Special International Section, Follows p. 40

70 COLOR MONITORS REVIEWED

BYTE

7.50

JANUARY 199

BYTE Awards: The Best Products of 1993

Chicago: Windows 4.0 Enters Beta Testing PAGE 18

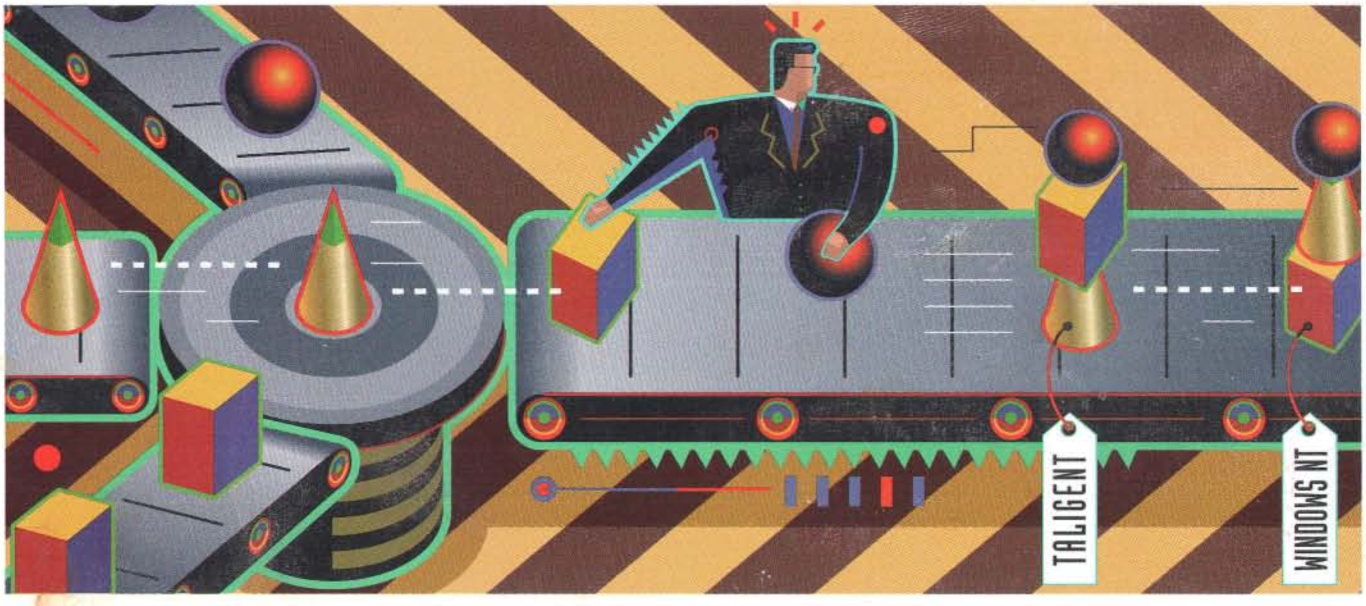
Apple's Mac Does Windows

THE WORLDWIDE COMPUTING AUTHORITY

SPECIAL REPORT

ADVANCED OPERATING SYSTEMS

A look inside the next generation from IBM, Apple, Microsoft, Novell/USL, Sun, Next, and Taligent



PLUS

- New Microprocessors Challenge Intel PAGE 74
- 4 Cross-Platform Toolkits Reviewed

International Edition

01

9 770360 528001

£1.95 U.K./Lit.8,000 ITALY/ASH 94
$3.50 U.S.A./A McGraw-Hill Publication/0360-5280

BYTE
JANUARY 1994 VOL.19, NO.1

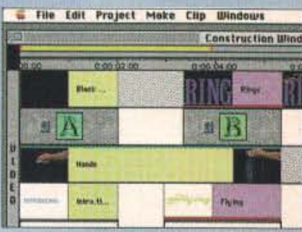
News & Views

Special Report

ADVANCED OPERATING SYSTEMS

Feature

Reviews


PAGE 179

PAGE 183

PAGE 197

State of the Art

Hands On

Opinions

PROGRAM LISTINGS

From BIX: Join "listings/frombyte94" and select the appropriate subarea (i.e., "jan94").

