ObjectiveView

The Journal For Managers and Technical Staff In Object and Component Development

Summer '98 Issue

Welcome to issue 2 of ObjectiveView. This issue sees the start of our Object Architecture and SubjectiveView series, which will look in-depth at object infrastructure products. Our ObjectDesign series continues with discussion of the Open Closed principle, and we have the results of a recent survey on business objects. Email us now (in confidence) for your FREE subscription.

Object Design Series

The Open Closed Principle

For: Object Architects and Designers

How do you make components extensible without modification?

Robert C. Martin discusses - page 37

Architecture/SubjectiveView Series

End To End Objects

For: Object Architects, Project Mangers, Technical Staff

Relational or Object Database?

Keiron McCammon - puts the ODBMS Case - page 21

Introducing Object Technology Series

Introducing CORBA

For Managers and Technical Staff New To CORBA

Author Thomas Mowbray introduces CORBA - Page 6

Architecture/SubjectiveView Series

Middleware Wars

For: Object Architects, Project Mangers, Technical Staff

How do you develop inter-operably for RPC, CORBA and COM at the same time?

Anne Thomas on Noblenet Nouveau -12

ObjectNews - page 2

OMG Analysis - page 36

Subscription Details

FOR YOUR **FREE SUBSCRIPTION** TO OBJECTIVE VIEW

Email: objective.view@ratio.co.uk-Tel: Rennie Garcia on 0181 579 7900 or simply complete the box below and post or FAXBACK (0181 579 9200) PLEASE SUPPLY THE FOLLOWING:

Name …….. ……Daytime Tel.No .....…….

Job Title.....……Email……

Address.....…… Company.....……..

…….. Preferred delivery: (Please tick) Hardcopy…… Email……

Postcode.....…….

Design/Layout: Kate Harper

Editor: Mark Collins-Cope

Object News

Quarter 2, 1998

Dion Hinchcliffe discusses the latest object news.

The second quarter of 1998 was an exciting one for the object technology industry and included a variety of important announcements, trends, and product introductions in the areas of CORBA, COM, UML, OO languages, and methodology.

CORBA

Although no major revisions to the commercial ORBs occured during 2Q, Java support in ORBs continued to make serious headway especially in many of the free ORBs such as JacORB and ORBacus 3.0.

OMG made available new interoperability specifications to improve language mapping between Java and IDL as well as other standard languages. The new specifications allow Java to work well with client or servers written in other languages and will allow Java to be used in far wider variety of CORBA settings.

The first 64-bit ORBs began making their way to market with an announcment in May that Expersoft's CORBAplus ORB is now available for Digital Unix. 64-bit ORBs, while still a novelty, may permit scalabilty beyond what is available for any other current distributed object technology.

Applications servers built around CORBA continued to pour into the market during the 2Q including BEA Systems M3, SilverStream's 2.0 app server, and Tengah's WebLogic 3.1 to name a few.

The Object Management Group announced the adoption of the Business Object Component Architecture for CORBA that will bring a standardized infrastructure for generation of business objects from design specifications.

COM

Two non-events during 2Q of 1998 illustrate the somewhat pensive situation in the Component Object Model world. The first is the lack of a COM+ beta which was scheduled to be released during this time period. COM+ is a forthcoming upgrade to COM which improves ease-of-use, provides standard services for security, object lifecycle, and Numerous new UML modeling tools were in beta during 2Q with release dates in 3Q. New transactions and largely does away with IDL. The COM+ beta is now tenatively going to be released in 3Q or 4Q 1998.

Also expected during the 2Q was NT 4 Service Pack 4 which was to bring interim features to COM including DCOM over HTTP which would bring full web protocol support to DCOM, a feature much in request by the COM user community. Service Pack 4 has now indefinitely been delayed and it's release is uncertain in the face scheduling turmoil within Microsoft's NT division.

COM for Solaris did make it out of Microsoft during 2Q 1998 and is now generally available and is fully supported externally by Microsoft. The price is steep at $3500 but may not matter much compared to the production cost of a full distributed object system on Solaris. COM for Solaris is the first of several products Microsoft is hoping to release to eliminate what many consider to be COM's Windows-only stigma.

An interesting trend began developing during 2Q 1998, that of bridges opening up between CORBA and COM. IONA, Expersoft, and other announced bridges between COM and CORBA in 1Q 1998 but then IONA made the startling announcment that Microsoft and IONA will integrate Microsoft Transaction Server with IONA's Obect Transaction Manager product that will allows transactions started in one products to be finished in the other. Microsoft has reportedly been reluctant in the past to built bridges to CORBA but is now apparently relenting.

Unified Modeling Language

The Unified Modeling Language 1.2 was adopted by the Object Management Group's UML Revision Task Force during 2Q although the text of the standard is still not fully available except in draft form.

tools scheduled for release included Object International's Together/J 2.0, NoMagic's MagicDraw UML, and Object Domain 2.0.

Object-oriented languages

Java continued to evolve with numerous new initiatives and APIs becoming available. JDK 1.1.6 was released during the 2Q and remains the currently Java Development Kit until the forthcoming and much-awaited JDK 1.2 is released. IBM also announced that Java will be a first class citizen in its CICS world and in fact, plans on bringing CICS's formidable capabilities to the Java masses over the next several quarters. Enterprise JavaBeans were another exciting area in the Java world and 2Q included some of the first functioning EJB contains from EJBHome, Weblogic, and NetDynamics to name a few.

C++ vendor Inprise (formerly known as Borland) released C++ Builder Enterprise that includes first class support for distributed object standards such as CORBA, COM, and Entera. Inprise release Delphi 4, in what many assumed to be a pre-emptive strike against Microsoft's forthcoming Visual J++ 6.0. The Delphi 4 product includes full support for the latest distributed object models. Interactive Software Engineering, makers of Eiffel, announced the availability of an open source version of their core library, EiffelBase.

Methodology

As expected, Rational Software announced at their Rational User Conference in June that their commercial development method, Objectory, will become the industry's Unified Process, at least as far as Rational is concerned. The original collaboration between the Three Amigos (Grady Booch, Jim Rumbaugh, and Ivar Jacobson), the ultimate product for which they came together (creating the Unified Modeling Language along the way) was a merger of their respective object-oriented development methods. Objectory will be renamed the Rational Unified Process and become available in the next few months. Although the UML has now attained standardization through the OMG and its documentation is freely available, the future and availability of the Unified Process remains unclear since it remains a commercial product from Rational.

Wrap up

Another major trend occuring during the second quarter was an apparent undercurrent of movement away from an object-centric industry, to one of components developed with objects. This trend was best signalled by the famous SIGS publication Object Magazine changing its name to Component Strategies during the second quarter. Look for more organisations to change their focus from objects to components during the 2nd half of 1998.

The object technology industry looks forward to major new events in the 2nd half of 1998 including the release of a COM+ beta, CORBA 3.0, UML 1.3, JDK 1.2, and many new exciting middle-tier announcements such as applications servers from Inprise and CICS for Java from IBM.

Object News (http://www.objectnews.com) is a free, daily object technology newsletter that focuses on issues in UML, COM, CORBA and methodology. Dion Hinchcliffe is co-editor of Object News with Paul Evitts and is an senior object technolgoy consultant with Object System Group. Mr. Hinchcliffe can be reached a dhinchcliffe@objectnews.com

A Master Class in
Advanced Principles of Object Oriented Design

Dates: December 1-2, 1998, London.

Presented by:* *Robert C. Martin

Author:* *Designing Object Oriented C++ Applications

Editor:* *C++ Report

Please Note: Classes are limited to a maximum of 16 paying attendees, so please book early.

Robert C. Martin
is an international consultant and president of Object Mentor Inc. He is author of "Designing Object Oriented Applications Using the Booch Method", and his forthcoming book: "*Patterns and Advanced Principles of OOD"* is soon to be published by Prentice Hall. Robert is also editor of C++ Report.

To book call: *0181 579 7900* and ask for Rennie Garcia, or email:**info@ratio.co.uk** with full contact details.
Please register with Ratio Group (email:info@ratio.co.uk) if you would like further information on up and coming masterclasses, including:
Analysis Patterns, CORBA Design Patterns, Designing Components Using Design Patterns

Do You Know The Object...

Object Oriented Public Course Schedule

Newly updated and developed courses starting September 1998

September			October		
Dates	**Course**	**Cost**	**Dates**	**Course**	**Cost**
7-8 ***(Expert Level)*** RAT151	Ralph Johnson A Master Class in Design Patterns and Frameworks	**FULL**	21-23 ***(Updated)*** RAT150	Software Engineering Using Design Patterns	£795
8 ***(New)*** RAT100	Intensive – OO Concepts	£425	5-9 ***(Updated)*** RAT220	Pragmatic OO Java Development Workshop	£1,195
9 ***(New)*** RAT104	Advanced Skills in OO Project Management	£425	12-16 ***(Updated)*** RAT102	OO Analysis & Design Using UML	£1,250

November			December		
Dates	**Course**	**Cost**	**Dates**	**Course**	**Cost**
2 RAT100	Intensive - OO Concepts	£425	1-2 ***(Expert Level)*** RAT151	Robert C. Martin A Master Class in Advanced Principles of Object Oriented Design	£995
4 RAT104	Advanced Skills in OO Project Management	£425	7-11 RAT102	OO Analysis & Design Using UML	£1,250
9-13 ***(Updated)*** RAT210	OO C++ Programming Workshop	**FULL**	14-18 RAT210	Pragmatic OO C++ Programming Workshop	£1,195

Ratio have invested heavily into the production of highly up to date course materials and case studies looking at the issues of today's OO & Component Based systems.

Please contact Rennie Garcia @ Ratio Group on 0181 579 7900 or email: info@ratio.co.uk
for details on:
Public training courses
In-house courses
Object oriented computer based training programme

Special Offer: *Free* C++ Video Set with every December Booking on OO C++ Development (Quote: ObjectiveView offer) Normal cost - £550.00.

Visit our web site:www.ratio.co.uk for hot OO links, new technical whitepapers and details of the *ObjectTeam* Case Tool.

Introducing CORBA

Thomas Mowbray, Author of the bestselling book "Corba Design Patterns", gives an introductory overview of CORBA.

Introduction

CORBA is a standard from the Object Management Group (OMG), an international industry consortium whose mission is to define interfaces for interoperable software using an object-oriented technology.

Their specification, the Common Object Request Broker Architecture (CORBA) is an industry consensus standard that defines a higher level facility for distributed computing.

In general, object-orientation enables the development of reusable, modular software, and it is moving technology towards plug-and-play software components.

The OMG's efforts are extending these benefits across distributed heterogeneous systems.

.

Object Management Architecture

Figure 1 is an overview of the OMG's object management architecture (OMA). The OMA Guide identifies four categories of software interfaces for standardization. The central component of the architecture is the Object Request Broker (ORB). The ORB functions as a communication infrastructure, transparently relaying object requests across distributed heterogeneous computing environments. The CORBA specification covers all the standard interfaces for ORBs. CORBAservices comprise a set of fundamental services such as object creation and event notification. CORBAfacilities comprise a set of high level services such as compound documents and system management.

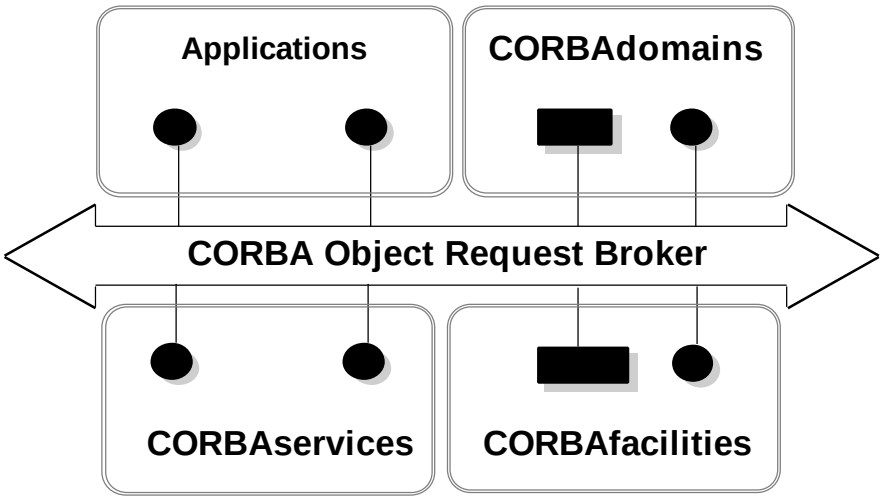


Figure 1. Object Management Architecture

CORBA domains comprise vertical market areas, such as financial services and healthcare. Application interfaces comprise all the remaining software interfaces such as proprietary commercial interfaces and legacy interfaces.

CORBA simplifies distributed systems in several ways. The distributed environment is defined using an object-oriented paradigm that hides all differences between programming languages, operating systems, and process locations. The object oriented approach allows diverse types of applications to interoperate at the same level, hiding implementation details and supporting reuse. CORBA defines a very useful notation for defining software interfaces

called the Interface Definition Language (ISO IDL). CORBAservices defined in ISO IDL have a dual role: ORB vendor provided services and application provided services. Developers are encouraged to reuse and extend the standard interfaces.

Adopted

The first ORB standard, CORBA 1, was adopted in December 1991 [2]. The current version, CORBA 2, was adopted in 1994; this specification is upwardly compatible with the CORBA 1 specification. CORBA 3 was completed in 1997 with the addition of portability interfaces and Java bindings. In particular, the interface definition language (ISO IDL) has been a stable part of the specification since 1991.

ISO IDL is the notation used for interface specification in the OMG standards and is used by other standards groups such as X/Open, ECMA, and ISO. The CORBA 3 specification includes standard language bindings for C, C++, Smalltalk, Ada95, COBOL, and Java.

There have been more than a dozen CORBAservices adopted such as: Naming, Event Management, Persistence Life Cycle, Concurrency, Externalization, Relationships, Transactions, Query, Licensing, Security, Time, Trading, Collections, and Properties.

There are many other standards in process at the time of this writing. Additional CORBAservices in process include asynchronous messaging. And the first set of domain interface specifications from telecommunications, healthcare, and financial services task forces.

