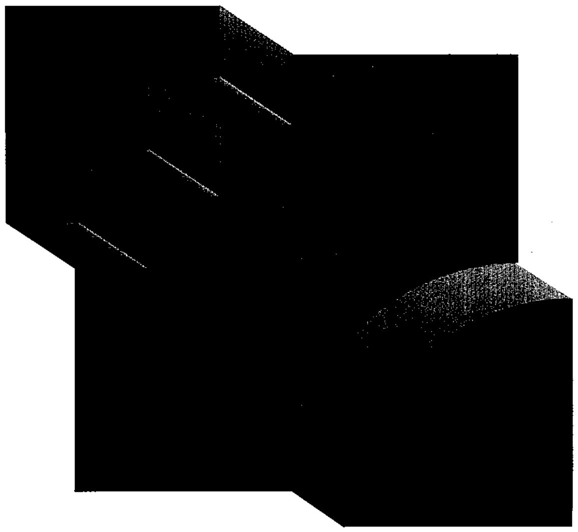
Thomas J. Mowbray • Ron Zahavi

THE ESSENTIAL CORBA

Systems Integration Using Distributed Objects



OMG®

OBJECT MANAGEMENT GROUP

The Essential CORBA: Systems Integration Using Distributed Objects

The Essential CORBA: Systems Integration Using Distributed Objects

Thomas J. Mowbray, PhD

Ron Zahavi

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Katherine Schowalter

Editor: Robert Elliott

Managing Editor: Micheline Frederick

Text Design & Composition: Integre Technical Publishing Co., Inc.

Figures 6.1–6.4 are reprinted courtesy of Microsoft Press from *Inside OLE2* by Kraig Brockschmidt, © 1994, Microsoft Press, Redmond, WA.

Figure 9.1 appears courtesy of IEEE. IEEE Micro, Vol. 13, No. 6 *Camp Development: The Art of Building a Market Through Standards* by Chris Halliwell, 1993.

ISBN 0-471-10611-9

Printed in the United States of America

10 9 8 7 6

This book is dedicated to our
lovely wives and lifelong friends:
Susan Zahavi and Kate Mowbray, C.P.A.

Let us raise a standard to which the honest and brave will repair.
—George Washington

Preface

Both of us have had a number of systems integration experiences before we started working together on a project called DISCUS (Data Interchange and Synergistic Collateral Usage Study) at The MITRE Corporation. We had the good fortune of working for a program management team that gave us the freedom to draw on our past experiences and develop a well thought out approach to a challenging problem: general application interoperability.

These ideas were based on the need for a carefully designed, well-documented software architecture, one that was designed with simplicity, extensibility, and cost implications in mind. We discovered through talking to our colleagues that these ideas, which seemed obvious to us, were not part of their training or backgrounds. In fact, these approaches are not adequately covered in the literature and not generally understood, even by experienced computer scientists. To transfer this knowledge, we put together this book, including a full coverage of the required background material, design approach, and implementation techniques.

Our approach builds on the standards foundations created by the Object Management Group (OMG). Our technical goals and the OMG goals are very much in sync. The OMG has made truly remarkable progress in generating quality standards and cultivating acceptance for its standards across industry, standards groups, and consortia. We want to see this vision realized even faster; hence, this book transfers our knowledge and approach for successful systems integration.

T.J.M.
R.Z.
McLean, Virginia
January 1995

Acknowledgments

Although this book was written on our own time, we must thank our project management for their visionary support, in particular: Webster Anderson, Dwight Brown, Steve Brown, Tim Daniel, Jeff Fleisher, Dolly Greenwood, Diane Haberstich, Gene Jarboe, Geoff Lipsey, John Polger, John Robusto, and Bill Ruh.

We sincerely appreciate the help and contributions provided by our friends and colleagues, including: Howard Cohen, Donna Cornwell, Jonathan Doughty, Ray Emami, Julie Gravalesse, Tom Herron, Michael Josephs, Melony Katz, Paul Klinker, Dr. Fred Kuhl, Raphael Malveau, Emanuel Mamatas, John Marsh, Diane Mularz, Chibuike Nwaeze, Craig Prall, Bill Quigley, Andy Reho, Jeff Rogers, Hank Seebeck, Lisa Strader, Shel Sutton, John Tisaranni, Doug Vandermade, Sheryl Veazy-Rudy, and Kendall White.

The Object Management Group has been essential to the publication of this work, thanks to Cheryl Bissonnette, Bill Hoffman, Kelly Kassa, Jon Siegel, John Slitz, Dr. Richard Soley, Geoff Speare, Chris Stone, and Lydia Thomas.

Special thanks to Steve Black, Mark Bramhall, David Chappell, Reggie Counts, Joe Croghan, John Eaton, Norman Eko, Alan Ewald, Charlie Green, Jim Green, Jed Harris, Jack Hassall, Ellis Horowitz, Andrew Hutt, Ole Jacobsen, Neil Jacobson, Huet Landry, Eric Leach, John Leary, Marie Lenzi, Jacob Levy, Geoff Lewis, Denise Lynch, Cliff Mauton, Todd Pelfrey, Marty Polluconi, Michael Powell, Richard Probst, Yllona Richardson, Bob Rockwell, Mark Roy, Mark Ryland, John Rymer, Dave Seres, Cathy Sloan, Craig Thompson, Fred Waskhewicz, Andrew Watson, Dave Zenie, and all the others who helped us in our quest for understanding. These people gave us inspiration, motivation, and important ideas.

Finally, we wish to thank our wives and families for their patience, love, and unconditional support while writing this book.

Contents

Executive Summary

In this book, we introduce the theory and practice of systems integration using standard object technology, Common Object Request Broker Architecture (CORBA). The purpose of systems integration is to provide interoperability between software components and to provide for system adaptability as the system evolves. This book goes beyond the basics of systems integration and CORBA to show readers how to maximize the benefits of these technologies and practices throughout the system life cycle.

PROBLEM DEFINITION

Today's software systems comprise islands of automation. Each software component performs some limited range of useful functions, but components do not interoperate effectively. The integration that exists is insufficient and does not evolve gracefully with the component technologies. The scale of this problem ranges from the the user's desktop, to interuser, interdepartment, and interorganizational levels. Key consequences of poor systems integration include:

- *Stovepipe systems.* These systems are poorly integrated systems that have ad hoc or proprietary integration solutions. Poorly integrated systems have high maintenance costs, resist adaptation to changing user requirements, and cannot evolve with commercial technology.
- *Organizational productivity impacts.* Poor integration leads to substantial organizational inefficiencies, such as redundant data entry, multistep data conversion, and ad hoc file transfers. These processes are

costly, time consuming, and error prone. Poor systems integration requires users to be knowledgeable about multiple disparate applications. Users trained on one system cannot transfer skills to other systems.

CORBA – THE BASIS FOR A SOLUTION

CORBA is an industry standard technology infrastructure for systems integration. Some of its key benefits include:

1. CORBA simplifies distributed computing and application integration. It is easier to use than other distributed computing and integration mechanisms; therefore, it saves time and reduces software costs. CORBA also provides many useful new flexibilities compared to other mechanisms.
2. CORBA is object oriented. This means that architects and developers can improve their software structure, make their software more flexible, reuse software, seamlessly integrate legacy software, and develop new capabilities rapidly. CORBA provides a uniform layer encapsulating other forms of distributed computing and integration mechanisms.
3. CORBA is a stable technology supported by a growing coalition of commercial software vendors. CORBA is a standard from the world's largest software consortium, the Object Management Group (OMG), and supported by X/Open, the Open Software Foundation (OSF), the Common Open Software Environment (COSE), CI Labs, X/Consortium, and others. The core elements of CORBA have been stable since 1991. With CORBA, OMG has reengineered the standards process to allow suppliers and end users to create new standards for interoperability efficiently.

SYSTEMS INTEGRATION USING CORBA – THE OPPORTUNITY

CORBA is an enabling technology. Bad systems integration practices can lead to stovepipe systems regardless of whether CORBA is used. This book is a guide on how to follow good architecture principles that utilize CORBA for maximum benefits.

Key benefits of our approach to CORBA-based systems integration include: faster system delivery, enhanced software reuse, increased system adaptability, reduced maintenance costs, focused research activities, enhanced system interoperability, enterprise systems integration, and other benefits.

A FRAMEWORK FOR CHANGE

In Chapter 4 we define an integration capability model, which enables organizations to assess their current systems integration practices. This model allows organizations to discover the substantial benefits of improved systems integration compared to current practices. The integration capability model includes the following levels:

Level 6. Standard Architectures
Level 5. Frameworks
Level 4. Distributed Objects (CORBA)
Level 3. Mature Remote Procedure Calls (OSF DCE)
Level 2. Miscellaneous Mechanisms
Level 1. Commercial Off-the-Shelf Solutions

The capabilities range from organizations that do no value-added integration (level 1) to organizations that drive industry standards (level 6). Most organizations today practice systems integration at levels 2 and 3. When an organization first adopts CORBA, it begins at level 4. Substantial benefits result from level 5 practices, and technology leadership is demonstrated at level 6. This book teaches readers how to perform systems integration in order to achieve level 5 and 6 benefits. The capability model is described in detail in Chapter 4.

SYNOPSIS OF THE BOOK

Chapter 1, "Introduction," introduces the paradigm shift that motivates the increasing role of systems integration. It provides a detailed introduction to systems integration problems including commercial technology integration, legacy systems integration, and other key issues. It establishes a vision for the organizational use of good systems integration practices and CORBA.

Chapter 2, "Standards Strategy," defines what standards mean and how they are used effectively. It provides guidance for understanding and exploiting available standards and shows how an organization can leverage the work of government organizations and standards groups to define a comprehensive standards strategy.

Chapter 3, "An Introduction to CORBA," provides essential tutorial information about OMG standards: CORBA, OMG's Interface Definition Language (OMG IDL), CORBAservices (Common Object Service Specifications), and CORBAfacilities (OMG Common Facilities).

Chapter 4, "Software Architecture Design," includes the foundational theories of systems integration. It describes the integration capability model and defines architecture and framework concepts. It establishes an integrated software architecture design process and defines details of how to use OMG IDL for controlling design trade-offs. It also compares the architectural patterns that support the theory and shows the quantitative impact of each architectural approach.

Chapter 5, "Security," discusses one of the most challenging aspects of systems integration. This chapter surveys key security technologies that contribute to an overall security solution and establishes a vision for how they will interact in future systems constructed with OMG standards.

Chapter 6, “Framework Examples,” describes Microsoft Object Linking and Embedding (OLE), OpenDoc, and X11R6 Fresco. These are some of the key commercial software architectures that will impact future systems integration.

Chapter 7, “In-Depth Example: The DISCUS Framework,” discusses DISCUS, one of the most mature CORBA-based application architectures. It is presented here as a detailed example of systems integration practices. DISCUS incorporates many useful design examples, techniques, and lessons that can be transferred to new system designs and implementations.

Chapter 8, “Object Wrapper Techniques,” discusses object wrapping, the core competency in systems integration for developers. This chapter is an essential tutorial about wrapping techniques for architects and developers. The chapter gives examples of object wrapping for the most commonly occurring integration challenges.

Chapter 9, “Systems Integration Guidance,” summarizes the whole book in terms of six key guidelines.

Appendix, “ORB Products,” contains product overviews of some of the major CORBA Object Request Brokers: Digital Equipment Corporation’s ObjectBroker, SunSoft’s DOE, IONA’s Orbix, and IBM’s System Object Model (SOM).

I

THE PRINCIPLES OF SYSTEMS INTEGRATION

1

Introduction

In this introductory chapter, we identify the key issues that motivate this book. In particular, we wish to heighten the contradictions in current systems integration practice. In this way, we can summarize all of the key technical and related nontechnical issues. In the balance of the book, we will explain Object Management Group (OMG) standards and other key concepts needed for successful systems integration. OMG's core standard, the Common Object Request Broker Architecture (CORBA), provides many technical benefits. We provide added value by addressing how to use the OMG standards effectively in systems integration. This chapter focuses on systems integration issues prior to the adoption of CORBA and prior to the application of the techniques presented in this book.

While most of the book focuses on the technical issues, this chapter will identify some of the factors outside the technical realm that influence systems integration. For example, software architectures often mirror our organizational structures, and systems integration success is dependent upon organizational support for the architects and developers.

Not long ago, a stand-alone program comprising a command-line interface and a few specialized functions was sufficient for a successful software application. Since then, end users' expectations have changed dramatically; the scope of requirements also has changed. Today it is expected that software systems will have graphical user interfaces; software systems will incorporate complex data types such as imagery, graphics, and multimedia; and software systems will have integrated database capabilities. Heterogeneous computing environments are a reality in most organizations where differences between platforms reinforce the need for multivendor integration solutions.

Systems integration requirements will become even more challenging in the near future. For example, universal data interchange between applications will be expected; applications will need to support computer-assisted human collaboration, and these capabilities will be needed in mobile distributed environments, supported on a wide range of platforms and networks. Standards support will be a key software product discriminator for end users; however these standards might not guarantee interoperability.

For most organizations, it is not feasible to build systems of this scope using custom software. As a result, the practice of building systems has changed from custom programming to systems integration from preexisting components. Custom applications have a role in meeting highly specialized needs, such as organization-specific user interfaces. As the diversity and capabilities of commercial software increase, the need for custom software decreases. For complex functions, it is usually more cost effective to reuse existing software than to create custom software. The majority of software in future systems will be preexisting, obtained from commercial vendors and other sources. Systems integration has a role in providing interapplication interoperability and system customization beyond the off-the-shelf capabilities; design and development of this custom software is a key focus of this book.

The growing supply of preexisting software is displacing the need for custom software. Preexisting software sources include in-house software, commercial software, public domain software, freeware, and shareware. In-house software can include legacy systems software, prototype software, and software designed for in-house reuse, such as class libraries. Commercial software is available directly from suppliers, retail outlets, industry conferences, and other advertising events and media. Public domain software is freely available software that is not copyrighted and can be used without restrictions. Freeware is similar, except that it carries a copyright notice. Shareware is software that is available for a nominal fee and also carries a copyright. Copyright can restrict the reselling of freeware and shareware, but the copyrights generally allow the usage of the software in end-user systems. Public domain, freeware, and shareware can be obtained from users' groups, from vendors, and from public networks (including the Internet and commercial computer networks).

The practice of systems integration can range from informal ad hoc programming to formal structured methodologies. In many organizations these practices coexist: the formal practices creating the documentation and the informal practices creating the working system.

The difficulties of integrating preexisting components create a need for highly creative programmers. Many successful programmers are masters of scrounging software components and integrating components in ingenious ways. These skills are essential to virtually every software engineering activity; ad hoc integration is expedient and leads to demonstrable results.

Often these integration solutions are brilliant, but these achievements are seldom sustainable. Ad hoc integration usually produces undocumented brittle systems that are expensive to maintain and costly to upgrade. When staff turnover occurs, organizations find themselves with cryptic code that is extremely difficult to understand. This code is practically unmodifiable and at some stage must be thrown away, possibly to be replaced by another programmer's undocumented code.

Formal design methodologies generally are not supportive of preexisting software integration. Methodologies typically address subsystem level issues that are relevant to the design of a single custom application. In addition, change to requirements during the system life cycle are a major system cost driver, perhaps involving up to 50 percent of all software costs [Horowitz, 93]. Most methodologies do not address the issues of integrating independently designed components and maintaining systems through long life cycles entailing requirements changes and multiple system extensions.

In this book, we offer some alternatives to the shortcomings of ad hoc integration and formal methodologies. We show how to leverage practical systems integration skills into the creation of documented, extensible, standards-based systems. This book offers a new way to think about systems integration and a new way to package the integration solution. Much of the ad hoc integration code will still reside in the resulting system, but it will be incorporated in a way that enables the reuse of integration solutions as well as the component software. We believe that our approach enables the development of robust software architectures that support enhanced reusability and extensibility. These software architectures contain an appropriate balance between the need for formal documentation and the realities of programming with preexisting components.

In this chapter we introduce the key issues of computing technology and systems integration. Then we overview the key concepts in our software architecture design and integration approach.

ISSUES IN COMPUTING: TECHNOLOGY, MARKET, AND ORGANIZATIONS

The following are some important factors in computing that tend to exacerbate systems integration challenges. These issues are challenging to resolve because they combine business, human, and technological factors. Each systems integration organization should be aware of these issues and possible approaches for mitigating the consequences.

Stovepipe Systems

A stovepipe system is a set of legacy applications that resists adaptation to user and organizational needs. To the user, stovepipe systems appear to be unfriendly legacy applications that lack interoperability. These deficiencies become obvious when compared with inexpensive PC software. Stovepipe systems are brittle; they do not tolerate change. In contrast, computing technology and facilities are changing rapidly. As systems change and age toward obsolescence, users experience an increase in bugs and system failures.

Some key characteristics of stovepipe systems include:

- Monolithic, vertically integrated applications
- Closed system: custom proprietary solution
- No discernible software architecture
  –system structure poorly understood by developers and maintainers
- Lack of provision for reuse and extension
- Slow development and deployment
- Expensive maintenance and evolution

Stovepipe systems are the source of many problems for end users, developers, maintainers, and management. Even though the problems are obvious, the solutions are not obvious, even to computer industry experts. We have consulted with many senior technologists and experts in the computer industry who believe that there are no effective solutions and that even the best next-generation systems of today will be the stovepipe systems of the future.

We believe that there are two fundamental causes of stovepipe systems:

1. The way that systems are acquired.
2. Lack of architectural focus.

New systems are typically acquired in a manner that facilitates the creation of a stovepipe system. Many acquisitions are driven by external system requirements, such as the user interface functionality. Most requirements documents are snapshots of user needs. Changing requirements are the major cost drivers in software development and maintenance. [Horowitz, 93]

Acquisitions typically involve organizational relationships that maximize the negative characteristics of stovepipe systems. For example, an inventory system for a parts supply department might be acquired from the outside contractor who submitted the lowest bid. The contractor's interest may be to deliver a minimal cost system that meets the requirements. To cut costs, the system may contain various ad hoc integration solutions and the presence of these may benefit the contractor with more work later, as the customer's requirements change. The parts supply department, focused on its own internal needs, may acquire the system without much consideration of how the new system will interoperate with the rest of the enterprise. Later, it may be discovered that integration with the parts consumer's systems is highly desirable, but difficult to achieve. In this example, the new inventory system becomes a stovepipe system primarily because of the way it is acquired.

The other fundamental cause of stovepipe systems involves a lack of architectural focus. Throughout this book, we present the principles and practices of how to use CORBA with a strong architectural focus in order to overcome the deficiencies of stovepipe systems. Architectural focus can be derailed in many ways. Inadequate training and experience is the most significant problem. Few university curricula include software architecture training. Architecture-level design is not widely practiced and taught by computer industry experts.

Individuals and organizations can easily lose their architectural focus to external influences, such as marketing information from commercial software vendors. Due to their substantial marketing expenditures, commercial vendors are the dominant sources of information for users and developers. Not all, but some marketing information is biased, and much of the information contradicts information from other vendors. These contradictions are the source of much confusion for users and developers. Architectural focus is strictly a user/developer concern that vendors may address in product-specific ways.

Proprietary Lock-In

A widely accepted practice of commercial vendors and systems integrators is to build proprietary integration solutions among software components that they develop and sell. Virtually all office automation packages are integrated in this manner. We call this practice proprietary lock-in. Proprietary lock-in can be the result of a specific marketing strategy. More likely, it is the natural result of an isolated system design and development. By default, independently developed computing systems will lack interoperability and adaptability.

Proprietary lock-in inhibits system extension. System extensions are the most commonplace of software activities and the most expensive driver of software costs [Horowitz, 93]. Once an organization commits to a proprietary solution, it is very difficult to extend the system without the vendor's involvement. Most end-user organizations have very little control over software vendors; hence software costs are driven by external organizations that have no direct interest in minimizing software costs.

Adaptability is the key quality of a system designed to minimize software costs. Software architecture is the functional structure of the system that provides adaptability.

Good software architectures are based on only two types of design elements: things that are stable and things that can be controlled. Standards represent the stable technology elements. If a standard is actively supported by multiple vendors, then the standard is likely to be the most stable, lowest-risk technology choice, compared to any proprietary technology. Controllable design elements are those that are under the direct control of the user organization as well as system characteristics that can be changed inexpensively.

A key part of the problem is that very few end users have identified software architecture as a deliverable part of a computing system [Horowitz, 93]. Traditionally, the focus of system acquisition has been on predefined user interface requirements. Other types of requirements are difficult to verify except at the computer-user interface. Since software architecture is the internal structure of the system, user requirements have little or no relationship to software architecture. Without a deliverable software architecture, the end user invites an undocumented lock-in integration solution.

Large-volume end users are beginning to demand open systems compliance, including conformance to standards profiles. (See Chapter 2.) Standards groups such as the Object Management Group (OMG) are making excellent progress creating multivendor interoperability standards, but available standards are far from addressing all of the end user interoperability needs. Systems integration and software architecture are needed to fill the gap between available standards and end user interoperability needs.

Another way vendors enforce lock-in is through data formats. Many packages provide data format translators for importing data from competitor's packages but no corresponding filters for exporting data to other applications. Once the end user imports data into a lock-in application, it becomes effectively trapped in the vendor's proprietary data format.

As a least common denominator, plain ASCII text usually can be transferred between applications, with compromise of data quality, such as pictures and formats. A number of third-party vendors, such as KeyPack and BlueBerry, sell high-quality filter packages to support multivendor data interchange. A software architecture in the end-user system can leverage these technologies to mitigate the lack of export filters in some commercial packages. In addition to application lock-in, vendors may hamper interoperability through platform lock-in.

Although decreasingly common among UNIX vendors, platform lock-in is still an active practice among PC operating systems vendors. In effect, platform dependencies restrict portability of third-party software and end-user software to foreign platforms. Many vendors need to maintain several independent versions of their products. A common description of this problem is that vendors need to maintain one version for Microsoft Windows platforms and another version for the Macintosh, RISC, and other platforms.

The cost of maintaining software with proprietary platform application program interfaces (APIs) is prohibitive for most volume end users, who find these platform API's unsatisfactory as de facto standards because they are beyond their development resources to use effectively. This excess cost has two aspects: excess complexity and technology stability.

Complex proprietary APIs often require highly trained development staff to utilize them effectively. These skills are not directly transferable between open systems and proprietary platforms. Training of programmers can take several years, and maintaining this separate skill base is a prohibitive cost factor for volume end users.

A key element of technology instability is technology evolution. Technology evolution requires maintenance costs, such as product version upgrade, upgrading integration code, testing, training, help-desk support, and so forth. The maintenance costs are increased due to the continual technology evolution and obsolescence of proprietary technologies.

At the desktop level, there are likely to continue to be several incompatible APIs, including Macintosh, Microsoft Windows, Common Open System Environment (COSE), Common Desktop Environment (CDE), OPENSTEP, and Taligent. In heterogeneous environments, there is likely to be some need for platform-dependent custom software. In the section titled Advanced Architecture Pattern: Separation of Facilities on page 97, we describe an architecture strategy for minimizing the potential costs supporting those platform dependent APIs.

Inconsistencies between Design and Implementation

The traditional waterfall system development process comprises a top-down design process starting with end-user requirements. The waterfall concept is implicit in most formal system design processes. Regardless of whether the analysis and design methodology is based on structured programming, object orientation, or design patterns, many methodologies rely exclusively on top-down analysis of the application problem. Waterfall methodologies assume that the application problem can be fully understood before implementation begins. We believe that this is a fundamentally flawed assumption. Implementation experience is essential to the creation of good software architectures.

The application problem (derived from end-user requirements) and the technical problem (for the software architect) are different. When the integration of preexisting components is the dominant development paradigm, we have found that the essential understanding of the technical problem is gained during the integration process. It is very difficult to scope this type of effort because the implementation requirements do not become apparent until each subsystem is integrated. Project planning involves substantial guesswork. Project managers cope with this uncertainty by employing highly talented programmers who can bail out a project, regardless of the complexity that is discovered during implementation. In practice, lessons learned during integration seldom bubble up to the formal system design.

There is a basic disconnect between formal development methodologies and the informal process of system implementation. Due to their respective roles, managers and senior staff are engaged in the formal process while programmers conduct the informal process. The authors have participated in projects where there is minimal communicationbetween these formal and

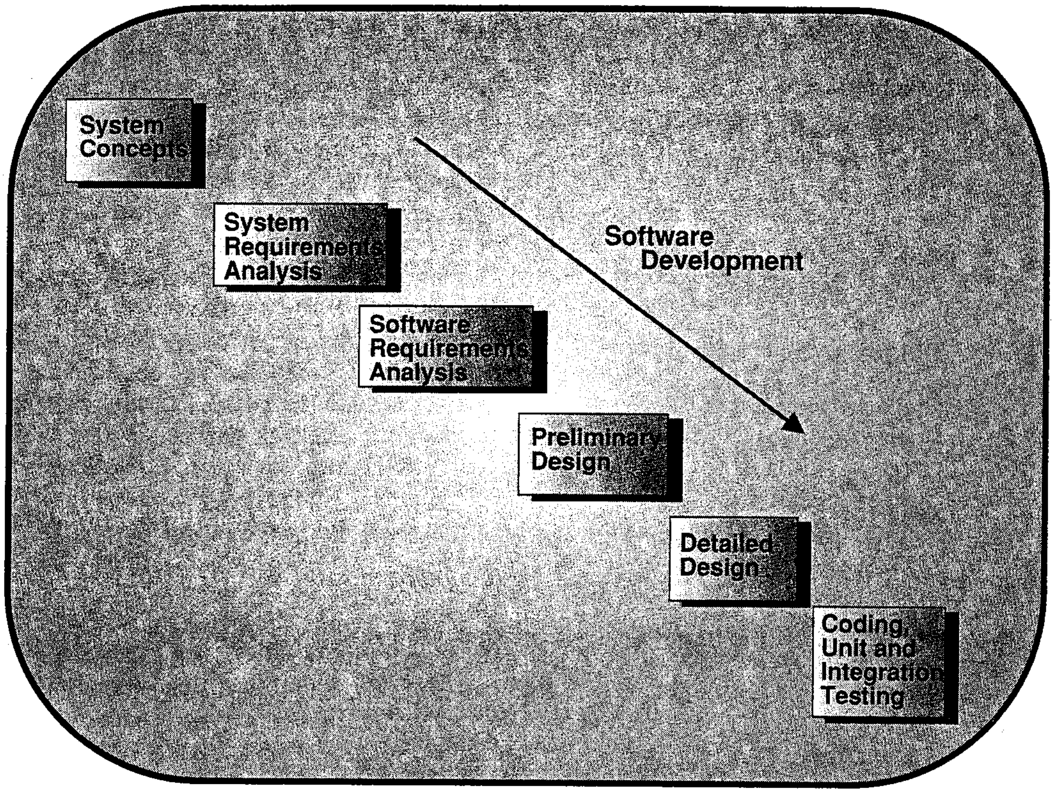

System Concepts
System Requirements Analysis
Software Requirements Analysis
Preliminary Design
Detailed Design
Coding, Unit and Integration Testing
Software Development

Figure 1.1. Traditional waterfall process.

informal processes, except for formal process documents that programmers are required to produce. Many programmers view the formal process as a burden and view their real task in the context of the informal process that produces the actual system implementation. Many managers are willing to relax conformance of the implementation to the formal specification in favor of successful implementation demonstrations.

Because the formal process appears to not provide substantive benefits to the informal process, senior staff and programmers have little motivation to maintain consistency between the design and the implementation. There is also a general lack of tools that can enforce consistency and reveal inconsistencies between implementation and architecture. At best, rigid conformance to formal methodologies is uncorrelated to project success, and there are many examples of projects where formal methodologies have had an adverse impact on project success [Coplien, 94]. What is needed is a balanced approach that can allow programmers the freedom to pursue informal integration processes within the context of a formal system architecture that is consistent with the implementation. Our guidance for this approach is summarized in Chapter 9. The balance of the book provides the details of the approach for software architects and developers.

Two non-waterfall-based methodologies include the *incremental development process* used in artificial intelligence and *Rapid Structured Prototyping* [Connell, 87]. These methodologies are called spiral development processes, in contrast to the waterfall-based processes.

