHODNOPARMS
10	DOUBLEMETHOD
11	DOUBLEMETHODNOPARMS
12	CURRENCYMETHOD
13	CURRENCYMETHODNOPARMS

Void methods seems to use ordinals 2 and 3. **CD**

Per-Arne Liljedahl is a Swedish member of Centura's TeamAssist, a group of talented developers who volunteer their time to help others through the Centura forum on CompuServe. His interests include OLE, Windows sockets, and other frustrating interface standards. PAL@collega.se.

Listing 1. SWOLE20.DLL declarations.

```
♦Library name: SWOLE20.DLL
  ♦Function: XDelete
    ◊Description:
    ◊Export Ordinal: 2
    ◊Returns
    ♦Parameters
     ◊String: HSTRING


♦Functional Class: cFoobar
  ◊Description:
  ◊Derived From
  ♦Class Variables
    ◊String: strDispatchPath
    ◊Number: numLocaleID
    ◊String: strClassID
  ♦Instance Variables
    ◊Number: numObject
  ♦Functions
    ♦Function: Initialize
     ◊Description:
     ◊Returns
     ♦Parameters
      ◊Number: numInitObject
     ◊Static Variables
     ◊Local variables
     ♦Actions
      ◊Set strClassID = "{00025E50-0000-0000-C000-000000000046}"
      ◊Set numLocaleID = 1033
      ◊Set numObject = SWinInitObject(numInitObject, numLocaleID,
             strClassID, strDispatchPath)
    ♦Function: Delete
     ◊Description:
     ◊Returns
     ♦Parameters
      ◊String: Name
     ◊Static Variables
     ◊Local variables
     ♦Actions
      ◊Call XDelete(numObject, 0x60030001, numLocaleID, "F" ||
       VT_VOID || VT_BSTR, Name)
    ♦Function: ReleaseObject
     ◊Description:
     ◊Returns
     ◊Parameters
     ◊Static Variables
     ◊Local variables
     ♦Actions
      ◊Call SWinReleaseObject(numObject)
```