OMG has reengineered standards group adoption processes to make them much more responsive to market needs. Instead of a typical 4 to 7 year process for many formal standards, the OMG process can converge on multi-vendor specifications in about a year. OMG has successfully completed this process more than a dozen times as evidenced by the available specifications. The adoption process also has a built-in assurance that the adopted specifications will be supported commercial products. This has worked well in practice, with more than a dozen available implementations of CORBA to choose from.

Getting Started with CORBA

There are many ways to prepare for your organization's technology transition to CORBA. The OMG has published the Object Management Architecture Guide and the CORBA Specification. The CORBA specification is available on-line (i.e. its FREE), at http://www.omg.org/corbask.htm

CORBA vendors offer comprehensive training courses covering their CORBA products. These courses offer insight into the standard and the technology. Successful CORBA development experiences have been documented Mowbray, T.J. and R. Zahavi,

THE ESSENTIAL CORBA: SYSTEMS INTEGRATION USING DISTRIBUTED OBJECTS, John Wiley & Sons, New York, 1995. (ISBN 0-471-10611-9))

For more CORBA and software architecture information, also visit:

http://www.serve.com/mowbray

Ratio Group can make various introductory and advanced training courses available on CORBA-based software development, components, frameworks, architectures, and design patterns. Contact: Rennie Garcia on +44 (0)181-579-7900 or email:info@ratio.co.uk

CORBA Market Share

It is estimated by OMG (and often quoted) that over 17% of the Fortune 500 companies are using CORBA today. Numerous mission-critical systems have been implemented using CORBA. For example, the UK passport control systems is a CORBA-based system that has been operational for years. An entire book by Jeri Edwards is devoted to CORBA case studies which includes many successful systems developments in a wide range of domains.

In our opinion, several UK-made CORBA products have some of the most advanced features and capabilities. For example, the ICL DAIS ORB (made in Manchester-Gorton UK) has the most complete complement of CORBAservices, including the essential Trader Service, a yellow pages directory based upon ISO standards. The Real Objects Ltd. InterBroker (made in Leamington Spa, UK) is the only known ORB which has successfully passed long-term load-testing benchmarks. This makes InterBroker the only proven ORB

in the world for 24 hour by 7 day (24X7) mission critical applications.

In comparison, the Microsoft infrastructures DCOM and COMplus have no measureable market share. There are also few known success stories with this technologies, but many disaster sagas about enterprises struggling with pernicious system management issues in the Microsoft operating systems. DCOM was created by Microsoft in 1995 to be a multi-media infrastructure for the Internet. But since there were no security provisions whatsoever engineered into DCOM, it simply cannot be used on the Internet for example, any DCOM-enabled ActiveX component that is downloaded from the Internet has complete and unlimited access to your Windows95 or Windows NT system resources.

DCOM is being actively obsolesced by Microsoft in favore of COMplus. COMplus is an announced future infrastructure from Microsoft. However, COMplus development tools are still immature, and Microsoft has not selected a delivery vehicle for COMplus. In other words, COMplus will not be included in Windows98 nor Windows NT5, and it is not decided how COMplus will be disseminated. This means that COMplus will not be widely available until at least the year 2001. In the meantime, CORBA is ready, working, and available now on virtually every operating system platform, including all Microsoft releases, on systems ranging from DOS to workstations to IBM mainframes.

Looking at one significant market, defense, the U.S. Department of Defense has adopted CORBA as its standard technology for object-oriented system development. Two infrastructures are mandated by the Joint Technical Architecture, an operational profile of the TAFIM, approved in FY96. CORBA is mandated for use on object-oriented DoD systems. OSF DCE is mandated for use on procedural DoD systems. Contact: http://www.itsi.disa.mil for more information. ISO IDL has been used on many DoD programs and in specifications. For example, the Common Imagery Interoperability Facilities (CIIF), the Image Access Services (IAS), and the Common Imagery Interoperability Profile (CIIP) comprise a prime example of how to architect DoD systems for interoperability. For more information, visit:

http://www.itsi.disa.mil/ismc/ciiwg

These case studies of interoperability are documented in Chapter 10 of the book Inside CORBA. The domain specifications are some of the most exciting developments in CORBA standardization. These interfaces enable application-level interoperability across multi-vendor systems. For example, the UK MoD contractors along with multi-national defense organizations and contractors have formed the OMG C4I Special Interest Group. OMG C4I has the mission of identifying remaining technical gaps in standards and technology for distributed object specifications and products, as well as adopt specifications for interoperability. This group is international in scope; consider the needs for multi-national regional conflict support as well as mainstream US DoD C4I systems implementation.

Formal Standards Groups and CORBA

The International Standards Organization (ISO) has approved a Draft International Standard based on CORBA. ISO DIS 14750 is a universal notation for defining application program interfaces (API). ISO IDL is identical to CORBA IDL.

The European Computer Manufacturers association (a peer group to US ANSI) has been using ISO IDL for several years to specify all application program interfaces (API). X/Open has used ISO IDL extensively in its specifications. Numerous other standards groups are following suit. ISO IDL-based specifications have the benefit of defining multiple programming language bindings automatically from one API specification. This advantage will be used by multiple ISO standards bodies.

CORBA Design Patterns

In 1991, MITRE began a research project (called DISCUS) on how to achieve universal interoperability between all forms of government application as well as commercial software. Early in the activity, we discovered that OMG had a similar mission, and we began working with CORBA technology. From the start, we identified ISO IDL as a very useful notation for describing government software APIs. We have based all our approaches around the use of ISO IDL ever since, regardless of the underlying infrastructure (ORB, RPC, library interface or other networking stack). Luckily, groups like X/Open and ISO picked up on this concept.

After years of successful use and reuse of the DISCUS API written in ISO IDL, we began other ISO IDL-based initiatives, and we began to mentor others in the use of CORBA technologies. To our surprise, we discovered that people were using CORBA just like they used sockets or RPC; there was no "paradigm shift" in their practices or the form of the resulting systems. Ideas that seemed obvious to us about software architecture and frameworks were not being practiced by government systems developers.

To remedy this problem, we began a campaign of education and evangelism to help people to build better systems. Our first book, The Essential CORBA, captured this guidance through discussion of the DISCUS architecture.

We also felt that a more structured form of guidance was needed, to make this available to the widest possible group of government developers. Our eventual goal for our work to evolve into a government guideline, analogous to the TAFIM Volume III, but containing much more specific implementation guidance. The product of this CORBA experience, mentoring, and research is CORBA DESIGN PATTERNS.

We have captured our lessons learned and guidance in a new book published by John Wiley & Sons Publishers and the Object Management Group. CORBA DESIGN PATTERNS is written by Dr. Tom Mowbray (author of The Essential CORBA) and Raphael Malveau, experienced CORBA architects, developers, and object mentors. CORBA Design Patterns is the quintessential guide to successful development using distributed objects. The "design patterns" format of the book makes the expertise readily applicable to real-world object-oriented design and development challenges. Containing 39 design patterns in a new pattern language and catalog.

The book breaks new ground for application development, software architecture, enterprise applications, and Internet technologies. The book provides a definitive reference model for software design levels and a comprehensive set of design patterns covering each level. The reference model details the key forces that drive software decision making. The reference models allows you to find the right patterns quickly and apply sophisticated solutions properly for significant benefits. The patterns include well-thought-out benefits and consequences; this makes justification of software decisions easy. Specially featured are patterns for applying Java(tm) applets and ORBlets, as well as, integrating legacy applications For information email:compbks@jwiley.com or visit web:www.serve.com/mowbray/CDPflyer.htm.

Conclusions

ISO IDL is a stable API notation that can be used today for specifying application architectures. The greatest benefits can be realized when ISO IDL is used to define reusable architectures and services. This allows organizations to leverage software development between projects and it simplifies the upgrade of distributed software systems. This enables application software architects and developers to rely upon stable architectural specifications and not upon specific vendor implementations, which are subject to obsolescence

Tom Mowbray, PhD, the Chief Scientist of Blueprint Technologies Incoporated, is the Architectures Columnist for OBJECT Magazine, and co-author of the bestsellers: The Essential CORBA, CORBA Design Patterns, Inside CORBA, and the new book: ANTIPATTERNS: Refactoring Software, Architectures, and Projects in Crisis. Dr. Mobray may be contacted on email:mowbray@www.serve.com

For a daily update on the latest object related news…

and a whole lot more…

check out the hottest object news site on the web…

http://www.objectnews.com

News
Online UML references
Online Methodology references
Online Process references
Online COM references
Online CORBA references
Online Patterns references
Online Online papers
Online People in objects
…

Patterns '98

The Reuse of Ideas in Software Development

The Britannia Hotel, Manchester, UK
October 28 & 29, 1998

Featuring
Bruce Anderson, Erich Gamma, & Thomas J Mowbray

Repeated by popular demand!
February 1998 event was oversubscribed

This conference aims to meet the growing needs of software developers to understand how they can enjoy productivity and software quality benefits using pattern-based techniques. World class speakers have been recruited from Europe and the USA to share their expertise with the audience.

Patterns are vehicles for re-using ideas. Patterns try to identify commonly recurring themes in building software. The interest in patterns has been steadily growing in recent years, although the concept of using them in software development was first identified some 20 years ago. The foundation work for creating design patterns was written in 1977 by Christopher Alexander. He described the use of design patterns to record the wisdom of great architects so that others could reproduce building designs which were known to be of high quality. Building on this work we now have patterns for software design, analysis, requirements, project management, organisation and process, architecture, software migration, data models and frameworks. Patterns are also emerging in end-user domains, including those in Accounting and Manufacturing.

This conference is aimed at software developers, analysts and designers of object oriented systems, software development managers and software architects.

For more information:
Tel: +44 (0)181-570-2182
Email:eric_leach@compuserve.com
Web:http://ourworld.compuserve.com/homepages/eric_leach

Sponsored by:
OMG
OBJECT MANAGEMENT GROUP

NobleNet Nouveau

Distributing OO Applications

One major dilemma facing many product vendors and development organisations is whether to use COBRA, COM or RPC to distribute their applications. All offer a potential solution, but at the cost of limiting the platforms on which applications may run.

Anne Thomas offers a potential solution in NobleNet's Nouveau product.

Introduction

Middleware Wars

In the age of client/server, distributed objects, and the Internet, heterogeneous systems are a fact of life. A growing number of middleware solutions are available to provide connectivity between heterogeneous systems. However each middleware solution provides its own unique set of infrastructure services, and these infrastructure services are incompatible. Therefore organizations often find they must standardize on a single middleware solution for all application systems in order to ensure easy interoperability. Relying on a single middleware solution can be a risky proposition, though, since application systems are very dependent on their middleware. The industry has not yet settled on a single middleware solution, and it is not clear which middleware vendors will survive.

The Leading Contenders

The leading contenders in the middleware wars are Microsoft's Component Object Model (COM); Object Management Group's Common Object Request Broker Architecture (CORBA); and various Remote Procedure Call (RPC) systems from the Open Group, Microsoft, and Sun. Recently, Sun's Java Remote Method Invocation (Java RMI) has also started to attract attention, but Java RMI supports only homogeneous (Java to Java) communications. Although all of these systems perform similar functions, each uses a different programming personality that makes the systems incompatible.

No Single Best Option

Unfortunately, it's very difficult to select a single middleware system. Each system has its advantages and disadvantages. COM provides a natural interface to desktop-based visual application development tools, but COM has almost no presence on Unix or the Internet. CORBA is a platform-independent middleware solution, but it doesn't provide as simple integration with Windows applications and tools as COM. RPC is the tried and true middleware solution for C applications, but it doesn't easily support object-oriented or scripting languages. The best solution would be a combined solution: use COM with visual development tools on Windows and NT, use CORBA with Java and C++ applications on Unix and the Internet, and use RPC with C applications. The challenge is getting the different systems to interoperate.

The Bridge Approach

Most CORBA and RPC systems provide integration with COM using a bridge approach. A *bridge* is an intermediary, or gateway, that converts a distributed request from one middleware format to another. For example, a COM/CORBA bridge supports transparent bi-directional interoperability between any COM client or server and any CORBA client or server.

Overhead

Although the bridge solution effectively establishes connectivity between the two worlds, it often imposes a significant amount of overhead on each request. For each call, the bridge converts and remarshals the datatypes and reissues the request using a different wire protocol.

Programming Mismatch

The bridge solution also does not address the native programming issue. COM, CORBA, and RPC present different constructs to the programmer, and interoperability is not always completely transparent

Native Internetworking Using Nouveau

NobleNet recommends a different type of solution: one based on native internetworking. NobleNet Nouveau is a middleware development and runtime environment that natively supports COM, CORBA, and RPC infrastructures. Middleware connectivity is

implemented directly in the client and server communication stubs within the applications. No intermediary bridges are required.

Nouveau IDL Compiler and CORBA 2.2 Compliant Runtime

Nouveau consists of an IDL-based development tool and a CORBA 2.2 compliant runtime middleware system. The Nouveau IDL compiler can compile interfaces written in CORBA IDL, Microsoft IDL, and RPC IDL. Regardless of the type of IDL used as input, the compiler generates client and server stub code in the developer's choice of CORBA, COM, or RPC programming interfaces.

Transparent Interoperability

At runtime, the Nouveau-generated stubs use the standard CORBA protocol, IIOP, thereby supporting full interoperability with any CORBA-compliant system. In addition, each stub supports the programming interface and dispatching mechanism of its native personality, thereby fully supporting both COM and RPC. Nouveau enables transparent interoperability among CORBA, COM, and RPC applications.

Innovative Approach

NobleNet uses an innovative approach to address the issues of heterogeneous interoperability. Nouveau seamlessly connects disparate systems using a common wire protocol, and yet it still preserves the full native programming experience. Developers can use whichever programming model is most appropriate for a specific application component, and then trust in Nouveau to reliably and efficiently connect it to the rest of the environment.

IDL Middleware

Heterogeneous Systems

In the days of monolithic systems, organizations developed applications based on tools supplied by a single vendor, and all systems could easily interoperate. But today, in the age of client/server, distributed objects, and the Internet, few organizations have the luxury of implementing homogenous systems. Heterogeneity is the rule. And it's up to the Information Technology (IT) organization to find the means to integrate disparate systems.