Both methodologies recognize that the problem is poorly understood at the beginning of the project and that understanding of the problem grows as prototypes generate feedback from end users. The end-user involvement reduces overall project risk by giving the user experience with the new system and impacting the development process as it unfolds. The incremental development process involves knowledge gathering from a domain expert, the creation of a prototype knowledge base, and the refinement of the knowledge with end-user involvement. Rapid Structured Prototyping is a phased methodology where each phase defines system extensions, builds the extensions, and gathers end-user feedback. The process is iterated with each

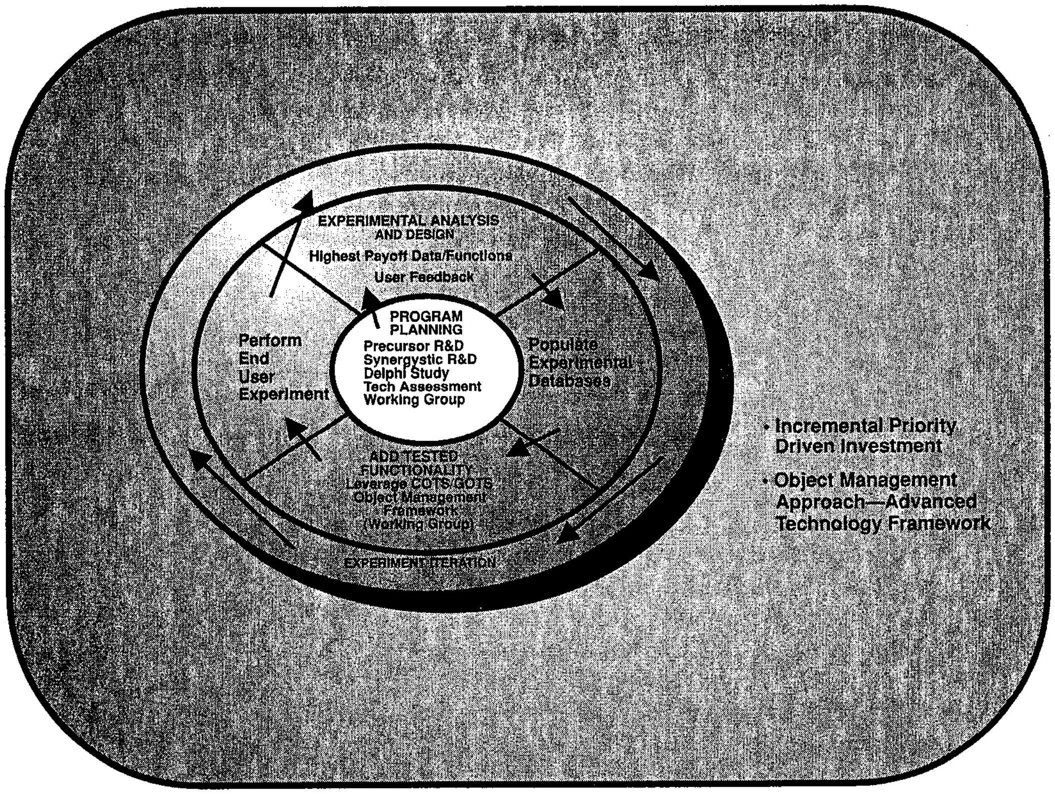

EXPERIMENTAL ANALYSIS AND DESIGN
Highest Payoff Data/Functions
User Feedback
PROGRAM PLANNING
Precursor R&D
Synergystic R&D
Delphi Study
Tech Assessment
Working Group
Perform End User Experiment
Populate Experimental Databases
ADD TESTED FUNCTIONALITY
Leverage COTS/GOTS
Object Management Framework
(Working Group)
EXPERIMENT ITERATION
• Incremental Priority Driven Investment
• Object Management Approach—Advanced Technology Framework

Figure 1.2. Spiral development.

experiment providing the planning for the next phase. Unfortunately, spiral processes do not produce system architectures as a natural outcome; nevertheless, they assume that programmers can reimplement and extend major features of the system at any time during the life cycle.

Standards Issues

Standards and the standards development process have been evolving over many years. Standards and their processes that precede OMG and CORBA have demonstrated some key strengths and weaknesses.

In general, standards hold out the promise of interoperability and component interchangeability. Current standards cover virtually every aspect of computer systems, and many organizations have compiled comprehensive profiles of standards that attempt to make some sense of the usage of standards in the context of system design. (See Chapter 2.)

Standards as a whole have not been effective in delivering the promised benefits of interoperability and interchangeability. Part of the reason is due to the organization of formal standards bodies. Formal standards are authored by representatives from profit-making industrial firms that have strong proprietary motivations. For many companies, the creation of a standard is viewed as a strategic opportunity. If a company can standardize its own product specifications in favor of a competitor's product specifications, then it can enter the market early with the first standards-compliant product while waving the flag of the standards organization.

Many standards play the role of business regulations—laws installed by one business to restrict the commerce of potential competitors. Excessively complex standards can effectively inhibit market entry of new products. For example, some have estimated that it would cost on the order of $100 million to create a standard relational database product from scratch.

In formal standards bodies, standards grow in complexity dramatically during their adoption process. One wonders if the size of a standard is only limited by the authors' ability to carry it. Large, complex standards have many technical loopholes. Various waivers are written into many standards to allow a wide range of compliant products. For most standards, there are products at all levels of compliance, and vendors can selectively support any part of the standards and ignore others. Compliance tests are useful, where they exist, but they do not cover the broad range of applicable standards in computing applications.

Even if standards addressed all these issues, a single standard is unlikely to meet all the needs of end-user environments. In any given standards area, there are competing standards. In order to build effective integration solutions, the coexistence of multiple standards and proprietary solutions for each capability must be considered.

A formal standards process requires four or more years; many formal standards are obsolete when they are finally adopted. Innovative vendors can easily outpace a formal standards process, providing significant new proprietary capabilities.

If traditional formal standards do not provide portability and interoperability, what is their purpose? One explanation is that standards play an essential role in establishing technology markets. A standard lends an air of credibility to a technology that translates into market growth. For example, without the Structured Query Language 1989 (SQL89) standard, supported from its inception by IBM, the relational database market would not have developed into the industry-dominating force that we know today. At both marketing and technical levels, standards reduce risk for both suppliers and consumers.

Prior to OMG, standards benefits were at the marketing level. For many consumers, the marketing level is equivalent to the political decision-making level. With OMG's focus on multi-language API specifications, OMG standards also potentially provide real technical benefits, such as interoperability and portability.

Lack of Tools for Creating Effective Abstractions

The creation of higher-level machine abstractions is a fundamental concept of computer science. For our purposes, an abstraction is a simplified interface that provides access to some underlying lower level mechanism. Programming languages are abstractions of the underlying machine code. Databases are abstractions of persistent storage. User interfaces are abstractions of computer programs. Abstractions are useful for managing complexity, and effective abstractions can reduce system cost by simplifying integration.

Object orientation is essentially an abstraction-building paradigm. A fundamental characteristic of objects is encapsulation, which is an abstraction for some combined unit of software and storage that hides internal data representations, algorithm choice, and other implementation details. At the level of systems integration, effective tools for creating abstractions are lacking with the possible exception of Object Management. As we discussed, system design and system implementation tend to diverge early in the life cycle. When the system design and the system implementation are completely separate descriptions, keeping the two in synchronization is a fundamental problem. Auditing of code and reverse engineering might reveal some consistency issues, but these techniques have not been widely applied in practice. Consider this question: What project manager would risk breaking a working system to enforce architecture consistency and would be willing to pay for it?

What we describe herein is an approach based on Object Management that addresses many of these issues. There is a machine verifiable relationship between the software architecture and the system implementation. To create highly effective abstractions of complex system components (so-called black box abstractions), an implementation-independent way to specify these abstractions is needed, such as that provided by the OMG's Interface Definition Language (IDL).

Lack of Security Facilities

Security is a pervasive issue in computing systems. The importance of security is exacerbated by the popularity of networking, since networking enables large groups to access vulnerable computing resources. For example, security is a critical issue on the current Internet because network hackers have penetrated and disrupted many systems that are directly connected, including systems containing vital operational functions, such as payroll or classified data.

Security is a requirement that most end users acknowledge, but few organizations are able to realize any workable solutions. Computer security in a networked interoperable environment involves both horizontal and vertical issues. Security must be enforced pervasively across all services and communication interfaces (horizontal). Security also must apply to every level of the system, from the applications down to the core of the operating system kernel. In practice, computer security is very difficult to achieve, short of complete system isolation.

Commercially available security technology is lacking. Technologies such as Remote Procedure Call (RPC) based Security Service are useful but only a partial solution. Effective solutions must be enforced pervasively across every type of integration technology, not just distributed RPC. Security technologies, such as special-purpose operating systems and Compartmented Mode Workstations, have never achieved the expected commercial success.

We believe that CORBA provides new opportunity for the creation of secure software architectures and commercially viable security technologies. Chapter 5 contains a discussion of computer security, including issues, standards activities, and security architecture approaches.

HETEROGENEOUS SYSTEMS INTEGRATION

Systems integration is an increasingly important discipline. As software technology continues to evolve, there is a growing trend toward the construction of systems from preexisting components. This trend is due in a large part to readily available high-quality commercial software packages that perform most major functions in today's information systems. In the 1970s, complex software systems were configured almost exclusively from custom software components, with the exception of perhaps a major component, such as a commercial database package. In the 1980s, the availability of bitmapped graphics workstations led to the development of all-in-one software packages so that complex systems could be configured around a major package, such as a Geographic Information System (GIS) or a document processing system. Other components were custom integrated as needed. In these systems, there remained a gap in interoperability from the end users' perspective; the system integrator's role was to provide a level of interoperability for the end user that was unavailable in the off-the-shelf software packages.

Users' expectations for interoperability have increased dramatically, due to the readily available highly interoperable platforms such as Microsoft Windows and Apple Macintosh. Unfortunately, vendor-specific solutions do not solve today's heterogeneous systems integration challenges. Information systems are increasingly heterogeneous; it is rare to find any organization with a monolithic computing base (all users on the same platform). Increasingly, an organization's interoperability needs span its customers' and suppliers' networked offices. Heterogeneity will increase as software becomes more portable and the role of computing becomes ubiquitous, with computing resources ranging from wireless notepads to massively parallel super computers.

Vision for Systems Integration

Ideally, systems integration should be as simple as plugging together a home audio system. In practice, it is a challenging activity and rarely results in a good software architecture. Systems integrators face problems with, among other things, different

- Hardware platforms
- Software languages
- Compiler versions
- Data access mechanisms
- Component/module interfaces
- Networking protocols

Custom integration approaches usually yield point-to-point integration solutions, and the system becomes difficult to extend due to the complexity and brittleness of the solution.

Our vision for systems integration is founded on the need for system adaptability through the system life cycle. In this book, we introduce software architectures (integration frameworks) as the core concept for realizing adaptability. We use the terms architecture and framework interchangeably.

Common interfaces is a key property of good software architectures. Ideally, the same common interfaces are supported by custom components and commercial software. Unfortunately, commercial interfaces are not always the most appropriate solutions; they may mismatch the cost, complexity, portability, or stability needs of the application system. The role of the soft-

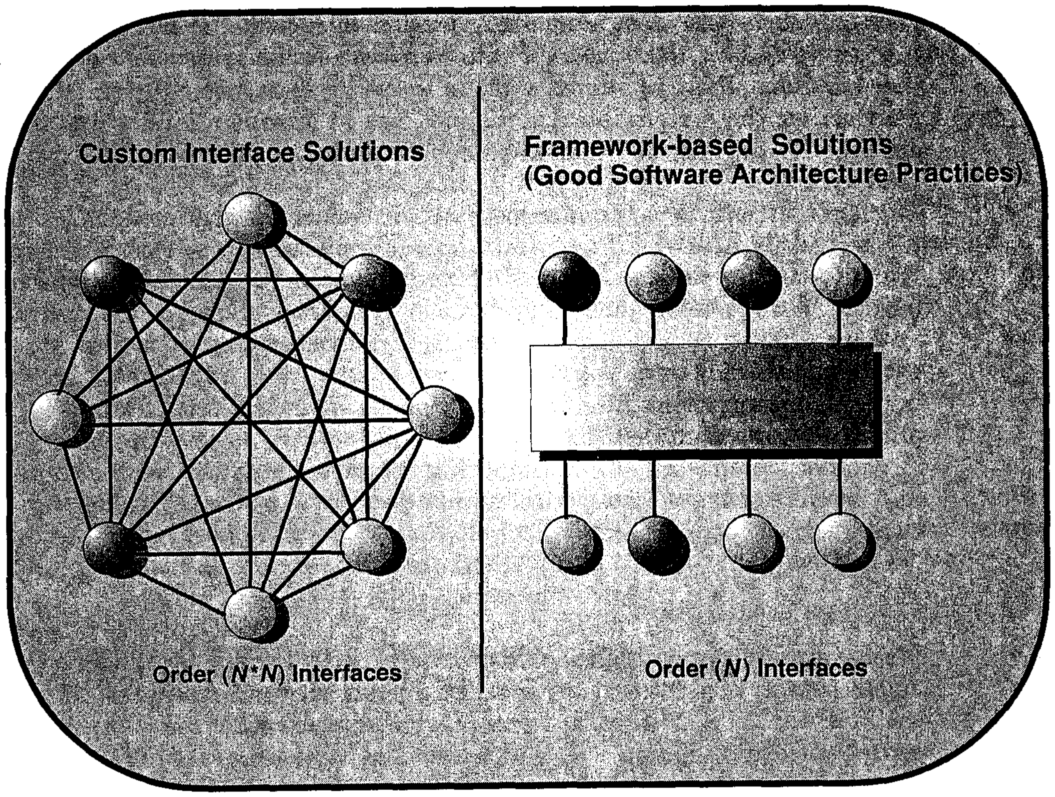

Custom Interface Solutions
Framework-based Solutions (Good Software Architecture Practices)
Order (N*N) Interfaces
Order (N) Interfaces

Figure 1.3. Custom interfaces vs. framework based.

ware architecture is to provide the right system-specific abstraction that can be interoperable with commercial interfaces.

The framework should conform to the object-oriented paradigm, for some practical reasons. For example, the object paradigm provides encapsulation that enhances component isolation. Object orientation supports design reuse through inheritance. Polymorphism supports component interchangeability. Object Management is a particularly appropriate technology for implementation of these concepts, although simply practicing these principles alone can yield measurable benefits. All of the characteristics of the object paradigm are potentially beneficial, but the realization of these benefits is closely tied to the design of the software architecture.

In this book, we emphasize that architectures should exploit methodology paradigms (such as structured programming, object-oriented analysis and design) instead of being driven by them. Methodologies are perhaps the fastest changing of any aspect of technology, and methodologies will go through many generations during a system life cycle. Good architectures must transcend ephemeral methodologies in order to provide their benefits consistently during the system life cycle.

In the implementation of an architecture, all the software subsystems support the integration framework. A layer of integration code called an object wrapper may be added to preexisting modules to provide the mapping between framework operations and application operations. A well-designed framework will require a minimum of integration code in this layer.

Adaptability comes from the ability to modify or add modules without having to modify the other components or their integration solutions. The system should be self-describing, containing sufficient metadata so that new subsystems can be discovered dynamically by preexisting subsystems. Semantic and data format translations should be layered into the system at the level of the integration framework, not tightly integrated to individual applications. These mappings can be installed incrementally as part of the system evolution, ensuring that legacy interfaces will be maintained as the system evolves and providing a graceful evolution path from initial prototypes throughout the system life cycle. Legacy systems provide a special integration challenge since they often are closed and may provide no access API or documentation. In Chapter 8 we present various object-wrapping techniques and examples that can be used during the system's migration and integration with other legacy or new systems.

THE ROLE OF CORBA

Interoperability between applications presents similar problems to that of interoperability between systems, just at a different scale and layer. Many advancements have been made in the area of systems integration and global networking.

Today communication networking capabilities are widely available between systems. Communication networks, phone lines, cables, and fiber are available; the necessary hardware also is available to connect systems to local area or wide area networks, hubs, and switches.

We also have well-defined protocols that allow systems to communicate by agreeing to certain standards that will be used. The Open Systems Interconnection (OSI) 7 layer model provides the definition of each layer and the services that it must perform.

The lowest layer, the *Physical* layer, describes how the physical network is accessed. The *Data Link* layer is concerned with reliable transmission across a physical link. The *Network* layer deals with connection establishment and routing. The *Transport* layer deals with reliable end-to-end transmission. The *Session* layer provides connection control. The *Presentation* layer is concerned with data syntax and transparency to the applications. Finally, the upper layer, the *Application* layer, provides end-user functional-

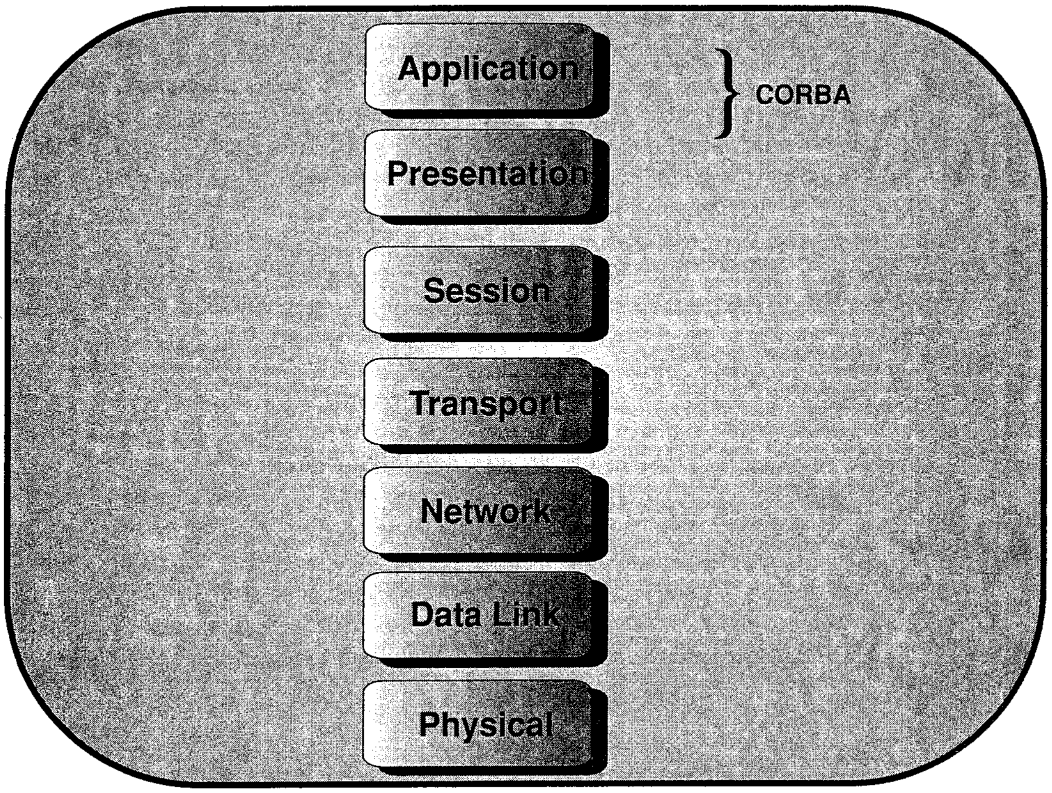


Figure 1.4. OSI 7 layer model.

ity. Much research also has been conducted in the area of security to provide the protection of data and to provide privacy and data integrity.

CORBA represents the next generation of client-server facilities that provide highly distributed systems and applications. Conceptually, CORBA sits in the application layer.

CORBA insulates the client and actually the programmers from the distributed heterogeneous characteristics of the information system.

- OMG IDL provides an operating system and programming language independent interface.
- Due to this higher level of abstraction, the programmer does not have to be concerned with the lower layer protocols.
- The programmer does not have to be concerned with the server location or activation state.
- The programmer does not have to be concerned with the server host hardware or operating system.

- Certain integration issues are simplified; for example, no longer does a client or server have to be concerned with byte ordering of data when transferring data between different platforms.

The Object Request Broker (ORB) provides a seamless infrastructure for distributed communication across heterogeneous systems. OMG IDL provides a common language and syntax for client and server access. A remaining issue is the selection of APIs used between applications. Technology has provided excellent connectivity between hardware platforms, but there is no analogous interoperability at the level of application functionality and application data interchange. CORBA paves the way for component object systems. A system builder should be able to select objects from several vendors and connect them as easily as we connect audio components at home today.

If commercial objects use OMG IDL interfaces and are otherwise CORBA compliant, we are a step closer to integrating the applications. CORBA enables objects to discover each other at runtime and invoke each other's services. Standardized APIs provide another level of interoperability. OMG has a growing set of API standards, called Object Services and Common Facilities. (See Chapter 3.) The Object Service definitions cover fundamental APIs, and the Common Facilities provide higher-level API standards.

Another level of specialization beyond standardized APIs addresses the interoperability needs of individual application systems. In the OMG architecture, these are called application objects. Systems integration concerns the design, implementation, and integration of application objects.

In this book, we describe the system-level design of these objects, which we call software architectures and frameworks (Chapter 4). We provide many important examples of architectures and frameworks (Chapters 6 and 7). We also describe how to implement these system-level designs given preexisting components, a practice that we call object wrapping (Chapter 8).

COMMENTS

This book is divided into two parts. Part I comprises Chapters 1 through 5 and covers the principles of systems integration. Here we present the concepts on which we have based our approach to the integration problems at hand. Chapter 2 covers strategies for using standards effectively through technical reference models and standards profiles. Chapter 3 presents a detailed introduction to CORBA and related standards. Design approaches for designing architectures and frameworks are covered in Chapter 4. We conclude Part I with an introduction to security problems and concepts (Chapter 5), and we discuss security issues as they relate to the CORBA distributed architecture.

Part II, comprised of Chapters 6 through 9, presents detailed integration examples, lessons learned and various integration methods used in different situations. Chapter 6 provides framework examples from the commercial world. Chapter 7 contains a detailed example of a desktop-independent and platform-independent interoperability framework developed using our approach. Chapter 8 discusses various object-wrapping techniques and gives program examples. Chapter 9 provides guidance both to managers and developers on using CORBA and system architecture. The appendix contains summary descriptions of key CORBA products.

2 Standards Strategy

In order to design a successful, long-lived software architecture, software engineers must have an understanding of standards. Selecting and implementing a standards strategy is the first step in our systems integration approach.

A standards strategy is a plan for how to develop and evolve information systems toward support for open systems standards and strategic standards creation. Standards strategies can have multiple levels, including reference models across multiple projects and project-specific standards profiles.

A key part of the standards strategy is a guidance document, called a standards profile, that influences choices of information system components based on their support for the right standards. As systems evolve, through component upgrades, extensions, and error correction, the standards profile identifies the standards objectives of the future system and guides the migration toward a goal architecture. Another key part of a standards strategy involves the development of new specifications, which are a part of the development of any new system or system extension. The organization must consider the role that these specifications play in a larger community context.

In essence, a standards strategy involves making sense out of the complex, dynamic arena of open systems standards and exploiting standards activities as opportunities to save costs and realize organizational goals. The cost of implementing the standards strategy can be highly leveraged by exploiting readily available information and harmonizing the organization's standards initiatives with emerging trends.

STANDARDS BACKGROUND

Historically, suppliers have been the key drivers of standards processes. For example, the fundamental concept behind "open systems" is standards. Most open systems standards have portability as their primary objective. This goal is motivated by a traditional view of the computing industry as driven by hardware suppliers; narrowly defined, portability means independence from particular hardware suppliers. An expanded view of portability also considers the full range of software APIs, such as operating systems, windowing interfaces, and higher level services.

The industry is experiencing some important new trends in the standards arena. There is a growing awareness of the need for areas of cooperation between organizations through standards because standards create markets for products. We are seeing the creation of many new industry alliances, consortia, and special interest groups that are forums for standardization. We are also seeing many strong alliances between major standards groups, such as the Common Open Software Environment (COSE) with the Open Software Foundation (OSF) and the Object Management Group (OMG) with X/Open, the International Standards Organization Open Distributed Processing (ISO ODP), X/Consortium, and so forth. End users are playing an increasing role in standardization, primarily in high leverage vertical market areas. The OMG is the only consortium defining standards for distributed systems based on Object Management technology. (See Chapter 3 for more details on the OMG.) OMG has done much to fuel these trends by streamlining the standardization process and making standardization much more accessible. Interoperability is an important standards goal that is a key focus of the OMG.

The establishment of a standard enhances the credibility of a technical specification, reduces risks for clients and implementors of the interface, and enables reuse. The scope of the standard defines the scope of reuse. Probably the smallest useful scope of reuse is across a multiproject organization. This is a minimal organizational scope to consider when designing software architectures. (See Section Basic Architecture Patterns.) If we extend the scope across company boundaries, we can characterize the standards as applicable to a vertical or specialty market. Countless specialty market alliances and consortia groups provide forums for these standards. Standards at the specialty market level and larger scopes are essential for establishing commercial software markets. The standard becomes the common, stable element that reduces risks and provides technical benefits to both suppliers and consumers.

The OMG has reengineered the business process for standards creation and made it much more feasible to create new standards to fill gaps in specialty markets. (See Chapter 3.)

We can think of standards as the product of an organizational process promoting reusability, stability, reduced risk, and interoperability. Standards may take on different levels of formality, depending on our goals and strategy. The process of standards creation is an extension of system design and development. The standards process differs principally in the intended scope of impact. Whereas an ad hoc integration may solve an interoperability problem between two specific subsystems, an integration standard might be a reusable solution across a project, organization, industry, or larger context. Management is responsible for defining the standards context of a new specification and establishing the intended standards role in the target community.

From a traditional perspective, there are three principal types of standards: formal standards, de jure standards, and de facto standards. Formal standards are specifications that are formally adopted by accredited standards bodies such as the Institute of Electrical and Electronics Engineers (IEEE), the Consultative Committee International Telecommunications (CCITT), the American National Standards Institute (ANSI), and the International Standards Organization (ISO). There is a hierarchical relationship between national standards groups such as ANSI (for the United States) and ISO (encompassing an international community of national standards groups). De jure standards are those mandated by legal authorities. Federal Information Processing Standards (FIPS) are an example; these are standards approved by the United States for use in government information systems. In another example, a company can select and mandate de jure standards for its own organization. De facto standards are informal standards resulting from popular usage. X Windows and Network File Server are examples of de facto standards that became established through popular usage. Dominant vendors and vendor alliances frequently establish interfaces that become de facto standards through usage.

Standards are a broad category of specifications that encompass a great deal more than just the approved releases of formal standards organizations. Some standards are established by other forms of standards groups, such as industry consortia. In the past, de facto standards could be established by a single dominant company; today, multivendor support is essential for establishing de facto standards. De facto standards can also be established through informal means, for example, by popular usage.

In this chapter, our definition of the term standard is broad and closely related to software reuse. Standards are recognized technical agreements between people or organizations. The constituency of the group defines the appropriate forum for standardization. It may be effective for informal standards to be established across a single organization or project. Standards that are international in scope should be addressed in international forums, such as ISO, X/Open, and the OMG. Informal standards can be established at any organizational level between these extremes. It is simply a question of the appropriate organizational forum recognizing the specification at its standard. If the right forum does not exist, many groups have created new organizations explicitly for this purpose.

Some vendor alliances that create standards include X/Open, the OMG, the X Consortia, the OSF, the COSE alliance, and the Open GIS Foundation (OGF).

BENEFITS OF STANDARDIZATION

Some of the key benefits that standards provide include:

- *Portability.* Standards can provide portability of software between hardware and operating systems platforms, windowing systems, networking protocols, and a variety of other hardware and systems dependencies. The benefit is the ability to move and access software among multiple platforms in a heterogeneous environment. Portability also can support transparent upgrade of underlying hardware and networks.
- *Interoperability.* Standards define common formats and interface conventions that provide interoperability between software systems. Interoperability benefits might include data interchange, event notification, object embedding, application control, browsing, and others.
- *Risk reduction.* Use of standards is an essential approach that frees software architectures from implementation-specific dependencies. Use of standards can facilitate multisource alternative components, which reduces risks in system development, maintenance, and future system extensibility.
- *Interchangeability.* Standards promote product and subsystem interchangeability. Interchangeability can be used in several ways. It can be used to allow a wider choice of system components, selection of which may optimize cost, legacy leverage, performance, or other factors. And interchangeability can support deployment of multiple system versions, customized to site-specific user requirements. In addition, interchangeability supports system upgrade, as commercial and proprietary technologies evolve.
- *Cost reduction.* Commercial support of standards and their subsequent use for system integration implies a cost reduction for the end user. Commercial vendor's support of standards results in economy of scale. Cost reduction also can be realized through competition between alternative vendors conforming to the same standard.
- *Deferred obsolescence / Life cycle extension.* Conformance to standards can reduce the risk of system obsolescence. Standards can be used as a hedge against obsolescence, since many standards represent the most stable technology interfaces, which in most cases are supported continuously as technology evolves into upwardly compatible products.

As systems become more complex, standards become increasingly important. Standards provide one of the most accessible ways of assuring component compatibility.