From the UUNET:ftp to ftp.uu.net, log on as "anonymous," and enter your user ID as your password. Type "cd/published/byte" and type "DIR." Files appear in subdirectories arranged by month.

From the BYTE BBS at 1200-9600 bps: Dial (603) 924-9820 and follow the instructions at the prompt.

BYTE (ISSN 0360-5280) is published monthly by McGraw-Hill, Inc. U.S. subscriber rate $29.95 per year. In Canada and Mexico, $34.95 per year. European surface mail subscriptions $60, airmail $80. Non-European subscriptions, $60 surface mail or $85 airmail. All foreign subscriptions are payable in U.S. funds that can be drawn on a U.S. bank. Single copies $3.50 in the U.S., $4.50 in Canada. Executive, Editorial, Circulation, and Advertising Offices: One Phoenix Mill Lane, Peterborough, NH 03458. Second-class postage paid at Peterborough, NH, and additional mailing offices. Postage paid at Winnipeg, Manitoba. Canada Post International Publications Mail Product Sales Agreement No. 246492. Registered for GST as McGraw-Hill, Inc., GST #123075673. Printed in the United States of America. **Postmaster**: Send address changes and fulfillment questions to BYTE Subscriptions, P.O. Box 552, Hightstown, NJ 08520.

COVER IMAGE: STEVEN LYONS © 1994

Subclassing in OLE 2.0

It's not just an API anymore: The Component Object Model of OLE 2.0 is the beginning of object-oriented system services

GEN KIYOOKA

Much has been written about OLE 2.0, and a great deal of it smacks of resistance and harsh criticism. Much of the furor over OLE 2.0 seems to arise from its purported complexity and from the apprehension and resistance that accompany a paradigm shift. It seems that software developers, faced with another challenging advance in software interoperability, are not amused.

The problem is one of perception. Many perceive OLE 2.0 as a newfangled cosmetic add-in for Windows 3.1, along with a needlessly complex set of specifications and implementation requirements. In fact, OLE 2.0 marks the delivery of new operating-system software and provides new tools for managing complexity and solving problems. From a marketing perspective, Microsoft has done a great job of packaging this new architecture. Unfortunately, lost amid the ensuing chaos is the greater impact of a fundamental improvement in how objects of "user-level" granularity are distributed and packaged in a GUI environment. Add in a dash of C++ fundamentalism, and you've got the makings of an object holy war.

Interface Rigor

If you get past the OLE rhetoric and market-speak about visual (i.e., in-place) editing, document centricity, automation, and so on, what remains is a fundamentally rigorous and practical architecture for packaging and reusing software objects. In OLE 2.0, this elegant underlying architecture is called the Component Object Model. Understand this architecture, and you've got OLE licked. Fail to grasp its nature, and you're forever mired in a sea of unfamiliar complexity. Fail to appreciate its value, and you're condemned to sit by and watch a software industry reorient itself around a standardized component marketplace.

The pundits raise three major issues: (1) the apparent disparity between OLE's object model and the inheritance-based C++ object model, (2) the amount of overhead involved in implementing an OLE object, and (3) the sacrilege of suggesting the impending disappearance of applications software from the GUI desktop. The first two can be resolved through a better understanding of the problems the Component Object Model was designed to solve. I'll tackle them one at a time. I leave the third issue to be resolved by time itself, and its realization a testament to the foresight of the OLE architects.

OLE and C++: A Match Made in Purgatory

The first issue might be restated as a proposition: "For Windows to be a true object-oriented system, it should be based on an object-oriented language (e.g., C++) featuring encapsulation, inheritance, and polymorphism. The role of the programmer is to refine the functionality of the base system." OLE does not fully subscribe to or endorse this model as the proper solution for system-level (large granularity) software interconnection.