Middleware

Middleware is the collective term used to describe integration software. Middleware automates the process of passing requests and data between applications that are physically or logically separated. Middleware comes in many different flavors, providing different levels of service and automation. Early SQL-based middleware products were designed to provide remote access to databases. Message oriented middleware products provide a versatile low-level programming interface to support asynchronous communications. IDL-based middleware products provide a high level of automation to support transparent synchronous distributed computing.

Remote Procedure Call

IDL-based middleware was a natural outgrowth of structured programming techniques. A structured program consists of a main routine with a set of procedures that are invoked using a local procedure call. Remote Procedure Call (RPC) middleware enables an application to invoke a procedure that is executing on a remote system using the same programming paradigm as a local procedure call. An RPC replaces the local procedure with a generated piece of code on the client, called a *proxy* or a *stub*. The stub transparently packages the local procedure call information and transfers it to the server application that contains the procedure on the remote system. The server contains a comparable stub routine that interprets the request and invokes the procedure using standard local procedure call mechanisms.

Interface Definition Language

A set of client and server stubs that process a remote service constitutes a remote interface. An interface is defined using an Interface Definition Language (IDL). An IDL identifies the procedures that can be called through the interface and describes the data that are passed back and forth for each procedure.

Language and Platform Independence

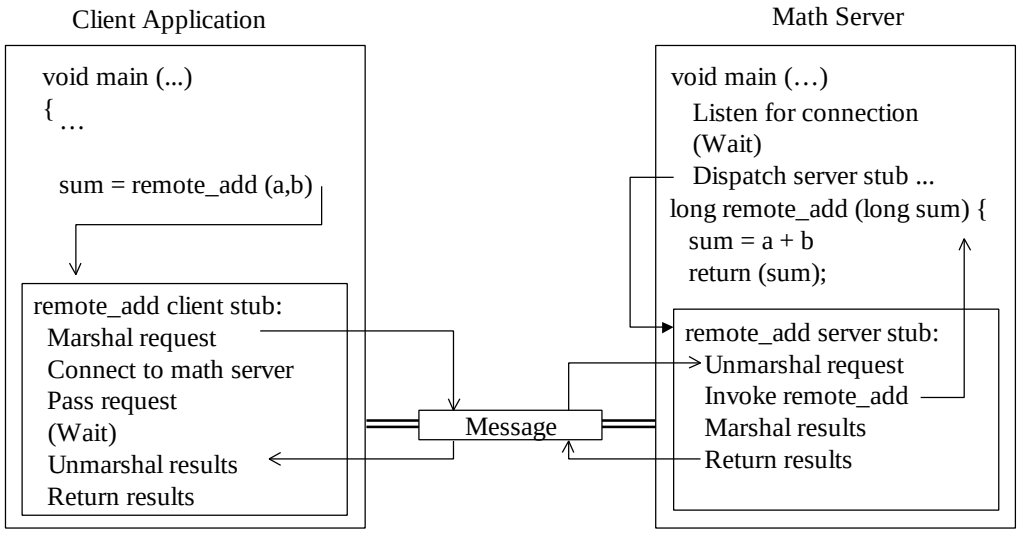
One of the problems associated with heterogeneous distributed computing is that every language and every platform represents data in a slightly different way. An IDL defines a set of canonical datatypes that are both language and platform independent. An RPC runtime system provides a marshaling service that automatically converts an application's native datatypes into the canonical datatypes for transport. The marshaling service ensures that the applications on both sides of the network communication can interpret the contents of any messages sent.

Stub Routine Services

All RPC communication code is generated

from the IDL definitions. An IDL compiler parses the IDL and generates the client and server stubs for the interface. The client and server stubs, along with a dispatcher in the server process, fully automate all remote-processing services. Illustration 1 shows the anatomy of an RPC call. The client application calls the client stub routine using a local procedure call, passing a parameter list. The client stub marshals the request and the parameter list into a message, it establishes a network connection with the remote server, and it transports the message to the server. The server receives the message and dispatches the request to the appropriate server stub. The server stub unmarshals the request, determines which procedure to invoke, and invokes the procedure

.




```
IDL:    interface math {
          long remote_add (
            [in]   long  a
            [in]   long  b
            [out]  long  sum  );
```

Illustration 1 - Anatomy of an RPC.

In an RPC, all remote processing is managed by the client and server stub routines that are generated from an IDL file. In this example, the math server provides a remote service called remote_add. A client specifies two numbers, and the remote_add procedure returns the sum of the two numbers. The client invokes the remote service using a simple local procedure call to the remote_add client stub. The remote_add client stub takes the request and marshals it into a message. Next it establishes a connection with the math server and passes the message to the dispatcher routine in the math server. The client application then waits for a response. The dispatcher identifies the interface being used and dispatches the remote_add server stub. The remote_add server stub unmarshals the request and invokes the remote_add procedure. The remote_add procedure returns the sum of the two numbers to the remote_add server stub, which marshals the results into a message and passes the message back to the remote_add client stub. The client stub unmarshals the results and returns them to the client application.

Remote Method Invocation

The interface stub mechanism is also used in distributed object systems to support remote method invocations. Both CORBA and COM use IDL to describe remote object interfaces, and IDL compilers generate proxy stub routines that transparently transport method invocation requests across the network. CORBA refers to the generated interface components as client stubs and server skeletons. A server *skeleton* provides a template for the server application in addition to the server communication routine.

Java RMI

Java also supports distributed object computing using a similar mechanism called Java remote method invocation (Java RMI).

Unlike CORBA and COM, Java RMI does not define its interfaces using IDL. IDL provides a mechanism to describe interfaces in a way that is not dependent on any specific programming language. Java RMI only supports remote communications between two Java applications; therefore language independence is not required. Java RMI simply defines interfaces in Java. But as with CORBA and COM, the Java interfaces are compiled using *rmic*, the RMI compiler, to generate proxy stub routines.

Middleware Differences

All IDL-based middleware accomplish effectively the same services. They transparently communicate requests from a client application to a server application and transfer the results back to the client. But although conceptually the systems are the same, the implementation details are different, as shown in Illustration 2. Each middleware system provides its own programming interface, uses its own IDL language, and supports its own set of datatypes. Each system uses its own marshaling service and its own wire protocol. Each system uses a different naming service to locate servers. Each system uses a different mechanism to dispatch requests to the server procedure or method. The distributed object systems also provide object management services that automatically create server object instances if they do not exist when a client issues a request. Even though middleware is designed to support integration, different IDL middleware systems do not interoperate.

Selecting IDL Middleware

In order to ensure universal interoperability, most organizations attempt to standardize on a single middleware solution for all application systems. But relying on a single middleware solution is a less than optimal solution. Each middleware system has its own strengths and weaknesses. For example, COM is easy to program, while CORBA offers excellent exception handling. The perfect middleware for one application might not be the best middleware for another application. Application systems are tightly bound to their middleware, causing significant vendor dependency. The middleware market is still young and competitive. No single middleware vendor has established dominance in the market, and it is not clear which middleware vendors will survive.

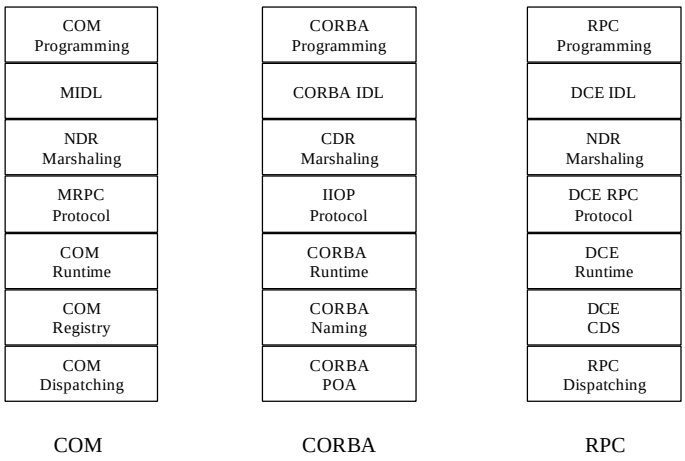


Illustration 2 - Middleware Silos.

Although each middleware system performs essentially the same functions, they each present a different programming interface, and they each use an incompatible set of middleware layers.

COM

Microsoft's COM will almost certainly survive since it is a core component of the Windows and NT operating systems. Microsoft has effectively recruited the desktop application and tools vendors to implement support for COM in most Windows and NT based products. Therefore COM provides the most natural interface to desktop-based visual application development tools. Recently COM has become available on a number of non-Windows operating systems, including Solaris, OpenVMS, and OS/390, but almost no

products or services are available that support COM. In addition, COM has almost no presence on the Internet.

CORBA

Object Management Group (OMG) CORBA is an industry standard cross-platform distributed object middleware architecture. CORBA implementations are available on numerous platforms from a number of vendors, including BEA Systems, Inprise (previously Borland International), Expersoft, ICL, IBM, Iona Technologies, and ObjectSpace. The future of the individual CORBA vendors is less certain than that of COM, as demonstrated by the recent acquisitions of Visigenic by Borland and Digital's ObjectBroker by BEA. The latest release of the specification, CORBA 2.2, defines a new dispatching mechanism, the Portable Object Adapter (POA). The POA enables CORBA objects to be easily ported across CORBA implementations, affording some vendor dependency protection. The POA combined with the Internet Inter-ORB Protocol (IIOP) ensures easy interoperability between CORBA implementations. Although the fates of individual CORBA vendors might be suspect, it seems likely that CORBA technology as a whole will survive. CORBA is the most widely deployed distributed object middleware on Unix and the Internet. CORBA is not as popular on Windows platforms, though, since it requires complex programming to make it work with visual development tools.

RPC

RPC systems have been used for years to build distributed client/server applications on Unix. Sun's public domain ONC RPC is available on most platforms, and development tools such as NobleNet RPC 3.0 have simplified distributed application development with ONC RPC. The industry standard Open Group DCE RPC is also available on most platforms. Microsoft has its own implementation of DCE RPC, the Microsoft RPC, which is integrated in Windows 95 and NT. Although RPC systems are very appropriate for C applications, they don't easily support object-oriented or scripting languages.

Combining the Best of Each System

Because each middleware system offers its own advantages and disadvantages, the optimal solution would involve all middleware systems, using each where it is most appropriate. COM should be used with visual development tools on Windows and NT. CORBA should be used with Java and C++ applications on Unix and the Internet. RPC should be used with C applications on any platform. The theory is nice, but the execution can be more challenging. Something is needed to enable the different systems to interoperate.

Middleware Interoperability

Bridging the Gap

Since Windows is often the default client environment, most CORBA and RPC vendors have made an effort to implement COM interoperability. The most popular approach used to implement COM interoperability is a protocol converter called a bridge. A *bridge* is an intermediary service that converts a distributed request from one middleware format to another.

COM/CORBA Internetworking

OMG has defined a standard specification for COM/CORBA Internetworking. According to the specification, a COM/CORBA bridge supports transparent bi-directional interoperability between any COM client or server and any CORBA client or server. From a COM client's point of view, a CORBA server looks like a COM server. From a CORBA client's point of view, a COM server looks like a CORBA server. A COM/CORBA bridge is implemented as a gateway server that supports both COM and CORBA interfaces. As Illustration 3 shows, each request goes through the gateway and gets transparently converted to the appropriate protocol.

Part A

The specification supports connectivity based on both IIOP and COM wire protocols. The COM/CORBA Internetworking Specification Part A defines connectivity using IIOP. This type of bridge is implemented as a combined COM and CORBA server running on a Windows platform. In many cases a different COM/CORBA bridge is generated for each CORBA server being accessed, and it is implemented as a DLL linked to the client application. The client makes a standard COM call to the COM/CORBA bridge, and the bridge converts the request into a CORBA call to the CORBA server. This approach requires that IIOP be installed on each client machine. Alternately, the bridge can be implemented as a standalone executable that is deployed on an NT server, and multiple clients can share the same bridge server. The clients make standard DCOM calls, and the COM/CORBA bridge converts the requests to CORBA calls. This approach requires IIOP only on the NT server. In the reverse process, CORBA clients can also use the bridge to access COM servers. The

bridge exposes a standard CORBA interface, so CORBA clients issue standard CORBA/IIOP requests to the bridge, which converts the requests into COM calls. Part A compliant COM/CORBA bridges are available from all CORBA vendors and from Visual Edge.

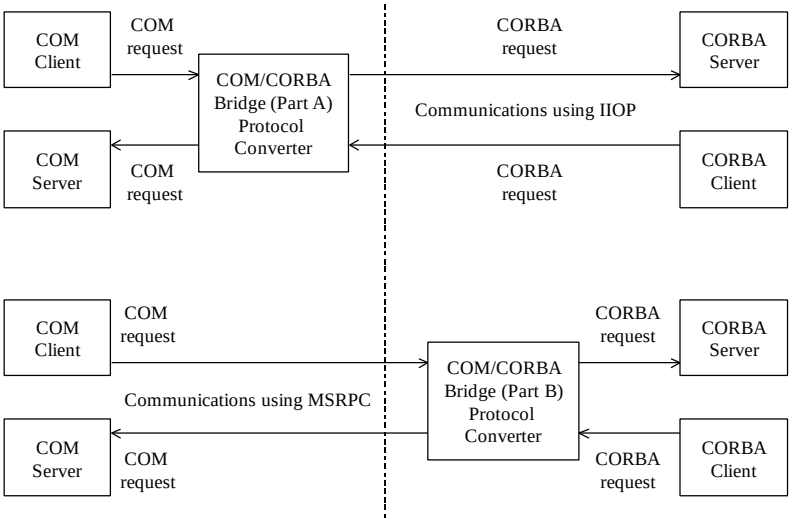


Illustration 3 - COM/CORBA Internetworking

COM/CORBA Internetworking uses a protocol converter to implement transparent connectivity between COM and CORBA systems. A COM/CORBA bridge supports both COM and CORBA interfaces. A COM client issues a COM request to the COM/CORBA bridge server, which converts the call into a CORBA request and reissues the call. Likewise, a CORBA client issues a CORBA request, and the COM/CORBA bridge converts the call into a standard COM request. The COM/CORBA bridge can be deployed on Windows 95, NT, Unix, or any other supported platform. The Part A specification defines communications based on the CORBA IIOP protocol. The Part B specification defines communications based on COM's distribution protocol, MSRPC.