SELECTING A STANDARDS STRATEGY

Available standards are quite numerous and individually complex. A strategy is needed to manage this complexity and diversity. Thousands of potentially applicable standards exist to cover the issues in most application systems. An effective standards strategy involves careful selection of standards among the many alternatives and selective use of parts of individual standards. After the selection, typical organizational standards profiles involve about 300 standards. Evaluating standards can be a complex and expensive task. The costs of strategy selection and implementation can be minimized by leveraging the efforts of standards groups and other organizations.

Application of a standard is not a simplistic decision. Most standards need to be utilized thoughtfully in order to yield their benefits. For example, most standards are very complex, and this complexity needs to be abstracted or subsetted in order to expose the desired functionality at the desired implementation cost. Standards also are designed for general applicability for multiple applications; many details must be added to the standard in order to make it useful within an application. For example, most standards are more general purpose and do not include any predefined data or metadata schemas (the schema is usually an application-specific profile); these schemas must be added to make use of the standard in the application system.

There are five potential goals to consider for your standards strategy.

Reducing Dependence on Custom Software A key goal of a standards strategy involves the migration of high-cost custom software functionality into low-cost-commodity commercial software functionality. For most organizations, this goal cannot be realized overnight; it involves a longer-term strategy to evolve system interfaces toward replaceable software components provided by commercial vendors. We can think of custom software playing several roles.

Custom software for systems integration fills interoperability gaps not supported directly by commercial packages. For example, we might need custom software to automate the process of moving data between two application programs. Originally, the custom software may have eliminated some rekeying and data entry operations, but as the system evolves and programming staff turns over, custom software can become a maintenance burden. As the system evolves, we can add support for standards in the application systems, which can ease the data transfer problem between the systems.

Custom software for applications might entail functions that are fully replaceable with commercially developed software, such as databases. Other custom applications, such as user interfaces, might be migrated to commercial application generators, which could reduce the cost of system modifications and new user interface development. The standards strategy provides the guidance and rationale to architects and developers for making these choices.

Leverage Current and Future Commercial Technology The standards strategy should identify commercial technology trends and place the organization in position to exploit support for standards as they become available. An important part of every standards strategy is knowing the mapping between commercial support to standards and predicting where commercial support is evolving.

Synergy with Standards Groups Standards activities are long-term public processes. Ascertaining their direction and predicting their progress is relatively easy to do by attending meetings, reading about them in the press, reviewing their publications, or talking to representatives. A highly effective standards strategy involves exploiting the products that the standards groups are producing as well as positioning internal development efforts to leverage emerging standards. Such products include:

- Reference implementations
- Vendor product disclosures
- White papers
- Plans and road maps
- Architecture

Your organization may be able to exploit some plans or preliminary specifications that standards groups will produce in the course of their process. Alternatively, your organization may be pursuing designs that you may consider proposing for standardization. A middle-ground strategy consists of designing architectures to accommodate emerging standards, instead of waiting for final approval.

Influencing Industry and Organizations A benefit of an effective standards strategy is the influence that the strategy asserts on commercial developers and other organizations. When a well-conceived strategy is explained clearly, it has significant impact on whoever is exposed to it. In our experience, we have noted positive changes in the product directions of many commercial firms after exposure to our standards approach. Many of the standards and technology issues that we identify are shared by other consumer organizations. Every time an issue is raised or concern is expressed about standards compliance, it has a measurable influence on suppliers. When many organizations are voicing their concerns, common themes emerge that raise the priorities for change. In a sense, each organization can play an activist role in shaping the standards strategies of other organizations. There are also direct ways of catalyzing technical change, such as reference technologies.

Reference Technologies If your organization is interested in establishing or promoting a new standard, an effective standards strategy involves developing reference technologies. If an implementation of the proposed standard that leverages support across a community can be built and disseminated, then it's adoption is greatly accelerated. Some successful examples of this strategy include X Windows, MOTIF, and Network File Server (NFS). In each case, technology access was provided either free or for a nominal fee, so that the technology could be transferred as rapidly as possible. The Internet greatly enables this kind of technology transfer, because it provides "free" electronic dissemination of software and "free" advertisements of software availability through bulletin boards and e-mail lists. Reference technologies can be built by individual companies, by academic institutions, by the government, and by consortia.

STANDARDS REFERENCE MODELS AND PROFILES

Because so many standards apply to computer systems, various well-organized lists of standards and rationale have been compiled as technical reference models and standards profiles. A technical reference model is a standards guideline for multiple computer application systems. A standards profile is a specific set of standards, which may apply to a particular computing system. Whereas a technical reference model will identify multiple alternative standards for each category, a standards profile will provide limited standards choices.

New standards reference models and profiles can be based on work that is already available. For example, IEEE Portable Open Systems X (POSIX) is a standards reference model that is generic enough to be a baseline reference model for most application domains. Tailored reference models are often extensions of generic reference models.

The X/Open Portability Guide (XPG) is a standards profile. XPG identifies specific standards choices for generic applications systems. X/Open goes further in certifying compliance of products and licensing a brand seal to products that meet XPG guidelines.

Regarding standards reference models, we believe that your organization should publish one and that it should be consulted as a normal part of decision making during computer system design, component selection, system extension, and purchases of commercial software. Creating your own standards reference model can be relatively inexpensive, given that multiple generic models exist. You must ask how your organization's application domain differs from the generic model and focus your attentions on adding more detail to the generic models in those areas.

Technical reference models and profiles should be used as guidelines, not rigid regulations. Rigorous standards compliance can be very costly, and a decision maker should balance the model's guidance with practical considerations. For example, if a function can be procured much cheaper or faster or with less risk without strict compliance, then it is wise to disregard the standards model; however, the rationale for this deviation should be documented and revisited later when the system is upgraded.

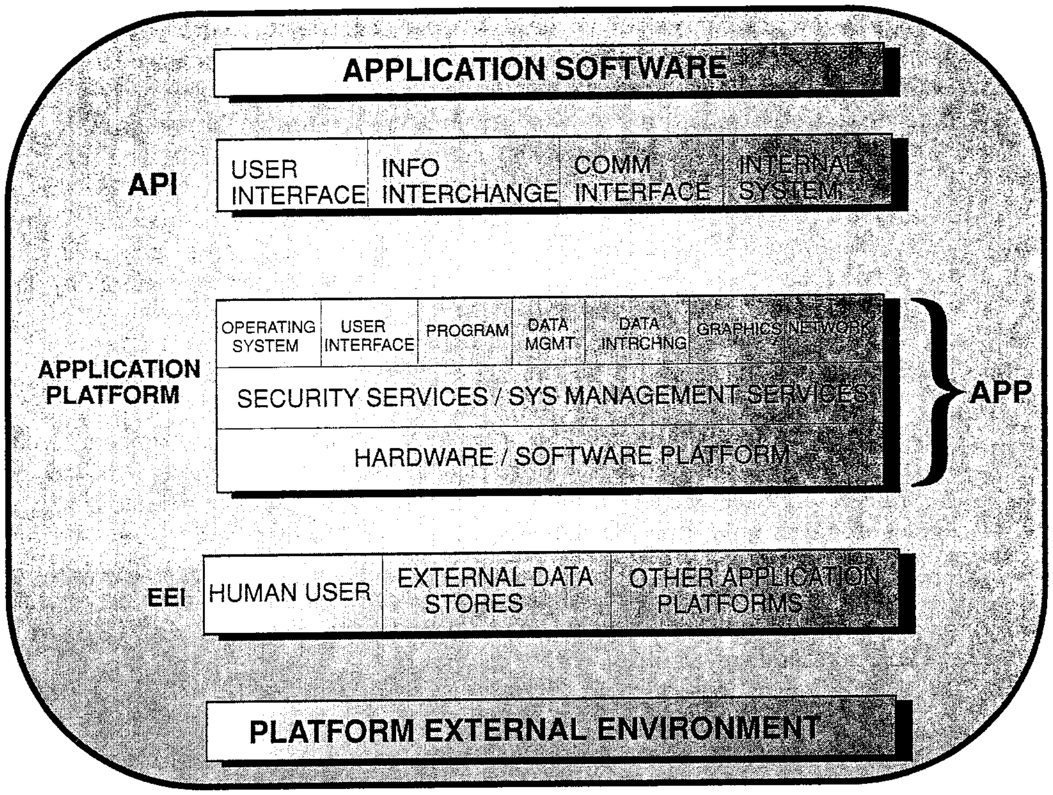


Figure 2.1. NIST Application portability profile (APP).

Another aspect of these models is that they need to be updated regularly. A standards model portrays a snapshot of a changing standards picture. The standards themselves can change or be replaced, such as when ISO adopted Structured Query Language 1992 (SQL92) to replace SQL89. Evaluating industry support for a standard is very important, and the situation can change as the market changes. Indirect changes occur when standards are added or deleted from XPG or POSIX that can have an affect on industry and standards support. Typically, these types of changes occur to one-third of the standards every year. This indicted that new standards are being introduced frequently, but does not impact the stability of individual standards.

IMPLEMENTING A STANDARDS STRATEGY

If utilized inappropriately, standards can increase costs in system development dramatically. In practice, only a handful of standards will have a direct impact on software development. In particular, major standards relating to the operating system, windowing system, and networking interface are the most critical. Many other standards are useful only as general guidelines, and their importance is quite subjective.

It is important to have a standards strategy and to leverage ongoing standards activities, particularly those that are attempting to catalogue the overall environment, those that directly impact software development, and those that impact your organization's specialty areas. This strategy starts with the selection of a generic profile, such as POSIX or X/Open XPG4. Then the particular specialty areas that are not adequately addressed by the generic profiles need to be identified. The profile should include expanded detail in these high-priority areas. Technology road-mapping is a useful exercise with respect to the specialty areas—for example, prediction of future industry trends based on past technology evolution and high-potential research. Any standards gaps that are identified should become strategic targets for the creation of organization-level standards, with future potential standardization in larger forums.

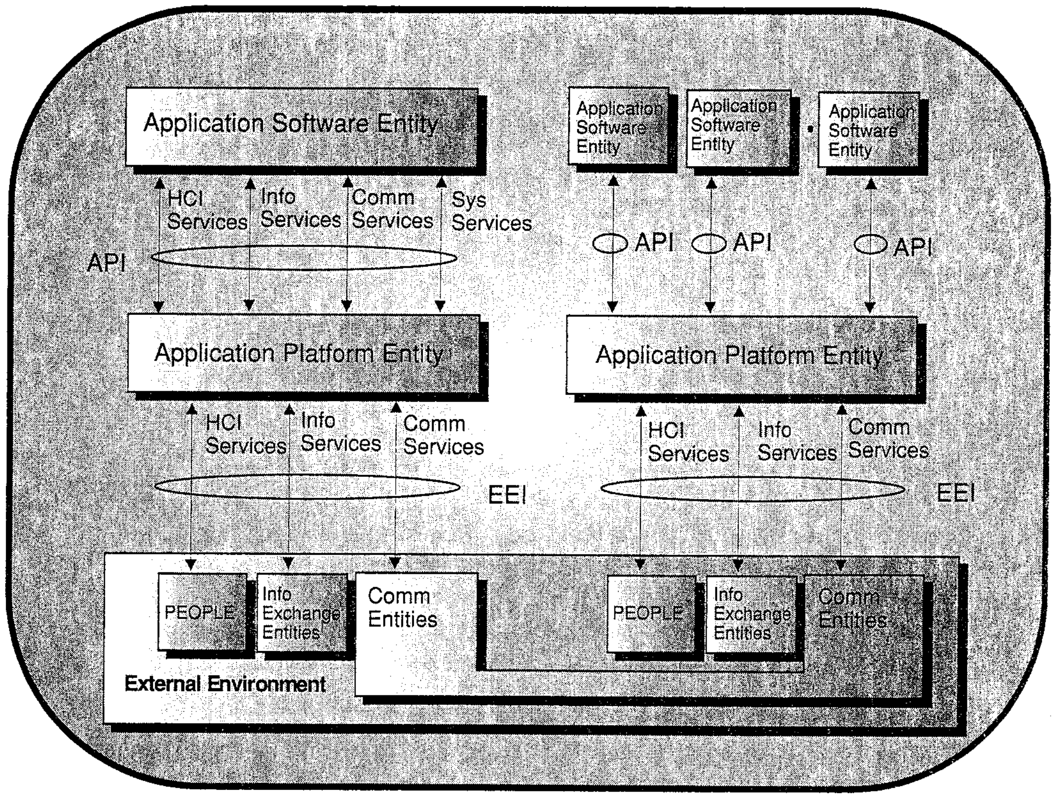


Figure 2.2. POSIX Open System Environment (OSE) reference model.

It is important to evaluate the reality of each standard and to assess what the real advantages are of its utilization. Detailed guidelines for utilization of key standards should be documented.

Standards creation and revision is a dynamic process that must be tracked. As much as a third of a comprehensive standards profile can change in the course of a year.

Standards are important to overall decision-making and development practices. Standards represent the long-term multiparty technology agreements that are the stable basis for software architectures and life cycle support.

COMMENTS

Weighing short-term needs against long-term needs is one of the most difficult problems that an organization faces. Often, decisions are based only on currently available technology and standards. While an organization may be able to answer some of its immediate needs using available market products, such decision making could be detrimental. Technology is moving at such an accelerated pace that often what is available today is already outdated by the time it is integrated. An organization must look into the future to determine its long-term needs and plan ahead using a combination of technology that is available today and a collection of emerging new standards. By using reference technologies for these new standards, an organization can experiment and collect experience that would enable it to position itself to leverage new products when they finally become available.

3

An Introduction to CORBA

Every major new distributed computing technology has held out the promise of interoperability between disparate systems and applications. Today connectivity between most types of operating systems platforms is readily available. However, interoperability at the application level remains elusive. Key factors include the inherent difficulty of distributed application programming and the lack of standard interfaces between applications. Since its introduction ten years ago, remote procedure call (RPC) technology has generated few popular interfaces, with Network File Server being perhaps the only widely used RPC interface definition.

The Object Management Group (OMG) is an industry consortium whose mission is to define a set of interfaces for interoperable software. Its first specification, the Common Object Request Broker Architecture (CORBA), is an industry consensus standard that defines a higher-level facility for distributed computing. CORBA simplifies distributed systems software in several ways. The distributed environment is defined using an object-oriented paradigm that hides all differences between programming languages, operating systems, and object location. CORBA's object-oriented approach enables diverse types of implementations to interoperate at the same level, hiding idiosyncrasies and supporting reuse. This interoperability is accomplished through well-defined interface specifications at the application level. CORBA provides a portable notation for defining interfaces called the OMG Interface Definition Language (IDL). The OMG participants are defining a comprehensive set of standard OMG IDL interfaces called object services and common facilities. Whereas CORBA simplifies the distributed software environment, object services and common facilities address the need for interface standards supporting application-level interoperability.

OBJECT MANAGEMENT ARCHITECTURE

The OMG's object management architecture is defined in a reference book which we highly recommend [OMG, 93]. 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. Common facilities are the set of shared high-level services, such as printing and e-mail. Object services are a shared set of lower-level services, such as object creation and event notification. Application objects comprise all the remaining software including developer's programs, commercial applications, and legacy systems.

Much of the OMG standards activities centers around the three architecture areas: ORB, object services, and common facilities. CORBA Version 1.1, covering ORBs, was adopted in December 1991 [OMG, 92a]. The CORBA revision process comprises upwardly compatible extensions. CORBA Version

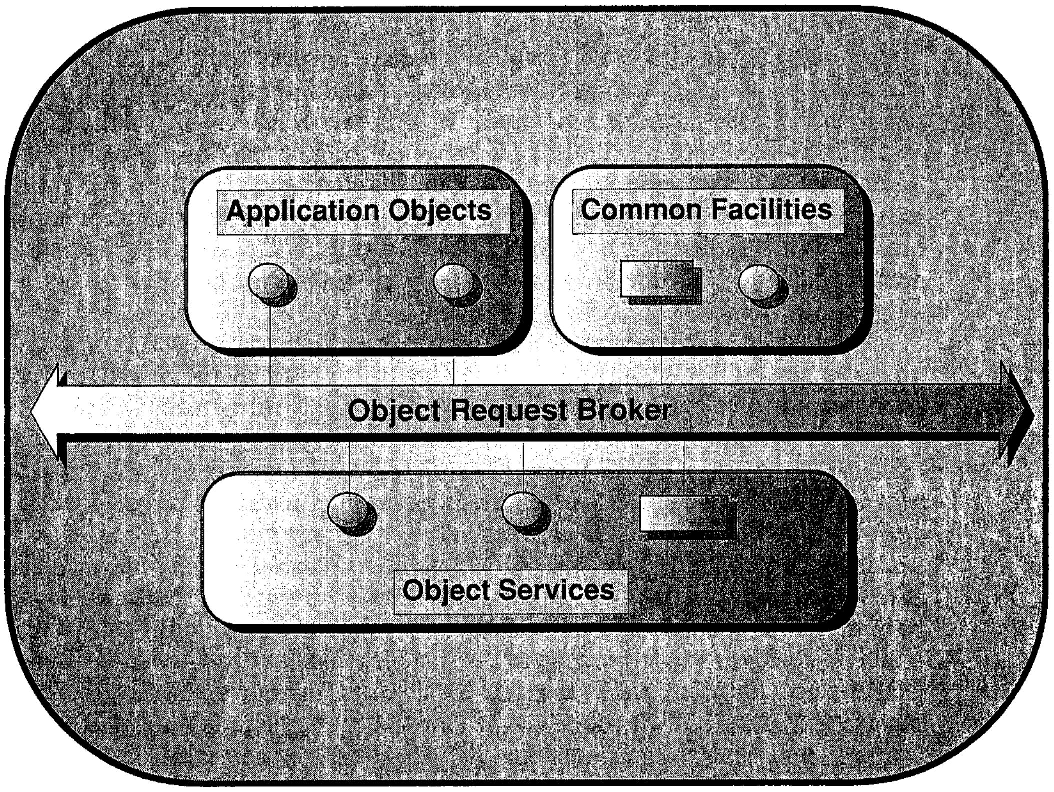


Figure 3.1. Object management architecture.

1.2, primarily a typographical revision, was adopted in June 1994. Extensions to CORBA, under the umbrella of CORBA 2.0, include ORB interoperability and further specification of the Interface Repository. Future CORBA 2.0 activities include additional language bindings such as Ada, Cobol, and Smalltalk. Approved object services specifications include event notification, object naming, and object life cycles. The OMG has released a comprehensive road-map schedule for the adoption of additional object services [OMG, 92b]. The adoption process for common facilities is under way. Whereas object services are fundamental enabling service specifications, common facilities are high-level service specifications that will provide high leverage to application developers.

Object services and common facilities supply interfaces for application-to-application interoperability as well as commercially supplied services. These standard interfaces have a dual role: Suppliers may supply implementations of standard services, but application developers also can reuse standard interfaces.

OBJECT REQUEST BROKER

CORBA is a specification for an application-level communication infrastructure. It provides communication facilities to applications through two mechanisms: static interfaces and the Dynamic Invocation Interface (DII). An Interface Repository stores on-line descriptions of known OMG IDL interfaces. Any interface can be used with either mechanism. The Basic Object Adapter (BOA) is an initial set of ORB interfaces for object implementations. CORBA also specifies a set of basic system objects, such as general purpose name-value lists.

CORBA is a peer-to-peer distributed computing facility where all applications are objects (in the sense of object orientation). Objects can alternate between client roles and server roles. An object is in a client role when it is the originator of an object invocation. An object is in a server role when it is the recipient of an object invocation. Server objects are called *object implementations*. Most objects probably will play both roles. More flexible architectures can be implemented using CORBA rather than the pure client-server architectures imposed by remote procedure calls.

Interface Definition Language

OMG IDL is a technology-independent syntax for describing object encapsulations. When used in software architectures, OMG IDL is the universal notation for defining software boundaries. In OMG IDL, interfaces that have attributes and operation signatures can be defined. OMG IDL supports inheritance between interface descriptions in order to facilitate reuse. Its specifications are compiled into header files and stub programs for direct use by developers. The mapping from OMG IDL to any programming language could potentially be supported. Vendors have implemented mappings to many languages, such as C, C++, and Smalltalk. OMG IDL mappings to many other languages are under construction. OMG IDL compilers are bundled with ORB products and allow programmers to define portable compiler-checked interfaces.

In addition to header files, the OMG IDL compiler generates stub and skeleton programs for each interface. The client program links directly to the OMG IDL stub. From the client's perspective, the stub acts like a local function call. Transparently, the stub provides an interface to the ORB that performs marshalling to encode and decode the operation's parameters into communication formats suitable for transmission. The OMG IDL skeleton program is the corresponding server-side implementation of the OMG IDL interface. When the ORB receives the request, the skeleton provides a callback to a server-supplied function implementation. When the server completes processing of the request, the skeleton and stub return the results

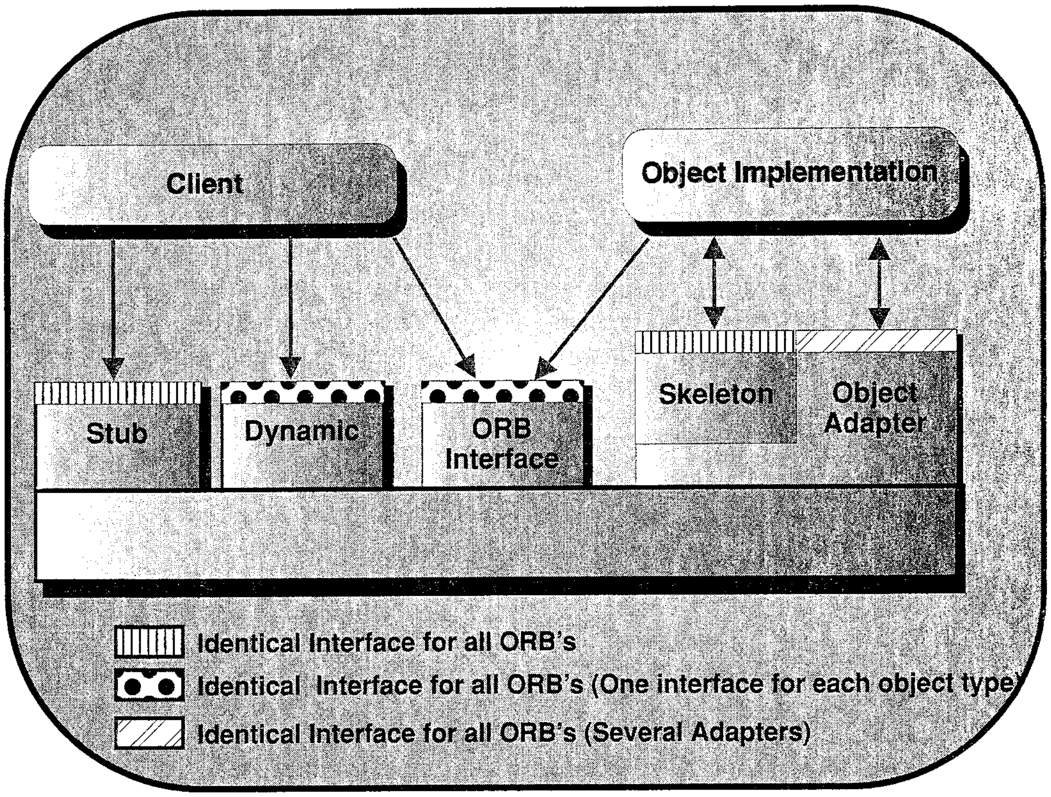

Client
Object Implementation
Stub
Dynamic
ORB Interface
Skeleton
Object Adapter
Identical Interface for all ORB's
Identical Interface for all ORB's (One interface for each object type)
Identical Interface for all ORB's (Several Adapters)

Figure 3.2. CORBA interfaces.

to the client program, along with any exception information. Exceptions can be generated by the server or by the ORB in case of errors.

The OMG IDL language is defined in CORBA Chapter 4 and the C language mapping is defined in CORBA Chapter 5. [OMG, 92a] The following is an example of OMG IDL and its mapping to the C language. This OMG IDL specification is for an interface with one operation. The interface name is example1, and the operation name is operation1. This operation has one user-defined parameter, param1. It also has clauses specifying a user-defined exception and a context expression.

```
// OMG IDL
exception USER_EXCEPTION1 { string explanation1; };
interface example1 {
    void operation1(inout long param1)
        raises (USER_EXCEPTION1)
        context( "CLIENT_CONTEXT1"); };
```

The corresponding C language mapping follows. The user-defined exception is mapped into a struct. The interface itself is a renamed type of CORBA-type Object. In the function prototype, the operation name is concatenated to the interface name with an underscore. This naming convention indicates the scoping of the names from the OMG IDL. The first three parameters are implicit parameters generated by the OMG IDL compiler. The first parameter is the implementation object handle; this indicates the destination object implementation for the example_operation1 message. The second parameter is the Environment parameter, which is an exception value returned to the client. The exception may be generated by the object implementation or the ORB to indicate operation failure. In addition to the user exception defined in the example, CORBA provides a comprehensive set of predefined exceptions [OMG, 92a]. The Context parameter is a set of attribute values specified by the client for usage by the ORB and object implementation. In general, these are a set of string values, associated with the attribute names in the context clause of the OMG IDL specification. Context is useful for the application programmer in order to pass default information about the client's environment. Following the implicit parameters is the list of user-defined parameters. For C, there is a complex set of conventions for how these parameters are passed, depending on whether the parameter is in, inout, or out and on the parameter's type.

```
/*  C Mapping */
typedef struct { char *explanation1; } USER_EXCEPTION1;
typecode Object example1;
void example1_operation1(
    example1 o,
    environment *ev,
    Context *ctx,
    long *param1);
```

The OMG IDL specification contains more documentation for the object interfaces. The header files and stub codes generated by the OMG IDL compilers are somewhat cryptic in comparison. For example, all of the following OMG IDL constructs are mapped into C struct definitions: OMG IDL structs, exception values, and union types. In addition, most OMG IDL compilers do not pass through the OMG IDL's comments. Programmers should use the OMG IDL for comprehension of the interfaces and use the compiler-generated header files as a reference for parameter handling and naming conventions.

Other OMG IDL language mappings appear quite different from the C mapping. Although the renaming of type Object does not seem to be an important factor in C, the specialization of type Object is an important factor in object-oriented languages such as C++ and CLOS, which depend on object type specialization in their runtime binding algorithms. The mapping to object-oriented languages is quite natural from OMG IDL, and the header files generally include native object class definitions corresponding to the OMG IDL interfaces.

Implementing OMG IDL Specifications

The standard specification defining OMG IDL is only 35 pages long [OMG, 92a], but it has a fundamental importance analogous to Backus-Naur Form (BNF). Like BNF, OMG IDL is a specification language. Where BNF is universally used to specify new language grammars, OMG IDL is universally applicable to the specification of APIs.

OMG IDL can be used in several ways: as built-in library interfaces, ORB interfaces, or RPC interfaces. An OMG IDL interface does not have to be used with an ORB product, or vice-versa. OMG IDL can be used separate from the balance of the CORBA specification, as a notation for specifying APIs implemented with CORBA and non-CORBA technologies.

In the built-in library interface case, OMG IDL provides the advantage of an implementation-free interface (Figure 3.3). However, the application is not distributed. The client and server are compiled and linked in as a single program. The ORB or RPC mechanism provides system distribution. The client still makes a local function call. However, the stub call is sent via the ORB or RPC to the remote skeleton function that in turn activates the server implementation (Figure 3.4).

IDL is language independent and supports multiple language mappings. Several mappings are already standardized by OMG: C, C++, and Smalltalk. When IDL is mapped into a programming language, for example, C, three arguments are generated automatically and the rest are user-defined parameters. These implicit parameters are the object handle, the environment, and the context (Figure 3.5).