Thus, under OLE 2.0, the use of C++ or inheritance is strictly relegated to an internal component implementation detail. Other languages and software techniques can be used to implement objects. Publishing an object to be employed by people requires that the object expose a standard and rigorous interface.

MARTY BRAUN © 1994

OLE's lack of support for a standardized inheritance mechanism merely indicates that inheritance is inappropriate for rigorous, standardized software interconnection between components due to be aggregated into appropriate solutions by the end user. Of the other criteria for object orientation, encapsulation holds the place of honor, with polymorphism—or rather, reuse of interface—playing a secondary role.

continued

INSTANCE.CPP: *A minimal OLE application that instantiates the* `CPoly` *object from the OLE developer's kit.*

```
#include <windows.h>
#include <ole2.h>
//
// This line is stolen from \OLE2\SAMP\DISPDEMO\CLSID.H, and it
// represents the definition of a unique class identifier for
// the CPoly polygon server used in the IDispatch sample.
//
DEFINE_OLEGUID(CLSID_CPoly,  0x00020462, 0, 0);

int PASCAL WinMain (HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
BOOL fOk = (OleInitialize(NULL) == NOERROR);
  if (fOk)
     {
   IUnknown FAR* pIUnknown;
   // Creates an instance of the
   // class identified by CLSID_CPoly
   HRESULT hResult = CoCreateInstance( CLSID_CPoly,
     NULL, CLSCTX_LOCAL_SERVER, IID_IUnknown,
     (void FAR* FAR*)&pIUnknown);
     if(hResult == NOERROR)
       {
       MessageBox( NULL,
         "pIUnknown is pointing to an instance of the CPoly class.",
         "Hello World", MB_OK );
       // Release is equivalent to 'destructing' the object
       pIUnknown->Release();
       }
     OleUninitialize();
     }
   return fOk;
}

// Ignore this for now
#include <initguid.h>
DEFINE_OLEGUID(CLSID_CPoly,  0x00020462, 0, 0);
```

OLE Object-Implementation Overhead

The second issue seems to arise when the neophyte OLE programmer is faced with the complex administrative burden of implementing an OLE object or, worse, of implementing a multiplicity of objects and interfaces in a fully OLE-enabled application.

Windows 3.1 introduced a new administrative tool, a version resource, which allowed proper upkeep of shared DLL and EXE packages. OLE 2.0 introduces interface contracts, a system registry for objects, globally unique class and interface identifiers, and a binary standard for exposing interface VTBLs (virtual function tables). The administrative overhead in implementing an OLE object under the Component Object Model is considerable, but it's a fundamental prerequisite to robust interoperability.

Interfaces as Contracts and Objects

Conceptually, an OLE interface (or *protocol*) specifies a contract between two parties (i.e., software components). For instance, to implement drag-and-drop under OLE 2.0, the source and target windows agree to a protocol that involves two interfaces, `IDropSource` and `IDropTarget`. Under the terms of the contract, the window capable of having objects dropped on it implements the functions defined by the `IDropTarget` interface. The window providing the objects that are dragged and dropped onto the target window implements the functions in the `IDropSource` interface. In this case, the protocol involves two separate parties and two separate interfaces. The rigor of this contract ensures that drag-and-drop functionality is implemented uniformly throughout the system.

Unlike an informal grouping of function calls, an OLE interface binds a set of function calls together into a unit as an opaque means for accessing an object. Contrast this with a more informal set of functions in a conventional API. The Component Object Model defines a binary specification of what an interface looks like. More concretely, it specifies a binary description of what an interface is.