Part B

The COM/CORBA Internetworking Specification Part B defines connectivity using COM. This type of bridge is implemented as a combined COM and CORBA server running on a Unix (or other) platform. In this case, a COM client makes a standard COM call to the COM bridge running on Unix. It converts the request into a CORBA request and calls the local CORBA runtime to dispatch the request. Likewise, a CORBA client can issue a standard CORBA request to the local COM/CORBA bridge, which converts the request to COM and reissues the call. This approach allows Windows clients to access Unix-based CORBA servers without installing IIOP on each desktop and without using an intermediary NT server box. A Part B compliant COM/CORBA bridge called COMet is available from Iona Technologies.

Overhead and Bottleneck

A COM/CORBA bridge solution provides connectivity between COM and CORBA, but it does not support the RPC environment. Similar bridges must be implemented to support connectivity between RPC and COM systems and between RPC and CORBA systems. More importantly, bridge systems impose a significant amount of overhead. For each call, the bridge unmarshals the request, converts and remarshals the datatypes, establishes a new connection, and reissues the call using a different wire protocol. In addition, when using dedicated bridge servers, the environment generates a potential bottleneck, since every request must be processed through the bridge servers

Naming

In addition to the differences in the runtime protocols, each system also provides a different mechanism to locate and get references to remote servers. In CORBA, each object is accessed through a unique object reference, and the reference is registered in a CORBA Object Service Naming service (COS Naming). COM objects are registered in the COM Registry. COM does not use unique

object references, but a COM Moniker can provide a unique name for an object. RPC servers may or may not use a naming service. In DCE, RPC servers are registered in the Cell Directory Service (CDS). To achieve transparent interoperability, developers must make sure that interoperable servers are registered in all appropriate naming services.

Direct Internetworking

A more efficient connectivity solution is one based on direct internetworking. NobleNet provides a middleware development and runtime environment that natively supports COM, CORBA, and RPC infrastructures. Rather than trying to bridge incompatible systems, NobleNet Nouveau implements middleware connectivity directly in the client and server communication stubs within the applications. No intermediary bridges are required, and Nouveau automatically registers the servers in all appropriate naming services.

NobleNet Nouveau

Development and Runtime System

NobleNet Nouveau is a model-independent middleware solution that supports the development and execution of COM, CORBA, and RPC applications. Nouveau consists of an IDL compiler development system and a CORBA 2.2 compliant runtime system. In addition, Nouveau provides an administrative tool to monitor and manage the deployment of application systems.

Multiple IDL Support

Nouveau addresses the interoperability issue at the IDL level. The Nouveau IDL compiler supports multiple IDL languages, including Microsoft IDL (MIDL), CORBA IDL, and RPC IDL. As seen in Illustration 4, the Nouveau IDL compiler can parse any IDL file and transform it into a model-independent internal data representation (IR). A set of code generators then generates client and server stub routines and makefiles for COM, CORBA, or RPC applications. The stub generators support C++ language bindings for COM and CORBA and C bindings for RPC. The IDL compiler can also generate standard CORBA IDL from any IDL input. Any CORBA-compliant IDL compiler can compile the CORBA IDL to generate CORBA bindings for Java, Smalltalk, or other languages.

Mix and Match Connectivity

The Nouveau environment enables complete mix and match connectivity. To allow a COM client to access an RPC server, the developer feeds RPC IDL in and requests a COM client interface and an RPC server interface out. To allow a CORBA client to access a COM server, the developer feeds MIDL in and requests a CORBA client interface and a COM server interface out. All interfaces support a common wire protocol (IIOP), but expose the native programming model.

Nouveau Infrastructure

The communication stubs generated by the Nouveau IDL compiler rely on a CORBA 2.2 compliant runtime infrastructure and the IIOP protocol. Nouveau applications run using the Nouveau runtime system. Even though all Nouveau applications use IIOP, Nouveau RPC applications act and behave like native RPC applications, and Nouveau COM applications act and behave like native COM applications. Nouveau provides a layer of abstraction between the application programming model and the object runtime.

Layered Services

The generated stubs rely on a set of layered services that ensure complete interoperability among the various programming models. Each layer is distinct, exposing a formal API. Individual layers can be replaced or supplemented to add support for additional programming models or alternate services. Illustration 5 shows the layered services used within the generated stub code. Nouveau IDL Compiler

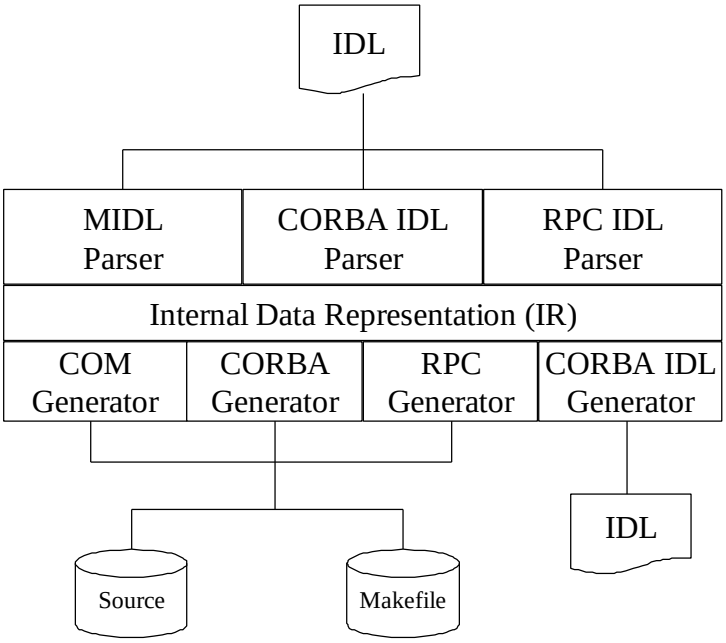


Illustration 4 - Nouveau IDL Compiler.

The Nouveau IDL compiler accepts MIDL, CORBA IDL, RPC IDL, or any combination of the three as input, and the parser converts the interface definitions into a model-independent internal data representation. Then, depending on the developer's specified options, the compiler uses the appropriate code generator to generate client and server stub source code and makefiles. Alternately, the compiler can generate a standard CORBA IDL file that in turn can be compiled by any CORBA IDL compiler.

Application Manager

The Application Manager is an optional service used to register application servers at deployment time and to dynamically start and stop application servers at runtime. Applications are registered in a naming service database.

Naming Service

The Nouveau Naming Service is a full implementation of the OMG CORBA Object Services Naming service (COS Naming) that can be used to locate or access object references. Nouveau can automatically generate COM monikers to allow COM applications to transparently interact with the COS Naming service. Any COS Naming service can be used in place of the Nouveau Naming service.

COM Runtime

COM clients interface with Nouveau application servers using standard COM runtime functions and the standard COM interface mechanism. Each Nouveau-generated COM interface exposes standard COM interfaces, including IUnknown, IMoniker, and IErrorInfo. The Nouveau COM runtime layer maps standard COM runtime functions such as CoCreateInstance and interface dispatch mechanisms to the lower-level Nouveau runtime services.

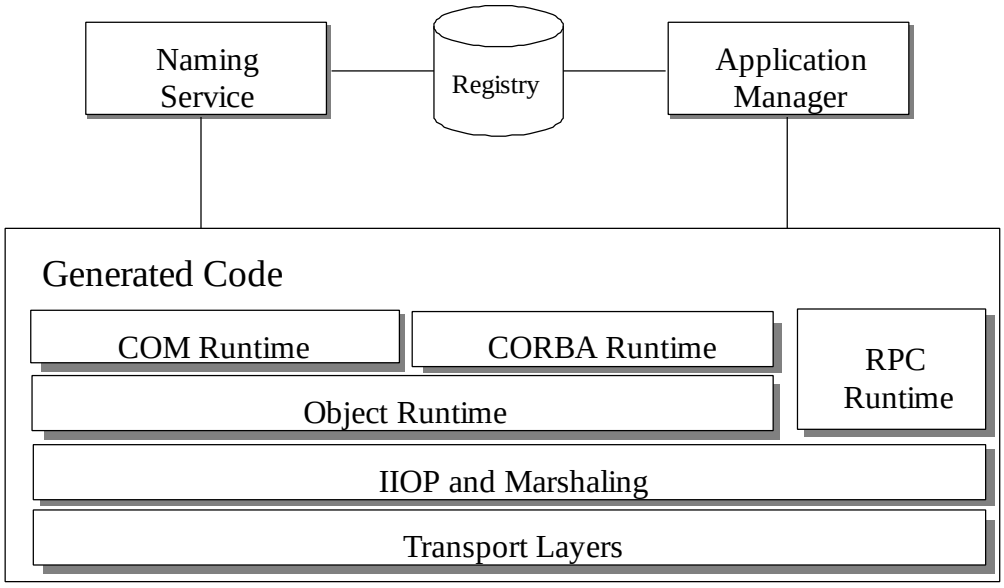


Illustration 5 - Layered Runtime Services.

The Nouveau runtime services are organized as a set of replaceable layered functions. The COM runtime, CORBA runtime, and RPC runtime layers supply a native programming interface to the developer that maps to the common runtime services. The object runtime layer provides core object management functions to the COM and CORBA runtime layers such as reference counting and interface navigation. The IIOP layer marshals requests into IIOP message formats. The transport layers manage network connections and transport IIOP messages. Nouveau supports TCP/IP and shared memory transports, although support for other network transports can be implemented if desired. The application manager registers applications and manages instances of running applications. Applications can locate other applications using an OMG COS Naming service.

CORBA Runtime

CORBA clients interface with Nouveau application servers using the standard CORBA object reference mechanism. Each Nouveau-generated CORBA object reference inherits from the standard CORBA::Object interface. The Nouveau CORBA runtime layer maps standard CORBA runtime functions such as lifecycle management and exception handling to the lower-level Nouveau runtime services.

Object Runtime

The Nouveau Object runtime layer performs object management services on behalf of the COM and CORBA runtime layers, including reference counting and interface navigation. The Object runtime layer includes an implementation of the OMG POA, which performs the object dispatching services for Nouveau CORBA servers. The Object runtime layer calls the IIOP runtime layer to prepare messages for transport.

RPC Runtime

RPC clients interface with Nouveau application servers using standard RPC invocation mechanisms. The Nouveau RPC runtime layer maps RPC runtime functions to the lower-level Nouveau runtime services. The RPC runtime layer calls the IIOP runtime layer to prepare messages for transport.

IIOP Runtime

The IIOP runtime layer is responsible for marshaling requests into IIOP message formats. Datatypes and object references are marshaled using OMG CDR marshaling services. The IIOP runtime layer calls the transport layers to relay the IIOP messages.

Transport Layers

The transport layers manage network connections and perform all message transports. The standard Nouveau environment supports TCP/IP and shared memory transport services, although additional transports can be implemented if desired.

Conclusions

Rather than cobbling together connections using bridges and gateways, Nouveau implements native interoperability directly in the applications.

Use the Right Tool for the Job

Nouveau removes the headaches associated with using multiple middleware systems. Organizations no longer need to be constrained by an arbitrary mandate to use one and only one middleware system. Instead an organization can use the most appropriate middleware system to suit the requirements of a particular application component. A single distributed application system can be implemented using different middleware services in different pieces of the application. New application systems can be integrated with existing application systems.

Native Interoperability

NobleNet Nouveau takes a new and direct approach to middleware interoperability

Preserving the Programming Model

Nouveau uses a common wire protocol, OMG IIOP, to provide direct connectivity between any COM, CORBA, or RPC client and server. Yet even though the low-level infrastructure is based on CORBA, Nouveau preserves the full capabilities of both the COM and the RPC programming model.

Less Overhead and Higher Reliability

By establishing direct connectivity between heterogeneous middleware systems, Nouveau reduces runtime overhead and removes an added layer of management complexity. Since bridge servers must be managed, monitored, and replicated to ensure the reliability and availability of the system, Nouveau's native connectivity will deliver higher performance and more reliable application systems than other connectivity solutions.

Anne Thomas works for the Patricia Seybold Group - see:www.psgroup.com. NobleNet inc. may be contacted on +1 (508) 229 4665. See also:www.noblenet.com for further information. Additional technical information and white papers can be obtained by emailing:ratio@noblenet.com or calling the above number. ***asking for Bill Bogaski***

ObjectArchitecture … … SubjectiveView

End-to-End Objects

The ODBMS Advantage

Kieron McCammon of Versant Object Technology argues the case for the OO Database, giving a thorough introduction along the way!

Introduction

The IT landscape faces a process of continual change, now more so than ever. Business demands of time to market, reduced costs, increased complexity and functionality all drive towards the single goal of being competitive in today's markets. From a technological standpoint these pressures have driven the adoption of object-oriented technologies, principles and approaches throughout the software development lifecycle. New acronyms like CORBA, new languages like Java and new approaches like component-based development are becoming the norm. So why would people turn to 20-year-old technology when it comes to selecting a database?

This article aims to investigate the roles for database technologies, focusing on how an Object Database Management System (ODBMS), like the one provided by Versant, can help an organisation to truly deliver end-to-end object solutions.

What's an Object Database?

Object Databases appeared on the horizon over ten years ago, back in 1988. The goal then was to deliver a new breed of database, designed and optimised to store and manipulate objects.

This goal was driven by the widespread adoption of object-oriented modelling techniques and languages. Design decisions made at this time differentiated object databases from existing relational databases. Primarily, instead of focusing on a data model (based on a fixed set of types) specified in some abstract, normalised form, the object database focused on the object model as defined within the O-O language.

The primary strength of an object database is its in-built ability to manage arbitrarily complex models (in terms of types) with arbitrarily complex relationships. Managing objects consisting of simple-valued attributes (integers, strings), multi-valued attributes (dynamic arrays of values) and complex structures is fundamental, but it's the ability to handle relationships that is key - not just one-to-one or one-to-many, but relationships that include semantics: like sets (uniqueness); lists (ordering); maps (associative lookup). These relationships may be complex objects in themselves, perhaps containing hashed values for efficient lookup and retrieval.