The object handle specifies the handle of the server that is to be activated. The Context can contain system- or user-specific properties that can

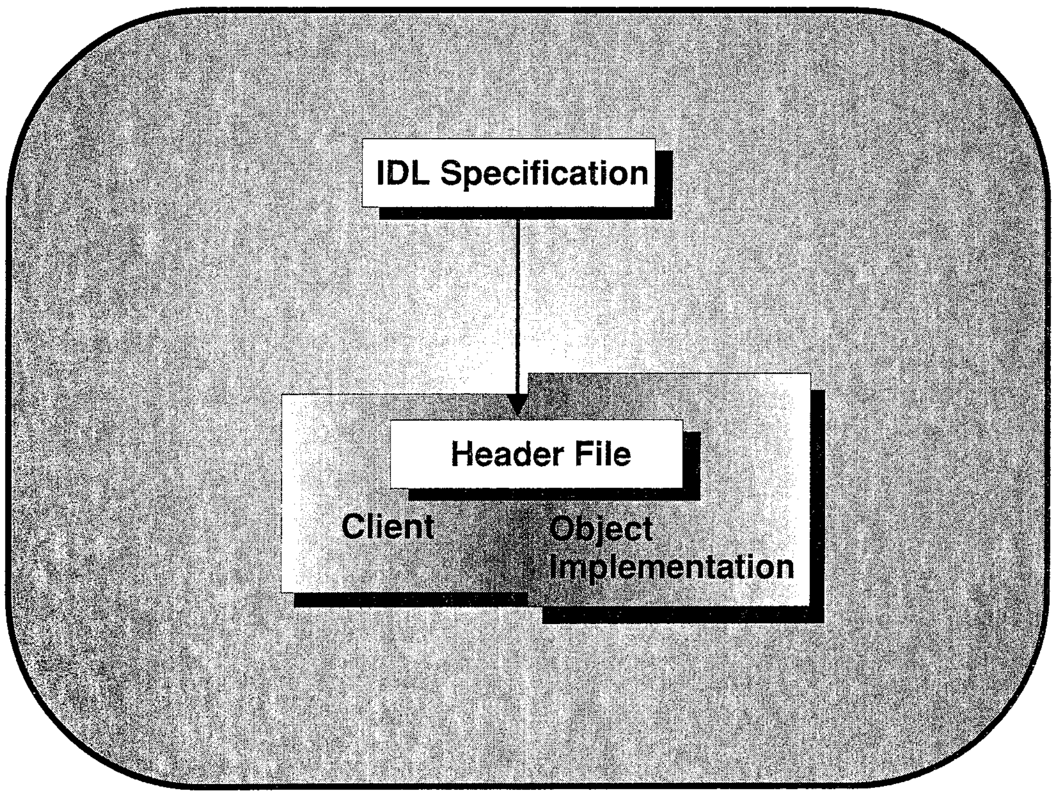


Figure 3.3. Built-in library interface: same compilation module.

impact ORB decisions or information for the server, such as window system preference, that may not be suitable for argument passing. Context objects contain a list of properties that consist of a name and a string value. Like environment variables, the properties represent information about the client, environment, or information about a request that can be passed as parameters [OMG, 92a].

Clients can pass this information to the server, which in turn can query about them. The information can be used for policy and binding decisions. Context objects may be chained. CORBA defines operations for creating, deleting, setting values, and getting values of properties.

The environment variable is provided to return exception information. The environment type is partially opaque and includes a major error type, minor code, and error identification string components. The major error type indicates whether an exception occurred, a system exception occurred, or a user exception was set. The minor code is a value that identifies the particular exception in a form easily utilized by programmers in a case statement. The identification string is a human-readable explanation of the

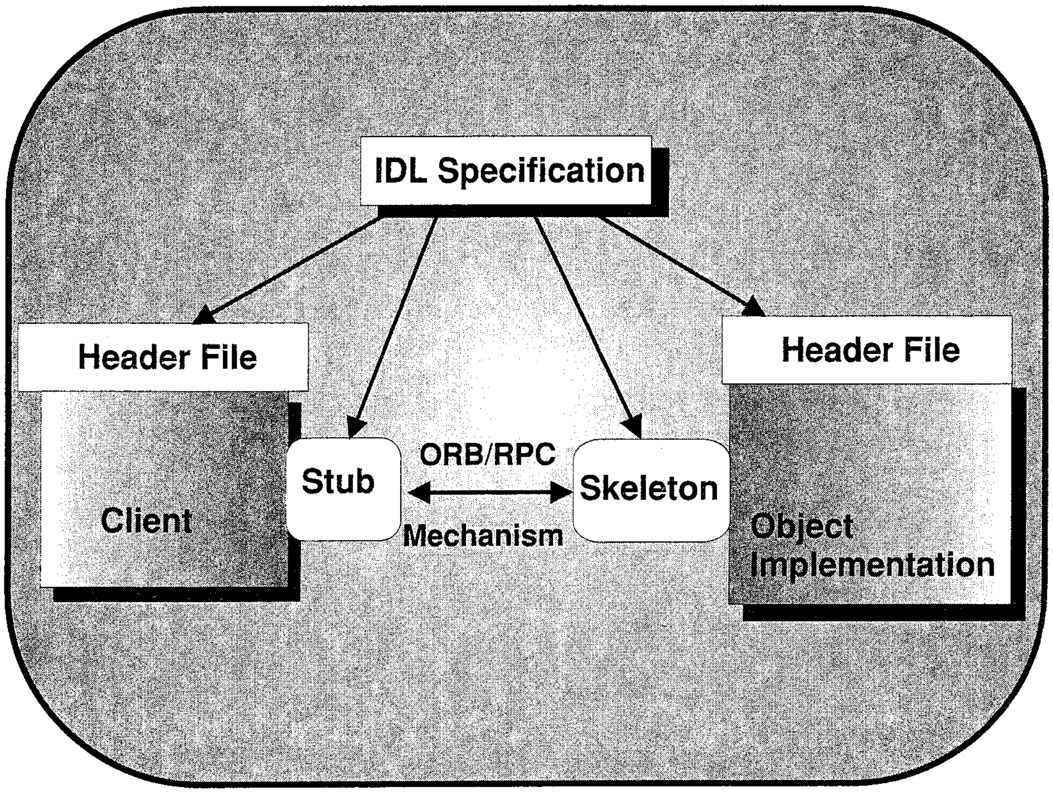


Figure 3.4. ORB or RPC mechanism.

error. CORBA defines a comprehensive set of standard exception types that can be returned by the ORB and by application object implementations. The predefined exceptions should be used wherever applicable.

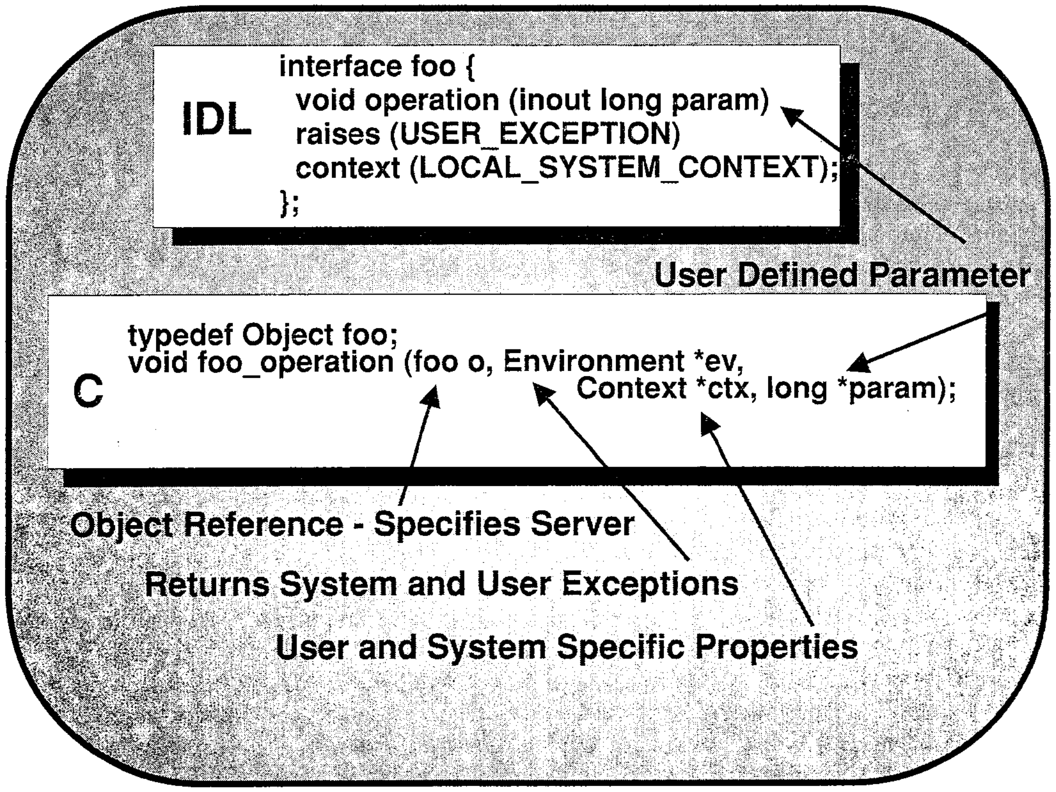
For special exceptions unique to a particular architecture, user exceptions can be readily defined in OMG IDL. User exceptions contain the basic environment attributes, and user-defined properties can be added to the exception structure. Exception handling is an important aspect of software architecture specification that is well supported by OMG IDL.

Dynamic Invocation Interface

Early ORB products were based almost entirely on the DII, which is an alternative to compiled OMG IDL static interfaces. The DII is a generic facility for invoking any operation with a runtime-defined parameter list. A runtime interface description of the operation signature can be retrieved on-line from the CORBA Interface Repository. Using the metadata, a legal request can be constructed to a previously unknown operation and unknown object type. Use of the DII instead of an OMG IDL static interface is transparent to the object implementation. In general, programming with OMG IDL static interfaces is much simpler and results in more robust code for the developer. However, the DII provides a level of flexibility that is necessary in some applications, such as desktops and operating systems.

Object Adapters

An *object adapter* comprises the interface between the ORB and the object implementation. Object adapters support functions such as registration of object implementations and activation of servers. There are many potential types of object adapters. There could be different adapters for general-purpose uses, for object database integration, for legacy integration, and so forth. CORBA 1.1 defines only the Basic Object Adapter (BOA), but it recognizes the need for these other types of adapters.



```
interface foo {
  void operation (inout long param)
  raises (USER_EXCEPTION)
  context (LOCAL_SYSTEM_CONTEXT);
};
```

```
typedef Object foo;
void foo_operation (foo o, Environment *ev,
                                     Context *ctx, long *param);
```

IDL
C
User Defined Parameter
Object Reference - Specifies Server
Returns System and User Exceptions
User and System Specific Properties

Figure 3.5. Example of IDL-to-C mapping.

The BOA is a general-purpose object adapter. When a client request specifies an inactive server, the BOA automatically activates the server process. The first responsibility of the server is to register its implementation with the BOA. The BOA stores this registration to use in future object requests. After an object is activated, it may receive client requests through the method callbacks in the OMG IDL skeleton. BOA services include exception handling and object reference management, among others.

CORBA Acceptance

CORBA and OMG IDL have gained wide acceptance by industry and consortia. OMG membership exceeds 500, including virtually all platform manufacturers and major independent software vendors. X/Open copublishes the CORBA specification and has included CORBA in the X/Open Portability Guide Release 4. OMG IDL is the Application Program Interface (API) specification language for the next release of X Windows, called X11R6 FRESCO. This makes X11R6 applicable to multiple programming languages instead of being tied exclusively to C. CORBA also is supported by specification of groups, such as Petroleum Open Systems Consortium and the Open Geographic Information System (GIS) Foundation.

CORBA acceptance in the U.S. government is growing rapidly. The Department of Defense Defense Information Systems Agency (DISA), the National Security Agency (NSA), the National Institute of Standards and Technology (NIST), the National Institute of Health (NIH), and MITRE have joined the OMG, and CORBA is included in several government standards profiles.

Product Availability

CORBA is a future standard of the Common Open Software Environment (COSE), an industry alliance including companies such as SunSoft, Digital Equipment, IBM, Hewlett-Packard, Novell, and Santa Cruz Operation (SCO). Successful elements of COSE are merging with the Open Software Foundation (OSF). All of these vendors have committed to deliver CORBA-compliant products, and CORBA will be bundled with most of their existing operating systems.

Several vendors are already delivering productized CORBA implementations including: IONA, DEC, HP, and IBM. Apple has announced future support for CORBA. Microsoft is supporting CORBA through its alliance with DEC in a cross-platform development product called the Common Object Model. DEC will support two APIs for developers, CORBA and the Microsoft Common Object Model, a more primitive alternative to CORBA. DEC and others have productized CORBA interfaces to Microsoft OLE. Microsoft is spearheading an OMG standards initiative to provide a standard definition of CORBA/Common Object Model (COM) interoperability. Available CORBA implementations cover most operating systems environments including Solaris, OS/2, AIX, HP-UX, DG UNIX, Virtual Memory System (VMS), OSF/1, IRIX, Macintosh, Windows, and Windows-NT.

Third-party vendors, such as Integrated Computer Solutions, WordPerfect, and Paragon Imaging, have announced future CORBA-compliant applications products. Expersoft, FORTE, and other companies are developing CORBA interfaces to major database products. As the common facilities standards are released, we anticipate direct CORBA support from most major independent software vendors. These companies are already actively involved in the OMG standards processes.

Underlying CORBA

CORBA is an abstract specification that does not constrain its underlying implementation. The ORB could be implemented as a linked library function interface, a layer directly over RPC, or as a higher-level communication facility. This flexibility allows vendors to utilize their existing networking facilities. Some vendors (such as IONA), are basing CORBA on ONC-compatible RPCs; some vendors (such as HP), are using OSF DCE; and some (such as Sunsoft) are bypassing the RPC layer and implementing CORBA at lower layers. Most vendors also provide developer support for implementing linked library code using OMG IDL interfaces.

The inherent flexibility underlying CORBA allows the software architect to separate design from implementation decisions. Relevant implementation decisions include process allocation and performance trade-offs. Because CORBA makes these implementation properties transparent, it is unnecessary to hard-code these details into the architecture design. This frees the designer to use OMG IDL to specify all architectural software boundaries and then choose the underlying communication mechanisms later. Many implementation decisions can be deferred until installation time, so system adaptability is maximized.

Today's CORBA products are like Ethernet products in the early days; the vendors are building to a common standard, and they are actively working toward interoperability with CORBA 2.0. Using earlier CORBA versions, if you want to build a multiplatform ORB application, you must choose an ORB vendor that runs on all the target machines. When CORBA products are bundled with operating systems, the CORBA 2.0 standard will enable multivendor ORB implementations to interoperate transparently. Most ORB vendors have announced and demonstrated interoperability between their platforms.

CORBA 2.0

CORBA 2.0 is a set of upwardly compatible standards that complete and augment the CORBA 1.2 specification. The key elements of CORBA 2.0 are:

- ORB-to-ORB interoperability specification
- Client and server initialization specification
- Additional programming language bindings: C++ and Smalltalk
- Interface repository specification

Among these, the interoperability specification is the most significant for the CORBA market. The specification provides for cross-ORB services between multiple vendor's products. The OMG adopted the Combined Submission for Interoperability, which is a merger of the major submissions. The core of the specification is a general architecture for interoperability called Universal Networked Objects (UNO).

Within UNO, there are two types of interoperability specifications: General Interoperability Protocols (GIOPs) and Environment Specific Interoperability Protocols (ESIOPs). The GIOPs are fully specified protocols that are mandated to provide for out-of-the-box interoperability. An initial GIOP called Internet Interoperability Protocol (IIOP) was included in UNO. IIOP is based on Transmission Control Protocol/Internet Protocol (TCP/IP), the ubiquitous protocol of the Internet. IIOP is a simplfied subset of an RPC mechanism that is specifically directed at ORB-to-ORB interoperability. The initial ESIOP specified within the Combined Submission is based on the Open Software Foundation's (OSF's) DCE, called the DCE Common Interoperability Protocol (DCE CIOP). Vendors can support ESIOPs and still be CORBA-compliant, as long as their products also support the GIOP. New ESIOPs and GIOPs can be added to the specification later through OMG technology adoption processes.

It is interesting to note that the choice of ORB interoperability solutions does not impact application software. Interoperability is an issue between the ORB vendors, not between vendors and users. Few, if any, application developers will ever be involved in programming with GIOPs and ESIOPs. Now that the ORB interoperability standard is in place, vendors can proceed to implement interoperable products with minimal risk. This standard also reduces risks for CORBA users, and it will soon provide for ubiquitous interoperability between ORB products.

The CORBA 2.0 initialization specification has much more direct impact on developers. The initialization specifications resolve important portability issues for application software. Initialization defines how clients and servers using CORBA establish initial communication with the ORB and obtain initial object references, such as the naming service handle. With CORBA 2.0 initialization, the interfaces and calling sequences will be consistent and portable between ORB products.

The language bindings for C++ and Smalltalk were standardized in CORBA 2.0. These bindings define how OMG IDL specifications are mapped into the API specifications of these programming languages. The bindings will provide consistent, natural support for these languages across ORB products. These bindings also provide for seamless interoperability between objects written in different languages. Several ORB products are supporting multiple language bindings.

The Interface Repository specification standardizes the access and management of on-line metadata describing all known OMG IDL interfaces. The CORBA 2.0 Interface Repository completes the specification in CORBA 1.2, which defined the basic retrieval interfaces.

OBJECT SERVICES

Object services comprise a fundamental set of system service interfaces. The adopted OMG Object Services comprise the Common Object Service Specification (COSS). COSS is a multivolume series, with one volume corresponding to each Object Services Request for Proposal (RFP). To date, the OMG Object Services Task Force has released four of the five planned RFPs: RFP1, RFP2, RFP3, and RFP4.

The RFP1 services include:

- Object Event Notification Service
- Object Life Cycle Service
- Object Naming Service
- Object Persistence Service

The RFP2 services include:

- Object Concurrency Service
- Object Externalization Service
- Object Relationships Service
- Object Transaction Service

The RFP3 services include:

- Object Security Service
- Object Time Service

The RFP4 services include:

- Object Licensing Service
- Object Properties Service
- Object Query Service

Each RFP takes about a year to complete the process and results in technology adoption. RFP1 and RFP2 have already been adopted at the time of this writing. RFP3 and RFP4 processes will be complete in 1995. RFP5 is due to be released in mid-1995 and complete in 1996.

The RFP5 services will likely include:

- Object Change Management Service
- Object Collections Service
- Object Trader Service
- Object Startup Service

It is anticipated that the International Standards Organization (ISO) will submit it's OMG IDL binding to the Open Distributed Process (ODP) Trader Service, which is in advanced stages of formal standards adoption.

The RFP5 services are the final services identified in the Object Services Architecture [OMG, 94]. Thus, RFP5 will complete the Object Services Architecture. By the end of RFP5, the standardization processes for both CORBA and Object Services will be essentially complete.

The first services to be standardized included the Object Naming Service and the Object Event Notification Service. The Naming Service enables the retrieval of object handles based on the string-valued name of an object server. Hierarchical naming contexts can be defined for groupings of objects. The Event Notification Service is a general facility for passing events between objects. Event interfaces are defined for administration and alternative forms of event posting and retrieval.

Servers that implement COSS interfaces can be constructed by the ORB vendor or by application developers. Allowing developers to build services is a very useful feature of the object management architecture called *generic objects*. Generic objects that reuse standard interfaces can interoperate transparently with any application supporting the standards and can be tailored for specific application needs. Generic objects can provide a common interface layer on top of noncompliant services. For example, Naming Service interfaces could be layered on top of various directory standards (such as DCE Cell Directory Service, X.500, or Internet naming) to provide simple consistent access to multiple services. Object services interfaces are designed to be extended through specialization. For example, a developer might specialize the standard event interface with facilities for a real-time application, while retaining interface compatibility with standard event clients and servers.

Other services adopted from RFP1 include the Object Life Cycle Service and the Object Persistence Service. The Life Cycle Service comprises operations for managing object creation, deletion, copy, and equivalence. The Persistence Service comprises facilities for storage of objects.

Some of the most commonly used object services, the Object Naming Service, the Object Event Service, and the Object Relationship Service, are summarized in the next sections.

Object Naming Service

The basic operations for accessing the Object Naming Service include bind, unbind, and resolve. Bind adds a (name, object handle) pair to a naming context. Unbind removes the pair. Resolve retrieves an object handle given a name. Some basic operations for accessing Naming contexts include: new_context, bind_context, and destroy. These correspond to creating a context, associating two contexts hierarchically, and removing a context.

Figure 3.6 is an example of a set of names and naming contexts. Names are defined as an OMG IDL sequence type, which allows the description of a hierarchical list of identifiers without imposing implementation-specific syntax. This is useful because the major directory standards all use different hierarchical naming syntax.

The naming service is one of the most basic and generally useful services. CORBA users should expect to have an Object Naming Service bundled with the ORB and layered over the native directory service. Most CORBA-based programs should use the naming service to locate other basic services, such as the Object Trader Service.

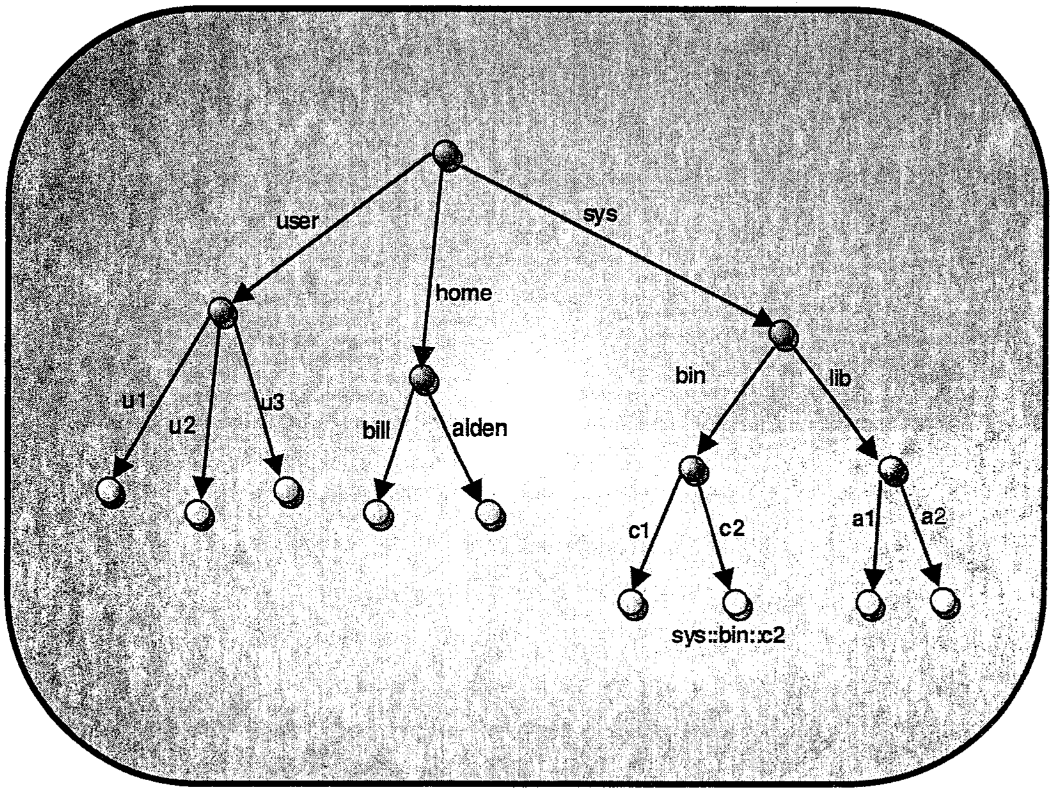


Figure 3.6. Naming contexts.

Object Event Service

The Object Event Service is a general-purpose, reusable set of interfaces for event posting and dissemination. The roles of the objects involved in event notification include suppliers, consumers, channels, and factories. The suppliers and consumers are usually application objects, and the event channel and the event channel factory provide the event services. Event channel interfaces operate in either push mode or pull mode. The IDL Consumer interface provides the push() operation. In push mode, a supplier can push() an event to the event channel object, and a consumer application will receive a corresponding push() invocation from the event channel object. Alternatively, applications may utilize the IDL Supplier interface, which provides pull() and try_pull() operations. The try_pull() operation polls for ready events, and the pull() operation retrieves the event. Using the administrative interfaces, applications dynamically register themselves as consumers indicating their interest in receiving event notification. The event channel

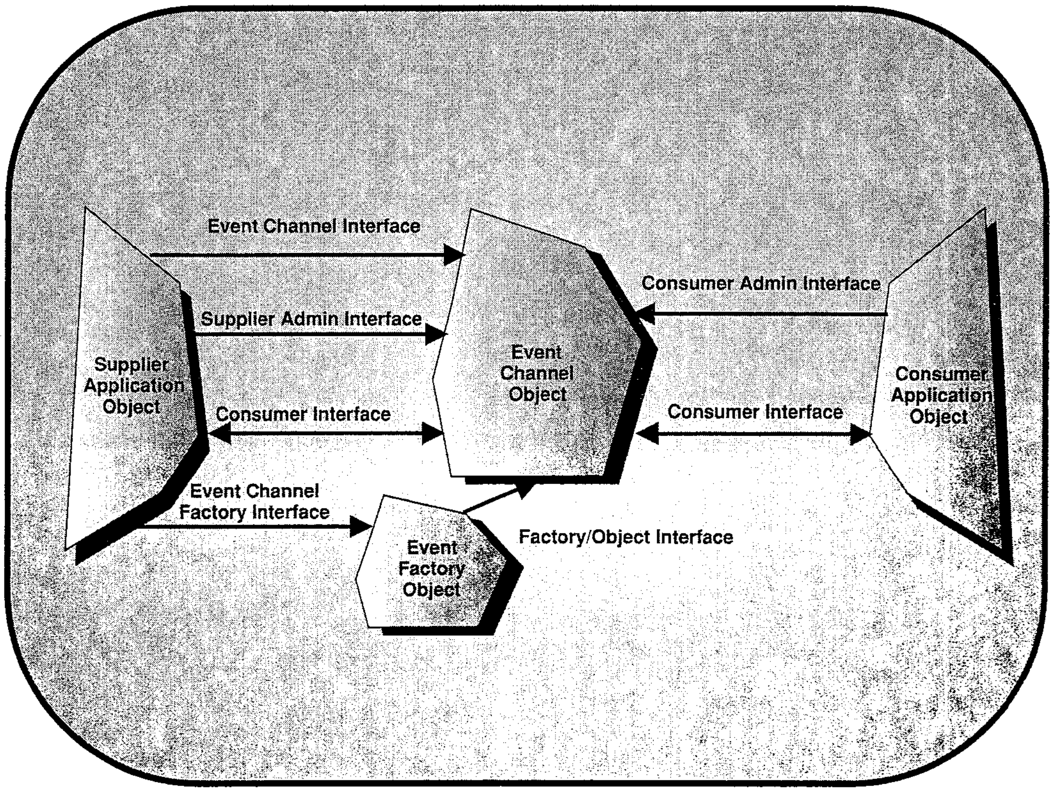


Figure 3.7. Event notification.

can support both push mode and pull mode interfaces with both supplier and consumer applications. This provides flexibility in selecting the event channel integration approach.

Object Relationship Service

The Object Relationship Service provides a capability for managing associations and linkages between objects. The service utilizes dedicated relationship objects that retain the object handles of the associated objects. Relationships are a fundamental service that can be used to implement many types of object linkages. For example, relationships can be used for desktop object linking and embedding. The relationship service is useful in combination with other services such as events, life cycle, and externalization.

The Object Properties service is a facility for attaching dynamic information to an object. A set of properties of any type can be attached to an object without changing the object's implementation. Among other uses, properties could be very useful for managing desktop objects, such as attaching icon representations to arbitrary objects.

COMMON FACILITIES

Common Facilities is the newest area of OMG standardization. Whereas CORBA and Object Services standardize the enabling infrastructure and services, Common Facilities represents higher-level specifications that complete the OMG's vision for interoperability. Now that CORBA 2.0 is finalized and most Object Services are standardized or near adoption, the OMG's focus is moving toward the standardization of the higher-level Common Facilities.

Common Facilities will provide richness and application-level focus to the ensemble of OMG technologies. Whereas ORB and Object Services are fundamental technologies, Common Facilities extend these technologies up to the application developer and independent software vendor level. Common Facilities may become the most important area of OMG standards because it is the level that most developers will utilize.

Common Facilities include specifications for higher-level services and vertical market specialty areas. Horizontal Common Facilities are application domain independent. Some examples include system management and compound documents. In addition, there are more specialized Vertical Market Facilities, such as geospatial data processing and financial services. Common facilities is an appropriate area for standards providing interoperability between independent software vendors' products.

The Common Facilities Task Force (CFTF) is the third permanent OMG Task Force. It complements the areas addressed by the ORB Task Force and the Object Services Task Force (OSTF). Common Facilities relates to the other OMG technology areas in that it is closer to the application level.

Ideally, the Common Facilities will include many specializations of the Object Services, specializations that extend the primitive Object Services into richer interfaces that directly address the needs of many applications and independent software vendors. Both Object Services and Common Facilities are intended to be reused by developers in application software and commercial software products.

The CFTF was created in December 1993 by vote of the OMG Technical Committee. CFTF is following the precedents established by the OSTF. The task force has released an industrywide request for information (RFI) and received many responses. The RFI responses are the source material for three key documents: the Common Facilities (CF) Architecture, the CF Road Map, and the first CF Request for Proposal (RFP). The Common Facilities Architecture identifies and describes the major categories of Common Facilities. The Common Facilities Road Map groups the CF categories by priority and establishes a schedule for facilities adoption through the RFP process. The first Common Facilities RFP (Compound Documents) initiates the technology adoption process by soliciting the first set of facilities specifications.