This binary specification has these four goals:

1. To provide a function-invocation mechanism that provides a compile-time-type-safe and opaque means for manipulating a software component object
2. To provide polymorphic interfaces for different classes of objects with similar behaviors
3. To provide a limited inheritance from a common shared interface, called `Unknown` (analogous to a base superclass called `Object` in a standard, singly rooted inheritance hierarchy)
4. To allow objects in the local process space and those in remote process spaces to be manipulated in a uniform manner

To achieve these goals, the Component Object Model uses a binary specification of an interface object as a pointer to an opaque chunk of memory whose first 32-bit element is a pointer to an array of function pointers representing the methods that encapsulate the object. This array of function pointers is a VTBL.

Do-It-Yourself Polymorphism

Consider the problems a VTBL interface sets out to solve. Imagine being exposed to Smalltalk in an educational setting and, in your first C programming assignment, being asked to implement an object-oriented, polymorphic class hierarchy with inheritance. The first practical C++ compiler for your operating environment would not be available for several years.

One solution would be a message-passing architecture similar to the one used in the window manager of Microsoft Windows. In this model, polymorphism is achieved through generic parameters whose contents are interpreted according to the message context. Inheritance is achieved by chaining uniformly defined message-handling functions. The message-handling function that first receives the message represents the most specialized subclass in the inheritance hierarchy. It can choose to discard, implement behavior for, or pass a message on to the handler of its immediate superclass. Unfortunately, this method is ill-equipped to handle data definitions at each successive subclass in a hierarchy.

Another solution strikes closer to the heart of the binary interface standard of the OLE Component Object Model: You envision the accretion of both data and functions as proceeding in an orderly fashion down from a general superclass to a specific subclass. Since you're a C programmer fond of `malloc()` and `free()`, you have decided that an object be instantiated by `malloc()` and destroyed by `free()`. To separate the behavior from the private data of these objects, you decide to make the first data element of every object a pointer to an array of function pointers. Each successive specialization in the class hierarchy can add its own new functions to the array of function pointers and its own new data to the private data definition. Only one array of function pointers need be maintained for each class.

This is exactly the binary model used in a single-inheritance C++ class hierarchy. C++ multiple inheritance introduces vulgarities to this otherwise comprehensible and clean model.

Proxy Interfaces for a Uniform Representation

The Component Object Model lets consumers manipulate objects only through the object's interface pointer. Given this opaque definition of an interface, the Component Object Model's final goal can be realized: accessing remote and local objects in a uniform way.

Consider a rectangular chart object that has been inserted into a spreadsheet application. The spreadsheet manipulates the chart object's contents by invoking functions on the object's OLE interfaces. But if another application program implements the chart object, the actual implementation is performed in another process space. Therefore, the interface pointer used by the spreadsheet application points not to the chart object itself, but to a proxy representation of the chart object's interface in the local process space. The proxy object forwards the methods invoked on this local interface (through a lightweight RPC, or remote procedure call) to the actual implementation in another process space.

This is the fundamental magic of OLE 2.0. By performing a major behind-the-scenes effort, OLE exposes a uniform and familiar (i.e., function through-pointer invocation) means for manipulating all objects in the system.

A Minimal OLE Program

Rather than jump into a fully capable OLE application with thousands of lines of code, look at the INSTANCE.CPP listing—the OLE equivalent of "Hello World." As you can see, the most basic requirements for an OLE application do not extend much beyond the basic requirements for a standard Windows application. Execution begins at `WinMain`, and two additional calls, `OleInitialize()` and `OleUninitialize()`, are required for a bona fide OLE application start-up and shutdown. The rest of the code involves instantiation, our next topic.

Remember (from OLE 1.0) that the OLE system maintains a system registry—essentially a hierarchical database containing information about each OLE-capable object server on your computer. Each OLE application is required, as part of its setup program, to merge its information with the registration database. Instantiating an OLE object is similar in principle to late binding or dynamic linking. Applications that know nothing of each other can communicate by invoking functions on objects owned by one another. The registration database is a key part of the process, providing a central repository (i.e., catalog) of system parts.