In the object world transactions involve navigating relationships and performing complex operations thereon. Object databases are optimised for this navigational access and the marshalling of objects between the database server and client. Figure 1 shows a typical ODBMS architecture.

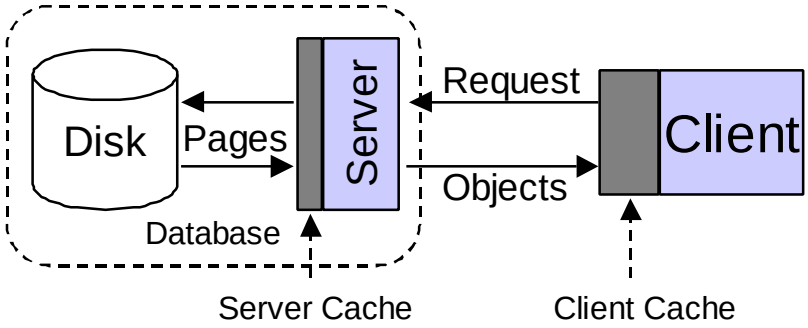


Figure 1 - Client-server architecture of an ODBMS

The server process provides concurrency and transactional control, ensuring recoverability (as with any database). The client-side cache manages the objects that have been transferred from the server, effecting transparent access through the programming language. As relationships are traversed, objects are requested from the server and instantiated in cache automatically. Once in cache they remain there until the transaction ends. When the client commits a transaction, any modified objects are transferred back to the server and the transaction ends.

So to answer the question:

"What is an object database?"

Simply put, it's a database for objects

But why can't I use a relational database?

Still not convinced? Well, lets look in detail at how you might store objects in a relational database. Figure 2 shows a simple object model that we will use as an example (notice that it doesn't include any real complexity, but it should serve to prove a point).

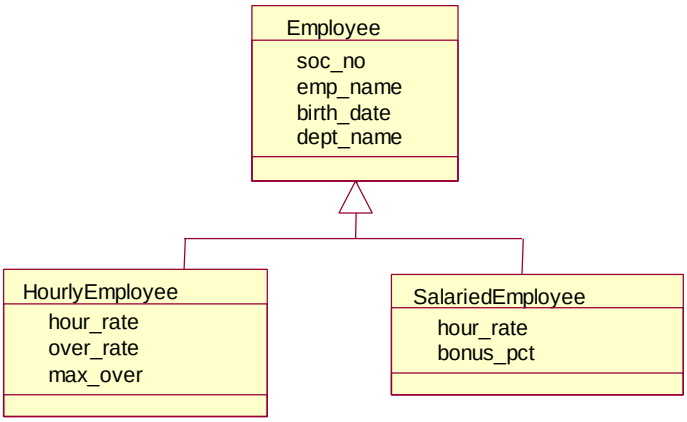


Figure 2 - Simple object model

There are a number of ways in which to map this model to a relational database. Each approach has its trades off. One way is to define a table for each class, where each column in the table represents an attribute of the class (fine for simple-valued attributes, requires more work and additional tables for multi-valued and structured attrbitutes, and don't even think about multi-media types). Figure 3 shows the likely schema.

Employee_Table

soc_no	emp_name	birth_date	dept_name

HourlyEmployee_Table

soc_no	hour_rate	over_rate	max_over

SalariedEmployee_Table

soc_no	month_rate	bonus_pct

Figure 3 - Relational schema

The attributes common to all Employees are stored in the Employee table, additional attributes for HourlyEmployees are stored in the HourlyEmployee table and so on. We may refer to this method of object-to-relational mapping as "Inheritance by Join". The other approach, analogous to the relational notion of denormalizeation, is "Inheritance by Copy". In this style, the attributes of the parent class are copied into each of its children.

Once the required schema has been defined the mapping code has to be written. This code is required to take an object, as created and manipulated in the programming language and de-construct it into the representation required by the database and conversley construct an object from the database representation so it can be used by the programming language. Figure 4 shows the SQL code required just to perform the mapping from an object to the relational schema (the same will be required to map from the relational schema to an object).

```
EXEC SQL INSERT INTO Employee_Table
   (soc_no, emp_name, birth_date, dept_name)
   VALUES
      (:emp->soc_no, :emp->name,
       :emp->birth_date, :emp->dept_name);
if (emp->type == 1)
   EXEC SQL INSERT INTO HourlyEmployee_Table
      (soc_no, hour_rate, over_rate, max_over)
   VALUE
      (:emp->soc_no, :emp->hour_rate,
       :emp->over_rate, :emp->max_over);
else if (emp->type == 2)
   EXEC SQL INSERT INTO SalariedEmployee_Table
      (soc_no, month_rate, bonus_pct)
   VALUES
      (:emp->soc_no,
       :emp->month_rate, :emp->bonus_pct)
EXEC SQL COMMIT WORK RELEASE;
```

Figure 4 - Mapping an object to the relational database

The Employee attributes are inserted into the Employee table, if the object is type "1" (some means to identify the class of the object), the additional attributes are inserted into the HourlyEmployee table and so on.
Compare this to the code in figure 5, which shows what is required by an object database to create and store an object.

```
HourlyEmployee* employee = new(db, "HourlyEmployee")
    HourlyEmployee("Versant","1/1/88","Object Technologies");
```

Figure 5 - Creating an object in an object database

A very graphic comparison of the effort involved. You can see why it is generally recognised in the industry that anything upto 35-40% of such an application's code is involved in overcomming this mapping of objects to and from a relational database, (often referred to as overcomming the "impedance mismatch").

And of course this is only a small part of the overall story: Retrieving an HourlyEmployee from the database involves joining the HourlyEmployee and Employee tables. Every time you retrieve an object of any of the child classes (e.g., HourlyEmployee), you will have to join several tables. Had we selected the "Inheritance by Copy" style, the retrieval of HourlyEmployee would avoid the inheritance joins, but that introduces other complications. Using "Inheritance by Copy", an application must instead join every time it wishes to access objects via a parent class (in this instance, Employee). Which of these two approaches is least expensive for a given application depends upon data access patterns and the object As complexity increases the effort and cost to develop and maintain this mapping layer increases exponentially.

model, but even mildly complex objects may require a 5-way join, across potentially large tables (as the number of objects increases).

None of the above costs take into account mapping complex relationships (sets, lists, maps), this can take the number of joins into double figures, and also introduce sorting. So even though the objects have been mapped into the relational database it isn't possible to get the required performance as the object model is navigated, (this of course leads to simplification of the object model and its relationships, to reduce the number of joins, thus negating the benefit of using an object approach in the first place)

Once the mapping layer has been implemented it has to be maintained. More importantly it has to be able to evolve as the application evolves. How would you add a PartTimeEmployee class? What code would need to change? What impact would this have on other parts of the system?

So How does an object database

work?

Back to basics

Firstly some basics, an important concept to understand is that of object identity. In the object world every object can be said to have an identity, this identity is orthogonal its state (the values of its attribute). Object identity is fundamental since it is the means by which objects are manipulated; object identity is used to build relationships between objects, and by navigation to determine which object to access next.

Whilst object identity is fundamental in the object world, it doesn't exist in the relational world, here data is accessed based on its value (using keys). Take a Circle object, should its colour change from "Yellow" to "Red", its identity remains the same. In the relational world it would no longer be possible to find the row that had previously contained the value "Yellow". This can (and indeed should) be addressed in a relational database design with unique keys not derived from any application attributes, but in a relational database these values are still potentially changeable by the application.

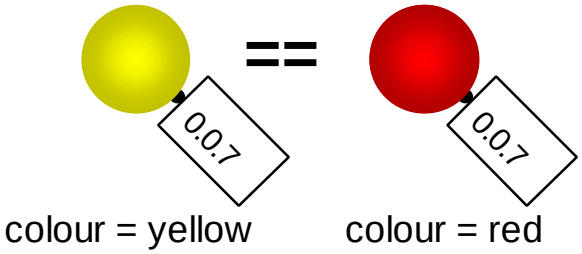


Figure 6 - Object Identity

Object database architectures

Object identity is likewise fundamental to an object database. Figure 7 shows in more detail the client-side architecture of an object database.

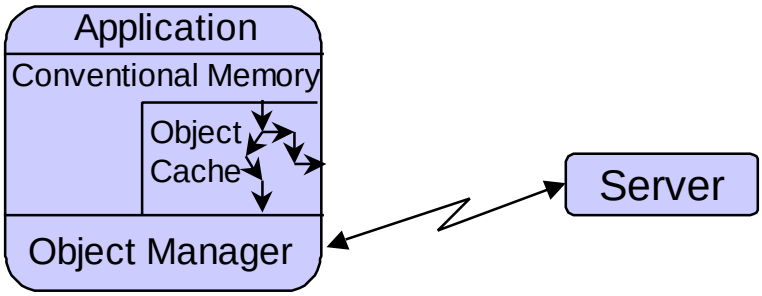


Figure 7 - Client-side architecture of an object database

The application is linked with the Object Manager (OM) provided by the database vendor, this provides transparent navigation (based on object identity) and management of persistent objects, fetching them on demand from the server into the client's object space. The application calls the OM APIs to manage the transactional boundaries; on calling commit (or rollback to discard changes) the OM sends the changed objects back to the server and ends the transaction. The database server enforces transactional integrity and isolation between multiple clients, using locking to ensure "cache-coherency" between the objects held in the client-cache and those in the server. As expected with any database, the server provides transactional recovery on failure and ensures objects held in the database are transactionally consistent.

Whilst the various object databases available in the market today all provide the same basic capability they have tended to approach the problem from different perspectives. Most focused on providing persistent extensions to the language, C++ or Smalltalk, and are often categorised as persistent storage engines. Others focused on providing a database for

objects. The difference can be exemplified by looking at how queries are processed.

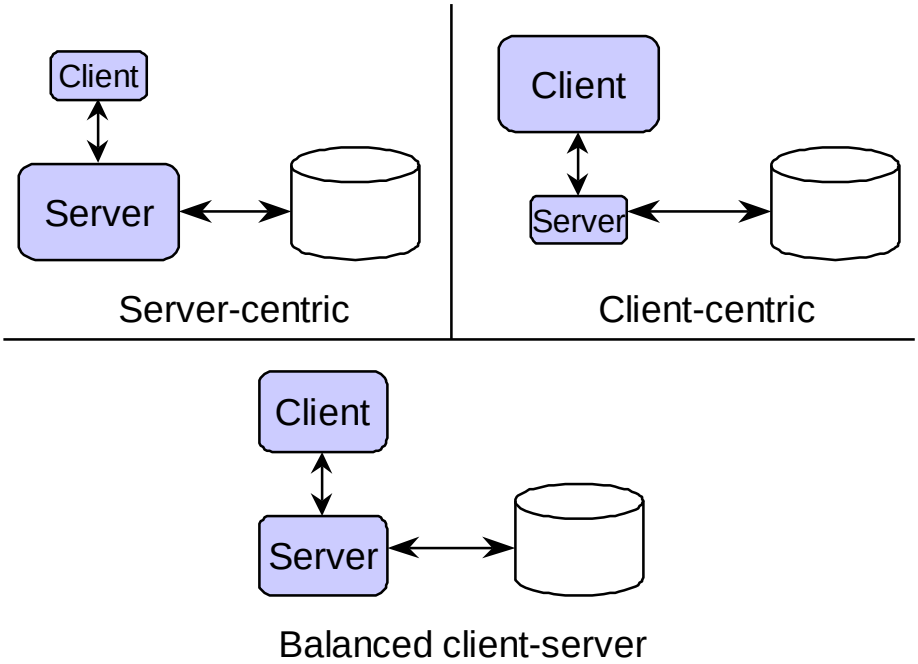


Figure 8 - Different database architectures

The persistent storage engines turned the server-centric approach of the relational database on its head, adopting instead a client-centric approach. Rather than off-loading the data processing to the database server (as when using SQL), they choose to simplify the database server so it is just responsible for managing pages (as in pages of memory). It sends these pages to clients on request (this type of database server is often referred to as a "page server"). The client then performs the required processing. To perform a query all the required objects must be transferred from the database server to the client, the client then determines which objects satisfy the specified criteria.

The second approach is to design a database for objects instead of for undifferentiated blocks of data.

This architecture allows three major benefits: Object-aware processing can occur at both the client and the server; objects can be transported as objects rather than as blocks of data; and Objects can be transported between differing platforms more easily.

The first benefit means that since the database is object-aware, the server can perform processing on those objects. It can perform navigation, queries, maintain inter-object relationships, and other complex functions. To perform a query for example, the client simply specifies the criteria using an SQL-like syntax and passes this to the server. The server optimises and executes the query, using multi-user indices if available, returning just the resultant objects to the client.

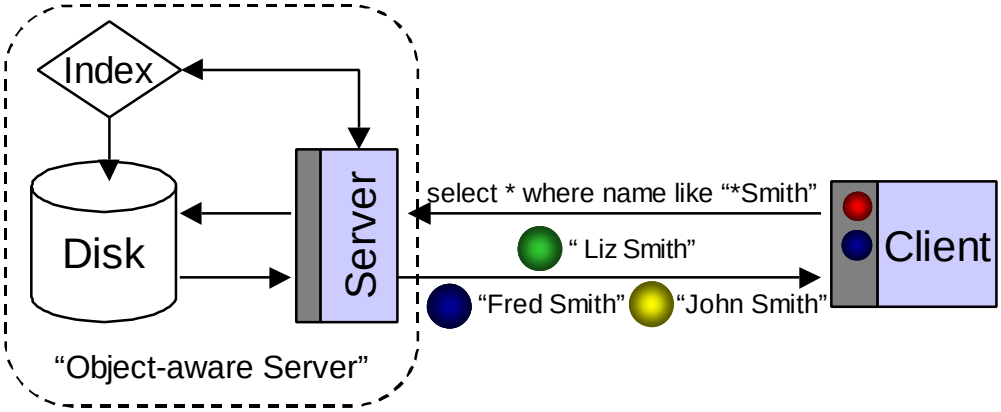


Figure 9 - Query processing

The second benefit of a database for objects is that the server and client can exchange and cache objects as objects. This reduces overhead in both transport and caching, and allows for much faster performance.