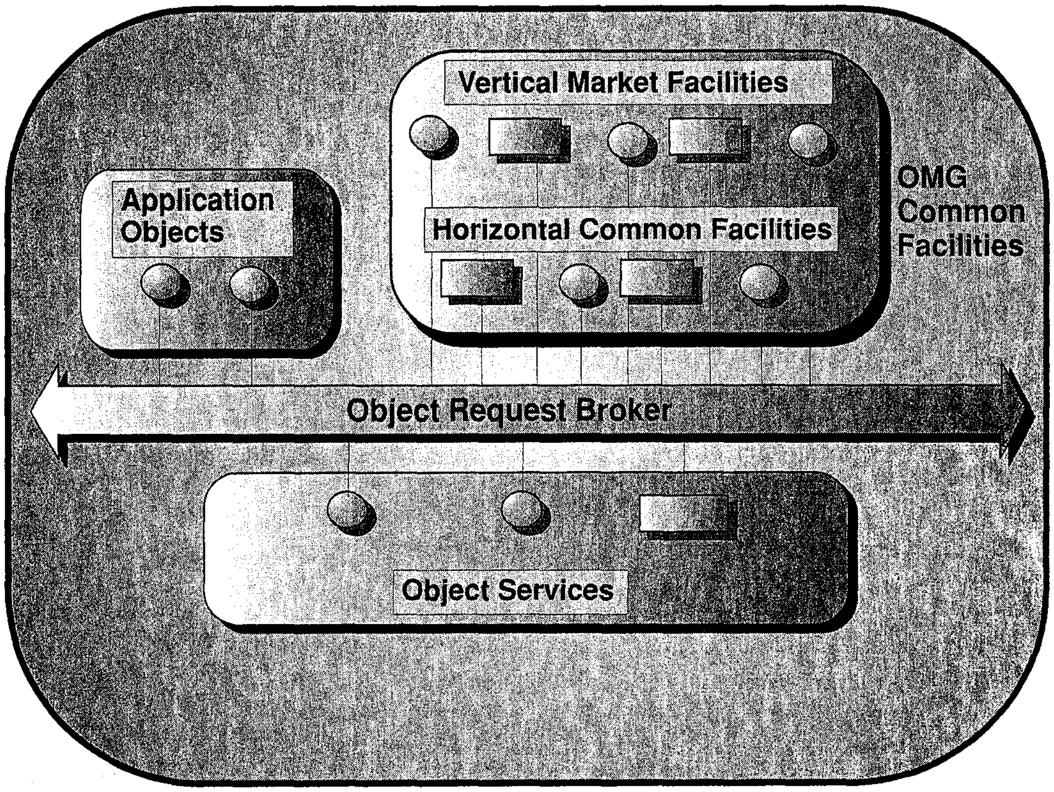


Figure 3.8. *Common facilities in object management architecture.*

In comparison with the other task forces, Common Facilities has a more diverse charter. Many Common Facilities probably will be important only to particular specialty markets. The OMG has a fast-track Request for Comment (RFC) process that can be used for technology adoption in areas where industry consensus already exists. For those high priority areas that require an RFP process, the CFTF has documented the schedule for RFPs in the Common Facilities Roadmap. [OMG, 95b]

As development budgets increasingly require system life cycle extension, reengineering, and multisystem consolidation, Common Facilities will support system migration. For application developers, standards are needed that mitigate long-term risks and increase leverage from commercial technology. For commercial software developers and R&D innovators, standards are needed that create entry points in applications systems for value-added products. We envision Common Facilities bridging these needs, by establishing standards that are mutually supportable by both application developers and independent software vendors.

COMMENTS

ORB technology is an infrastructure technology area that is near maturity. Many CORBA implementations are available today, and platform vendors are investing substantially in bundled CORBA products. (See Appendix.) CORBA will be ubiquitous in the UNIX market, with bundled implementations on most platforms. CORBA also will be widely available on volume platforms (Apple, Windows), bundled with packages (such as OpenDoc) from major software vendors such as Claris, WordPerfect, Lotus, and Taligent.

Because it is independent of computer language and operating system, OMG IDL is universally applicable. One does not have to have an ORB or even use CORBA in order to obtain the benefits of the OMG IDL. OMG IDL compilers can be used independently from other parts of the system in order to allow the system architect and software developers to produce clean, well-defined interfaces.

There are many ways to prepare for the impending technology transition to CORBA. Most ORB vendors and some independent consultants offer comprehensive training courses covering CORBA products that offer insight into the standard and the technology. The OMG has published the Object Management Architecture Guide [OMG, 93], the CORBA Specification [OMG, 92a], and the Common Object Services Specification [OMG, 94b]. These publications are John Wiley & Sons books that are generally available. Finally, OMG IDL is a stable language that can be used for structuring architectures today. OMG IDL is the specification language for the next generation of X Windows, X11R6 Fresco from the X Consortium. A public domain OMG IDL compiler toolkit is available from the OMG (via ftp from omg.org); the toolkit readily supports OMG IDL syntax checking and can be extended for additional applications of OMG IDL.

4 Software Architecture Design

The important decisions in design are not what to put in but what to leave out.
—Attributed to Tony Hoare by Per Brinch Hansen, U.S.C.

There exists a range of alternative solutions for most application problems. It is well known that object-oriented design processes do not yield unique solutions. Design alternatives will vary in cost effectiveness. Designers should be aware of the alternatives, and cost should be a key consideration in design trade-offs.

When we talk about design alternatives in this chapter, we are discussing trade-offs in the design of software architectures. There is no consensus in the computing literature on the definition of the phrase software architecture. In our view, software architecture defines the boundaries between the major components comprising a software system. The boundaries are interface specifications that can be expressed in Object Management Group Interface Definition Language (OMG IDL) with appropriate sequencing constraints and semantics. Sequencing constraints define the conventions for the ordering of operation invocations. The semantics define the operation meanings (usually in prose). These definitions are comparable to a typical OMG specification. By defining software architecture in this way, OMG standards can be applied directly to application software architectures, and elements of high-quality architectures can be migrated into standards.

There are many other issues beyond software architecture in the design and implementation of an application system. We categorize the other relevant design information as comprising the system architecture; everything else is captured in the system implementation. Software architecture is a subset of system architecture; software implementation is a subset of system implementation. The software implementation comprises the major subsystems, which we sometimes call applications or components.

We use the terms software architecture and framework interchangeably. An architecture is also a collection of frameworks, where at a minimum a framework comprises an Application Program Interface (API) and sequencing constraints. In our view, a framework also includes metadata and other interoperability conventions, such as supported data formats.

The design concepts that we present are applicable directly at the level of the software architecture design. We avoid presenting higher-level issues requiring formal Computer Aided Software Engineering (CASE) methodologies and notations; instead we refer readers to one of the many existing methodologies [Hutt, 94]. In this chapter we focus on design at the OMG IDL level, a level that is language and methodology independent but has a straightforward mapping to both of these other levels. An increasing number of useful OMG IDL standards are available and provide many useful design patterns.

Typically, about 70 percent of the cost of a software system is incurred for operations and maintenance (O&M) after the system is operational [Horowitz, 93]. Of the O&M expense, about two-thirds is due to system extensions needed because of changes in requirements. These figures indicate that *adaptability* is the key characteristic of cost-effective software architectures. Our discussion of software architecture focuses on system-level strategies that can minimize these substantial costs by enhancing system adaptability.

Savings due to having well-structured software architectures versus unstructured architectures typically exceed more than 50 percent [Horowitz, 93]. When using CORBA-based software architectures, we believe the savings can be even greater. CORBA has simplified the creation of good software architectures by providing architecture notations such as OMG IDL, useful API standards, and the inherent system-level flexibilities built into Object Request Brokers (ORBs).

This chapter covers many effective design elements of highly adaptable systems. Perhaps the most powerful (but underutilized) concept for building adaptable systems is *metadata*. Metadata is any self-descriptive information contained in the software architecture and implementation. Typically, metadata describes attributes of the components of the architecture, so that it enables more adaptability, by allowing certain attributes to be determined at runtime. Many helpful metadata standards exist to aid in the implementation of metadata knowledge that a wide community of programmers can utilize. Metadata is the key to resource discovery in distributed, adaptable systems. It enables the system to configure itself, and to adapt to system extensions and changes automatically.

ESTABLISHING ARCHITECTURAL VISION

There are two major types of organization for designing software architecture: design by individual "chief" architects and design by teams. Individuals can design conceptually coherent and elegantly simple architectures. Most successful architectures are based on a strong architectural vision. The vision is generally due to one chief architect, and it is difficult to maintain a consistent vision across teams of designers.

Communication of the architectural vision to the team of developers is a very important element of a successful architecture. About half of software development activities involve system discovery, or trying to understand the system's structure. If developers do not understand the architectural vision, then they can easily make implementation choices that violate architectural assumptions.

With no direct relationship between separate design processes and implementation processes, the implementations rapidly diverge from the designed architecture. CORBA provides some important tools, such as OMG IDL, for communicating design information in a way that can be verified in the system implementation. Communication of vision must go beyond OMG IDL to include other documentation. Creation of a tutorial-format architectural walkthrough is an excellent way to communicate vision to a team of developers. Capture the tutorial on videotape, and use it as a mandatory part of each new developer's and maintainer's training.

Design by teams is an approach used more widely than design by an individual architect. Team design can result in robust architectures that incorporate the contingencies envisioned by a whole group of architects. Many team designs are large and complex because it is often easier to include additional complexity than to compromise and merge viewpoints. Design complexity exacerbates the problem of maintaining a consistent architectural vision that developers understand.

Both approaches require iterative design and revision. An effective architecture can seldom be designed in a top-down manner with a priori knowledge. To yield architecture benefits, an architecture design needs to be prototyped, then utilized and updated through experience and knowledge gained in the prototyping process. This process is most effective if changes are made on a predictable release cycle, where the architecture is stable between releases. Architecture revision is essentially a code cleanup task on a systemwide level. As experience is gained with an immature architecture, lessons learned can be used to create a new improved architecture that will reduce O&M costs for future system extensions.

ROLE OF CORBA IN SOFTWARE ARCHITECTURE

CORBA is an enabling infrastructure for good software architectures. Its architecture benefits are derived from two primary sources: OMG IDL and the CORBA-based ORBs.

OMG IDL is an important notational tool for software architects. Because it contains no implementation information, it provides a clean separation between design and implementation. In the developed system, this provides encapsulation of components and isolation between subsystems. Component isolation is an important property of a good software architecture because it enables the reconfiguration and replacement of components during the system life cycle.

OMG IDL is a universal notation for specifying APIs. It can be used without a commercial ORB product; for example, it can be used as a layer directly on top of the Open Software Foundation Distributed Computing Environment (OSF DCE). OSF has provided a guideline for this mapping in its CORBA interoperability proposal [OSF CORBA 2.0]. APIs denote system-level software boundaries. Defining good software boundaries is the primary goal of a good software architecture. OMG IDL is the best standard notation available for this purpose. It defines APIs concisely and rigorously, covering important issues such as error handling.

By using OMG IDL for specifying all architectural boundaries, good software architectures support a uniform abstraction layer with uniform access to services. Where to place OMG IDL interfaces and where to use a commercial ORB are two different decisions. The former is a design decision; and the latter, an implementation decision. OMG IDL can support library function interfaces just as well as distributed objects across a network. CORBA allows the deferral of many allocation decisions to installation time.

OMG IDL can be layered on top of virtually any communication layer. Many different communication layers are available from standard, de facto, and proprietary sources, and they are rapidly evolving; software architects should consider an approach that isolates the application software from these underlying layers. Direct integration of application software puts the architecture at risk of obsolescence as the technologies evolve and are replaced with higher-level layers. There are successor technologies on the horizon for virtually every communication layer available today, such as DCE, ToolTalk, and so forth.

An alternative approach is to define a custom API layer encapsulating the communication-layer software. Commercial vendors use this technique routinely to enable them to substitute mechanisms for different system builds. If this custom API layer is defined in OMG IDL, then the API specification can support multiple language bindings and be easily upgraded to utilize an ORB product. This technique also can be used to mask differences between layers if multiple communications layers must be supported within an architecture. In this case, the OMG IDL defines the uniform service access layer, independent of mechanism.

Stability is a key characteristic of good software architectures. No specification is more stable than a commercial or de facto standard with multivendor support. Among standards, OMG IDL is a very stable standard because

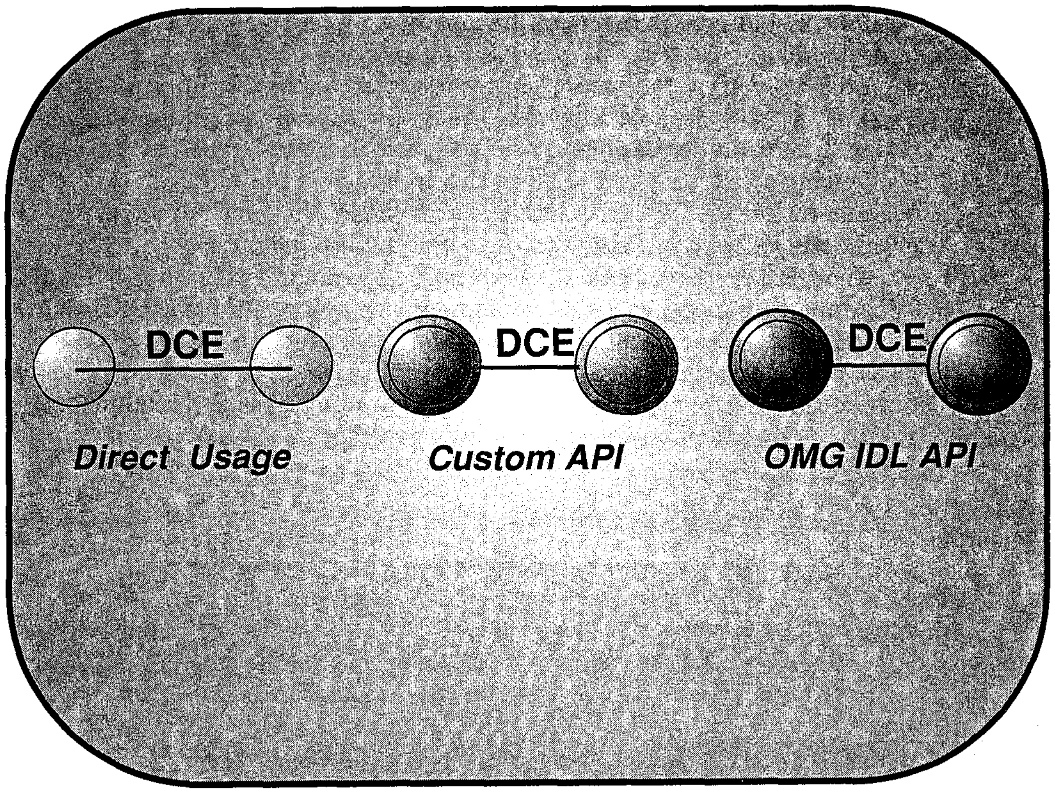


Figure 4.1. Three ways to utilize a low-level communication layer.

it is the basis for the OMG's standards process and many commercial ORB products. An adopted OMG standard comprises an OMG IDL specification and a description of the sequencing constraints on the interfaces. By using OMG IDL, software architectures gain these same benefits: conciseness, rigorousness, stability, and commercial support.

In addition, a reusable software architecture specified in OMG IDL is an informal standard for an organization. High-quality architectural specifications that address industrywide interoperability problems can be upgraded to standards. OMG IDL supports the migration of a specification from software architecture to organizational reuse to industry standard. This process may eventually enable general interoperability between applications.

OMG IDL provides a compilable linkage between the software architecture and the implementation. As OMG IDL is pure design information, it is part of the software architecture design. OMG IDL also is compilable to header files and stub programs, which the compiler uses to verify architectural constraints in the application software. These architectural constraints include enforcement of encapsulations, parameter type checking, and so forth.

This approach is a dramatic improvement on analysis and design processes that provide no compilable linkage between design and implementation. The compiler cannot check some architectural constraints, such as sequencing constraints. Therefore, the approach is not bulletproof; the programmer's knowledge of the architectural vision is needed to incorporate the architecture's benefits fully into the software implementation.

The ORB provides benefits for software architectures by supporting flexibility and transparency in the implementation. For example, CORBA's location transparency eliminates the need for direct code dependency on object location and enables the relocation and replication of services after the system is coded. Its use of automatic server startup greatly simplifies client software's involvement in server administration.

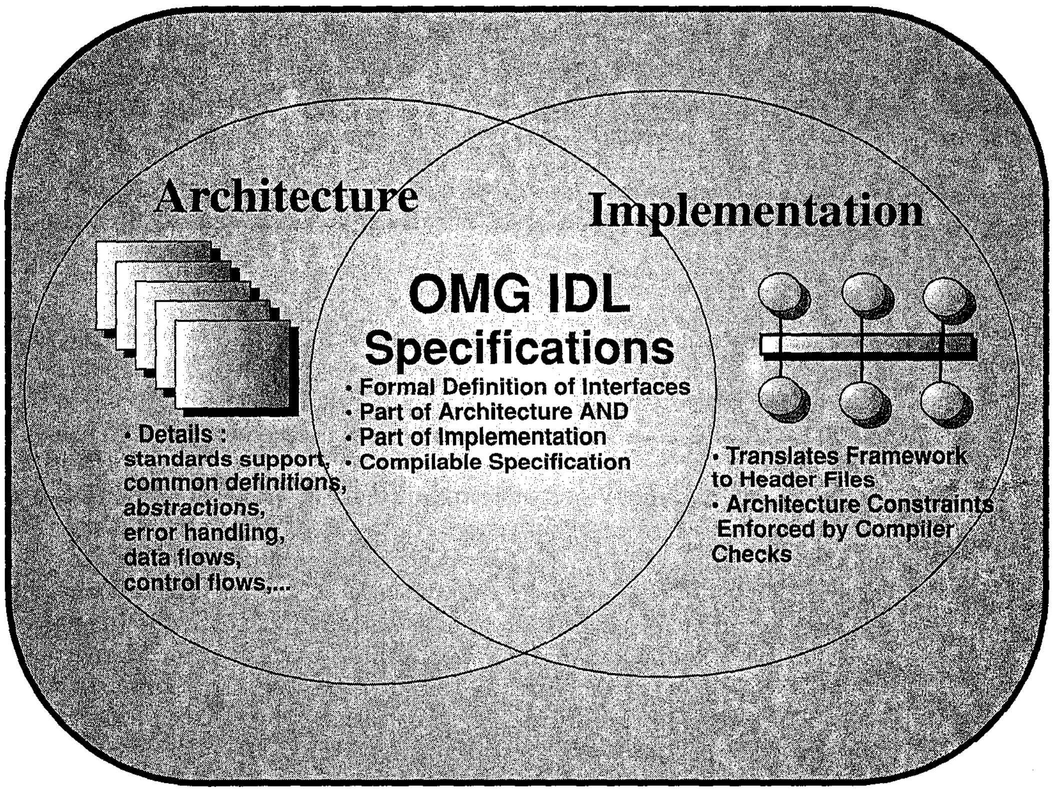

Architecture
Implementation
OMG IDL Specifications
• Formal Definition of Interfaces
• Part of Architecture AND
• Part of Implementation
• Compilable Specification
• Details : standards support, common definitions, abstractions, error handling, data flows, control flows,...
• Translates Framework to Header Files
• Architecture Constraints Enforced by Compiler Checks

Figure 4.2. OMG IDL role in software architecture and implementation.

CORBA is a great enabler of software architectures, but it does not design or define the software architecture. CORBA is a very general-purpose mechanism that supports almost any form of architecture. Many design decisions are the responsibility of the software architect and the implementors.

ARCHITECTURE PROCESS COMPARED TO METHODOLOGY

Many processes are involved in creating a software architecture. The overall process is difficult to define, because the architect's intuitive vision plays a more important role than any particular methodology. This fact is becoming more widely recognized as the methodology community shifts its focus from object-oriented design to "design patterns" [Gamma, 94].

In practice, formal design methodologies have had mixed results. More than two dozen documented methodologies based on object orientation currently exist. The OMG has published a comprehensive review and comparison of these methodologies [Hutt, 94]. In general, the quality of the people working on a project is much more critical to success than the methodology employed. The Hillside Group, a group of design patterns researchers, states that successful software architectures have been designed in spite of formal methodology. This group intends to study and document the successful software architect's expertise. In this book, we are relaying this expertise directly, bypassing the methodologists.

The process of software architecture design goes beyond people and methodology. We have met many competent people who have no concept of what a good software architecture is and why it is needed. This indicates that there is a serious educational gap. Software architecture is a top priority mainly for technology consumers such as corporate developers, systems integrators, and government organizations.

INTEGRATION CAPABILITY MATURITY MODEL

The architectural awareness of an organization can be characterized in terms of a capability model, as shown in Figure 4.3. The model provides categories for organizational use of systems integration technologies. It is useful as a self-assessment model or for gauging the awareness of architecture and development groups. This model applies primarily to corporate developers and systems integrators, but it is interesting to gauge these categories against technology supplier organizations in terms of how they support the practice of software architecture concepts.

At Level 1, the organization performs no value-added integration and produces no custom software; it is a straightforward user of commercially available software. At Level 2, the organization does produce software for its own needs, but it is indifferent to the integration technology utilized. Level 2 organizations use whatever integration technologies are readily available,

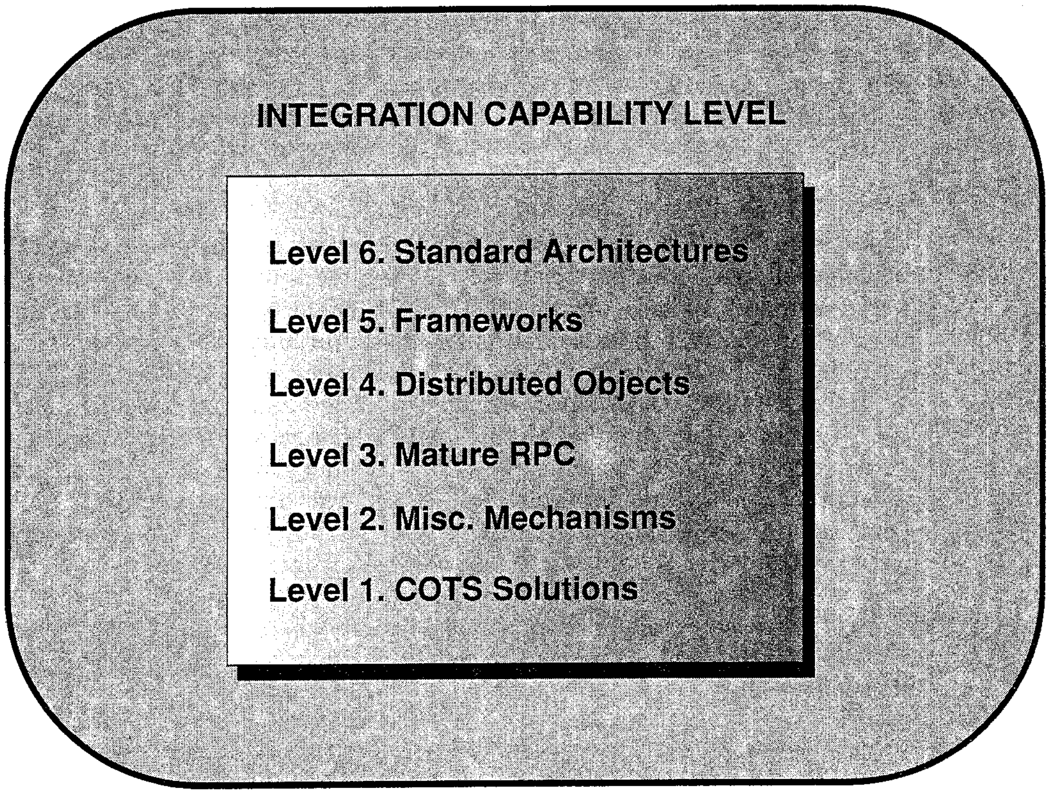

INTEGRATION CAPABILITY LEVEL
Level 6. Standard Architectures
Level 5. Frameworks
Level 4. Distributed Objects
Level 3. Mature RPC
Level 2. Misc. Mechanisms
Level 1. COTS Solutions

Figure 4.3. Integration capability maturity model.

such as Transmission Control Protocol/Internet Protocol (TCP/IP) sockets and Object Network Computing Remote Procedure Call (ONC RPC), which are available or bundled on most platforms. Level 2 organizations typically implement ad hoc software architectures because they have no real concern for creating a consistent architectural abstraction in the implemented system.

Level 3 organizations recognize the need for a uniform integration approach and have adopted a "mature" RPC technology, such as OSF DCE. OSF DCE adds security, naming, and other basic services to the RPC model, which provides advantages compared to Level 2 users. Level 3 organizations are technically conservative and are at risk of obsolescence. They are risking being caught on a nonmainstream technology base that may be expensive to support over the life cycle. Since these Level 3 organizations are not exploiting new technology advances, they are paying a high cost for software development based on the use of lower-level communication layers.

Level 4 organizations use CORBA technology actively, generally in a product-dependent manner. These organizations are primarily benefitting from the inherent advantages of the ORB technology as a higher-level RPC mechanism. Level 4 organizations insert the ORB only at distributed or heterogeneous system transitions. They do not have a concept of a software architecture other than to provide distributed processing capabilities. Their architectural interfaces utilize other legacy mechanisms when distributed processing is not required. Most of these groups do not rely on the CORBA specification, and they use the ORB products in a product-dependent manner. Level 4 groups are at risk of technology obsolescence wherever they are using vendor-dependent interfaces and do not realize any real software architecture benefits. Basically, they are using an ORB product as an improved form of RPC technology.

Level 5 organizations have developed software architecture frameworks for project-specific needs. Their frameworks include both local and remote uses of OMG IDL interfaces. Their frameworks embody sound software architecture principals. They realize software architecture adaptability benefits and cost savings over the entire system life cycle. Level 5 groups still have problems with intersystem interoperability, and they have minimal reuse across projects and systems. Level 5 groups have minimal external impact on their industry and the commercial market.

Level 6 organizations produce high-quality software architectures and services for reuse across multiple projects. They rely on the CORBA standard in preference to vendor-dependent interfaces. This gives these organizations the ability to support multiple platforms and ORBs and eliminates the risk of vendor-driven technology obsolescence. Level 6 organizations drive the limits of CORBA beyond the immediate support by the ORB vendor. For example, level 6 groups layer OMG IDL interfaces on a variety of mechanisms, including alternative protocols to address specialized application performance needs. These groups have external impact across systems and their industry. They are the generators of interoperability standards for their industries and enjoy business success by being industry leaders.

Level 6 organizations are superstar performers. Not every organization can produce technology at the world-class level. We believe that most organizations are capable of Level 5 performance. With the proper interoperability standards in place, level 5 organizations can produce level 6 results. It is the responsibility of the level 6 technology leaders to create the interoperability standards needed to make this possible.

In this book, we define a general process, provide some important examples, and give a set of techniques for software architecture design and implementation. Software architecture design is different from and not in competition with formal methodologies and CASE tools. The methodologies and tools have their roles within the process; these concepts are covered extensively by other authors. [Hutt, 94] In defining the process, we are most interested in the practical strategies that lead to good software architectures and system success.

SOFTWARE ARCHITECTURE DESIGN PROCESS

The software architecture design process comprises a number of heuristic steps. The process defined here is not a fixed methodology but an eclectic, flexible process that guides the architect on a path of learning necessary insights to create an effective architecture. We do not believe that good architecture can be created in a purely top-down manner. Good architecture involves experimentation and should be pursued in an environment that can tolerate some initial failures (or wrong turns). Overall, the pursuit of good architecture is a risk-reduction activity that builds robustness into the system design and adds adaptability that will save substantial costs over the system life cycle.

Software architecture design is more like an art than a science. Formal notations may provide the architect/artist's media, but in pursuing a formal methodology, it is important to keep the focus on the real source, the artist, not the media. Formal methodology constraints should take second priority to the creative directions of the architect. In this process, the architect uses his or her learning ability and creativity to formulate a strawman solution, then refines the working design through experience. In this description, we use metaphors from various industries to describe the creative processes (Figure 4.4).

At the start, the key learning activities include farming and mining. In farming, the architect pursues various domain analysis activities to create abstractions of the system requirements. In mining, the architect studies previous solutions and legacy systems to create generalizations of the component subsystems. These first two steps serve as exercises for edifying the architect, and the resulting artifacts might not be used directly in the architecture. In the composition process, the architect (as artist) synergizes the lessons learned to create an initial software architecture (the strawman architecture).