Creating an Instance of an OLE Object

Look at INSTANCE.CPP again. This code shows how to create an instance of an object. To instantiate an object, you need to know its globally unique class identifier. These class identifiers are stored in the registration database (where you'd normally be obtaining it). But for the sake of clarity, I've taken a copy of the definition of the class identifier for `CPoly` from the code to DISPDEMO, included in the OLE developer's kit, and placed it directly in INSTANCE.CPP.

The Component Object Model allows the consumer to manipulate objects only through the object's interface pointer.

The interface pointer returned by the `CoCreateInstance()` function represents the instantiated object. Having a pointer to the `IUnknown` interface is like having a pointer to `CObject`, the root class, in MFC (Microsoft Foundation Classes). You know nothing specific about the object except how to release your reference to it (`IUnknown::Release()`) and how to ask it for other interfaces it may support (`IUnknown::QueryInterface`).

To run INSTANCE.EXE on your system, your computer has to have the OLE 2.0 developer's library installed. If the `Dispatch` polygon sample programs function correctly, INSTANCE .EXE should, too. This is a large-model program created with Microsoft Visual C++; it links implicitly to the OLE2.LIB (OLE2 .DLL) and COMPOBJ.LIB (COMPOBJ .DLL) import libraries.

The Subtleties of Using Interfaces

Now that you have had a chance to examine the instance application, you should be familiar with the basics of OLE objects:

QUERYINTERFACESOF *is a method of the CBrowseDlg class, which, when given a pointer to the* `IUnknown` *interfaces, invokes the* `QueryInterface` *method to see what interfaces are supported by the object. For each interface that is supported, it adds the name to the listbox.*

```
void CBrowseDlg::QueryInterfacesOf(IUnknown FAR* pIUnknown)
{
POSITION Position = m_RegInterfaces.GetHeadPosition();
COLEInterface *pInterface;
  while ( Position && (pInterface =
    (COLEInterface *)m_RegInterfaces.GetAt(Position))) {
      const char * Name = (const char *)(*pInterface->GetName());
      IID InterfaceId;
      if (NOERROR==IIDFromString( (char*)(const char *)
        (*pInterface->GetIID()), &InterfaceId)) {
          IUnknown FAR* pQueriedInterface;
          if (pIUnknown->QueryInterface(InterfaceId,
            (void **)&pQueriedInterface)==NOERROR) {
               m_InterfaceLB.AddString( Name );
               pQueriedInterface->Release();
          }
      }
      pInterface =
        (COLEInterface *)m_RegInterfaces.GetNext( Position );
    }
    m_InterfaceLB.AddString( "IUnknown" );
}
```

NOW, HAND OUT INSTANT INFO VIA FAX.

FaxFacts is the revolutionary fax-on-demand system that allows your customers and prospects to request information by simply pressing a few phone buttons. In seconds, a fax "full of facts" is sent to the caller.

- Instant information available 24 hrs. a day.
- Reduces operating costs.
- Ideal for dispersing literature, specs, pricing or customer support.
- Provides answers, while customer's interest is peaked.
- Credit card charge per fax.
- Easy to install by non-technical person.

TRY THIS DEMO:
708/924-3030
DOC. NO. 889812

Faxfacts® by Copia International Ltd.
Wheaton, Illinois 60187 708/682-8898

In Touch With Tomorrow's Consumer
The Los Angeles Computer Fair
March 18, 19, 20, 1994

Opening in the nation's largest computer retail market, this unique, first-ever event combines the exhibits and educational programs of traditional trade shows with the high-volume excitement of today's superstores.

The Los Angeles Computer Fair is the first in a series of regional consumer-oriented computer trade fairs presented by National Productions. Come see, compare, and learn more about hundreds of products. Take advantage of special show pricing, great bargains and build your knowledge base.

If you're looking for great buys and solid information, the Los Angeles Computer Fair is the place to be. Look for us in Los Angeles in March, '94 and Atlanta in October, '94.