The third major benefit of the "object-aware server" is the ability to easily share objects between different platforms, compilers and languages. Because the database understands and stores objects, not pages of memory, NT clients can talk to Solaris servers, Java applications can manipulate objects created by C++ applications and vice-versa. All issues of heterogeneity are taken care of by the database, an important concept to help future-proof today's applications.

Who wants to discover they can't access their objects from a new machine/compiler/language because its memory layout is incompatible with that stored in the database, (32bit addressing verses 64bit addressing, for example)? Or to increase the pain of cross-platform development?

Developing an application

The Object Data Management Group (ODMG) has defined standard language bindings for C++, Smalltalk and Java to an object database (see:**Error! Reference source not found.** for more information). The standard defines:

- How an object model is defined and captured in the database (Object Definition Language). The standard allows for a separate ODL, but generally C++, Smalltalk or Java is used as the definition language.
- How the application interacts with the Object Manager to manipulate the objects (Object Manipulation Language). This defines the language interface for C++, Smalltalk and Java.
- How the application can query the database (Object Query Language). Each language interface provides means of allowing OQL queries to be expressed and executed. Only object databases with "object-aware servers" are able pass the query to the server to be executed. Others use the client to execute the query, transferring all the objects from the database and checking for a match.

As an example lets look at a simple, ODMG C++ application. Figure 10 shows the UML model for the application.

/

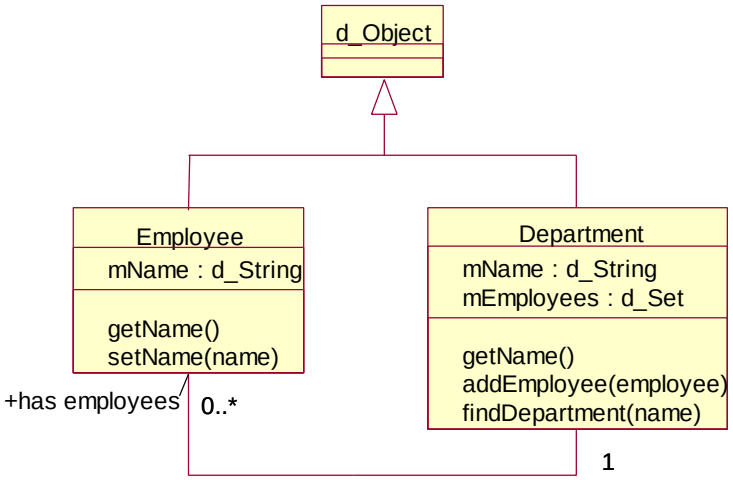


Figure 10 - UML for example

It consists of two classes. An `Employee` class and a `Department` class, each has a name. The `Department` class has a one-to-many relationship to `Employee` (denoting that a department contains many employees).

First lets declare the `Employee` class:

```
class Employee : public d_Object
{
public:
// Constructors/Destructors
       Employee ( const char* name ) : mName(name) {}
       virtual ~Employee () {}

// Accessor Methods
       const char* getName ( ) const { return mName; }

// Mutator Methods
       void setName ( const char* name )
       {
              mark_modified();
              mName = name;
       }

private:
// Attributes
d_String mName;
}; // class Employee
```

Figure 11 - Employee class

`Employee` inherits from the ODMG persistence base-class, `d_Object`. This means the class is now persistent capable (transient or persistent instances of `Employee` can be created). The class has a single attribute, `mName`, which uses the ODMG `d_String` string class rather than a `const char*` (since memory pointers cannot be stored in the database). The mutator method, used to set an employees name, first calls `mark_modified()` to tell the Object Manager that the object is going to be changed (depending on the database, this may result in a write lock being granted to ensure no one else tries to modify the same object).

If we now look at the `Department` class:

```
class Department : public d_Object
{
public:
// Constructors/Destructors
Department ( const char* name ) : mName(name) {}
virtual ~Department ( ) {}

// Accessor Methods
       const char* getName ( ) { return mName; }

// Mutator Methods
void addEmployee ( Employee* employee )
{
       mark_modified();
       mEmployees.add(employee);
}

// Static Methods
static d_Ref<Department> findDepartment ( const char* name);

private:
// Attributes
       d_String                mName;
       d_Set<d_Ref<Employee> > mEmployees;
}; // class Department
```

Figure 12 - Department class

Again the `Department` class inherits from `d_object` and has a name attribute. Each `Department` has a set of `Employees`, this relationship is defined using an ODMG `d_Set<>` (`d_Set<>` is an unordered collection class) of ODMG references, `d_Ref<>`, to `Employee`. The mutator method to add an employee to a department calls `mark_modified()`to signify its intention to update itself and adds the given employee to its set of employees. The static method `findDepartment()` returns a reference to a `Department`. The `d_Ref<>` class replaces the usual use of C++ pointers. C++ pointers are not safe when manipulating persistent objects, a pointer to an object is only valid while it remains in memory. Also pointers are limited by the addressing model of the operating system. The implementation of `findDepartment()` will be looked at later.

Once the database schema has been captured from the class definitions and loaded into the database, persistent objects can be created using the `new` operator:

```
Employee* emp1 =
new Employee("Mickey");                    // Transient instance

Employee* emp2 =
new(db, "Employee") Employee("Donald");  // Persistent instance
```

Figure 13 - Creating an object

The ODMG overloaded `new` operator takes two additional parameters: the database in which the object is to be created; and the class of the object. Otherwise it functions as the standard `new` operator. To create a transient instance of a persistent capable class just call the standard `new` operator.

Figure 14 shows a simple application that opens a database, starts a transaction, creates a department and some employees and commits:

```
main()
{
       d_Database    db;
       d_Transaction txn;

       db.open("Staff.db");   // Open a database connection
       txn.begin();           // Start a transaction

       // Create a persistent Department and some Employees
       Department* dep  = new(db, "Department") Department("RD“);
       Employee*   emp1 = new(db, "Employee")   Employee("Mickey");
       Employee*   emp2 = new(db, "Employee")   Employee("Donald");

       // Add the Employees to the Department
       dep->addEmployee(emp1);
       dep->addEmployee(emp2);

       txn.commit();          // Commit the transaction
       db.close();            // Close the database connection
}
```

Figure 14 - A simple application

The ODMG `d_Database` and `d_Transaction` classes interface to the Object Manager and are used to control database connections and transaction boundaries. The rest is standard C++ code. Finally, taking a look at the implementation of `findDeparment()` demonstrates the use of queries:

```
d_Ref<Department> Department::findDepartment ( const char *name )
{
       d_Set<d_Ref<Department> > results;   // Results of query
       d_VQL_Query               query      // Query object
          ("select SelfOID from Department where name like $1");

       query << name;                       // Bind name to $1
       d_oql_exectute(query, results);      // Execute the query

       if (results.cardinality() > 1)
       {
              // Opps, found more than one department
              throw InvalidDepartmentException(name);
       }
       else if (results.cardinality() == 1)
       {
              // Found a single match, return the first element in set
              return *results.begin();
       }
       else
       {
              // No match found, return NULL
              return NULL;
       }
}
```

Figure 15 - Performing a query

The query object is built using the SQL-like select syntax, `SelfOID` denotes that the result is a collection of object references of matching objects (rather than a projection of attributes from the matched objects). `d_oql_execute()` executes the query.

Conclusion

As can be seen, building a C++ application that interfaces to an object database is straightforward. The ODMG language binding hides all the complexity of storing and retrieving objects. Eliminating the complex mapping required to use a relational database results in less code to design, write, debug, test and maintain. This can lead to significantly shorter development time scales, one of the fabled promises of object-orientation.

By using the same object representation in the database as that manipulated by the application, providing direct support for complex relationships and navigation, an object database can deliver significant performance gains over a relational database.

Using an object database that has a "balanced client-server" architecture ensures maximum flexibility in application design, allowing complex navigation to be performed by the client and ad-hoc queries by the server, delivering the best performance in both cases, and supporting cross-platform communication.

But what's the impact on my development process?

Getting the design right

The use of an object database is more pervasive in the development process than a relational database. Thought must be given to issues of concurrency, performance and scalability during analysis and design (rather than during deployment, by relying on the DBA to optimise the data model and define appropriate indexing strategies). This is not because the object database itself requires it, but because concurrency, performance and scalability are real-world requirements that need to be reflected throughout analysis, design and into implementation. The use of a database (object or otherwise) is often driven by requirements related to concurrency, performance and scalability.

Since part of the strength of an object database comes from its ability to support complex relationships, allowing clients to navigate and perform complex manipulations, it is essential to ensure that the object model includes the navigational paths dictated by the system's transactions. This is often seen as a drawback of using an object database, the emphasis on getting the class model right. But is it any less true if using a relational database? It may be argued that a DBA can tune a data model; de-normalise tables, define indexes, build join tables, and so on to achieve performance and concurrency requirements. And in a purely data driven application this may be true, but few applications are purely data-driven. It was shown earlier that the impact on the mapping code when changing the database schema is potentially catastrophic and indexes or join tables can only be created where common keys already exist. If a particular access path wasn't envisaged then neither relational or object database can help you. There is no exception to the rule:

"There is no substitute for good design"

Due in large part to the much greater cost of fixing problems in the implementation or maintenance phases, today's object-oriented methodologies place a great importance on getting the design right. They place emphasis on the consideration of "use cases" or "scenarios" that help identify interactions (transactions) within a system and bring to the fore, thoughts of performance and concurrency. Object databases are a natural part of this approach. The role of the ODBA should be more proactive. Instead of becoming involved only at the end of the project to tune and manage the data, the ODBA has more of an architectural role, and works as part of the analysis, design and implementation teams. The ODBA helps to consider issues of concurrency, performance and scalability; ensuring that transactions are designed making best use the features of the chosen technology. And since the ODBA is taking a more proactive role within the project, why not introduce the concept of a class librarian, facilitating reuse by providing a repository for class libraries and components.

Understanding access patterns

Access patterns are the ways in which a system accesses and manipulates its objects. Consideration of access patterns helps to identify the navigational paths required by a system's transactions. The performance requirements of these transactions will in turn sway implementation decisions, ensuring that required relationships are present or supporting classes available.

A persistent storage engine places great importance on defining relationships, since navigation is the only means of access. If an object can't be reached via navigation then it is garbage (which is why some technologies

advocate garbage collection within the database). This client-centric approach places a great emphasis on maintaining collections of objects to support the required access patterns. Taking a look at the earlier example, to support the `findDepartment()` method all `Department` objects would have to be inserted into a collection of departments (the collection would likely be a map, associating a department name to a department very efficiently).

This is an obvious requirement, but its impact on concurrency may not be. Each time a department is created it must be inserted into the collection of departments. Not a problem for an application using an in-memory model, but it can act as a potential concurrency "hot-spot" otherwise; since only one transaction can create a department at any time (due to the need to update the department collection). This may not be evident during initial prototyping and isn't pleasant to discover during deployment as the number of users, levels of concurrency or transaction rates increase. And whilst the access path is available to navigate from a department to its employees, what for a latter requirement to find all employees with a given name?

This would have to be performed by retrieving each department to the client and navigating to each employee, checking for a match, one by one; unless this was foreseen during design and a collection of employees added (in which case a query over the collection would suffice). But how would either approach scale to manage millions of objects.

Compare this approach to that of using an object database with an "object-aware server". The class declarations remain the same, but since the server automatically maintains a collection of all objects in a given class, referred to as a class extent, it is not necessary to maintain a department collection. `findDepartment()` would be implemented using the query capability of the database. Likewise for a latter requirement to find an employee, rather than navigating to every employee via a department, a query could be used (with an index if available) over the `Employee` class extent. Of course finding the employees of a department is still a simple means of navigation, unlike a relational database that would require a query. With this approach it is possible to balance concurrency, performance and scalability requirements. Navigation has the potential to deliver the best performance at the expense of having to maintain the required relationships. Querying, whilst not as fast, offers a greater degree of concurrency (no need for collections) and is more scalable, being able to manage millions of objects and deliver near constant performance through the use of indexes.

The elimination of potential concurrency "hot-spots" when using a balanced client-server approach makes it much more suited to multi-user, highly concurrent database applications. And since every object is reachable via a query, there is no need for garbage collection to reclaim space.

Developing transactions

Every database client needs to execute one or more transactions. For those not familiar with developing database applications deciding on what constitutes a transaction can be one of the toughest problem. A transaction should consist of a logical unit of work that is either done in its entirety or not done at all. An object transaction will typically start by creating an object, navigating from an ODMG root object (simply an object which has been given a name) or performing a query. Thereafter anything can happen. A transaction finishes when the application issues a commit or rollback.

When developing transactions a few simply rules apply:

Keep transactions short.

Thought should be given to the impact of holding locks for extended periods of time in multi-user systems. For GUI applications, an optimistic locking model may be more appropriate (where locks aren't held, instead timestamps ensure multiple updates don't occur).

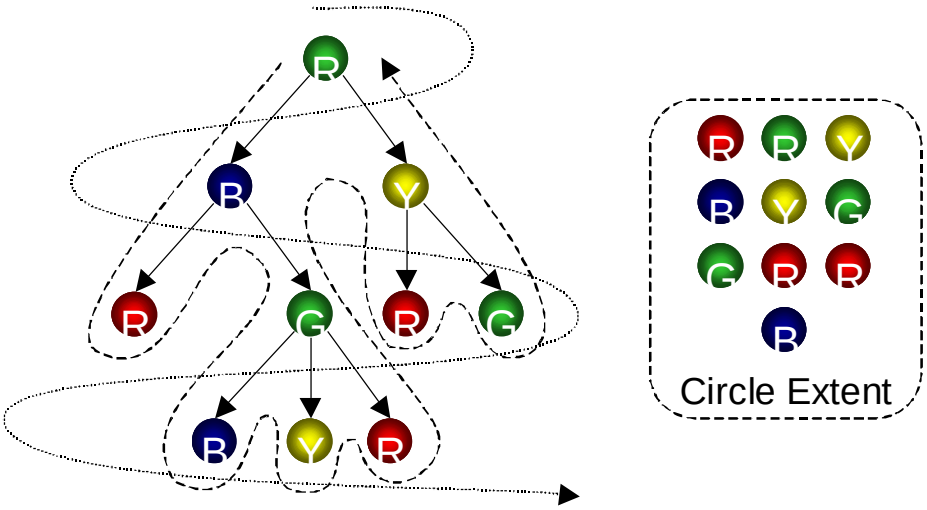
Use navigation.