The balance of the process involves a series of iterative refinements to the architecture. The strawman architecture should be reviewed by the developers and refined by the architect a final time before prototyping begins. During prototyping, the architecture design should be frozen into stable versions. The fixed architecture provides a stable basis for parallel development of applications. As the architecture matures, the frequency of changes should decrease and the impact of any changes should decrease dramatically.

When the architecture is immature, prototyping should be relatively small scale. Limiting the commitment of prototype software at this phase enables more frequent and dramatic upgrades to the architecture.

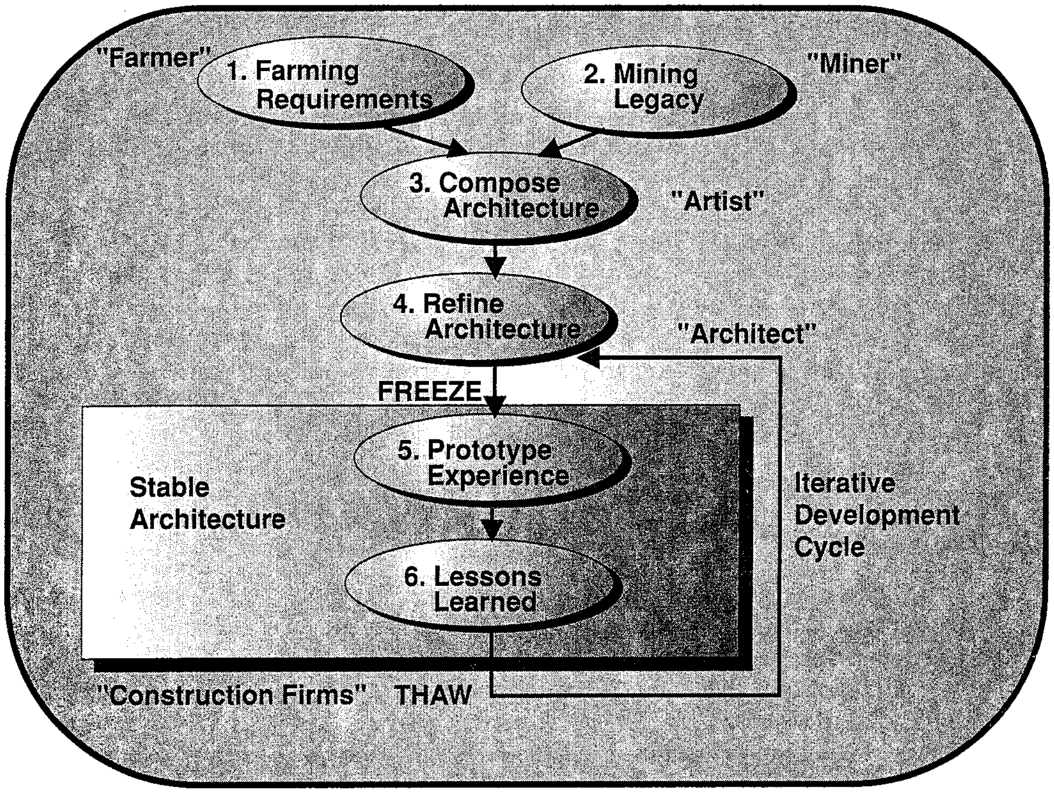


Figure 4.4. Architecture process.

Architecture refinement decisions may involve substantial project resources. Each time the architecture is modified, most or all of the component subsystems are impacted. This may be expensive if the changes are dramatic. The primary goal of the architecture process is to accommodate future change.

The architecture is the stable basis for the system. As technology evolves, components are added, interchanged, and upgraded while the architecture remains stable. The architecture life cycle is equivalent to the system life cycle and transcends the life cycles of individual component subsystems.

The architecture process is not an egalitarian group process. A work of fine art such as symphony composition or a museum painting are almost always the work of an individual artist's vision. Good software architectures also are based substantially on a chief architect's vision and control of the designs that implement the vision. The common notion that a group of developers all "charge up the hill" as equals generally does not result in effective software architectures. Good developers can always produce an effective demonstration, but ad hoc demonstrations rarely evolve into robust full life cycle architectures. Individual developers in a group will have different interpretations and concepts for a common architectural vision. It is a key architect's responsibility to communicate the system vision to developers and arbitrate between the differences of interpretation.

Software architecture design is a learning process. The design process is analogous to the incremental development concept from artificial intelligence. When the system is designed initially, the architect does not know many of the key facts and intuitions necessary to create an effective solution. The architect pursues an initial learning process through farming, mining, and composing to create a strawman architecture. The strawman architecture is evolved into the robust, finished architecture through a series of refinements, which involve new insights gained through the prototyping experience.

The Mining Process

The mining process involves the study of current technology and legacy systems. From these former designs, we hope to characterize the architecture-level support provided by these systems. In this step, we can also generalize the results so that a robust generalized component model emerges.

Figure 4.5 shows the set of models constructed during the mining process. This is a bottom-up process; the first step involves clarifying the models at the base of the diagram. The process proceeds to consolidate the models. In the last step, the architect refines the design for completeness and robustness.

The process begins by modeling the existing subsystems. These are models of specific concrete components. For example, if we are modeling database products, then we would create models for Sybase, Oracle, and other specific vendor products. In these models, we are seeking to represent an idealized API for each concrete component. These API models represent access to all of the functionality available, both that provided by the concrete API and that provided through the user interface. Often the functionality available through the concrete API is quite different from that offered through the user interfaces. The model API should capture a superset of both. OMG IDL is a suitable notation for describing the model APIs.

This first step serves to help the architect to understand a variety of alternative concrete subsystems in detail. In practice, we have found the artifacts of this step to be of minimal value. However, the learning process is invaluable, because the architect learns the commonality and differences between components. With this understanding, he or she can begin to create a robust architectural design that captures the commonality and abstracts the differences. Component interchangeability in the final architecture is a

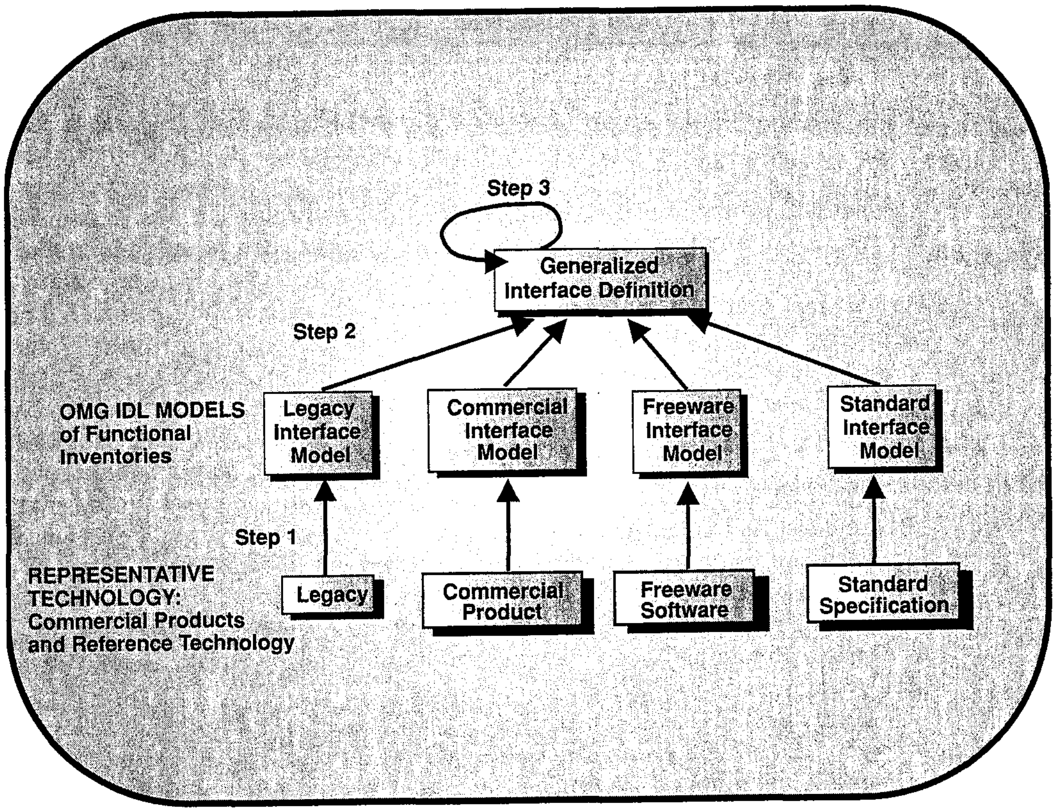

Step 3
Generalized Interface Definition
Step 2
OMG IDL MODELS of Functional Inventories
Legacy Interface Model
Commercial Interface Model
Freeware Interface Model
Standard Interface Model
Step 1
REPRESENTATIVE TECHNOLOGY: Commercial Products and Reference Technology
Legacy
Commercial Product
Freeware Software
Standard Specification

Figure 4.5. Mining process.

key goal of this exercise. To understand each component fully, the architect should consider taking developer training courses and conferring with expert developers in addition to reviewing the component's technical documentation in depth.

The second step is a process of generalization. With an understanding of the specialized components, the architect defines a common base class definition. The common base class (or generalized model) should represent a fully functional component API in its own right. It is possible to document the mapping from the concrete component models to the generalized model. In practice, we have not found much benefit in pursuing this exercise for more than one concrete model. The one documented mapping can be used as implementation guidance to the developer doing the integration.

The third step in the mining process involves the refinement of the design. Additional design concerns beyond capturing and abstracting the concrete subsytems need to be infused into the design. Whereas the first step in mining could be approached scientifically, the second and third steps increasingly involve the architect's artistic judgment, balancing cost concerns (i.e., complexity versus simplicity) with functionality concerns.

An important variable in this process is the breadth of selection of concrete components. If only one concrete component is studied, then the results will certainly be biased toward one implementation—in other words, the generalized model will be product dependent. If two concrete components are studied, component interchangeability is assured for at least those two products. In many technology markets, two products can represent a majority of market share, and this may be sufficient. If three or more products are studied, then the generalized model begins to capture the trends in the technology market. A large set of products represent current technology, but they also provide insight into the market niches and product differentiators. Some of these unique features will be adopted in the future by the market as a whole. In fact, we believe that most of the innovative features of future products can be found distributed throughout the current technology base, particularly in the products of small, technically advanced niche market vendors. As our mining study expands to cover these future component features, we can build very robust generalized models that can capture the visible market trends for the foreseeable future.

These concepts also apply to custom-software legacy systems. Two types of projects involving these systems might include migration systems and intersystem interoperability.

A migration system project involves the consolidation of several legacy systems providing similar functionality. The United States Department of Defense is currently pursuing more than 100 migration systems projects in order to downsize and save costs. The design process used to define the migration system architecture should include a mining study of the legacy systems. Such study can provide insights into the varieties of API interfaces and functionality needed to create a robust architecture. The mining study should consider external systems not directly involved in migration that can provide a more comprehensive model and provide for future contingencies. A major goal of migration systems should be to move to improved software architectures that will provide adaptability and reduce life-cycle costs.

Many organizations are pursuing interoperability projects involving legacy systems. In an interoperability project, the goal is to provide data and functionality interchange between independently developed systems. In general, this is a technically challenging and expensive proposition. A mining study of both systems can be used to define common architectural abstractions. Initially, the common abstraction can be used to provide interoperability by integrating each system to the common framework. In the long run, the common abstraction may define the architecture target for evolution of the independent systems toward a common architecture.

The Farming Process

Farming is the process of representing the system requirements at the software architecture level. Most requirements for systems focus on user interface characteristics. Software architectures deal with software-to-software boundaries. Good architectures are independent of user interfaces because user interfaces are some of the most dynamically changing parts of the system.

Formal requirements can provide an overall shopping list of the kinds of components that will be needed in the system, describe the general operation of the system, and give other useful characteristics. It is important to understand the requirements thoroughly, but it is unreasonable to expect to derive an architecture mechanically directly from requirements. Even if it was possible, we believe it would be a mistake. A requirements document is a snapshot of the user's current needs. A good software architecture should support these requirements at the system level and transcend them to provide enough adaptability to support all anticipated future needs and many unanticipated ones.

Consider the potential dramatic changes over a 10- to 15-year system life cycle. An important measure of the quality of the architecture will be how well it adapts to those dramatic unanticipated changes. If the architecture is heavily dependent on transient requirements, such as a particular commercial, off-the-shelf product API, then the architecture probably will become obsolete within three years, as soon as there is a major product upgrade. If the architecture is heavily dependent on the current business process, then it may become obsolete following a reengineering activity (perhaps within five years). The skilled software architect knows how to isolate these dependencies on transient requirements. We provide some strategies and techniques later.

Requirements-driven design does not result in effective architectures because the requirements are not written with an architectural perspective. Generally, written requirements documents do not contain enough architectural substance to impact architecture design choices substantively. While this gives the architect some freedom, requirements documents often neglect to specify the need for a software architecture. If a software architecture is not part of the system deliverable, the developer seldom makes an extra investment to create one. Lack of a deliverable software architecture is a serious requirements defect; most requirements will change substantially over the system life cycle, and the software architecture is the only deliverable facility that provides for overall system adaptability and stability.

In addition to the formal requirements, the architect also must perform some additional research on the problem domain. This can take the form of domain analysis, which allows the architect to formulate some new detailed requirements as well as to gain an intuitive understanding of the problem domain that may prove invaluable. Domain analysis allows the architect to view the system from the perspective of the end user. It can use formal methodologies and notations, including object-oriented analysis, object-oriented design, and business process reengineering.

Composing and Refinement

The composing process is an artistic process of creating the software architecture. It follows the background research processes, which may include farming and mining. Composition is a very individualized process. Some architects begin with a strawman design that they can refine. Some propose multiple design concepts and trade off the alternatives. For others it is a gestalt process where they create the architectural concept from knowledge of the problem domain, experience, and intuition. Most architects have some prior system success upon which they base their philosophy and concept for the new architecture. In the composing process, the architectural vision is first created and elaborated.

Informal design reviews play an important role in the creative process. After initial composition and some refinement, it is time to take the design to friendly audiences, including some other various experts, managers, and developers. Soliciting initial feedback is necessarily an informal process. As the design gains more exposure and feedback, the architect can refine the design to incorporate new ideas and test cases. In addition to developing the technical design, the architect is learning to sell the architectural vision. It is crucial for the architect to choose the right way of explaining the design so that people understand it and have confidence in it. Management can play a key role in supporting the architect's organizational charter to control the design and protect its integrity. Consulting with various technical experts is useful to refine specific portions of the design. For example, if the architect is proposing an innovative metadata scheme, he or she can gain confidence, knowledge, and credibility by consulting with a metadata expert to refine that portion of the design.

Ultimately the interaction with the developers is the most important of all, in that, through this relationship, the architect will gain necessary implementation feedback to determine the true effectiveness of the design. Participating in the development team is an added benefit that can provide the architect with hands-on knowledge of the prototyping issues and enhanced rapport with the development staff.

Communication of the architectural vision to the developers is a key factor to system success. As we have suggested, communication tools should include a tutoriallike architecture review. The tutorial should cover all the key design concepts, relevant new technologies, and standards. This is an important leveling step that is essential to communicating the vision. This type of design information has been very elusive in previous systems; we believe a videotape record of the architecture tutorial should be a mandatory part of every new developer's training.

Prototyping and Lessons Learned

The software architecture should be stabilized (or frozen) whenever it supports active development. After prototyping begins architectural changes should be considered very carefully, and the changes should be implemented in discrete updates, after which all the prototype software is upgraded to support the current version.

Architecture updates and prototyping should be synchronized carefully. The architecture updates should occur at natural transition points in the prototyping activity. Figure 4.6 shows an evolutionary development process that alternates architecture updates with prototyping activities. In this case, the design and development process is a series of iterative steps, which guar-

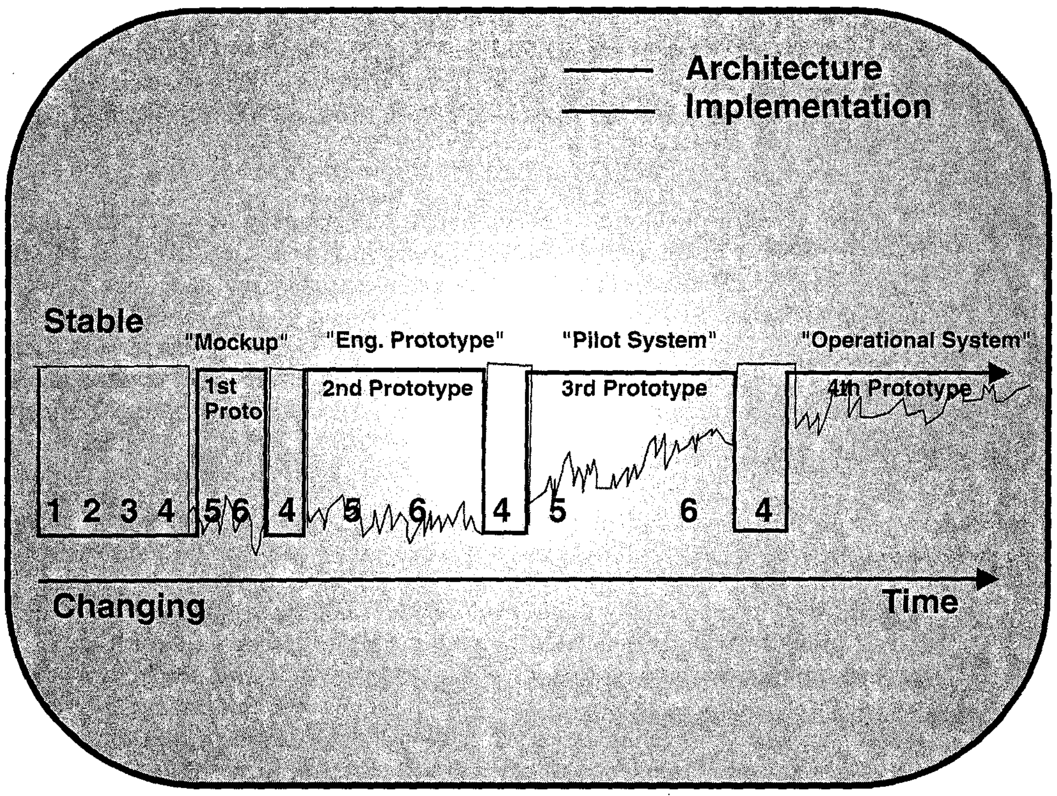


Figure 4.6. Evolutionary development.

antees architecture stability during development, and enhances architecture maturity (robustness) in each iteration.

Knowledge is gained at many levels during each architecture iteration. For example, there may be end-user deliverables that are driving the process. Through end-user and developer feedback, the architect gains insight into the architecture's strengths and weaknesses. The developer gains experience with the subsystems and component technologies as the system evolves.

Architecture updates should be driven primarily by real testbed experiences in implementing the design. Updates should be consistent with the architectural vision and avoid compromises for specific products, short-term performance needs, or component implementation dependencies. The lessons learned in the prototyping experience contain the detailed feedback that confirm or deny architecture choices and indicate the need for changes. In relating the lessons learned, key questions for the developer and architect to discuss include:

1. What parts of the architecture were used? Why and how? What parts were not used?
2. Does the developer understand the interoperability, flexibility, and extensibility features of the architecture? If so, how were these used?
3. Did the developer extend or need to work around the architecture in any way?

Wherever the developers are compelled to work around the architecture, there is likely to be an architectural flaw or a lack of communication. In fact, it is useful to put some feature in the architecture, such as a very flexible message-passing API, that will provide an escape valve to address these unforeseen needs. This is preferable to having the developers invent their own workaround strategies that may not be easily discovered or changed in an architecture update. Heavy use of these very flexible APIs may indicate a flaw in the architecture or an incorrect transfer of the architectural vision to the developers.

INTERFACE DESIGN TRADE-OFFS USING OMG IDL

Careful design trade-offs at the level of OMG IDL specifications are necessary to make the software architecture optimally effective. An idiom is an expression that is peculiar to a particular language. In this section we study some key IDL idioms. OMG IDL is distinct from other computing languages in its syntax and purpose. In order to give the reader the full rationale for these idioms, we introduce concepts from an important related area, programming language design.

Programming language design is a discipline closely associated with interface definition. It is a research area of computer science in which scientists define new high-level abstractions specifying the actions of computer systems. Whereas a software architect defines the programmer's interfaces to subsystem components, language designers define the programmer's interface to the machine. Some of the best languages are concisely specified, very easy to learn, and very flexible in their applicability. Good languages adhere to many of the architecture design principles that we are espousing in this book. Programming language design has more degrees of freedom than the practice of defining OMG IDL interfaces. Nevertheless, there are a number of important concepts from language design that are highly applicable to OMG IDL interfaces.

A key trade-off in programming language design involves the restriction of inefficient operations. A programming language defines a higher-level abstraction of the underlying machine. This abstraction must balance the convenience of specifying operations with the cost of the operations in terms of machine resources. In high-end systems, there has been an interesting shift in the underlying machine model toward vector and massively parallel processing. These changes are affecting new language definitions. For example, changes in FORTRAN from the 1960s to the 1990s show the increased importance of vector parallel processing.

An effective language design must restrict programmers from inadvertently invoking highly inefficient operations. For example, most languages restrict the passing of arrays and structures by value in function calls, an operation that might require memory allocation and copying of the structure.

These programming language restrictions actually make it more difficult for the programmer to do certain types of operations. To the software architecture designer this concept is very important. In many design choices, the architect controls the ease or difficulty of using an architectural feature. For example, if we want to guarantee certain properties in the architecture, such as interoperability or synchronization, we need to make it as difficult as possible for the programmers to work around our intentions. In other cases, we want to facilitate the ease of use. For example, we could include a powerful feature supporting extensibility that enables the developer to extend the architecture in useful ways. Generally it is a good idea to provide a range of restrictive and extensible operations in an architecture.

Data types provide some important idioms for controlling architecture restrictions using OMG IDL. The purpose of data types in OMG IDL is to allow the definition of the parameters of operations. There is a full range of available types, including scalar types and complex user defined types. Perhaps the most restrictive OMG IDL parameter type is the enumeration. The architect can define a fixed set of values that can be passed as a parameter. For example:

```
interface FruitBasket1 {
      enum  Fruit  { APPLE, ORANGE, PEAR };
      Fruit takeone(in Fruit preference);
      };
```

As a parameter, the Fruit enumeration will allow only the specification of apple, orange, or pear. Specifying a grapefruit or a watermelon would not be possible. Developers might try to work around this by specifying an illegal enumeration value. That would be a conscious subversion of an architectural restriction. The rationale for why an enumeration is used should be clearly defined in the architecture documentation and perhaps could be explained directly in the OMG IDL comments.

Other than modifying the specification, there is no legal way to extend an OMG IDL enumeration. Architects should use enumerations judiciously where they are necessary or where they are unlikely to change over the system life cycle.

An alternative approach that has much more extensibility is to substitute a string parameter for an enumeration, as in the following example:

```
interface FruitBasket2 {
      typedef string Fruit;
      Fruit takeone(in Fruit Preference);
      };
```

In this case, the choice of fruit preferences is unlimited. There also is no guidance to the developer as to what the potential choices are. (From the IDL, the developer does not know what fruit to ask for and has no way to anticipate of what fruit will be returned.) This dilemma can be solved in several ways. Without extending the interface, we could provide some constant definitions to establish a set of fruit terminology, for example:

```
//OMG IDL - Constant Definitions for Fruit Basket Interface
//Developers may define their own fruit types as new string
//literals.
const string APPLE  = "APPLE";
const string ORANGE = "ORANGE";
const string PEAR   = "PEAR";
```

We also could include comments such as the one just given explaining how developers can safely extend this interface by defining new string-valued fruit types. Architects want to resist creating new APIs whenever possible, because they add to system cost and complexity.

Note that the OMG IDL file containing the fruit types could be maintained separately from the OMG IDL file defining the APIs. In this way, we can extend the defined set of fruit without changing the base class definition file. This technique is discussed in the separation of Hierarchies section on page 86.

OMG IDL is a strongly typed language. It allows architects to define simple and complex types of many forms that are compile-time checked by the implementation language compilers. Strongly typed languages are great for defining type restrictions, and OMG IDL extends these benefits across many programming languages.

OMG IDL also supports dynamic typing through the type "any." Values of type "any" can be of an arbitrary OMG IDL type. A value of type "any" includes a typecode, which indicates the type. Typecode symbols generated by OMG IDL compilers, improve the convenience of the handling of type "any." Generally if the value of type "any" needs to be accessed, the typecode can be tested and the value recast to the appropriate predefined type.

Type "any" is very important to software architects interested in building extensible systems. For example, type "any" enables software to pass unforeseen types through a set of robust architecture APIs.

In some cases, the architect might avoid using type "any" because of its extreme extensbility. For example, type "any" should be avoided if we want to provide some set of restrictions on parameters that guarantee interoperability.

A more restricted alternative to type "any" is to use self-descriptive parameters. For example, the following is a very flexible type that can represent any file format, such as data formatted in TIFF or PostScript:

```
struct FormattedData {
      string           representation;
      sequence<octet>  value;
      };
```

The structure for FormattedData contains a tag "representation" that identifies the format. Because this is a string-valued tag, it is user extensible. The "value" is any arbitrary sequence of bytes, which makes this a very general-purpose form of "flat-file" data.

The previous example uses a sequence type. The OMG IDL sequence type is like a variable-length array; the elements have homogeneous values (including type "any") and the length can be defined at run time. Sequences are very useful for passing lists of entities. The lists can be any runtime length, and they even can be object handles.

We have found OMG IDL to be an outstanding language for architectural specification. It allows a great range of control of architectural restrictions in interfaces, and these restrictions are enforced by compiler-time checks in multiple programming languages.

SOFTWARE ACHITECTURE DESIGN PRINCIPLES

As in programming language design, software architecture is more art than science. Nevertheless, there are many important architectural design principles (or design patterns) that can be useful in creating more effective software architectures. The following sections describe some of these key design principles and strategies.

Abstraction/Simplicity

Many organizations do not even consider abstraction or simplicity to be an important consideration in software design. We consider simplicity to be perhaps the most important architectural quality; that is why we are presenting it first among our architecture principles. Simplicity is the visible characteristic of a software architecture that has successfully managed system complexity.

Functionality and simplicity are often considered to be opposing goals. Functionality can be increased by adding new features to the design—by increasing the complexity. In a committee design process, it is usually more politically acceptable to add features than to remove them. In that way, everyone can contribute ideas to a design and no one's ideas get excluded. Without much difficulty, it is easy to create specifications that no one person understands because they are too large, too complex, and inconsistent. Unfortunately, most specifications that you will encounter in practice have these characteristics. Traditional, non-OMG standards have swelled to an almost unbelievable complexity. In one generation, Structured Query Language (SQL) has increased in complexity by an order of magnitude (from less than 100 pages to more than 500 pages). Technology suppliers are driven to excess complexity because having a lot of features is considered a marketing benefit. Understanding enough of one complex specification to provide some useful capability is a full-time challenge. To build a useful system, numerous specifications must be utilized and integrated. This is one of the key challenges of systems integration, and it is the software architecture's role to manage this complexity.

An important example of excess complexity is provided by EMACS, a text editor on UNIX systems. EMACS was developed in an academic setting, where multiple developers could contribute their software extensions. The user interface evolved into a classic committee design. EMACS commands comprise hundreds of unintuitive control sequences. For example, CONTROL-X+CONTROL-S and CONTROL-META-LINEFEED provide alternative ways to save the text file. EMACS is programmable, so that users may add their own control sequences. We know people who have spent major portions of their careers creating their personalized EMACS environments. We have used EMACS for many years but now find it quite anachronistic compared to today's mature editor technologies, which are quite intuitive and so simple that they can be learned and used easily without documentation.

EMACS provides an important analogy for software architectures. Like modern text editors, good software architectures are simple, elegant, and mature designs that are easy to understand. Ease of understanding can yield substantial cost savings. About half of all software development time involves "system discovery" (trying to understand how the system works).

A primary difference between software architects and programmers is that only the architect is concerned about the cost of the design. Architects deal with system-level issues where cost is a serious concern. A mistake or inefficiency in the architecture can have consequences in every subsystem, whereas a mistake in a subsystem is usually an isolated software defect.

There is a direct relationship between simplicity and cost. In computing, many phenomena occur in factors of 2. Let's consider doubling the size of an architecture specification, for example, providing twice as many APIs than a more simplified design for integrating legacy and COTS components. In an architecture, every new API adds development cost to every subsystem's integration code. Suppose several servers support each API as well as several clients. Doubling the number of APIs could more than double the cost of initial systems integration. Each time we add a new client or server, on average, we need to support twice as many APIs to provide interoperability; thus our system extension cost can double as well. System extension due to changing requirements involves about half of all software life-cycle costs. The new APIs can double the size of development documentation and can double the cost of interface design (both substantial cost factors). On average, testing and debugging will involve twice as many APIs, which will more than double the cost of these activities. Training of software developers can take double the investment and require double the replacement cost due to staff turnover. Overall, we pay a big price for complexity.