Call for more information: 1-800-800-5600

Presented by National Productions, California's leading producer of computer retail shows for over ten years.

Co-Sponsors:

MICROTIMES PIBMUG Pasadena IBM Users Group

National Productions, Inc.
16175 Monterey Road, Ste. 4
Morgan Hill, CA 95037
TEL (408) 778-5200
FAX (408) 779-1374

Los Angeles COMPUTER FAIR

constructing an instance of a class, invoking functions on the interface pointer (`instance`), and destructing the instance (`Release`). It's time to expand the scope of this discussion.

Obviously, the interface `IUnknown`, with just three members (i.e., `QueryInterface`, `AddRef`, and `Release`), has limited applicability for creating a compound document implementation with in-place editing, OLE automation, and the like. Under OLE 2.0, you generally do object instantiation by requesting a new instance represented by a pointer to the `IUnknown` interface (as in INSTANCE.CPP). The consumer using this object then queries the object about its capabilities by requesting further, more specific, interfaces through the `QueryInterface()` function.

If the object is capable of supporting the functionality implied by the interface, `QueryInterface()` gives the consumer additional and more capable means of manipulating the object. An application developer can begin by implementing a few interfaces and successively add functionality until the complete OLE feature set is realized.

IBROWSE.EXE is a small program that demonstrates the generalized model of locating and instantiating objects by way of the system registry. It also demonstrates the use of `QueryInterface()` as a means of interrogating an object to determine that object's capabilities. The program enumerates all the object classes in the registry. You simply choose a class from the first listbox, and you see a second listbox populated with the names of the interfaces that an object of the selected class is capable of supporting.

IBROWSE.CPP simply instantiates the object requesting the `IUnknown` interface. Once this interface has been obtained, IBROWSE.CPP enumerates all the interfaces listed in the system registry, calling `QueryInterface()` on the `IUnknown` pointer for each interface type. By simultaneously examining this program and exploring the structure of the system registration database, it's easy to understand the role of the registry in OLE 2.0. QUERYINTERFACESOF is from the IBROWSE source code. (You can explore the structure of the system by using the REGEDIT.EXE utility with `/v` on the command line. This utility is distributed with the source code associated with this article.)

Because of the opaque nature of interfaces, and because the implementation and interface of an object can span process and even machine boundaries, managing memory in this object model has some inherent complexity. Unfortunately, the Component Object Model places the burden of managing this complexity squarely on the shoulders of the implementer. Ironically, just as C++ introduced a convenient automatic constructor/destructor model for reducing memory management complexity common in C programs, the Component Object Model introduces a reference-counting system. But unlike in Smalltalk, which provides automatic language (i.e., transparent) support for object reference counting, the C++ or C programmer has to be mindful of a bevy of reference-counting rules. Two steps forward, one step back.

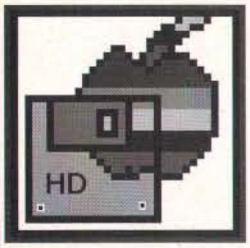
For simple programming examples like the ones accompanying this article, the use of reference counting is trivial. As the implementer creates an instance of an interface pointer for the consumer, the reference count on that interface is bumped by one. This is not evident in the accompanying listings because it takes place in the private code of the application (or DLL). The use of `Release()` (invoked on `IUnknown`) is visible in IBROWSE .CPP: Each interface pointer obtained from `QueryInterface()` is released, as is the initial interface pointer obtained by `CoCreateInstance()`.

continued

Circle 150 on Inquiry Card.

Circle 143 on Inquiry Card.

Your PC goes Mac!!



MacDisk*

Reads, writes and formats Mac 1,44 MB floppies.

Under Windows, file exchanges between Word, Excel, PageMaker, XPress, and other sibling applications, without any hardware modification.

MacSQ*

Reads/writes 44/88 MB Mac SyQuest cartridges.