Navigation should be the primary means of interaction with the database.

Judiciously use queries.

For object databases with "object-aware" servers this can help maintain near-constant performance as numbers of objects increase by minimising the objects transferred to the client and eliminating "hot-spots".

The ability to perform queries adds a forth dimension to the way an object application would otherwise be developed. As an example, figure 16 shows a directed acyclic graph of circles.



select * from Circle where colour = “red”

Figure 16 - A DAG

Most transactions would navigate the tree-structure, depth or breadth first, making the most of the object database's ability to handle complex structures. And where a tree walk is required navigation is the optimal approach, offering performance far superior to that achievable with a comparable relational solution. But for searches, navigation may not prove the most efficient means of solving the problem, especially as the graph grows into the millions (as the number of objects doubles so does the time to navigate them). Instead a query over all instances of `Circle` (its class extent) to identify those matching the specified criteria will return in near-constant time, irrespective of the number of objects (when used in conjunction with an index).

Conclusion

The use of an object database provides a less segregated approach to object-oriented development than using a relational database. The emphasis of current methodologies on understanding the real-world requirements of a system dovetails neatly with the use of an object database. The lesson from objects, as with other technologies is that it's important to get the design right, no technology can overcome poor design. The argument for keeping the database (and data definition) segregated from the application to ensure that it can be optimised, independent of the application, is a fallacy. Mapping objects to tables results in convoluted relational schemas and large amounts of mapping code.

Key issues to be addressed revolve around concurrency, performance and scalability requirements. These will invariably impact class design and implementation, with consideration being given to the capabilities offered by the database itself. Persistent storage engines may prove sufficient for single-user applications that require persistence (embedded systems, CAD/CAM packages). But only an object database with the capability and flexibility to be a database for objects allows multiple applications, with different transaction types and access patterns, to each optimise their use of the database to deliver maximum performance and concurrency.

And how do I manage a deployed system?

Deploying an object database involves similar considerations to deploying any database. It consists of a number of data volumes stored on disk, that need to be backed up and managed. Hardware considerations like disk speed, number of CPUs and speed, I/O bandwidth will all affect the behaviour of the database. Understanding the database architecture helps in extrapolating likely cause and effects. Availability and recoverability requirements dictate how a database is managed and which policies and procedures are put in place. Therefore the skills normally equated with the DBA are still a necessary part of deploying an object database, but the focus of the DBA does change.

The role of the ODBA

The role of the database administrator in the object database world is different than it is in the relational world. The task of optimisation happens earlier in the process, with the ODBA providing architectural input to help get the design right, instead of after the fact. Therefore some tasks disappear (for example, building join tables). The class model identifies access patterns by virtue of its relationships; therefore much of the optimisation is implicit in this approach. The development team, including the ODBA in an expanded role, has optimised the transactions to make the best use of available features to ensure the performance, scalability and concurrency requirements are met. There is little design repair work for the ODBA to do here after the fact.

Day-to-day management of the database is, of course, still necessary. Backup strategies need to be defined and implemented, additional volumes need to be added, high availability solutions need to be put in place, and as the applications evolve the database schema needs to evolve with it. These are all tasks that still fall into the realm of the DBA. This pure administrative role is perhaps 20% of the workload associated with a typically RDBMS DBA. This shift in workload presents an opportunity for the DBA to become more active within the overall development process, participating in class and transaction design, sharing knowledge regarding the impact of using a database.

Administration Tasks

Each object database offers different levels of administrative support depending upon the underlying capabilities of their architecture. One aspect to consider is whether the tasks can be performed on-line (particularly important if backing up, adding extra data volumes or evolving schema). If tasks require the database to be shutdown it will significantly effect how the database can be deployed. The following list identifies typical administrative tasks:

Creating and deleting a database

In a development environment developers usually undertake this. In a deployment environment databases should be created and managed by the ODBA.

Starting and stopping a database

It is useful if the database starts automatically upon first connection, undergoing automatic recovery if required, to help reduce administrative intervention. An option to stop the database after a specified time of inactivity also helps.

Backing up and recovering a database

Backup should be an on-line task; the database should be available during a backup. Support for incremental backups helps reduce backup times particularly for large databases. The option to archive transaction records since the last backup is useful in facilitating full recovery. A backup is restored and the archived transaction records replayed.

Adding additional data volumes to a database

Adding additional space to a database should be an on-line task; where a database consists of many data volumes, each potentially on different physical devices for reasons of performance or reliability.

Defining clustering strategies

It should be possible to create logical partitions, each consisting of several physical volumes, and be able to specify which classes are stored in which partitions to support physical optimisation of storage if required.

Reorganising a database

A database should support automatic, on-line space reclamation and reuse to ensure optimal performance over time. An option to undergo off-line reorganisation is useful to optimise internal structures periodically.

Adding and removing indexes

Adding and removing indexes should be an on-line task performed at any time after database creation (for those databases that support server-side querying).

Evolving the schema of a database

Evolving the database schema should be an on-line task allowing the addition of a new class or removal of an existing one, addition, renaming or removal of attributes of a class. Support for "lazy" schema evolution allows the schema to be evolved without changing the actual objects themselves, especially important for large databases. Objects are evolved into the new representation as accessed, over time. Schema evolution, by virtue of its potential to break applications, should be controlled by the ODBA.

Granting access to a database

Granting access rights to a database should be on-line. As a minimum it should be possible to specify user-level on a read or read/write

basis.

Browsing and querying a database

It should be possible to browse the database schema and objects and perform ad-hoc queries (where supported). An option to insert, update, delete via a browser is useful but can break the encapsulation of the class model since an object's attributes are manipulated directly (however as a developer/DBA tool its invaluable to help fix potential problem areas).

Monitoring and tuning a database

It should be possible to monitor the health of the database as well as the overall performance to aid tuning.

Configuring fault tolerance

Not all object databases offer a fault tolerant solution, but for those that do it should be the responsibility of the DBA to configure and manage. Fault tolerance should be transparent

to the application; not needing changes to the application code.

Coexistence

An object database deployed in isolation or embedded as part of an application has little or no requirement to coexist with other technologies. Often however the reality is that the object database must be part of an overall infrastructure. Particularly if this infrastructure consists of existing reporting or data access tools. It is important that these tools can access the information stored within the object database to satisfy the user's needs.

Most object databases vendors can provide an ODBC interface to their database. This should automatically map the class model to a relational model and present this through an ODBC interface allowing ODBC-compliant tools to create, update and delete information in the database, thus preserving the investment in these tools.

Conclusion

Deploying an object database is like deploying any database, it requires consideration of the hardware and database capabilities, along with the requirements for availability and recoverability. Typically object databases simplify the deployment process by automating many of the administrative tasks and allowing most to be performed on-line.

Conclusion (of conclusions!)

Object databases are an integral part of an object end-to-end approach. They address the shortfalls of existing database technologies by managing objects as objects, with all their inherent complexities, allowing the developer to concentrate on developing the application and business logic rather than focusing on overcoming the "impedance mismatch" between object and relational models.

As part of an encompassing O-O approach, with the emphasis on getting the design right, they can address issues of:

- Time to market
- Ease of use
- Performance
- Flexibility
- Scalability
- Concurrency

But consideration must be given to choosing the most appropriate product for the job.

Object database's have different architectures and different capabilities. A database with an "object-aware server" offers the highest level of concurrency, scalability and flexibility, all pre-requisites for it to be considered part of an infrastructure supporting a suite of evolving applications. Importantly the object database should be a database first and foremost. So what are the advantages of an ODBMS? Well that's up to you!

Kieron McCammon is a Technical Manager with Versant Object Technology Ltd. Kieron may be contacted at 'kmccammon@versant.com' or on +44 (0) 1753 701013

OMG Analysis

By Eric Leach, OMG's UK Representative

OMG Meeting, Orlando, Florida

At OMG's Technical Committee Meeting (TCM) in Orlando, Florida, USA, June 8-12, members completed work on a new messaging service for linking CORBA to Message Oriented Middleware. The messaging specification will be the first international standard for messaging. Specifications for security (Firewall), realtime systems (Minimum CORBA), telecommunications (CORBA/TMN interworking), DCE/CORBA interworking and a Persistent State service were also discussed and voted on at the meeting. At the meeting IBM, Unisys and Oracle proposed an industry standard for streamlining collaborative application development efforts on the Web.

The standard, known as Metadata Interchange Format specification was created in response to developers' needs for standardised methods of sharing data, regardless of tool or programming language, in collaborative development environments. The XMI specification aims to integrate Extensible Modelling Language and Meta Object Facility to be the cornerstone of an open information interchange model.

OMG Meeting, Helsinki, Finland

At the OMG TCM in Helsinki, Finland, July 27-31, the OMG Board of Directors approved the Currency Facility specification, making it the first OMG financial domain standard. It was also announced at the meeting that the International Accounting Standards Committee (IASC) had joined the OMG. The other substantive specifications adopted at the Helsinki meeting were Objects by Value, Printing Facility, IDL to Java Mapping, Java to IDL Specification, Lexicon, Query Services, PDM Enabler, Notification and Patient Identifiers (PIDS). OMG's next TCM is in Seattle, WA, USA, hosted by Boeing, September 14-18. See:www.omg.org for details.

Evaluating the Business Potential Of CORBA in Telecoms

OMG's Richard Soley was the keynote speaker at IIR's "Evaluating The Business Potential Of CORBA In The Telecommunications Industry" held at The Cumberland Hotel, London, UK on 15-17 June. 80 of the 100 delegates were from mainland Europe and all of them were adopting or had adopted CORBA implementations.

CORBA applications discussed and showcased included those for Telecom operators, Internet and multi-media services, Legacy integration, TMN and GIS. The high degree of OMG interest in and interworking with other related standards groups and consortia was very apparent. The groups and standards include The Network Management Forum, CMIP/OMIP, GDMO, ISO, ITU-IS, ETSI, DAVIC and TINA-C. "The Telecoms industry has managed to create standard interfaces, even though the underlying technologies and de facto and de jure standards are assembled together in a somewhat messy fashion. But it does work. OMG aims to do the same in the IT industry," said Richard Soley in his keynote speech.

Ovum tips CORBA IDL for Stardom

In May, Ovum (www.ovum.com) published its long-awaited "Componentware: Building It, Buying It, Selling It" report. The report identified OMG CORBA IDL as a future (2001) principal component interface standard, only rivalled by Microsoft's COM IDL. Ovum sees CORBA IDL effectively encapsulating Sun's JavaBeans because of the latter's language and platform dependencies. The report is descriptive, but not judgmental, when it comes to evaluating component execution platforms. It describes offerings from IBM, Microsoft, Oracle and Sybase, but advises IS Managers and CIOs to "court a trusted supplier – preferably one that can provide tools, infrastructure, service and applications to support your component execution platform."

Ovum defines a software component as "…a

unit of software that:

- implements some known function
- hides the implementation of that function behind one or more unambiguous "interfaces" that it exposes to its environment"

Ovum coined the term Componentware to describe software assembled from a set of components.

CORBA Desktop Tools

On the CORBA desktop tools front, Inprise (formerly Borland, see: www.inprise.com) demonstrated the beta version of its Delphi 4.0 Client/Server Suite at PC Expo in the USA in June. Delphi 4.0, which shipped in July, supports CORBA, HTTP, RPC, Sockets and DCOM. Dephi 4.0's tight integration with Inprise's own CORBA ORB, Visibroker, offers an attractive alternative to VB 6.0 on the desktop for those whose world extends beyond the Microsoft closed world of Windows.

In June, BEA Systems (www.beasys.com) completed its acquisition of Top End from NCR. BEA is committed to integrate Top End with its own Tuxedo software. BEA is compiling a technology cocktail from third party ingredients it has acquired in recent years. These include ObjectBroker and DECmessageQ originally developed by DEC.

Object Design … Object Design …Object Design …Object Design

The Open-Closed Principle

How can software be extended without being modified?
Robert C. Martin tells all!

There are many heuristics associated with object oriented design. For example, "all member variables should be private", or "global variables should be avoided", or "using run time type identification (RTTI) is dangerous". What is the source of these heuristics? What makes them true? Are they *always* true? This article investigates the design princi-ple that underlies these heuristics -- the open-closed principle. As Ivar Jacobson said: "All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version."

How can we create designs that are stable in the face of change and that will last longer than the first version? Bertrand Meyer gave us guidance as long ago as 1988 when he coined the now famous open-closed principle.

To paraphrase him:
"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION."

When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with "bad" design. The program becomes fragile, rigid, unpredictable and unreusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that *never change*. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Description

Modules that conform to the open-closed principle have two primary attributes.

Description

1. They are "Open For Extension". This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the applica-tion change, or to

meet the needs of new applications.
2. They are "Closed for Modification". The source code of such a module is inviolate. No one is allowed to make source code changes to it. It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

Abstraction is the Key.

In C++, using the principles of object oriented design, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and the unbounded group of possible behaviors is repre-sented by all the possible derivative classes. It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction. **Figure 1** shows a simple design that does not conform to the open-closed principle. Both the `Client` and `Server` classes are concrete. There is no guarantee that the member functions of the `Server` class are virtual. The `Client` class *uses* the `Server` class. If we wish for a `Client` object to use a different server object, then the `Client` class must be changed to name the new server class.

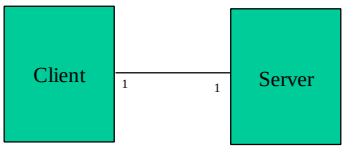


Figure 1
Closed Client

Figure 2 shows the corresponding design that conforms to the open-closed principle. In this case, the `AbstractServer` class is an abstract class with pure-virtual member functions. the `Client` class uses this abstraction. However objects of the `Client` class will be using objects of the derivative `Server` class. If we want `Client` objects to use a different server class, then a new derivative of the `AbstractServer` class can be cre-ated. The `Client` class can remain unchanged.