Only about one out of five software developers has the capability to create good abstractions [Coplien, 94]. Part of the problem is that abstraction is not a concept widely conveyed in computer science education. These one in five are different from the one in 20 developers who display exceptional programming productivity. In fact, abstraction abilities and exceptional programming abilities are somewhat incompatible because they represent two fundamentally different perspectives of computer systems, one desiring a simplified system model, the other thriving in the complexity of the details. In order to create good software systems, the people who have the abstraction ability should be recognized and promoted into positions of responsibility and authority as software architects.

Interoperability versus Extensibility

Interoperability and extensibility are also important properties of a software architecture. Here we define these properties and describe how they interact in an architecture design.

Interoperability is the ability to exchange functionality and interpretable data between two software entities. Interoperability can be defined in terms

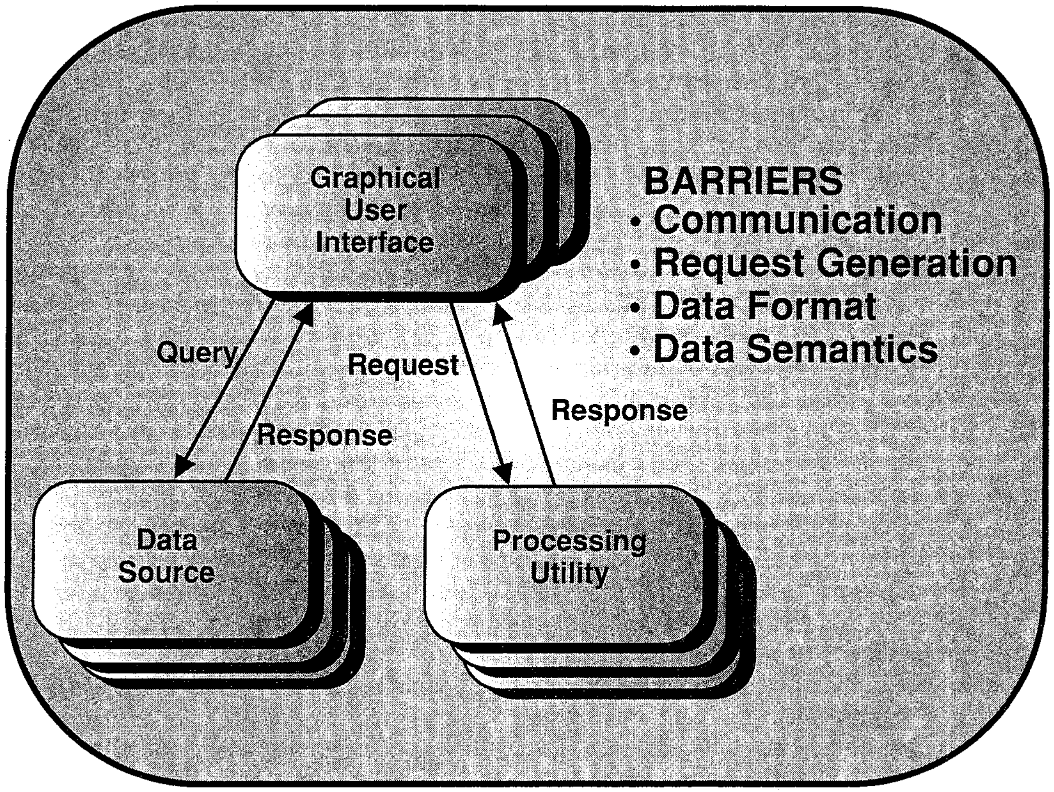


Figure 4.7. Definition of interoperability.

of four enabling requirements: communication, request generation, data format, and semantics. The software entities require a communication channel with a common communication protocol. Across this channel, the entities need to be able to formulate and transmit an interpretable request for functions or data. The result from the request must be returned to the recipient in a data interchange. Data interchange also implies a requirement for a data format that can be parsed syntactically by the recipient. The last requirement is that both entities understand the request and data through some form of semantic translation.

The interoperability problem in current software systems is that there are too many conflicting solutions for each of these requirements. There are many different communication protocols. Commercial and legacy software contain many different/conflicting protocols and levels. There are many standard and proprietary request generation languages and conflicting dialects of these languages, such as SQL, Wide Area Information Services (WAIS), scripting languages, and so on. There are numerous data formats; virtually every software package defines a new one. The semantic requirement is still primarily a research area, so the available solutions are divergent and immature.

CORBA simplifies the problem of interoperability somewhat. It provides a consistent, uniform service access mechanism that enables ubiquitous transparent communications. Through Object Service definitions such as the naming service and the query service, OMG standards provide standard ways to generate requests. Object Services such as the data interchange service will address data interchange issues. The OMG standards process can support the standardization and commercialization of new services covering semantic mediation and other areas of interoperability.

Extensibility is the characteristic of an architecture to support unforeseen uses and adapt to new developer requirements. Extensibility is a very important property for long life cycle architectures where many new requirements will be levied against the design. Built-in extensibility is necessary in order to support the needs of developers as they add new system improvements throughout the life cycle.

Interoperability and extensibility are sometimes conflicting goals in an architecture design. Interoperability requires a constrained relationship between software entities, which provides guarantees of mutual compatibility of request syntax and data formats. A flexible relationship is necessary for extensibility, which can easily extend into areas of incompatibility. In general, it is very easy to make two software entities incompatible; minor changes in data format or request syntax can easily prevent interoperability. It is no wonder that separately developed applications share so little software reuse and interoperability; lack of compatibility is the natural result of the brittleness of technology. Creating interoperability requires rigorous interchange conventions and extraordinary cooperation.

The architect can facilitate interoperability by designing some operations into the architecture that constrain the parameters to guarantee interoperability. The following is an example of a well-constrained interface for exchanging positional information. The "position" struct self-identifies its own unit value. It uses an enumeration to constrain the choice of units to two possibilities. It is a simple interface comprising send and receive operations. The exception values cover all of the reasonable cases, and provide good error traceability to the client. All of the data types are strongly typed. With this type of interface, interoperability is likely to be assured between a client and a server supporting it.

```
// OMG IDL
// Example Constrained for Interoperability
interface PositionInterchange1 {
      enum Units {INCHES, CENTIMETERS};
      struct Position {
            Units unitkind;
            double    x, y, z;
            };
```

```
        exception X_OUT_OF_RANGE { Position position_parameter; };
        exception Y_OUT_OF_RANGE { Position position_parameter; };
        exception Z_OUT_OF_RANGE { Position position_parameter; };
        void send_position(in Position current_position)
              raises(  X_OUT_OF_RANGE,
                       Y_OUT_OF_RANGE,
                       Z_OUT_OF_RANGE );
        void retrieve_position(out Position current_position);
        };
```

The architect can facilitate extensibility by designing some operations into the architecture that provide highly extensible parameters and request syntax. The extensible operations need to provide extra conventions in the IDL or documentation in order to encourage some level of interoperability. The following is an example of a more extensible version of PositionInterchange. Note that this example is an exaggeration of the changes that you might make to add extensibility. In this case, the Units type is an arbitrary string. We have established some units conventions using constants. The position structure contains a sequence of type "any" to specify the coordinates. This would allow for four-dimensional coordinates or virtually unlimited types of coordinate specifications. The operations now pass sequences of positions so that multiple entities can be monitored. Each position has a self-describing entity ID string that enables the association between entities and positions. The exception and the operations have an extra type "any" parameter, which would allow additional request or response information. The applications that use these interfaces can extend the uses of this interface widely. This interface also could be misused to convey messages unrelated to position interchange. Note also that this is a substantially more complicated interface to program than the restricted example; it requires more code to implement simple usages and a great deal of code to handle general cases of its usage.

```
// OMG IDL
// Example Unrestricted for Extensibility
interface PositionInterchange2 {
      typedef string Units;
      const string INCHUNITS = "inches";
      const string CENTIMETERUNITS = "centimeters";
      struct Position {
      string          entityid;
      Units           unitkind;
            sequence<any> coordinates;
            };
      typedef sequence<Position> PositionSeq;
      exception OUT_OF_RANGE { Position invalid_position;
                               any error_data };
exception BAD_REQUEST { any error_data };
void send_positions(    in any request,
                        in PositionSeq current_positions,
                        out any status )
                        raises(BAD_REQUEST, OUT_OF_RANGE );
void retrieve_position( in any request,
                        out PositionSeq current_positions,
                        out any status )
                        raises( BAD_REQUEST );
};
```

It is important to include a range of operation extensibility in software architectures. Some operations should guarantee interoperability, and some should provide for extensibility. It is unnecessary to go to the extremes of extensibility as shown in the previous example, but it is useful to have at least one highly extensible operation in the architecture as a last resort for developers to implement architecture workarounds. An analysis of the ways that developers utilize the extensible operation will reveal the need for improved communication of the architectural vision and provide important feedback to consider in architecture revisions.

Symmetry

Symmetry in the architecture design is a key property for achieving component interchange, code simplification, and reconfigurability. Symmetry is the practice of using a common interface for a wide range of software components. It can be realized as a common interface implemented by all subsystems or as a common base class with specializations for each subsystem. The common interface should embody the basic interoperability services provided by the architecture.

In a symmetric architecture, the clients are not hard-coded to specific services; instead they are coded to the common interface framework. Since all objects support the common interface, general interoperability is guaranteed. It is also possible to add, subtract, and interchange services without affecting the existing software.

In contrast, an asymmetric architecture is one in which every application provides a different software interface. In that case, the client code becomes dedicated to particular services. If a client uses multiple services, it must have separate code for each service. When a new application is added, new code must be written in every client using the service. Asymmetry leads to cascading costs, requiring architecture-wide code modifications whenever the system is extended or reconfigured.

Symmetric architectures have many advantages, and we believe that symmetry should be a principle behind most general-purpose architectures. If symmetry is implemented through a common base class, then the special-

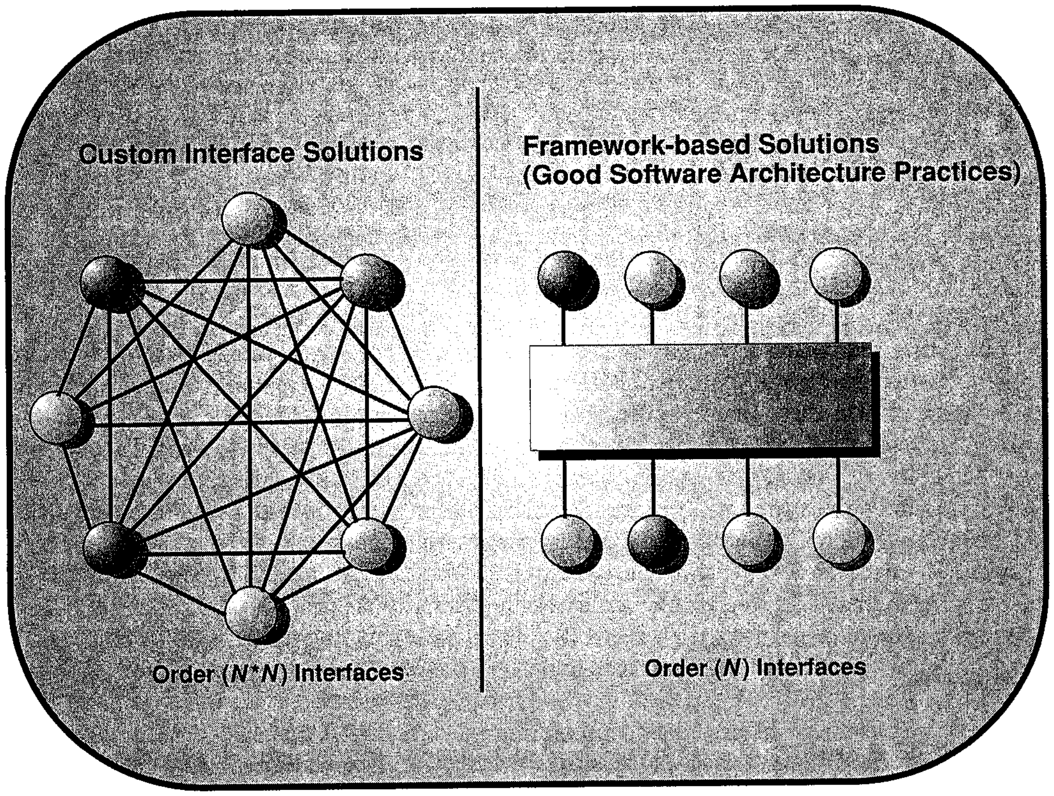

Custom Interface Solutions
Framework-based Solutions (Good Software Architecture Practices)
Order (N*N) Interfaces
Order (N) Interfaces

Figure 4.8. Custom interfaces vs. framework-based solutions.

izations play the role of an asymmetric architecture overlying the symmetric architecture. There are many cases where this is reasonable. Two possible motives are for special performance and special functionality. Suppose an architecture needs general interoperability, but some subsystems need to interchange very large images. The image applications might need a specialized interface to support efficient image transfers. Another example is where there are special functionality needs, for example, between a back-end map database and a front-end mapping user interface. In this case, the mapping user interface could reasonably be dedicated to use a specialized back-end map service. If a standard is available supporting the special functionality, it is preferable to utilize the standard rather than a custom interface. Use of standards improves architecture stability, enables component interchange, and reduces risks.

Component Isolation

Component isolation is the architectural property that limits the scope of changes as the system evolves. Component isolation means that a change in one subsystem will not require changes in other subsystems. A good architecture limits the scope of changes within modules instead of across the system.

Object encapsulation is a basic concept behind component isolation. Encapsulation means that the users of an interface (clients) are isolated from the details of the implementation behind the interface. In other words, the implementation can be completely replaced or modified without impacting clients. OMG IDL is a useful tool for specifying good encapsulations. CORBA provides excellent isolation between clients and object implementations. Implementations can be replaced, modified, recompiled, in different languages, in different activation states, in different locations, and all these changes can be made without impacting client software at the OMG IDL level.

Conventions for handling parameter data also have an impact on component isolation. Different implementations can handle parameters differently, and the code that creates and utilizes this data may be impacted by changes in the implementation.

For example, if we have two SQL databases that support a common query service, and these databases use different dialects of SQL, then the interchange of these components could impact the client software that generates the dynamic queries. Wherever possible, the architect should anticipate the impact of changes at the parameter level and provide some approach to enhance component isolation. Possible approaches include: insertion of a mediator or middleware package that provides vendor-neutral SQL, use of self-identifying data to identify the SQL dialect, use of server metadata to provide some query templates, or the provision of some OMG IDL operations with constraints that guarantee interoperability. To elaborate this last point, we could include a special query operation that partitions the dynamic query into type-checked parameters that are independent of SQL dialect.

In the earlier description of the Mining Process, we discussed how to design generalized interfaces that provide a common abstraction of multiple commercial products and legacy systems. This is an important technique for isolating a system from component product dependencies.

Metadata

Metadata in an architecture is self-descriptive information. Metadata can describe both services and information. Service metadata describes the available services and functions that the reachable applications can provide. Information metadata describes the structure and access procedures for persistent information. An example of service metadata is the Macintosh chooser, which is an on-line directory of printers, file servers, and other services. An example of information metadata is the facilities within SQL92 that standardize how relational databases store and retrieve their schema data.

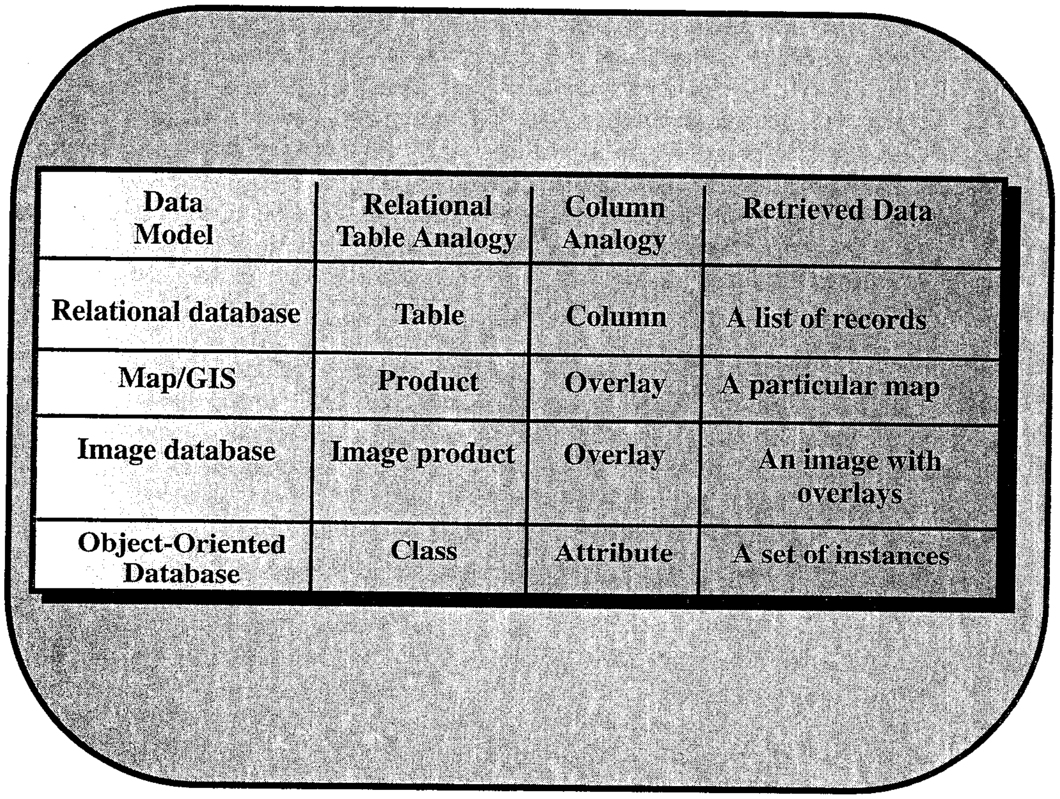
Metadata is essential for system reconfigurability. With metadata, new services can be added to a system and discovered at runtime. Metadata is the resource that allows client software to be written without hard-coding all calls to particular servers. This enables system reconfiguration, component interchange, and the possibility of multiple components providing similar services. For example, we could create an architecture that provides on-line metadata to identify the object handle of the database. In another installation of the software, we might have the clients access another database, which is at their location. In the future, we may want to add multiple databases to the system. The metadata supports the dynamic binding of the clients to the servers and enables the creation of multiple symmetric servers.

Metadata is most useful when it is provided in a consistent form across many services. Using metadata provides many benefits, but it does require additional software. The amount of software needed can be minimized if metadata is defined consistently and is accessed in the simplest possible manner to relay the essential information. Metadata's utility is maximized when it is used ubiquitously. The benefits of metadata greatly outweigh its cost and complexity.

Metadata does not have to be complicated to be useful. We have found that relatively simple forms of metadata that identify servers and provide a consistent denotation of schemas covers most of the metadata needs in an architecture. Metadata standards are very complex and expensive to implement. As an example, the Information Resource Dictionary system is a formal standard of nearly 1,000 pages of specification.

One simplifying technique for metadata involves consistent representation of schemas across many kinds of data sources. Figure 4.9 shows an example of a common scheme for representing schemas of relational databases, mapping databases, and object-oriented databases. This is a very high level of data abstraction, but it may be all the detail that is needed at the architecture level to provide interoperability. This particular approach will not meet all the needs of specialists in any of these fields, but it does provide an inexpensive common way for general-purpose clients and browsers to discover data across all types of data sources. Specialists can rely on much more complex vertical market standards for their needs, and they can do this without requiring every client to support the extra complexity.

The OMG has several current and future standards supporting metadata: the interface repository (IR), the naming service, and the trader service. The IR is an on-line source of interface descriptions. It is an integral part of



Data Model	Relational Table Analogy	Column Analogy	Retrieved Data
Relational database	Table	Column	A list of records
Map/GIS	Product	Overlay	A particular map
Image database	Image product	Overlay	An image with overlays
Object-Oriented Database	Class	Attribute	A set of instances

Figure 4.9. Consistent metadata abstraction.

the CORBA standard [OMG, 92a]. The IR supports runtime discovery and invocation of operations. The OMG IDL interface objects in the IR represent all the accessible object types. The naming service is an adopted specification, and part of the Common Object Services Specification [OMG, 94b]. The naming service provides a directory service analogous to the telephone book white pages. If the client knows the string-valued name of an object, the naming service retrieves the object handle. The trader service is currently a draft ISO standard, part of the ISO Open Distributed Process series of standards. When the ISO standard is stable, it will be submitted through the OMG Fast Track process for adoption. The trader service is a directory service analogous to the telephone book yellow pages. If the client knows the type of service, the trader can return a list of candidate services, including some key server characteristics.

The three OMG metadata services cover most architectural needs. Note that the naming service is supplied without a set of naming contexts, which should be specified by the architect. Similarly, the trader service is specified without a schema, which requires the architect to structure the service types and service characteristics.

Other types of metadata may be useful in an architecture. We have found that associating a metadata object with each server is a useful technique. The server metadata can provide more detailed information than stored in the trader. Since the server manages access to this data, it can be more sensitive in nature than information advertised in a public directory. It also can contain server-specific information, such as information about the schema, documentation, request syntax, sample requests, and request templates.

Separation of Hierarchies

Good software architecture provides a stable basis for component and systems integration. In a particular architecture problem, some aspects are more stable than others. By separating the problem into pieces, often we can enhance the stability of the whole.

Some parts of the architecture need to be very stable; other parts can be more flexible. The stability of the API designs is critical to architecture success. If the API designs are unstable, any changes can lead to systemwide software upgrades. Flexible elements can include virtually any domain-specific information. For example, the list of domain-specific properties might be relatively unknown at initial system design time and may even evolve while the system is operational. An important design strategy is to separate the uncertain or changing elements from the stable elements of the architecture. This separation can enhance the system's adaptability and provide a guideline for developers on how to extend the system.

In terms of OMG IDL, separation of hierarchies means creating separate IDL files. Perhaps the most critical OMG IDL files are sets of common base-class interfaces that provide the common set of operations for all components in the software architecture. Other OMG IDL files can contain the specialty API interfaces so that they can be included at the discretion of programmers. All of the API files need to be very stable, so they should contain the proper balance of interoperability guarantees and built-in extensibility. Other files in the flexible category include definitions of domain-specific object properties, specialty data types, domain-specific data types, and domain-specific constants. All of these specifications should be configured in a way so that individual developers can add extensions without changing the common definitions.

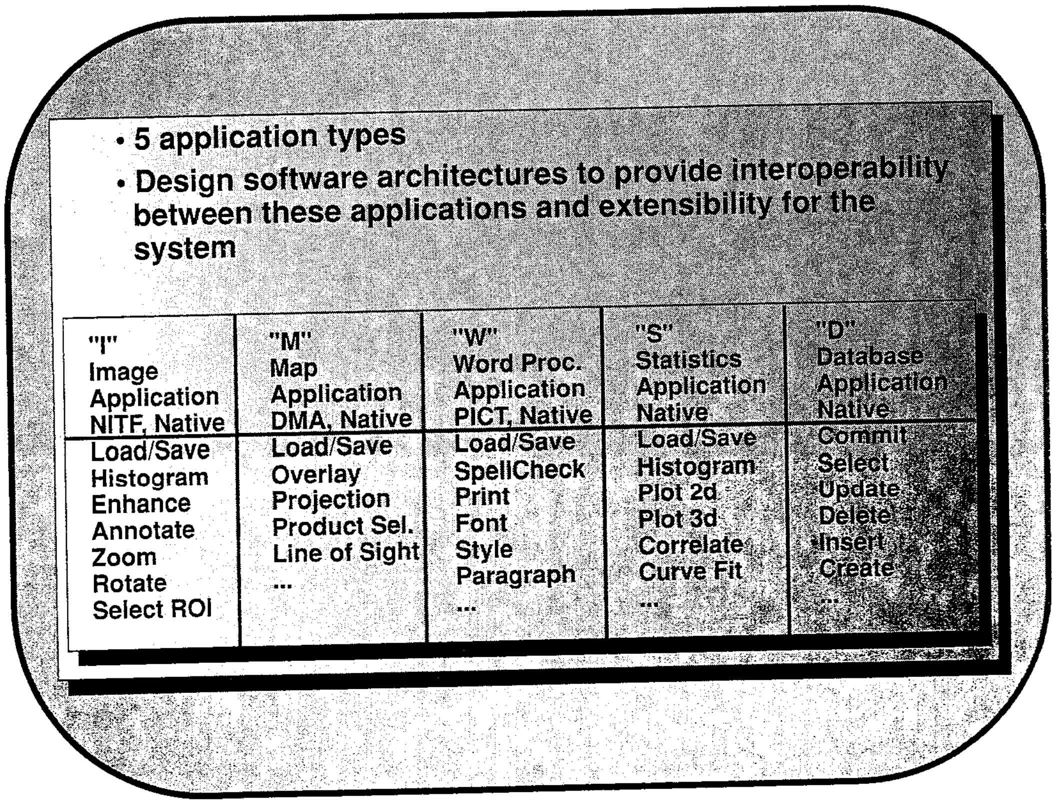
SOFTWARE ARCHITECTURE PATTERNS

The following sections describe a set of architectural patterns that present key tradeoffs in the design of effective software architectures. Since these are abstract architecture concepts, our intention is to provide enough information for you to understand the basic concept without investing excessive time. We present these patterns in particular because they highlight our architectural guidance.

The patterns are presented in three groups, the first focusing on the basic architecture concepts, the second explaining a particular family of advanced patterns, and the third looking at a wide range of architectural concepts to present these ideas within the more general context of commercial technologies.

Basic Architecture Patterns

The basic architecture patterns correspond to some general design perspectives. These patterns include custom, vertical, horizontal, and hybrid. The scope of these patterns relates to the specific architecture and the multiproject community context in which the design is formulated. This community scope is important because it represents the potential audience for reuse of



- 5 application types
- Design software architectures to provide interoperability between these applications and extensibility for the system

"I" Image Application NITF, Native	"M" Map Application DMA, Native	"W" Word Proc. Application PICT, Native	"S" Statistics Application Native	"D" Database Application Native
Load/Save	Load/Save	Load/Save	Load/Save	Commit
Histogram	Overlay	SpellCheck	Histogram	Select
Enhance	Projection	Print	Plot 2d	Update
Annotate	Product Sel.	Font	Plot 3d	Delete
Zoom	Line of Sight	Style	Correlate	Insert
Rotate	...	Paragraph	Curve Fit	Create
Select ROI		...	...	...

Figure 4.10. An example of an architecture design problem.

design and transfer of software. Our work differs from research on Design Patterns, which often has a focus on programming issues well below the architecture level. [Coplien, 94]

Here we compare the four basic patterns using a common design problem, the integration of five types of applications. A sixth application type is introduced when the system is extended.

In order to provide the community context, we consider the impact of the architecture pattern across two similar systems. The analysis includes a consideration of some common architectural changes: system extension, subsystem replacement, and system migration. In the latter case, we are considering the merger of the two similar systems into a common system. This corresponds to many current downsizing and migration projects. The analysis results define how the development costs scale with system size and provide some interesting quantitative arguments supporting our architectural guidance.

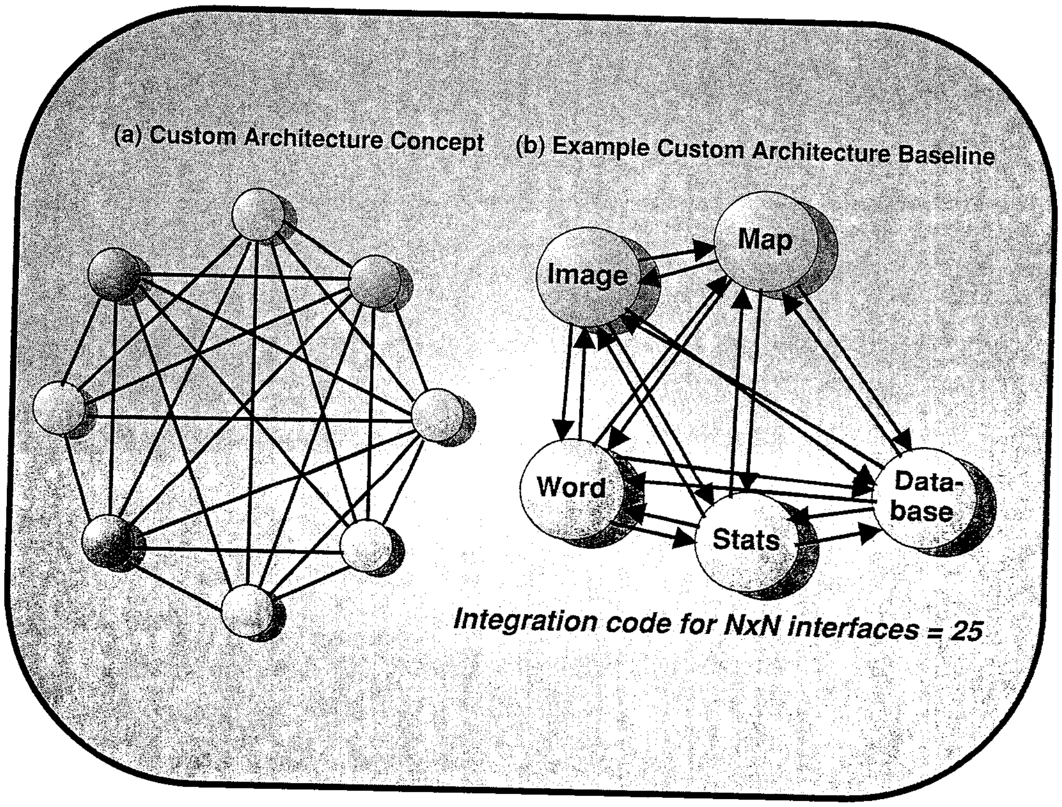
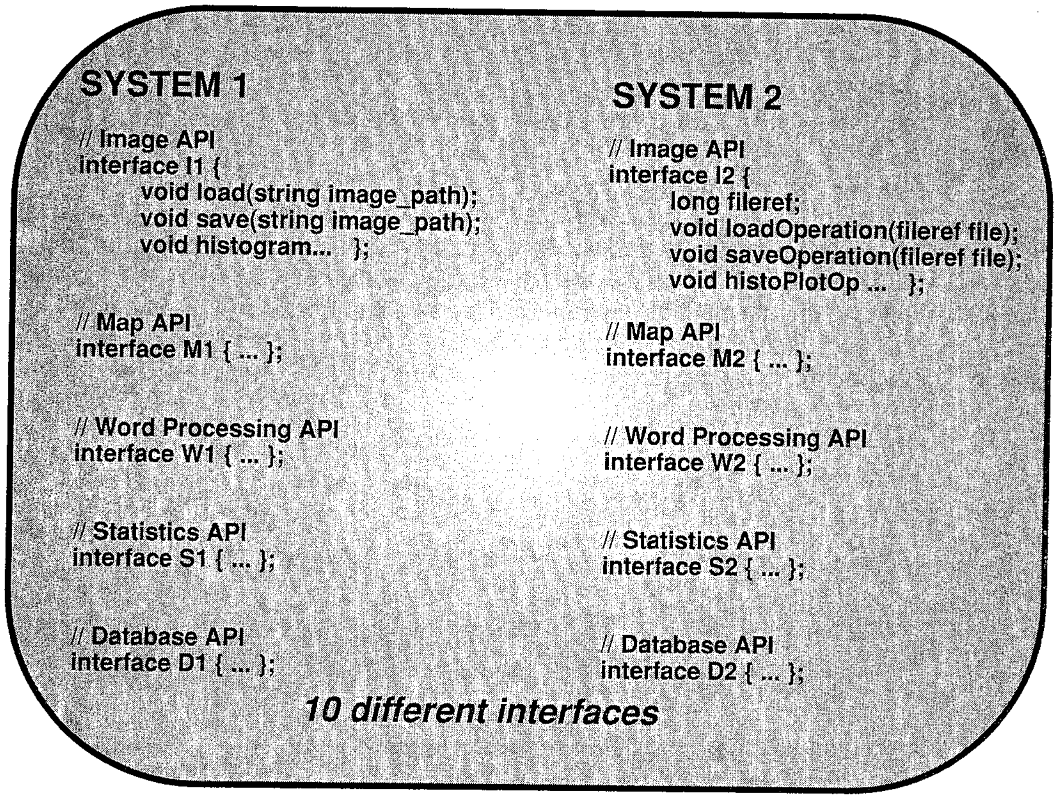

(a) Custom Architecture Concept
(b) Example Custom Architecture Baseline
Map
Image
Word
Data-base
Stats
Integration code for NxN interfaces = 25

Figure 4.11. Custom architecture pattern and baseline architecture.



SYSTEM 1

```
// Image API
interface I1 {
        void load(string image_path);
        void save(string image_path);
        void histogram...  };

// Map API
interface M1 { ... };

// Word Processing API
interface W1 { ... };

// Statistics API
interface S1 { ... };

// Database API
interface D1 { ... };
```

SYSTEM 2

```
// Image API
interface I2 {
        long fileref;
        void loadOperation(fileref file);
        void saveOperation(fileref file);
        void histoPlotOp ...  };

// Map API
interface M2 { ... };

// Word Processing API
interface W2 { ... };

// Statistics API
interface S2 { ... };

// Database API
interface D2 { ... };
```

10 different interfaces

Figure 4.12. Custom architecture PIDL.

Custom Architecture A custom architecture is designed in terms of application-specific APIs. Each application has a unique API that is not based particularly on industry standards. A summary of the Pseudo IDL (PIDL) is provided in Figure 4.12. This IDL shows that two systems with five applications both designed as custom architectures will yield a set of ten unique APIs.

The custom APIs are utilized on an as-needed basis within the implementation. Our user model (based on two comprehensive surveys) indicates that there is a strong rationale for providing most or all of these connections; we would anticipate paying the development cost for fully populating these connections as system extensions over the system life cycle. In terms of our analysis, we shall assume that this is done as part of the initial development.

Custom architecture might be considered a negative example. Custom architectures are the natural result of design without architectural focus. In the integration capability model in Figure 4.3, the custom architecture is a typical product of organizations at levels 2 to 4. These are groups that are using distributed computing without applying architecture principles such as those presented in this book. For this analysis, custom architecture represents the experimental control group that provides a baseline for comparison of effectiveness of the other basic architectural patterns.

Vertical Architecture A vertical architecture is the likely result of an architecture based on formal industry standards. Since many formal industry standards represent particular vertical market specialty areas, the vertical architecture assumes that there are standards defining APIs for each of the application areas. This assumption is true if at least the two systems we are considering are both using the same vertical specifications. For example, within the image area, a standard such as ISO Programmer's Imaging Kernel/Image Interchange Format (PIK/IIF) would define portability and interoperability between image applications for image interchange and image processing services.

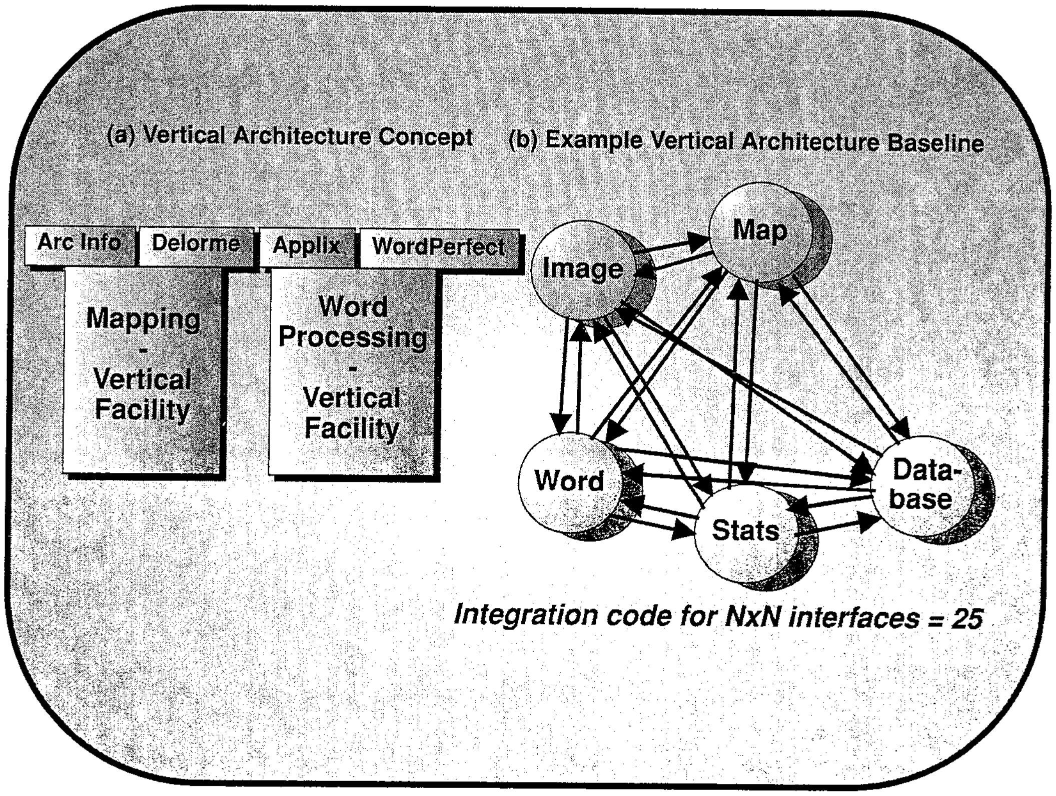

(a) Vertical Architecture Concept
(b) Example Vertical Architecture Baseline
Arc Info
Delorme
Applix
WordPerfect
Mapping - Vertical Facility
Word Processing - Vertical Facility
Image
Map
Word
Stats
Data-base
Integration code for NxN interfaces = 25

Figure 4.13. Vertical architecture pattern and baseline architecture.

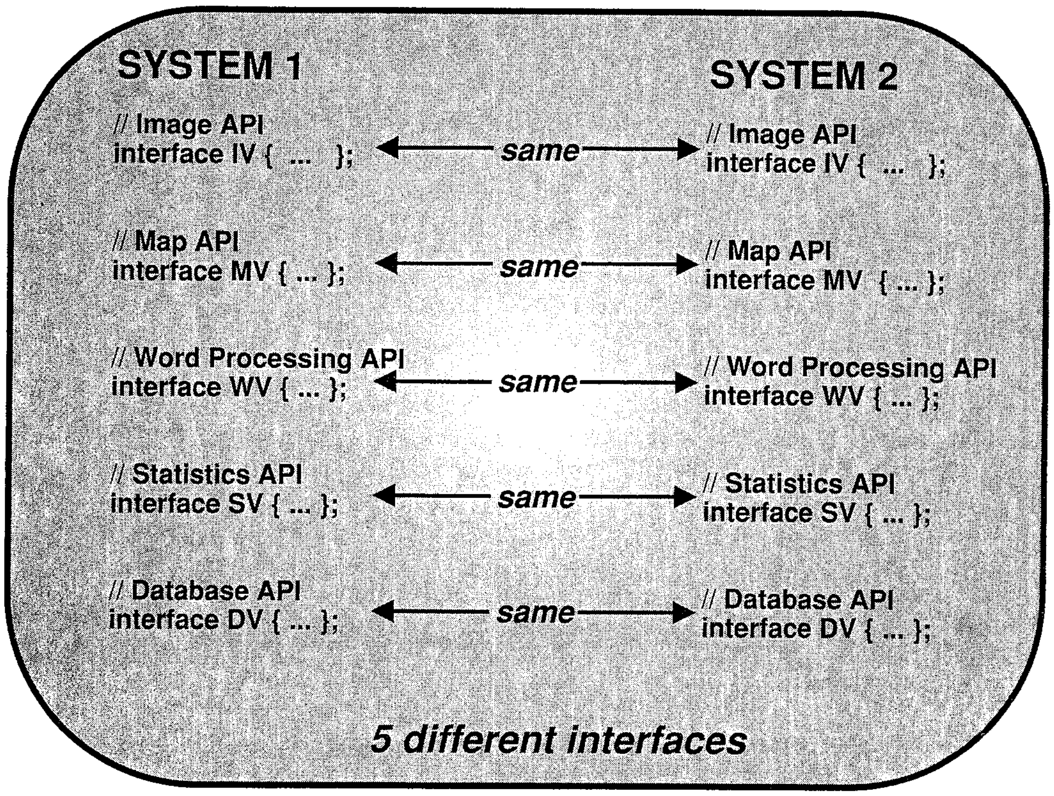


Figure 4.14. Vertical architecture PIDL.

The PIDL for the vertical architecture is shown in Figure 4.14. The vertical standards appear in the PIDL because the interfaces for each kind of application are identical across systems. Within each system, there are still five different API interfaces, one for each application. Between the two systems, there are five different software interfaces.

Horizontal Architecture The horizontal architecture is an example of a fully symmetric architecture where the interfaces are common between applications. Integration of an application to a horizontal architecture requires only one interface to be constructed. Horizontal architectures are very extensible because of the low integration cost.

The PIDL for the horizontal architecture appears in Figure 4.16 on page 92. Since all interfaces are common, only one type of interface is needed for integrating all applications in both systems. DISCUS (see Chapter 7) and Object Linking and Embedding (OLE2)'s Uniform Data Transfer are real-world examples of horizontal architectures. The limitation of these architectures is

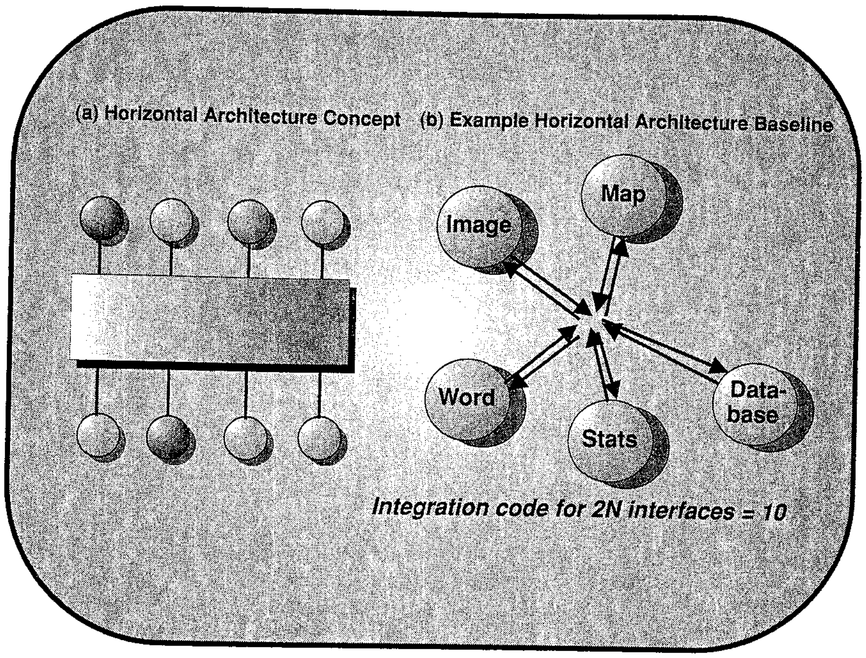
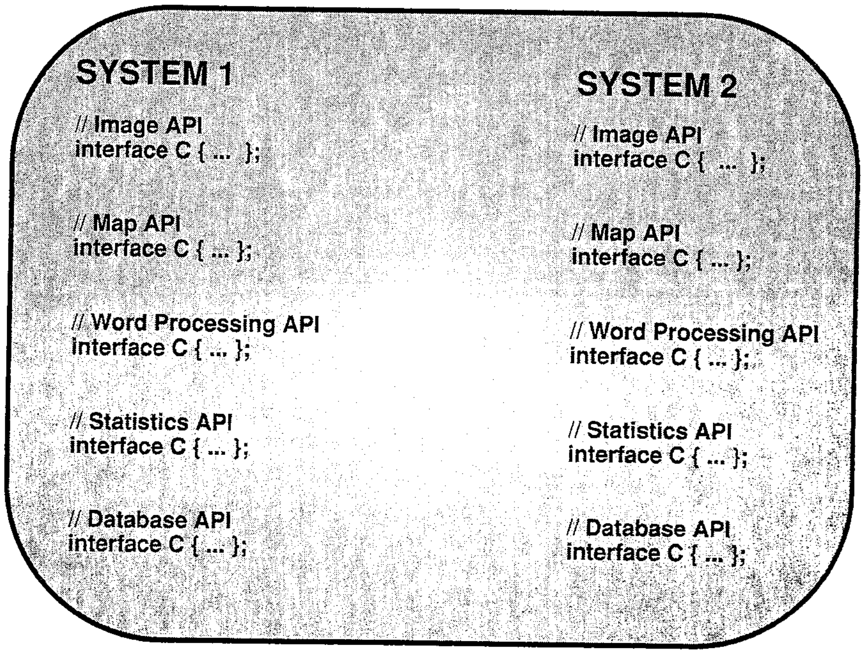

(a) Horizontal Architecture Concept
(b) Example Horizontal Architecture Baseline
Image
Map
Word
Stats
Data-base
Integration code for 2N interfaces = 10

Figure 4.15. Horizontal architecture pattern and baseline architecture.



SYSTEM 1

```
// Image API
interface C { ... };

// Map API
interface C { ... };

// Word Processing API
interface C { ... };

// Statistics API
interface C { ... };

// Database API
interface C { ... };
```

SYSTEM 2

```
// Image API
interface C { ... };

// Map API
interface C { ... };

// Word Processing API
interface C { ... };

// Statistics API
interface C { ... };

// Database API
interface C { ... };
```

Figure 4.16. Horizontal architecture PIDL.

that they do not address specialized needs. They require an unprecedented level of consensus and coordination both within and between projects. With these limitations, pure horizontal architectures probably are not practical for large communities of developers. The next pattern, the hybrid architecture, uses both horizontal and vertical standards eclectically.

Hybrid Horizontal/Vertical Architecture The hybrid architecture is a combination of both the horizontal and vertical concepts. The hybrid architecture uses a common horizontal interface as an inherited base class for each of the application interfaces. This guarantees a minimal acceptable level of interoperability between all applications. The hybrid also uses vertical standards where needed. The vertical standards are applied as specializations of the horizontal interface.

There is some latitude in the design of hybrid architectures. The horizontal and vertical standards represent a range of integration options. At a minimum, all applications should provide horizontal interoperability. Verti-

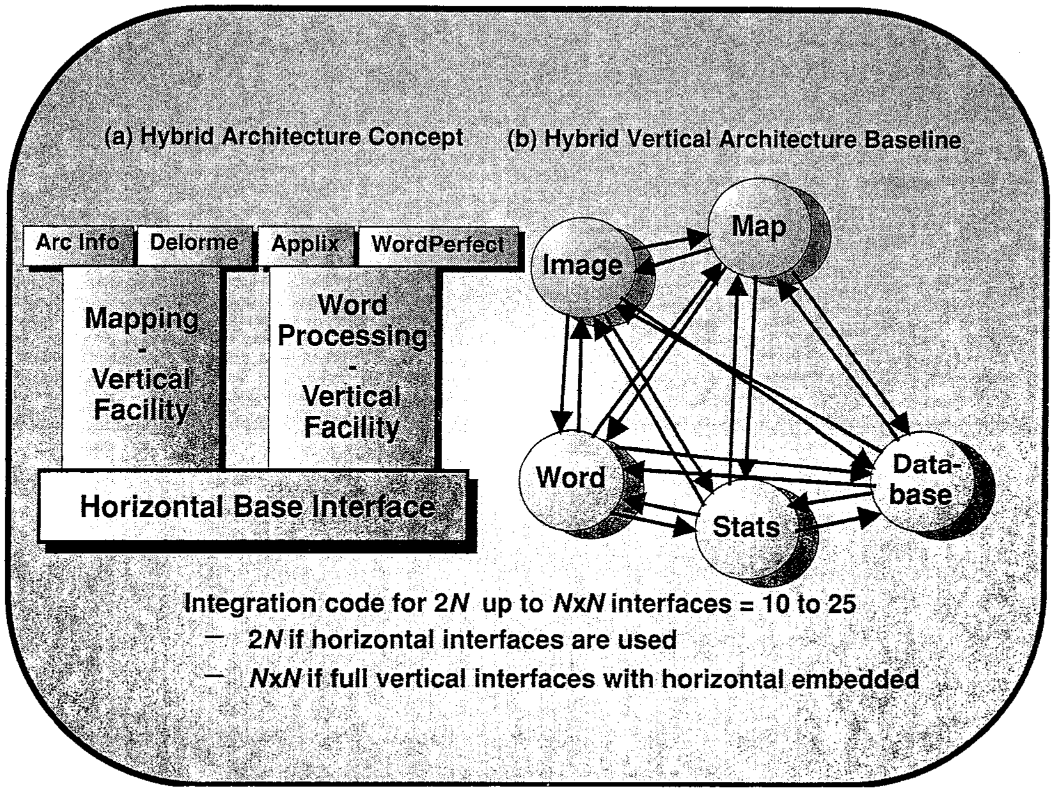

(a) Hybrid Architecture Concept
(b) Hybrid Vertical Architecture Baseline
Arc Info
Delorme
Applix
WordPerfect
Mapping - Vertical Facility
Word Processing - Vertical Facility
Horizontal Base Interface
Map
Image
Word
Stats
Data-base
Integration code for 2N up to NxN interfaces = 10 to 25
– 2N if horizontal interfaces are used
– NxN if full vertical interfaces with horizontal embedded

Figure 4.17. Hybrid architecture pattern and baseline architecture.

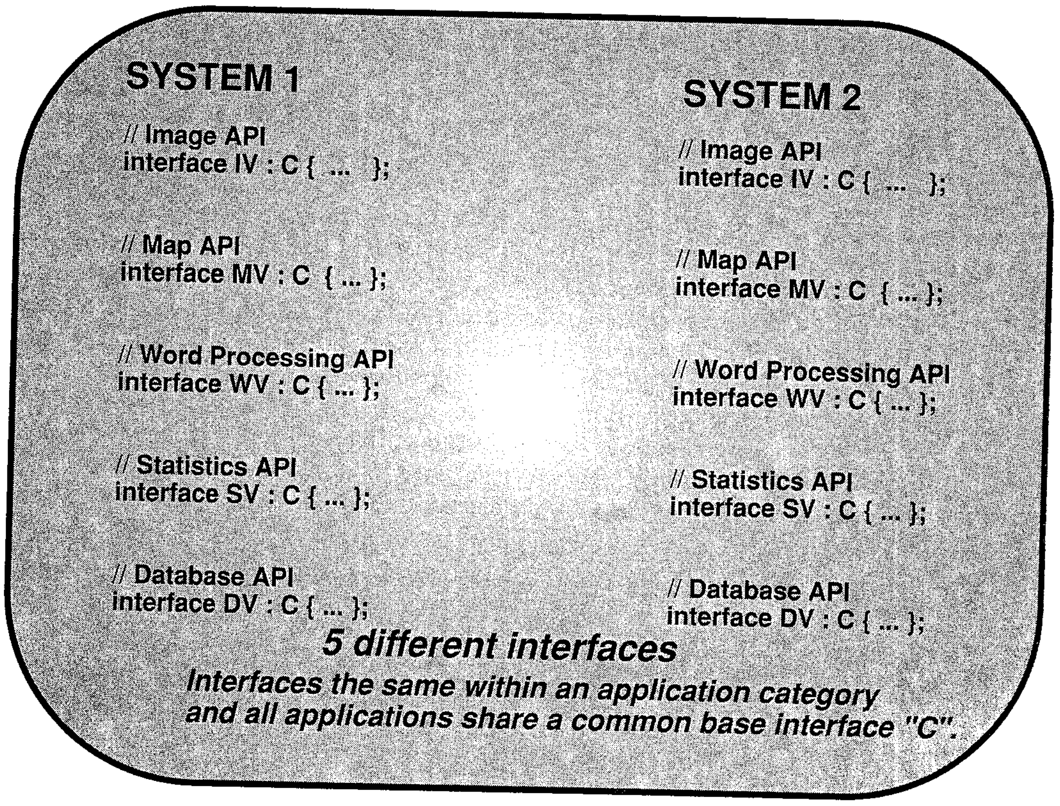

SYSTEM 1
// Image API
interface IV : C { ... };
// Map API
interface MV : C { ... };
// Word Processing API
interface WV : C { ... };
// Statistics API
interface SV : C { ... };
// Database API
interface DV : C { ... };
SYSTEM 2
// Image API
interface IV : C { ... };
// Map API
interface MV : C { ... };
// Word Processing API
interface WV : C { ... };
// Statistics API
interface SV : C { ... };
// Database API
interface DV : C { ... };
5 different interfaces
Interfaces the same within an application category
and all applications share a common base interface "C".

Figure 4.18. Hybrid architecture PIDL.

cal standards can be added to that optionally, based on the need for specialized integration. For example, the word processing application might need only a horizontal interface to the map application, but the image and map applications may need to support each other's specialized interfaces for efficiency or functionality reasons. The hybrid architecture presents a range of integration costs based on interoperability needs. The DISCUS Framework, presented in Chapter 7, is an example of a horizontal framework that can be used in the Hybrid architecture.

Analysis of the Basic Architecture Patterns The basic patterns establish an interesting model of architecture development that we can compare to different design practices and levels of community architecture coordination.

In this analysis of the basic patterns, we use the implementation code for an interface to estimate development and system extension cost. Figure 4.19 on page 95 shows the integration of an application in the custom architecture. The preexisting application is considered to be commercial or legacy code that is supplied by an external technology source. To create the system, we add a layer of integration code providing support for the appropriate interfaces. In the case of the custom architecture, this integration code includes four client interfaces and support for one service interface.

Each piece of integration code must address many issues beyond basic support for the architecture APIs. The integration must address the mapping between the application's APIs and the architecture APIs. The integration must address the format mappings and required conversions. Security and error handling are also important issues that must be addressed in the integration code. In some cases there may also be a need for semantic translations of information. By bundling these issues into the integration code cost, we can total the number of interfaces to estimate the relative software cost of architecture development and modification for different architecture patterns.

Four development cost scenarios are evaluated for each of the architecture patterns, as follows. The first scenario is initial system development. The development cost reflects all of the initial integration code required to

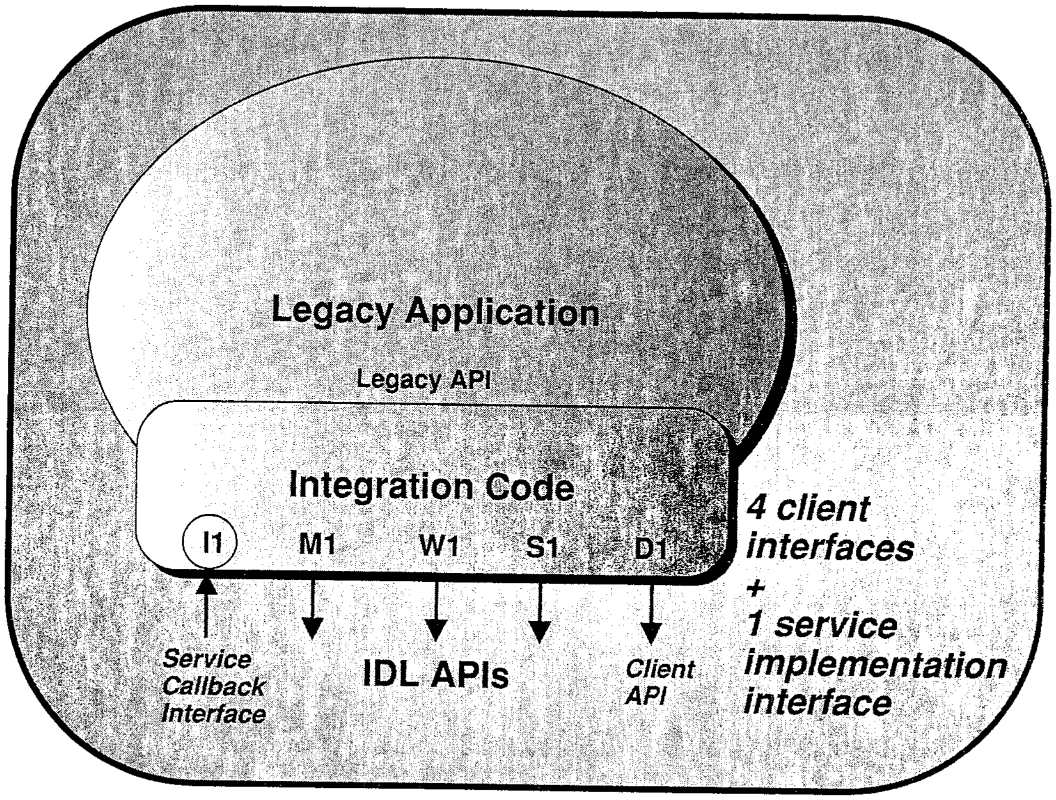

Legacy Application
Legacy API
Integration Code
I1
M1
W1
S1
D1
4 client interfaces + 1 service implementation interface
Service Callback Interface
IDL APIs
Client API

Figure 4.19. Application code vs. integration code.