Transfer of scan files, colour separations. Supports Adaptec/ASPI-compatible adapters, low-cost adapters and parallel port solutions (Puma, D2-Para). Under Windows with ASPI, under DOS for other configurations.

MacScuzzy*

A superset of MacSQ, formerly announced and advertised as MacDisk Pro, also drives 128 MB MO cartridges and 90 MB Bernouillis and soon even bigger Mac SCSI volumes (600 MB MO).

Logiciels & Services DUHEM

21, rue La Bruyère - F-75009 PARIS (France)
Tél. (33.1) 49 70 04 55/Fax (33.1) 49 70 04 56
Free leaflet and demo disk. Offer good until supply lasts.
MacDisk **125 $,** MacSQ **200 $,** MacScuzzy **245 $** (without S&H fees).
EEC residents, add French VAT (18,6 %). We accept most credit cards.
Dealers welcome. Software in French/English/German, Manual in French/English.
* Registered trademarks of L$D in France and trademarks of LSD elsewhere.

Subclassing and OLE Interfaces

I mentioned earlier that the Component Object Model does not allow subclassing—that is, taking an existing interface and refining the behavior of methods in that interface. However, just as every great musician eventually learns the appropriate time to break rules concerning embouchure, harmony, and form, making maximum use of the OLE architecture smacks of misbehavior.

Subclassing under OLE 2.0 is as simple as providing an intermediary any time a request is made for an interface pointer, through either `QueryInterface()` or standard function calls to the OLE libraries, like `CoCreateInstance()`. The intermediary forwards the call to obtain the interface pointer. However, it also creates a mock interface, stores the actual interface pointer in this mock interface's private data, and passes the mock interface back to the caller. Since the details of the interface methods are well known, any function invoked on the mock interface can be forwarded to the actual interface with any desired pre- or post-processing.

Not surprisingly, this is also an accurate description of the proxy-interface stubs and marshalling used by the OLE system to forward method invocations over process boundaries. Using these techniques completely violates the pure theoretical underpinnings of interfaces under the Component Object Model, so if anyone asks how you happened on the notion, I'd appreciate it if you recall how it came to you in a dream.

Hooking into standard function calls like `CoCreateInstance()` can be messy. However, there's a much easier, flexible, and general method, illustrated by the HANDLER sample in the OLE 2.0 developer's kit. The sample illustrates the use of subclassing in creating a nifty debugging tool.

HANDLER is an example of exposing OLE interfaces from a DLL instead of by a separate EXE. It stands as an intermediary directly between the consumer and the provider of interfaces by rewriting the registration database entries to trick the OLE system into requesting interfaces from the handler instead of from the actual objects (tucking the original entries away for safekeeping).

The Component Object Model affords other conveniences for large-granularity reuse through aggregation. But the number of practical circumstances in which granular objects can be aggregated into new entities is yet to be adequately demonstrated. For some, genetically mutating Excel with PageMaker through aggregation may be the fulfillment of a lifelong dream; for others, decidedly not.

Dispelling any initial impressions of OLE 2.0 that you may have gathered may not always be easy. However, aside from the thorns surrounding reference-counting semantics, the model underlying OLE 2.0 is well thought out, clean, and simple. In fact, the uniformity and simplicity of the model make it possible to attempt such wondrous feats of software as in-place editing.

No software designer committed to object-oriented methods, usability, or the benefits of software systems that can be successfully managed will think twice about employing OLE. Those who do hesitate will one day complain that Microsoft unfairly dominates component software solutions, even though today we all stand as equals on the threshold of this new order. ■

Editor's note: *Both source code and executable files are available electronically; see page 5 for details.*

Gen Kiyooka (San Diego, CA) likes OLE and its implications and, as a tool developer, welcomes any suggestions that will make the construction of OLE software components pleasurable and productive. You can reach him on the Internet or BIX at gen@bix.com.