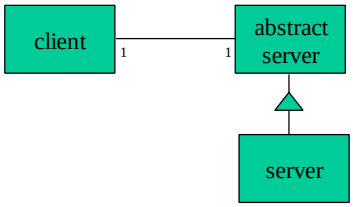


Figure 2
Open Client

The Shape Abstraction

Consider the following example. We have an application that must be able to draw circles and squares on a standard GUI. The circles and squares must be drawn in a particular order. A list of the circles and squares will be created in the appropriate order and the pro-gram must walk the list in that order and draw

each circle or square. In C, using procedural techniques that do not conform to the open-closed principle, we might solve this problem as shown in Listing 1. Here we see a set of data structures that have the same first element, but are different beyond that. The first element of each is a type code that identifies the data structure as either a circle or a square. The function `DrawAllShapes` walks an array of pointers to these data structures, examining the type code and then calling the appropriate function (either `DrawCircle` or `DrawSquare`).

Listing 1

Procedural Solution to the Square/Circle Problem

```
enum ShapeType {circle, square};
struct Shape
{
  ShapeType itsType;
};
struct Circle
{
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
struct Square
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
//
// These functions are implemented
// elsewhere
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);
typedef struct Shape *ShapePointer;

void DrawAllShapes
  (ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
      case square:
      DrawSquare((struct
                  Square*)s);
      break;
      case circle:
      DrawCircle((struct
                  Circle*)s);
      break;
   }
  }
}
```

The function `DrawAllShapes` does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function to be able to draw a list of shapes that included triangles, I would have to modify the func-tion. In fact, I would have to modify the function for any new type of shape that I needed to draw. Of course this program is only a simple example. In real life the `switch` statement in the `DrawAllShapes` function would be repeated over and over again in various func-tions all over the application; each one doing something a little different.

Adding a new shape to such an application means hunting for every place that such `switch` statements (or `if/else` chains) exist, and adding the new shape to each. Moreover, it is very unlikely that all the `switch` statements and `if/else` chains would be as nicely struc-tured as the one in `DrawAllShapes`. It is much more likely that the predicates of the `if` statements would be combined with logical operators, or that the `case` clauses of the `switch` statements would be combined so as to "simplify" the local decision making.

Thus the problem of finding and understanding all the places where the new shape needs to be added can be non-trivial. Listing 2 shows the code for a solution to the square/circle problem that conforms tothe open-closed principle. In this case an abstract `Shape` class is created. This abstract class has a single pure-virtual function called `Draw`. Both `Circle` and `Square` are derivatives of the `Shape` class.

Listing 2

OOD solution to Square/Circle problem.

```
class Shape
{
  public:
  virtual void Draw() const = 0;
};
class Square : public Shape
{
  public:
  virtual void Draw() const;
};
class Circle : public Shape
{
  public:
  virtual void Draw() const;
};

void
DrawAllShapes(Set<Shape*>&list)
{
  for (Iterator<Shape*>i(list); i;
       i++)
     (*i)->Draw();
}
```

Strategic Closure

It should be clear that no significant program can be 100% closed. For example, consider what would happen to the `DrawAllShapes` function from Listing 2 if we decided that all `Circles` should be drawn before any `Squares`. The `DrawAllShapes` function is not closed against a change like this. In general, no matter how "closed" a module is, there will always be some kind of change against which it is not closed. Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. He then makes sure that the open-closed principle is invoked for the most probable changes.

Using Abstraction to Gain Explicit Closure.

How could we close the `DrawAllShapes` function against changes in the ordering of drawing? Remember that closure is based upon abstraction. Thus, in order to close `DrawAllShapes` against ordering, we need some kind of "ordering abstraction". The specific case of ordering above had to do with drawing certain types of shapes before other types of shapes. An ordering policy implies that, given any two objects, it is possible to discover which ought to be drawn first. Thus, we can define a method of `Shape` named `Precedes` that takes another `Shape` as an argument and returns a `bool` result. The result is `true` if the `Shape` object that receives the message should be ordered before the `Shape` object passed as the argument.

In C++ this function could be represented by an overloaded `operator<` function. Listing 3 shows what the `Shape` class might look like with the ordering methods in place.

Now that we have a way to determine the relative ordering of two `Shape` objects, we can sort them and then draw them in order. Listing 4 shows the C++ code that does this. This code uses the `Set`, `OrderedSet` and `Iterator` classes from the `Components` category developed in my book (if you would like a free copy of the source code of the `Components` category, send email to `rmartin@oma.com`). This gives us a means for ordering Shape objects, and for drawing them in the appropriate order. But we still do not have a decent ordering abstraction. As it stands, the individual `Shape` objects will have to override the `Precedes` method in order to specify ordering. How would this work? What kind of code would we write in `Circle::Pre-cedes` to ensure that `Circles` were drawn before `Squares`? Consider Listing 5.

Listing 3

Shape with ordering methods.

```
class Shape
{
  public:
  virtual void Draw() const = 0;
  virtual bool Precedes(const
                  Shape&) const = 0;
  Bool operator<(const Shape& s)
  {return Precedes(s);}
};
```

Listing 4

DrawAllShapes with Ordering

```
void DrawAllShapes(Set<Shape*>&
                     list)
{
  // copy elements into OrderedSet
  // and then sort.
  OrderedSet<Shape*> orderedList =
                       list;
  orderedList.Sort();
  for
  (Iterator<Shape*>i(orderedList);
   i; i++)
    (*i)->Draw();
}
```

Listing 5

Ordering a Circle

```
Bool Circle::Precedes(const Shape&
                        s) const
{
  if (dynamic_cast<Square*>(s))
    return true;
  else
    return false;
}
```

It should be very clear that this function does not conform to the open-closed principle. There is no way to close it against new derivatives of `Shape`. Every time a new derivative of `Shape` is created, this function will need to be changed.

Using a "Data Driven" Approach to Achieve Closure.

Closure of the derivatives of `Shape` can be achieved by using a table driven approach that does not force changes in every derived class. Listing 6 shows one possibility. By taking this approach we have successfully closed the

DrawAllShapes function against ordering issues in general and each of the Shape derivatives against the creation of new Shape derivatives or a change in policy that reorders the Shape objects by their type. (e.g. Changing the ordering so that Squares are drawn first.)

Listing 6

Table driven type ordering mechanism

```
#include <typeinfo.h>
#include <string.h>
enum {false, true};
typedef int bool;
class Shape
{
  public:
  virtual void Draw() const = 0;
  virtual bool Precedes(const
                      Shape&) const;
  bool operator<(const Shape& s)
                               const
  {return Precedes(s);}

  private:
  static char* typeOrderTable[];
};

char* Shape::typeOrderTable[] =
{
  "Circle",
  "Square",
  0
};

// This function searches a table
// for the class names.
// The table defines the order in
// which the
// shapes are to be drawn. Shapes
// that are not
// found always precede shapes that
// are found.
//
bool Shape::Precedes(const Shape&
                           s) const
{
  const char* thisType =
     typeid(*this).name();
  const char* argType =
     typeid(s).name();
  bool done = false;
  int thisOrd = -1;
  int argOrd = -1;

  for (int i=0; !done; i++)
  {
    const char* tableEntry =
      typeOrderTable[i];

    if (tableEntry != 0)
    {
      if (strcmp(tableEntry,
                 thisType) == 0)
        thisOrd = i;
      if (strcmp(tableEntry,
                 argType) == 0)
        argOrd = i;
      if ((argOrd>0)&& (thisOrd>0))
        done = true;
    }
    else // table entry == 0
      done = true;
  }
  return thisOrd < argOrd;
}
```

The only item that is not closed against the order of the various Shapes is the table itself. And that table can be placed in its own module, separate from all the other modules, so that changes to it do not affect any of the other modules.

Extending Closure Even Further.

This isn't the end of the story. We have managed to close the Shape hierarchy, and the DrawAllShapes function against ordering that is dependent upon the type of the shape. However, the Shape derivatives are not closed against ordering policies that have nothing to do with shape types. It seems likely that we will want to order the drawing of shapes according to some higher level structure. A complete exploration of these issues is beyond the scope of this article; however the ambitious reader might consider how to address this issue using an abstract OrderedObject class contained by the class OrderedShape, which is derived from both Shape and OrderedObject.

Heuristics and Conventions

As mentioned at the begining of this article, the open-closed principle is the root motivation behind many of the heuristics and conventions that have been published regarding OOD over the years. Here are some of the more important of them.

Make all Member Variables Private.

This is one of the most commonly held of all the conventions of OOD. Member variables of classes should be known only to the methods of the class that defines them. Member variables should never be known to any other class, including derived classes. Thus they should be declared private, rather than public or protected. In light of the open-closed principle, the reason for this convention ought to be clear. When the member variables of a class change, every function that depends upon those variables must be changed. Thus, no function that

depends upon a variable can be closed with respect to that variable.

In OOD, we expect that the methods of a class are not closed to changes in the member variables of that class. However we *do* expect that any other class, including sub-classes *are closed* against changes to those variables. We have a name for this expectation, we call it: *encapsulation*. Now, what if you had a member variable that you knew would never change? Is there any reason to make it `private`? For example, Listing 7 shows a class `Device` that has a `bool status` variable. This variable contains the status of the last operation. If that operation succeeded, then `status` will be `true`; otherwise it will be `false`.

Listing 7

```
non-const public variable
class Device
{
  public:
  bool status;
};
```

We know that the type or meaning of this variable is never going to change. So why not make it `public` and let client code simply examine its contents? If this variable really never changes, and if all other clients obey the rules and only query the contents of `status`, then the fact that the variable is `public` does no harm at all. However, consider what happens if even one client takes advantage of the writable nature of `status`, and changes its value. Suddenly, this one client could affect every other client of `Device`. This means that it is impossible to close any client of `Device` against changes to this one misbehaving module. This is probably far too big a risk to take. On the other hand, suppose we have the `Time` class as shown in Listing 8. What is the harm done by the public member variables in this class? Certainly they are very unlikely to change. Moreover, it does not matter if any of the client modules make changes to the variables, the variables are supposed to be changed by clients. It is also very unlikely that a derived class might want to trap the setting of a particular member variable. So is any harm done?

Listing 8

```
class Time
{
  public:
  int hours, minutes, seconds;
  Time& operator-=(int seconds);
  Time& operator+=(int seconds);
  bool operator< (const Time&);
  bool operator> (const Time&);
  bool operator==(const Time&);
  bool operator!=(const Time&);
};
```

One complaint I could make about Listing 8 is that the modification of the time is not atomic. That is, a client can change the `minutes` variable without changing the `hours` variable. This may result in inconsistent values for a `Time` object. I would prefer it if there were a single function to set the time that took three arguments, thus making the setting of the time atomic. But this is a very weak argument. It would not be hard to think of other conditions for which the `public` nature of these variables causes some problems. In the long run, however, there is no *overriding* reason to make these variables `private`. I still consider it bad *style* to make them `public`, but it is probably not bad *design*. I consider it bad style because it is very cheap to create the appropriate inline member functions; and the cheap cost is almost certainly worth the protection against the slight risk that issues of closure will crop up. Thus, in those rare cases where the open-closed principle is not violated, the proscrip-tion of `public` and `protected` variables depends more upon style than on substance.

No Global Variables -- Ever.

The argument against global variables is similar to the argument against `pubic` member variables. No module that depends upon a global variable can be closed against any other module that might write to that variable. Any module that uses the variable in a way that the other modules don't expect, will break those other modules. It is too risky to have many modules be subject to the whim of

one badly behaved one. On the other hand, in cases where a global variable has very few dependents, or can-not be used in an inconsistent way, they do little harm. The designer must assess how much closure is sacrificed to a global and determine if the convenience offered by the glo-bal is worth the cost. Again, there are issues of style that come into play.

The alternatives to using globals are usually very inexpensive. In those cases it is bad style to use a technique that risks even a tiny amount of closure over one that does not carry such a risk. However, there are cases where the convenience of a global is significant. The

global variables `cout` and `cin` are common examples. In such cases, if the open-closed principle is not violated, then the convenience may be worth the style violation.

RTTI is Dangerous.

Another very common proscription is the one against `dynamic_cast`. It is often claimed that `dynamic_cast`, or any form of run time type identification (RTTI) is intrinsically dangerous and should be avoided. The case that is often cited is similar to Listing 9 which clearly violates the open-closed principle. However Listing 10 shows a similar program that uses `dynamic_cast`, but does not violate the open-closed principle. The difference between these two is that the first, Listing 9, *must* be changed when-ever a new type of `Shape` is derived. (Not to mention that it is just downright silly). How-ever, nothing changes in Listing 10 when a new derivative of `Shape` is created. Thus, Listing 10 does not violate the open-closed principle. As a general rule of thumb, if a use of RTTI does not violate the open-closed principle, it is safe.

Listing 9

RTTI violating the open-closed principle.

```
class Shape {};
class Square : public Shape
{
  private:
  Point itsTopLeft;
  double itsSide;
  friend DrawSquare(Square*);
};

class Circle : public Shape
{
  private:
  Point itsCenter;
  double itsRadius;
  friend DrawCircle(Circle*);
};

void DrawAllShapes(Set<Shape*>& ss)
{
  for (Iterator<Shape*>i(ss);i;i++)
  {
    Circle* c =
    dynamic_cast<Circle*>(*i);
    Square* s =
      dynamic_cast<Square*>(*i);
    if (c)
      DrawCircle(c);
    else if (s)
      DrawSquare(s);
  }
}
```

Listing 10

RTTI that does not violate the open-closed Principle.

```
class Shape
{
public:
virtual void Draw() cont = 0;
};
class Square : public Shape
{
  // as expected.
};

void DrawSquaresOnly(Set<Shape*>
                                &ss)
{
  for (Iterator<Shape*>i(ss); i;
                                i++)
  {
    Square*
      s=dynamic_cast<Square*>(*i);
    if (s)
      s->Draw();
  }
}
```

Conclusion

There is much more that could be said about the open-closed principle. In many ways this principle is at the heart of object oriented design. Conformance to this principle is what yeilds the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability. Yet conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change. This article is an extremely condensed version of a chapter from my new book: *The Principles and Patterns of OOD*, to be published soon by Prentice Hall.

Reprinted with permission from C++ Report

This article is an extremely condensed version of a chapter from Robert Martin' s new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall.